Федеральное государственное образовательное бюджетное учреждение высшего образования

«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»
(Финансовый университет)

Факультет информационных технологий и анализа больших данных

Департамент анализа данных и машинного обучения

Р.И. Горохова, Е.П. Догадина, В.И. Долгов, С.В. Макрушин

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

УЧЕБНОЕ ПОСОБИЕ

Для студентов,
обучающихся по направлениям
01.03.02 «Прикладная математика и информатика»,
09.03.03 «Прикладная информатика»,
10.03.01 «Информационная безопасность»,
38.03.05 «Бизнес-информатика»
(программа подготовки бакалавра)

Москва 2023

Федеральное государственное образовательное бюджетное учреждение высшего образования

«ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»
(Финансовый университет)

Факультет информационных технологий и анализа больших данных

Департамент анализа данных и машинного обучения

Р.И. Горохова, Е.П. Догадина, В.И. Долгов, С.В. Макрушин

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

УЧЕБНОЕ ПОСОБИЕ

Для студентов,
обучающихся по направлениям
01.03.02 «Прикладная математика и информатика»,
09.03.03 «Прикладная информатика»,
10.03.01 «Информационная безопасность»,
38.03.05 «Бизнес-информатика»
(программа подготовки бакалавра)

Одобрено Советом Департамента анализа данных и машинного обучения (протокол № 6 от 28 августа 2023 г.)

Москва — 2023

УДК 004.432
ББК 32.973.2
П69

Авторы:

Горохова Р.И., канд. пед. наук, доцент, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

Догадина Е.П., канд. тех. наук, доцент, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

Долгов В.И., канд. физ.-мат. наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

Макрушин С.В., канд. эк. наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

Рецензент:

Андриянов Н.А., канд. техн. наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

П69 Практикум

Практикум по программированию на языке Python. Учебное пособие. Для студентов, обучающихся по направлениям 01.03.02 «Прикладная математика и информатика», 09.03.03 «Прикладная информатика», 10.03.01 «Информационная безопасность», 38.03.05 «Бизнес-информатика», (программа подготовки бакалавра). — М.: Финансовый университет, департамент анализа данных и машинного обучения, 2023. — 320 с.

Цель учебного пособия — предоставление необходимого методического обеспечения по выполнению практических заданий по дисциплинам «Алгоритмы и структуры данных в языке Python» и «Практикум по программированию». Учебное пособие содержит теоретические сведения, примеры решения задач и задачи по основным разделам дисциплин «Алгоритмы и структуры данных в языке Python» и «Практикум по программированию».

Учебное пособие предназначено для бакалавров направления подготовки 01.03.02 «Прикладная математика и информатика», 09.03.03 «Прикладная информатика», 10.03.01 «Информационная безопасность», 38.03.05 «Бизнес-информатика».

УДК 004.432
ББК 32.973.2

Учебное издание

Горохова Римма Ивановна, Догадина Елена Петровна,
Долгов Виталий Игоревич, Макрушин Сергей Вячеславович

«Практикум по программированию на языке Python. Учебное пособие»

Для студентов, обучающихся по направлениям 01.03.02 «Прикладная математика и информатика», 09.03.03 «Прикладная информатика», 10.03.01 «Информационная безопасность», 38.03.05 «Бизнес-информатика» (программа подготовки бакалавра)

Компьютерный набор, верстка Е. П. Догадина, Р.И. Горохова, В.И. Долгов, С.В. Макрушин

Формат 60x90/16. Гарнитура *Times New Roman.*
Усл. п.л. 20. Изд. № — 2023. Заказ № ____
Электронное издание

СОДЕРЖАНИЕ

Предисловие

Учебное пособие предназначено для студентов, обучающихся по направлениям 01.03.02 «Прикладная математика и информатика», 09.03.03 «Прикладная информатика», 10.03.01 «Информационная безопасность», 38.03.05 «Бизнес-информатика», и для использования в учебном процессе по дисциплинам «Алгоритмы и структуры данных в языке Python» и «Практикум по программированию» (семестры 1 и 2).

В учебном пособии содержатся материалы, необходимые для освоения студентами компетенций. В описание включены материалы по разработке программных продуктов на языке Python. Рассматриваются вопросы различных типов данных и инструкций в языке Python, работы с функциями и файлами, модульного, объектно-ориентированного и функционального программирования на Python. Также рассмотрены алгоритмы сортировки данных и структуры данных. По каждой теме представлены примеры решения задач различного уровня сложности и предложены задачи для самостоятельного решения, включая задания для дисциплины «Практикум по программированию».

1. Числовые типы данных и приоритеты операций в языке Python

1.1. Работа с числами

В языке Python выделяют числовые типы данных целые числа, числа с плавающей точкой (вещественные). Тип каждой переменной может динамически изменяться по ходу выполнения программы. Определить, какой тип имеет переменная, можно с помощью команды type().

Целое число в Python имеет тип int. Вещественное число в Python имеет тип float. Они записывается как последовательность цифр, перед которой также может стоять знак минус. В качестве разделителя целой и дробной части используется точка.

Для преобразования чисел из вещественных в целые и, наоборот, данные функции применяются следующим образом: int(38.125) выведет 38, float(38) выведет результат 38.0.

Основные математические операции с целыми и вещественными типами данных:

- *A* + *B* — сумма;
- *A* – *B* — разность;
- *A* * *B* — произведение;
- *A* / *B* — деление, (результатом этого действия является вещественное число, даже если *A* нацело делится на *B*);
- *A* % *B* — взятие остатка от деления *A* на *B*;
- *A* // *B* — взятие целой части от деления *A* на *B*, остаток отбрасывается независимо от типа чисел;
- *A* ** *B* — возведение в степень;
- унарный – (минус) и унарный + (плюс).

Приоритеты операций в Python совпадают с приоритетом операций в математике.

Встроенные функции: abs() — вычисление модуля (абсолютного значения), abs(–24) = 24, pow() — возведение в степень, pow(3,4) = 81, round() — округление, round(25/4) = 6, round(25/7.3) = 3.4).

Для работы с математическими функциями нужно импортировать библиотеку math: import math. После этого к функциям из этой библиотеки можно обращаться следующим образом: math.имя_функции().

Функции в библиотеке math:

- ceil(x) — возвращает округленное *x* как ближайшее целое значение типа int, большее или равное *x* (округление «вверх»).
- fabs(x) — возвращает абсолютное значение (модуль) числа *x*. В Python есть встроенная функция abs, но она возвращает модуль числа с тем же типом, что и само число, в то время как функция fabs всегда возвращает значение типа float.
- factorial(x) — возвращает факториал целого числа *x*, если *x* не целое возбуждается исключение ValueError.
- floor(x) — в противоположность ceil(x) возвращает округленное *x* как ближайшее целое значение типа int, меньшее или равное *x* (округление «вниз»).
- frexp(x) — представляет число в экспоненциальной записи и возвращает мантиссу *m* (действительное число, модуль которого лежит в интервале от 0.5 включительно до 1 не включительно) и порядок *n* (целое число) как пару чисел (*m, n*). Если *x* = 0, то возвращает (0.0, 0)
- fsum(iterable) — возвращает float сумму от числовых элементов итерируемого объекта.
- isinf(x) — проверяет, является ли float объект *x* плюс или минус бесконечностью, результат соответственно True или False.
- isnan(x) — проверяет, является ли float объект *x* объектом NaN (not a number).
- ldexp(x, i) — возвращает значение, то есть осуществляет действие, обратное функции frexp(x).
- modf(x) — возвращает дробную и целую часть float числа. Оба результата сохраняют знак исходного числа *x* и представлены типом float.
- trunc(x) — возвращает целую часть числа *x* в виде int объекта.

Степенные и логарифмические функции:

- exp(x) — возвращает *e* в степени *x*, e^x.
- log(x[, base]) — при передаче функции одного аргумента *x*, возвращает натуральный логарифм *x* (логарифм по основанию *e* = 2.7182…). При передаче двух аргументов, второй берется как основание логарифма.
- log10(x) — возвращает десятичный логарифм *x*.
- pow(x, y) — возвращает *x* в степени *y*. В отличие от операции **, приводит оба аргумента к типу float.
- sqrt(x) — квадратный корень из *x*.

Тригонометрические функции:

- acos(x) — возвращает арккосинус *x* в радианах.
- asin(x) — возвращает арксинус *x* в радианах.
- atan(x) — возвращает арктангенс *x* в радианах.
- atan2(y, x) — возвращает арктангенс угла в радианах. Результат лежит в интервале [–*π*, *π*]. Вектор, конец которого задается точкой (*x*, *y*), образует угол с положительным направлением оси *x*. Поэтому эта функция имеет более общее назначение, чем предыдущая. Например, и atan(1) и atan2(1, 1) дадут в результате *π*/4, но atan2(–1, –1) это уже –3*π*/4.
- cos(x) — возвращает косинус *x*, где *x* выражен в радианах.
- hyp(x, y) — возвращает sqrt(x**2 + y**2). Удобно для вычисления гипотенузы (hyp) и длины вектора.
- sin(x) — возвращает синус *x*, где *x* выражен в радианах.
- tan(x) — возвращает тангенс *x*, где *x* выражен в радианах.

Радианы в градусы и наоборот:

- degrees(x) — конвертирует значение угла *x* из радиан в градусы.
- radians(x) — конвертирует значение угла *x* из градусов в радианы.

1.2. Ввод-вывод в Python

Для считывания строки со стандартного ввода используется функция input(), которая считывает строку с клавиатуры и возвращает значение считанной строки, которое сразу же можно присвоить переменным:

```
a = input().
```

Можно объединить считывание строк и преобразование типов, если вызывать функцию int для того значения, которое вернет функция input():

```
a = int(input()).
```

Для вывода данных используется функция print(), может выводить не только значения переменных, но и значения любых выражений.

Пример 1.1.

```
a = 1
b = 2
print(a, '+', b, '=', a + b)
```

```
1 + 2 = 3
```

Основные принципы синтаксиса языка Python

- Конец строки является концом инструкции (точка с запятой не требуется).

```
a = 5
b = 3
print(a + b)
```

```
8
```

- Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым, главное, чтобы в пределах одного вложенного блока отступ был одинаков.

```
if a == 5:
    print('yes')
    a += 1
```

```
yes
```

- Вложенные инструкции в Python записываются в соответствии с одним и тем же шаблоном, когда основная инструкция завершается двоеточием, вслед за которым располагается вложенный блок кода, обычно с отступом под строкой основной инструкции.

1.3. Строки, функции и методы работы со строками

Строкой (объект класса str) называется последовательность символов: букв, цифр, знаков препинания и т.д.

Основные операции со строками

- *A* + *B* — конкатенация (строка *B* приписывается к строке *A*);
- *A* * *n* — повторение *n* раз, значение *n* должно быть целого типа;
- in — проверка вхождения подстроки в строку.

Существует ряд специальных символов для работы со строками. Наиболее часто используются последовательности: \n — перевод строки, \r — возврат каретки, \t — табуляция, ' — апостроф, " — кавычки, \f — перевод формата.

Строки в тройных кавычках могут содержать несколько строк текста, а также одновременно использовать кавычки и апострофы без необходимости их экранировать. Пробелы от начала строки входят в текст.

Пример 1.2.

```
s = """Happy birthday to you,
           My darling Findus!
                      Your hostess."""
print(s)
```

```
Happy birthday to you,
           My darling Findus!
                      Your hostess.
```

Строки являются неизменяемыми последовательностями (массивами) и все функциональные возможности, существующие у неизменяемых последовательностей, могут использоваться и у строк.

Срез (slice) — извлечение из данной строки одного символа или некоторого фрагмента (подстроки). Есть три формы срезов: seq[start], seq[start:end], seq[start:end:step], где start, end, step — индекс начала строки, индекс конца строки и шаг соответственно. Индексирование строк начинается с 0 и до (длины строки –1). Возможна отрицательная нумерация с конца строки.

Пример 1.3.

```
s = input() # На вводе дана строка.
print(s[2]) #Выведем третий символ этой строки.
print(s[-2]) #Выведем предпоследний символ этой строки.
print(s[0:5]) #Выведем первые пять символов этой строки.
print(s[:-2]) #Выведем всю строку, кроме последних двух символов.
print(s[::2]) #Выведем все символы с четными индексами (считая, что индексация начинается с 0,
#поэтому символы выводятся начиная с первого).
print(s[1::2]) #Выведем все символы с нечетными индексами, то есть начиная со второго символа строки.
print(s[::-1]) #Выведем все символы в обратном порядке.
print(s[::-2]) #Выведем все символы строки через один в обратном порядке, начиная с последнего.
```

```
Happy birthday to you!
p
u
Happy
Happy birthday to yo
Hpybrha oyu
ap itdyt o!
!uoy ot yadhtrib yppaH
!o tydti pa
```

Методы строк:

- strip() — удаляет пробельные или указанные символы в начале и конце строки.
- join() — преобразует последовательность (в частности список) в строку. Элементы добавляются через указанный разделитель. Формат метода: ' '.join().
- find() — ищет подстроку в строке.

- isdigit() — возвращает True, если строка содержит только цифры, в противном случае — False;
- isalpha() — возвращает True, если строка содержит только буквы, в противном случае False. Если строка пустая, то возвращается значение False.
- islower() — возвращает True, если строка содержит буквы, и они все в нижнем регистре, в противном случае — False.
- isupper() — возвращает True, если строка содержит буквы, и они все в верхнем регистре, в противном случае — False.
- split() — разделяет строку на подстроки по указанному разделителю и возвращает созданный из них список.
- index() — метод аналогичен методу find(), но если образец в строку не входит, то возникает ошибка.
- replace() — создает новую строку, в которой фрагмент исходной строки, указанный в первом аргументе, заменяется на строку, указанную во втором аргументе. Третий аргумент определяет количество замен. По умолчанию заменяются все вхождения.

Для изменения регистра символов используют upper(), lower(), title(), swapcase(), capitalize(), title().

1.4. Операторы сравнения. Логические операторы. Инструкция ветвления if...else

Операторы сравнения:

- < Меньше — условие верно, если первый операнд меньше второго.
- > Больше — условие верно, если первый операнд больше второго.
- <= Меньше или равно.
- >= Больше или равно.
- == Равенство. Условие верно, если два операнда равны.
- != Неравенство. Условие верно, если два операнда неравны.

Операторы сравнения в Python можно объединять в цепочки (в отличие от большинства других языков программирования, где для этого нужно использовать логические связки), например, a == b == c или 1 <= x <= 10.

Операторы сравнения возвращают значения специального логического типа bool.

Логические операторы: логическое И — and, логическое ИЛИ — or, логическое НЕ — not.

Например, необходимо проверить, что два данных целых числа *n* и *m* являются четными:

```
n % 2 == 0 and m % 2 == 0.
```

Проверим, что хотя бы одно из чисел *a* или *b* оканчивается на 0:

```
a % 10 == 0 or b % 10 == 0.
```

Проверим, что число *a* — положительное, а *b* — неотрицательное:

```
a > 0 and not (b < 0).
```

Операторы ветвления (***условные операторы***) предназначены для выбора к исполнению одного из возможных действий (операторов) в зависимости от некоторого условия (при этом одно из действий может быть пустым, т. е. отсутствовать). В качестве условий выбора используется значение логического выражения.

Синтаксис условного оператора:

```
if выражение_1:
    инструкции_1
elif выражение_2:
    инструкции_2
elif выражение_N:
    инструкции_N
else:
    инструкции
```

В условной инструкции могут отсутствовать части elif, else и последующий блок.

Общий вид синтаксиса тернарного условного оператора:

```
true_result if инструкция_1 else false_result.
```

Пример 1.4. Дано число *x*. Заменить его на абсолютную величину *x*.

```
x = int(input())
if x < 0:
    x = -x
print('x =', x)
```

```
-11
x = 11
```

Пример 1.5. Написать программу, определяющую четверть координатной плоскости, в которой находится точка с координатами (*x*, *y*).

```
x = int(input())
y = int(input())
if x > 0 and y > 0:
    print("Первая четверть")
elif x > 0 and y < 0:
    print("Четвертая четверть")
elif y > 0:
    print("Вторая четверть")
else:
    print("Третья четверть")
```

```
5
-2
Четвертая четверть
```

В такой конструкции условия if, …, elif проверяются по очереди, выполняется блок, соответствующий первому из истинных условий. Если все проверяемые условия ложны, то выполняется блок else, если он присутствует.

1.5. Цикл while в Python

Цикл while («пока») позволяет выполнять одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл while используется, когда невозможно заранее определить точно количество шагов.

Синтаксис цикла while:

```
while условие:
    блок инструкций
else:
    инструкции_else
```

При выполнении цикла while сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла while. Если условие истинно, то выполняется инструкция, после чего условие проверяется снова и снова выполняется инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Использование инструкции break в теле цикла приводит к немедленному прекращению цикла, при этом не выполняется ветка else. Следующей будет выполняться инструкция, следующая сразу за циклом.

Использование инструкции continue в теле цикла приводит к тому, что все оставшиеся инструкции тела цикла пропускаются, происходит переход на строку заголовка и проверка условия цикла. Далее все выполняется как обычно.

Пример 1.6. Написать программу, которая напечатает на экран квадраты всех целых чисел от 1 до 10**.**

```
i = 1
while i <= 10:
    print(i)
    i += 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Пример 1.7. Определить количество цифр натурального числа n.

```
n = int(input())
length = 0
while n > 0:
    n //= 10
    length += 1
print('length =', length)
```

```
45685
length = 5
```

Цикл while используется тогда, когда мы заранее не знаем, сколько раз должны повторять некоторое действие.

1.6. Цикл for в Python

Синтаксис цикла вида for:

```
for переменная in объект:
    инструкции
else:
    инструкции_else
```

Блок else может отсутствовать.

Для цикла for применимы функции range и enumerate.

Функция range возвращает целочисленный итератор. Способы обращения к функции range:

- range(stop) — с одним аргументом (stop) итератор представляет последовательность целых чисел от 0 до (stop – 1).
- range (start, stop) — с двумя аргументами (start, stop) итератор представляет последовательность целых чисел от start до (stop – 1).
- range (start, stop, step) — с тремя аргументами итератор представляет последовательность целых чисел от start до (stop – 1) с шагом step.

Пример 1.8. Найти сумму чисел от 1 до *n*.

```
n = 7
sum = 0
for i in range(1, n + 1):
    sum += i
print('sum =', sum)
```

```
sum = 28
```

Функция enumerate обычно используется в циклах for...in, чтобы получить последовательность кортежей (index, item), где значения индексов отсчитывается от 0 или от значения start.

Синтаксис функции enumerate:

- enumerate(i) генерируется последовательность кортежей (index, item), где значения индексов отсчитывается от 0.
- enumerate(i, start) генерируется последовательность кортежей (index, item), где значения индексов отсчитывается от значения start.

Пример 1.9. Вывести пару (элемент и его индекс) с помощью enumerate. Значения индексов отсчитывать а) с 0 и б) с 10.

```python
# а)
s = 'Happy birthday'
for index, item in enumerate(s):
    print (f"символ '{item}' имеет индекс {index}")
```

```
символ 'H' имеет индекс 0
символ 'a' имеет индекс 1
символ 'p' имеет индекс 2
символ 'p' имеет индекс 3
символ 'y' имеет индекс 4
символ ' ' имеет индекс 5
символ 'b' имеет индекс 6
символ 'i' имеет индекс 7
символ 'r' имеет индекс 8
символ 't' имеет индекс 9
символ 'h' имеет индекс 10
символ 'd' имеет индекс 11
символ 'a' имеет индекс 12
символ 'y' имеет индекс 13
```

```python
# б)
s = 'Happy birthday'
for index, item in enumerate(s, 10):
    print (f"символ '{item}' имеет индекс {index}")
```

```
символ 'H' имеет индекс 10
символ 'a' имеет индекс 11
символ 'p' имеет индекс 12
символ 'p' имеет индекс 13
символ 'y' имеет индекс 14
символ ' ' имеет индекс 15
символ 'b' имеет индекс 16
символ 'i' имеет индекс 17
символ 'r' имеет индекс 18
символ 't' имеет индекс 19
символ 'h' имеет индекс 20
символ 'd' имеет индекс 21
символ 'a' имеет индекс 22
символ 'y' имеет индекс 23
```

1.7. Задачи для самостоятельного решения

Задачи 1.1. Оператор присваивания. Математические операции.

1) Вычислите значение функции: $y = 3x + \sin(x + 2)$.
2) Вычислите значение функции: $y = ax + \cos(2x + 1)$.
3) Вычислите значение функции: $y = ax + b \cdot \sin(2x + 2)$.
4) Вычислите значение функции: $y = ax^3 + cos(3x + 1)$.
5) Вычислите значение функции: $y = \frac{x^2}{a} + \cos(2x - 1)$.
6) Вычислите значение функции: $y = \frac{x}{a} + 2x$.
7) Вычислите значение функции: $y = 3x - 2x + 1$.
8) Вычислите значение функции: $y = \frac{1}{-}x - 3x + 1$.
9) Вычислите значение функции: $y = \frac{1}{x^2+1} - a$.
10) Вычислите значение функции: $y = \frac{a}{x^2+1} - \cos(2x - 1)$.
11) Вычислите значение функции: $y = x^3 - 2x + 4$.
12) Вычислите значение функции: $y = ax + bx^3 - 8$.
13) Вычислите значение функции: $y = a\sqrt{x + 4} - b$.
14) Вычислите значение функции: $y = \cos(2x - 1) + \sin x$.
15) Вычислите значение функции: $y = a\sqrt{x} + bx$.
16) Вычислите значение функции: $y = abs(3e^x + 3 - 2\ln(x)) + 1)$.
17) Вычислите значение функции: $y = tg(x) - abs(2\sin 2x + 7.8\cos x) + 10$.
18) Вычислите значение функции: $y = (x - 2\sin x)/abs(8x - 5\, arctg(3x + 1))$.
19) Вычислите значение функции: $y = 3x^3 + cos(x + 1)$.
20) Вычислите значение функции: $y = \frac{x^3}{a} - \sin(2x + 1)$.
21) Вычислите значение функции: $y = \frac{\cos(x)}{a} + 2\sin(x)$.
22) Вычислите значение функции: $y = 2x + 3x + 5$.
23) Вычислите значение функции: $y = \frac{-}{3}x^3 + 5x - 1$.
24) Вычислите значение функции: $y = \frac{x-1}{x^2+3} - ax$.
25) Вычислите значение функции: $y = \frac{3-a}{x^2+1} + \sin(2x + 1)$.
26) Вычислите значение функции: $y = -x^3 + 2x + 4$.
27) Вычислите значение функции: $y = -ax + bx + 4$.
28) Вычислите значение функции: $y = 2\sqrt{x^3 + 1} - 2b$.

29) Вычислите значение функции: $y = -\cos(3x + 1) + \sin 2x$.

30) Вычислите значение функции: $y = 2a\sqrt{x + 2} + b(x + 1)$.

Задачи 1.2. Оператор присваивания. Ввод-вывод информации.

1) Даны длины ребер *a*, *b*, *c* прямоугольного параллелепипеда. Найти его объем $V = a \cdot b \cdot c$ и площадь поверхности $S = 2 \cdot (a \cdot b + b \cdot c + a \cdot c)$.

2) Найти длину окружности *L* и площадь круга *S* заданного радиуса *R*: $L = 2 \cdot \pi \cdot R$, $S = \pi \cdot R^2$.

3) Даны катеты прямоугольного треугольника *a* и *b*. Найти его гипотенузу c и периметр *P*: $c = \sqrt{a^2 + b^2}$, $P = a + b + c$.

4) Даны два круга с общим центром и радиусами R_1 и R_2 ($R_1 > R_2$). Найти площади этих кругов S_1 и S_2 , а также площадь S_3 кольца, внешний радиус которого равен R_1, а внутренний радиус равен R_2: $S_1 = \pi \cdot (R_1)$, $S = \pi \cdot (R)$, $S_3 = S_1 - S$.

5) Даны координаты двух противоположных вершин прямоугольника: $(x_1, y_1), (x , y)$. Стороны прямоугольника параллельны осям координат. Найти периметр и площадь данного прямоугольника.

6) Найти расстояние между двумя точками с заданными координатами $(x_1, y_1), (x , y)$ на плоскости. Расстояние вычисляется по формуле $\sqrt{(x - x_1) + (y - y_1)}$.

7) Даны координаты трех вершин треугольника: (x_1, y_1), (x_2, y_2), (x_3, y_3). Найти его периметр и площадь, используя формулу для расстояния между двумя точками на плоскости. Для нахождения площади треугольника со сторонами *a* , *b*, *c* использовать формулу Герона: $S = \sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)}$, где $p = (a + b + c)/2$ — полупериметр.

8) Дано значение температуры T в градусах Фаренгейта. Определить значение этой же температуры в градусах Цельсия. Температура по Цельсию T_C и температура по Фаренгейту T_F связаны следующим соотношением: $T_C = (T_F - 32)\ 5/9$.

9) Известно, что *X* кг шоколадных конфет стоит *A* рублей, а *Y* кг ирисок стоит B рублей. Определить, сколько стоит 1 кг шоколадных конфет, 1 кг ирисок, а также во сколько раз шоколадные конфеты дороже ирисок.

10) Найти решение системы линейных уравнений вида $\begin{cases} A_1 \cdot x + B_1 \cdot y = C_1 \\ A \cdot x + B \cdot y = C \end{cases}$, заданной своими коэффициентами A_1, B_1, C_1, A , B , C , если известно, что данная система имеет единственное решение. Воспользоваться формулами:

$$x = \frac{C_1 \cdot B_2 - C_2 \cdot B_1}{D},\ y = \frac{A_1 \cdot C_2 - A_2 \cdot C_1}{D}, \text{где } D = A_1 \cdot B - A \cdot B_1.$$

11) Дано значение угла α в градусах ($0 < \alpha < 360$). Определить значение этого же угла в радианах, учитывая, что 180 градусов равно π радианов.
12) Скорость первого автомобиля V_1 км/ч, второго — V_2 км/ч, расстояние между ними S км. Определить расстояние между ними через T часов, если автомобили удаляются друг от друга. Данное расстояние равно сумме начального расстояния и общего пути, проделанного автомобилями; общий путь = время × суммарная скорость.
13) Даны три точки *A, B, C* на числовой оси. Найти длины отрезков *AC* и *BC* и их сумму.
14) Даны два ненулевых числа. Найти сумму, разность, произведение и частное их квадратов.
15) Скорость лодки в стоячей воде V км/ч, скорость течения реки U км/ч ($U < V$). Время движения лодки по озеру T_1 ч, а по реке (против течения) — T_2 ч. Определить путь S , пройденный лодкой (путь = время × скорость). Учесть, что при движении против течения скорость лодки уменьшается на величину скорости течения.
16) Вводится вещественное число a. Не пользуясь никакими арифметическими операциями, кроме умножения, получить a^4 за две операции.
17) Вводится вещественное число a. Не пользуясь никакими арифметическими операциями, кроме умножения, получить a^8 за три операции.
18) Вводится вещественное число a. Не пользуясь никакими арифметическими операциями, кроме умножения, получить a^9 за четыре операции.
19) Вводится вещественное число a. Не пользуясь никакими арифметическими операциями, кроме умножения, получить a^{10} за четыре операции.
20) Найти сумму цифр введенного четырехзначного числа.
21) Введено трехзначное число. Вывести число в зеркальном отображении.
22) Определить сумму квадратов цифр введенного трехзначного числа.
23) Вводится четырехзначное число. Из его цифр получить два двузначных числа. Первое состоит из первой и третьей цифр исходного числа, второе — из второй и четвертой. Например, 3765 → 36 и 75.

24) Вводятся два трехзначных числа. Получить шестизначное число, состоящее из цифр исходных чисел. Например, 265 и 145 → 265145.
25) Вводится трехзначное число. Из его цифр получить два двузначных числа. Первое состоит из первой и третьей цифр исходного числа, второе — из второй и третьей. Например, 765 → 75 и 65.
26) Вводятся два числа: двузначное и трехзначное. Получить пятизначное число, состоящее из цифр исходных чисел. Например, 25 и 137 → 25137.
27) Составить программу, которая преобразует введенное с клавиатуры дробное число в денежный формат. Например, число 45.7 должно быть преобразовано к виду 45 руб. 70 коп.
28) Составить программу для перевода суммы из долларов в рубли. Вводится текущий курс доллара и сумма в долларах. Результат должен выводиться в денежном формате, например, 345 руб. 50 коп.
29) Составить программу для вычисления величины дохода по вкладу. Вводятся величина вклада, процентная ставка (в процентах годовых) и время хранения (в днях).
30) Написать программу для вычисления стоимости поездки на дачу (туда и обратно. Исходные данные: расстояние до дачи (в км), количество бензина, которое потребляет автомобиль на каждые 100 км, цена 1 л бензина.

Задачи 1.3. Строки.

1) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Найти количество слов в строке.
2) Дана строка, состоящая из русских слов, набранных заглавными буквами и разделенных пробелами (одним или несколькими). Найти количество слов, которые начинаются и заканчиваются одной и той же буквой.
3) Дана строка, состоящая из русских слов, набранных заглавными буквами и разделенных пробелами (одним или несколькими). Найти количество слов, которые содержат хотя бы одну букву «А».
4) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Найти длину самого короткого слова.
5) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Вывести строку, содержащую эти же слова, разделенные одним пробелом и расположенные в обратном порядке.

6) Дана строка, состоящая из русских слов, набранных заглавными буквами и разделенных пробелами (одним или несколькими). Вывести строку, содержащую эти же слова, разделенные одним пробелом и расположенные в алфавитном порядке.
7) Дан символ C и строки S, S_0. Перед каждым вхождением символа C в строку S вставить строку S_0.
8) Дана строка-предложение с избыточными пробелами между словами. Преобразовать ее так, чтобы между словами был ровно один пробел.
9) Дана строка. Подсчитать общее количество содержащихся в ней строчных латинских и русских букв.
10) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Найти длину самого длинного слова.
11) Дана строка-предложение. Зашифровать ее, поместив вначале все символы, расположенные на четных позициях строки, а затем, в обратном порядке, все символы, расположенные на нечетных позициях (например, строка «Программа» превратится в «ргамамроП»).
12) Дана непустая строка S и целое число N (> 0). Вывести строку, содержащую символы строки S, между которыми вставлено по N символов «*» (звездочка).
13) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Вывести строку, содержащую эти же слова, разделенные одним символом «.» (точка). В конце строки точку не ставить.
14) Дана строка, содержащая *полное имя файла*, то есть имя диска, список каталогов (путь), собственно имя и расширение (например, d:\ivanov\primer**prog**.ру). Выделить из этой строки имя файла (без расширения).
15) Дана строка, состоящая из русских слов, разделенных пробелами (одним или несколькими). Вывести самое длинное слово в предложении. Если таких слов несколько, то вывести последнее из них.
16) Дана строка-предложение с повторяющимися словами. Преобразовать ее и убрать повторение слов.
17) Дана строка, состоящая из английских слов, разделенных точкой с запятой (одной или несколькими). Найти длину самого короткого слова, которое является палиндромом (пишется одинаково слева направо и наоборот).
18) Дана строка, состоящая из английских слов, разделенных запятыми (одной или несколькими). Зашифровать ее, поместив в начале все символы, находящиеся на нечетных позициях строки, а затем, в обратном порядке, все символы, находящиеся на четных позициях.

19) Дана непустая строка *S* и целое число *N* (> 0). Вывести строку, содержащую символы строки *S*, между которыми вставлено по *N* символов "#".
20) Дана строка, содержащая полный путь к файлу в формате C:\Folder\Subfolder\filename.ext. Извлечь из этой строки только расширение файла (без имени и точки).
21) Дана строка, состоящая из английских слов, разделенных запятыми (одной или несколькими). Найти самое длинное слово в предложении. Если таких слов несколько, вывести первое из них.
22) Дана строка, состоящая из английских слов, разделенных точкой с запятой (одной или несколькими). Перевернуть порядок слов в строке.
23) Дана строка, состоящая из английских слов, разделенных пробелами (одним или несколькими). Вывести самое короткое слово в предложении. Если таких слов несколько, вывести первое из них.

Задачи 1.4. Строки, преобразования.

1) Подсчитайте количество символов (частоты символов) в заданной с клавиатуры строке.
2) Получите строку, состоящую из первых двух и последних двух символов заданной строки. Если длина строки меньше двух, то выведите соответствующее сообщение.
3) Получите строку из заданной, в которой все вхождения ее первого символа были изменены на '$', кроме самого первого символа.
4) Получите одну строку из двух заданных строк, разделенных пробелом, и поменяйте местами первые два символа каждой строки.
5) Добавьте 'ing' в конце заданной строки (длина должна быть не менее 3). Если данная строка уже заканчивается на 'ing', добавьте вместо этого 'ly'. Если длина заданной строки меньше 3, оставьте ее без изменений.
6) Удалите из непустой строки каждый *n*-й символ, начиная с первого.
7) Измените заданную строку на новую строку, в которой первый и последний символы поменяны местами.
8) Удалите символы, которые имеют нечетные значения индексов в заданной строке.
9) Дана строка-предложение. Подсчитайте вхождение каждого слова в данное предложение.
10) Переверните заданную строку в обратном порядке, если ее длина кратна четырем.
11) Переведите в заданной строке все буквы в верхний регистр, если она содержит не менее двух символов верхнего регистра в первых четырех символах.

12) Дана строка-предложение. Зашифровать ее, выполнив циклическую замену каждой буквы на следующую за ней в алфавите и сохранив при этом регистр букв («A» перейдет в «B», «a» — в «b», «B» — в «D», «z» — в «a» и т. д.). Знаки препинания и пробелы не изменять.
13) Найти первый неповторяющийся символ в заданной строке.
14) Перевести в верхний регистр первую и последнюю буквы каждого слова заданной строки.
15) Дана строка, содержащая цифры и строчные латинские буквы. Если буквы в строке упорядочены по алфавиту, то вывести 0; в противном случае вывести номер первого символа строки, нарушающего алфавитный порядок.
16) Подсчитайте количество гласных и согласных букв в заданной строке.
17) Получите строку из заданной, в которой каждое слово начинается с заглавной буквы.
18) Получите строку из заданной, в которой каждое слово написано в обратном порядке.
19) Получите строку-палиндром из заданной строки, путем добавления ее обратного порядка в конец.
20) Замените все вхождения заданного слова в строке на другое заданное слово.
21) Удалите все символы пунктуации из заданной строки.
22) Замените все гласные буквы в заданной строке на символ "*".
23) Удалите все пробелы из заданной строки.
24) Преобразуйте строку-предложение так, чтобы каждое слово начиналось с большой буквы, а остальные буквы были в нижнем регистре.
25) Удалите все дублирующиеся символы из заданной строки.
26) Преобразуйте заданную строку так, чтобы каждое слово в ней состояло из заглавных букв, а между словами был только один пробел.
27) Получите строку из заданной, в которой все буквы заменены на следующую букву в алфавите (например, 'a' → 'b', 'z' → 'a').
28) Дана строка. Замените все буквы в словах строки на порядковые номера их в алфавите.
29) Замените все символы, не являющиеся буквами или цифрами, в заданной строке на пробелы.
30) Подсчитайте количество слов в заданной строке, которые состоят только из заглавных букв.

Задачи 1.5. Условный оператор. Вычисление функции.

1) Вычислите значение функции: $y = \begin{cases} ax + 1, & x > 0, \\ ax - 1, & x \le 0. \end{cases}$

2) Вычислите значение функции: $y = \begin{cases} x - 1, & x < 1, \\ ax + 1, & x \ge 1. \end{cases}$

3) Вычислите значение функции: $y = \begin{cases} 3a, & x < 0, \\ 4ax - 1, & x \ge 0. \end{cases}$

4) Вычислите значение функции: $y = \begin{cases} 2a, & x > 4, \\ 3x - 1, & x \le 4. \end{cases}$

5) Вычислите значение функции: $y = \begin{cases} 2ax - 2, & x > 2, \\ 3a - 2x, & x \le 2. \end{cases}$

6) Вычислите значение функции: $y = \begin{cases} 2ax - 1, & x > 1, \\ x, & x \le 1. \end{cases}$

7) Вычислите значение функции: $y = \begin{cases} x, & x > 2, \\ 2a - 1, & x \le 2. \end{cases}$

8) Вычислите значение функции: $y = \begin{cases} \cos(2x - 1), & x > 2, \\ \sin(3x + 1), & x \le 2. \end{cases}$

9) Вычислите значение функции: $y = \begin{cases} 2x^3 - 2x - 1, & x > 2, \\ 3x - 2x + 1, & x \le 2. \end{cases}$

10) Вычислите значение функции: $y = \begin{cases} 2ax - 1, & x > 1, \\ 1/a, & x \le 1. \end{cases}$

11) Вычислите значение функции: $y = \begin{cases} a\sqrt{x} + 1, & x \ge 0, \\ ax - 1, & x < 0. \end{cases}$

12) Вычислите значение функции: $y = \begin{cases} \sqrt{x} + a, & x \ge 0, \\ a/x - 1, & x < 0. \end{cases}$

13) Вычислите значение функции: $y = \begin{cases} 1/x + a, & x > 0, \\ x - 1, & x \le 0. \end{cases}$

14) Вычислите значение функции: $y = \begin{cases} \cos(x), & x > \pi/2, \\ \sin(x), & x \le \pi/2. \end{cases}$

15) Вычислите значение функции: $y = \begin{cases} \sqrt{x - 2}, & x > 2, \\ (x - 2) + 1, & x \le 2. \end{cases}$

16) Вычислите значение функции: $y = \begin{cases} ax - 3, & x > 2, \\ ax - 3, & x \le 2. \end{cases}$

17) Вычислите значение функции: $y = \begin{cases} x - 2x + 1, & x < 3, \\ ax + 3, & x \ge 3. \end{cases}$

18) Вычислите значение функции: $y = \begin{cases} 3xa + 1, & x < 0, \\ 2x - a, & x \ge 0. \end{cases}$

19) Вычислите значение функции: $y = \begin{cases} 2a + x, & x > 2, \\ 4x - 3, & x \le 2. \end{cases}$

20) Вычислите значение функции: $y = \begin{cases} 3ax - 1, & x > 0, \\ 2a - 3x, & x \le 0. \end{cases}$

21) Вычислите значение функции: $y = \begin{cases} 2ax - 3x + 1, & x > 1, \\ 3x - 4, & x \le 1. \end{cases}$

22) Вычислите значение функции: $y = \begin{cases} 2x + 1, & x > 2, \\ 2ax - 1, & x \le 2. \end{cases}$

23) Вычислите значение функции: $y = \begin{cases} \cos(2x + 3), & x > 3, \\ \sin(2x - 1), & x \le 3. \end{cases}$

24) Вычислите значение функции: $y = \begin{cases} 2x^3 - 2x - 2, & x > 2, \\ 3x + 2x - 1, & x \le 2. \end{cases}$

25) Вычислите значение функции: $y = \begin{cases} 2ax + 1, & x > 0, \\ 2/ax, & x \le 0. \end{cases}$

26) Вычислите значение функции: $y = \begin{cases} 2a\sqrt{x} + 3, & x \ge 0, \\ 3ax - 1, & x < 0. \end{cases}$

27) Вычислите значение функции: $y = \begin{cases} 2\sqrt{x} + 3a, & x \ge 0, \\ \frac{5a}{x} + 2, & x < 0. \end{cases}$

28) Вычислите значение функции: $y = \begin{cases} 3/x + ax, & x > 0, \\ 2x + 1, & x \le 0. \end{cases}$

29) Вычислите значение функции: $y = \begin{cases} \cos(x) + 2, & x > \pi/2, \\ \sin(x) - 1, & x \le \pi/2. \end{cases}$

30) Вычислите значение функции: $y = \begin{cases} 3\sqrt{x - 2} - 1, & x > 2, \\ 2(x - 2) + 3, & x \le 2. \end{cases}$

Задачи 1.6. Условный оператор. Сложные условия.

1) Вводятся координаты точки (*x*, *y*). Определить попадает ли точка в область заданную условиями: $y \le 5 \sin x,\ y \ge 0,\ 0 \le x \le \pi$.
2) Вводятся координаты точки *(x, y)*. Определить попадает ли точка в область заданную условиями: $y \le 3 \sin x,\ y \ge 0,\ 0 \le x \le \pi$.
3) Вводятся координаты точки *(x, y)*. Определить попадает ли точка в область заданную условиями: $x^2 + y^2 \le 4,\ x^2 + y^2 \ge 1$.
4) Введено трехзначное число. Найти сумму четных цифр.
5) Введено четырехзначное число. Содержится ли в записи этого числа цифра 7?
6) Введено трехзначное число. Если в записи числа встречается цифра 5, то записать число в зеркальном отображении.
7) Введено трехзначное число. Если сумма его цифр нечетна, то увеличить число вдвое.
8) Введено четырехзначное число. Найти сумму цифр, кратных трем.
9) Вводятся *X* и *Y*. Если хотя бы одно из этих чисел положительно, то найти их произведение. Иначе — найти их сумму.

10) Вводятся *X* и *Y*. Если *X* больше *Y*, то произвести их обмен.
11) Из чисел *A, B, C, D* выбрать максимальное.
12) Вводятся *X* и *Y*. Заменить большее из этих чисел разностью большего и меньшего.
13) Сколько среди заданных чисел *A, B, C, D* нечетных.
14) Вводятся *A, B, C, D*. Найти среднее арифметическое максимального и минимального.
15) Вводятся числа *A, B, C*. Упорядочить их по убыванию.
16) Из чисел *A, B, C, D* выбрать минимальное.
17) Определить, какие из чисел *A, B, C, D* принадлежат интервалу (–10, 15).
18) Определить, есть ли среди цифр заданного трехзначного числа одинаковые.
19) Составить программу, определяющую является ли билет с шестизначным номером счастливым (счастливым является билет, у которого сумма первых трех десятичных цифр равна сумме трех последних).
20) Вводится трехзначное число. Определить, является ли оно числом Армстронга. Число Армстронга — натуральное число, которое равно сумме своих цифр, возведённых в степень, равную количеству его цифр. Например, число 153 — число Армстронга, потому что $1^3 + 5^3 + 3^3 = 153$.
21) Вводятся положительные *a, b, c d*. Определить, можно ли прямоугольник со сторонами *a, b* поместить внутри прямоугольника со сторонами c, d так, чтобы каждая из сторон первого прямоугольника была параллельна или перпендикулярна каждой стороне второго треугольника.
22) Составить программу, проверяющую знание таблицы умножения. В программе выбираются случайным образом целые числа X ($1 \le X \le 9$) и Y ($1 \le Y \le 9$) и предлагается пользователю ввести ответ. Результат выполнения программы: «Правильно» или «Неправильно».
23) Вводятся числа *A, B, C*. Упорядочить их по возрастанию.
24) Составить программу, которая по введенной начальной букве выводит название цветов радуги (красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый).
25) Составить программу, которая по введенному порядковому номеру выводит название дня недели.
26) Составить программу, которая позволяет ввести номер месяца и вывести его название.
27) По числу текущего месяца определить день недели.

28) Напишите программу, которая по номеру дня недели — целому числу от 1 до 7 — выдает в качестве результата количество занятий в вашей группе в соответствующий день.
29) В зависимости от стажа работы педагогам введена надбавка в размере: для работающих от 5 до 10 лет — 10%; для работающих от 10 до 15 лет — 15%; для работающих свыше 15 лет — 20%. Составьте программу, которая по заданному стажу работы и размеру оклада определит размер заработной платы.
30) Составить программу, которая после введенного с клавиатуры числа (в диапазоне от 1 до 999), обозначающего денежную величину, дописывает слово «рубль» в правильной форме. Например, 5 рублей, 21 рубль, 173 рубля.

Задачи 1.7. Операторы цикла.

1) Даны целые числа K и N ($N > 0$). Вывести N раз число K.
2) Даны два целых числа A и B ($A < B$). Вывести в порядке возрастания все целые числа, расположенные между A и B (включая сами числа A и B), а также количество N этих чисел.
3) Даны два целых числа A и B ($A < B$). Вывести в порядке убывания все целые числа, расположенные между A и B (включая сами числа A и B), а также количество N этих чисел.
4) Одна штука некоторого товара стоит 20,4 руб. Напечатать таблицу стоимости для 1, 2, …, 10 штук этого товара.
5) Напечатать квадраты всех целых чисел от A до B ($A < B$) с шагом H.
6) Напечатать все положительные числа из диапазона от A до B ($A < B$) с шагом H.
7) Даны два целых числа A и B ($A < B$). Найти сумму всех целых чисел от A до B включительно.
8) Даны два целых числа A и B ($A < B$). Найти произведение всех целых чисел от A до B включительно.
9) Даны два целых числа A и B ($A < B$). Найти сумму квадратов всех целых чисел от A до B включительно.
10) Дано положительное вещественное число — цена 1 кг конфет. Вывести стоимость 1.2, 1.4, …, 2 кг конфет.
11) Дано целое число N (> 0). Найти квадрат данного числа, используя для его вычисления следующую формулу: $N^2 = 1 + 3 + 5 + \cdots + (2 \cdot N - 1)$.

12) Дано вещественное число A и целое число N (> 0). Вывести все целые степени числа A от 1 до N.

13) Дано целое число N (> 0). Найти наибольшее целое число K, квадрат которого не превосходит N: $K^2 \le N$.

14) Дано целое число N (>1). Найти наибольшее целое число K , при котором выполняется неравенство $3^K < N$.

15) Дано целое число N (>1) и две вещественные точки на числовой оси: A, B ($A < B$). Отрезок $[A,B]$ разбит на N равных отрезков. Вывести H — длину каждого отрезка, а также набор точек A, $A+H$, $A+2H$, $A+3H$, …, B, образующий разбиение отрезка $[A,B]$.

16) Подсчитать сумму двузначных чисел, сумма цифр которых не превышает 10.

17) Протабулировать функцию $y = x^3 - 1$ на интервале [–1,3] с шагом 0.2.

18) Протабулировать функцию $y = \sin x - \cos x$ на интервале $[-\pi, \pi]$ с шагом $\pi/10$.

19) Протабулировать функцию $y = x \cdot \cos x$ на интервале $[-\pi, \pi]$ с шагом $\pi/10$.

20) Вы положили S рублей в банк под p% ежегодного прироста. Определить: а) когда сумма вклада утроится? б) какой будет сумма вклада после 10 лет?

21) Готовясь к соревнованиям, лыжник в первый день побежал 10 км, затем каждый день увеличивал расстояние на 10%. Сколько километров пробежал он за неделю тренировок? На какой день он пробежал больше 15 км?

22) Малое предприятие в первый день работы выпустило P единиц товарной продукции. Каждый последующий день оно выпускало продукции на Q единиц больше, чем в предыдущий. Сколько дней потребуется предприятию, чтобы общее количество выпущенной продукции за все время работы впервые превысило запланированный объем T?

23) Составить программу, которая находит и выводит на печать все четырехзначные числа $abcd$, для которых выполняются следующие условия: a, b, c, d — разные цифры и $ab - cd = a + b + c + d$. Здесь запись ab означает, что число составлено из цифр a и b.

24) Числа Фибоначчи определяются как $a(0) = 1$, $a(1) = 1$, $a(i) = a(i-1) + a(i-2)$. Найти: а) N-е число Фибоначчи; б) сумму первых N чисел Фибоначчи.

25) Определить, является ли число n простым.

26) Определить, является ли число n совершенным. Совершенное число — натуральное число, равное сумме всех своих собственных делителей (т. е. всех положительных делителей, отличных от самого числа). Например, $6 = 1 + 2 + 3$.
27) Найти наименьшее общее кратное (НОК) двух натуральных чисел.
28) Вводится *K* пар натуральных чисел. Найти НОД каждой пары.
29) Вводится *N* натуральных чисел. Найти среднее арифметическое цифр каждого из них.
30) Вводится *N* натуральных чисел. Найти количество четных цифр в каждом из них.

Задачи 1.8. Операторы цикла FOR

1) Для заданного натурального *n* подсчитать сумму:
$$\frac{1}{} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$
2) Для заданного натурального *n* и действительного *x* подсчитать сумму:
$$cos\, x + \frac{cos^2 x}{} + n\frac{cos^3 x}{3} + \cdots + \frac{cos^2 x}{n}.$$
3) Для заданного натурального *n* подсчитать сумму:
$$1 - 3 + 3 - 3^3 + \cdots + (-1)^n 3^n.$$
4) Для заданного натурального *n* подсчитать сумму:
$$\frac{1}{sin\, 1} + \frac{1}{sin} + \cdots + \frac{1}{sin\, n}.$$
5) Для заданного натурального *n* подсчитать сумму:
$$1 - 2^3 + 3^3 - \cdots + (-1)^{n+1} n^3\ .$$
6) Для заданного натурального *n* подсчитать сумму:
$$cos 1 - cos\, 2 + cos\, 3 - \ldots + (-1)^{n+1} cos\, n\ .$$
7) Для заданного натурального *n* и действительного *x* подсчитать сумму:
$$x - \frac{x^2}{} + \frac{x^3}{3} - \cdots + \frac{(-1)^{n-1} x^n}{n}\ (|x| < 1).$$
8) Для заданного натурального *n* подсчитать сумму:
$$1! - 2! + 3! - \cdots + (-1)^{n+1} n!$$
9) Для заданного натурального *n* и действительного *x* подсчитать сумму:
$$sinx + sinx + sinx^3 + \ldots + sinx^n\ .$$
10) Для заданного натурального *n* подсчитать сумму:
$$1 + \frac{1}{!} + \frac{1}{3!} + \cdots + \frac{1}{n!}.$$
11) Для заданного натурального *n* подсчитать сумму:
$$n + (n+1) + (n+2) + \cdots + (2 \cdot n)\ .$$
12) Для заданного натурального *n* подсчитать сумму:

$$1! + 2! + 3! + \cdots + n!$$

13) Для заданного натурального *n* и действительного *x* подсчитать сумму:

$$1 + x + \frac{x^2}{!} + \cdots + \frac{x^n}{n!}.$$

14) Для заданного натурального *n* и действительного *x* подсчитать сумму:

$$1 - x + x \ - x^3 + \cdots + (-1)^n x^n .$$

15) Для заданного натурального *n* и действительного *x* подсчитать сумму:

$$x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots + (-1)^n \frac{x^{2n+1}}{n+1} \ (|x| < 1).$$

16) Дано натуральное число *n*. Вычислить *n!* ($n! = 1 \times 2 \times 3 \times ... \times n$).

17) Дано натуральное число *n*. Вычислить: $\sin x + \sin 2x + \sin 3x + \ldots$ (*n* слагаемых).

18) Дано натуральное число *n*. Найти сумму $S = 2/5 + 2/9 + 2/13 + \ldots$ (*n* слагаемых).

19) Дано натуральное число *n*. Найти сумму $S = 1 + 1/3 + 3/5 + 5/7 + \ldots$ (*n* слагаемых).

20) Дано натуральное число *n*. Вычислить: $\cos x + 2cox2x + 3\cos 3x + \ldots$ (*n* слагаемых).

21) Дано натуральное число *n*. Вычислить: $x + 2x + 3x + \ldots$ (*n* слагаемых).

22) Дано натуральное число *n*. Вычислить: $2x + 4x + 6x + \ldots$ (*n* слагаемых).

23) Дано натуральное число *n*. Найти сумму $S = 1 - 1/5 + 1/9 - 1/13 + \ldots$ (*n* слагаемых).

24) Дано натуральное число *n*. Вычислить: $1 \times 2 + 2 \times 3 + \ldots + n \times (n+1)$.

25) Дано натуральное число *n*. Найти произведение: $P = (1+x)(3+2x)(5+3x)\ldots$ (*n* множителей).

26) Найти значение суммы ряда $\frac{1}{2} + \frac{1}{4^2} + \cdots$ с точностью $\varepsilon = 10^{-4}$.

27) Задана арифметическая прогрессия. 7.6; 6.3; … . Сколько членов прогрессии нужно сложить, чтобы полученная сумма стала < 0.

28) Задана арифметическая прогрессия 2; 2.8; … . Сколько членов прогрессии нужно сложить, чтобы полученная сумма стала >20.

29) Задана арифметическая прогрессия. 7.1; 5.3; … . Сколько членов прогрессии нужно сложить, чтобы полученная сумма стала < 0.

30) Подсчитать количество двузначных чисел, кратных трем.

2. Списки, кортежи, множества, словари

2.1. Работа с одномерными списками. Создание списка. Операции над списками. Методы списков

Список представляет собой последовательность элементов, пронумерованных от 0, как символы в строке. Для создания списка используется конструктор list.

Создание списков:

- явно указав все элементы списка в тексте (элементы списка перечисляются в квадратных скобках через запятую):

```
# пустой список
lst=[]

# список, состоящий из чисел
Primes = [2, 3, 5, 7, 11, 13]

# список, состоящий из элементов разных типов
lst1= [111, 'Hello', '222', True]

#список, состоящий из строк
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

- с помощью функции list():

```
lst = list()
lst1 = list('Hello')
lst2 = list(range(10,15))
print(lst, lst1, lst2, sep='\n')
```

```
[]
['H', 'e', 'l', 'l', 'o']
[10, 11, 12, 13, 14]
```

- поэлементное заполнение списка, используя метод append(), который добавляет в конец списка указанное значение:

```
A = []
for i in range(int(input('Введите количество элементов: '))):
    A.append(int(input('Введите элемент: ')))

print(f'Список: {A}')
```

```
Введите количество элементов: 4
Введите элемент: 5
Введите элемент: -7
Введите элемент: 42
Введите элемент: -67
Список: [5, -7, 42, -67]
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец.

Операции над списками:

- Конкатенация списков (добавление одного списка в конец другого).
- Повторение списков (умножение списка на число), повторяет список указанное количеств раз.

Пример 2.1.

```
A = [1, 2, 3]
B = [4, 5]
C = A + B
D = B * 3
print ('C =',C)
print ('D =',D)
```

```
C = [1, 2, 3, 4, 5]
D = [4, 5, 4, 5, 4, 5]
```

- Проверка на вхождение (in, not in). Оператор in проверяет наличие значения в списке, оператор not in — отсутствие значения. Возвращают значения True или False.
- Извлечение среза. Со списками, также как и со строками, можно делать срезы. А именно:
  - A[i : j] — срез из j – i элементов A[i], A[i + 1], …, A[j – 1].
  - A[i : j : –1] — срез из i – j элементов A[i], A[i-1], …, A[j + 1] (то есть меняется порядок элементов).
  - A[i : j : k] — срез с шагом k: A[i], A[i + k], A[i + 2*k], … . Если значение k<0, то элементы идут в противоположном порядке.

  Каждое из чисел i или j может отсутствовать, что означает «начало строки» или «конец строки».

Операции со списками в Python:

- x in A — проверить, содержится ли элемент в списке. Возвращает True или False.
- x not in A — то же самое, что not(x in A).
- min(A) — наименьший элемент списка.
- max(A) — наибольший элемент списка.
- A.index(x) — индекс первого вхождения элемента *x* в список, при его отсутствии генерирует исключение ValueError.
- A.count(x) — количество вхождений элемента *x* в список.
- A.sort() — сортировка списка (меняет сам список, ничего не возвращает).
- sum(A) — возвращает сумму элементов в списке.

- A.append(x) — добавить элемент *x* в конец списка.
- A.extend(L) — добавить все элементы списка *L* в конец списка *A*.

Синтаксис генераторов списков в Python:

```
[выражение for переменная in список],
```

где переменная — идентификатор некоторой переменной, список — список значений, который принимает данная переменная (как правило, полученный при помощи функции range), выражение — некоторое выражение, которым будут заполнены элементы списка, как правило, зависящее от использованной в генераторе переменной.

Пример 2.2. Создать список, состоящий из *n* нулей, можно и при помощи генератора.

```
n = 6
A = [0 for i in range(n)]
print('A =', A)
```

```
A = [0, 0, 0, 0, 0, 0]
```

Пример 2.3. Создать список, заполненный квадратами целых чисел от 0 до *n* и от 1 до *n* + 1.

```
n = 6
A = [i ** 2 for i in range(n)]
A1 = [i ** 2 for i in range(1, n + 1)]
print('A =', A)
print('A1 =', A1)
```

```
A = [0, 1, 4, 9, 16, 25]
A1 = [1, 4, 9, 16, 25, 36]
```

Пример 2.4. Создать список, заполненный случайными числами от 1 до 9 (используя функцию randint() из модуля random).

```
from random import randint

n = 6
lst = [randint(1, 9) for i in range(n)]
print('lst =', lst)
```

```
lst = [9, 2, 4, 4, 8, 1]
```

Пример 2.5. Создать список из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции range), потом — заданное количество строк.

```
lst = [input('Введите элемент: ') for i in range(int(input('Введите количество элементов: ')))]
print('lst =', lst)
```

```
Введите количество элементов: 5
Введите элемент: q1
Введите элемент: 4w5
Введите элемент: 561
Введите элемент: 12.3
Введите элемент: hello
lst = ['q1', '4w5', '561', '12.3', 'hello']
```

Метод строки split() возвращает список строк, разрезав исходную строку на части по пробелам по умолчанию: A = input().split(). Если при запуске этой программы ввести строку 1 2 3, то список A будет равен ['1 ', '2 ', '3 '].

У метода split есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка. Например, метод split(';') вернет список, полученный разрезанием исходной строки по символам '; '.

Метод строки join(lst) выводит список при помощи однострочной команды. У этого метода один параметр: список строк. В результате получается строка, полученная соединением элементов списка (которые переданы в качестве параметра) в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод

Пример 2.6.

```
A = ['red', 'green', 'blue']
print(' '.join(A))
print(''.join(A))
print('***'.join(A))
```

```
red green blue
redgreenblue
red***green***blue
```

2.2. Многомерные списки: работа с двумерными массивами

В языке программирования Python таблицу можно представить в виде списка строк, каждый элемент которого является в свою очередь списком, например, чисел. Например, создать числовую таблицу из двух строк и трех столбцов можно так:

A = [[1, 2, 3], [4, 5, 6]].

Здесь первая строка списка A[0] является списком из чисел [1, 2, 3]. То есть A[0][0] == 1, значение A[0][1] == 2, A[0][2] == 3, A[1][0] == 4, A[1][1] == 5, A[1][2] == 6.

Для обработки и вывода списка, как правило, используется два вложенных цикла. Первый цикл по номеру строки, второй цикл по элементам внутри строки. Например, вывести двумерный числовой список на экран построчно, разделяя числа пробелами внутри одной строки, можно так:

```
for i in range(len(A)):
    for j in range(len(A[i])):
        print(A[i][j], end=' ')
    print()
```

```
1 2 3
4 5 6
```

То же самое, но циклы не по индексу, а по значениям списка:

```
for row in A:
    for elem in row:
        print(elem, end=' ')
    print()
```

```
1 2 3
4 5 6
```

Для вывода одной строки можно воспользоваться методом join:

```
for row in A:
    print(' '.join(list(map(str, row))))
```

```
1 2 3
4 5 6
```

Используем два вложенных цикла для подсчета суммы всех чисел в списке:

```
S = 0
for i in range(len(A)):
    for j in range(len(A[i])):
        S += A[i][j]

print('S =', S)
```

```
S = 21
```

Или то же самое с циклом не по индексу, а по значениям строк:

```
S = 0
for row in A:
    for elem in row:
        S += elem

print('S =', S)
```

```
S = 21
```

2.3. Кортежи. Создание кортежа

Кортеж — это неизменяемый список. Кортеж не может быть изменён никаким способом после его создания.

Пример 2.7. Создание кортежа и его использование.

```
a_tuple = ("a", "b", "mpilgrim", "z", "example")
print('1:', a_tuple)
print('2:', a_tuple[0])
print('3:', a_tuple[-1])
print('4:', a_tuple[1:3])
```

```
1: ('a', 'b', 'mpilgrim', 'z', 'example')
2: a
3: example
4: ('b', 'mpilgrim')
```

Выводы:

1. Кортеж определяется так же, как список, за исключением того, что набор элементов заключается в круглые скобки, а не в квадратные.
2. Элементы кортежа заданы в определённом порядке, как и в списке. Элементы кортежа индексируются с нуля, как и элементы списка, таким образом, первый элемент не пустого кортежа — это всегда a_tuple[0].
3. Отрицательные значения индекса отсчитываются от конца кортежа, как и в списке. Последний элемент имеет индекс –1.
4. Создание среза кортежа («slicing») аналогично созданию среза списка. Когда создаётся срез списка, получается новый список; когда создаётся срез кортежа, получается новый кортеж.

Особенности кортежа:

- Нельзя добавить элементы к кортежу. Кортежи не имеют методов append() или extend().
- Нельзя удалять элементы из кортежа. Кортежи не имеют методов remove() или pop().
- Можно искать элементы в кортежи, поскольку это не изменяет кортеж.
- Можно использовать оператор in, чтобы проверить существует ли элемент в кортеже.

Встроенная функция tuple() принимает список и возвращает кортеж из всех его элементов, функция list() принимает кортеж и возвращает список.

Кортежи в логическом контексте можно использовать в операторе if.

- В логическом контексте пустой кортеж является ложью.

- Любой кортеж, состоящий, по крайней мере, из одного элемента, — истина.
- Любой кортеж, состоящий, по крайней мере, из одного элемента, — истина. Значения элементов не важны.
- Чтобы создать кортеж из одного элемента, необходимо после него поставить запятую. Без запятой Python предполагает, что вы просто добавили еще одну пару скобок, что не делает ничего плохого, но и не создает кортеж.

2.4. Множества. Создание множеств

Множество — это неупорядоченная коллекция уникальных элементов, с которой можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Множество может состоять из различных элементов, порядок элементов во множестве не определен.

Элементами множества может быть любой неизменяемый тип данных: числа, строки, кортежи. Изменяемые типы данных не могут быть элементами множества, в частности, нельзя сделать элементом множества список (но можно сделать кортеж) или другое множество. Каждый элемент может входить во множество только один раз, порядок задания элементов не важен.

Задание множеств:

- множество задается перечислением всех его элементов в фигурных скобках.

```
A = {1, 2, 3}
```

- с помощью функции set()

```
# пустое множество
s1 = set()

# преобразуем строку в множество
s2 = set('asdfghj')

# преобразуем список в множество
s3 = set ([1, 2, 3, 1, 2, 3])

# преобразуем словарь в множество
s4 = set({'a': 1, 'b': 2})

print(s1, s2, s3, s4, sep='\n')
```

```
set()
{'a', 's', 'g', 'f', 'd', 'h', 'j'}
{1, 2, 3}
{'a', 'b'}
```

Пример 2.8. Сформировать множество из строки.

```
A = set('qwerty')
print(A)
```

```
{'e', 'q', 'r', 't', 'y', 'w'}
```

Операции над множествами:

- Чтобы проверить, принадлежит ли значение множеству, используйте оператор in. Он работает так же, как и для списков.
- Метод union() (объединение (|)) возвращает новое множество, содержащее все элементы каждого из множеств.
- Метод intersection() (пересечение (&)) возвращает новое множество, содержащее все элементы, которые есть и в первом множестве, и во втором.
- Метод difference() (разность (–)) возвращает новое множество, содержащее все элементы, которые есть во множестве a_set, но которых нет во множестве b_set.
- Метод symmetric_difference() (симметрическая разность (^)) возвращает новое множество, которое содержит только уникальные элементы обоих множеств.

Примеры применения операций:

```
a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
```

```
30 in a_set
```

```
True
```

```
31 in a_set
```

```
False
```

```
b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
a_set.union(b_set)
```

```
{1, 2, 3, 4, 5, 6, 8, 9, 12, 15, 17, 18, 21, 30, 51, 76, 127, 195}
```

```
a_set.intersection(b_set)
```

```
{2, 5, 9, 12, 21}
```

```
a_set.difference(b_set)
```

```
{4, 30, 51, 76, 127, 195}
```

```
a_set.symmetric_difference(b_set)
```

```
{1, 3, 4, 6, 8, 15, 17, 18, 30, 51, 76, 127, 195}
```

- copy() — создает копию множества.
- add() — добавляет во множество.

- remove() — удаляет из множества. Если элемент не найден, то возбуждается исключение KeyError.
- discard() — удаляет из множества, если он присутствует.
- pop() — удаляет произвольный элемент из множества и возвращает его. Если элементов нет, то возбуждается исключение KeyError.
- clear() — удаляет все элементы из множества.

Обход элементов множества в цикле for:

```
s2 = set ([1, 2, 3, 4, 5])
for v in s2:
    print(v, end=' ')
```

```
1 2 3 4 5
```

С множествами в Python можно выполнять обычные для математики операции над множествами:

- A |= B или A.update(B) — добавляет во множество *A* все элементы из множества *B*.
- A &= B или A.intersection_update(B) — оставляет во множестве *A* только те элементы, которые есть во множестве *B*.
- A –= B или A.difference_update(B) — удаляет из множества *A* все элементы, входящие во множество *B*.
- A ^= B или A.symmetric_difference_update(B) — записывает в *A* симметрическую разность множеств *A* и *B*.
- A <= B или A.issubset(B) — возвращает True, если *A* является подмножеством *B*.
- A >= B или A.issuperset(B) — возвращает True, если *A* является надмножеством *B*.
- A < B — эквивалентно A <= B and A != B.
- A > B — эквивалентно A >= B and A != B.

При использовании формы A.update(B) и аналогичных, параметр *B* может также являться списком, строкой и вообще любым итерируемым объектом. Так, запись A.update([1, 2, 1]) корректна (во множество будут добавлены элементы 1 и 2), а запись A |= [1, 2, 1] — породит исключение.

2.5. Словари. Создание словаря. Операции над словарями. Методы для работы со словарями

Словари в языке Python относится к типу данных, которые представляют собой неупорядоченные коллекции элементов, состоящих из пар ключ-значение. Словари создаются с помощью фигурных скобок {} и могут быть изменяемыми.

Способы создания словарей:

- использование фигурных скобок {}:

```
# пустой словарь
my_dict = {}

# словарь с несколькими парами ключ-значение
student = {
    "имя": "Анна",
    "возраст": 20,
    "группа": "А7"
}
```

- использование функции dict():

```
# пустой словарь
my_dict = dict()

# словарь с несколькими парами ключ-значение
student = dict(имя="Анна", возраст=20, группа="А7")
```

- использование функции zip():

```
keys = ["имя", "возраст", "группа"]
values = ["Анна", 20, "А7"]
student = dict(zip(keys, values))
print(student)
```

```
{'имя': 'Анна', 'возраст': 20, 'группа': 'А7'}
```

- использование генератора словаря:

```
student = {f"предмет_{i}": i * 10 for i in range(1, 6)}
print(student)
```

```
{'предмет_1': 10, 'предмет_2': 20, 'предмет_3': 30, 'предмет_4': 40, 'предмет_5': 50}
```

- использование метода fromkeys():

```
keys = ["имя", "возраст", "группа"]
default_value = None
student = dict.fromkeys(keys, default_value)
print(student)
```

```
{'имя': None, 'возраст': None, 'группа': None}
```

Операции над словарями:

- доступ к элементам по ключу:

```
my_dict = {"имя": "Анна", "возраст": 20}
name = my_dict["имя"]
age = my_dict.get("возраст")
print(name)
print(age)
```

```
Анна
20
```

- проверка наличия ключа в словаре:

```
my_dict = {"имя": "Анна", "возраст": 20}
name = my_dict["имя"]
age = my_dict.get("возраст")
print(name)
print(age)
```

```
Анна
20
```

- del: удаляет из словаря элемент с заданным ключом. Если элемент с таким ключом отсутствует, то возникает ошибка (возбуждается исключение KeyError).

Основные методы работы со словарями:

- keys(): возвращает список всех ключей в словаре.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
keys = my_dict.keys()
print(keys)
```

```
dict_keys(['имя', 'возраст', 'группа'])
```

- values(): возвращает список всех значений в словаре.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
values = my_dict.values()
print(values)
```

```
dict_values(['Анна', 20, 'А7'])
```

- items(): возвращает список кортежей (пар ключ-значение) в словаре.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
items = my_dict.items()
print(items)
```

```
dict_items([('имя', 'Анна'), ('возраст', 20), ('группа', 'А7')])
```

- get(): возвращает значение, связанное с указанным ключом. Если ключ не существует, возвращает указанное значение по умолчанию.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
age = my_dict.get("возраст", 0)
grade = my_dict.get("класс", "-")
print(age, grade, sep='\n')
```

```
20
-
```

- pop(): удаляет элемент из словаря по указанному ключу и возвращает его значение. Если ключ не найден, возвращает указанное значение по умолчанию.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
name = my_dict.pop("имя")
grade = my_dict.pop("класс", "-")
print(name, grade, sep='\n')
```

```
Анна
-
```

- update(): обновляет словарь, добавляя пары ключ-значение из другого словаря или итерируемого объекта.

```
my_dict = {"имя": "Анна", "возраст": 20}
additional_info = {"группа": "А7", "рейтинг": 4.5}
my_dict.update(additional_info)
print(my_dict)
```

```
{'имя': 'Анна', 'возраст': 20, 'группа': 'А7', 'рейтинг': 4.5}
```

- clear(): удаляет все элементы из словаря.

```
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
my_dict.clear()
print(my_dict)
```

```
{}
```

Перебор элементов словаря. Перебор всех элементов словаря выполняется с помощью цикла for. Если в заголовке цикла указать просто имя словаря, то переменная цикла на каждой итерации будет равна ключу очередного элемента словаря.

```python
# перебор ключей
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
for key in my_dict:
    print(key)
```

```
имя
возраст
группа
```

Если использовать в заголовке цикла метод values(), то переменная цикла на каждой итерации будет равна значению очередного элемента словаря.

```python
# перебор значений
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
for value in my_dict.values():
    print(value)
```

```
Анна
20
А7
```

Если в заголовке цикла использовать метод keys(), то переменная цикла на каждой итерации будет равна ключу очередного элемента словаря.

```python
# перебор ключей
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
for k in my_dict. keys():
    print(k)
```

```
имя
возраст
группа
```

Если использовать в заголовке цикла метод items(), то переменные цикла (k и v) на каждой итерации будут равны ключи и соответствующему значению очередного элемента словаря.

```python
# перебор пар ключ-значение
my_dict = {"имя": "Анна", "возраст": 20, "группа": "А7"}
for key, value in my_dict.items():
    print(f'{key}: {value}')
```

```
имя: Анна
возраст: 20
группа: А7
```

2.6. Задачи для самостоятельного решения

Задачи 2.1. Списки.

1) Дан список, включающий N элементов, и целые числа K и L ($1 \le K \le L \le N$). Найти среднее арифметическое элементов списка с порядковыми номерами от K до L включительно.
2) Дан список, включающий N элементов, и целые числа K и L ($1 < K \le L \le N$). Найти сумму всех элементов списка, кроме элементов с порядковыми номерами от K до L включительно.
3) Дан список, включающий N элементов, и целые числа K и L ($1 < K \le L \le N$). Найти среднее арифметическое всех элементов списка, кроме элементов с порядковыми номерами от K до L включительно.
4) Дан список. Вывести вначале все содержащиеся в данном списке четные числа в порядке возрастания их индексов, а затем — все нечетные числа в порядке убывания их индексов.
5) Дан список. Вывести вначале его элементы с четными номерами (в порядке возрастания номеров), а затем — элементы с нечетными номерами (также в порядке возрастания номеров).
6) Дан список. Найти произведение всех элементов списка, расположенных справа от максимального элементами, не включая максимальный элемент.
7) Дан список. Найти сумму всех элементов списка, расположенных между его минимальным и максимальным элементами, включая минимальный и максимальный элементы.
8) Дан список. Обнулить элементы списка, расположенные между его минимальным и максимальным элементами (не включая минимальный и максимальный элементы).
9) Дан список. Увеличить все четные числа, содержащиеся в списке, на заданное число. Если четные числа в списке отсутствуют, то оставить список без изменений.
10) Дан список. Обнулить все нечетные числа, содержащиеся в списке. Если нечетные числа в списке отсутствуют, то оставить список без изменений.
11) Дан список. Найти минимальный элемент из его элементов с четными номерами.
12) Дан список. Найти номера тех элементов списка, которые больше своего правого соседа, и количество таких элементов. Найденные номера выводить в порядке их возрастания.
13) Дан список. Найти два соседних элемента, сумма которых максимальна, и вывести

эти элементы в порядке возрастания их индексов.

14) Дан список. Поменять местами его минимальный и максимальный элементы.
15) Даны целые числа N (> 2), A и B. Сформировать и вывести список размера N, первый элемент которого равен A, второй равен B, а каждый последующий элемент равен сумме всех предыдущих.
16) Дан список, включающий N элементов, и целые числа K и L ($1 \le K \le L \le N$). Найти среднее арифметическое всех четных элементов списка с порядковыми номерами от K до L включительно.
17) Дан список, включающий N элементов, и целые числа K и L ($1 < K \le L \le N$). Найти сумму всех нечетных элементов списка, кроме элементов с порядковыми номерами от K до L включительно.
18) Дан список, включающий N элементов, и целые числа K и L ($1 < K \le L \le N$). Найти среднее арифметическое всех нечетных элементов списка, кроме элементов с порядковыми номерами от K до L включительно/
19) Дан список. Вывести вначале два числа с максимальными значениями в порядке убывания их индексов, а затем — два числа с минимальными значениями в порядке убывания их индексов.
20) Дан список. Вывести вначале его элементы с нечетными номерами (в порядке возрастания номеров), а затем — элементы с четными номерами (также в порядке возрастания номеров).
21) Дан список. Найти произведение всех элементов списка, расположенных справа от минимального элемента, не включая минимальный элемент.
22) Дан список. Найти произведение всех элементов списка, расположенных между его минимальным и максимальным элементами, включая минимальный и максимальный элементы.
23) Дан список. Увеличить в два раза все элементы списка, расположенные между его минимальным и максимальным элементами (не включая минимальный и максимальный элементы).
24) Дан список. Увеличить все нечетные числа, содержащиеся в списке, на заданное число. Если нечетные числа в списке отсутствуют, то оставить список без изменений.
25) Дан список. Обнулить все четные числа, содержащиеся в списке. Если четные числа в списке отсутствуют, то оставить список без изменений.
26) Дан список. Найти максимальный элемент из его элементов с нечетными номерами.

27) Дан список. Найти количество пар элементов списка, которые являются взаимно противоположными.
28) Дан список. Найти два соседних элемента, сумма которых нулю, и вывести эти элементы в порядке возрастания их индексов.
29) Дан список. Получить список, в котором не будет повторяющихся элементов..
30) Даны целые числа N (> 2), A и B. Сформировать и вывести список размера N, первый элемент которого равен A, а каждый последующий элемент увеличивается на B относительно предыдущего.

Задачи 2.2. Сортировка списков.

1) Дан список. Выбрать из списка все нечетные числа и упорядочить их по убыванию.
2) Дан список. Выбрать из списка все положительные числа и упорядочить их по возрастанию.
3) Дан список. Выбрать из списка все положительные числа и упорядочить их по убыванию.
4) Дан список. Выбрать из списка все четные числа и упорядочить их по возрастанию.
5) Дан список. Выбрать из списка все числа больше заданного числа k и упорядочить их по убыванию.
6) Дан список. Выбрать из списка все числа больше 10 и упорядочить их по возрастанию.
7) Дан список. Выбрать из списка все числа кратные 5 и упорядочить их по убыванию.
8) Дан список. Выбрать из списка все числа меньше заданного числа k и упорядочить их по возрастанию.
9) Дан список. Выбрать из списка все числа меньше 15 и упорядочить их по убыванию.
10) Дан список. Выбрать из списка все числа кратные 3 и упорядочить их по возрастанию.
11) Дан список. Выбрать из списка все числа кратные заданному числу k и упорядочить их по убыванию.
12) Дан список. Выбрать из списка все отрицательные числа и упорядочить их по возрастанию.
13) Дан список. Выбрать из списка все числа на нечетных позициях и упорядочить эти числа по убыванию.
14) Дан список. Выбрать из списка все двузначные числа и упорядочить эти числа по

возрастанию.

15) Дан список. Выбрать из списка все числа на четных позициях и упорядочить эти числа по возрастанию.
16) Дан список. Выбрать из списка все двузначные числа и упорядочить эти числа по возрастанию их суммы цифр.
17) Дан список. Выбрать из списка все двузначные числа и упорядочить эти числа по возрастанию последней цифры.
18) Дан список, состоящий из трехзначных чисел. Выполнить сортировку элементов списка по убыванию суммы цифр элементов списка.
19) Дан список, состоящий из трехзначных чисел. Выполнить сортировку элементов списка по возрастанию суммы цифр элементов списка.
20) Дан список, состоящий из двузначных чисел. Выполнить сортировку элементов списка по убыванию остатков от деления на число k.
21) Дан список строк. Отсортируйте его в алфавитном порядке.
22) Дан список строк. Отсортируйте его по длине строк — от самых коротких к самым длинным.
23) Дан список строк. Отсортируйте список в порядке убывания алфавитного порядка, но игнорируйте регистр букв.
24) Дан список чисел. Отсортируйте список, переместив все отрицательные числа в начало списка, а положительные — в конец.
25) Дан список строк. Отсортируйте список в порядке убывания алфавитного порядка по первой букве каждой строки, регистр букв учитывайте.
26) Дан список строк. Отсортируйте список в порядке количества символов в элементах строки.
27) Дан список. Выбрать из списка все двузначные числа и упорядочить эти числа по возрастанию первой цифры.
28) Дан список. Выбрать из списка все двузначные числа и упорядочить эти числа по убыванию первой цифры.

Задачи 2.3. Вложенные списки.

1) Дана целочисленная матрица размера $M \times N$. Найти номер первого из ее столбцов, содержащих только нечетные числа. Если таких столбцов нет, то вывести 0.
2) Дана матрица размера $M \times N$. Преобразовать матрицу, поменяв местами минимальный и максимальный элемент в каждой строке.
3) Дана матрица размера $M \times N$. Поменять местами строки, содержащие минимальный

и максимальный элементы матрицы.

4) Дана матрица размера $M \times N$. Зеркально отразить ее элементы относительно горизонтальной оси симметрии матрицы (при этом поменяются местами строки с номерами 1 и M, 2 и M — 1 и т. д.).
5) Дана матрица размера $M \times N$. Удалить строку, содержащую минимальный элемент матрицы.
6) Дана матрица размера $M \times N$. Удалить столбец, содержащий максимальный элемент матрицы.
7) Дана матрица размера $M \times N$. Зеркально отразить ее элементы относительно вертикальной оси симметрии матрицы (при этом поменяются местами столбцы с номерами 1 и N, 2 и N — 1 и т. д.).
8) Дана матрица размера $M \times N$. Продублировать строку матрицы, содержащую ее максимальный элемент.
9) Дана матрица размера $M \times N$. В каждом ее столбце найти количество элементов, больших среднего арифметического всех элементов этого столбца.
10) Дана матрица размера $M \times N$ (M и N — четные числа). Поменять местами левую верхнюю и правую нижнюю четверти матрицы.
11) Дана матрица размера $M \times N$. Для каждой строки матрицы с нечетным номером (1, 3, …) найти среднее арифметическое ее элементов.
12) Дана матрица размера $M \times N$. Найти номер ее строки с наибольшей суммой элементов и вывести данный номер, а также значение наибольшей суммы.
13) Дана матрица размера $M \times N$. Перед строкой матрицы, содержащей минимальный элемент матрицы, вставить строку из нулей.
14) Дана целочисленная матрица размера $M \times N$. Найти номер последней из ее строк, содержащих только четные числа. Если таких строк нет, то вывести 0.
15) Дана матрица размера $M \times N$. Поменять местами столбец с номером 1 и последний из столбцов, содержащих только положительные элементы. Если требуемых столбцов нет, то вывести матрицу без изменений.
16) Дана матрица размера $M \times N$. Найти номер первого столбца, содержащего максимальное количество отрицательных элементов. Если таких столбцов нет, вывести 0.
17) Дана матрица размера $M \times N$. Найти номер последнего столбца, содержащего только нулевые элементы. Если таких столбцов нет, вывести 0.
18) Дана матрица размера $M \times N$. Найти номер последней из ее строк, содержащих только положительные элементы. Если таких строк нет, вывести 0.

19) Дана матрица размера $M \times N$. Преобразовать матрицу, поменяв местами минимальный и максимальный элемент в каждом столбце.
20) Дана матрица размера $M \times N$. Удалить столбец, содержащий минимальный элемент матрицы.
21) Дана матрица размера $M \times N$. Зеркально отразить ее элементы относительно главной диагонали (при этом поменяются местами строки и столбцы).
22) Дана матрица размера $M \times N$. Продублировать столбец матрицы, содержащий ее минимальный элемент.
23) Дана матрица размера $M \times N$. В каждой ее строке найти количество элементов, меньших среднего арифметического всех элементов этой строки.
24) Дана матрица размера $M \times N$. Найти номер ее столбца с наименьшей суммой элементов и вывести данный номер, а также значение наименьшей суммы.
25) Дана матрица размера $M \times N$. Перед столбцом матрицы, содержащим максимальное значение элемента матрицы, вставить столбец из нулей.
26) Дана матрица размера $M \times N$. Найти сумму элементов каждой ее строки и вывести номера строк, сумма элементов которых является простым числом.
27) Дана матрица размера $M \times N$. Заменить все отрицательные элементы матрицы их модулем.
28) Дана матрица размера $M \times N$. Заменить все элементы матрицы, которые превышают среднее арифметическое всех элементов матрицы, их квадратом.
29) Дана матрица размера $M \times N$. Найти номер последней из ее строк, содержащей только отрицательные элементы. Если таких строк нет, вывести 0.
30) Дана матрица размера $M \times N$. Поменять местами столбцы с наименьшим и наибольшим количеством отрицательных элементов. Если таких столбцов нет, вывести матрицу без изменений.

Задача 2.4. Кортежи.

1) Напишите программу: а) для создания кортежа; б) для преобразования кортежа в словарь.
2) Напишите программу: а) для создания кортежа с различными типами данных; б) для распаковки списка кортежей в отдельные списки.
3) Напишите программу: а) для создания кортежа с числами и выведите на экран первый элемент; б) для того, чтобы перевернуть элементы кортежа в обратном порядке.
4) Напишите программу: а) для распаковки кортежа в несколько переменных; б) для

преобразования списка кортежей в словарь.

5) Напишите программу: а) для добавления элемента в кортеж; б) для печати кортежа со строковым форматированием. Например, задан кортеж: (100, 200, 300), на экран вывести: это кортеж (100, 200, 300).
6) Напишите программу: а) для преобразования кортежа в строку; б) для замены последнего элемента в каждом кортеже в списке кортежей на заданное значение.
7) Напишите программу: а) чтобы получить четвёртый элемент кортежа с начала и четвёртый элемент с конца; б) для удаления пустого кортежа (или кортежей) из списка кортежей.
8) Напишите программу: а) для создания кортежа словарей; б) для сортировки кортежа по его элементу, являющегося числом с плавающей точкой.
9) Напишите программу: а) для поиска повторяющихся элементов кортежа; б) для подсчёта элементов в списке до тех пор, пока элемент не будет кортежем.
10) Напишите программу: а) чтобы проверить, включает ли кортеж заданный; б) преобразующую заданный список строк в кортеж.
11) Напишите программу: а) для создания кортежа из списка, исключая повторяющиеся элементы; б) для удаления элемента с заданным значением из кортежа.
12) Напишите программу: а) для получения уникальных элементов двух кортежей; б) для объединения двух кортежей в один.
13) Напишите программу: а) для проверки, является ли кортеж пустым; б) для нахождения количества вхождений элемента в кортеж.
14) Напишите программу: а) для создания списка кортежей из заданного списка; б) для сортировки списка кортежей по последнему элементу в каждом кортеже.
15) Напишите программу: а) для получения индекса заданного значения в кортеже; б) для создания нового кортежа, содержащего только четные элементы из заданного кортежа.
16) Напишите программу: а) для создания кортежа из строки, содержащей числа, разделенные запятой; б) для сортировки кортежа по длине строк в нем.
17) Напишите программу: а) для создания кортежа из списка и его сортировки в обратном порядке; б) для преобразования строки в кортеж, разбивая ее на слова.
18) Напишите программу: а) для создания кортежа из заданного списка, исключая повторяющиеся элементы; б) для объединения двух кортежей, исключая повторяющиеся элементы.
19) Напишите программу: а) для сравнения двух кортежей на равенство; б) для объединения заданного кортежа с элементами другого кортежа.

20) Напишите программу: а) для создания кортежа из заданного списка чисел и его сортировки; б) для замены всех вхождений заданного элемента в кортеже на новое значение.
21) Напишите программу: а) для удаления заданного элемента из кортежа; б) для получения подкортежа из заданного кортежа, указывая начальный и конечный индексы.
22) Напишите программу: а) для создания кортежа из списка, исключая элементы, не являющиеся числами; б) для получения списка ключей и значений из словаря, а затем создания кортежа из этих списков.
23) Напишите программу: а) для создания кортежа из списка чисел и его сортировки по убыванию; б) для вывода на экран элементов кортежа в столбик.
24) Напишите программу: а) для создания кортежа из заданного списка строк и его сортировки по алфавиту; б) для нахождения максимального и минимального элементов в кортеже.
25) Напишите программу: а) для получения первых трех элементов кортежа; б) для вывода на экран уникальных значений из кортежа.
26) Напишите программу: а) для проверки, является ли кортеж пустым; б) для подсчета количества вхождений каждого элемента в кортеже.
27) Напишите программу: а) для получения среза из кортежа, начиная с третьего элемента и заканчивая предпоследним; б) для конкатенации двух кортежей.
28) Напишите программу: а) для создания кортежа из заданного списка, заменяя каждое четное значение на ноль; б) для проверки, все ли элементы кортежа удовлетворяют заданному условию.
29) Напишите программу: а) для получения пересечения двух кортежей; б) для проверки, есть ли заданный элемент в кортеже.
30) Напишите программу: а) для создания кортежа, содержащего элементы кортежей из заданного списка; б) для проверки, является ли кортеж симметричным, т.е. совпадают ли его элементы при обратном порядке.

Задача 2.5. Множества.

1) Дана непустая последовательность символов. Построить и напечатать множества, элементами которых являются встречающиеся в последовательности: а) цифры от «0» до «9» и знаки арифметических операций; б) буквы от «A» до «F» и от «X» до «Z».
2) Используя множества, подсчитать общее количество цифр и знаков «+», «–», «*» в строке, введённой с клавиатуры.

3) Составить программу формирования множества строчных латинских букв, входящих в строку, введённую с клавиатуры, и подсчёта количества знаков препинания в ней.
4) Используя множества, вывести общие буквы трёх предложений.
5) Используя множества, вывести наибольшие цифры трёх целых чисел.
6) Используя множества, подсчитать количество цифр в заданной строке и напечатать их.
7) Используя множества, вывести различные буквы трёх предложений, то есть такие, какие есть только в одном из них.
8) Даны три строки. Используя множества, определить, можно ли из символов первых двух строк получить третью строку.
9) Используя множества, напечатать по одному разу в алфавитном порядке все строчные русские гласные буквы, входящие в заданный текст.
10) Используя множества, напечатать в возрастающем порядке все цифры, входящие в десятичную запись данного десятичного числа.
11) Дан текст. Используя множества, определить, каких букв больше — гласных или согласных.
12) Дано натуральное число. Используя множества, напечатать в возрастающем порядке все цифры, которых нет в записи данного числа.
13) Написать программу, чтобы удалить из первого множества пересечение второго множества с первым.
14) Дан текст. Используя множества, напечатать в алфавитном порядке все согласные буквы, которые не входят в текст.
15) Проверить, нет ли у двух заданных множеств общих элементов.
16) Используя множества, определить, состоят ли две заданные строки из одинаковых символов.
17) Даны два списка. Используя множества, найти общие элементы в обоих списках.
18) Используя множества, найти разность двух заданных множеств.
19) Даны две строки. Используя множества, определить, есть ли общие символы в этих строках.
20) Дан список. Используя множества, найти количество уникальных элементов в списке.
21) Даны два множества. Используя множества, определить, является ли одно множество подмножеством другого.
22) Даны два списка. Используя множества, определить, есть ли общие элементы

в этих списках.

23) Используя множества, определить, является ли заданная строка палиндромом.
24) Даны два множества. Используя множества, найти объединение данных множеств.
25) Дан список. Используя множества, определить, есть ли дублирующиеся элементы в этом списке.
26) Даны два множества. Используя множества, найти пересечение данных множеств.
27) Дан список. Используя множества, определить, является ли список упорядоченным в порядке возрастания.
28) Даны два множества. Используя множества, определить, есть ли общие элементы в этих множествах.
29) Дан список. Используя множества, определить, являются ли все элементы в списке уникальными.
30) Даны два множества. Используя множества, найти разность между этими двумя множествами.

Задача 2.6. Словари.

1) Напишите программу: а) для сортировки (по возрастанию и убыванию) словаря по значению; б) для объединения двух словарей и суммирующую значения для общих ключей; в) для проверки наличия нескольких ключей в словаре.
2) Напишите программу: а) для добавления ключа в словарь; б) для печати всех уникальных значений в словаре; в) для подсчёта количества элементов в значениях словаря, представляющих собой списки.
3) Напишите программу: а) для объединения словарей в один новый словарь; б) для сортировки словаря по значению; в) чтобы очистить список значений в словаре, если значениями словаря являются списки.
4) Напишите программу: а) чтобы проверить, существует ли данный ключ в словаре; б) для создания и отображения всех комбинаций букв, выбирая каждую букву из другого ключа в словаре, например, задан словарь {'1': ['a', 'b'], '2': ['c', 'd']}, результат: ac, ad, bc, bd; в) для замены значений словаря их средним значением.
5) Напишите программу: а) для перебора словарей с использованием цикла for; б) чтобы найти три наибольших значения соответствующих ключей в словаре; в) для сопоставления значений ключей в двух словарях.
6) Напишите программу для создания и печати словаря, содержащего число (от 1 до n) в форме (x, x*x), например, при $n = 5$ получить словарь {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}; б) для объединения значений в списке словарей; в) для удаления пус-

тых элементов из заданного словаря.

7) Напишите программу: для печати словаря, где ключами являются числа от 1 до 15 (оба включены), а значения представляют собой квадрат ключей; б) для создания словаря из строки, где значение — число вхождений символа в строку; в) для фильтрации словаря на основе значений.
8) Напишите программу: а) для объединения двух словарей; б) для печати словаря в табличном формате; в) для преобразования нескольких списков во вложенный словарь.
9) Напишите программу: а) для перебора словарей с использованием цикла for; б) для подсчёта суммы значений, связанных с ключом в списке словарей; в) для фильтрации высоты и ширины учащихся, которые хранятся в справочнике.
10) Напишите программу: а) для суммирования всех элементов в словаре; б) для преобразования списка во вложенный словарь ключей; в) чтобы проверить, все ли значения в словаре одинаковы.
11) Напишите программу: а) для перемножения всех элементов в словаре; б) для сортировки списка в алфавитном порядке в словаре; в) для создания словаря, группирующего последовательность пар ключ-значение в словарь списков.
12) Напишите программу: а) для удаления ключа из словаря; б) для удаления пробелов из ключей словаря; в) для разделения данного словаря списков на список словарей.
13) Напишите программу: а) для сопоставления двух списков в словаре; б) чтобы получить три самых дорогих товара в магазине; в) для удаления указанного словаря из заданного списка.
14) Напишите программу: а) для сортировки заданного словаря по ключу; б) чтобы получить ключ, значение и элемент в словаре; в) для преобразования строковых значений данного словаря в целочисленные (или числа с плавающей точкой).
15) Напишите программу: а) чтобы получить максимальное и минимальное значение в словаре; б) для печати в отдельные строки ключей и значений словаря; в) для создания словаря ключей *x*, *y* и *z*, где каждый ключ имеет значение списка из 11–20, 21–30 и 31–40 соответственно. Получите доступ к пятому значению каждого ключа из словаря.
16) Напишите программу, которая принимает список словарей и объединяет их в один словарь.
17) Напишите программу, которая проверяет, все ли значения в словаре являются уникальными.
18) Напишите программу, которая находит ключ с наибольшим значением в словаре.

19) Напишите программу, которая находит сумму всех значений в словаре.
20) Напишите программу, которая заменяет все значения в словаре на их квадраты.
21) Напишите программу, которая находит наибольшее значение в словаре и выводит соответствующий ключ.
22) Напишите программу, которая удаляет все ключи со значениями, меньшими заданного числа.
23) Напишите программу, которая находит ключи с самыми длинными значениями в словаре.
24) Напишите программу, которая сортирует словарь по ключам в обратном порядке.
25) Напишите программу, которая удаляет все элементы из словаря, у которых ключи не начинаются с заданной буквы.
26) Напишите программу, которая находит пересечение двух словарей.
27) Напишите программу, которая находи все ключи, содержащие заданную подстроку.
28) Напишите программу, которая находит сумму всех численных значений в словаре.
29) Напишите программу, которая находит ключ с минимальным значением в словаре.
30) Напишите программу, которая находит разницу между двумя словарями.

3. Функции. Анонимные функции

3.1. Функции. Способы задания функций

Функции в Python — это блоки кода, которые могут принимать аргументы, выполнять определенные действия и возвращать результаты. Они позволяют группировать код для выполнения конкретной задачи и повторно использовать его в программе. Функции делают код более модульным, упрощают его чтение и обеспечивают повторное использование кода.

Функция в Python имеет следующую структуру:

```
def function_name(argument1, argument2, ...):
    # Код функции
    return result
```

- def — это ключевое слово, которое говорит о начале определения функции.
- function_name — это имя функции, которое должно быть допустимым идентификатором.
- argument1, argument2, ... — это аргументы функции, это значения, которые можно передать в функцию для ее выполнения.
- return — это ключевое слово для возврата результата.

Функция вызывается с помощью конструкции:

```
имя_функции([значения_параметров]).
```

Пример 3.1. Написать функцию, которая принимает два числа и возвращает их сумму.

```
def add_numbers(a, b):
    sum = a + b
    return sum
```

```
add_numbers(5, 6)
```

```
11
```

Способы задания функции в Python:

- без аргументов:

```
def greet():
    print("Hello!")
```

```
greet()
```

```
Hello!
```

- с одним аргументом:

```
def square(x):
    return x * x
```

```
square(5)
```

```
25
```

- с несколькими аргументами:

```
def multiply(a, b):
    return a * b
```

```
multiply(12, 5)
```

```
60
```

- с аргументами по умолчанию:

```
def power(base, exponent=2):
    return base ** exponent
```

```
power(2)
```

```
4
```

```
power(2, 8)
```

```
256
```

В данном случае, если аргумент exponent не указан при вызове функции, будет использовано значение по умолчанию равное 2.

- передача аргументов с использованием их имени:

```
def greet(name, age):
    print("Hello,", name + "! You are", age, "years old.")
```

```
greet(name="Alice", age=25)
```

```
Hello, Alice! You are 25 years old.
```

В данном случае, при вызове функции greet() указываются имена аргументов и их значения.

- возвращение нескольких значений:

```
def calculate(x, y):
    summa = x + y
    difference = x - y
    return summa, difference
```

```
result = calculate(10, 5)
print(result)
```

```
(15, 5)
```

В данном случае, функция calculate() возвращает два значения — сумму и разность переданных аргументов.

3.2. Глобальные и локальные переменные

В Python переменные могут быть локальными или глобальными в зависимости от области их видимости.

- Локальные переменные объявляются внутри функций или методов и доступны только внутри этой функции или метода. Они создаются при входе в функцию и уничтожаются при выходе из нее. Пример:

```
def calculate():
    num1 = 10
    num2 = 20
    result = num1 + num2
    print(result)
```

```
calculate()
```

```
30
```

```
print(num1)
```

```
NameError: name 'num1' is not defined
```

В данном примере переменные num1, num2 и result являются локальными для функции calculate. Они не видны вне функции, поэтому при попытке обратиться к переменной num1 после вызова функции возникает ошибка.

- Глобальные переменные объявляются вне функций или методов и доступны во всей программе. Они обычно объявляются до использования функций или методов, чтобы они были видны внутри этих блоков кода. Пример:

```
global_var = 10
def print_global_var():
    print(global_var)
```

```
print_global_var()
```

```
10
```

```
def change_global_var():
    global global_var
    global_var = 20
```

```
change_global_var()
print(global_var)
```

```
20
```

В данном примере переменная global_var объявляется в глобальной области видимости. Она видна как внутри функции print_global_var, так и внутри функции change_global_var. Если мы хотим изменить значение глобальной переменной внутри функции, то должны использовать ключевое слово global перед ее именем.

3.3. Анонимные функции

Анонимные функции в Python создаются с использованием ключевого слова lambda, за которым следует список аргументов в скобках, двоеточие и выражение, которое будет выполнено при вызове функции. Возвращаемое значение указывается после ключевого слова lambda. Рассмотрим примеры:

- функция, возвращающая квадрат числа:

```
square = lambda x: x ** 2
print(square(5))
```

```
25
```

- функция, складывающая два числа:

```
add = lambda a, b: a + b
print(add(5, 3))
```

```
8
```

- использование анонимной функции внутри другой функции:

```
def apply_operation(x, y, operation):
    return operation(x, y)

result = apply_operation(4, 2, lambda a, b: a * b)

print(result)
```

```
8
```

Анонимные функции полезны в случаях, когда не требуется создание именованных функций или когда нужна функция только для использования в одном месте программы.

3.4. Задачи для самостоятельного решения

Задача 3.1. Функции.

1) Описать функцию Mean, вычисляющую среднее арифметическое AMean и среднее геометрическое GMean двух положительных чисел *X* и *Y* формулам:

$$AMean = (X + Y)/2,\ GMean = X * Y.$$

2) Описать функцию Circle, находящую площадь и длину окружности круга радиуса *R*.
3) Описать функцию TrianglePS, вычисляющую по стороне *a* равностороннего треугольника его периметр $P = 3a$ и площадь $S = 3/4a^2$.
4) Описать функцию RingS, находящую площадь кольца, заключённого между двумя окружностями с общим центром и радиусами *R* и R_2.
5) Описать функцию RectPS, вычисляющую периметр *P* и площадь *S* прямоугольника со сторонами, параллельными осям координат, по координатам (x_1, y_1), (x_2, y_2) его противоположных вершин.
6) Описать функцию TriangleP, находящую периметр равнобедренного треугольника по его основанию *a* и высоте *h*, проведенной к основанию. Для нахождения боковой стороны *b* треугольника использовать *теорему Пифагора*: $b^2 = (a/2)^2 + h^2$.
7) Описать функцию InvertDigits, меняющую порядок следования цифр целого положительного числа *K* на обратный.
8) Описать функцию SumRange, находящую сумму всех целых чисел от *A* до *B* включительно (*A* и *B* — целые). Если *A*>*B*, то функция возвращает 0.
9) Описать функцию TypeTrl, которая позволяет по заданным координатами вершин треугольника определить вид треугольника (равнобедренный, равносторонний, прямоугольный или обычный).
10) Описать функцию Quarter, определяющую номер координатной четверти, в которой находится точка с ненулевыми вещественными координатами (*x*, *y*).
11) Описать функцию DigitCountSum, находящую количество цифр целого положительного числа *K* , а также их сумму *S* .
12) Описать функцию Calc, выполняющую над ненулевыми вещественными числами *A* и *B* одну из арифметических операций и возвращающую её результат. Вид операции определяется целым параметром *Op*: 1 — вычитание, 2 — умножение, 3 — деление, остальные значения — сложение.
13) Описать функцию LuckyNum, которая позволяет определить, является ли четырёх-

значный номер счастливым. Счастливым называется номер, у которого сумма первых двух цифр номера равна сумме последних двух цифр. Получить все четырёхзначные счастливые номера.

14) Описать функцию DegToRad, находящую величину угла в радианах, если дана его величина D в градусах ($0 < D < 360$). Воспользоваться следующим соотношением: $180° = \pi$ радианов.
15) Описать функцию IsLeapYear, определяющую, является ли год Y високосным. *Високосным* считается год, делящийся на 4, за исключением тех годов, которые делятся на 100 и не делятся на 400.
16) Описать функцию SphereVS, вычисляющую объем V и площадь поверхности S сферы радиусом r.
17) Описать функцию CylinderVS, находящую объем V и площадь поверхности S цилиндра с радиусом основания r и высотой h.
18) Описать функцию ConeVS, вычисляющую объем V и площадь поверхности S конуса с радиусом основания r и высотой h.
19) Описать функцию RectangularBoxVS, находящую объем V и площадь поверхности S прямоугольного параллелепипеда со сторонами a, b и c.
20) Описать функцию EllipseS, вычисляющую площадь эллипса с полуосями a и b.
21) Описать функцию TrapezoidPS, находящую периметр P и площадь S трапеции с основаниями a и b, и высотой h.
22) Описать функцию RhombusS, вычисляющую площадь ромба с диагоналями d_1 и d_2.
23) Описать функцию ParallelogramPS, находящую периметр P и площадь S параллелограмма со сторонами a и b, и углом между ними α.
24) Описать функцию PrimeNumbers, которая выводит все простые числа, меньшие или равные заданному числу N.
25) Описать функцию BinaryToDecimal, которая преобразует заданное двоичное число в десятичное число.
26) Описать функцию PerfectNumber, определяющую, является ли заданное положительное число совершенным числом. Совершенным числом называется число, равное сумме своих делителей (кроме самого числа).
27) Описать функцию GCD, вычисляющую наибольший общий делитель двух заданных чисел.
28) Описать функцию MinMax, находящую минимальное и максимальное значение из заданного списка чисел.

29) Описать функцию PrimeFactors, которая находит все простые множители заданного числа N и выводит их.
30) Описать функцию ArmstrongNumber, определяющую, является ли заданное положительное число числом Армстронга. Число Армстронга — это число, которое равно сумме своих цифр, возведенных в степень, равную количеству цифр в числе.

Задача 3.2. Функции обработки матриц.

1) Описать функцию RemoveRowCol, удаляющую из матрицы A размера $M \times N$ строку и столбец, которые содержат элемент $A_{K, L}$ (предполагается, что $M > 1$ и $N > 1$; если $K > M$ или $L > N$, то матрица не изменяется).
2) Описать функцию Split, формирующую по заданному списку A два новых списка B и C; при этом список B содержит все чётные числа из списка A, а список C — все нечётные числа (в том же порядке).
3) Описать функцию SwapRow, осуществляющую перемену местами строк вещественной матрицы A размера $M \times N$ с номерами K_1 и K_2.
4) Описать функцию Transp, выполняющую транспонирование (то есть зеркальное отражение относительно главной диагонали) квадратной матрицы A порядка M.
5) Описать функцию MtrProd, вычисляющую произведение двух матриц.
6) Описать функцию SumMtr, вычисляющую сумму двух матриц.
7) Дана квадратная матрица. Вычислить сумму элементов главной или побочной диагонали в зависимости от выбора пользователя. Сумма элементов любой диагонали должна вычисляться в одной и той же функции.
8) Описать функцию RemoveRows, удаляющую из матрицы A размера $M \times N$ строки с номерами от K_1 до K_2 включительно (предполагается, что $1 < K_1 \le K_2$). Если $K_1 < M$, то матрица не изменяется; если $K_2 > M$, то удаляются строки матрицы с номерами от K_1 до M.
9) Описать функцию InvertStr, возвращающую инвертированную подстроку строки S, содержащую в обратном порядке N символов строки S, начиная с её K-го символа. Если K превосходит длину строки S, то возвращается пустая строка; если длина строки меньше $K + N$, то инвертируются все символы строки, начиная с её K-го символа.
10) Описать функцию SwapCol, осуществляющую перемену местами столбцов матрицы A размера $M \times N$ с номерами K_1 и K_2.

11) Описать функцию CompressStr, выполняющую сжатие строки S по следующему правилу: каждая подстрока строки S , состоящая из более чем четырёх одинаковых символов C, заменяется текстом вида «$C\{K\}$», где K — количество символов C (предполагается, что строка S не содержит фигурных скобок «{» и «}»). Например, для строки S = *bbbccccce* функция вернет строку *bbbc*{5}*e*.
12) Описать функцию IsIdent, проверяющую, является ли строка S допустимым идентификатором, то есть непустой строкой, которая содержит только латинские буквы, цифры и символ подчеркивания «_» и не начинается с цифры.
13) Описать функцию Chessboard, формирующую по целым положительным числам M и N матрицу A размера $M \times N$, которая содержит числа 0 и 1, расположенные в «шахматном» порядке, причём $A_1 = 0$.
14) Описать функцию Bell, меняющую порядок элементов списка L на следующий: наименьший элемент списка располагается на первом месте, наименьший из оставшихся элементов — на последнем, следующий по величине располагается на втором месте, следующий — на предпоследнем и т. д. (в результате график значений элементов будет напоминать колокол).
15) Описать функцию RemoveCols, удаляющую из матрицы A размера $M \times N$ столбцы с номерами от K_1 до K_2 включительно (предполагается, что $1 < K_1 \le K_2$). Если $K_1 >$ N, то матрица не изменяется; если $K_2 > N$, то удаляются столбцы матрицы с номерами от K_1 до N .
16) Описать функцию SplitEvenOdd, которая разделяет список чисел на два списка: список четных чисел и список нечетных чисел.
17) Описать функцию ReverseTuples, которая принимает список кортежей и возвращает новый список, где каждый кортеж перевернут в обратном порядке.
18) Описать функцию ReplaceDuplicates, которая заменяет все дублирующиеся элементы в списке на заданное значение.
19) Описать функцию SquaredSum, которая вычисляет сумму квадратов элементов списка.
20) Описать функцию UniqueIntersection, которая принимает два списка и возвращает список уникальных общих элементов обоих списков.
21) Описать функцию SplitByCondition, которая разделяет список на два списка: один содержит элементы, удовлетворяющие заданному условию, а второй содержит остальные элементы.

22) Описать функцию RemoveDuplicatesList, которая удаляет все повторяющиеся элементы из списка.
23) Описать функцию TransposeMatrix, которая принимает матрицу в виде списка списков и возвращает ее транспонированную версию.
24) Описать функцию MergeDicts, которая принимает произвольное количество словарей и объединяет их в один словарь.
25) Описать функцию CountSublists, которая принимает список и возвращает количество вложенных списков внутри него.
26) Описать функцию SwapElements, которая принимает список и индексы двух элементов и меняет их местами.
27) Описать функцию CountOccurrences, которая подсчитывает количество вхождений каждого элемента в списке и возвращает словарь, где ключи — элементы списка, а значения — количество их вхождений.
28) Описать функцию SortNestedList, которая принимает список вложенных списков и сортирует его по возрастанию суммы элементов в каждом вложенном списке.
29) Описать функцию FlattenList, которая принимает вложенный список и возвращает одноуровневый список, содержащий все элементы из вложенных списков.
30) Описать функцию RemoveEmptyStrings, которая удаляет пустые строки из списка строк.

Задача 3.3. Рекурсивные функции.

1) Описать рекурсивные функции Fact и Fact2, вычисляющие значения факториала *N*! и двойного факториала *N*!! соответственно (*N* > 0 — параметр целого типа).
2) Описать рекурсивную функцию PowerN, находящую значение *n*-й степени числа *x* по формуле:

$$x^0 = 1, x^n = x \cdot x^{n-1} \text{ при } n > 0, \qquad x^n = \frac{1}{x^{-n}} \text{ при } n < 0$$

$$(x \geq 0 - \text{вещественное число},\ n\ - \text{целое}).$$

3) Описать рекурсивную функцию SqrtK(*x*, *k*, *n*), находящую приближенное значение корня *k* -й степени из числа *x* по формуле:

$$y(0) = 1, \qquad y(n+1) = y(n) - (y(n) - \frac{x}{y^{k-1}(n)})/k,$$

где *y*(*n*) обозначает SqrtK(*x*, *k*, *n*) (*x* — вещественный параметр, *k* и *n* — целые; x > 0, *k* > 1, *n* > 0).

4) Описать рекурсивную функцию FibRec, вычисляющую N-е число Фибоначчи $F(N)$ по формуле:

$$F(1) = F(2) - 1,\ F(k) = F(k-2) + F(k-1),\ k = 3,4,\ldots$$

С помощью этой функции найти пять чисел Фибоначчи с указанными номерами и вывести эти числа вместе с количеством рекурсивных вызовов функции FibRec, потребовавшихся для их нахождения.

5) Описать рекурсивную функцию $C(m, n)$, находящую число сочетаний из n элементов по m, используя формулу:

$$C(0, n) = C(n, n) = 1,\ C(m,\ n) = C(m,\ n-1) + C(m-1, n-1)$$

при $0 < m < n$ (m и n — целые параметры; $n > 0$, $0 \le m \le n$).

Дано число N и пять различных значений M. Вывести числа $C(M, N)$ вместе с количеством рекурсивных вызовов функции C, потребовавшихся для их нахождения.

6) Описать рекурсивную функцию NOD(A, B), находящую наибольший общий делитель двух натуральных чисел A и B, используя алгоритм Евклида:

$$\mathrm{NOD}(A, B) = \mathrm{NOD}(B \bmod A, A), \text{ если } A \neq 0,$$

$$\mathrm{NOD}(0, B) = B.$$

7) С помощью этой функции найти наибольшие общие делители пар A и B, A и C, A и D , если даны числа A, B, C, D .

8) Описать рекурсивную функцию MinRec, которая находит минимальный элемент массива A размера N, *не используя оператор цикла*. С помощью этой функции найти минимальные элементы трёх заданных массивов.

9) Описать рекурсивную функцию Digits, находящую количество цифр в строке S *без использования оператора цикла*. С помощью этой функции найти количество цифр в заданной строке.

10) Описать рекурсивную функцию Simm, проверяющую, является ли симметричной строка S , *без использования оператора цикла*. С помощью этой функции проверить заданную строку.

11) Задано положительное и отрицательное число в двоичной системе. Составить программу вычисления суммы этих чисел, используя функцию сложения чисел в двоичной системе счисления.

12) Описать рекурсивную функцию Root, которая методом деления отрезка пополам находит с точностью ε корень уравнения $f(x) = 0$ на отрезке $[a, b]$ (считать, что $\varepsilon > 0$, $a < b$, $f(a) - f(b) < 0$ и $f(x)$ — непрерывная и монотонная на отрезке $[a, b]$ функция).

13) Дано число *n*, десятичная запись которого не содержит нулей. Получите число, записанное теми же цифрами, но в противоположном порядке. При решении этой задачи нельзя использовать циклы, строки, списки разрешается только рекурсия и целочисленная арифметика. Функция должна возвращать целое число, являющееся результатом работы программы, выводить число по одной цифре нельзя.
14) Описать рекурсивную логическую функцию Simm, проверяющую, является ли симметричной часть строки S , начинающаяся i-м и заканчивающаяся j-м её элементами.
15) Дано натуральное число N. Вычислите сумму его цифр. При решении этой задачи нельзя использовать строки, списки и циклы.
16) Дано слово, состоящее только из строчных латинских букв. Проверьте, является ли это слово палиндромом. Выведите *yes* или *no*. При решении этой задачи нельзя пользоваться циклами, также нельзя использовать срезы с шагом, отличным от 1.
17) Описать рекурсивную функцию BinarySearch, которая находит индекс заданного элемента в отсортированном массиве.
18) Описать рекурсивную функцию GCD, находящую наибольший общий делитель двух чисел A и B с помощью алгоритма Евклида.
19) Описать рекурсивную функцию Palindrome, которая проверяет, является ли заданная строка палиндромом.
20) Описать рекурсивную функцию SumDigits, вычисляющую сумму цифр числа.
21) Описать рекурсивную функцию BinaryToDecimal, которая переводит число из двоичной системы счисления в десятичную.
22) Описать рекурсивную функцию CountVowels, находящую количество гласных букв в строке S без использования оператора цикла. С помощью этой функции найти количество гласных букв в заданной строке.
23) Описать рекурсивную функцию CountConsonants, находящую количество согласных букв в строке S без использования оператора цикла. С помощью этой функции найти количество согласных букв в заданной строке.
24) Описать рекурсивную функцию CountSpaces, находящую количество пробелов в строке S без использования оператора цикла. С помощью этой функции найти количество пробелов в заданной строке.
25) Описать рекурсивную функцию CountUppercase, находящую количество прописных букв в строке S без использования оператора цикла. С помощью этой функции найти количество прописных букв в заданной строке.
26) Описать рекурсивную функцию CountLowercase, находящую количество строчных

букв в строке *S* без использования оператора цикла. С помощью этой функции найти количество строчных букв в заданной строке.

27) Описать рекурсивную функцию CountDigits, находящую количество цифр в строке *S* без использования оператора цикла. С помощью этой функции найти количество цифр в заданной строке.
28) Описать рекурсивную функцию CountSymbols, находящую количество определенного символа в строке *S* без использования оператора цикла. С помощью этой функции найти количество определенного символа в заданной строке.
29) Описать рекурсивную функцию IsSubstringInOrder, которая проверяет, является ли одна строка подстрокой другой строки в определенной последовательности символов.
30) Описать рекурсивную функцию IsAlternating, которая проверяет, является ли заданная строка последовательностями символов, чередующихся по типу (буква-цифра-буква-цифра и т.д.).
31) Описать рекурсивную функцию CountOccurrence, находящую количество вхождений подстроки в строке *S* без использования оператора цикла. С помощью этой функции найти количество вхождений подстроки в заданной строке.

4. Файлы. Работа с файлами

4.1. Работа с текстовыми файлами

Работа с текстовыми файлами в Python позволяет считывать данные из файлов, записывать данные в файлы и выполнять другие операции с файлами. В Python работа с текстовыми файлами осуществляется с использованием функций модуля io или операторов with и open. Вот основные операции, которые можно выполнить с текстовыми файлами в Python:

1. Открытие файла:

Для работы с текстовым файлом сначала нужно открыть его с помощью функции open(). Функция принимает имя файла и режим доступа в качестве параметров. Режимы доступа могут быть "r" (чтение), "w" (запись) или "a" (добавление). Например:

```
file = open("file.txt", "r")
```

2. Чтение данных из файла:

После открытия файла вы можете считать его содержимое с помощью метода read() или построчно с помощью метода readline(). Например:

```
file = open("file.txt", "r")

# считать все содержимое файла
data = file.read()

# считать одну строку
line = file.readline()
```

Для чтения текстового файла используется функция open с аргументом "r" (read), а для записи или создания нового файла — функция open с аргументом "w" (write) или "a" (append) для дописывания данных в конец файла.

Пример чтения текстового файла:

```
with open("file.txt", "r") as file:
    content = file.read()
    print(content)
```

В этом примере мы создаем файловый объект file, указывая имя файла "file.txt" и режим чтения "r". Далее, при помощи метода read(), мы считываем содержимое файла в переменную content и выводим ее на экран.

3.Запись данных в файл:

Пример записи в текстовый файл:

```
with open("file.txt", "w") as file:
    file.write("Hello, World!")
```

В этом примере мы создаем или открываем файл "file.txt" в режиме записи "w". При помощи метода write() мы записываем строку "Hello, World!" в файл.

Пример дописывания данных в конец файла:

```
with open("file.txt", "a") as file:
    file.write("\nThis is a new line.")
```

В этом примере мы открываем файл "file.txt" в режиме дописывания "a". Мы записываем строку "\nThis is a new line." в файл, добавляя новую строку с помощью символа перевода строки \n.

При работе с текстовыми файлами можно также использовать другие методы, например:

- readline() — для чтения отдельных строк из файла;
- readlines() — для чтения всех строк из файла и возвращения их в виде списка.

Также важно помнить о закрытии файла после завершения работы с ним. В примерах выше оператор with автоматически осуществляет закрытие файла, но при использовании функции open без оператора with следует вызвать метод close() для закрытия файла.

4.2. Работа с файлами CSV

Файлы формата CSV (Comma-Separated Values) являются текстовыми файлами, в которых значения разделяются запятыми. Работа с файлами CSV в Python осуществляется с помощью встроенной библиотеки csv. Вот основные операции, которые можно выполнить при работе с файлами CSV в Python:

1. Чтение данных из файла CSV:

Для чтения данных из файла CSV в Python используйте класс csv.reader. Он позволяет считывать данные построчно или как список значений. Вот пример чтения данных из файла:

```
import csv
```

```
with open("data.csv", "r") as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        # распечатать каждую строку как список значений
        print(row)
```

2. Запись данных в файл CSV:

Чтобы записать данные в файл CSV, вам нужно открыть файл в режиме записи ("w" или "a") и использовать класс csv.writer. Он позволяет записывать данные строка за строкой или по списку значений. Вот пример записи данных в файл:

```
import csv
```

```
with open("data.csv", "w") as file:
    csv_writer = csv.writer(file)

    # записать заголовки
    csv_writer.writerow(["Name", "Age"])

    # записать строку данных
    csv_writer.writerow(["Alice", 25])
```

3. Работа с разделителями и кавычками:

В файле CSV могут использоваться различные разделители и символы кавычек. По умолчанию, в Python разделителем является запятая и кавычками — двойные кавычки. Однако, вы можете указать другие разделители и кавычки при необходимости, передав их в соответствующие аргументы функции csv.reader или csv.writer.

4. Работа с заголовками:

Если файл CSV содержит заголовки, вы можете использовать методы next() или __next__() для пропуска первой строки. Это позволит вам начать чтение или запись данных с первой фактической строки.

5. Обработка CSV-данных:

Считанные или записанные данные CSV обычно представляются в виде списков значений для каждой строки. Вы можете использовать индексирование или итерацию для обработки отдельных значений или строк в файле CSV.

6. Дополнительные возможности:

Библиотека csv предоставляет и другие полезные функции, такие как установка разделителя десятичной запятой, обработка NULL-значений, пропуск строк без данных и т.д. Подробности можно найти в документации Python.

При работе с файлами CSV важно учитывать особенности форматирования данных, такие как кавычки и экранирование символов, чтобы избежать проблем при чтении или записи.

4.3. Работа с модулем pickle

Модуль pickle в Python предоставляет возможность сериализации (преобразования объектов Python в байтовую последовательность) и десериализации (восстановления объектов Python из байтовой последовательности) данных. Это позволяет сохранять сложные структуры данных, такие как списки, словари, классы и т.д., в файлы или передавать их по сети. Чтобы использовать модуль pickle, нужно импортировать его следующим образом: import pickle.

Сериализация объекта происходит при помощи функции pickle.dump(), которая принимает два аргумента: объект, который нужно сериализовать, и файловый объект, в который нужно записать сериализованные данные. Например:

```
import pickle
```

```
data = [1, 2, 3, 4, 5]
with open("data.pkl", "wb") as file:
    pickle.dump(data, file)
```

В этом примере мы создаем список data, который мы хотим сериализовать. Затем мы открываем файл "data.pkl" в режиме записи байтов ("wb") и с помощью функции pickle.dump() записываем сериализованные данные в файл.

Для десериализации данных из файла используется функция pickle.load(), которая принимает файловый объект и возвращает восстановленный объект. Например:

```
import pickle
```

```
with open("data.pkl", "rb") as file:
    loaded_data = pickle.load(file)
    print(loaded_data)
```

В этом примере мы открываем файл "data.pkl" в режиме чтения байтов ("rb"). Затем мы используем функцию pickle.load(), чтобы загрузить данные из файла в переменную loaded_data и выводим ее значение на экран.

Модуль pickle также предоставляет другие функции, такие как pickle.dumps() (сериализация в строку), pickle.loads() (десериализация из строки), pickle.dump() и pickle.load() для работы с сокетами.

Важно отметить, что модуль pickle может использоваться только в среде, где нет доверия небезопасным данных. При десериализации объектов из ненадежных источников может быть выполнен вредоносный код. Поэтому, будьте осторожны при использовании pickle с данными, полученными из ненадежных источников.

4.4. Задачи для самостоятельного решения

Задача 4.1. Файлы.

1) Дано целое число K ($0 < K < 10$) и текстовый файл, содержащий более K строк. Создать новый текстовый файл, содержащий K последних строк исходного файла.
2) Дано целое число K ($0 < K < 10$) и текстовый файл, содержащий более K строк. Удалить из файла последние K строк.
3) Дано целое число K и текстовый файл. Удалить из каждой строки файла первые K символов (если длина строки меньше K, то удалить из нее все символы).
4) Даны два текстовых файла. Добавить в конец каждой строки первого файла соответствующую строку второго файла. Если второй файл короче первого, то оставшиеся строки первого файла не изменять.
5) Дано целое число K и текстовый файл. Удалить из файла строку с номером K. Если строки с таким номером нет, то оставить файл без изменений.
6) Дан текстовый файл. Создать символьный файл, содержащий все знаки препинания, встретившиеся в текстовом файле (в том же порядке).
7) Дан текстовый файл, каждая строка которого изображает целое число, дополненное слева и справа несколькими пробелами. Вывести количество этих чисел и их сумму.
8) Дана строка S и текстовый файл. Заменить в файле все пустые строки на строку S.

9) Дан текстовый файл, содержащий текст, выровненный по левому краю. Выровнять текст по центру, добавив в начало каждой непустой строки нужное количество пробелов (ширину текста считать равной 50). Строки нечетной длины перед центрированием дополнять слева пробелом.
10) Дан текстовый файл. Найти количество абзацев в тексте, если абзацы отделяются друг от друга одной или несколькими пустыми строками.
11) Дан текстовый файл, содержащий текст, выровненный по левому краю. Выровнять текст по правому краю, добавив в начало каждой непустой строки нужное количество пробелов (ширину текста считать равной 50).
12) Даны два текстовых файла. Добавить в начало первого файла содержимое второго файла.
13) Дано целое число *K* и текстовый файл. Вставить пустую строку перед строкой файла с номером *K*. Если строки с таким номером нет, то оставить файл без изменений.
14) Дан текстовый файл. Продублировать в нем все пустые строки.
15) Дан непустой текстовый файл (число строк > 2). Удалить из него первую и последнюю строки.

Задача 4.2. Файлы. Обработка данных.

1) Дан файл *f*, компоненты которого являются действительными числами. Найти произведение компонент файла.
2) Дан файл *f*, компоненты которого являются целыми числами. Ни одна из компонент файла не равна нулю. Файл *f* содержит столько же отрицательных чисел, сколько и положительных. Переписать компоненты файла *f* в файл *g* так, чтобы в файле *g* сначала шли положительные, потом отрицательные числа.
3) Дан файл *f*, компоненты которого являются целыми числами. Получить в файле *g* все компоненты файла *f*, являющиеся точными квадратами.
4) Дан файл *f*, компоненты которого являются действительными числами. Найти сумму наибольшего и наименьшего из значений компонент.
5) Дан файл, содержащий различные даты. Каждая дата — это число, месяц и год. Найти год с наименьшим номером.
6) Дан файл *f*, компоненты которого являются целыми числами. Найти количество чётных чисел среди компонент.
7) Дан файл *f*. В файле не менее двух компонент. Определить, являются ли два первых символа файла цифрами. Если да, то установить, является ли число, образованное этими цифрами чётным.

8) Дан файл *f*, компоненты которого являются целыми числами. Получить в файле *g* все компоненты файла *f*, являющиеся чётными числами.
9) Дан файл *f*, компоненты которого являются действительными числами. Найти наибольшее из значений модулей компонент с нечётными номерами.
10) Дан файл, содержащий различные даты. Каждая дата — это число, месяц и год. Найти все весенние даты.
11) Дан файл *f*, компоненты которого являются целыми числами. Получить в файле *g* все компоненты файла *f*, делящиеся на 3 и не делящиеся на 7.
12) Дан файл *f*, компоненты которого являются действительными числами. Найти наименьшее из значений компонент с чётными номерами.
13) Записать в файл *g* все чётные числа файла *f*, а в файл *h* все нечётные. Порядок следования чисел сохраняется.
14) Дан файл *f*, компоненты которого являются целыми числами. Ни одна из компонент файла не равна нулю. Файл *f* содержит столько же отрицательных чисел, сколько и положительных. Переписать компоненты файла *f* в файл *g* так, чтобы в файле *g* не было двух соседних чисел с одним знаком.
15) Дан файл *f*, компоненты которого являются целыми числами. Ни одна из компонент файла не равна нулю. Файл *f* содержит столько же отрицательных чисел, сколько и положительных. Переписать компоненты файла *f* в файл *g* так, чтобы в файле *g* числа шли в следующем порядке: два положительных два отрицательных, два положительных, два отрицательных и т.д. (предполагается, что число компонент в файле *f* делится на 4).
16) Дан файл *f*, содержащий строки с именами студентов и их оценками. Найти средний балл каждого студента и записать его в новый файл *g*.
17) Дан файл *f*, содержащий информацию о продажах товаров. Найти суммарную выручку за каждый месяц и записать ее в новый файл *g*.
18) Дан файл *f*, содержащий строки с текстом. Найти количество слов в каждой строке и записать результат в новый файл *g*.
19) Дан файл *f*, содержащий список сотрудников компании и их заработные платы. Найти среднюю зарплату всех сотрудников и записать ее в новый файл *g*.
20) Дан файл *f*, содержащий список дат рождения людей. Найти самую старшую и самую младшую даты рождения и записать их в новый файл *g*.
21) Дан файл *f*, содержащий строки с информацией о товарах в интернет-магазине. Найти товар с самой высокой стоимостью и записать его название в новый файл *g*.

22) Дан файл *f*, содержащий координаты точек на плоскости. Найти точку с наибольшим расстоянием от начала координат и записать ее координаты в новый файл *g*.
23) Дан файл *f*, содержащий список автомобилей и их стоимость. Найти автомобиль с наименьшей стоимостью и записать его название в новый файл *g*.
24) Дан файл *f*, содержащий строки с информацией о товарах на складе. Найти товар с наибольшим количеством и записать его название в новый файл *g*.
25) Дан файл *f*, содержащий список городов и их население. Найти город с наибольшим населением и записать его название в новый файл *g*.
26) Дан файл *f*, содержащий строки с информацией о книгах в библиотеке. Найти книгу с самым длинным названием и записать его название в новый файл *g*.
27) Дан файл *f*, содержащий координаты вершин треугольников. Найти треугольник с наибольшей площадью и записать его координаты в новый файл *g*.
28) Дан файл *f*, содержащий строки с информацией о товарах в магазине. Найти товар с наименьшим количеством и записать его название в новый файл *g*.
29) Дан файл *f*, содержащий список студентов и их средние оценки. Найти студента с самой высокой средней оценкой и записать его имя в новый файл *g*.
30) Дан файл *f*, содержащий строки с данными о погоде. Найти дату с самой высокой температурой и записать ее в новый файл *g*.

Задача 4.3. Файлы. Обработка матриц.

1) В первом файле хранится k матриц размерности $m \times n$, во втором — l матриц размерности $m \times n$. Те матрицы из первого файла, у которых $a_{00} = 0$, перенести в конец второго файла. Вывести на экран содержимое обоих файлов.
2) В первом файле хранится k матриц размерности $m \times n$, во втором — l матриц размерности $m \times n$. Убрать из файла, в котором больше матриц, лишние матрицы в третий файл. Вывести на экран содержимое всех трёх файлов.
3) Файл состоит из k компонент структуры, где каждая компонента содержит две матрицы: первая размерности $m \times n$, вторая размерности $m \times l$. Получить k произведений соответствующих матриц и записать их во второй файл. Вывести на экран содержимое обоих файлов.
4) В первом файле хранится k матриц размерности $m \times n$, во втором — l матриц размерности $m \times n$. Добавить во второй файл те матрицы из первого, которых нет во втором. Вывести на экран содержимое обоих файлов.
5) В файле хранится k матриц размерности $m \times n$. Для каждой матрицы из файла вычислить сумму её положительных четных элементов. Все матрицы с чётными сум-

мами записать в другой файл, заменив их в исходном файле единичными матрицами. Вывести на экран содержимое обоих файлов.

6) В первом файле хранится k матриц размерности $m \times n$, во втором — l матриц размерности $m \times n$. Поменять местами все нечётные (по порядковому номеру в файле) матрицы из первого и второго файлов (до конца меньшего из файлов). Вывести на экран содержимое обоих файлов.
7) В первом файле хранится k квадратных матриц порядка n, во втором — l квадратных матриц. Если $k \neq 1$, то в файл с меньшим числом матриц добавить в конец файла недостающее количество единичных матриц. Вывести на экран содержимое обоих файлов.
8) В файле хранится k матриц размерности $n \times n$. Для каждой матрицы из файла вычислить сумму её диагональных элементов. Все матрицы с нечётными суммами записать в другой файл, заменив их в исходном файле транспонированными матрицами. Вывести на экран содержимое обоих файлов.
9) В первом файле хранится k квадратных матриц. Записать в другой файл из исходного файла все симметрические матрицы ($A = A^T$), в третий файл — остальные. Вывести на экран содержимое всех трёх файлов.
10) В первом файле хранится k матриц размерности $m \times n$, во втором — k матриц размерности $n \times l$. Получить k произведений соответствующих матриц из первого и второго файлов и записать их в третий файл в виде компонент структуры, где каждая компонента содержит три матрицы: а) первая размерности $m \times n$ из первого файла; б) вторая размерности $n \times l$ из второго файла; в) третья, матрица размерности $m \times l$, результат произведения. Вывести на экран содержимое всех файлов.
11) В первом файле хранится k матриц порядка $m \times n$, во втором — l матриц. Поменять местами все нечётные (1, 3, 5,… по порядковому номеру в файле) матрицы из первого файла с чётными матрицами (0, 2, 4,…) второго файла (до конца меньшего из файлов). Оставшиеся в большем файле матрицы переписать в третий файл. Вывести на экран содержимое всех файлов.
12) Файл состоит из k компонент структуры, где каждая компонента содержит две матрицы: первая размерности $m \times n$, вторая размерности $m \times n$. Получить k сумму соответствующих матриц и записать их во второй файл. Вывести на экран содержимое обоих файлов.

13) В файле хранится k матриц размерности $m \times n$. Для каждой матрицы из файла вычислить сумму её отрицательных нечётных элементов. Все матрицы с нечётными суммами записать в другой файл, заменив их в исходном файле единичными матрицами. Вывести на экран содержимое обоих файлов.
14) В первом файле хранится k матриц размерности $m \times n$, во втором — l матриц размерности $m \times n$. Те матрицы из первого файла, у которых сумма первой строки больше заданного числа перенести в конец второго файла. Вывести на экран содержимое обоих файлов.
15) В файле хранится k матриц размерности $n \times n$. Для каждой матрицы из файла вычислить произведение элементов её главной диагонали. Все матрицы, у которых произведение больше заданного числа, записать в другой файл. Вывести на экран содержимое обоих файлов.
16) В первом файле хранятся данные о покупках пользователя в виде пар «название товара: цена». Найти суммарную стоимость всех покупок и записать ее в новый файл.
17) В первом файле хранится информация о студентах в формате «ФИО: возраст: средняя оценка». Во втором файле хранится информация о курсе, на котором учится каждый студент в формате «ФИО: номер курса». Объединить информацию из обоих файлов в третий файл, добавив информацию о курсе для каждого студента.
18) В первом файле хранятся результаты тестов по различным предметам в формате «название предмета: оценка». Найти среднюю оценку по каждому предмету и записать результаты в новый файл.
19) В первом файле хранится информация о продуктах в формате «название продукта: количество: цена». Найти общую стоимость каждого продукта (умножение цены на количество) и записать результаты в новый файл.
20) В первом файле хранятся результаты игроков в компьютерной игре в виде «имя игрока: количество очков». Найти игрока с наибольшим количеством очков и записать его имя в новый файл.
21) В первом файле хранятся результаты соревнования по бегу в виде «имя спортсмена: время». Найти спортсмена с наименьшим временем и записать его имя в новый файл.
22) В первом файле хранится информация о сотрудниках компании в формате «ФИО: должность: заработная плата». Найти сотрудника с самой высокой заработной платой и записать его ФИО в новый файл.

23) В первом файле хранится информация о студентах в формате «ФИО: группа: средний балл». Найти студента с наивысшим средним баллом и записать его ФИО и группу в новый файл.
24) В первом файле хранятся результаты опроса в виде «вопрос: ответ». Найти вопрос, на который большинство опрошенных ответило положительно, и записать его в новый файл.
25) В первом файле хранятся данные о транзакциях на банковском счете в формате «дата: сумма: описание». Найти суммарную сумму всех транзакций и записать ее в новый файл.
26) В первом файле хранятся результаты соревнования по шахматам в виде «имя игрока: количество побед: количество поражений». Найти игрока с наибольшим количеством побед и записать его имя в новый файл.
27) В первом файле хранятся данные о продажах автомобилей в формате «марка: количество продаж: общая стоимость». Найти марку автомобиля с наибольшей общей стоимостью продаж и записать ее в новый файл.
28) В первом файле хранится информация о работниках компании в формате «ФИО: должность: зарплата». Найти среднюю зарплату по каждой должности и записать результаты в новый файл.
29) В первом файле хранятся результаты лотереи в виде «номер билета: выигрыш». Найти номер билета с наибольшим выигрышем и записать его в новый файл.
30) В первом файле хранится информация о студентах в формате «ФИО: группа: средняя оценка». Найти студента с самой низкой средней оценкой и записать его ФИО и группу в новый файл.

5. Исключения и их обработка

5.1. Обработка исключительных ситуаций

Исключения — это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

- синтаксические — это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем. Пример синтаксической ошибки:

```
print("Нет завершающей кавычки!)
```

```
  File "<ipython-input-131-a3916d1aed16>", line 1
    print("Нет завершающей кавычки!)
                                     ^
SyntaxError: EOL while scanning string literal
```

- логические — это ошибки в логике работы программы, которые можно выявить только по результатам работы скрипта. Как правило, интерпретатор не предупреждает о наличии ошибки. А программа будет выполняться, т. к. не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить и исправить;
- ошибки времени выполнения — это ошибки, которые возникают во время работы скрипта. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль.

```
print(10/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-132-fe01563e1bc6> in <module>
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

В языке Python исключения возбуждаются не только при ошибке, но и как уведомление о наступлении каких-либо событий. Например, метод index () возбуждает исключение ValueError, если искомый фрагмент не входит в строку:

```
"Строка".index("текст")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-133-0ed78ddd04aa> in <module>
----> 1 "Строка".index("текст")

ValueError: substring not found
```

Инструкция try...except...else...finally. Для обработки исключений предназначена инструкция try. Формат инструкции:

```
try:
    <блок, в котором перехватываются исключения>
[except [<исключение_1> [ as <объект исключения>]]:
    <блок, выполняемый при возникновении исключения>
...
[except [<исключение_N> [ as <объект исключения>]]:
    <блок, выполняемый при возникновении исключения>]]
[else: <блок, выполняемый, если исключение не возникло>]
[finally: <блок, выполняемый в любом случае>].
```

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока try. В блоке except в параметре <исключение_1> указывается класс обрабатываемого исключения.

Пример 5.1. Обработать исключение, возникающее при делении на ноль.

```
x = 0
try: # перехватываем исключения
    x = 1/0 # ошибка: деление на 0
except ZeroDivisionError: # указываем класс исключения
    print(x)

print("Обработали деление на 0")
```

```
0
Обработали деление на 0
```

Если в блоке try возникло исключение, то управление передается блоку except. В случае если исключение не соответствует указанному классу, управление передается следующему блоку except. Если ни один блок except не соответствует исключению, то исключение «всплывает» к обработчику более высокого уровня. Если исключение нигде не обрабатывается в программе, то управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке. Таким образом, в обработчике может быть несколько блоков except с разными классами исключений.

Пример 5.2. Обработать исключение, возникающее при делении на ноль с несколькими блоками except с разными классами исключений.

```
x = 0
try: # обрабатываем исключения
    try: # оложенном обработчик
        x = 1/0 # ошибка: деление на 0
    except NameError:
        print("Неопределенный идентификатор")
    except IndexError:
        print("Несуществующий индекс")
    print("Выражение после вложенного обработчика")
except ZeroDivisionError:
    print("Обработка деления на 0")

print(x)
```

```
Обработка деления на 0
0
```

Пример 5.3. Обработать исключение, возникающее при делении на ноль, в инструкции except указать сразу несколько исключений.

```
x = 0
try: # обрабатываем исключения
    x = 1/0 # ошибка: деление на 0
except (NameError, IndexError, ZeroDivisionError):
    x = 0
print(x)
```

```
0
```

Пример 5.4. Обработать исключение, возникающее при делении на ноль и получить информацию об обрабатываемом исключении можно через второй параметр в инструкции except.

```
x = 0
try: # обрабатываем исключения
    x = 1/0 # ошибка: деление на 0
except (NameError, IndexError, ZeroDivisionError) as err:
    print(err.__class__.__name__) # название класса исключения
    print(err) # текст сообщения об ошибке
```

```
ZeroDivisionError
division by zero
```

Если в инструкции except не указан класс исключения, то такой блок перехватывает все исключения. На практике следует избегать пустых инструкций except, т.к. можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой.

```
x = 0
try: # обрабатьшаем исключения
    x = 1/0 # ошибка: деление на 0
except:
    print(x) # выведет:0
```

```
0
```

Если в обработчике присутствует блок else, то инструкции внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком finally.

Пример 5.5. Обработать исключение, возникающее при делении на ноль с блоками else и finally.

```
x = 0
try: # обрабатываем исключения
    x = 1/0 # ошибка: деление на 0
except ZeroDivisionError:
    print("Деление на 0")
    x = 0
else:
    print('Блок else')
finally:
    print('Блок finally')
print(x) # выведет: 0
```

```
Деление на 0
Блок finally
0
```

При наличии исключения и отсутствии блока except инструкции внутри блока finally будут выполнены, но исключение не будет обработано. Оно продолжит «всплывание» к обработчику более высокого уровня. Если пользовательский обработчик отсутствует, то управление передается обработчику по умолчанию, который прерывает выполнение программы и выводит сообщение об ошибке.

```
x = 0
try: # обрабатьшаем исключения
    # x = 1/0 # ошибка: деление на
    x = 1/10
finally:
    print('Блок finally')
```

```
Блок finally
```

Классы встроенных исключений. Основное преимущество использования классов для обработки исключений заключается в возможности указания базового класса для перехвата всех исключений соответствующих классов-потомков. Например, для перехвата деле-

ния на ноль мы использовали класс ZeroDivisionError. Если вместо этого класса указать базовый класс ArithmeticError, то будут перехватываться исключения классов FloatingPointError, OverflowError и ZeroDivisionError.

Рассмотрим основные классы встроенных исключений:

- BaseException — является классом самого верхнего уровня;
- Exception — именно этот класс, а не BaseException, необходимо наследовать при создании пользовательских классов исключений;
- AssertionError — возбуждается инструкцией assert;
- AttributeError — попытка обращения к несуществующему атрибуту объекта;
- EOFError — возбуждается функцией input() при достижении конца файла;
- IOError — ошибка доступа к файлу;
- ImportError — невозможно подключить модуль или пакет;
- IndentationError — неправильно расставлены отступы в программе;
- IndexError — указанный индекс не существует в последовательности;
- KeyError — указанный ключ не существует в словаре;
- Keyboardlnterrupt — нажата комбинация клавиш Ctrl+C (или Ctrl+Break в некоторых операционных системах);
- NameError — попытка обращения к идентификатору до его определения;
- Stoprteration — возбуждается методом __next_() как сигнал об окончании итераций;
- SyntaxError — синтаксическая ошибка;
- TypeError — тип объекта не соответствует ожидаемому;
- UnboundLocalError — внутри функции переменной присваивается значение после обращения к одноименной глобальной переменной;
- UnicodeDecodeError — ошибка преобразования последовательности байтов в строку;
- UnicodeEncodeError — ошибка преобразования строки в последовательность байтов;
- ValueError — переданный параметр не соответствует ожидаемому значению;
- ZeroDivisionError — попытка деления на ноль.

5.2. Пользовательские исключения. Инструкция assert

Пользовательские исключения. Инструкция raise возбуждает указанное исключение. Она имеет несколько форматов:

```
raise <экземпляр класса>
raise <название класса>
```

```
raise <экземпляр или название класса> from <объект исключения>
raise
```

В первом формате инструкции raise указывается экземпляр класса возбуждаемого исключения. При создании экземпляра можно передать данные конструктору класса. Эти данные будут доступны через второй параметр в инструкции except.

```
try:
    raise ValueError("Описание исключения")
except ValueError as msg:
    print(msg) # выведет: Описание исключения
```

```
Описание исключения
```

```
try:
    raise ValueError
except ValueError:
    print('Сообщение об ошибке')
```

```
Сообщение об ошибке
```

Инструкция assert. Инструкция assert возбуждает исключение AssertionError, если логическое выражение возвращает значение False. Инструкция имеет следующий формат:

```
assert <логическое выражение> [, <сообщение>].
```

Инструкция assert эквивалентна следующему коду:

```
if __debug__:
    if not <логическое выражение>:
        raise AssertionError(<Сообщение>).
```

Если при запуске программы используется флаг -o, то переменная __debug__ будет иметь ложное значение. Таким образом, можно удалить все инструкции assert из байт-кода.

```
try:
    x = -3
    assert x >= 0, "Сообщение об ошибке"
except AssertionError as err:
    print(err) # выдает: Сообщение об ошибке
```

```
Сообщение об ошибке
```

Пример 5.6. Обойти и вывести на экран все элементы списка (например: [0, 1, 2, 3, 4]) при помощи итератора, функции next() и обработать исключительную ситуацию StopIteration.

```
m = [0, 1, 2, 3, 4, 12]
m = iter(m)
try:
    while True:
        print(next(m))
except StopIteration:
    print("Это конец")
```

```
0
1
2
3
4
12
Это конец
```

Пример 5.7. Реализовать функцию cathetus_2 , осуществляющую поиск второго катета по длине гипотенузы и первого катета с проверкой входных значений и генерацией исключений с содержательными сообщениями при получении некорректных параметров.

```
def cathetus_2(k, g):
    assert type(k) == int, "Что-то не то с катетом"
    assert type(g) == int, "Что-то не то с гипотенузой"
    assert g > k, "Что-то не то с треугольником"

    import math

    return math.sqrt(g*g - k*k)
```

```
def f(k, g):
    try:
        print(cathetus_2(k, g))
    except AssertionError as t:
        print(t)
```

```
f(3, 5)
f(5, 5)
f("1", "5")
f(1, "5")
```

```
4.0
Что-то не то с треугольником
Что-то не то с катетом
Что-то не то с гипотенузой
```

Пример 5.8. Запросить у пользователя имя файла и попытаться открыть файл с этим именем. Прочитать и вывести первую строку из этого файла. Корректно отреагировать на ошибку, связанную с отсутствием этого файла и на другие потенциальные проблемы, которые могут возникнуть во время выполнения задачи.

```
def redlion(i):
    try:
        f = open(i, mode='r')
        print(f.readline())
        f.close()
    except FileNotFoundError:
        print("404, lol")
```

```
redlion("lol.txt")
```

```
404, lol
```

5.3. Задачи для самостоятельного решения

Задача 5.1. Исключения. Пользовательские исключения. Инструкция assert.

1) Напишите функцию, которая возвращает факториал числа *n*. Проработать исключения, что аргумент должен быть целым, положительным и не равен 0.
2) Написать функцию сложения двух положительных чисел. Вызвать исключение AssertionError при вводе пользователем отрицательных чисел.
3) Напишите функцию, которая принимает имя файла. Обработать исключительную ситуацию при условии, что такого файла не существует, или неверно задан путь к файлу.
4) Написать функцию сложения чисел. В функцию может передаваться любое количество чисел. Если аргумент не является число, то вызвать исключение.
5) Написать функцию вычисления 2/*s*. На вход функции поступает переменная *s*. Перехватить все исключительные ситуации.
6) Написать функцию вычисления квадратного уравнения. На вход функции поступают коэффициенты *a*, *b*, *c*. Перехватить все исключительные ситуации.
7) Написать функцию подсчета суммы главной диагонали. Если матрица не является квадратной, должна вызываться ошибка.
8) Написать функцию подсчета суммы побочной диагонали. Если матрица не является квадратной, должна вызываться ошибка.
9) Написать функцию вычисления площади квадрата. Обработать исключительную ситуацию при условии, что стороны квадрата не равны.
10) Написать функцию вычисления периметра прямоугольника. Обработать исключительную ситуацию при условии, что стороны прямоугольника одинаковые.

11) Напишите функцию, которая принимает имя файла и дополнительно расширение файла. Если файл имеет расширение отличное от введенного, то возвращает текст «Файл имеет другое расширение», иначе возвращает содержимое файла.
12) Написать калькулятор для строковых выражений вида 'a <операция> b', где 'a' и 'b' — целые числа, а <операция> — одна из операций +, –, *, / (деление нацело), % (остаток от деления), ^ (возведение в степень). Пример calc('13 – 5') → 8. Проверять переданную строку на корректность ввода и выводить отладочную ошибку.
13) В функцию передаются три кортежа — координаты *x, y* точек. Необходимо найти площадь треугольника. Данные проверять в режиме отладки.
14) Элементами матрицы являются случайные положительные целые числа из заданного диапазона. Число строк и столбцов матрицы задается с клавиатуры. Написать функцию подсчета среднего арифметического элементов над главной диагональю и количество четных элементов под ней. Если матрица не является квадратной, должно генерироваться исключение.
15) Написать функцию сложения двух положительных чисел. Вызвать исключение AssertionError при вводе пользователем отрицательных чисел.
16) Написать функцию вычитания двух положительных чисел. Вызвать исключение AssertionError при вводе пользователем отрицательных чисел.
17) Написать функцию деления двух положительных чисел. Вызвать исключение ZeroDivisionError при делении на ноль.
18) Написать функцию деления числа на себя. Вызвать исключение ValueError при попытке деления на ноль.
19) Написать функцию, которая принимает список чисел и возвращает их сумму. Обработать исключение TypeError если в переданном списке присутствуют нечисловые значения.
20) Написать функцию, которая принимает строку и возвращает количество символов в строке. Обработать исключение TypeError если в переданном аргументе не строка.
21) Создать функцию для чтения текстового файла и возврата его содержимого. Обработать исключение FileNotFoundError если файл не найден.
22) Написать функцию для возведения числа в степень. Обработать исключение TypeError если аргументы не являются числами.
23) Написать функцию, которая принимает список и возвращает его первый элемент. Обработать исключение IndexError если список пуст.

24) Написать функцию, которая принимает две строки и возвращает их объединение. Обработать исключение TypeError если аргументы не строки.
25) Написать функцию, которая принимает словарь и ключ, и возвращает значение по данному ключу. Обработать исключение KeyError если ключа нет в словаре.
26) Написать функцию, которая принимает число и возвращает его квадратный корень. Обработать исключение ValueError если введено отрицательное число.
27) Написать функцию, которая принимает список и число, и возвращает количество раз, которое число встречается в списке. Обработать исключение ValueError если в аргументе список не передан.
28) Написать функцию, которая принимает строку и символ, и возвращает количество раз, которое символ встречается в строке. Обработать исключение TypeError если в аргументе не строка или символ не передан.
29) Написать функцию, которая принимает число и проверяет, является ли оно простым. Обработать исключение ValueError если введено отрицательное число.
30) Написать функцию, которая принимает список списков и возвращает новый список, содержащий только уникальные элементы. Обработать исключение TypeError если в аргументе передан не список списков.

6. Объектно-ориентированное программирование

В ***объектно-ориентированном программировании*** **(*ООП*)** программа разбивается на объекты — специальные структуры, состоящие из полей данных, которые характеризуют состояние объекта, и методов — подпрограмм, которые можно применять к объектам для изменения или запроса их состояния. Объекты могут быть разных видов, то есть содержать разные поля данных и методы для их обработки. Виды объектов описываются классами, которые могут быть связаны друг с другом в виде иерархической структуры, наследуя при этом от родителя к потомку те или иные поля данных и методы.

Итак, еще раз коснёмся терминологии: класс, объект, атрибут, метод. Класс описывает модель объекта, его свойства и поведение. Класс содержит набор переменных и функций. Переменные называются атрибутами. В них хранятся характеристики объекта (название, размер, цвет, количество и т.д.). Функции внутри класса называются методами. Метод определяет действие, которое объект может выполнять над самим собой или другими объектами. Объект — это экземпляр класса. Программа в ООП представляет собой процесс взаимодействия объектов.

Какие же преимущества дает ООП? Для решения несложных задач, позволяющих обойтись короткой программой, ООП не дает никаких преимуществ. Скорее наоборот, ООП может усложнить программирование, если попытаться искусственно придумывать классы и объекты там, где и без этого алгоритм решения ясен. Совсем другое дело, если приходится разрабатывать сложные проекты, особенно связанные с моделированием объектов и процессов реального мира или воображаемого мира разрабатываемой компьютерной игры, с управлением сложным технологическим оборудованием, с отображением сложных графических сцен и прочее. В таких случаях, чтобы не запутаться в логике требуемых действий, следует перед началом программирования выполнить объектно-ориентированный анализ рассматриваемой ситуации. Какие объекты участвуют в моделируемом процессе? Какие характеристики этих объектов следует учитывать? Какие действия связаны с объектами? Как объекты взаимодействуют друг с другом? Если такой анализ успешно проведен, то при разработке программ сказывается основное преимущество ООП: все данные и алгоритмы, связанные с объектами определенного класса, собраны вместе, а не разбросаны по разным частям и модулям большого программного проекта. А из этого вытекает, что

- легче понять логику большой программы;
- легче находить ошибки и отлаживать программу;

- легче модифицировать программу, внося изменения в характеристики и поведение объектов;
- легче повторно использовать код ранее разработанных классов в новом проекте, зная только методы классов и не вдаваясь в детали реализации;
- более понятна документация программы, состоящая из описания работы отдельных классов.

6.1. Принципы объектно-ориентированного программирования

Концепция объектно-ориентированного программирования содержит достаточно много терминов и понятий, ключевыми среди которых являются понятия, без понимания которых не может быть построена сама логика функционирования программы:

- ***абстракция*** — выделение основных свойств объекта и исключение несущественных деталей;
- ***инкапсуляция*** — позволяет совместно описывать данные и методы для работы с ними, открывая при этом только ту часть функций и данных, которая нужна для внешних пользователей, а остальное пряча внутри класса;
- ***наследование*** — позволяет делать производные структуры на основе базовых, тем самым давая возможность осуществлять повторное использование этих структур;
- ***полиморфизм*** — позволяет программировать на основе одинаковых по описаниям интерфейсов, имеющих различные реализации для объектов каждого класса. Полиморфизм осуществляется в объектно-ориентированных языках путём использования виртуальных методов, что является очень удобным и безопасным.

Рассмотрим каждый принцип ООП подробнее. Объектно-ориентированная парадигма рассматривает модель мира как множество объектов, взаимодействующих друг с другом. Все объекты имеют внутреннее устройство и состояние, свойства, поведение. Чтобы справиться со сложностью окружающего мира нам приходится игнорировать, упрощать некоторые особенности объектов. Такой прием называется ***абстракцией***.

Для разных задач существенные свойства могут быть совершенно разными (рис. 6.1). Например, автомобиль для водителя — это средство передвижения, характеризующееся скоростью, массой перевозимого груза, потреблением бензина. Для автослесаря автомобиль — это набор деталей и механизмов, которые имеют определенные физические размеры и параметры работоспособности. Для дизайнера автомобиль — это форма и цвет корпуса, детали интерьера, материал обивки салона и т.д. В каждом из этих случаев при-

менение абстракции дает разные модели одного и того же объекта реального мира, отражающие его различные свойства и цели моделирования.

Рисунок 6.1 — Пример принципа абстракции

Можно привести и другой пример. Представьте, что вы попросили нескольких человек описать в общих чертах, что такое телефон и как им пользоваться: пусть это будут бабушка, мама и подруга. Бабушка вспомнит про дисковые телефоны и трубки с витым проводом. Мама расскажет про радиотелефоны, у которых есть база и есть трубка, с которой можно ходить по всей квартире, а подруга начнёт описывать мобильник. Несмотря на то, что рассказы будут сильно отличаться между собой, у них будет несколько общих моментов про телефон:

- у телефона есть трубка;
- в трубку мы говорим, из трубки — слушаем;
- можно набрать номер нужного человека и позвонить ему;
- если вам позвонят по телефону, вы это услышите и примете звонок.

Получается, что если представить абстрактный телефон, то получится такое устройство с динамиком, микрофоном и средством набора номера.

Это и есть абстракция: когда описываются только самые существенные детали, которые важны для задачи. В нашем случае задача такая — понять, что такое телефон и как им пользоваться. Поэтому микрофон и динамик для этой задачи важен, а способ связи телефона с сетью — нет. Устройство набора номера важно, а то, какая мелодия играет при вызове — нет.

Итак, абстракция считается хорошей, если она отбрасывает несущественные детали и подчеркивает именно те, которые важны для решения конкретной задачи. В ООП абстракция означает, что для каждого объекта мы задаём минимальное количество методов,

полей и описаний, которые позволят нам решить задачу. Чем меньше характеристик, тем лучше абстракция, но ключевые характеристики убирать нельзя.

Для того, чтобы применить принцип абстракции в программировании, нужно представить программу в виде множества объектов каждый из которых обладает своими свойствами и поведением. Построенная таким образом модель называется объектной. А решение задачи сводится к моделированию взаимодействия объектов. На рис. 6.2 приведен пример неформального описания двух объектов. Набор атрибутов и методов зависит от того, какие задачи должна решать программа, использующие объекты.

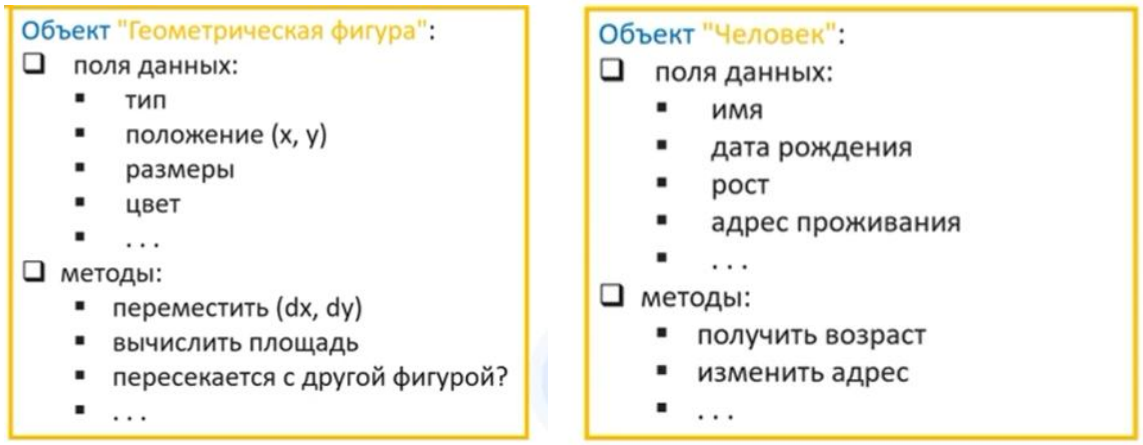


Рисунок 6.2 — Абстрактное представление объектов

Инкапсуляция (в техническом смысле ООП) — хранение алгоритмов работы с данными вместе с данными (рис. 6.3). Инкапсуляция позволяет «спрятать» атрибуты объекта.

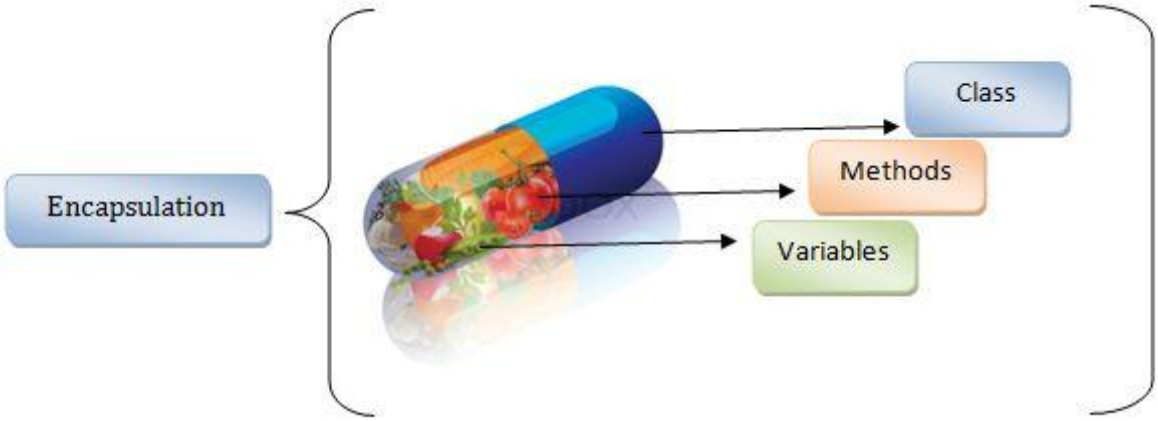


Рисунок 6.3. Пример принципа инкапсуляции

Давайте посмотрим на объект «Часы», представленный на рис. 6.4. У этого объекта есть несколько свойств, описывающих текущее время (часы и минуты) и время, в которое должен прозвенеть сигнал будильника (тоже часы и минуты). Чтобы работать с этими атрибутами у объекта есть несколько методов, назначение которых вполне понятно. Однако не все эти методы одинаковы с точки зрения взаимодействия объекта с внешним миром.

Очевидно, что ряд методов (например, установить_время(), прочитать_время()) могут вызываться из других объектов и описывают интерфейс объекта. А такие методы, как например, увеличить_минуты(), подать_сигнал() характеризуют «внутренние» принципы функционирования объекта и не должны вызываться извне.

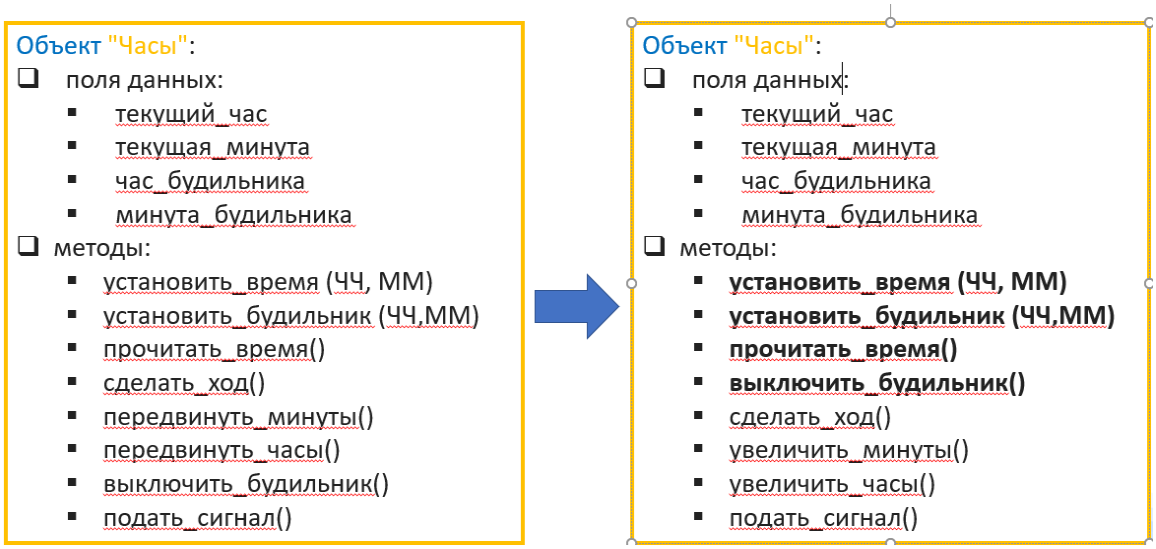


Рисунок 6.4 — Пример принципа инкапсуляции как сокрытия методов

Абстракция и инкапсуляция дополняют друг друга. В центре внимание абстракции находится наблюдаемое «снаружи» поведение объекта, а инкапсуляция сосредоточена на реализации, обеспечивающей заданное поведение

Наследование — создание специализированных классов на основе базовых (позволяет избегать написания повторного кода). На рис. 6.5 и 6.6 показан пример наследования в формальном виде и программном.

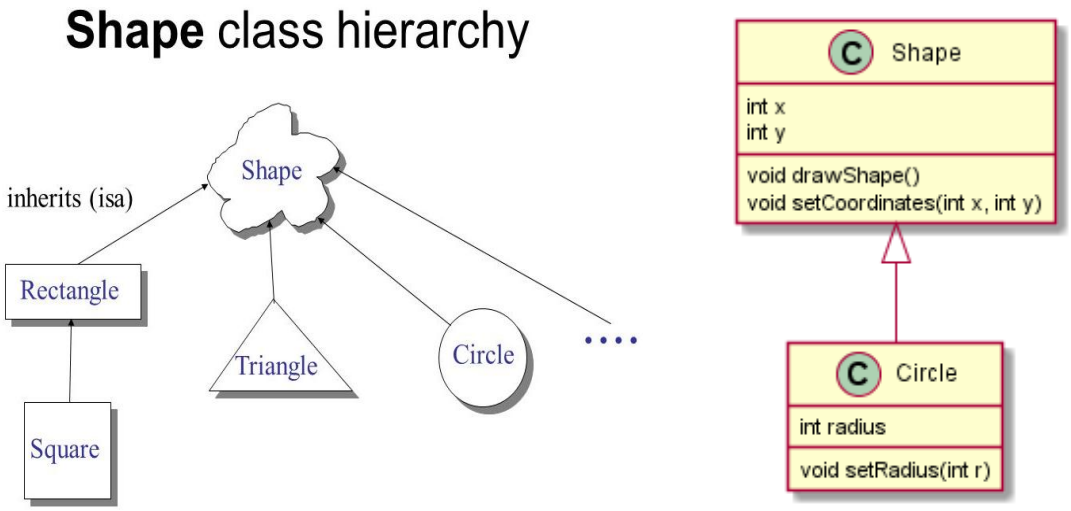


Рисунок 6.5 — Пример принципа наследования

```
class Auto:
  name = 'Auto'

class Car:
  def get_tax(self):
    print('Подсчет транспортного налога')
```

```
car = Car()
car.get_tax()
```

```
Подсчет транспортного налога
```

```
car.name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-80ccf145a0e0> in <module>
----> 1 car.name

AttributeError: 'Car' object has no attribute 'name'
```

```
class Auto:
    name = 'Auto'

class Car(Auto):
    def get_tax(self):
        print('Подсчет транспортного налога')
```

```
car = Car()
car.get_tax()
```

```
Подсчет транспортного налога
```

```
car.name
```

```
'Auto'
```

Рисунок 6.6 — Реализация принципа наследования

Полиморфизм — в разных объектах одно и то же сообщение (вызов функции) может приводить к выполнению различных реализаций функции (рис. 6.7). Полиморфизм в объектно-ориентированном программировании — это возможность обработки разных типов данных, т. е. принадлежащих к разным классам, с помощью «одной и той же» функции, или метода. На самом деле одинаковым является только имя метода, его исходный код зависит от класса. Кроме того, результаты работы одноименных методов могут существенно различаться. Поэтому в данном контексте под полиморфизмом понимается множество форм одного и того же слова — имени метода. Полиморфизм как один из ключевых элементов ООП существует независимо от наследования. Классы могут быть не родственными, но иметь одинаковые методы.

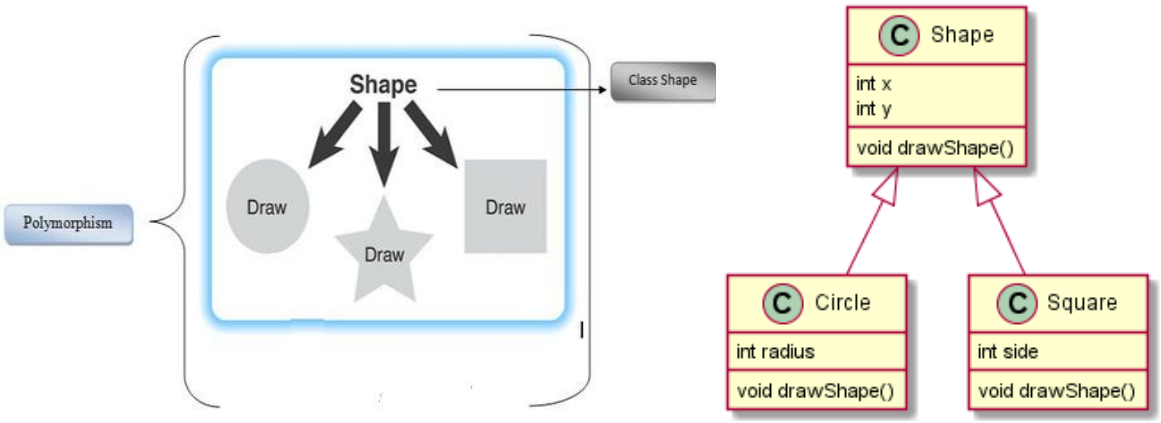


Рисунок 6.7 — Пример принципа полиморфизма

6.2. Создание классов и объектов. Конструктор. Деструктор

Перейдем непосредственно к реализации класса на python.

Синтаксис класса выглядит следующим образом:

```
class Имя_Класса():
    переменная = значение
    def имя_метода(self,...,...):
        self.переменная = значение
```

Создать объект (экземпляр класса) можно следующим образом:

```
имя_объекта = Имя_Класса ([параметры конструктора]),
```

где [параметры конструктора] — необязательный параметр.

Методы экземпляра класса имеют обязательный первый параметр (*self*), который содержит ссылку на объект, для которого вызван метод.

Пример 6.2.1. Создать класс Person(). Создать объекты person1 и person2 класса Person.

```
class Person():
    pass
person1 = Person()
person2 = Person()
print(type(person1),type(person2))
print(person1, person2)
print(person1 == person2)
```

```
<class '__main__.Person'> <class '__main__.Person'>
<__main__.Person object at 0x0000017F9EA88790> <__main__.Person object at 0x0000017F9EA886D0>
False
```

Каждый объект принадлежит определенному классу (или, как говорят, является ***экземпляром*** класса). Экземпляры одного класса имеют обычно одинаковые наборы атрибутов, но конкретные значения полей данных у каждого объекта свои.

При создании объекта в памяти под него выделяется место. Если инструкция создания объекта присваивается в качестве значения переменной (как в примере 6.2.1), то такая переменная будет ссылаться на объект и фактически значением переменной является не сам объект, а его адрес. Первый оператор print выводит типы этих экземпляров. Оба они принадлежат классу Person, а имя __main__ — это имя системного базового класса, подклассами которого являются все классы Python. Второй print выдает сами экземпляры. В этом случае для каждого из них указывается длинное шестнадцатеричное число — идентификатор экземпляра (можно считать, что это адрес в памяти, где расположен объект). Два объекта имеют разные идентификаторы, то есть расположены в разных участках памяти. Следующий оператор print показывает, что эти два объекта не равны друг другу, это разные объекты, хотя созданы от одного и того же класса и ничем не отличаются.

Добавим в описание класса Person один атрибут — поле name. Программный код приведен в примере 6.2.2. Поскольку при создании класса неизвестны имена конкретных людей — объектов класса, разумно определить этот атрибут как пустую строку.

Далее, после создания двух экземпляров этого класса выводим значение поля name для каждого созданного объекта. Из примера 6.2.2 видно, что в обоих случаях это пустая строка. Далее, после создания экземпляров, даем объектам person1 и person2 имена «Петр» и «Павел». Кроме этого, для Петра мы указываем динамически еще один атрибут (gender) — мужской. Первый оператор print выполняется успешно и выдает пустую строку, т.к. изначально атрибут name = ' '. После переопределения имен, второй оператор print выдает имена обоих наших персонажей, а для Петра еще и его пол. А вот попытка выдать пол Павла приводит к ошибке, ведь у Павла нет атрибута gender.

Пример 6.2.2. Создать класс Person(). Создать объекты person1 и person2 класса Person. Создать статический атрибут name класса Person(). Переопределить атрибут name для каждого объекта. Динамически создать атрибут gender для одного из объектов.

```
class Person():
    name=''
person1 = Person()
person2 = Person()
print(person1.name, person2.name)
person1.name = "Дмитрий"
person1.gender = "мужской"
person1.name = "Илья"
print(person1.name, person1.gender, person2.name)
print(person1.name, person1.gender, person2.name, person2.gender)
```

```
Илья мужской

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-b07dbbc8fe94> in <module>
      8 person1.name = "Илья"
      9 print(person1.name, person1.gender, person2.name)
---> 10 print(person1.name, person1.gender, person2.name, person2.gender)

AttributeError: 'Person' object has no attribute 'gender'
```

Существуют методы со специальными именами, предназначенные для выполнения определенных действий. Одним из таких магических методов является метод __init__() (в начале и конце имени здесь стоят по два символа подчеркивания), который по аналогии с другими языками программирования называют конструктором. Конструктор автоматически вызывается при создании экземпляра класса. Обычно используется для задания начальных значений атрибутов. Этот метод, как любая функция, может иметь параметры (помимо self), которые указываются при создании объекта. Служебное слово self означает именно тот объект, для которого вызывается метод. Явно указывать имя объекта при его создании не нужно, в Python при вызове метода класса первый аргумент — ссылку на текущий экземпляр класса — интерпретатор всегда добавляет автоматически (этим, собственно, вызов метода класса и отличается от вызова обычной функции).

Пример 6.2.3. Создать класс Person(). Создать объекты person1 и person2 класса Person, используя конструктор.

```
class Person():
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender
person1 = Person("Дмитрий","мужской")
person2 = Person("Илья","мужской")
print(person1.name, person1.gender)
print(person2.name, person2.gender)
person3=Person()
```

```
Дмитрий мужской
Илья мужской

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-9fba5d00d602> in <module>
      7 print(person1.name, person1.gender)
      8 print(person2.name, person2.gender)
----> 9 person3=Person()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'gender'
```

Благодаря такому подходу интерпретатор в примере 6.2.3 сам понимает, что self в первом случае указывает на объект person1, а во втором — на объект person2.

Начальные значения полей в примере 6.2.3 задано непосредственно при создании объекта. Обратите внимание, что в конструктор не нужно передавать параметр, соответствующий параметру self, который указан в заголовке метода __init__(). Первый аргумент интерпретатор Python добавит автоматически.

Бывает, что надо допустить создание объекта, даже если никакие данные в конструктор не передаются (пример 6.2.4). В таком случае параметрам конструктора класса задаются значения по умолчанию, как это сделано для полей name и gender.

Пример 6.2.4. Создать класс Person(). Создать объекты person1 и person2 класса Person, используя конструктор с параметрами по умолчанию.

```
class Person():
    def __init__(self,name,gender,age=19):
        self.name=name
        self.gender=gender
        self.age=age
person1 = Person("Дмитрий","мужской",20)
person2 = Person("Илья","мужской")
print(person1.name, person1.gender, person1.age)
print(person2.name, person2.gender, person2.age)
person3=Person()
```

```
Дмитрий мужской 20
Илья мужской 19

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-a3fcaa16ac95> in <module>
      8 print(person1.name, person1.gender, person1.age)
      9 print(person2.name, person2.gender, person2.age)
---> 10 person3=Person()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'gender'
```

Если объект класса создается с параметрами, то используются те значения, которые указаны при вызове (как при создании объекта person1).

Если описание конструктора предполагает использование и параметров по умолчанию, и параметров, которые требуют явной инициализации, то параметры по умолчанию должны быть указаны в конце списка аргументов — это позволит опустить их при вызове методы для создания объекта (как это сделано при создании объекта person3).

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся длинными и громоздкими. В такой ситуации часть одного класса нередко можно записать в виде отдельного класса. Большой код разбивается на меньшие классы, которые работают во взаимодействии друг с другом.

Например, при создании класса Car (пример 6.2.5), оказалось, что в нем может появиться слишком много атрибутов, относящихся к аккумуляторной батарее. В таком случае можно переместить все эти атрибуты и методы в отдельный класс с именем Battery. Затем экземпляр Battery становится атрибутом объекта класса Car.

Пример 6.2.4. Создать класс Car, хранящий информацию о машине, и класс Battery, хранящий информации о размере аккумуляторной батареи. Организовать вложенность объектов.

```
class Battery():
    def __init__(self,size=50):
        self.battery_size=size
class Car():
    def __init__(self,make,model,year):
        self.make=make
        self.model=model
        self.year=year
        self.odometer_reading=0
        self.battery=Battery()
car1=Car('Lada','Granta',2022)
print(car1.make,car1.model,car1.year,car1.battery.battery_size)
```

```
Lada Granta 2022 50
```

Когда потребуется вывести емкость аккумулятора, необходимо обратиться к атрибуту battery_size через вложенный объект car1.battery, как показано в примере 6.2.4. Эта строка подсказывает Python обратиться к экземпляру car1, найти его атрибут battery и обратиться к полю battery_size, связанному с соответствующим экземпляром класса Battery.

В языке программирования Python объект уничтожается, когда исчезают все связанные с ним переменные или им присваивается другое значение, в результате чего связь со старым объектом теряется. В явном виде удалить какую-либо переменную можно с помощью команды языка del. В ООП кроме конструктора объектов есть обратный ему метод — ***деструктор***. Он вызывается, когда объект не создается, а уничтожается.

Пример 6.2.5. Создать класс Person(). Создать объекты person1 и person2 класса Person, используя конструктор. Удалить созданные объекты с помощью деструктора.

```
class Person():
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender
    def __del__(self):
        print('Объект', self.name, 'удаляется!')
person1 = Person("Дмитрий","мужской")
person2 = Person("Илья","мужской")
print(person1.name, person1.gender)
print(person2.name, person2.gender)
```

```
Дмитрий мужской
Илья мужской
Объект Дмитрий удаляется!
Объект Илья удаляется!
```

В классах Python функцию деструктора выполняет метод с заголовком def __del__(self). Деструктор вызывается перед удалением объекта для выполнения каких-либо действий по устранению следов, которые могли остаться от объекта. Удаление объектов, в частности, происходит при завершении работы программы. В нашем простом примере в деструкторе нет никакой необходимости, но для полноты картины добавим к классу Person деструктор, который будет просто сообщать об удалении объектов. В Python деструктор используется редко, так как интерпретатор и без него хорошо убирает «мусор» из памяти.

6.3. Атрибуты класса и атрибуты объекта

В классе Person (пример 6.2.6) описаны три поля — name, count и surname. Но для того, чтобы работать с этими полями, нам совсем необязательно создавать объекты — мы можем делать это через имя класса Person. Обратим внимание на то, что атрибуты, описанные в классе, относятся к так называемому пространству имен класса или, проще говоря, к атрибутам класса. ***Атрибуты класса*** объявляются вне любого метода.

В то же время к атрибутам класса можно обращаться через созданные объекты person1 и person2. В конструкторе с помощью указателя self обращаются не к классу, а к конкретному объекту, для которого вызван этот конструктор.

Внутри конструктора можно также обратиться к ***атрибуту объекта***. Для этого нужно использовать либо встроенный атрибут __class__, либо имя родительского объекта. В приведенном фрагменте кода (пример 6.3.1) показаны оба варианта, поэтому при создании каждого экземпляра класса Person значение поля count инкрементируется дважды.

Если требуется посмотреть значения атрибутов класса Person, то можно убедиться, что у класса имеются три атрибута. Поля name и surname по-прежнему содержат пустые строки, а поле count равно четырем.

Пример 6.3.1.

```
class Person():
    name=''
    surname=''
    count=0
    def __init__(self,name,surname):
        self.name=name
        self.surname=surname
        Person.count+=1
        __class__.count+=1
person1 = Person("Дмитрий","Петров")
person2 = Person("Илья","Иванов")
person1.count+=10
print(person1.__dict__)
person2.count+=100
print(person2.__dict__)
print(Person.__dict__)
```

```
{'name': 'Дмитрий', 'surname': 'Петров', 'count': 14}
{'name': 'Илья', 'surname': 'Иванов', 'count': 104}
{'__module__': '__main__', 'name': '', 'surname': '', 'count': 4,..
```

6.4. Методы класса, методы объекта и статические методы

Существует три вида методов в парадигме объектно-ориентированного программирования:

- методы класса;
- методы экземпляра класса (объекта);
- статические методы.

Рассмотрим принципы работы этих методов подробнее.

До сих пор мы определяли методы просто как функции внутри класса. У каждого такого метода, как правило, первым идет параметр self — ссылка на экземпляр класса (то есть объект), из которого метод был вызван.

Такие методы — то есть методы экземпляра класса или методы объекта можно вызывать двумя способами:

- через экземпляр класса, как это показано в примере 6.4.1 для объекта person1;
- напрямую через класс, но в этом случае необходимо в качестве первого аргумента подставлять имя объекта данного класса (которое внутри метода используется как self), как это показано для объекта person2 в примере 6.4.1.

Пример 6.4.1.

```
class Person():
    def __init__(self,name,surname):
        self.name=name
        self.surname=surname
    def get_fio(self):  #метод объекта
        return self.name+' '+self.surname

person1 = Person("Дмитрий","Петров")
person2 = Person("Илья","Иванов")
print(person1.get_fio())
print(person2.get_fio())
```

```
Дмитрий Петров
Илья Иванов
```

Теперь добавим в класс Person несколько новых полей — возраст начала и возраст окончания трудовой деятельности, а также с помощью специальной конструкции Python — «декоратора» @classmethod, опишем метод validate(). Задача метода validate() — проверить, попадает ли переданный параметр в допустимые границы возраста человека. Но в данном примере кроме метода validate() появился декоратор @classmethod. Декораторы — это такие «обёртки» вокруг Python-функций (или классов), которые изменяют работу того, к чему они применяются. Использование декоратора @classmethod, в частности, приводит к тому, что в метод в качестве первого параметра передается не указатель self на экземпляр класса (то есть на объект, для которого вызван этот метод), а указатель на класс cls, которому принадлежит данный объект. Благодаря такой замене внутри этого методы, с одной стороны, получаем доступ к атрибутам класса (полям start_age и finish_age), но с другой — теряем доступ к полям объекта, так как у нас теперь нет указателя self.

Поскольку метод validate() благодаря декоратору @classmethod теперь связан с классом Person, а не с конкретным его объектом, то можно вызывать этот метод напрямую применительно к классу. Обратите внимание, что когда в конструкторе класса Person проверяется возраст на соответствие допустимому диапазону, то вызывается метод класса validate(), причем делаем это относительно класса Person. Благодаря такому вызову внутри метода validate() идет обращение к полям класса Person и переданный аргумент проверяется на допустимость. Если age попадает в допустимый диапазон, то оно заносится в поле объекта ag (как в случае объекта prs1), а если выходит за границы диапазона, то в поле объекта ag остается значение 0 (как для объекта prs2).

Пример 6.4.2.

```
class Person():
    start_age=18
    finish_age=60
    def __init__(self,name,surname,age):
        self.name=name
        self.surname=surname
        self.ag=0
        if Person.validate(age):
            self.ag=age
    def get_fio(self):
        return self.name+' '+self.surname
    @classmethod
    def validate(cls,arg):  #метод класса
        return cls.start_age<=arg<=cls.finish_age

print(Person.validate(35))
person1 = Person("Дмитрий","Петров",44)
person2 = Person("Илья","Иванов",12)
print(person1.__dict__)
print(person2.__dict__)
```

```
True
{'name': 'Дмитрий', 'surname': 'Петров', 'ag': 44}
{'name': 'Илья', 'surname': 'Иванов', 'ag': 0}
```

Третий тип методов — статические методы, определяются декоратором @staticmethod (пример 6.4.3). Это методы, которые не имеют доступа ни к атрибутам класса, ни к атрибутам его экземпляров, то есть, некая независимая, самостоятельная функция, объявленная внутри класса. Они работают только с теми данными, которые им передаются в качестве аргументов. Обычно, это делают для удобства, т.к. их функционал так или иначе связан с тематикой класса.

В приведенном примере метод comp_exp() объявлен с декоратором @staticmethod, что делает его статическим. Это значит, что при вызове в этот метод не передаются ни указатель на класс, для которого он вызывается, ни указатель на объект. Поэтому внутри этого метода можно работать только с аргументами, переданными в него при вызове. В приведенном примере метод comp_exp() возвращает длительность трудового стажа для текущего возраста относительно возраста начала трудовой деятельности, причем оба эти параметра он получает как аргументы при вызове.

Пример 6.4.3.

```
class Person:
    start_age = 18
    finish_age = 60

    def __init__(self, nm, srn, age):
        self.name = nm
        self.surname = srn
        self.ag = 0
        if Person.validate(age):
            self.ag = age
        self.experience = Person.comp_exp(self.ag, Person.start_age)

    @classmethod
    def validate(cls, arg):
        return cls.start_age <= arg <= cls.finish_age

    @staticmethod
    def comp_exp(age, start_age):
        if age > start_age:
            return age - start_age
        else:
            return 0
```

```
prs1 = Person("Ivan", "Petrov", 44)
prs2 = Person("Sergey", "Ivanov", 12)
```

```
print(prs1.__dict__)
print(prs2.__dict__)
```

```
{'name': 'Ivan', 'surname': 'Petrov', 'ag': 44, 'experience': 26}
{'name': 'Sergey', 'surname': 'Ivanov', 'ag': 0, 'experience': 0}
```

Подведем итог различных типов методов в классах. Обычные методы объектов, как правило, вызываются из экземпляров классов и работают с атрибутами экземпляров и атрибутами классов. Методы классов обычно вызываются через класс и имеют доступ только к атрибутам самого класса, в котором объявлены. Статические методы — это совершенно изолированные функции, которые работают только с параметрами вызова и не имеют доступа к атрибутам класса или его экземпляров.

6.5. Перегрузка операторов. Магические методы

Перегрузка операторов в Python это возможность переопределять поведения магических методов в классах, в зависимости от контекста вызова метода.

Магические методы в Python — это специальные методы, которые значительно облегчают использование объектов. Магические методы легко узнать в коде классов, потому что они заключены в двойные подчеркивания «__». Например. __init__, является одним из тех самых «волшебных» методов в Python.

Магические методы не предназначены для прямого вызова, однако их вызов происходит внутри класса при определенном действии. Например, мы явно не вызываем метод __init__ при создании нового объекта класса, но вместо этого этот метод вызывается внутренне.

Магический метод __str__ представляет объект в виде строки. Метод вызывается при выводе с помощью функции print(), а также при использовании функции str().

Рассмотрим пример 6.5.1 неявного вызова магических методов __init__ и __str__.

Пример 6.5.1. Создать класс Bank(), моделирующий работу банковского счета. Возможные методы класса: изменение суммы банковского счета, вывод информации о количестве счетов владельца и суммах счетов, а также методы конструктора класса (__init__) и вывода полной информации о номере счета (__str__).

```
class Bank():
    count = 0
    def __init__ (self,number):
        self.number = number
        self.balance = 0
        Bank.count = Bank.count+1
    def __str__(self):
        r = 'Номер счета равен '+str(self.number)
        return r
    def changer(self, balance):
        self.balance = balance
    def display_count (self):
        print('Номер',self.number, 'баланс', self.balance)
        print('Сколько счетов', Bank.count)
b= Bank(123456678899) #срабатывает магический метод __init__
print(b) #срабатывает магический метод __str__
b.changer(1000)
b.display_count()
```

```
Номер счета равен 123456678899
Номер 123456678899 баланс 1000
Сколько счетов 1
```

Магический метод __call__ позволяет вызывать экземпляры класса подобно функциям через оператор круглые скобки.

Если записать следующую команду, то возникнет ошибка: «TypeError: 'Counter' object is not callable» (рис. 6.8).

```
class Counter:
    def __init__(self):
        self.__counter = 0

c = Counter()
c()
```

```
TypeError: 'Counter' object is not callable
```

Рисунок 6.8 — Пример вызова объекта как функции без использования метода __call__

Можно поправить эту ошибку, если явно в классе Counter прописать магический метод __call__, например, так:

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, *args, **kwargs):
        print("__call__")
        self.__counter += 1
        return self.__counter

c = Counter()
c()
```

```
__call__
```

```
1
```

Рисунок 6.9 — Пример вызова объекта как функции с использованием метода __call__

На рис. 6.9 выводим сообщение, что был вызван данный метод, затем увеличиваем счетчик counter для текущего объекта на 1 и возвращаем его.

Классы, экземпляры которых можно вызывать подобно функциям, получили название ***функторы***.

В примере метод __call__ возвращает значение счетчика, поэтому с объектом можно работать, следующим образом:

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, *args, **kwargs):
        print("__call__")
        self.__counter += 1
        return self.__counter
```

```
c = Counter()
c()
```

```
__call__
```

```
1
```

```
c()
```

```
__call__
```

```
2
```

```
res = c()
```

```
__call__
```

```
print(res)
```

```
3
```

Рисунок 6.9 — Функторы

На рис. 6.9 три раза вызывается метод __call__ и счетчик __counter трижды увеличивается на единицу. Поэтому в консоли видно значение 3.

В методе __call__ записаны параметры *args, **kwargs. Это значит, что при вызове объектов можно передавать им произвольное количество аргументов. Например, на рис. 6.10 указано значение изменения счетчика при текущем вызове.

```
class Counter:
    def __init__(self):
        self.__counter = 0

    def __call__(self, step=1, *args, **kwargs):
        self.__counter += step
        return self.__counter
```

```
c = Counter()
c(2)
```

```
2
```

```
c(10)
```

```
12
```

```
res = c()
res
```

```
13
```

```
res2 = c(-5)
res2
```

```
8
```

Рисунок 6.10 — Функторы с параметрами

Здесь появился в явном виде первый параметр step с начальным значением 1.

Ниже представлен программный код с ответом на вопрос, где может пригодиться использование магического метода __call__. Приведем пример с использованием класса с методом __call__ вместо замыканий функций. Рассмотрим класс StripChars (пример 6.5.2), который удаляет вначале и в конце строки заданные символы:

Пример 6.5.2.

```
class StripChars:
    def __init__(self, chars):
        self.__chars = chars

    def __call__(self, *args, **kwargs):
        if not isinstance(args[0], str):
            raise ValueError("Аргумент обязан быть строкой")
        return args[0].strip(self.__chars)
```

```
s1 = StripChars("?:!.; ")
s2 = StripChars(" ")
```

```
res = s1(" Hello World! ")
res2 = s2(" Hello World! ")
```

```
print(res, res2, sep='\n')
```

```
Hello World
Hello World!
```

Для этого в инициализаторе сохраняем строку __chars — удаляемые символы, а затем, при вызове метода __call__ удаляем символы через строковый метод strip для символов __chars. То есть теперь можно создать экземпляр класса и указать те символы, которые следует убирать, а затем вызвать объект s1 подобно функции. В результате объект s1 будет отвечать за удаление указанных символов в начале и конце строки. Также можно определить другие объекты этого класса с другим набором символов. То есть, объект s2 уже отвечает только за удаление пробелов, а s1 и некоторых других символов.

Магический метод __len__ позволяет применять функцию len() к экземплярам класса. Рассмотрим пример использования магической функции __len__() на следующей задаче.

Пример 6.5.3. Создать класс Marks, который хранит оценки ученика по определенному предмету, а также дату получения этой оценки в виде списка кортежей. Например, Маша имеет оценки [('матем.',5,10.02.2023), ('история',4,10.02.2023), ('химия',5,11.02.2023), ('матем.',5,11.02.2023)]. Определить, сколько оценок получила Маша в определенный день, например, 11.02.2023. Для решения задачи использовать магический метод __len__().

```
class Marks():
    def __init__(self):
        self.spisok = []
        predmet = input('Введите предмет: ')
        ocenka = int(input(f'Введите оценку по предмету {predmet}: '))
        data = input('Введите дату вида dd.mm.yyyy: ')
        self.spisok.append((predmet, ocenka, data))

    def get_mark(self, p, o, d):
        self.spisok.append((p, o, d))

    def __len__(self):
        kol = 0
        for i in self.spisok:
            if i[-1]=='11.02.2023':
                kol += 1
        return kol
```

```
masha = Marks()
print(masha.__dict__)
```

```
Введите предмет: Математика
Введите оценку по предмету Математика: 5
Введите дату вида dd.mm.yyyy: 11.02.2023
{'spisok': [('Математика', 5, '11.02.2023')]}
```

```
predmet = input('Введите предмет: ')
ocenka = int(input(f'Введите оценку по предмету {predmet}: '))
data = input('Введите дату вида dd.mm.yyyy: ')
masha.get_mark(predmet, ocenka, data)
```

```
Введите предмет: Физика
Введите оценку по предмету Физика: 4
Введите дату вида dd.mm.yyyy: 12.02.2023
```

```
predmet = input('Введите предмет: ')
ocenka = int(input(f'Введите оценку по предмету {predmet}: '))
data = input('Введите дату вида dd.mm.yyyy: ')
masha.get_mark(predmet, ocenka, data)
```

```
Введите предмет: Химия
Введите оценку по предмету Химия: 5
Введите дату вида dd.mm.yyyy: 11.02.2023
```

```
print(masha.__dict__)
print(len(masha))
```

```
{'spisok': [('Математика', 5, '11.02.2023'), ('Физика', 4, '12.02.2023'), ('Химия', 5, '11.02.2023')]}
2
```

Магические методы для работы с арифметическими операторами:

- __add__() — для операции сложения;
- __sub__() — для операции вычитания;
- __mul__() — для операции умножения;
- __truediv__() — для операции деления.

Магические методы для реализации операторов сравнения:

- __eq__() — для равенства ==;
- __ne__() — для неравенства !=;
- __lt__() — для оператора меньше <;
- __le__() — для оператора меньше или равно <=;

- __gt__() — для оператора больше >;
- __ge__() — для оператора больше или равно >=.

Рассмотрим пример перегрузки арифметических и логических операторов (пример 6.5.4.).

Пример 6.5.4. Создать класс Number, хранящий значение числа. Создать несколько объектов класса Number. Осуществить сложение, вычитание, умножение, деление и сравнение данных объектов, используя принцип перегрузки операторов. В качестве магических методов использовать __add__(), __sub__(), __mul__(), __truediv__() для арифметических операций и __lt__(), __le__(), __eq__(), __ne__(), __gt__(), __ge__() для логических операций.

```
class Number():
    def __init__(self, value) :
        self.value = value

    def __add__(self, other):
        return self.value + other.value

    def __sub__(self, other):
        return self.value - other.value

    def __mul__(self, other):
        return self.value * other.value

    def __truediv__(self, other):
        return self.value / other.value

    def __lt__(self, other):
        return self.value < other.value

    def __le__(self, other):
        return self.value <= other.value

    def __eq__(self, other):
        return self.value == other.value

    def __ne__(self, other):
        return self.value != other.value

    def __gt__(self, other) :
        return self.value > other.value

    def __ge__(self, other) :
        return self.value >= other.value
```

```
n1 = Number(5)
n2 = Number(8)
```

```
print(n1.__dict__)
print(n2.__dict__)
```

```
{'value': 5}
{'value': 8}
```

```
print(f'__add__: {n1 + n2}')
print(f'__sub__: {n1 - n2}')
print(f'__mul__: {n1 * n2}')
print(f'__truediv__: {n1 / n2}')
print(f'__lt__: {n1 < n2}')
print(f'__le__: {n1 <= n2}')
print(f'__eq__: {n1 == n2}')
print(f'__ne__: {n1 != n2}')
print(f'__gt__: {n1 > n2}')
print(f'__ge__: {n1 >= n2}')
```

```
__add__: 13
__sub__: -3
__mul__: 40
__truediv__: 0.625
__lt__: True
__le__: True
__eq__: False
__ne__: True
__gt__: False
__ge__: False
```

Рассмотрим магические методы __iter__() и __next__() и принципы их использования.

- __iter__(self) — получение итератора для перебора объекта;
- __next__(self) — переход к следующему значению и его считывание

В структурном и функциональном программировании перебрать значения итерируемого объекта можно через итератор (пример 6.5.5).

Пример 6.5.5.

```
a = iter([0, 1, 2, 3, 4])

print(a)
print(next(a))
print(next(a))
print(next(a))
print(next(a))

while True:
    try:
        print(next(a))
    except StopIteration:
        print('Конец последовательности')
        break
```

```
<list_iterator object at 0x0000028722CEFCA0>
0
1
2
3
4
Конец последовательности
```

В конце последовательности генерируется исключение StopIteration.

Таким же образом можно работать с объектами класса в объектно-ориентированном программировании (пример 6.5.6), используя магические методы __iter__() и __next__().

Пример 6.5.6.

```
class New():
    def __init__(self):
        self.a=iter([0, 1, 2, 3, 4])

    def __next__(self):
        return next(self.a)
```

```
b = New()
print (b.__dict__)
```

```
{'a': <list_iterator object at 0x0000028722CD2040>}
```

```
print(next(b))
print(next(b))
print(next(b))
print(next(b))
```

```
0
1
2
3
```

Рассмотрим еще один пример использования магических методов __iter__ и __next__. Создадим класс FRange (пример 6.5.7), который будет выдавать последовательность вещественных чисел арифметической прогрессии. Класс FRange должен восприниматься как итерируемый объект с возможностью перебора функцией next().

Пример 6.5.7.

```
class FRange:
    def __init__(self, start=0.0, stop=0.0, step=1.0):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        self.value = self.start - self.step
        return self

    def __next__(self):
        if self.value + self.step < self.stop:
            self.value += self.step
            return self.value
        else:
            print('StopIteration')
```

```
fr = FRange(0, 2, 0.5)
it = iter(fr)
print(next(it), end=' ')
print(next(it), end=' ')
print(next(it), end=' ')
print(next(it), end=' ')
```

```
0.0 0.5 1.0 1.5
```

```
print(next(it), end=' ')
```

```
StopIteration
None
```

Здесь функция next() вызывает метод __next__ и возвращенное им значение, возвращается функцией next().

6.6. Примеры решения задач

1. Вы — бизнесмен. На нескольких планетах галактики были обнаружены залежи аноботанимуа. До планет разное расстояние в световых годах и разная цена разработки одного килограмма минерала в условных единицах. Каждый световой год пути обходится в 5.000.000 у.е. Корабль может поднять на борт до 100.000 т. груза. Нужно создать колонию, завезти оборудование и т.д. На некоторых из этих планет существуют враждебные племена аборигенов и хищные животные. Для защиты рабочих требуется содержать военизированные подразделения. Написать объектно-ориентированную программу с использованием двух классов, которая определит наиболее выгодный проект. Класс может содержать поля: название планеты, расстояние до планеты в световых годах, стоимость разработки одного килограмма минерала в у.е. словарь дополнительных расходов, котором указывается название дополнительных расходов и затраты на его обслуживание; и методы: заполнение полей, нахождение расходов на добычу и доставку, заполнение словаря дополнительных расходов, расчет общей прибыли, вывод результатов на экран.

```
class Planet():
    y1c=5000000
    sbmax=100000000
    #константы из условия, стоимость одного светового года в пути и вместимость корабля(в кг)
    def __init__(self,name,rang,co,mar):
        #название планеты
        self.name=name
        #расстояние до планеты
        self.rang=rang
        #стоимость разработки 1 кг минерала(сколько стоит добыть 1 кг)
        self.co=co
        #стоимость продажи 1 кг минерала(сколько можно получить за 1 кг)
        self.mar=mar
        #словарь дополнительных издержек для работы на планете
        self.costs={}
    def addcost(self,*val):
        #добавляем дополнительные издержки в словарь self.costs
        for i in val:
            self.costs[i[0]]=i[1]
    def costsf(self):
        #стоимость доставки минералов туда-обратно и стоимость реализации полностью загруженного корабля
        s=Planet.y1c*2*self.rang+Planet.sbmax*self.co
        #если словарь не пустой,то также учитываем дополнительные издержки
        if bool(self.costs)==True:
            for i in self.costs.values():
                s+=i
        return s
    def income(self):
        #максимальная вместимость корабля на стоимость 1 кг
        return self.mar*Planet.sbmax
    def profit(self):
        #разность доходов и расходов
        return self.income()-self.costsf()
    def __str__(self):
        #вывод информации о планете
        print(f'{"Информация о планете":>25} {self.name:<}')
        return f'{"Расходы:":>25} {self.costsf():<}\n \
        {"Доходы:":>25} {self.income():<} \n {"Прибыль:":>25} {self.profit():<} '
```

```
p1=Planet('Марс',100,10,25)
#добавляем дополнительные издержки
p1.addcost(('Охрана',20000000),('Оборудование',300000000))
print(p1)
```

```
     Информация о планете Марс
                 Расходы: 2320000000
                          Доходы: 2500000000
                  Прибыль: 180000000
```

```
p2=Planet('Луна',25,10,15)
#добавляем дополнительные издержки
p2.addcost(('Охрана',2000000),('Оборудование',10000000),('Налог на использование спутника Земли',900000000))
print(p2)
```

```
     Информация о планете Луна
                 Расходы: 2162000000
                          Доходы: 1500000000
                  Прибыль: -662000000
```

2. Вы — HR-менеджер крупной компании. В связи с изменением политики компании появились новые вакантные места, и ряд должностей перестал быть необходимым. Вам нужно пересмотреть существующие кадры, при необходимости провести обучение и переподготовку и (или) нанять новых работников. Каждая вакантная должность связана с рядом компетенций, которыми должен обладать претендент (или, если он член компании, должен их получить в ходе обучения). У вас есть список резюме претендентов. Написать объектно-ориентированную программу с использованием двух классов, которая по-

зволит выбрать наиболее достойных. Первый класс может содержать поля: название вакантной должности, критерии отбора (массив, состоящий из элементов: необходимое качество, значимость по 10 бальной шкале, возможность коррекции по 10 бальной шкале, допустимый порог); методы: заполнение полей, вывод результатов на экран. Второй класс может содержать поля: ФИО претендента, член компании или нет, уровень готовности к работе (массив, состоящий из элементов: необходимое качество, балл по 10 бальной шкале); методы: заполнение полей: определение уровня готовности претендента (находится по формуле: сумма произведений набранных баллов по каждому критерию на их вес), список необходимых качеств, которые нужно «подтянуть» с указанием критичности каждого из них, вывод результатов на экран.

```
class Vacation():
    # объявление новой вакансии, name - название, skills - требуемые навыки (можно задать с помощью addskill)
    def __init__(self, name, skills=[]):
        self.name = name
        self.skills = skills

    # добавляет требование к навыкам, skill - название навыка, ball - необходимый балл,
    # corr - на сколько баллов возможно
    # обучить работника, minimum - заданный порог по навыку
    def addskill(self, skill, ball, corr, minimum):
        self.skills.append((skill, ball, corr, minimum))
        self.skills = list(set(self.skills))

    # вывод информации о вакансии (название, необходимые навыки)
    def __str__(self):
        skills = '\n- '.join([f"{skill} - {ball}, возможно обучение на {corr} баллов,\
        порог - {mn}" for skill, ball, corr, mn in self.skills])
        return f"Должность вакантного места: {self.name}\n Необходимые качества:\n- {skills}"
```

```
class Person():
    # объявление работника, name - его имя, incompany (True/False) - работает ли он уже в компании или нет,
    # skills - требуемые навыки (можно задать с помощью addskill)
    def __init__(self, name, incompany, skills=[]):
        self.name = name
        self.incompany = incompany
        self.skills = skills

    # добавляет навык работника, skill - название навыка, ball - балл навыка
    def addskill(self, skill, ball):
        self.skills.append((skill, ball))
        self.skills = list(set(self.skills))

    # возвращает уровень всех навыков по формуле в условии, vac - вакансия, на которую человек подаётся
    def getlevel(self, vac):
        result = 0
        for skill, ball in self.skills:
            for skill_vac in vac.skills:
                if skill == skill_vac[0]:
                    result += ball * skill_vac[1]
                    break
        return result
```

```
    # возвращает необходимые требования, до которых работник не дотягивает на вакансии vac (если они есть)
    # или которые можно подтянуть
    def requirements(self, vac):
        result = []
        for skill_vac in vac.skills:
            if skill_vac[0] not in [skill[0] for skill in self.skills]:
                result.append(f"Отсутсвует необходимый навык: {skill_vac[0]}")
            for skill, ball in self.skills:
                if skill == skill_vac[0]:
                    if ball < skill_vac[3]:
                        result.append(f"Не пройден порог по навыку:\
{skill_vac[0]} (текущий балл - {ball}, порог - {skill_vac[3]})")
                        break
                    if ball >= skill_vac[1]:
                        break
                    if ball >= skill_vac[1] - skill_vac[2]:
                        result.append(f"Можно обучить навык: \
{skill_vac[0]}(текущий балл - {ball}, можно обучить до - {ball + skill_vac[2]})")
        if len(result) == 0:
            result = ['Все навыки соответсвуют должности']
        return '\n'.join(result)

    # вывод информации о работнике (имя, является ли работником компании, его навыки)
    def __str__(self):
        skills = '\n- '.join([f"{skill} - {ball} баллов" for skill, ball in self.skills])
        incompany = 'в компании' if self.incompany else 'не в компании'
        return f"{self.name}, {incompany}, навыки:\n- {skills}"
```

```
v1 = Vacation('Экономист')
v1.addskill('Знание математики', 9, 2, 7)
v1.addskill('Усидчивость', 10, 5, 5)
v1.addskill('Знание экономики', 10, 3, 7)
print(v1)
```

```
Должность вакантного места: Экономист
 Необходимые качества:
- Усидчивость - 10, возможно обучение на 5 баллов,        порог - 5
- Знание математики - 9, возможно обучение на 2 баллов,        порог - 7
- Знание экономики - 10, возможно обучение на 3 баллов,        порог - 7
```

```
p1 = Person('Василий Иванов', True)
p1.addskill('Знание математики', 6)
p1.addskill('Терпеливость', 10)
#p1.addskill('Усидчивость', 10)
p1.addskill('Знание экономики', 7)
print(p1)
```

```
Василий Иванов, в компании, навыки:
- Знание экономики - 7 баллов
- Терпеливость - 10 баллов
- Знание математики - 6 баллов
```

```
print(p1.getlevel(v1))
```

```
124
```

```
print(p1.requirements(v1))
```

```
Отсутсвует необходимый навык: Усидчивость
Не пройден порог по навыку:Знание математики (текущий балл - 6, порог - 7)
Можно обучить навык: Знание экономики(текущий балл - 7, можно обучить до - 10)
```

3. Магазин компьютерной техники покупает по оптовым ценам компьютерные комплектующие. Техническая служба магазина собирает под заказ из них компьютеры. Общая цена компьютера складывается из цены комплектующих, наценки и стоимости ра-

бот. Наценка составляет 20% на комплектующие дешевле и 15% дороже 2000 руб. Если суммарная цена набора составляет более 50000 руб., то стоимость сборки не включается в общую цену товара. В противном случае стоимость сборки составляет 5% от общей цены набора. Создать следующий класс с рекомендуемыми полями: номер заказа, список комплектующих компьютера с указанием их названия и оптовой цены; а также методы: расчет цены компьютера, вывод результатов на экран.

```
class Shop():
    def __init__(self,id_zakaza,spisok):
        self.id_zakaza=id_zakaza
        self.spisok=spisok
    def rachet(self):
        nacenka=[]
        for i in range(len(self.spisok)):
            q=spisok[i].split(':')
            print(q)
            if float(q[1])<2000:
                nacenka.append(float(q[1])+float(q[1])/100*20)
            else:
                nacenka.append(float(q[1])+float(q[1])/100*15)
        print(nacenka)
        self.nac=sum(nacenka)
        if self.nac>50000:
            self.cena_itog=self.nac
        else:
            self.cena_itog=self.nac+self.nac/100*5
    def __str__(self):
        return f'компьютер состоит из комплектующих {self.spisok} с итоговой ценой {self.cena_itog} '
```

```
spisok=[]
while True:
    nazv=input("Введите название комплектующего")
    cena=float(input('Введите цену комплектующего'))
    a=nazv+':'+str(cena)
    spisok.append(a)
    escho=input('Будете ли вводить еще одно комплектующее?')
    if escho.lower()=='no':
        break
    elif escho.lower()=='yes':
        continue
    else:
        print('можно вводить или yes или no')
        break

computer1=Shop(202103,spisok)
print(computer1.__dict__)
computer1.rachet()
print(computer1.__dict__)
print(computer1)
```

6.8. Задачи для самостоятельного решения

1) Опишите класс Book, заданный названием, автором, издательством и годом издания. Включите в описание класса методы: вывода информации о книге на экран, проверки, является ли книга новой (изданной в последние 5 лет), и свойство, позволяющее установить жанр книги.
2) Опишите класс Car, заданный маркой, моделью, годом выпуска и пробегом. Включите в описание класса методы: вывода информации о машине на экран, проверки, нужно ли произвести техническое обслуживание (пробег больше 10 000 км с последнего ТО), и свойство, позволяющее установить тип топлива (бензин, дизель и т. п.).
3) Опишите класс Employee, заданный фамилией, именем, должностью и зарплатой. Включите в описание класса методы: вывода информации о сотруднике на экран, проверки, является ли зарплата высокой (больше 100 000 рублей), и свойство, позволяющее установить стаж работы.
4) Опишите класс Product, заданный названием, ценой и количеством. Включите в описание класса методы: вывода информации о товаре на экран, проверки, есть ли товар в наличии (количество больше 0), и свойство, позволяющее установить категорию товара.
5) Опишите класс Circle, заданный радиусом. Включите в описание класса методы: вывода информации о круге на экран, расчета длины окружности и площади круга, и свойство, позволяющее установить цвет круга.
6) Создайте класс АВТОМОБИЛЬ с методами, позволяющими вывести на экран информацию об автомобиле, а также определить, подходит ли данный автомобиль для заданных условий. Создайте дочерние классы ЛЕГКОВОЙ АВТОМОБИЛЬ (марка, модель, год выпуска, тип кузова), ГРУЗОВОЙ АВТОМОБИЛЬ (марка, модель, год выпуска, грузоподъемность), АВТОБУС (марка, модель, год выпуска, количество мест) со своими методами вывода информации на экран и определения соответствия заданным условиям. Создайте список автомобилей, выведите полную информацию из базы на экран, а также организуйте поиск автомобилей с заданной маркой или годом выпуска.
7) Создайте класс СТУДЕНТ с методами, позволяющими вывести на экран информацию о студенте, а также определить, подходит ли данный студент для заданных условий. Создайте дочерние классы БАКАЛАВР (имя, фамилия, возраст, курс), МАГИСТР (имя, фамилия, возраст, специализация), АСПИРАНТ (имя, фамилия, воз-

раст, тема диссертации) со своими методами вывода информации на экран и определения соответствия заданным условиям. Создайте список студентов, выведите полную информацию из базы на экран, а также организуйте поиск студентов с заданным именем или курсом.

8) Создайте класс РЕСТОРАН с методами, позволяющими вывести на экран информацию о ресторане, а также определить, подходит ли данный ресторан для заданных условий. Создайте дочерние классы ИТАЛЬЯНСКИЙ РЕСТОРАН (название, адрес, тип кухни, рейтинг), ЯПОНСКИЙ РЕСТОРАН (название, адрес, тип кухни, рейтинг), ФРАНЦУЗСКИЙ РЕСТОРАН (название, адрес, тип кухни, рейтинг) со своими методами вывода информации на экран и определения соответствия заданным условиям. Создайте список ресторанов, выведите полную информацию из базы на экран, а также организуйте поиск ресторанов с заданным типом кухни или рейтингом.
9) Создайте класс СПОРТСМЕН с методами, позволяющими вывести на экран информацию о спортсмене, а также определить, подходит ли данный спортсмен для заданных условий. Создайте дочерние классы ЛЕГКОАТЛЕТ (имя, фамилия, возраст, дисциплина), ПЛОВЕЦ (имя, фамилия, возраст, дистанция), БОКСЕР (имя, фамилия, возраст, весовая категория) со своими методами вывода информации на экран и определения соответствия заданным условиям. Создайте список спортсменов, выведите полную информацию из базы на экран, а также организуйте поиск спортсменов с заданным именем или возрастом.
10) Создайте класс ТОВАР с методами, позволяющими вывести на экран информацию о товаре, а также определить, подходит ли данный товар для заданных условий. Создайте дочерние классы ЭЛЕКТРОНИКА (название, производитель, цена, тип устройства), ОДЕЖДА (название, производитель, цена, размер), ПРОДУКТЫ ПИТАНИЯ (название, производитель, цена, срок годности) со своими методами вывода информации на экран и определения соответствия заданным условиям. Создайте список товаров, выведите полную информацию из базы на экран, а также организуйте поиск товаров с заданным названием или ценой.
11) Создайте класс «Магазин» с атрибутами название и список товаров. Каждый товар представлен классом «Товар» с атрибутами название, цена и количество. Напишите методы для добавления товара в магазин, удаления товара из магазина и вычисления общей стоимости товаров в магазине. Используйте магический метод __len__ для определения количества товаров в магазине.

12) Создайте класс «Задача» с атрибутами название, описание и статус (выполнена или нет). Напишите методы для изменения статуса задачи на выполненную и вывода информации о задаче в виде «Задача '{название}': {описание}, статус — {статус}». Используйте магический метод __str__ для вывода информации о задаче.
13) Создайте класс «Банк» с атрибутами название и список счетов. Каждый счет представлен классом «Счет» с атрибутами номер и баланс. Напишите методы для добавления счета в банк, удаления счета из банка и вычисления общего баланса всех счетов в банке. Используйте магический метод __len__ для определения количества счетов в банке.
14) Создайте класс «Студент» с атрибутами имя, фамилия, возраст и список оценок. Напишите методы для добавления оценки, вычисления среднего балла и вывода информации о студенте в виде «Студент {имя} {фамилия}, возраст — {возраст}, средний балл — {средний балл}». Используйте магический метод __len__ для определения количества оценок у студента.
15) Создайте класс «Автомобиль» с атрибутами марка, модель, год выпуска и скорость. Напишите методы для увеличения/уменьшения скорости автомобиля, вывода информации об автомобиле в виде «Автомобиль {марка} {модель}, год выпуска — {год}, скорость — {скорость}». Используйте магический метод __eq__ для сравнения скорости двух автомобилей.
16) Создайте класс фрукты Fruit, хранящий информацию о форме фрукта, его цвете и вкусе. Для класса фрукты Fruit создать: — метод подсчета общего веса текущего экземпляра класса; — метод сравнения общего веса данного экземпляра класса с любым другим; — метод вывода полной информации об экземпляре класса в виде таблицы.
17) Реализовать класс — простейший калькулятор, который в качестве методов реализует арифметические операции. Программа на вход принимает комбинацию чисел и операций, разделенных пробелами. Например, 6 – 7 + 4. В качестве операций для работы с числами использовать сложение и вычитание.
18) Создайте класс окружность и класс прямоугольник, а также методы подсчета периметра для этих фигур. Выведите информацию по периметру фигур, используя принцип полиморфизма.
19) Создать класс Дома и класс Рабочие. Класс дома должен содержать атрибуты: этажность, количество подъездов, район, рабочие, которые строят этот объект. Класс Рабочие должен содержать: название строительной компании, квалификацию рабочего. Создать объекты класса Дома с указанием, в какие сроки планирует-

ся осуществить постройку и каким коллективом рабочих. Определить в скольких постройках рабочий задействован в один год одновременно. Данные по датам оформить в виде формата вида 01.01.2022

20) Создать родительский класс Сотрудник магазина, который содержит метод вывода информации о сотруднике (имя, зарплата, стаж). Создать дочерний класс Кассир и дочерний класс Мерчендайзер, которые также имеют возможность вывода информации (имя, зарплата, стаж).
21) Создать родительский класс «Прямоугольник», который содержит методы вычисления площади и периметра прямоугольника. Реализовать дочерний класс «Квадрат» с методами вычисления площади и периметра квадрата.
22) Создать класс Лифт, объекты которого содержат атрибут текущей грузоподъемности. Разработать метод, который сравнивает текущую грузоподъемность лифта с максимальной грузоподъемностью лифта и дает рекомендации по переполнению или недобору грузоподъемности.
23) Создать класс Квадрат, который будет рисовать закрашенный квадрат, зная координаты точек левого верхнего угла квадрата и длину его стороны. Координаты точки левого верхнего угла и длину стороны квадрата сделать приватными.
24) Создать класс Book, который описывает следующие данные о книге: author (автор), title (название), year (год), price (цена). Найти самую дорогую книгу, используя магические методы сравнения.
25) Создайте класс с методами, формирующими вложенную последовательность. Пользователю должна быть предоставлена возможность заполнить ее либо случайными числами в интервале [–10; 10], либо осуществить ввод данных с клавиатуры.
26) Создайте класс Person с методами, позволяющими вывести на экран информацию о персоне, а также определить ее возраст (в текущем году). Создайте дочерние классы: АБИТУРИЕНТ (фамилия, дата рождения, факультет), СТУДЕНТ (фамилия, дата рождения, факультет, курс), ПРЕПОДАВАТЕЛЬ (фамилия, дата рождения, факультет, должность, стаж), со своими методами вывода информации на экран и определения возраста. Создайте список из n персон, выведите полную информацию из базы на экран, а также организуйте поиск персон, чей возраст попадает в заданный диапазон.
27) Создайте класс ТРЕУГОЛЬНИК, заданный длинами двух сторон и угла между ними, с методами вычисления площади и периметра треугольника, а также методом, выводящим информацию о фигуре на экран. Создайте дочерние классы ПРЯМОУГОЛЬНЫЙ, РАВНОБЕДРЕННЫЙ, РАВНОСТОРОННИЙ со своими методами

вычисления площади и периметра. Создайте список n треугольников и выведите полную информацию о треугольниках на экран.

28) Создайте класс ТРАНСПОРТ с методами, позволяющими вывести на экран информацию о транспортном средстве, а также определить грузоподъемность транспортного средства. Создайте дочерние классы АВТОМОБИЛЬ (марка, номер, скорость, грузоподъемность), МОТОЦИКЛ (марка, номер, скорость, грузоподъемность, наличие коляски, при этом если коляска отсутствует, то грузоподъемность равна нулю), ГРУЗОВИК (марка, номер, скорость, грузоподъемность, наличие прицепа, при этом если есть прицеп, то грузоподъемность увеличивается в два раза) со своими методами вывода информации на экран и определения грузоподъемности. Создайте список из n машин, выведите полную информацию на экран, а также организуйте поиск машин, удовлетворяющих требованиям грузоподъемности.
29) Создайте класс ТОВАР с методами, позволяющими вывести на экран информацию о товаре, а также определить, может ли приобрести товар покупатель, имеющий заданную сумму денег. Создайте дочерние классы ПРОДУКТ (название, цена, дата производства, срок годности), ПАРТИЯ (название, цена за штуку, количество штук, дата производства, срок годности), ТЕЛЕФОН (название, цена) со своими методами вывода информации на экран и определения соответствия заданной цене. Создайте список из n товаров, выведите полную информацию из базы на экран, а также организуйте поиск товара, который может приобрести покупатель, имеющий заданную сумму денег.
30) Создайте класс ТОВАР с методами, позволяющими вывести на экран информацию о товаре, а также определить, предназначен ли он для заданного возраста потребителя. Создайте дочерние классы ИГРУШКА (название, цена, производитель, материал, возраст, на который рассчитана), КНИГА (название, автор, цена, издательство, возраст, на который рассчитана), СПОРТИНВЕНТАРЬ (название, цена, производитель, возраст, на который рассчитан) со своими методами вывода информации на экран и определения соответствия возрасту потребителя. Создайте список из *n* товаров, выведите полную информацию из базы на экран, а также организуйте поиск товаров для потребителя в заданном возрастном диапазоне.
31) Создайте класс ТЕЛЕФОННЫЙ_СПРАВОЧНИК с методами, позволяющими вывести на экран информацию о записях в телефонном справочнике, а также определить соответствие записи критерию поиска. Создайте дочерние классы ПЕРСОНА (фамилия, адрес, номер телефона), ОРГАНИЗАЦИЯ (название, адрес, телефон,

факс, контактное лицо), ДРУГ (фамилия, адрес, номер телефона, дата рождения) со своими методами вывода информации на экран и определения соответствия заданной фамилии. Создайте список из n записей, выведите полную информацию из базы на экран, а также организуйте поиск в базе по фамилии.

32) Создайте класс КЛИЕНТ с методами, позволяющими вывести на экран информацию о клиентах банка, а также определить соответствие клиента критерию поиска. Создайте дочерние классы ВКЛАДЧИК (фамилия, дата 269 открытия вклада, размер вклада, процент по вкладу), КРЕДИТОР (фамилия, дата выдачи кредита, размер кредита, процент по кредиту, остаток долга), ОРГАНИЗАЦИЯ (название, дата открытия счета, номер счета, сумма на счету) со своими методами вывода информации на экран и определения соответствия дате (открытия вклада, выдаче кредита, открытия счета). Создайте список из n клиентов, выведите полную информацию из базы на экран, а также организуйте поиск клиентов, начавших сотрудничать с банком в заданную дату.

33) Создайте класс ПРОГРАММНОЕ_ОБЕСПЕЧЕНИЕ с методами, позволяющими вывести на экран информацию о программном обеспечении, а также определить соответствие возможности использования (на текущую дату). Создайте дочерние классы СВОБОДНОЕ (название, производитель), УСЛОВНО_БЕСПЛАТНОЕ (название, производитель, дата установки, срок бесплатного использования), КОММЕРЧЕСКОЕ (название, производитель, цена, дата установки, срок использования) со своими методами вывода информации на экран и определения возможности использования на текущую дату. Создайте список из n видов программного обеспечения, выведите полную информацию из базы на экран, а также организуйте поиск программного обеспечения, которое допустимо использовать на текущую дату.

7. Функциональное программирование

В ходе развития декларативной парадигмы сформировался ***функциональный подход***, в котором процесс вычисления трактуется как вычисление значений функций в их математическом понимании (иногда используется термин «чистая функция»). «Чистая» функция зависит только от ее входных данных и результатов работы других функций, а не от внешнего контекста. Таким образом, явное хранение или изменение состояния программы не предусмотрено, поэтому функциональному программированию, как правило, присущи такие свойства, как:

- отсутствие глобальных и локальных переменных;
- неизменность переменных, т.е. переменную после инициализации нельзя менять, но можно на ее основе создать другую переменную с новым значением;
- функции высших порядок, когда функция может принимать другую функцию в качестве своего аргумента или возвращать другую функцию в качестве результата;
- представление аргументов и результатов работы каждой функции в виде списков.

Одним из самых популярных и наглядных элементов функциональной парадигмы является механизм рекурсии. При рекурсивной реализации для получения результата функция может многократно вызывать саму себя с новым набором аргументов до выполнения определенного стоп-условия, которое позволяет выдать готовый ответ для какого-либо тривиального значения аргумента.

Преимущества функционального программирования во многом связаны именно с отсутствием побочных эффектов при выполнении каждой функции — результат работы зависит исключительно от параметров вызова функции. Это позволяет повысить надежность кода, удобство организации модульного тестирования, дает возможность выполнять эффективное распараллеливание программы в ходе ее исполнения. Однако код на языках функционального программирования чаще всего трудно понимать, а отсутствие механизма изменения существующих переменных приводит к постоянному выделению и освобождению памяти, поэтому обязательным компонентом функциональной программы является встроенный эффективный сборщик мусора.

В настоящее время популярными языками, наиболее полно поддерживающими функциональную парадигму, являются Haskell, Erlang, Scala, а отдельные элементы функциональной парадигмы (например, рекурсивные вызовы функций, неизменность переменных и функции высших порядков) поддерживаются большинством современных языков программирования (в том числе языком Python).

7.1. Основы: функции первого класса, функции высшего порядка, замыкание

Неотъемлемой частью функционального программирования является возможность передачи функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным или сохранение в структурах данных. То есть основная идея заключается в том, что функции (или методы, если речь одет об ООП) имеют тот же статус, что и другие объекты данных. К слову, функции, которые могут передаваться в качестве аргументов, называют ***функциями первого класса***, а функции, которые их могут принимать в качестве аргументов (или возвращать в качестве результата) — ***функциями высшего порядка***.

```
def square(x):
    return x * x

def cube(x):
    return x * x * x

def my_map(func, arg_list):
    result = []
    for i in arg_list:
        result.append(func(i))
    return result

sq = my_map(square, [1, 2, 3, 4])
print(sq)

cb = my_map(cube, [1, 2, 3, 4])
print(cb)
```

```
[1, 4, 9, 16]
[1, 8, 27, 64]
```

Рисунок 7.1 — Использование функции как аргумента

На рис. 7.1 приведен пример использования функций square() и cube() в качестве аргумента при вызове другой функции — my_map(). Здесь важно понимать, что в функцию my_map() в качестве аргумента передается не результаты выполнения функции square() или cube(), а именно сама функция (точнее, указатель на нее). Это позволяет вызывать нужную функцию из тела функции my_map(). Благодаря такому подходу внутри функции my_map() можно вычислить нужную математическую функцию для заданного набора чисел, меняя лишь аргументы вызова функции высшего порядка my_map().

```
def set_func(value):
    base = value

    def add_func(shift):
        temp = base + shift
        return (temp)

    return add_func

f1 = set_func(10)
print(f1)
print(f1(2))

f2 = set_func(100)
print(f2)
print(f2(22))

print(f1(14))
```

```
<function set_func.<locals>.add_func at 0x0000028722CFB5E0>
12
<function set_func.<locals>.add_func at 0x0000028722CFBA60>
122
24
```

Рисунок 7.2 — Использование функции как результата

На рис. 7.2 приведен более сложный пример. Обратите внимание, что здесь внутри функции set_func() описана вложенная функция — add_func(). В то же время нигде не видно, как вызывается эта функция: оператор ***return add_func*** в конце функции set_func() не вызывает функцию add_func() — это следует из отсутствия скобок после имени add_func, а возвращает саму функцию (точнее, ее адрес). Поэтому результатом работы функции set_func() является адрес функции add_func(), т.е. в данном примере функция является возвращаемым значением.

Замыкание — это функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своей области видимости.

Функция замыкания обязательно определена внутри тела другой функции и создаётся каждый раз во время её выполнения. При этом внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит, во-первых, создание нового экземпляра области видимости и, во-вторых, создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции. Важно отметить, что ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если

внешняя функция закончила работу, и переменные вышли из области видимости — в этом и состоит суть замыкания.

Выполнение оператора f1 = set_func(10) приведет к тому, что будет создана область видимости в пределах функции set_func() (она показана фигурной скобкой), и в переменную f1 будет записан адрес функции add_func(). Поэтому вызов оператора print() для f1 приведет к выводу информации об этой функции — видно, что она является локальной функцией для функции set_func().

Поскольку в переменную f1 записан адрес функции, то можно вызвать эту переменную как обычную функцию, указав в скобках необходимый параметр вызова — в данном случае число 2. В результате этого шага будет вызван экземпляр функции add_func(), который соответствует области видимости со значением переменной base, равным 10. Поэтому результатом вызова будет результат операции 10+2. Благодаря замыканию объекты из окружающей области видимости записываются и сохраняются после того, как функция (в данном случае set_func()) закончила свое выполнение. Эти объекты защищены от сборщика мусора.

В следующей строке вызывается функцию set_func() с другим аргументом и записывается результат вызова в переменную f2. Очевидно, что записанное в f2 значение будет совпадать со значением в f1 (и там, и там будет содержаться адрес функции add_func()). Однако при новом вызове set_func() будет создан новый экземпляр области видимости, в котором значение base будет равно 100. Поэтому теперь, при косвенном вызове функции add_func() через переменную f2 получим в результате значение 100 + 22 = 122.

Обратите, внимание, что область видимости, связанная с переменной f1 сохранилась. Если мы захотим снова вызвать функцию add_func() через переменную f1, то получим доступ к первому экземпляру области видимости, где base равно 10.

Замыкание, так же, как и экземпляр объекта, есть способ представления функциональности и данных, связанных и упакованных вместе, только здесь инкапсуляция данных и методов для их обработки осуществляется не на уровне классов (как в объектно-ориентированном программировании), а на уровне функций (функциональное программирование).

7.2. Декораторы

Декоратор — это функция, которая принимает функцию или метод в качестве единственного аргумента и возвращает новую функцию или метод, включающую декорированную функцию или метод, с дополнительными функциональными возможностями.

Существует два виде декорирования функции, зависящих от того, есть ли у декорируемой функции входные параметры или нет.

7.2.1. Декорирование функции без параметров

Рассмотрим процесс ***декорирования функции без параметров***.

Представим процесс декорирования функции my_func, которая в теле функции не содержит return (рис. 7.3).

```
def my_func():
    print(f'Работа функции my_func')
```

```
my_func()
```

```
Работа функции my_func
```

```
def func_decorator(func):
    def wrapper():
        print('_ _ _Действия перед вызовом функии my_func_ _ _')
        func()
        print('_ _ _Действия после выполнения функции my_func_ _ _')
    return wrapper
```

```
my_func = func_decorator(my_func)
my_func()
```

```
_ _ _Действия перед вызовом функии my_func_ _ _
Работа функции my_func
_ _ _Действия после выполнения функции my_func_ _ _
```

Рисунок 7.3 — Декорирование функции без параметров

Декоратор очень похож на замыкание, но у него есть две особенности:

1) В качестве параметра внешняя функция обязательно получает адрес какой-либо «третьей» функции, функционирование которой, собственно, и надо изменить с помощью декоратора. В примере 7.3 в роли такой декорируемой функции выступает функция my_func () с единственным параметром вызова. Первый вызов оператора print() показывает, что эта функция является обычным объектом программы, не связанным с какими-либо другими функциями. При вызове функции my_func() на консоль выводится константная строка «Работа функции my_func».

2) Далее, результат вызова внешней функции func_decorator() записывается не просто в какую-нибудь произвольную переменную, а в переменную, имя которой совпадает с именем той функции, поведение которой необходимо изменить (т.е. декорируемой функции). В примере 7.3 в качестве такой переменной выступает my_func. Получаем, что вначале программы это имя связано с функцией my_func(), а в конце программы — с внутренней функцией wrapper(). Теперь, при вызове в программе функции my_func() сначала будет осуществляться переход на функцию wrapper(), затем вывод строки «Действия перед вызовом функции my_func». После этого из вызванной функции wrapper() вызывается с одним параметром функция my_func (), в которой на консоль выводится «Работа функции my_func» и выполняется вывод строки «Действия после выполнения функции my_func».

Таким образом, получается, что при помощи декоратора изменили поведение функции, имя которой записано в переменную my_func ().

В Python предусмотрена возможность использования декораторов в коде с применением специального синтаксиса с символом **@ (*коммерческое at*)**.

Представим процесс декорирования функции my_func, которая возвращает значение в основной программный код (рис. 7.4).

```
def func_decorator(func):
    def wrapper():
        print('_ _ _Действия перед вызовом функии my_func_ _ _')
        res = func()
        print('_ _ _Действия после выполнения функции my_func_ _ _')
        return res
    return wrapper
```

```
@func_decorator
def my_print():
    return f'Работа функции my_print'
my_print()
```

```
_ _ _Действия перед вызовом функии my_func_ _ _
_ _ _Действия после выполнения функции my_func_ _ _

'Работа функции my_print'
```

Рисунок 7.4 — Декорирование функции без параметров, возвращающей результат

Синтаксис описания функции-декоратора с использованием коммерческого at повышает читаемость кода, однако по сути является ничем иным, как «синтаксическим сахаром», без которого вполне можно обойтись.

7.2.2. Декорирование функции с произвольным числом аргументов

Рассмотрим еще один пример применения декоратора. На рис. 7.5 приведен программный код, в котором одну функцию decor_func() необходимо применить для двух разных функций: display() и display_info(). Вариант с display() работает корректно, а вот вариант с display_info() выдает ошибку. Это связано с тем, что в соответствие с логикой работы декорируемой функции она должна брать параметры из своего локального окружения (области видимости). Однако при создании декоратора для display_info() эти параметры не были переданы в область видимости, так как у функции wrapper() нет аргументов и она не может их передать в функцию orig_f() при вызове последней.

Следовательно, если необходимо через один и тот же декоратор выполнять вызовы разных функций, то у всех этих функций должны быть одинаковые списки параметров вызова и эти аргументы должны «пробрасываться» через декорируемую функцию.

```
def decor_func(orig_f):
    def wrapper():
        print('этот код выполняется перед вызовом {}'.format(orig_f.__name__))
        return orig_f()
    return wrapper

@decor_func
def display():
    print('выполняется display')

@decor_func
def display_info(name, age):
    print('выполняется display info с аргументами ({}, {})'.format(name, age))

display()
display_info('Ivan', 25)
```

```
этот код выполняется перед вызовом display
выполняется display

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-beb6ccb2ad03> in <module>
     14
     15 display()
---> 16 display_info('Ivan', 25)

TypeError: wrapper() takes 0 positional arguments but 2 were given
```

Рисунок 7.5 — Декоратор для различных функций

Существует универсальный вариант разработки декоратора для нескольких функций с различными входными параметрами. В Python можно передать переменное количество аргументов двумя способами:

- в виде списка неименованных (или еще говорят «позиционных» аргументов) — в этом случае они помещаются в кортеж;
- в виде списка именованных аргументов — в этом случае они помещаются в словарь.

Аргументами функции wrapper() стали объекты с именами *args и **kwargs. Аргумент *args собирает позиционные аргументы, а **kwargs — именованные. Например, в вызове: wrapper(1, 'a', x=5, y=None) значение args — это кортеж (1, 'a'), а kwargs — это словарь {'x': 5, 'y': None}. Если позиционных аргументов при вызове функции нет, args — пустой, и если нет именованных аргументов, пустой — kwargs (рис. 7.6).

```
def decor_func(orig_f):
    def wrapper(*args, **kwargs):
        print('этот код выполняется перед вызовом {}'.format(orig_f.__name__))
        return orig_f(*args, **kwargs)
    return wrapper

@decor_func
def display():
    print('выполняется display')

@decor_func
def display_info(name, age):
    print('выполняется display info с аргументами ({}, {})'.format(name, age))

display()
display_info('Ivan', 25)
```

```
этот код выполняется перед вызовом display
выполняется display
этот код выполняется перед вызовом display_info
выполняется display info с аргументами (Ivan, 25)
```

Рисунок 7.6 — Декоратор для различных функций с применением позиционных и именованных аргументов

Благодаря такому изменению способа «пробрасывания» параметров вызываемая через декоратор функция может иметь произвольное число аргументов, то есть можно использовать один и тот же декоратор для функций с разным числом аргументов (рис. 7.7).

```
def my_decorator(func):
    def wrapper(*args, **kwars):
        print('_ _ _Действия перед вызовом функии my_func_ _ _')
        func(*args, **kwars)
        print('_ _ _Действия после выполнения функции my_func_ _ _')
    return wrapper
```

```
@my_decorator
def my_print(param):
    print(f'Функции выводит {param}')
my_print('Hello world')
```

```
_ _ _Действия перед вызовом функии my_func_ _ _
Функции выводит Hello world
_ _ _Действия после выполнения функции my_func_ _ _
```

Рисунок 7.7 — Декоратор для различных функций с применением позиционных и именованных аргументов.

7.2.3. Декоратор @property и функция property()

В Python есть несколько встроенных декораторов, также существуют декораторы в различных разделах стандартной библиотеки Python. С примерами встроенных декораторов, такими как @staticmethod и @classmethod мы уже знакомы. Теперь давайте познакомимся с одним из наиболее часто используемых в ООП декоратором — **@property**.

Декоратор **@property** перед методом класса говорит о том, что создается атрибут для чтения с таким же именем и этот метод возвращает его значение. То есть вместо работы с методом класса получаем возможность работы с атрибутом с тем же именем как с полем данных. Таким образом организуется доступ к внутреннему состоянию объекта, имитирующий обращение к его полю, правда только для чтения.

В последней строке кода мы можем обратиться к методу area экземпляра класса Rectangle как к полю данных (рис. 7.8). То есть не нужно вызывать метод area(). Вместо этого при обращении к area как к атрибуту (то есть без использования скобок), соответствующий метод вызывается неявным образом. Это возможно благодаря декоратору @property, который работает поверх встроенной функции property().

Размещение конструкции @property перед определением функции равносильно использованию конструкции вида area = property(area).

Функция property() — интересная встроенная функция языка Python, которая принимает одну или несколько других функций в качестве аргумента и использует их вместо обращения к полю данных для чтения, записи или удаления — так, как это показано на примере в правой части рис. 7.8.

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @property    # area = property(area)
    def area(self):
        return self.a * self.b

rect = Rectangle(5, 6)
print(rect.area)
```

```
30
```

```
class Pen:
    def __init__(self, new_color):
        self.__color = new_color

    def __getColor(self):
        return self.__color

    def __setColor(self, new_color):
        self.__color = new_color

    color = property(__getColor, __setColor)

pen1 = Pen(5)
print(pen1.color)
pen1.color = 22
print(pen1.color)
```

```
5
22
```

Рисунок 7.8 — Декоратор @property и функция property()

Благодаря функции property() можно оставить методы getColor() и setColor() приватными методами класса Pen и не открывать к ним доступ для сторонних объектов. А вместо этого использовать доступ к искусственному полю color на чтение или на запись для вызова этих методов (рис. 7.8).

7.3. Задачи для самостоятельного решения

Задача 7.1. Решите задачи, используя лямбда-функцию.

1) а) Напишите лямбда-функцию, которая добавляет 15 к заданному числу, переданному в качестве аргумента, а также лямбда-функцию, которая умножает аргумент *x* на аргумент *y*, и выведите результат. б) Найти максимальное и минимальное значения в заданном списке кортежей с помощью лямбда-функции. в) Извлечь строки с заданной длиной из списка строковых значений с помощью лямбда-функции.
2) а) Напишите лямбда-функцию, которая принимает один аргумент, и этот аргумент должен быть умножен на неизвестное заданное число. б) Найти числа, делящиеся на девятнадцать или тринадцать, из списка чисел, используя лямбда-функцию. в) Напишите программу для подсчёта вещественных чисел в заданном смешанном списке с помощью лямбда-функции.
3) а) Используя лямбда-функцию, отсортировать список кортежей. *Пример*: [('English', 88), ('Science', 90), ('Maths', 97), ('Social sciences', 82)] →[('Social sciences', 82), ('English', 88), ('Science', 90), ('Maths', 97)]. б) Найдите палиндромы в заданном списке строк с помощью лямбда-функции. в) Найти индекс и значение максимального и минимального значенийв заданном списке чисел.
4) а) Используя лямбда-функцию, отсортировать список словарей. *Пример*: [{'make': 'Nokia', 'model': 216, 'color': 'Black'}, {'make': 'Mi Max', 'model': '2', 'color': 'Gold'}, {'make': 'Samsung', 'model': 7, 'color': 'Blue'}] → [{'make': 'Nokia', 'model': 216, 'color': 'Black'}, {'make': 'Samsung', 'model': 7, 'color': 'Blue'}, {'make': 'Mi Max', 'model': '2', 'color': 'Gold'}]. б) Найти все анаграммы в заданном списке строк с помощью лямбда- функции. *Пример*: 'abcd', ['bcda', 'abce', 'cbda', 'cbea', 'adcb'] → ['bcda', 'cbda', 'adcb']. в) Напишите программу для сортировки заданного смешанного списка целых чисел и строк с помощью лямбда-функции. Числа должны быть отсортированы перед строками. *Пример*: [19, 'red', 12, 'green', 'blue', 10, 'white', 'green', 1] → [1, 10, 12, 19, 'blue', 'green', 'green', 'red', 'white'].

5) а) Отфильтровать список целых чисел на чётные и нечётные числа с помощью лямбда-функции. б) Для удаления определённых слов из заданного списка используйте лямбда-функцию. *Пример*: дан список слов ['orange', 'red', 'green', 'blue', 'white', 'black'], удалить слова ['orange', 'black'] → ['red', 'green', 'blue', 'white']. в) Используя лямбда-функцию, проверить, отсортирован ли указанный список или нет.

6) а) Возвести в квадрат и куб каждое число в заданном списке целых чисел с помощью лямбда-функции. б) Умножьте каждое число заданного списка на заданное число, при это используйте лямбда-функцию. в) Используйте лямбда-функцию для извлечения *n*-го элемента кортежа из заданного списка кортежей. *Пример*: дан список кортежей [('Greyson Fulton', 98, 99), ('Brady Kent', 97, 96), ('Wyatt Knott', 91, 94), ('Beau Turnbull', 94, 98)], при $n = 0$ → ['Greyson Fulton', 'Brady Kent', 'Wyatt Knott', 'Beau Turnbull'], при $n = 2$ → [99, 96, 94, 98].

7) а) Используя лямбда-функцию, найти, начинается ли данная строка с заданного символа. б) Суммируйте длину имён заданного списка после удаления имён, начинающихся со строчной буквы. Используйте лямбда-функцию. *Пример*: ['sally', 'Dylan', 'rebecca', 'Diana', 'Joanne', 'keith'] → 16. в) Удалите определённые слова из заданного списка с помощью лямбда-функции.

8) а) Используя лямбда-функцию, из текущего времени извлеките год, месяц, дату и время. б) Вычислите сумму положительных и отрицательных чисел заданного списка чисел, используя лямбда-функцию. в) Для подсчёта вхождений элементов в заданный список используйтелямбда-функцию. *Пример*: [3, 4, 5, 8, 0, 3, 8, 5, 0, 3, 1, 5, 2, 3, 4, 2] → {3: 4, 4: 2, 5: 3, 8: 2, 0: 2, 1: 1, 2: 2}.

9) а) С помощью лямбда-функции проверьте, является ли данная строка числом или нет. б) Напишите программу для поиска чисел в заданном диапазоне, где каждое число делится на каждую цифру, которую оно содержит. *Пример*: задан диапазон от 1 до 22 → [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]. в) Удалите все элементы из заданного списка, присутствующие в другомсписке. *Пример*: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [2, 4, 6, 8] → [1, 3, 5, 7, 9, 10].

10) а) Вывести ряд Фибоначчи до *n* чисел, используя лямбда-функцию. б) Найти следующее после заданного число, полученное перестановкой цифрзаданного числа. *Примеры*: 12 → 21, 10 → False, 201 → 210, 102 → 120, 445 → 454. в) Напишите программу для поиска элементов заданного списка строк,содержащих определённую подстроку, используя лямбда-функцию. *Пример*: ['red', 'black', 'white', 'green', 'orange']; ack → ['black']; abc → [].

11) а) Найти пересечение двух заданных списков с помощью лямбда-функции. *При-*

мер: [1, 2, 3, 5, 7, 8, 9, 10], [1, 2, 4, 8, 9] → [1, 2, 8, 9]. б) Найти список с максимальной и минимальной длиной, используялямбда-функцию. *Пример*: [[0], [1, 3], [5, 7], [9, 11], [13, 15, 17]] → (3, [13, 15, 17]), (1, [0]). в) Найти вложенные элементы списков, которые присутствуют в другомсписке с помощью лямбда-функции. *Пример*: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], [[12, 18, 23, 25, 45], [7, 11, 19, 24, 28], [1, 5, 8, 18, 15, 16]] → [[12], [7, 11], [1, 5, 8]].

12) а) Используя лямбда-функцию, в заданном массиве отсортируйте числа следующим образом: сначала идут положительные числа по возрастанию, затем отрицательные — также по возрастанию. б) Напишите программу для сортировки каждого подсписка строк в заданном списке списков. Используйте лямбда-функцию. в) Вычислить произведение элементов заданного списка чисел с помощью лямбда-функции.
13) а) Подсчитайте чётные и нечётные числа в заданном списке целых чисел, используя лямбда-функцию. б) Отсортируйте заданный список списков по длине и значению, используя при этом лямбда-функцию. *Пример*: [[2], [0], [1, 3], [0, 7], [9, 11], [13, 15, 17]] → [[0], [2], [0, 7], [1, 3], [9, 11], [13, 15, 17]]. в) Используйте лямбда-функцию для вычисления среднего значения чисел в заданном кортеже кортежей. *Примеры*: ((10, 10, 10), (30, 45, 56), (81, 80, 39), (1, 2, 3)) → (30.5, 34.25, 27.0); ((1, 1, –5), (30, –15, 56), (81, –60, –39), (–10, 2, 3)) → (25.5, –18.0, 3.75).
14) а) С помощью лямбда-функции в заданном списке слов найти слова,длина которых равна шести. б) Найти максимальное значение в заданном гетерогенном списке с помощью лямбда-функции. *Пример*: ['Python', 3, 2, 4, 5, 'version'] → 5. в) Удалите значения None из заданного списка с помощью лямбда-функции. *Пример*: [12, 0, None, 23, None, –55, 234, 89, None, 0, 6, –12] → [12, 0, 23, –55, 234, 89, 0, 6, –12].
15) а) Сложите два заданных списка с помощью лямбда-функции и функции map. б) Используйте лямбда-функцию для сортировки данной матрицы в порядке возрастания по сумме элементов её строк. *Пример*: [[1, 2, 3], [2, 4, 5], [1, 1, 1]] → [[1, 1, 1], [1, 2, 3], [2, 4, 5]]; [[1, 2, 3], [–2, 4, –5], [1, –1, 1]] → [[–2, 4, –5], [1, –1, 1], [1, 2, 3]]. в) Напишите программу для сортировки заданного списка строк (чисел) численно с помощью лямбда-функции. *Пример*: ['4', '12', '45', '7', '0', '100', '200', '–12', '–500'] → ['–500', '–12', '0', '4', '7', '12', '45', '100', '200'].

Задача 7.2. Решите задачу, используя функцию map().

1) Утройте все числа из заданного списка целых чисел, используя функцию map.
2) Создайте списки, содержащие определенные элементы из кортежа, и преобразуйте строковые значения в целое число. *Пример*: [('Alberto Franco', '15/05/2002', '35kg'), ('Gino Mcneill', '17/05/2002', '37kg'), ('Ryan Parkes', '16/02/1999', '39kg'), ('Eesha Hinton', '25/09/1998', '35kg')] → ['Alberto Franco', 'Gino Mcneill', 'Ryan Parkes', 'Eesha Hinton']; ['15/05/2002', '17/05/2002', '16/02/1999', '25/09/1998']; [35, 37, 39, 35].
3) Сложите три заданных списка, используя map и лямбда-функцию.
4) Сложите два заданных списка и найдите разницу между ними, используя функцию map. *Пример*: [6, 5, 3, 9], [0, 1, 7, 7] → [(6, 6), (6, 4), (10, –4), (16, 2)].
5) С помощью функции map преобразуйте каждую строку из заданного списка строк в список символов. *Пример*: ['Red', 'Blue', 'Black', 'White', 'Pink'] → [['R', 'e', 'd'], ['B', 'l', 'u', 'e'], ['B', 'l', 'a', 'c', 'k'], ['W', 'h', 'i', 't', 'e'], ['P', 'i', 'n', 'k']].
6) С использованием функции map создайте список, содержащий заданные числа, возведенные в соответствующую степень. *Пример*: bases = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100], exponents = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] → result = [10, 400, 27000, 2560000, 312500000, 46656000000, 8235430000000, 1677721600000000, 387420489000000000, 10000000000000000000].
7) Разделите заданный словарь списков на список словарей, используя функцию map. *Пример*: {'Science': [88, 89, 62, 95], 'Language': [77, 78, 84, 80]} → [{'Science': 88, 'Language': 77}, {'Science': 89, 'Language': 78}, {'Science': 62, 'Language': 84}, {'Science': 95, 'Language': 80}].
8) Преобразуйте заданный список целых чисел и кортеж целых чисел в список строк. *Пример*: [1, 2, 3, 4], (0, 1, 2, 3) → ['1', '2', '3', '4'], ('0', '1', '2', '3').
9) Преобразуйте заданный список строк в список списков с помощью функции map. *Пример*: ['Red', 'Green', 'Black', 'Orange'] → [['R', 'e', 'd'], ['G', 'r', 'e', 'e', 'n'], ['B', 'l', 'a', 'c', 'k'], ['O', 'r', 'a', 'n', 'g', 'e']].
10) Преобразуйте заданный список кортежей в список строк с помощью функции map. *Пример*: [('red', 'pink'), ('white', 'black'), ('orange', 'green')] → ['red pink', 'white black', 'orange green']; [('Sheridan', 'Gentry'); ('Laila', 'Mckee'), ('Ahsan', 'Rivas'), ('Conna', 'Gonzalez')] → ['Sheridan Gentry', 'Laila Mckee', 'Ahsan Rivas', 'Conna Gonzalez'].
11) Напишите программу для подсчета пар одинаковых значений в двух заданных списках. Используйте функцию map. *Пример*: [1, 2, 3, 4, 5, 6, 7, 8], [2, 2, 3, 1, 2, 6, 7, 9] → 4.

12) Найдите соотношение положительных чисел, отрицательных чисел и нулей в массиве целых чисел. *Пример*: [0, 1, 2, –1, –5, 6, 0, –3, –2, 3, 4, 6, 8] → (0.54, 0.31, 0.15); [2, 1, 2, –1, –5, 6, 4, –3, –2, 3, 4, 6, 8] → (0.69, 0.31, 0.0).
13) Используя функцию map(), переведите все символы заданной последовательности в верхний и нижний регистр и удалите повторяющиеся буквы. *Пример*: 'a', 'b', 'E', 'f', 'a', 'i', 'o', 'U', 'a' → {('E', 'e'), ('B', 'b'), ('U', 'u'), ('I', 'i'), ('A', 'a'), ('O', 'o'), ('F', 'f')}.
14) Вычислите квадраты первых *N* чисел Фибоначчи, используя функцию map, и сгенерируйте из них список чисел. Пример: *N* = 10 → [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] → [0, 1, 1, 4, 9, 25, 64, 169, 441, 1156].
15) Возведите в квадрат элементы списка с помощью функции map.

Задача 7.3. Используйте при решении задачи функции filter() и reduce().

1) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все нечетные числа и упорядочить их по убыванию. Используя функцию reduce(), найти их произведение.
2) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все положительные числа и упорядочить их по возрастанию. Используя функцию reduce(), найти их сумму.
3) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все положительные числа и упорядочить их по убыванию. Используя функцию reduce(), найти их произведение.
4) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все четные числа и упорядочить их по возрастанию. Используя функцию reduce(), найти их сумму.
5) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа больше заданного числа и упорядочить их по убыванию. Используя функцию reduce(), найти их произведение.
6) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа больше 10 и упорядочить их по возрастанию. Используя функцию reduce(), найти их сумму.
7) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа кратные 5 и упорядочить их по убыванию. Используя функцию reduce(), найти их произведение.

8) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа меньше заданного числа и упорядочить их по возрастанию. Используя функцию reduce(), найти их произведение.
9) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа меньше 15 и упорядочить их по убыванию. Используя функцию reduce(), найти их сумму.
10) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа кратные 3 и упорядочить их по возрастанию. Используя функцию reduce(), найти их произведение.
11) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа кратные заданному числу и упорядочить их по убыванию. Используя функцию reduce(), найти их сумму.
12) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все отрицательные числа и упорядочить их по возрастанию. Используя функцию reduce(), найти их произведение.
13) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа на нечетных позициях и упорядочить эти числа по убыванию. Используя функцию reduce(), найти их сумму.
14) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все двузначные числа и упорядочить эти числа по возрастанию. Используя функцию reduce(), найти их произведение.
15) Сгенерировать список целых чисел. С помощью функции filter() выбрать из списка все числа на четных позициях и упорядочить эти числа по возрастанию. Используя функцию reduce(), найти их сумму.

Задача 7.4. Решите задачу на создание декоратора.

1) Создайте декоратор, который будет выводить время выполнения функции и сохранять его в файл.
2) Создайте декоратор, который будет выводить на экран результат выполнения функции.
3) Создайте декоратор, который будет выводить на экран аргументы функции и их типы.
4) Создайте декоратор, который будет выводить на экран имя функции и модуль, где она определена.

5) Создайте декоратор, который будет выводить на экран количество вызовов функции за определенный период времени.
6) Создайте декоратор, который будет выводить на экран статистику вызовов функции и сохранять ее в базе данных.
7) Создайте декоратор, который будет кэшировать результаты выполнения функции и очищать кэш при превышении заданного размера.
8) Создайте декоратор, который будет логировать ошибки, возникающие при выполнении функции, и отправлять уведомления об этих ошибках.
9) Создайте декоратор, который будет проверять аргументы функции на корректность и выбрасывать исключение при обнаружении некорректных данных.
10) Создайте декоратор, который будет проверять возвращаемое значение функции на корректность и заменять его на предопределенное значение при обнаружении некорректных данных.
11) Создайте декоратор, который будет заменять исключения, возникающие при выполнении функции, на заданное значение и логировать эти замены.
12) Создайте декоратор, который будет ограничивать время выполнения функции и выбрасывать исключение при превышении этого времени.
13) Создайте декоратор, который будет ограничивать количество вызовов функции за определенный период времени.
14) Создайте декоратор, который будет принимать аргументы и передавать их в функцию в определенном порядке.
15) Создайте декоратор, который будет принимать аргументы и передавать их в функцию в качестве ключевых параметров с заданными значениями по умолчанию.
16) Создайте декоратор, который будет принимать аргументы и передавать их в функцию в качестве позиционных параметров с заданными значениями по умолчанию.
17) Создайте декоратор, который будет принимать аргументы и передавать их в функцию в качестве позиционных параметров и ключевых параметров с заданными значениями по умолчанию.
18) Создайте декоратор, который будет принимать несколько аргументов и передавать их в функцию в определенном порядке.
19) Создайте декоратор, который будет принимать несколько аргументов и передавать их в функцию в качестве ключевых параметров с заданными значениями по умолчанию.

20) Создайте декоратор, который будет принимать несколько аргументов и передавать их в функцию в качестве позиционных параметров с заданными значениями по умолчанию.
21) Создайте декоратор, который будет принимать несколько аргументов и передавать их в функцию в качестве позиционных параметров и ключевых параметров с заданными значениями по умолчанию.
22) Создайте декоратор, который будет принимать список аргументов и передавать его в функцию в определенном порядке.
23) Создайте декоратор, который будет принимать словарь аргументов и передавать его в функцию с заданными значениями по умолчанию.
24) Создайте декоратор, который будет принимать список и словарь аргументов и передавать их в функцию в определенном порядке и с заданными значениями по умолчанию.
25) Создайте декоратор, который будет заменять значение аргумента на заданное значение только если оно удовлетворяет определенному условию.
26) Создайте декоратор, который будет заменять значение аргумента на результат выполнения заданной функции только если оно удовлетворяет определенному условию.
27) Создайте декоратор, который будет заменять значение аргумента на результат выполнения заданной функции с использованием других аргументов функции только если оно удовлетворяет определенному условию.
28) Создайте декоратор, который будет заменять значение аргумента на результат выполнения заданной функции с использованием других аргументов функции и некоторого контекста только если оно удовлетворяет определенному условию.
29) Создайте декоратор, который будет заменять значение аргумента на результат выполнения заданной функции с использованием других аргументов функции и контекста извне только если оно удовлетворяет определенному условию.
30) Создайте декоратор, который будет заменять значение аргумента на результат выполнения заданной функции с использованием других аргументов функции и контекста извне, если условие выполняется и логировать эти замены.

8. Структуры данных: массивы, стеки, очереди, списки

8.1. Введение в анализ сложности алгоритмов

В данном разделе рассматриваются основные структуры данных, широко используемые в программировании. Особое внимание уделяется массивам, стекам, очередям и спискам. Хотя язык программирования Python используется в качестве основного языка, эти понятия являются универсальными и фундаментальными для большинства современных языков программирования. Понимание их работы и применение в практике программирования являются неотъемлемой частью успешной карьеры разработчика. Однако перед рассмотрением конкретных структур данных необходимо обсудить базовые понятия, связанные с оценкой сложности алгоритмов.

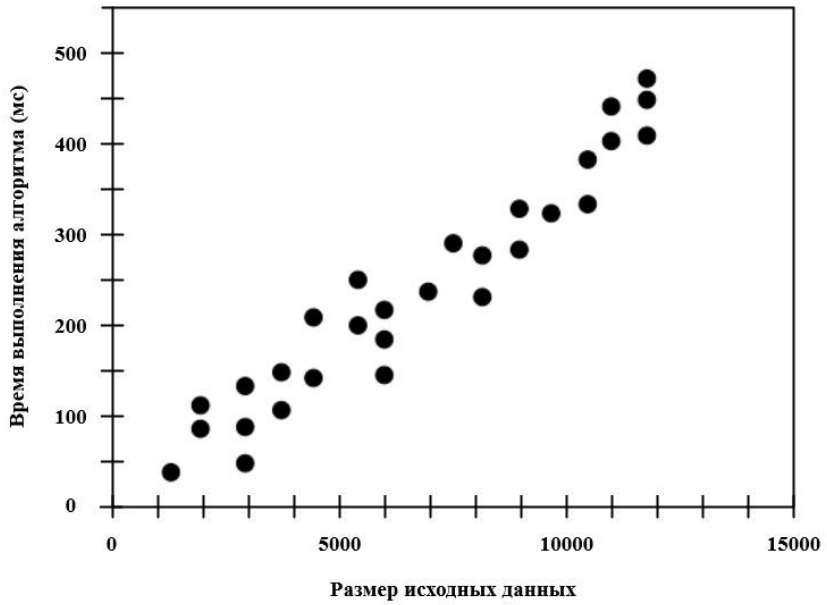


Рисунок 8.1 — Зависимость времени выполнения алгоритма от размера входных данных

Представим себе задачу, которую нужно решить с помощью программирования, и мы хотим оценить скорость работы алгоритма. На первый взгляд, это может показаться простой задачей: мы написали программу, запустили ее на определенных данных и замерили время выполнения. Однако оказывается, что это задача далеко не тривиальная. Почему? Потому что время выполнения алгоритма зависит от множества факторов. Например, оно зависит от конкретной реализации алгоритма, используемого языка программирования, аппаратного обеспечения, на котором запускается программа, и, конечно же, от набора данных, на котором алгоритм выполняется. Чем больше данных, тем больше времени тре-

буется для их обработки. Кроме того, время выполнения может зависеть не только от объема данных, но и от их внутренней структуры.

Если провести достаточно много экспериментов, можно получить график, который показывает зависимость времени выполнения от размера входных данных (рис. 8.1). Интересно, что при одном и том же размере данных можно получить разные результаты. Это может быть вызвано различиями во внутренней структуре данных или состоянии компьютера в момент выполнения (например, если параллельно запущены другие системные задачи).

Кроме того, возникает проблема сравнения результатов с результатами других разработчиков. Ведь каждый может использовать разные наборы данных или аппаратное обеспечение. Поэтому экспериментальное сравнение не всегда является лучшим подходом.

Для решения этой проблемы было разработано несколько подходов. Один из них основан на подсчете количества базовых операций в алгоритме. Для оценки сложности алгоритма таким способом не обязательно иметь готовую реализацию на конкретном языке программирования, достаточно иметь описание алгоритма на псевдокоде. Тогда можно подсчитать количество базовых операций для определенной ситуации.

К базовым операциям относятся присвоение значения переменной, выполнение арифметических операций, сравнение чисел, обращение к памяти (например, получение элемента из списка по индексу), вызов функций и т.д. При этом каждая операция может иметь свой вес, так как время выполнения может различаться. Таким образом, даже без запуска алгоритма можно оценить его сложность.

При оценке сложности алгоритма принято концентрироваться на наихудшем случае работы алгоритма, хотя иногда также анализируются средние значения, если у нас есть несколько вариантов входных данных.

При оценке и записи сложности алгоритма в научных исследованиях и учебных пособиях применяется специальная нотация, известная как нотация «О большое» и «О малое» (реже используется нотация «О малое», но, тем не менее, она также имеет свое применение). В курсе математического анализа вы, вероятно, уже сталкивались с этими обозначениями, однако давайте еще раз повторим, что они означают.

Итак, пусть функции $f(n)$ и $g(n)$ отображают положительные целые числа на положительные действительные числа. Тогда $f(n)$ является $O(g(n))$, если существует действительная константа $c > 0$ и целая константа $n_0 \geq 1$ такая, что $f(n) \leq cg(n)$, для $n \geq n_0$.

Графическое представление этого определения позволяет лучше понять его суть: в данном случае n представляет собой характеристику входных данных в алгоритм (например,

размер массива), $f(n)$ является оценкой сложности, а $g(n)$ — оценкой. Говорим, что оценка $f(n)$ является «О большое» от $g(n)$, если начиная с некоторого значения n_0, функция $f(n)$ не превышает C, умноженное на $g(n)$, где C — это фиксированное положительное значение. Это означает, что $g(n)$ можно использовать в качестве качественной оценки изменения времени выполнения программы при изменении объема данных. В табл. 11.1 приведены функции, которые чаще всего используются в качестве $g(n)$.

Таблица 8.1 — Функции, используемые для асимптотических оценок сложности алгоритмов

constant	**logarithm**	**linear**	**n-log-n**	**quadratic**	**cubic**	**exponential**
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Временные сложности алгоритмов:

- Константное время: сложность алгоритма является «О большое» от константы. Это означает, что существует константа C такая, что начиная с некоторого значения n_0 и более, время работы алгоритма не будет превышать константу C.
- Логарифмическое время: начиная с некоторого значения n_0, сложность алгоритма растет не быстрее, чем логарифм числа n.
- Линейное время: начиная с некоторого значения n_0, сложность алгоритма, зависящая от объема данных, растет не быстрее, чем линейная функция.
- Функции, связанные с полиномиальным ростом: n, умноженное на логарифм числа n, n в квадрате, в кубе, и т.д. На самом деле можно было бы записать n в степени α.
- Экспоненциальное время: n является показателем степени некоторого положительного значения.

Чтобы понять разницу между этими вариантами, можно рассмотреть графики этих функций в двойном логарифмическом масштабе (рис. 8.2). Они позволяют увидеть, как эти функции меняются в зависимости от объема данных. Для многих задач, где используются большие объемы данных, наиболее часто используются оценки линейно-логарифмической или линейной функции. Более сложные алгоритмы обычно не рассматриваются, поскольку достижение требуемого объема данных становится нереалистичным.

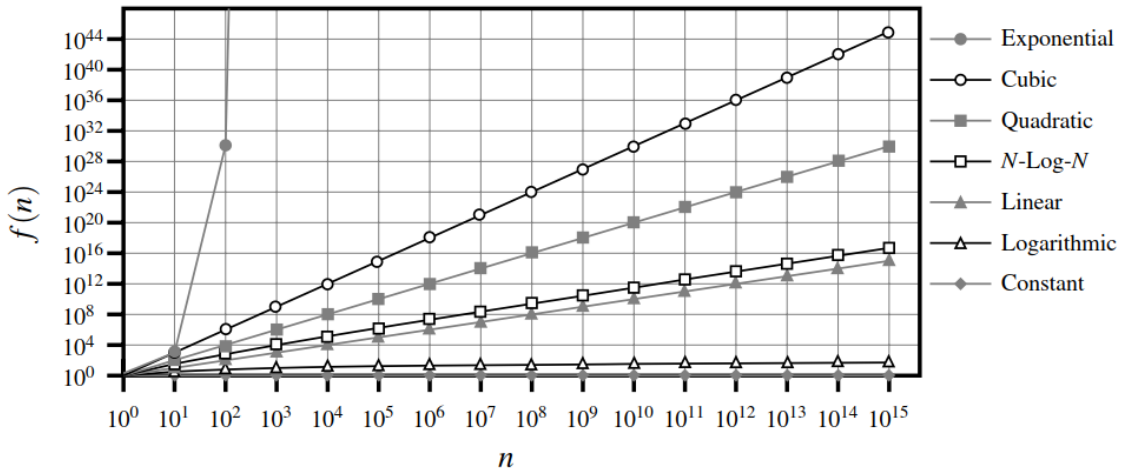


Рисунок 8.2 — Графики функций, используемые для асимптотических оценок сложности алгоритмов

Кроме «О большое» могут встречаться и другие оценки (рис. 8.3). Например, оценка «Ω большое» — это нижняя асимптотическая оценка роста временной функции, а оценка «Θ большое» является одновременно нижней и верхней асимптотической оценкой. Это означает, что функция $f(n)$ ограничена сверху и снизу функцией $g(n)$, умноженной на положительные константы C_1 и C_2. Однако такие оценки встречаются реже. Чаще всего вы будете сталкиваться с оценкой «О большое».

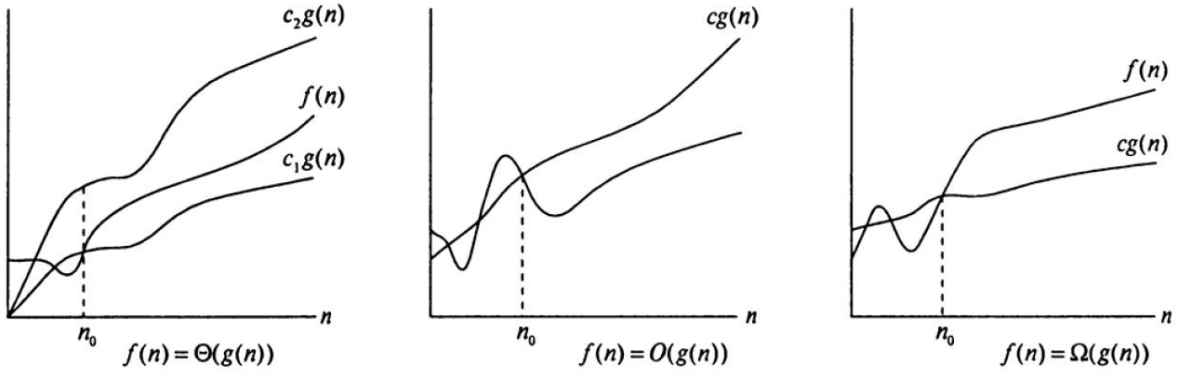


Рисунок 8.3 — Другие виды асимптотических оценок

Таким образом, рассмотрены основные понятия, связанные с оценкой сложности алгоритмов. Теперь можно перейти к изучению структур данных, которые играют важную роль в разработке эффективных алгоритмов.

8.2. Массивы

Одной из основных структур данных в языке программирования Python являются списки. В отличие от массивов, используемых в других языках программирования, списки обладают большей гибкостью. Однако для более полного понимания возможностей языка Python необходимо изучить и массивы. Рассмотрим различия между списками и массивами, а также их преимущества и недостатки.

Прежде чем перейти к обсуждению списков и массивов, вспомним, что такое оперативная память компьютера. С логической точки зрения оперативная память представляет собой набор адресованных ячеек, где каждая ячейка содержит один байт информации (рис. 8.4). Каждой ячейке оперативной памяти соответствует свой адрес, который можно использовать для записи или чтения данных. Память представляет собой непрерывный массив байтов с последовательными адресами.

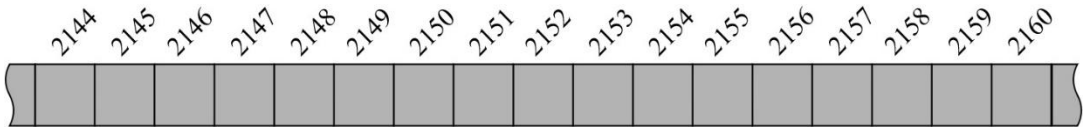


Рисунок 8.4 — Область памяти (RAM)

Теперь рассмотрим строку «SAMPLE» как пример. Высокоуровневая абстракция строки представляет собой неизменяемый список символов, к которым можно обращаться по индексам, начиная с нуля (рис. 8.5).

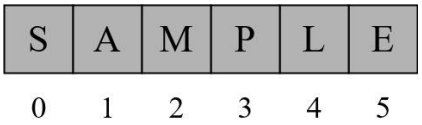


Рисунок 8.5 — Высокоуровневая абстракция строки

Однако фактическое расположение строки в памяти имеет некоторые особенности (рис. 8.6). Начиная с третьей версии языка Python, символы строки хранятся в формате Юникод, который требует два байта для записи одного символа. Это позволяет использовать большинство распространенных алфавитов естественных языков. Если обратиться к символу с индексом ноль, то мы фактически обратимся к двум последовательно расположенным в памяти байтам. Однако все это было бы применимо только в случае хранения строк в виде массивов.

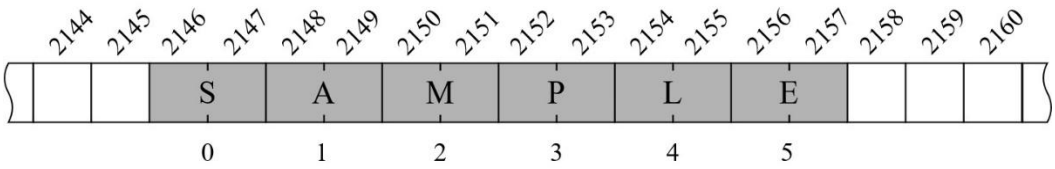


Рисунок 8.6 — Реальное расположение строки в памяти (RAM)

Язык Python предлагает более удобный подход. В Python строки хранятся так же, как и списки, но они не могут изменяться. Каждый элемент списка является ссылкой на хранимое значение. Например, если мы используем список, состоящий из строк (рис. 8.7), то сам список представляет собой набор ссылок, а строки хранятся отдельно (рис. 8.8).

```
list_of_strings = ['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', 'Mary']
```

Рисунок 8.7 — Список строк

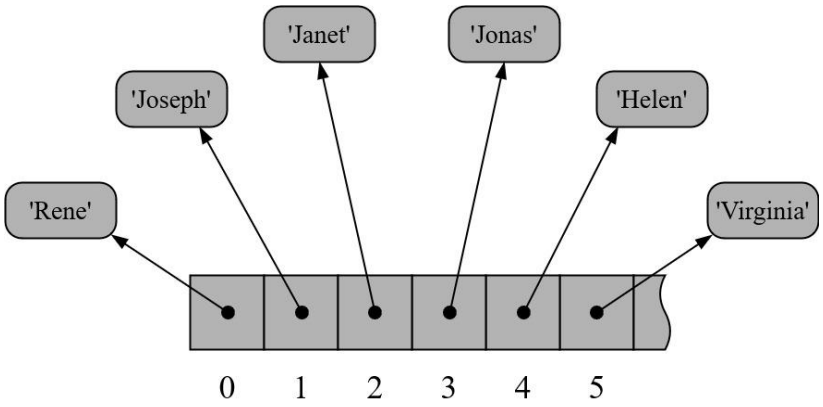


Рисунок 8.8 — Работа массива ссылок

Таким образом, каждый символ строки «SAMPLE» хранится отдельно, занимая два байта, а сама строка представляет собой ссылку на эти символы.

В языке Python также имеется возможность использовать массивы, которые более похожи на массивы в других языках программирования. Для работы с массивами необходимо импортировать модуль array (https://docs.python.org/3/library/array.html). Массив array хранит объекты непосредственно, а не ссылки на них. Это настоящий массив, который обеспечивает более быстрый доступ и меньший объем памяти. Более быстрый доступ обусловлен тем, что нам не нужно проходить через список для обращения к отдельному символу. Экономия памяти достигается за счет отсутствия необходимости хранить ссылки и объекты отдельно. Например, использование массива для хранения целочисленных данных может сэкономить до 4–5 раз больше памяти по сравнению со списком. Однако использование массивов сопряжено с рядом ограничений, которые рассмотрим далее. В большинстве случаев на практике предпочтительно использовать списки, а не массивы.

На рис. 8.9 представлен способ создания массива с использованием модуля array. Для создания массива мы обращаемся к конструктору array и передаем ему тип элементов массива, так же, как это делается в других языках программирования. Размер элементов вычисляется автоматически на основе переданных данных. Важно отметить, что array является динамическим массивом, что означает, что он может изменять свой размер в процессе выполнения программы. Доступ к элементам массива осуществляется по индексу, аналогично доступу к элементам списка.

```
import array
```

```
ar1 = array.array('i' , [2, 3, 5, 7, 11, 13, 17, 19, 21, 25])
len(ar1)
```

```
10
```

```
ar1[0], ar1[4]
```

```
(2, 11)
```

```
import sys
sys.getsizeof(ar1)
```

```
104
```

```
ls2 = [2, 3, 5, 7, 11, 13, 17, 19, 21, 25]
sys.getsizeof(ls2), len(ls2)
```

```
(136, 10)
```

```
sys.getsizeof(2), sys.getsizeof(777)
```

```
(28, 28)
```

```
sys.getsizeof(ls2) + len(ls2) * sys.getsizeof(2)
```

```
416
```

Рисунок 8.9 — Работа с массивом array

Теперь рассмотрим преимущества использования массива. Для этого воспользуемся методом getsizeof() модуля sys, который позволяет нам определить размер элементов в памяти в байтах. Из результатов видно, что массив ar1 занимает 104 байта. Если мы сравним это с аналогичным списком, то увидим, что список занимает 144 байта. На первый взгляд разница не кажется существенной. Однако следует помнить, что в списке хранятся ссылки на числа, а не сами числа. Сами числа хранятся отдельно. Если мы рассмотрим размер отдельного целого числа, например, число 2 или число 777, то увидим, что каждое из них занимает в памяти 28 байтов. Таким образом, весь список занимает в памяти 424 байта, что в 4 раза больше, чем массив. Это говорит о том, что использование массива позволяет сэкономить память и повысить эффективность программы.

Массив array дает достаточно много функциональных возможностей. Ниже приведен список его основных методов.

- array.typecode — TypeCode символ, использованный при создании массива;
- array.itemsize — размер в байтах одного элемента в массиве;
- array.append(x) — добавление элемента в конец массива;
- array.buffer_info() — кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций;
- array.byteswap() — изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов;
- array.count(x) — возвращает количество вхождений x в массив;
- array.extend(iter) — добавление элементов из объекта в массив;
- array.frombytes(b) — делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве;
- array.fromfile(F, N) — читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив;
- array.fromlist(список) — добавление элементов из списка;
- array.index(x) — номер первого вхождения x в массив;
- array.insert(n, x) — включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива;
- array.pop(i) — удаляет i-й элемент из массива и возвращает его. По умолчанию удаляется последний элемент;
- array.remove(x) — удалить первое вхождение x из массива;
- array.reverse() — обратный порядок элементов в массиве;
- array.tobytes() — преобразование к байтам;
- array.tofile(f) — запись массива в открытый файл;
- array.tolist() — преобразование массива в список.

Одним из наиболее интересных методов, доступных при работе с массивами, является метод append(). Этот метод позволяет добавлять новые элементы в конец массива. Особенностью данного метода является то, что добавляемый элемент должен быть того же типа, что и уже существующие элементы в массиве. Это обеспечивает согласованность типов данных внутри массива и позволяет эффективно использовать его функциональные возможности.

Еще одним полезным методом является метод count(). Этот метод похож на аналогичный метод, используемый при работе со списками. Он возвращает количество вхождений определенного элемента в массиве. Таким образом, можно легко определить, сколько раз конкретный элемент встречается в массиве и использовать эту информацию для дальнейшей обработки данных.

Метод insert() предоставляет возможность добавить новый элемент в массив в заданную позицию. Это особенно полезно, когда требуется вставить элемент на определенное место в массиве, не нарушая порядок следования других элементов. Такой подход позволяет более гибко управлять структурой массива и упрощает его дальнейшую обработку.

Метод pop() является еще одной важной функцией, доступной при работе с массивами. Этот метод позволяет удалить элемент из массива и вернуть его значение. Если не указать параметр, то удаляется последний элемент массива. Это удобно, когда требуется удалить конкретный элемент из массива и использовать его значение в дальнейшей программе.

Остальные методы, доступные для работы с массивами, можно рассмотреть самостоятельно. Они предоставляют дополнительные возможности для манипуляции данными в массиве и могут быть полезны в различных ситуациях.

8.3. Динамические массивы

Теперь рассмотрим динамический массив, который представляет собой важный случай структуры данных array. В отличие от других языков программирования, где обычно используются статические массивы с указанием типа элементов и их количества при создании, динамические массивы позволяют изменять свой размер в процессе выполнения программы. Чтобы проиллюстрировать это, рассмотрим пример (рис. 8.10).

Предположим, имеется список, и необходимо изменить его размер до значения n. Переменная a хранит количество элементов в списке, а переменная b — текущий размер списка в байтах. Выводим текущее состояние списка на экран, а затем добавляем в него новый элемент None. После выполнения этого кода мы можем наблюдать, как меняется размер списка. Когда список пустой, он уже занимает 56 байтов. С добавлением одного элемента его размер увеличивается до 88 байтов, и дальше размер остается неизменным для двух, трех и четырех элементов. Однако когда в списке появляется пять элементов, его размер снова увеличивается, и так далее до определенного момента, когда размер списка становится постоянным. Можно наблюдать скачкообразное увеличение размера списка.

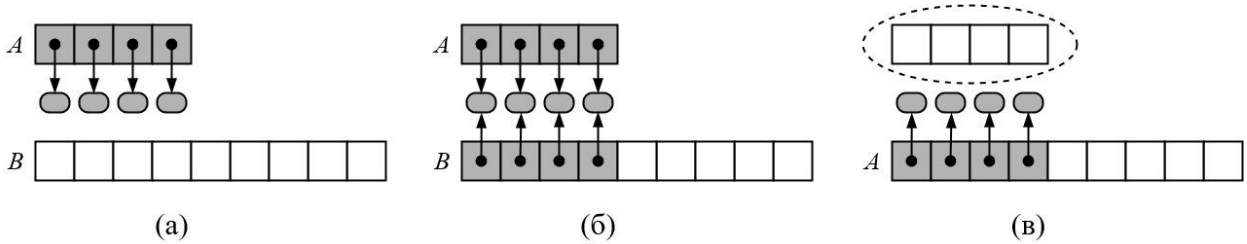
```python
import sys

data = []
n = 33
for k in range(n):
    a = len(data) # количество элементов в списке
    b = sys.getsizeof(data) # текущий размер списка в байтах
    print(f'Length: {a:3d}; Size in bytes: {b:4d}')
    data.append(None) # добавляем элемент в список
```

```
Length:   0; Size in bytes:   56
Length:   1; Size in bytes:   88
Length:   2; Size in bytes:   88
Length:   3; Size in bytes:   88
Length:   4; Size in bytes:   88
Length:   5; Size in bytes:  120
Length:   6; Size in bytes:  120
Length:   7; Size in bytes:  120
Length:   8; Size in bytes:  120
Length:   9; Size in bytes:  184
Length:  10; Size in bytes:  184
Length:  11; Size in bytes:  184
Length:  12; Size in bytes:  184
Length:  13; Size in bytes:  184
Length:  14; Size in bytes:  184
Length:  15; Size in bytes:  184
Length:  16; Size in bytes:  184
Length:  17; Size in bytes:  256
Length:  18; Size in bytes:  256
Length:  19; Size in bytes:  256
Length:  20; Size in bytes:  256
Length:  21; Size in bytes:  256
Length:  22; Size in bytes:  256
Length:  23; Size in bytes:  256
Length:  24; Size in bytes:  256
Length:  25; Size in bytes:  256
Length:  26; Size in bytes:  336
Length:  27; Size in bytes:  336
Length:  28; Size in bytes:  336
Length:  29; Size in bytes:  336
Length:  30; Size in bytes:  336
Length:  31; Size in bytes:  336
Length:  32; Size in bytes:  336
```

Рисунок 8.10 — Изменение размера динамического массива

Причина такого поведения заключается в том, что выделение памяти для динамического массива выгодно производить не при каждом добавлении элемента, а скачкообразно. На рис. 8.11 представлена схема этого процесса.



(а) (б) (в)

Рисунок 8.11 — Процесс роста динамического массива

Исходный массив состоит из четырех элементов, и в нем хранятся четыре ссылки. Когда хотим добавить пятую ссылку, нельзя сделать это в исходном массиве, поэтому создается новый массив. В новый массив копируются все ссылки из исходного массива, и появляется несколько свободных мест для добавления новых элементов. Таким образом, можно добавить 5, 6, 7 и 8 элементы. Когда требуется добавить 9 элемент, процесс повторяется. После копирования ссылок в новый массив исходный массив больше не нужен, поэтому память для него освобождается.

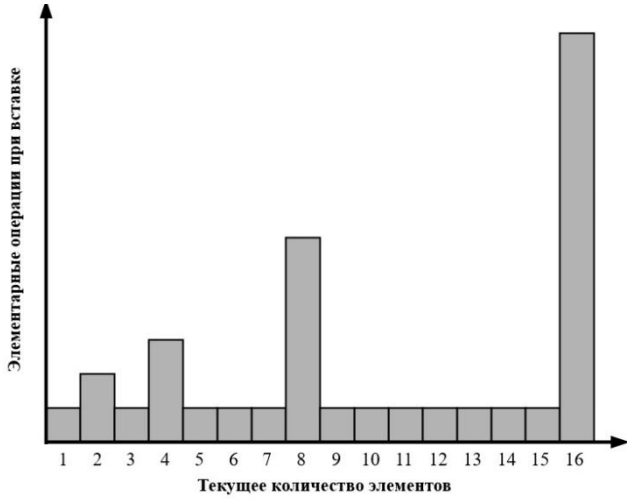


Рисунок 8.12 — Длительность вставки элемента в конец динамического массива

Если рассмотрим время вставки элемента в конец динамического массива, то увидим, что в большинстве случаев вставка происходит быстро, но иногда может занимать больше времени (рис. 8.12). Время, необходимое для выделения памяти для нескольких новых элементов, увеличивается с увеличением размера массива, так как приходится копировать все больше элементов из короткой версии массива в его более длинную версию. Промежутки между долгими вставками не фиксированы и увеличиваются каждый раз.

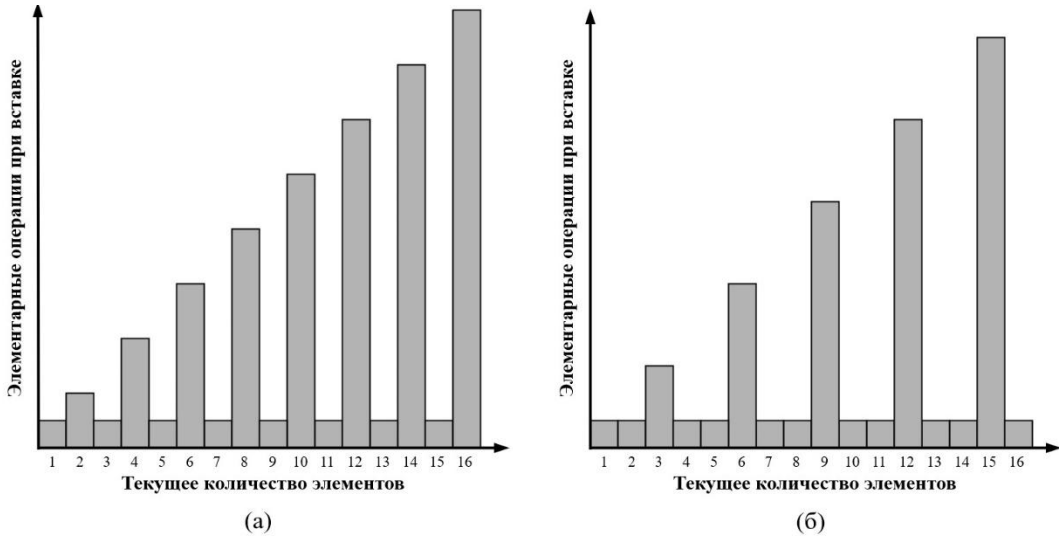


Рисунок 8.13 — Длительность вставки элемента в конец динамического массива при арифметической прогрессии длины массива

Такая стратегия работы с динамическим массивом оказывается эффективнее, чем использование фиксированных промежутков между долгими вставками. Предполагается, что с увеличением размера массива будет выполняться все больше операций по работе с ним. Поэтому при небольших массивах выделяется меньше дополнительного места для будущего хранения новых элементов. Однако с каждым увеличением массива объем выделяемой памяти также увеличивается. Это позволяет сокращать время, необходимое для расширения массива.

В табл. 8.2 представлена сложность выполнения операций работы с массивом. Операция определения длины массива занимает фиксированное время, так как требуется просто получить информацию о количестве элементов в массиве. Это время не зависит от размера массива и составляет O(1).

Таблица 8.2 — Асимптотическая эффективность операций для списка (кортежа), не меняющих содержимое структуры данных

Операция	**Время выполнения**
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k + 1)$
value in data	$O(k + 1)$
data1 == data2 (!=, <, <=, >, >=)	$O(k + 1)$
data[j:k]	$O(k - j + 1)$
data1 + data2	$O(n_1 + n_2)$
c * data	$O(cn)$

Операция извлечения элемента по индексу также осуществляется за фиксированное время. Поскольку тип данных элементов известен и имеется доступ к их адресам, можно легко рассчитать местоположение нужного элемента. Поэтому время выполнения этой операции также составляет $O(1)$.

Операция подсчета количества вхождений в массив элемента с заданным значением требует обхода всего массива. Поэтому время выполнения этой операции линейно зависит от размера массива и составляет $O(n)$, где n — количество элементов в массиве.

Операция нахождения индекса искомого значения также зависит от местоположения этого элемента, обозначаемого как k. Если элемент находится в начале массива, то время выполнения этой операции будет $O(1)$. Однако если элемент находится в конце массива,

то время выполнения будет $O(n)$, так как потребуется пройти все элементы до конца массива, чтобы найти искомое значение.

Проверка вхождения определенного элемента в массив и сравнение значений двух элементов также зависит от их позиции в массиве. Если элементы находятся рядом, то время выполнения этих операций будет $O(1)$. Однако если элементы находятся на значительном удалении друг от друга, время выполнения будет $O(n)$, так как потребуется пройти все элементы до нужного местоположения.

В конце таблицы представлена сложность операций получения среза, сложения двух элементов и умножения элемента на константу. Все эти операции имеют фиксированное время выполнения и не зависят от размера массива.

Таблица 8.3 — Асимптотическая эффективность операций для списка, меняющих содержимое структуры данных

Операция	**Время выполнения**
data[j] = val	$O(1)$
data.append(value)	$O(1)$
data.insert(k, value)	$O(n - k + 1)$
data.pop()	$O(1)$
data.pop(k) del data[k]	$O(n - k)$
data.remove(value)	$O(n)$
data1.extend(data2) data1 += data2	$O(n_2)$
data.reverse()	$O(n)$
data.sort()	$O(n \log n)$

Далее рассмотрим операции изменения значения элемента и добавления элемента в конец массива (табл. 8.3). Обе эти операции имеют фиксированное время выполнения, так как требуется лишь изменить или добавить элемент. Операция вставки элемента в заданную позицию требует сдвига всех элементов после этой позиции. Поэтому время выполнения этой операции линейно зависит от размера массива и составляет $O(n)$, где n — количество элементов в массиве.

Далее рассматриваются различные способы удаления элементов из массива, такие как использование методов pop() и remove(), а также инструкции del. Все эти операции имеют фиксированное время выполнения и не зависят от размера массива. Также оценивается сложность операции расширения массива, обращения к элементу массива и сортировки массива. Операция расширения массива требует выделения нового блока памяти и копи-

рования всех элементов из старого массива в новый. Поэтому время выполнения этой операции линейно зависит от размера массива и составляет $O(n)$, где n — количество элементов в массиве.

Операция обращения к элементу массива также имеет фиксированное время выполнения, так как можно легко получить доступ к элементу по его индексу. Сортировка массива имеет линейно-логарифмическую оценку сложности. Время выполнения этой операции зависит от размера массива и составляет $O(n \log n)$, где n — количество элементов в массиве.

Рассмотрим, как среднее время выполнения операции вставки элемента в массив в позицию k зависит от длины списка и позиции, в которую вставляем элемент. Есть три варианта: можно вставить элемент в начало массива, его середину и в конец.

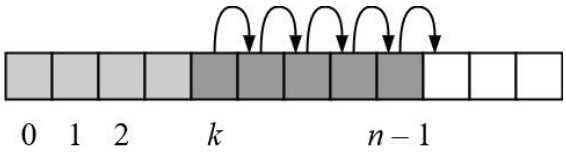


Рисунок 8.14 — Освобождение места для вставки элемента в список

При небольшом размере массива длительность операции почти одинакова и не зависит от того, куда вставляем элемент. Однако с увеличением размера массива длительность операции вставки элемента в начало и середину массива значительно возрастает. Это связано с тем, что перед вставкой элемента необходимо освободить место для его размещения, а значит, сдвинуть все элементы вправо относительно этой позиции. Чем больше массив, тем больше времени потребуется на этот сдвиг.

Таблица 8.4 — Среднее время операции insert(k, val), измеренное в микросекундах (N — длина списка)

	N				
	100	1000	10000	100000	1000000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

8.4. Стек

Стек — это одна из классических структур данных, которая представляет собой коллекцию элементов, управляемую по принципу «последний вошел, первым вышел» (LIFO — last in, first out). Физический аналог стека можно найти в таких объектах, как стопка тарелок или детская игрушка, выдающая конфеты (рис. 8.15).

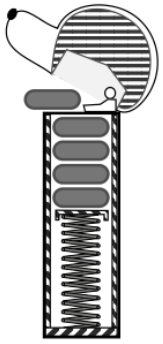


Рисунок 8.15 — Пример физической реализации стека

Базовый стек поддерживает две основные операции — push и pop. Операция push добавляет элемент на вершину стека, а операция pop удаляет верхний элемент стека. Если стек пуст, то при выполнении операции pop будет возвращена ошибка.

Для демонстрации работы стека можно использовать класс deque из модуля collections. Класс deque предоставляет не только функциональность стека, но и другие возможности. Можно создать собственный класс Stack на основе deque и реализовать в нем операцию push, используя метод append (рис. 8.16).

```
from collections import deque

class Stack(deque):
    def push(self, a):
        self.append(a)
```

Рисунок 8.16 — Создание класса Stack

Рассмотрим пример работы со стеком (рис. 8.17). Создадим пустой стек и добавим в него значение 5. Затем при помощи операции push можно добавить элементы 3 и 4. Таким образом, получим стек 5, 3, 4, где элемент 4 был последним добавленным элементом.

```
s1 = Stack()
s1
```

```
Stack([])
```

```
s1.push(5)
s1
```

```
Stack([5])
```

```
s1.push(3)
s1
```

```
Stack([5, 3])
```

```
s1.push(4)
s1
```

```
Stack([5, 3, 4])
```

Рисунок 8.17 —Добавление элементов в стек

Если нужно извлечь элемент из стека, то сначала будет удален элемент из вершины стека. В данном случае, удалим элемент 4. После этого стек будет состоять из двух элементов — 5 и 3. Добавим новый элемент 7 в вершину стека, получив стек 5, 3, 7. Если применим операцию pop и попытаемся определить количество элементов в стеке, то функция len вернет значение 2.

```
s1.pop()
```

```
4
```

```
s1
```

```
Stack([5, 3])
```

```
s1.push(7)
s1
```

```
Stack([5, 3, 7])
```

```
s1.pop(), len(s1)
```

```
(7, 2)
```

Рисунок 8.18 — Работа со стеком

Далее, применив метод pop дважды, удалим два оставшихся элемента из стека. При третьей попытке применить операцию pop, получим сообщение об ошибке, так как стек будет пуст (рис. 8.19).

```
s1.pop()
```

```
3
```

```
s1
```

```
Stack([5])
```

```
s1.pop()
```

```
5
```

```
s1
```

```
Stack([])
```

```
s1.pop()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-41-095118de218b> in <module>
----> 1 s1.pop()

IndexError: pop from an empty deque
```

Рисунок 8.19 — Удаление элементов из стека

Чтобы избежать ошибок при работе со стеком, рекомендуется проверять наличие элементов в вершине стека перед выполнением операций. Также иногда полезно получить значение элемента на вершине стека без его удаления. Для этого можно использовать метод top, который скрывает операции pop и push. Сначала будет удален элемент из вершины стека, затем он будет показан пользователю и возвращен обратно в стек.

Оценим сложность операций работы со стеком, рассмотрев их сопоставление с операциями для работы с обычным списком (рис. 8.5).

Таблица 8.5 — Реализация стека на базе списка

Метод стека	**Реализация на базе списка**
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[–1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Для начала, рассмотрим операцию push. Она эквивалентна методу append, который добавляет элемент в конец списка. Таким образом, операция push имеет сложность $O(1)$, так как добавление элемента в конец списка занимает постоянное время. Следующая операция — pop. Она соответствует удалению элемента из списка. Также, как и в случае с push, сложность этой операции составляет $O(1)$, так как удаление элемента из списка также занимает постоянное время.

Теперь рассмотрим операцию top, которая представляет собой обращение к последнему элементу списка. Сложность этой операции также составляет $O(1)$, так как обращение к определенному элементу списка выполняется за постоянное время. Операция empty проверяет, равна ли длина списка нулю. Эта операция также имеет сложность $O(1)$, так как проверка на равенство длины списка нулю выполняется за постоянное время. Наконец, функция len аналогична определению длины списка. Она также имеет сложность $O(1)$, так как определение длины списка выполняется за постоянное время.

Таблица 8.6 — Производительность реализации стека на базе динамического массива

Метод стека	Реализация на базе списка
S.push(e)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Из табл. 8.6 можно сделать вывод, что все операции работы со стеком имеют фиксированное время выполнения, за исключением операций push и pop, которые зависят от размера структуры. Тем не менее, стек все равно является быстрой структурой данных.

8.5. Очередь

Очередь — это структура данных, которая представляет собой упорядоченную коллекцию элементов, где добавление новых элементов происходит в конец очереди, а извлечение — из начала. Принцип работы очереди заключается в том, что элементы обрабатываются в порядке их добавления: первый пришел, первый обработан (FIFO — first-in, first-out).

Для работы с очередью используются базовые операции — добавление элемента в конец очереди и извлечение первого элемента из очереди. Если очередь пуста, то при попытке извлечения элемента будет выдано сообщение об ошибке.

Для реализации очереди можно использовать класс deque, который предоставляет методы для добавления и удаления элементов как в начало, так и в конец очереди. Создав класс Queue на основе класса deque, можно использовать метод append для добавления элемента в конец очереди и метод popleft для извлечения элемента из начала очереди (рис. 8.20).

```
class Queue(deque):
    def enqueue(self, a):
        self.append(a)

    def dequeue(self):
        return self.popleft()
```

Рисунок 8.20 — Создание класса Queue

Рисунок 8.21 — Работа с очередью

Рассмотрим пример работы с очередью (рис. 8.21). Предположим, есть очередь за билетами, где каждый пришедший человек добавляется в конец очереди. Для начала создадим пустую очередь и последовательно добавим в нее три элемента: 5, 3 и 4. Затем начнем извлекать элементы из очереди. В отличие от стека, где извлекали элементы с конца, в очереди будем извлекать элементы с начала. Так, первым элементом, который будет извлечен, будет 5. После этого добавим новый элемент 7 в конец очереди. Теперь очередь будет содержать элементы 3, 4 и 7. Последовательно применяя операцию удаления элементов из начала очереди, извлечем оставшиеся элементы. В итоге, после нескольких операций удаления, в очереди не останется элементов, и при попытке снова применить операцию удаления элемента из начала очереди возникнет ошибка, так как очередь будет пуста.

Кроме базовых операций, для работы с очередью могут быть полезными и дополнительные операции. Например, операция, которая возвращает первый элемент очереди без его удаления, операция проверки, является ли очередь пустой, и операция нахождения количества элементов в очереди. Эти дополнительные операции позволяют более гибко управлять и использовать очередь в различных ситуациях.

Рассмотрим реализацию очереди на базе массива с использованием динамического массива и закольцованного массива. Для начала, обратим внимание на то, что операция добавления элемента в конец динамического массива выполняется быстро и имеет фиксированное время выполнения. В отличие от добавления в середину или начало массива, которые требуют расширения массива, добавление элемента в конец не требует этого. Однако операция извлечения элемента из начала очереди требует переноса всех элементов на одну позицию влево, что всегда требует n операций. Это является невыгодным.

Одной из идей решения этой проблемы является извлечение элемента массива и освобождение места без сдвига остальных элементов. На рис. 8.22 демонстрируется такая реализация, где указатель f (f от слова first) указывает на первый элемент. Таким образом, ускоряется операция извлечения первого элемента. Однако с течением времени в начале очереди накапливается неиспользуемая область памяти, что увеличивает неэффективное использование памяти. Чтобы избежать этого, можно использовать закольцованный массив.

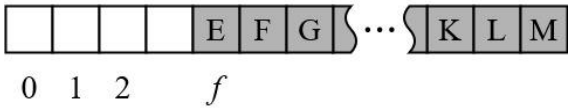


Рисунок 8.22 — Реализация очереди на базе массива: позволяем началу очереди смещаться относительно начала массива

Идея закольцованного массива заключается в выделении фиксированной длины массива для динамической очереди. Когда место для добавления элементов справа заканчивается, начинается использование освободившейся памяти из начала очереди, тем самым создавая зацикленность массива. На рис. 8.23 показано, как сдвигаются элементы в таком массиве. Если регулярно извлекать и добавлять элементы на протяжении жизни очереди, то фиксированного массива должно хватить.

Рисунок 8.23 — Реализация очереди на базе закольцованного массива

Однако если в какой-то момент не хватит места, можно выделить динамический массив большей длины. При этом все элементы будут перенесены в новый массив, а первые элементы будут размещены в самом начале новой области, чтобы обеспечить больше места для новых вставок (рис. 8.24).

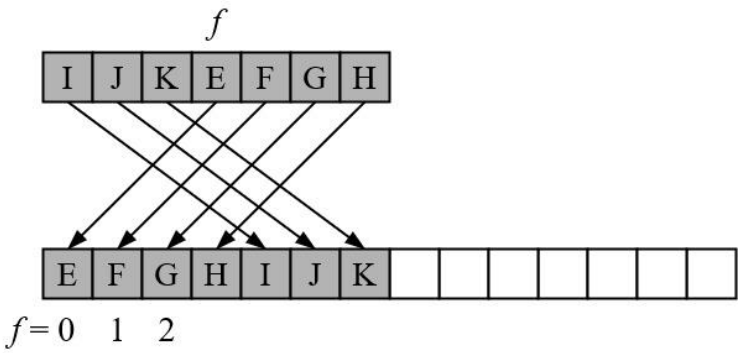


Рисунок 8.24 — Изменение размера закольцованного массива, на базе которого реализована очередь

Из табл. 8.7 видно, что все операции работы с очередью на базе динамического массива имеют фиксированную длительность, за исключением случая, когда требуется увеличить размер массива.

Таблица 8.7 — Производительность реализации очереди на базе динамического массива

Операция	**Время выполнения**
Q.enqueue(e)	$O(1)$
Q.dequeue()	$O(1)$
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

Для реализации классов стека и очереди использовался класс deque из модуля collections (https://docs.python.org/3.1/library/collections.html#collections.deque). Этот класс обеспечивает эффективные операции добавления и удаления значений с начала или конца очереди. Операции с концом очереди (append и pop) аналогичны операциям для списка и выполняются быстро. Также в deque имеются операции с началом очереди, которых нет в списке. Операция appendleft добавляет элемент в левую часть очереди, а операция popleft извлекает элемент из левой части очереди.

Реализация deque основана на двунаправленных связных списках, состоящих из массивов фиксированной длины. deque поддерживает итерацию, операцию получения длины, операцию инвертирования очереди и проверку вхождения элемента с помощью оператора in. Операции получения элемента по индексу быстрые для двух концов очереди, но длительные для элементов в середине списка. Для быстрого произвольного доступа к элементам рекомендуется использовать список вместо deque.

8.6. Связные списки

Связные списки являются одной из основных структур данных, используемых в программировании. Однонаправленный связный список — это простая реализация связного списка, состоящая из узлов. Каждый узел содержит два атрибута: ссылку на хранимые

данные и ссылку на следующий элемент списка. В данной реализации можно двигаться только вправо (рис. 8.25).

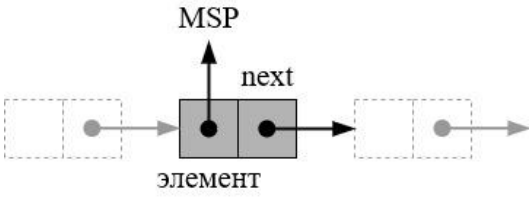


Рисунок 8.25 — Пример узла, являющегося частью однонаправленного связного списка

Для наглядности рассмотрим пример хранения кодов международных аэропортов (рис. 8.26). Данные, на которые ссылается каждый узел, будут размещены в ячейках списка. У структуры также есть атрибут head (голова), указывающий на первый элемент списка, и атрибут tail (хвост), указывающий на последний элемент. Последний элемент не ссылается на другой узел и хранит значение None.

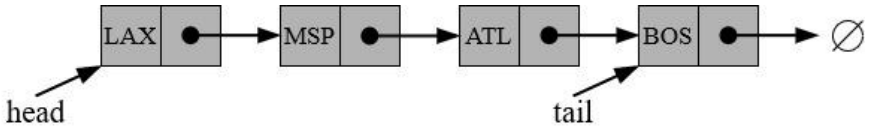


Рисунок 8.26 — Упрощенная иллюстрация однонаправленного связного списка

Рассмотрим базовые операции со связными списками. Первая операция — вставка узла в начало списка. Начальное состояние списка показано на рис. 8.27 под пунктом (а). Для добавления нового узла создаем новый элемент, который ссылается на старый первый элемент, а затем перенаправляем head на новый элемент. Операция вставки не зависит от длины списка, поскольку не требуется перемещать другие элементы. Время выполнения операции фиксировано.

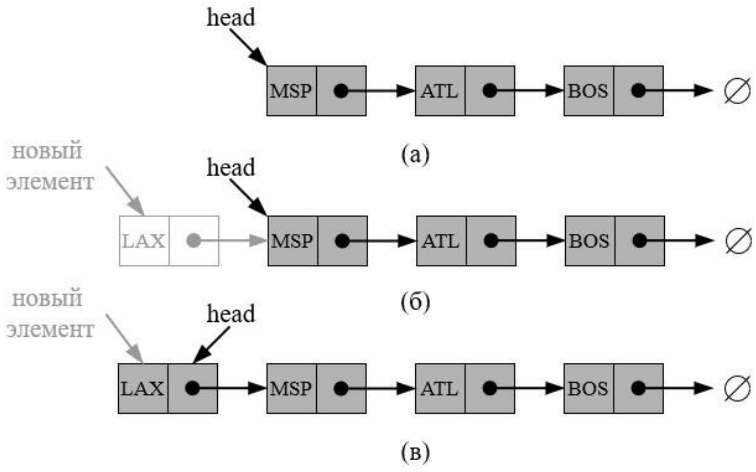


Рисунок 8.27 — Процедура вставки узла в начало однонаправленного связного списка

Теперь рассмотрим вставку узла в конец списка (рис. 8.28). Эта операция также выполняется за фиксированное время, так как последний элемент списка, на который можно быстро получить доступ с помощью ссылки tail, перенаправляется на новый созданный элемент. Таким образом, получается список с добавленным в конец узлом. Время выполнения операции не зависит от длины списка.

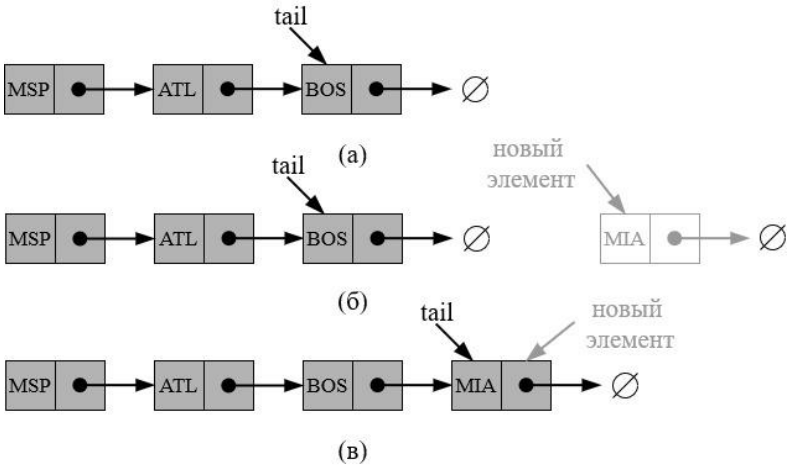


Рисунок 8.28 — Процедура вставки узла в конец однонаправленного связного списка

Удаление узла из начала списка также не представляет проблем (рис. 8.29). Просто переносим head на следующий элемент и удаляем первый узел. При этом нет необходимости обходить весь список, поэтому время выполнения операции также фиксировано.

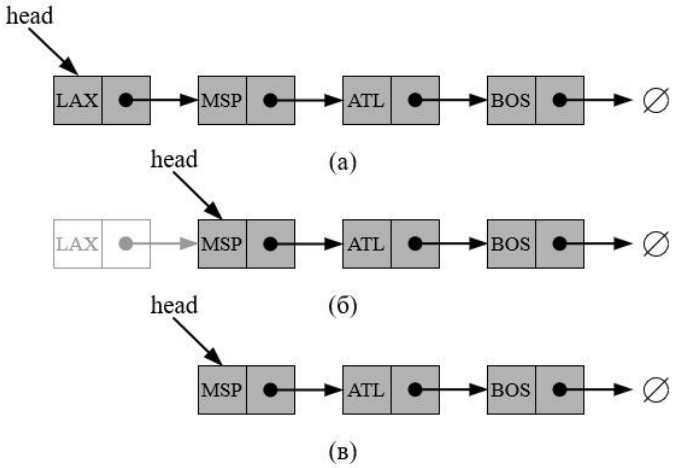


Рисунок 8.29 — Процедура удаления узла из начала

В табл. 8.8 видно, что все рассмотренные операции для однонаправленного связного списка имеют константную оценку сложности. Однако у таких списков есть проблема с доступом к произвольному узлу. Чтобы получить элемент с индексом k, необходимо сделать k переходов от начала списка, и нет способа быстро попасть в середину структуры. Время выполнения операции получения элемента по индексу в однонаправленном связном списке пропорционально длине списка.

Таблица 8.8 — Производительность реализации стека на основе связного списка

Операция	**Время выполнения**
S.push(e)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Существуют и другие варианты реализации связных списков, например, список с циклической структурой. В этом случае последний элемент списка указывает на начало списка, создавая циклическую петлю (рис. 8.30).

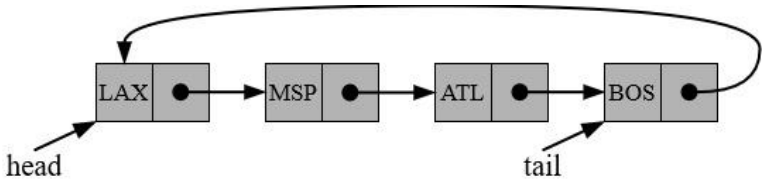


Рисунок 8.30 — Пример однонаправленного связного списка с циклической структурой

Циклические связные списки позволяют легко изменять последовательность элементов, просто сдвигая данные. Это напоминает процесс работы с закольцованными динамическими массивами, но циклические связные списки являются более сложными структурами данных (рис. 8.31).

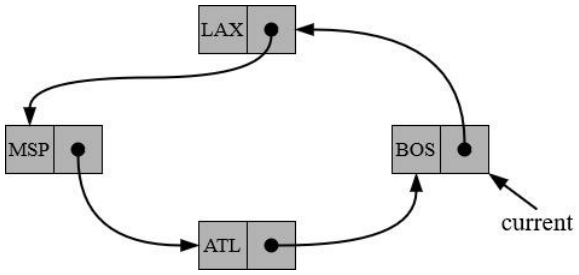


Рисунок 8.31 — Пример однонаправленного связного списка с циклической структурой

Более распространенным и применяемым случаем являются двунаправленные связные (двусвязные) списки (рис. 8.32). В таких списках каждый элемент содержит не только ссылку на следующий элемент, но и на предыдущий элемент. Благодаря этому, имея ссылку на произвольный элемент, можно перемещаться как вправо, так и влево по списку. Однако это увеличивает сложность операций вставки и удаления, так как требуется отслеживать ссылки как влево, так и вправо.

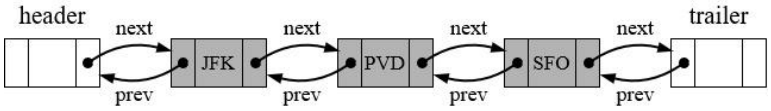


Рисунок 8.32 — Пример двунаправленного связного списка с ограничителями

Рассмотрим операцию добавления узла в двунаправленный связный список (рис. 8.33). После создания нового элемента, необходимо правильно перенаправить ссылки на предыдущий и следующий элементы. Хотя это требует дополнительной работы, оно оправдывается возможностью движения в обоих направлениях по списку.

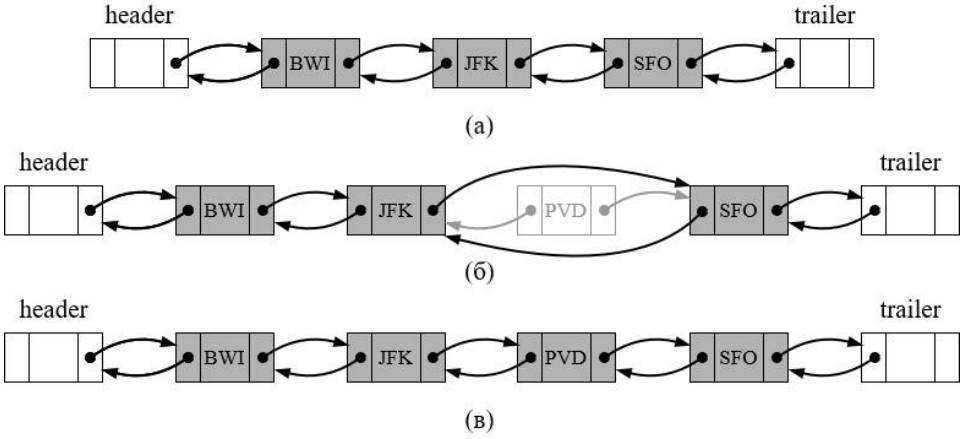


Рисунок 8.33 — Добавление узла в двунаправленный связный список с ограничителями

Теперь рассмотрим процедуру добавления узла в начало двунаправленного связного списка. На рис. 8.34 представлена схема этой процедуры.

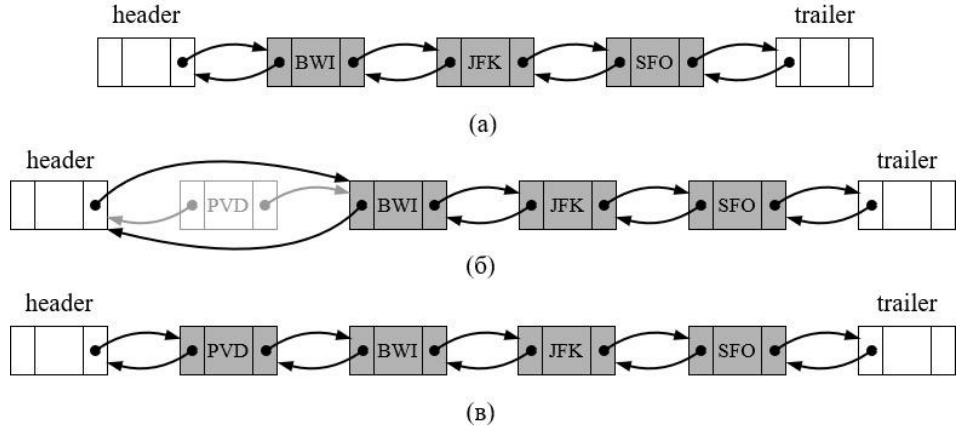


Рисунок 8.34 — Добавление узла в начало двунаправленного связного списка с ограничителями

Сначала создается новый узел с передаваемым значением. Затем новый узел связывается с текущим первым узлом так, что его атрибут next указывает на текущий первый узел, а атрибут prev текущего первого узла указывает на новый узел. После этого header обновляется так, чтобы указывать на новый узел. Это делает новый узел первым в списке.

Наконец, рассмотрим схему удаления произвольного элемента из двунаправленного связного списка (рис. 8.35). Это позволяет лучше понять, как происходит удаление элемента из такой структуры данных.

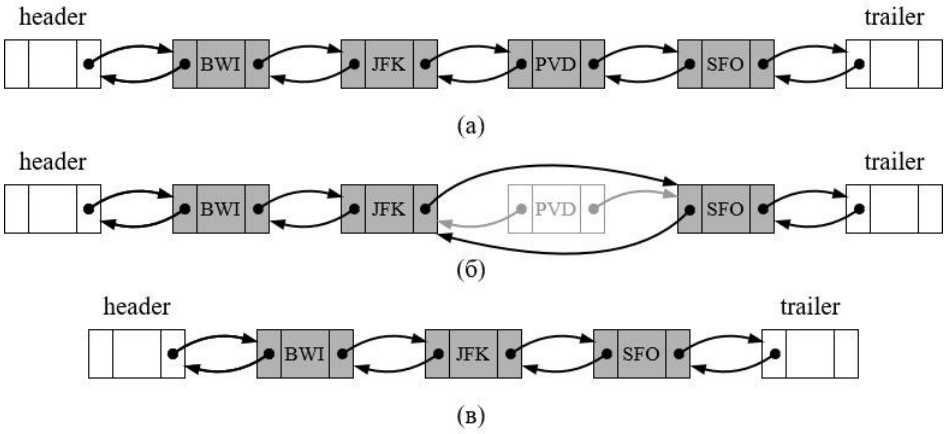


Рисунок 8.35 — Удаление элемента из двунаправленного связного списка с ограничителями

Сначала необходимо найти удаляемый элемент. Если этот элемент находится посередине, то нужно обновить ссылки у соседних элементов так, чтобы они указывали друг на друга, минуя удаляемый элемент. Затем освобождаем память от удаляемого элемента.

8.7. Примеры решения задач

Пример 8.1. Дано множество A из N точек с целочисленными координатами x, y. Найти сумму расстояний от каждой точки до начала координат и отсортировать точки по возрастанию расстояний до начала координат.

Решение. Импортируем необходимые модули, запрашиваем у пользователя количество точек (целое число) и сохраняем его в переменную *N*. Затем создаем массив points для хранения координат точек с помощью модуля array из стандартной библиотеки Python.

```
import math
from array import array
```

```
# Ввод количества точек
N = int(input("Введите количество точек: "))

# Создание массива для хранения координат точек
points = array('i', [])
```

```
Введите количество точек: 3
```

Рисунок 8.36 — Импорт модулей и ввод количества точек

Запрашиваем у пользователя координаты каждой точки (целые числа) и добавляем их в массив points с помощью метода extend().

```
# Ввод координат точек
for i in range(N):
    x = int(input("Введите координату x для точки {}: ".format(i+1)))
    y = int(input("Введите координату y для точки {}: ".format(i+1)))
    points.extend([x, y])
```

```
Введите координату x для точки 1: 3
Введите координату y для точки 1: 4
Введите координату x для точки 2: -5
Введите координату y для точки 2: 6
Введите координату x для точки 3: 2
Введите координату y для точки 3: 4
```

Рисунок 8.37 — Ввод координат точек

После этого вычисляем расстояние от каждой точки до начала координат с помощью формулы sqrt(x**2 + y**2), используя функцию sqrt из модуля math, и сохраняем результаты в массив distances. Затем выводим сумму расстояний от каждой точки до начала координат, используя функцию sum() и метод format() для форматирования вывода.

```
# Вычисление суммы расстояний и добавление ее в массив
distances = array('f', [])
for i in range(0, len(points), 2):
    x = points[i]
    y = points[i+1]
    distance = math.sqrt(x**2 + y**2)
    distances.append(distance)

print("Сумма расстояний от каждой точки до начала координат: {:.2f}".format(sum(distances)))
```

```
Сумма расстояний от каждой точки до начала координат: 17.28
```

Рисунок 8.38 — Вычисление суммы расстояний от точек до начала координат

Создаем массив indexes для хранения индексов точек. Индексы точек сортируются по возрастанию расстояний до начала координат с помощью функции sorted() и лямбда-функции. После этого выводим отсортированные точки, используя отсортированные индексы для доступа к координатам точек в массиве points. Координаты точек выводятся с помощью метода format().

```
# Создание массива для хранения индексов точек
indexes = array('i', range(N))

# Сортировка индексов точек по возрастанию расстояний до начала кординат
indexes = sorted(indexes, key=lambda i: distances[i])

# Вывод отсортированных точек
print("Отсортированные точки по возрастанию расстояний до начала координат:")
for i in indexes:
    x = points[i*2]
    y = points[i*2+1]
    print("({}, {})".format(x, y))
```

```
Отсортированные точки по возрастанию расстояний до начала координат:
(2, 4)
(3, 4)
(-5, 6)
```

Рисунок 8.39 — Сортировка точек и вывод результата

Пример 8.2. Дана квадратная матрица A порядка M. Найти среднее арифметическое элементов каждой ее диагонали, параллельной побочной (начиная с одноэлементной диагонали A_{00})).

Решение. Для работы с многомерными массивами обычно используется специализированная библиотека Numpy, предоставляющий мощные средства для работы с многомерными массивами, включая поддержку математических операций на массивах, таких как сложение и умножение матриц, транспонирование, взятие обратной матрицы и т.д., что делает NumPy более подходящим выбором для работы с многомерными массивами в Python.

```
import numpy as np

# Создаем квадратную матрицу порядка 3
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

# Находим среднее арифметическое элементов каждой диагонали, параллельной побочной
for i in range(len(A)-1, -len(A), -1):
    print(f"Среднее арифметическое элементов диагонали {i}: {np.mean(np.diagonal(np.fliplr(A), offset=i))}")
```

```
Среднее арифметическое элементов диагонали 2: 1.0
Среднее арифметическое элементов диагонали 1: 3.0
Среднее арифметическое элементов диагонали 0: 5.0
Среднее арифметическое элементов диагонали -1: 7.0
Среднее арифметическое элементов диагонали -2: 9.0
```

Рисунок 8.40 — Решение задачи

Сначала создается матрица A с помощью функции np.array(). Матрица состоит из трех строк и трех столбцов, заполненных числами от 1 до 9. Затем с помощью цикла for происходит перебор диагоналей матрицы, начиная с самой верхней диагонали и заканчивая самой нижней. В каждой итерации цикла выводится сообщение среднего арифметического элементов текущей диагонали.

Для нахождения элементов диагонали используется функция np.diagonal(), которая возвращает одномерный массив элементов указанной диагонали. Параметр offset задает сдвиг диагонали относительно главной диагонали (положительное значение сдвигает вверх, отрицательное — вниз). Функция np.fliplr() переставляет столбцы матрицы A симметрично относительно вертикальной оси, чтобы побочная диагональ стала главной. Это нужно для правильного нахождения параллельных побочной диагоналей.

Пример 8.3. Написать функцию для нахождения первого четного элемента стека.

Решение. Сначала импортируем модуль deque из библиотеки collections, а также модуль random. Затем создаем класс Stack, который наследуется от класса deque. В этом классе определен метод push, который добавляет элемент в стек с помощью функции append.

```
from collections import deque
import random
```

```
class Stack(deque):

    def push(self, a):
        self.append(a)
```

Рисунок 8.41 — Класс Stack

Далее определяем функцию find_first_even(), которая принимает на вход стек. Внутри этой функции выполняется цикл, пока стек не пустой. На каждой итерации извлекается элемент из стека с помощью функции pop. Затем проверяется, является ли этот элемент четным числом с помощью оператора %. Если это так, то функция возвращает этот элемент. Если не найдено ни одного четного элемента, то функция возвращает None.

```
def find_first_even(stack):
    while stack:
        element = stack.pop()
        if element % 2 == 0:
            return element
    return None
```

Рисунок 8.42 — Функция find_first_even()

Создаем экземпляр класса Stack под названием stack. В цикле от 0 до 9 генерируются случайные числа от 1 до 20 и добавляются в стек с помощью метода push. После этого выводим содержимое стека с помощью функции print. Наконец, вызываем функция find_first_even со стеком в качестве аргумента, и результат выводим на экран.

```
stack = Stack()
for i in range(10):
    stack.push(random.randint(1,20))

print(stack)
```

```
Stack([19, 15, 13, 18, 16, 14, 20, 20, 18, 17])
```

```
print(f"Первый четный элемент стека: {find_first_even(stack)}")
```

```
Первый четный элемент стека: 18
```

Рисунок 8.43 — Создание стека и поиск первого четного элемента в нем

Пример 8.4. Написать функцию для добавления нового элемента в стек после первого нечётного элемента.

Решение. Создаем класс Stack, который наследует функциональность класса deque из модуля collections. Это позволяет использовать методы стека, такие как push и pop.

```
from collections import deque
```

```
class Stack(deque):
    def push(self, a):
        self.append(a)
```

Рисунок 8.44 — Класс Stack

Определяем функцию add_element_after_odd(), которая принимает стек и элемент в качестве аргументов. Сначала создаем новый пустой стек new_stack. В цикле while происходит извлечение элементов из исходного стека stack с помощью метода pop. Если найден первый нечетный элемент и переменная odd_found равна False, то этот элемент и новый элемент добавляются в новый стек new_stack с помощью метода push. Переменная odd_found устанавливается в True. Если элемент четный или уже был найден первый нечетный элемент, то он просто добавляется в новый стек new_stack с помощью метода push. После завершения цикла while, все элементы из нового стека new_stack добавляются обратно в исходный стек stack с помощью метода push. Возвращается исходный стек stack с добавленным элементом после первого нечетного элемента.

```python
def add_element_after_odd(stack, element):
    odd_found = False
    new_stack = Stack()
    while stack:
        item = stack.pop()
        if not odd_found and item % 2 != 0:
            new_stack.push(item)
            new_stack.push(element)
            odd_found = True
        else:
            new_stack.push(item)
    while new_stack:
        stack.push(new_stack.pop())
    return stack
```

Рисунок 8.45 — Функция add_element_after_odd()

Теперь создаем объект stack класса Stack с исходными элементами [1, 2, 3, 4, 5]. Задаем элемент, равный 6, который будет добавлен в стек. Для этого вызываем функцию add_element_after_odd() с аргументами stack и element. Результат функции сохраняем в переменную result и выводим на экран.

```python
stack = Stack([1, 2, 3, 4, 5])
element = 6
result = add_element_after_odd(stack, element)
print(result)
```

```
Stack([1, 2, 3, 4, 6, 5])
```

Рисунок 8.46 — Решение задачи

Пример 8.5. С помощью стека проверить сбалансированность скобок в математическом выражении.

Решение. Определяем класс Stack, который наследуется от класса deque из модуля collections. Класс Stack имеет метод push, который добавляет элемент в конец стека.

```python
from collections import deque
```

```python
class Stack(deque):
    def push(self, a):
        self.append(a)
```

Рисунок 8.47 — Класс Stack

Функция is_balanced() принимает в качестве аргумента математическое выражение и проверяет его на сбалансированность скобок. Она создает экземпляр стека stack и определяет список открывающих и закрывающих скобок. Затем она проходит по каждому символу в выражении. Если символ является открывающей скобкой, он добавляется в стек. Если символ является закрывающей скобкой, то проверяется, есть ли элементы в стеке. Если стек пуст, то выражение не сбалансировано. Затем извлекается верхний эле-

мент из стека и проверяется соответствие открывающей и закрывающей скобок. Если они не соответствуют друг другу, то выражение не сбалансировано. В конце проверяется, пуст ли стек. Если стек пуст, то выражение сбалансировано.

```
def is_balanced(expression):
    stack = Stack()
    opening_brackets = ['(', '[', '{']
    closing_brackets = [')', ']', '}']
    for char in expression:
        if char in opening_brackets:
            stack.push(char)
        elif char in closing_brackets:
            if len(stack) == 0:
                return False
            top = stack.pop()
            if opening_brackets.index(top) != closing_brackets.index(char):
                return False
    return len(stack) == 0
```

```
def check_balance(expression):
    print("Скобки сбалансированы" if is_balanced(expression) else "Скобки не сбалансированы")
```

Рисунок 8.48 — Функции is_balanced() и check_balance()

Функция check_balance() принимает в качестве аргумента математическое выражение и выводит сообщение «Скобки сбалансированы», если выражение сбалансировано, и «Скобки не сбалансированы», если выражение не сбалансировано.

После определения класса Stack и функций is_balanced() и check_balance() запрашиваем у пользователя ввод двух математических выражений и вызываем функцию check_balance() для каждого из них.

```
expression1 = input("Введите математическое выражение: ")
check_balance(expression1)
```

```
Введите математическое выражение: (1 + 2*6)**2 - 5*6
Скобки сбалансированы
```

```
expression2 = input("Введите математическое выражение: ")
check_balance(expression2)
```

```
Введите математическое выражение: 2*(5**{6 -3/5) - 120) + sin(x)
Скобки не сбалансированы
```

Рисунок 8.49 — Применение функций

Пример 8.6. С помощью стека реализовать функцию для вычисления математических выражений в обратной польской нотации.

Решение. Обратная польская нотация (ОПН) — это форма записи математических выражений, в которой операторы располагаются после операндов. ОПН не использует скобки и упрощает вычисление выражений. Вместо того чтобы операторы находились между

операндами, в ОПН они идут после них. Для выполнения операций в ОПН используется стек, который позволяет извлекать операнды и выполнять операции с ними.

```
from collections import deque
```

```
class Stack(deque):
    def push(self, a):
        self.append(a)
```

Рисунок 8.50 — Класс Stack

Сначала импортируется модуль deque из стандартной библиотеки collections. Затем создается класс Stack, который наследуется от deque. Класс Stack расширяет функциональность deque и добавляет метод push для добавления элемента в стек.

```
def evaluate(expression):
    stack = Stack()
    for token in expression:
        if token.isdigit():
            stack.push(int(token))
        else:
            operand_2 = stack.pop()
            operand_1 = stack.pop()
            if token == '+':
                result = operand_1 + operand_2
            elif token == '-':
                result = operand_1 - operand_2
            elif token == '*':
                result = operand_1 * operand_2
            elif token == '/':
                result = operand_1 / operand_2
            stack.push(result)
    return stack.pop()
```

Рисунок 8.51 — Функция evaluate()

Функция evaluate() принимает строку expression, которая представляет собой выражение в обратной польской нотации. Сначала создается пустой стек stack. Затем происходит итерация по каждому токену в выражении. Если токен является числом, то оно преобразуется в целое число и добавляется в стек с помощью метода push. Если токен является оператором (+, –, *, /), то из стека извлекаются два операнда, выполняется соответствующая операция и результат добавляется обратно в стек. В конце функции из стека извлекается результат вычисления и возвращается.

```
# математическое выражение (3 + 4) * 2 - 1
expression = '3 4 + 2 * 1 -'
print(f"Значение выражения {expression} равно {evaluate(expression.split())}")
```

```
Значение выражения 3 4 + 2 * 1 - равно 13
```

Рисунок 8.52 — Решение задачи

Пример 8.7. Написать функцию для нахождения первого нечетного элемента очереди.

Решение. Реализуем класс Queue, который наследуется от класса deque из модуля collections. Класс Queue имеет два метода — enqueue и dequeue, которые добавляют элемент в конец очереди и удаляют элемент из начала очереди соответственно.

```
from collections import deque
import random
```

```
class Queue(deque):

    def enqueue(self, a):
        self.append(a)

    def dequeue(self):
        return self.popleft()
```

Рисунок 8.53 — Класс Queue

Определим функцию find_first_odd(), которая принимает на вход объект queue типа Queue. Внутри функции выполняется цикл до тех пор, пока очередь не станет пустой. На каждой итерации из очереди извлекается элемент с помощью метода dequeue. Если элемент нечетный (element % 2 != 0), то он возвращается из функции. Если все элементы в очереди четные, то функция возвращает None.

```
def find_first_odd(queue):
    while queue:
        element = queue.dequeue()
        if element % 2 != 0:
            return element
    return None
```

Рисунок 8.54 — Функция find_first_odd()

Для примера создаем объект queue типа Queue и в него добавляем несколько чисел. Затем вызывается функция find_first_odd, которая находит первое нечетное число в очереди и выводит его на экран.

```
queue = Queue()

for i in range(10):
    queue.enqueue(random.randint(1,20))

print(queue)
```

```
Queue([10, 16, 7, 1, 15, 19, 8, 17, 12, 1])
```

```
print(f"Первый нечетный элемент очереди: {find_first_odd(queue)}")
```

```
Первый нечетный элемент очереди: 7
```

Рисунок 8.55 — Решение задачи

Пример 8.8. Написать функцию для добавления нового элемента в очередь перед первым четным элементом.

Решение. Для решения задачи используется класс Queue, который наследуется от класса deque из модуля collections. Это позволяет использовать методы append и popleft для добавления и удаления элементов из очереди.

```
from collections import deque
```

```
class Queue(deque):

    def enqueue(self, a):
        self.append(a)

    def dequeue(self):
        return self.popleft()
```

Рисунок 8.56 — Класс Queue

Функция add_before_first_even() принимает два аргумента: queue — исходная очередь, и element — элемент, который нужно добавить перед первым четным элементом. Внутри функции создается новая очередь new_queue и переменная found_even, которая указывает, был ли найден четный элемент. Затем выполняется цикл while, пока длина исходной очереди queue больше нуля. В каждой итерации извлекается элемент из очереди с помощью метода dequeue. Если еще не был найден четный элемент и текущий элемент item является четным (item % 2 == 0), то элемент element добавляется в новую очередь перед текущим элементом. Переменная found_even устанавливается в True, и текущий элемент item добавляется в новую очередь. После завершения цикла проверяется, был ли найден четный элемент. Если нет, то элемент element добавляется в конец новой очереди. Наконец, функция возвращает новую очередь new_queue.

```
def add_before_first_even(queue, element):
    new_queue = Queue()
    found_even = False

    while len(queue) > 0:
        item = queue.dequeue()
        if not found_even and item % 2 == 0:
            new_queue.enqueue(element)
            found_even = True
        new_queue.enqueue(item)

    if not found_even:
        new_queue.enqueue(element)

    return new_queue
```

Рисунок 8.57 — Функция add_before_first_even()

Пример использования показывает, как создать исходную очередь queue с элементами [1, 3, 5, 8, 9, 6], и затем вызывает функцию add_before_first_even, передавая эту очередь и элемент 2.

```
queue = Queue([1, 3, 5, 8, 9])
new_queue = add_before_first_even(queue, 2)
print(new_queue)
```

```
Queue([1, 3, 5, 2, 8, 9])
```

Рисунок 8.58 — Решение задачи

Пример 8.9. Разработать систему бронирования билетов на концерты. Запросы на бронирование поступают в очередь и обрабатываются в порядке их поступления. Каждый запрос содержит информацию о концерте, количестве билетов и контактных данных клиента.

Решение. Класс Concert представляет концерт с названием тура, исполнителем, датой, местоположением, местом проведения, стандартной ценой, валютой и количеством билетов.

```
class Concert:
    def __init__(self, tour_name, artist, date, locality, venue, standard_price, currency_unit, tickets):
        self.tour_name = tour_name
        self.artist = artist
        self.date = date
        self.locality = locality
        self.venue = venue
        self.standard_price = standard_price
        self.currency_unit = currency_unit
        self.tickets = tickets

    def __str__(self):
        return (f'НАЗВАНИЕ ТУРА / ИСПОЛНИТЕЛЬ: {self.tour_name} – {self.artist}\n'
                f'ДАТА: {self.date}\n'
                f'МЕСТО ПРОВЕДЕНИЯ / МЕСТОПОЛОЖЕНИЕ: {self.venue} ({self.locality})\n'
                f'ЦЕНА: {self.currency_unit}{self.standard_price}\n'
                f'КОЛИЧЕСТВО БИЛЕТОВ: {self.tickets}\n')
```

Рисунок 8.59 — Класс Concert

Класс Reservation представляет бронирование на концерт и включает информацию о концерте, имени клиента, электронной почте и количестве билетов.

```
class Reservation:
    def __init__(self, concert, customer_name, email, tickets):
        self.concert = concert
        self.customer_name = customer_name
        self.email = email
        self.tickets = tickets

    def __str__(self):
        return (f'НАЗВАНИЕ ТУРА: {self.concert.tour_name}\n'
                f'ИМЯ КЛИЕНТА: {self.customer_name}\n'
                f'ЭЛ. ПОЧТА: {self.email}\n'
                f'КОЛИЧЕСТВО БИЛЕТОВ: {self.tickets}\n')
```

Рисунок 8.60 — Класс Reservation

Класс Queue представляет очередь бронирований.

```
from collections import deque
```

```
class Queue(deque):

    def enqueue(self, reservation):
        self.append(reservation)

    def dequeue(self):
        return self.popleft()
```

Рисунок 8.61 — Класс Queue

Класс TicketBookingSystem представляет систему бронирования билетов и позволяет добавлять концерты (метод add_concert()) и делать бронирования на них. Метод make_reservation() проверяет наличие свободных билетов на указанный концерт и добавляет бронирование в очередь, если есть свободные места. Метод process_queue() обрабатывает очередь бронирований и выводит информацию о каждом успешном бронировании.

```
class TicketBookingSystem:
    def __init__(self):
        self.concerts = []
        self.queue = Queue()

    def add_concert(self, tour_name, artist, date, locality, venue, standard_price, currency_unit, tickets):
        concert = Concert(tour_name, artist, date, locality, venue, standard_price, currency_unit, tickets)
        self.concerts.append(concert)

    def make_reservation(self, concert_name, customer_name, email, tickets):
        for concert in self.concerts:
            if concert.tour_name == concert_name:
                if concert.tickets >= tickets:
                    reservation = Reservation(concert, customer_name, email, tickets)
                    self.queue.enqueue(reservation)
                    concert.tickets -= tickets
                    return True
                else:
                    return False
        return False

    def process_queue(self):
        reservation = self.queue.dequeue()
        if reservation is not None:
            concert = reservation.concert
            print(f"Бронирование {reservation.tickets} билетов на концерт {concert.tour_name.upper()} ({concert.artist}) ")
            print(f"{concert.date} в {concert.venue.upper()} ({concert.locality}) ")
            print(f"для {reservation.customer_name.upper()} ({reservation.email})")
            print(f"ИТОГО: {reservation.tickets} x {concert.currency_unit}{concert.standard_price}", end=" ")
            print(f"= {concert.currency_unit}{reservation.tickets * concert.standard_price}\n")
        else:
            print("Очередь пуста")
```

Рисунок 8.62 — Класс TicketBookingSystem

Определяем словари concerts и customers, которые содержат информацию о концертах и клиентах соответственно.

```
concerts = {"Возвращение на сцену": ["Исполнитель 1", "12 февраля 2022", "Москва", "Олимпийский стадион", 1500, "₽", 5000],
            "Города России": ["Исполнитель 2", "25 марта 2022", "Санкт-Петербург", "Ледовый дворец", 1000, "₽", 4000],
            "Время любви": ["Исполнитель 3", "10 апреля 2022", "Екатеринбург", "Уральский молния", 1200, "₽", 6000],
            "Родные города": ["Исполнитель 4", "5 мая 2022", "Нижний Новгород", "Нижегородский стадион", 800, "₽", 3500],
            "Подмосковные вечера": ["Исполнитель 5", "20 июня 2022", "Мытищи", "Арена Мытищи", 500, "₽", 3000],
            "Самые яркие города": ["Исполнитель 6", "15 июля 2022", "Ростов-на-Дону", "Ростов-Арена", 900, "₽", 4500],
            "Юбилейный тур": ["Исполнитель 7", "30 августа 2022", "Краснодар", "Краснодарская арена", 700, "₽", 4000],
            "Золотая осень": ["Исполнитель 8", "10 сентября 2022", "Новосибирск", "Академгородок", 800, "₽", 3500],
            "Московские ночи": ["Исполнитель 9", "25 октября 2022", "Москва", "Кремлевский дворец", 2000, "₽", 8000],
            "Север-Юг": ["Исполнитель 10", "5 ноября 2022", "Сочи", "Большой дворец спорта", 1200, "₽", 6000]
            }

customers = {'Александр Иванов': 'a.ivanov@example.ru',
             'Мария Петрова': 'm.petrova@example.ru',
             'Дмитрий Смирнов': 'd.smirnov@example.ru',
             'Анастасия Соколова': 'a.sokolova@example.ru',
             'Сергей Кузнецов': 's.kuznetsov@example.ru',
             'Ольга Попова': 'o.popova@example.ru',
             'Максим Волков': 'm.volkov@example.ru',
             'Елена Федорова': 'e.fedorova@example.ru',
             'Иван Сидоров': 'i.sidorov@example.ru',
             'Наталья Морозова': 'n.morozova@example.ru',
             'Андрей Коваленко': 'a.kovalenko@example.ru',
             'Екатерина Иванова': 'e.ivanova@example.ru',
             'Алексей Воробьев': 'a.vorobev@example.ru',
             'Дарья Козлова': 'd.kozlova@example.ru',
             'Роман Егоров': 'r.egorov@example.ru'
             }
```

Рисунок 8.63 — Определение словарей concerts и customers

Создаем экземпляр класса TicketBookingSystem() и для каждого концерта из словаря concerts вызываем метод add_concert(), который добавляет концерт в систему.

```
import random

system = TicketBookingSystem()

for concert in concerts:
    artist, date, locality, venue, standard_price, currency_unit, tickets = tuple(concerts[concert])
    system.add_concert(concert, artist, date, locality, venue, standard_price, currency_unit, tickets)
```

Рисунок 8.64 — Создание экземпляра класса TicketBookingSystem()

Выполняем случайное перемешивание списка клиентов и для каждого из них вызываем метод make_reservation() с выбранным случайным концертом, именем клиента и его электронной почтой, а также случайным количеством билетов.

```
customer_list = list(customers.keys())
random.shuffle(customer_list)

for customer in customer_list:
    system.make_reservation(random.choice(list(concerts.keys())), customer, customers[customer], 2*random.randint(4,20))
```

Рисунок 8.65 — Внесение данных в систему

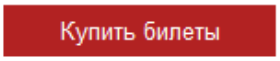
Создаем кнопку button, при нажатии на которую вызывается метод process_queue(), обрабатывающий очередь бронирований и выводящий информацию о каждом успешном бронировании. Для работы кнопки используются модули IPython.display и ipywidgets.

```
from IPython.display import display
from ipywidgets import Button

button = Button(description='Купить билеты', button_style='success')
button.style.button_color = 'FireBrick'

def on_button_click(b):
    system.process_queue()

button.on_click(on_button_click)
display(button)
```




```
Бронирование 36 билетов на концерт ПОДМОСКОВНЫЕ ВЕЧЕРА (Исполнитель 5)
20 июня 2022 в АРЕНА МЫТИЩИ (Мытищи)
для МАРИЯ ПЕТРОВА (m.petrova@example.ru)
ИТОГО: 36 × ₽500 = ₽18000

Бронирование 34 билетов на концерт СЕВЕР-ЮГ (Исполнитель 10)
5 ноября 2022 в БОЛЬШОЙ ДВОРЕЦ СПОРТА (Сочи)
для ИВАН СИДОРОВ (i.sidorov@example.ru)
ИТОГО: 34 × ₽1200 = ₽40800

Бронирование 24 билетов на концерт ВОЗВРАЩЕНИЕ НА СЦЕНУ (Исполнитель 1)
12 февраля 2022 в ОЛИМПИЙСКИЙ СТАДИОН (Москва)
для ОЛЬГА ПОПОВА (o.popova@example.ru)
ИТОГО: 24 × ₽1500 = ₽36000
```

Рисунок 8.66 — Пример работы системы

В примерах 8.10 и 8.11 будут использоваться два класса.

Класс Node представляет узел списка и имеет три атрибута: data (данные, хранящиеся в узле), prev (ссылка на предыдущий узел) и next (ссылка на следующий узел).

Класс DoublyLinkedList представляет сам список и имеет один атрибут: head (ссылка на первый узел списка).

Метод add_node(data) добавляет новый узел с данными data в конец списка. Если список пустой, то новый узел становится первым (head). В противном случае, происходит перебор узлов списка до последнего узла, после чего новый узел становится следующим для последнего узла, а предыдущим для нового узла становится последний узел.

Метод delete_node(data) удаляет первый узел с данными data из списка. Если список пустой, то ничего не происходит. Если первый узел содержит данные data, то он удаляется из списка и следующий узел становится первым. В противном случае, происходит перебор узлов списка до тех пор, пока не будет найден узел с данными data или пока не будет достигнут конец списка. Если найденный узел не является последним, то он удаляется из списка и следующий узел становится следующим для предыдущего узла.

Метод __len__() возвращает количество узлов в списке. Он проходит по всем узлам списка, увеличивая счетчик count на 1 для каждого узла, и возвращает значение count.

Метод __str__() возвращает строковое представление списка. Если список пустой, то возвращается сообщение «Двусвязный список пустой». В противном случае, происходит перебор узлов списка и добавление их данных в строку dllist_str с помощью символа «⇄» в качестве разделителя. Затем строка dllist_str обрезается с помощью метода lstrip(« ⇄ «), чтобы удалить начальные разделители. Полученная строка представляет собой данные всех узлов списка, разделенные символом «⇄».

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None
```

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def add_node(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = new_node
            new_node.prev = current

    def delete_node(self, data):
        if self.head is None:
            return
        elif self.head.data == data:
            if self.head.next is not None:
                self.head = self.head.next
                self.head.prev = None
            else:
                self.head = None
        else:
            current = self.head
            while current.next is not None and current.next.data != data:
                current = current.next
            if current.next is None:
                return
            else:
                current.next = current.next.next
                if current.next is not None:
                    current.next.prev = current
```

Рисунок 8.67 — Класс DoublyLinkedList

```
    def __len__(self):
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next
        return count

    def __str__(self):
        if self.head == None:
            return f"Двусвязный список пустой"
        current = self.head
        dllist_str = ""
        while current:
            dllist_str += " ⇄ " + str(current.data)
            current = current.next
        return dllist_str.lstrip(" ⇄ ")
```

Рисунок 8.68 — Магические методы в классе DoublyLinkedList

Пример 8.10. Написать функцию, удваивающую каждый четный элемент в двусвязном списке, т.е. добавляет рядом с четным элементом такой же элемент.

Решение. Реализуем функцию double_even_nodes(), которая удваивает каждый четный элемент в двусвязном списке. Функция принимает в качестве аргумента объект класса DoublyLinkedList. В начале устанавливается указатель current_node на голову списка. Затем происходит итерация по списку с помощью цикла while.

На каждой итерации проверяется, является ли значение текущего узла четным числом. Если да, то создается новый узел new_node с таким же значением, как и текущий узел. Затем происходит изменение ссылок между узлами: new_node.next указывает на следующий узел после текущего, new_node.prev указывает на текущий узел, а если следующий узел существует, то его prev ссылка указывает на new_node. Затем next ссылка текущего узла указывает на new_node, а current_node переходит к следующему узлу после new_node. Если значение текущего узла нечетное, то current_node просто переходит к следующему узлу. После завершения цикла while двусвязный список будет содержать удвоенные четные элементы.

```
def double_even_nodes(dllist):
    current_node = dllist.head
    while current_node:
        if current_node.data % 2 == 0:
            new_node = Node(current_node.data)
            new_node.next = current_node.next
            new_node.prev = current_node
            if current_node.next:
                current_node.next.prev = new_node
            current_node.next = new_node
            current_node = new_node.next
        else:
            current_node = current_node.next
```

Рисунок 8.69 — Функция double_even_nodes()

Создаем объект dll класса DoublyLinkedList и добавляем случайные элементы в список. После этого выводим на экран исходный двусвязный список. Наконец, вызываем функцию double_even_nodes() для удвоения четных элементов списка. Измененный двусвязный список отображаем на экране.

```
dll = DoublyLinkedList()
```

```
from random import randint

for i in range(randint(5,10)):
    dll.add_node(randint(1,20))

print(f'Двусвязный список (len = {len(dll)}): {dll}')
```

```
Двусвязный список (len = 5): 5 ⇄ 6 ⇄ 16 ⇄ 8 ⇄ 20
```

```
double_even_nodes(dll)

print(f'Двусвязный список (len = {len(dll)}): {dll}')
```

```
Двусвязный список (len = 9): 5 ⇄ 6 ⇄ 6 ⇄ 16 ⇄ 16 ⇄ 8 ⇄ 8 ⇄ 20 ⇄ 20
```

Рисунок 8.70 — Пример работы с двусвязным списком

Пример 8.11. Написать функцию для удаления всех отрицательных элементов из двусвязного списка.

Решение. Определим функцию delete_negative_nodes(), которая принимает двусвязный список в качестве аргумента и удаляет все отрицательные элементы из списка. В начале функции устанавливается переменная current_node в голову списка. Затем происходит цикл, который продолжается до тех пор, пока current_node не станет равным None. Внутри цикла проверяется значение data текущего узла. Если оно меньше 0, то выполняются следующие действия:

- Проверяется, есть ли у текущего узла предыдущий узел (current_node.prev). Если есть, то связь между предыдущим узлом и следующим узлом обновляется, чтобы пропустить текущий узел.
- Если у текущего узла есть следующий узел (current_node.next), то связь между следующим узлом и предыдущим узлом также обновляется.
- Переменная current_node обновляется, чтобы перейти к следующему узлу списка.

После выполнения цикла отрицательные элементы должны быть удалены из списка.

```python
def delete_negative_nodes(dllist):
    current_node = dllist.head
    while current_node:
        if current_node.data < 0:
            if current_node.prev:
                current_node.prev.next = current_node.next
            else:
                dllist.head = current_node.next
            if current_node.next:
                current_node.next.prev = current_node.prev
        current_node = current_node.next
```

Рисунок 8.71 — Функция delete_negative_nodes()

Создаем объект dll класса DoublyLinkedList. В цикле добавляем случайные элементы в двусвязный список, используя метод add_node, где значения элементов генерируются с помощью функции randint(). После этого на экран выводим текущее состояние списка.

Наконец, вызываем функцию delete_negative_nodes(), чтобы удалить отрицательные элементы из списка. После удаления отрицательных элементов на экран выводится обновленное состояние списка.

```python
dll = DoublyLinkedList()
```

```python
from random import randint

for i in range(randint(5,10)):
    dll.add_node(randint(-20,20))

print(f'Двусвязный список (len = {len(dll)}): {dll}')
```

```
Двусвязный список (len = 9): -8 ⇄ 11 ⇄ 5 ⇄ -6 ⇄ -8 ⇄ -7 ⇄ 5 ⇄ -6 ⇄ 14
```

```python
delete_negative_nodes(dll)

print(f'Двусвязный список (len = {len(dll)}): {dll}')
```

```
Двусвязный список (len = 4): 11 ⇄ 5 ⇄ 5 ⇄ 14
```

Рисунок 8.72 — Пример работы с двусвязным списком

Пример 8.12. Используя двусвязный список, создать эффективную структуру данных для хранения и обработки логов серверов.

Решение. Сначала сгенерируем случайные лог-записи для веб-сервера и запишем их в файл server.log.

```python
import random
import datetime

# создание списка HTTP-методов
http_methods = ['GET', 'POST', 'PUT', 'DELETE']

# генерация случайной даты и времени
def random_date(start, end):
    return start + datetime.timedelta(seconds=random.randint(0, int((end - start).total_seconds())))

# генерация случайного IP-адреса
def random_ip():
    return ".".join(str(random.randint(0, 255)) for _ in range(4))

# генерация случайной строки запроса
def random_request():
    return "/path/to/resource?param=" + str(random.randint(0, 100))

# генерация файла с логами
def generate_logs(filename, start_date, end_date, num_entries):
    with open(filename, "w") as f:
        for i in range(num_entries):
            date_time = random_date(start_date, end_date)
            ip_address = random_ip()
            http_method = random.choice(http_methods)
            request = random_request()
            log_entry = f"{date_time} {ip_address} {http_method} {request}\n"
            f.write(log_entry)
```

```python
# создание лог-файла
start_date = datetime.datetime(2023, 1, 1)
end_date = datetime.datetime(2023, 4, 17)
num_entries = 10
generate_logs("server.log", start_date, end_date, num_entries)
```

Рисунок 8.73 — Создание лог-файла

Теперь создадим структуру данных для хранения и обработки логов серверов. Класс LogEntry представляет запись журнала и содержит информацию о дате и времени, IP-адресе, HTTP-методе и запросе.

```python
class LogEntry:
    def __init__(self, date_time, ip_address, http_method, request):
        self.date_time = date_time
        self.ip_address = ip_address
        self.http_method = http_method
        self.request = request
```

Рисунок 8.74 — Класс LogEntry

Класс Node представляет узел двусвязного списка и содержит ссылку на объект класса LogEntry, а также ссылки на предыдущий и следующий узлы.

```python
class Node:
    def __init__(self, log_entry):
        self.log_entry = log_entry
        self.prev = None
        self.next = None
```

Рисунок 8.75 — Класс Node

Класс DoublyLinkedList представляет двусвязный список и содержит ссылки на его голову и хвост. В класс включены методы для добавления записей в список, поиска записей по заданному параметру и удаления записей из списка, а также метод для печати списка.

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def add_entry(self, log_entry):
        new_node = Node(log_entry)
        if not self.head:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def find_entry(self, param):
        current_node = self.head
        while current_node:
            if param in current_node.log_entry.__dict__.values():
                return current_node.log_entry
            current_node = current_node.next
        return None

    def delete_entry(self, log_entry):
        current_node = self.head
        while current_node:
            if current_node.log_entry == log_entry:
                if current_node.prev:
                    current_node.prev.next = current_node.next
                else:
                    self.head = current_node.next
                if current_node.next:
                    current_node.next.prev = current_node.prev
                else:
                    self.tail = current_node.prev
                return True
            current_node = current_node.next
        return False

    def print_list(self):
        current_node = self.head
        while current_node:
            print(current_node.log_entry.__dict__)
            current_node = current_node.next
```

Рисунок 8.76 — Класс DoublyLinkedList

Функция read_logs() считывает лог-файл, создает объекты класса LogEntry для каждой строки файла и добавляет их в двусвязный список. В конце функция возвращает этот список.

```
def read_logs(filename):
    dll = DoublyLinkedList()
    with open(filename, "r") as f:
        for line in f:
            line = line.strip().split()
            date_time = datetime.datetime.strptime(line[0] + " " + line[1], "%Y-%m-%d %H:%M:%S")
            ip_address = line[2]
            http_method = line[3]
            request = line[4]
            log_entry = LogEntry(date_time, ip_address, http_method, request)
            dll.add_entry(log_entry)
    return dll
```

Рисунок 8.77 — Функция read_logs()

Пример использования вышеописанных классов.

```
log_list = read_logs("server.log")

log_list.print_list()
```

```
{'date_time': datetime.datetime(2023, 2, 23, 13, 32, 15), 'ip_address': '136.231.165.9', 'http_method': 'PUT', 'request':
'/path/to/resource?param=77'}
{'date_time': datetime.datetime(2023, 4, 2, 14, 10, 32), 'ip_address': '112.17.213.21', 'http_method': 'DELETE', 'request
': '/path/to/resource?param=45'}
{'date_time': datetime.datetime(2023, 1, 28, 11, 55, 13), 'ip_address': '243.175.134.170', 'http_method': 'DELETE', 'reque
st': '/path/to/resource?param=15'}
{'date_time': datetime.datetime(2023, 3, 29, 11, 30, 43), 'ip_address': '211.193.64.17', 'http_method': 'GET', 'request':
'/path/to/resource?param=0'}
{'date_time': datetime.datetime(2023, 3, 2, 3, 52, 51), 'ip_address': '86.7.7.65', 'http_method': 'DELETE', 'request': '/p
ath/to/resource?param=25'}
{'date_time': datetime.datetime(2023, 1, 26, 3, 8, 46), 'ip_address': '196.201.79.171', 'http_method': 'PUT', 'request':
'/path/to/resource?param=23'}
{'date_time': datetime.datetime(2023, 3, 8, 23, 11, 38), 'ip_address': '211.29.206.217', 'http_method': 'PUT', 'request':
'/path/to/resource?param=17'}
{'date_time': datetime.datetime(2023, 3, 15, 6, 59, 13), 'ip_address': '215.47.78.153', 'http_method': 'GET', 'request':
'/path/to/resource?param=92'}
{'date_time': datetime.datetime(2023, 1, 27, 14, 23, 54), 'ip_address': '92.144.99.230', 'http_method': 'PUT', 'request':
'/path/to/resource?param=59'}
{'date_time': datetime.datetime(2023, 4, 5, 5, 43, 25), 'ip_address': '236.76.120.94', 'http_method': 'GET', 'request': '/
path/to/resource?param=89'}
```

Рисунок 8.78 — Чтение лог-файла

```
ip_to_find = "92.144.99.230"
entry_to_delete = log_list.find_entry(ip_to_find)
if entry_to_delete:
    log_list.delete_entry(entry_to_delete)
    print(f"Запись с IP-адресом {ip_to_find} удалена")
else:
    print(f"Запись с IP-адресом {ip_to_find} не найдена")
```

```
Запись с IP-адресом 92.144.99.230 удалена
```

```
log_list.print_list()
```

```
{'date_time': datetime.datetime(2023, 2, 28, 9, 13, 39), 'ip_address': '12.81.147.130', 'http_method': 'DELETE', 'request
': '/path/to/resource?param=44'}
{'date_time': datetime.datetime(2023, 1, 22, 1, 33, 54), 'ip_address': '105.131.237.103', 'http_method': 'PUT', 'request':
'/path/to/resource?param=14'}
{'date_time': datetime.datetime(2023, 4, 15, 19, 46, 5), 'ip_address': '74.176.140.144', 'http_method': 'PUT', 'request':
'/path/to/resource?param=79'}
{'date_time': datetime.datetime(2023, 4, 12, 23, 53, 31), 'ip_address': '69.178.244.146', 'http_method': 'GET', 'request':
'/path/to/resource?param=60'}
{'date_time': datetime.datetime(2023, 1, 3, 20, 25, 29), 'ip_address': '101.248.209.51', 'http_method': 'PUT', 'request':
'/path/to/resource?param=30'}
{'date_time': datetime.datetime(2023, 4, 13, 15, 14, 53), 'ip_address': '153.147.212.186', 'http_method': 'POST', 'request
': '/path/to/resource?param=4'}
{'date_time': datetime.datetime(2023, 1, 5, 8, 33, 54), 'ip_address': '164.32.185.25', 'http_method': 'DELETE', 'request':
'/path/to/resource?param=93'}
{'date_time': datetime.datetime(2023, 1, 24, 11, 50, 30), 'ip_address': '79.222.165.116', 'http_method': 'DELETE', 'reques
t': '/path/to/resource?param=73'}
{'date_time': datetime.datetime(2023, 3, 19, 2, 9, 56), 'ip_address': '86.57.34.130', 'http_method': 'PUT', 'request': '/p
ath/to/resource?param=91'}
```

Рисунок 8.79 — Удаление записи

Для примеров 8.13 и 8.14 определим классы Node и CircularDoublyLinkedList.

Класс Node представляет узел в циклическом (кольцевом) двусвязном списке. Он имеет три атрибута: data, prev и next. Атрибут data хранит значение узла, а атрибуты prev и next хранят ссылки на предыдущий и следующий узлы в списке соответственно. Метод __init__() инициализирует эти атрибуты значениями по умолчанию.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.prev = None
        self.next = None
```

Рисунок 8.80 — Класс Node

Класс CircularDoublyLinkedList представляет циклический двусвязный список. Он имеет два атрибута: head и tail. Атрибут head хранит ссылку на первый узел в списке, а атрибут tail хранит ссылку на последний узел в списке. Метод __init__() инициализирует эти атрибуты значением None.

Метод append() добавляет новый узел с заданными данными в конец списка. Если список пуст, новый узел становится и головой, и хвостом списка. В противном случае новый узел вставляется после хвостового узла, и обновляются необходимые ссылки. Метод prepend() добавляет новый узел с заданными данными в начало списка. Если список пуст, новый узел становится и головой, и хвостом списка. В противном случае новый узел вставляется перед головным узлом, и обновляются необходимые ссылки. Метод delete() удаляет узел с заданным ключом из списка. Он перебирает список, начиная с головного узла, и проверяет, соответствует ли данные каждого узла ключу. Если найдено совпадение, узел удаляется путем обновления необходимых ссылок.

Метод __len__() возвращает длину списка, подсчитывая количество узлов. Он начинает с головного узла и перебирает список, пока не достигнет снова головы. Метод __str__() возвращает строковое представление списка. Он начинает с головного узла и добавляет данные каждого узла к строке, разделяя их символом « ⇄ ». Он останавливается, когда достигает снова головного узла.

```
class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.prev = self.tail
            new_node.next = self.head
        else:
            new_node.prev = self.tail
            new_node.next = self.head
            self.tail.next = new_node
            self.head.prev = new_node
            self.tail = new_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
            new_node.prev = self.tail
            new_node.next = self.head
        else:
            new_node.prev = self.tail
            new_node.next = self.head
            self.head.prev = new_node
            self.tail.next = new_node
            self.head = new_node
```

Рисунок 8.81 — Класс CircularDoublyLinkedList

```
    def delete(self, key):
        current_node = self.head
        while current_node:
            if current_node.data == key:
                if current_node == self.head:
                    self.head = current_node.next
                    self.tail.next = self.head
                    self.head.prev = self.tail
                elif current_node == self.tail:
                    self.tail = current_node.prev
                    self.head.prev = self.tail
                    self.tail.next = self.head
                else:
                    current_node.prev.next = current_node.next
                    current_node.next.prev = current_node.prev
                return
            current_node = current_node.next

    def __len__(self):
        count = 0
        current_node = self.head
        while current_node:
            count += 1
            current_node = current_node.next
            if current_node == self.head:
                break
        return count

    def __str__(self):
        cdllist_str = ""
        current_node = self.head
        while current_node:
            cdllist_str += str(current_node.data) + " ⇄ "
            current_node = current_node.next
            if current_node == self.head:
                break
        return " ⇄ " + cdllist_str
```

Рисунок 8.82 — Класс CircularDoublyLinkedList

Пример 8.13. Написать функцию, возводящую в квадрат все отрицательные элементы в циклическом двусвязном списке.

Решение. Определим функцию square_negative_values, которая принимает в качестве аргумента объект класса CircularDoublyLinkedList и проходит по всем элементам списка. Если текущий элемент отрицательный, его значение возводится в квадрат. Затем происходит переход к следующему элементу. Если достигнут конец списка (т.е. текущий элемент равен головному элементу списка), цикл прерывается.

```
def square_negative_values(cdllist):
    current_node = cdllist.head
    while current_node:
        if current_node.data < 0:
            current_node.data = current_node.data ** 2
        current_node = current_node.next
        if current_node == cdllist.head:
            break
```

Рисунок 8.83 — Функция square_negative_values()

Создаем объект класса CircularDoublyLinkedList и добавляем случайные элементы в список. После добавления элементов список выводится на экран с помощью функции print. Затем вызываем функцию square_negative_values(), возводящую в квадрат все отрицательные элементы в списке.

```
cdll = CircularDoublyLinkedList()
```

```
from random import randint

for i in range(randint(5,10)):
    cdll.append(randint(-20,20))

print(f'Циклический двусвязный список (len = {len(cdll)}): {cdll}')
```

```
Циклический двусвязный список (len = 10):  ⇄ 4 ⇄ -15 ⇄ 6 ⇄ -3 ⇄ -11 ⇄ -9 ⇄ 2 ⇄ -13 ⇄ 10 ⇄ -13 ⇄
```

```
square_negative_values(cdll)

print(f'Циклический двусвязный список (len = {len(cdll)}): {cdll}')
```

```
Циклический двусвязный список (len = 10):  ⇄ 4 ⇄ 225 ⇄ 6 ⇄ 9 ⇄ 121 ⇄ 81 ⇄ 2 ⇄ 169 ⇄ 10 ⇄ 169 ⇄
```

Рисунок 8.84 — Пример работы с циклическим двусвязным списком

Пример 8.14. Написать функцию для удаления всех элементов из циклического двусвязного списка, кратных 5.

Решение. Реализуем функцию delete_multiples_of_5(), которая принимает объект циклического двусвязного списка cdllist в качестве аргумента. Внутри функции происходит итерация по элементам списка. Если значение текущего узла делится на 5 без остатка, то он удаляется из списка с помощью метода delete. Затем указатель переходит к следующему узлу. Если указатель достигает головного узла списка (т.е. произошел один полный обход списка), цикл прерывается.

```
def delete_multiples_of_5(cdllist):
    current_node = cdllist.head
    while current_node:
        if current_node.data % 5 == 0:
            cdllist.delete(current_node.data)
        current_node = current_node.next
        if current_node == cdllist.head:
            break
```

Рисунок 8.85 — Функция delete_multiples_of_5()

Создаем объект cdll класса CircularDoublyLinkedList и генерируется случайное количество элементов от 5 до 10, которые добавляются в список с помощью метода append. Вызываем функцию delete_multiples_of_5 для удаления всех элементов, кратных 5, из списка и выводим на экран содержимое списка для проверки удаления элементов.

```
cdll = CircularDoublyLinkedList()
```

```
from random import randint

for i in range(randint(5,10)):
    cdll.append(randint(-20,20))

print(f'Циклический двусвязный список (len = {len(cdll)}): {cdll}')
```

```
Циклический двусвязный список (len = 7):  ⇄ 17 ⇄ -1 ⇄ 5 ⇄ 10 ⇄ -5 ⇄ 10 ⇄ -5 ⇄
```

```
delete_multiples_of_5(cdll)

print(f'Циклический двусвязный список (len = {len(cdll)}): {cdll}')
```

```
Циклический двусвязный список (len = 2):  ⇄ 17 ⇄ -1 ⇄
```

Рисунок 8.86 — Пример работы с циклическим двусвязным списком

8.8 Задачи для самостоятельного решения

Задача 8.1. Используя модуль array, решить следующие задачи.

16) Дан массив размера N. Найти количество участков, на которых его элементы возрастают.

17) Дано множество A из N точек с целочисленными координатами x, y. Для хранения данных о каждом наборе точек следует использовать по два массива: первый массив для хранения абсцисс, второй — для хранения ординат. Порядок на координатной плоскости определим следующим образом: $(x_1, y_1) < (x_2, y_2)$, если либо $x_1 + y_1 < x_2 + y_2$, либо $x_1 + y_1 = x_2 + y_2$ и $x_1 < x_2$. Расположить точки данного множества по убыванию в соответствии с указанным порядком.

18) Дано число R и массив A размера N. Найти элемент массива, который наиболее близок к числу R.

19) Дано число R и массив размера N. Найти два соседних элемента массива, сумма которых наиболее близка к числу R, и вывести эти элементы в порядке возрастания их индексов.

20) Дан массив размера N. Найти номера двух ближайших элементов из этого массива (т. е. элементов с наименьшим модулем разности) и вывести эти номера в порядке возрастания.

21) Дан целочисленный массив размера N. Найти максимальное количество его одинаковых элементов.

22) Даны два массива A и B одинакового размера N. Сформировать новый массив C того же размера, каждый элемент которого равен максимальному из элементов массивов A и B с тем же индексом.

23) Дан массив A размера N. Сформировать новый массив B того же размера по следующему правилу: элемент B_K равен сумме элементов массива A с номерами от 1 до K.

24) Дан массив A размера N. Сформировать новый массив B того же размера по следующему правилу: элемент B_K равен среднему арифметическому элементов массива A с номерами от 1 до K.

25) Дан массив A размера N. Сформировать новый массив B того же размера по следующему правилу: элемент B_K равен сумме элементов массива A с номерами от K до N.

26) Дан массив размера N. После каждого отрицательного элемента массива вставить элемент с нулевым значением.

27) Дан целочисленный массив размера N. Удалить из массива все элементы, встречающиеся ровно два раза, и вывести размер полученного массива и его содержимое.

28) Дан целочисленный массив размера N. Удалить из массива все элементы, встречающиеся менее трех раз, и вывести размер полученного массива и его содержимое.

29) Дано множество A из N точек (точки заданы своими координатами x, y). Для хранения данных о каждом наборе точек следует использовать по два массива: первый массив для хранения абсцисс, второй — для хранения ординат. Среди всех точек этого множества, лежащих во второй четверти, найти точку, наиболее удаленную от начала координат. Если таких точек нет, то вывести точку с нулевыми координатами.

30) Дано целое число L (> 1) и целочисленный массив размера N. Заменить каждую серию массива, длина которой меньше L, на один элемент с нулевым значением. Серией называется группа подряд идущих одинаковых элементов, а длиной серии — количество этих элементов (длина серии может быть равна 1).

Задача 8.2. Используя библиотеку Numpy, решить следующие задачи.

16) Дана квадратная матрица A порядка M. Найти сумму элементов каждой ее диагонали, параллельной главной (начиная с одноэлементной диагонали $A_{1,M}$).

17) Дана квадратная матрица A порядка M. Повернуть ее на угол 90° в отрицательном направлении, т. е. по часовой стрелке (при этом элемент $A_{1,1}$ перейдет в $A_{1,M}$, элемент $A_{1,M}$ — в $A_{M,M}$ и т. д.). Вспомогательную матрицу не использовать.

18) Дана квадратная матрица порядка M. Обнулить элементы матрицы, лежащие на побочной диагонали и ниже нее. Условный оператор не использовать.

19) Дана целочисленная матрица размера $M \times N$. Найти номер первого из ее столбцов, содержащих максимальное количество одинаковых элементов.

20) Дана целочисленная матрица размера $M \times N$, элементы которой могут принимать значения от 0 до 100. Различные строки матрицы назовем похожими, если совпадают множества чисел, встречающихся в этих строках. Найти количество строк, похожих на первую строку данной матрицы.

21) Дана целочисленная матрица размера $M \times N$. Найти количество ее столбцов, все элементы которых различны.

22) Дана целочисленная матрица размера $M \times N$. Найти элемент, являющийся максимальным в своей строке и минимальным в своем столбце. Если такой элемент отсутствует, то вывести 0.

23) Дана матрица размера $M \times N$. Элемент матрицы называется ее локальным минимумом, если он меньше всех окружающих его элементов. Заменить все локальные минимумы данной матрицы на нули.

24) Дана квадратная матрица A порядка M. Зеркально отразить ее элементы относительно побочной диагонали (при этом элементы побочной диагонали останутся на прежнем месте, элемент $A_{1,1}$ поменяется местами с $A_{M,M}$, элемент $A_{1,2}$ — с $A_{M-1,M}$ и т. д.). Вспомогательную матрицу не использовать.

25) Дана матрица размера $M \times N$. Упорядочить ее строки так, чтобы их первые элементы образовывали возрастающую последовательность.

26) Дана матрица размера $M \times N$. Упорядочить ее столбцы так, чтобы их последние элементы образовывали убывающую последовательность.

27) Дана матрица размера $M \times N$. Упорядочить ее столбцы так, чтобы их максимальные элементы образовывали возрастающую последовательность.

28) Дана квадратная матрица A порядка M. Найти максимальный элемент для каждой ее диагонали, параллельной побочной (начиная с одноэлементной диагонали $A_{1,1}$).

29) Дана матрица размера $M \times N$. Элемент матрицы называется ее локальным максимумом, если он больше всех окружающих его элементов. Поменять знак всех локальных максимумов данной матрицы на противоположный.

30) Дана квадратная матрица A порядка M. Найти среднее арифметическое элементов каждой ее диагонали, параллельной главной (начиная с одноэлементной диагонали $A_{1,M}$).

Задача 8.3. Решить следующие задачи, используя стек.

1) Дан стек. Необходимо перевернуть его содержимое так, чтобы верхний элемент стал нижним, а нижний — верхним.

2) Дан стек. Необходимо удалить из него все элементы, которые не являются простыми числами.

3) Дан стек. Необходимо отсортировать его содержимое по возрастанию или убыванию, в зависимости от переданного параметра (False — по возрастанию, True — по убыванию).

4) Дан стек. Необходимо проверить, есть ли в нем повторяющиеся элементы. Вывести повторяющиеся элементы, если они есть.

5) Дан стек. Необходимо найти максимальный или минимальный элемент в нем, в зависимости от переданного параметра (True — максимальный, False — минимальный).

6) Дан стек и два элемента A и B. Необходимо удалить из стека все элементы, которые находятся между A и B (включая сами A и B).

7) Дан стек. Необходимо разделить его на два так, чтобы в первом были элементы с четными индексами, а во втором — с нечетными.

8) Дан стек. Необходимо проверить, является ли его содержимое палиндромом (то есть читается одинаково как слева направо, так и справа налево).

9) Дан стек. Необходимо удалить из него все отрицательные элементы.

10) Дан стек. Необходимо проверить, является ли его содержимое последовательностью арифметической прогрессии.

11) Дан стек. Необходимо найти среднее арифметическое всех его элементов.

12) Дан стек и значение A. Необходимо удалить из стека все элементы, которые больше A.

13) Дан стек и число k. Необходимо найти k-й по счету элемент в стеке.

14) Дан стек. Необходимо проверить, является ли его содержимое последовательностью геометрической прогрессии.

15) Дан стек и число N, необходимо удалить из стека все элементы, которые кратны N.

Задача 8.4. Используя очередь, решить следующую задачу.

1) Создать класс очереди, который будет хранить только уникальные элементы. При добавлении элемента, если он уже есть в очереди, то он не должен добавляться.

2) Создать класс очереди, который будет хранить только четные или только нечетные числа в зависимости от параметра, передаваемого при инициализации объекта класса очереди (True — хранить только четные числа, False — нечетные числа). При добавлении элемента, если его значение не соответствует заданному условию, то он не должен добавляться.

3) Создать класс очереди с ограниченной емкостью. Если при добавлении элемента очередь уже заполнена, то новый элемент должен заменить первый элемент в очереди.

4) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и удаления элемента из середины очереди по индексу.

5) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди, удаления элемента из начала очереди и сортировки элементов в очереди по возрастанию.

6) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди, удаления элемента из начала очереди и поиска минимального элемента в очереди.

7) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и удаления всех элементов из очереди, которые меньше заданного значения.

8) Создать класс двусторонней очереди, который будет поддерживать операции добавления элемента в начало и конец очереди, удаления элемента из начала и конца очереди, а также удаления минимального и максимального элементов из очереди.

9) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и удаления всех повторяющихся элементов из очереди.

10) Создать класс очереди, который будет хранить только элементы определенного типа данных. Тип элементов задается при инициализации объекта класса очереди. При добавлении элемента, если его тип не соответствует заданному, то он не должен добавляться.

11) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и удаления элемента из середины очереди по значению.

12) Создать класс очереди, который будет хранить только элементы, кратные заданному числу, которое определяется при инициализации объекта класса очереди. При добав-

лении элемента, который не является кратным заданному числу, то он не должен добавляться.

13) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и добавления элемента в середину очереди по индексу.

14) Создать класс очереди, который будет поддерживать операции добавления элемента в конец очереди и удаления всех элементов из очереди, которые больше заданного значения.

15) Создать класс очереди, который будет хранить только простые числа. При добавлении элемента, не являющийся простым числом, то он не должен добавляться.

Задача 8.5. Создайте двусвязный список согласно условию задачи.

1) Создайте двусвязный список для хранения и управления информацией о студентах в университете. Каждый элемент списка должен содержать имя, фамилию, номер студенческого билета и список курсов, на которые студент записан.

2) Создайте двусвязный список для хранения информации о покупках в интернет-магазине. Каждый элемент списка должен содержать название товара, цену, количество и дату покупки.

3) Создайте двусвязный список для хранения информации о задачах в проекте. Каждый элемент списка должен содержать название задачи, описание, дату начала и дату окончания.

4) Создайте двусвязный список для хранения информации о заказах в ресторане. Каждый элемент списка должен содержать название блюда, цену, количество, дату заказа и статус заказа (например, «в обработке», «готов», «доставлен»).

5) Создайте двусвязный список для хранения информации о задачах в календаре. Каждый элемент списка должен содержать название задачи, описание, дату начала и дату окончания, а также список напоминаний о задаче (например, за день до окончания, за час до начала и т.д.).

6) Создайте двусвязный список для хранения информации о задачах в системе управления проектами для строительства зданий. Каждый элемент списка должен содержать название задачи, описание, дату начала и дату окончания, а также информацию о том, какие инженеры и рабочие работают над этой задачей.

7) Создайте двусвязный список для хранения информации о пациентах в больнице. Каждый элемент списка должен содержать имя и фамилию пациента, диагноз, дату поступления и выписки, а также список проведенных процедур и назначенных лекарств.

8) Создайте двусвязный список для хранения информации о задачах в проекте по разработке программного обеспечения. Каждый элемент списка должен содержать название задачи, описание, дату начала и дату окончания, а также информацию о том, какие разработчики работают над этой задачей.

9) Создайте двусвязный список для хранения информации о посетителях музея. Каждый элемент списка должен содержать имя и фамилию посетителя, дату посещения, список экспонатов, которые он посмотрел, и оценку каждого из них.

10) Создайте двусвязный список для хранения информации о заказах в интернет-магазине. Каждый элемент списка должен содержать номер заказа, дату создания, список товаров, их количество и стоимость, а также информацию о доставке и оплате.

11) Создайте двусвязный список для хранения информации о студентах в университете. Каждый элемент списка должен содержать имя, фамилию, номер группы, список предметов, которые студент изучает, и оценки по каждому предмету.

12) Создайте двусвязный список для хранения информации о сотрудниках в компании. Каждый элемент списка должен содержать имя, фамилию, должность, дату приема на работу и список проектов, над которыми сотрудник работает.

13) Создайте двусвязный список для хранения информации о задачах в игре. Каждый элемент списка должен содержать название задачи, описание, уровень сложности, дату начала и дату окончания, а также список игроков, которые работают над этой задачей.

14) Создайте двусвязный список для хранения информации о книгах в библиотеке. Каждый элемент списка должен содержать название книги, автора, год издания, количество экземпляров и список читателей, которые взяли эту книгу.

15) Создайте двусвязный список для хранения информации о задачах в системе управления проектами. Каждый элемент списка должен содержать название задачи, описание, дату начала и дату окончания, а также список комментариев и прикрепленных файлов к задаче.

Задача 8.6. Реализуйте функцию для работы с двусвязным списком.

1) Реализовать функцию, которая переворачивает двусвязный список.

2) Реализовать функцию, которая объединяет два отсортированных двусвязных списка в один отсортированный список.

3) Реализовать функцию, которая проверяет, является ли двусвязный список палиндромом.

4) Реализовать функцию, которая удаляет все повторяющиеся элементы из двусвязного списка.

5) Реализовать функцию, которая находит средний элемент в двусвязном списке.

6) Реализовать функцию, которая находит *k*-й элемент с конца в двусвязном списке.

7) Реализовать функцию, которая удаляет каждый *n*-й элемент из двусвязного списка.

8) Реализовать функцию, которая находит пересечение двух двусвязных списков.

9) Реализовать функцию, которая находит сумму двух чисел, представленных в виде двусвязных списков.

10) Реализовать функцию, которая разделяет двусвязный список на два списка, один из которых содержит все элементы, меньшие заданного значения, а другой — все элементы, большие или равные заданному значению.

11) Реализовать функцию, которая находит среднее арифметическое всех элементов двусвязного списка.

12) Реализовать функцию, которая удаляет все элементы с заданным значением из двусвязного списка.

13) Реализовать функцию, которая находит медиану двусвязного списка.

14) Реализовать функцию, которая находит сумму всех элементов двусвязного списка, кроме первого и последнего.

15) Реализовать функцию, которая находит сумму всех элементов двусвязного списка, удовлетворяющих заданному условию.

Задача 8.7. Реализуйте функцию для работы с двусвязным списком.

1) Реализовать функцию, которая находит индекс первого вхождения заданного элемента в кольцевой двусвязный список.

2) Реализовать функцию, которая перемещает заданное количество элементов из начала кольцевого двусвязного списка в его конец.

3) Реализовать функцию, которая переворачивает кольцевой двусвязный список.

4) Реализовать функцию, которая удаляет все элементы кольцевого двусвязного списка, равные заданному значению.

5) Реализовать функцию, которая сортирует кольцевой двусвязный список по возрастанию.

6) Реализовать функцию, которая удаляет все элементы кольцевого двусвязного списка, меньшие заданного значения.

7) Реализовать функцию, которая возвращает элемент по заданному индексу в кольцевом двусвязном списке.

8) Реализовать функцию, которая вставляет элемент на заданную позицию в кольцевом двусвязном списке.

9) Реализовать функцию, которая удаляет все элементы кольцевого двусвязного списка, большие заданного значения.

10) Реализовать функцию, которая проверяет, содержится ли заданный элемент в кольцевом двусвязном списке.

11) Реализовать функцию, которая удаляет все повторяющиеся элементы из кольцевого двусвязного списка.

12) Реализовать функцию, которая находит максимальный элемент в кольцевом двусвязном списке.

13) Реализовать функцию, которая сортирует кольцевой двусвязный список по убыванию.

14) Реализовать функцию, которая проверяет, является ли кольцевой двусвязный список палиндромом.

15) Реализовать функцию, которая возвращает сумму всех элементов кольцевого двусвязного списка.

9. Алгоритмы поиска и сортировки

9.1. Поиск в списках/массивах. Бинарный поиск

Поиск в списках/массивах — это процесс нахождения определенного элемента в заданном списке или массиве. Существует несколько методов поиска, каждый из которых имеет свои преимущества и недостатки. Выбор метода поиска зависит от размера списка, упорядоченности элементов, распределения элементов и доступности дополнительной памяти.

- *Линейный поиск*. Этот метод заключается в последовательном переборе всех элементов списка до тех пор, пока не будет найден искомый элемент или пока не будут пройдены все элементы списка. Линейный поиск прост в реализации, но его время выполнения может быть довольно большим, особенно если список очень большой.
- *Бинарный поиск*. Этот метод подходит для упорядоченных списков. Он заключается в делении списка пополам и проверке, находится ли искомый элемент в первой или второй половине списка. Если элемент находится в первой половине, поиск продолжается только в этой части списка, если он находится во второй половине, поиск продолжается только в этой части списка. Этот метод быстрее линейного поиска, но требует упорядоченности списка и дополнительной памяти для хранения индексов.
- *Интерполяционный поиск*. Этот метод подходит для равномерно распределенных элементов. Он основан на формуле, которая предсказывает местоположение элемента в списке на основе его значения. Этот метод быстрее линейного и бинарного поиска, но может давать неправильные результаты для неравномерно распределенных элементов.
- *Хеш-таблицы*. Этот метод использует хеш-функцию для преобразования ключа элемента в индекс в массиве. Если два ключа имеют одинаковое значение хеш-функции, они помещаются в одну ячейку массива (это называется коллизией). Хеш-таблицы быстрее всех других методов, но требуют больше памяти и сложнее в реализации.

Бинарный поиск — это алгоритм поиска элемента в отсортированном массиве, который осуществляет поиск путем деления массива пополам и проверки элемента в середине.

Если элемент равен искомому, поиск завершается. Если искомый элемент меньше элемента в середине, то поиск продолжается только в левой половине массива, иначе — только в правой. Этот процесс повторяется до тех пор, пока искомый элемент не будет найден или не останется больше элементов для проверки.

Бинарный поиск имеет сложность $O(\log n)$, что делает его очень эффективным для поиска в больших массивах. Однако он работает только для отсортированных массивов, поэтому если массив не отсортирован, нужно сначала выполнить сортировку, что может занять дополнительное время.

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Рисунок 9.1 — Пример реализации бинарного поиска

На рис. 9.1 функция binary_search() принимает два аргумента: отсортированный массив arr и элемент x, который нужно найти. Внутри функции создаются переменные low и high, которые задают границы поиска — начальный и конечный индексы массива. Затем запускается цикл while, который продолжается, пока low не станет больше high

Внутри цикла вычисляется индекс среднего элемента массива с помощью формулы (low + high) // 2. Если значение в середине равно искомому элементу, то функция возвращает его индекс. Если значение в середине меньше искомого, то поиск продолжается только в правой половине массива, поэтому переменная low устанавливается на значение mid + 1. Если значение в середине больше искомого, то поиск продолжается только в левой половине массива, поэтому переменная high устанавливается на значение mid – 1. Если элемент не найден, то функция возвращает –1.

```
import random

# создание списка из 20 случайных чисел от 0 до 99
arr = random.sample(range(100), 20)
arr
```

```
[43, 48, 84, 9, 72, 2, 77, 11, 89, 68, 94, 57, 28, 83, 50, 67, 96, 59, 56, 90]
```

```
# сортировка списка и определение элемента x для поиска
arr.sort()
x = 77
arr
```

```
[2, 9, 11, 28, 43, 48, 50, 56, 57, 59, 67, 68, 72, 77, 83, 84, 89, 90, 94, 96]
```

```
# поиск элемента в списке
result = binary_search(arr, x)
if result != -1:
    print("Элемент найден на позиции", result)
else:
    print("Элемент не найден")
```

```
Элемент найден на позиции 13
```

Рисунок 9.2 — Пример бинарного поиска

9.2. Простые методы сортировки

Сортировка — это процесс упорядочивания элементов в определенном порядке, который находит свое применение в различных областях компьютерных наук и используется для решения множества прикладных задач. Существует множество алгоритмов сортировки, каждый из которых имеет свои уникальные преимущества и недостатки. Наиболее распространенные алгоритмы сортировки включают в себя сортировку пузырьком, выбором, вставками и быструю сортировку.

Одним из наиболее распространенных применений сортировки является поиск и выборка данных. Сортировка может использоваться для поиска определенного значения в массиве данных или для выборки элементов из базы данных в порядке убывания или возрастания. Использование сортировки может быть полезным в различных прикладных задачах, таких как ускорение поиска элемента в списке, фильтрация данных, обработка больших объемов данных и анализ данных.

Сортировка также используется в алгоритмах машинного обучения, где необходимо отсортировать данные по определенному признаку. Это может быть полезно, например, для обучения модели предсказывать стоимость недвижимости на основе ее характеристик. Кроме того, сортировка широко используется в компьютерной графике и играх, где необходимо отображать объекты в определенном порядке. При отображении объектов на экране игры необходимо отсортировать их по глубине, чтобы правильно определить порядок их отображения. Таким образом, сортировка является важным инструментом в различных областях компьютерных наук и имеет множество применений в прикладных задачах. Выбор определенного алгоритма сортировки зависит от конкретной задачи, которую необходимо решить.

Обменная сортировка — это один из простейших алгоритмов сортировки, который основан на поочередном сравнении и обмене соседних элементов массива.

Простая обменная сортировка, также известная как сортировка пузырьком (Bubble Sort), является одним из наиболее простых алгоритмов сортировки. Этот алгоритм получил свое название из-за того, что наибольшие элементы массива «всплывают» на поверхность по мере прохода по массиву. Суть алгоритма заключается в том, что на каждом проходе сравниваются пары соседних элементов, и если они находятся в неправильном порядке, то они меняются местами. Процесс продолжается до тех пор, пока массив не будет отсортирован.

Алгоритм сортировки пузырьком состоит из следующих шагов:

- Проходим по всем элементам массива данных, начиная с первого элемента.

- Сравниваем текущий элемент со следующим элементом.
- Если текущий элемент больше следующего элемента, меняем их местами.
- Продолжаем проходить по массиву данных до тех пор, пока все элементы не будут отсортированы.

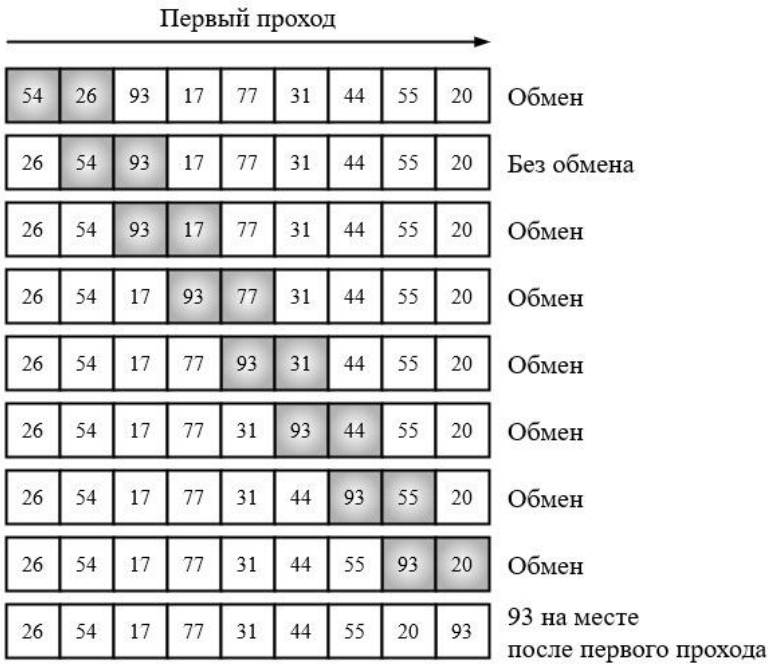


Рисунок 9.3 — Пример первого прохода алгоритма сортировки пузырьком

Пример реализации алгоритма сортировки пузырьком показан на рис. 9.4.

```
# реализация алгоритма сортировки пузырьком
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j+1], arr[j]
    return arr
```

Рисунок 9.4 — Реализация алгоритма сортировки пузырьком

Сначала определяется длина списка n. Затем внешний цикл for i in range(n) проходит по элементам списка arr от первого до последнего, а внутренний цикл for j in range(n – i – 1) проходит по элементам списка arr от первого до предпоследнего, т.е. на каждом проходе внешнего цикла самый большой элемент «всплывает» на последнюю позицию. Внутри второго цикла проверяется, если текущий элемент arr[j] больше следующего за ним элемента arr[j + 1], то они меняются местами с помощью оператора присваивания arr[j], arr[j + 1] = arr[j + 1], arr[j]. Таким образом, на каждом проходе внутреннего цикла самый большой элемент «всплывает» на последнюю позицию. После завершения работы внешнего

цикла, список arr будет отсортирован в порядке возрастания. Отсортированный список возвращается из функции.

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {bubble_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.5 — Использование сортировки пузырьком

Можно реализовать алгоритм сортировки пузырьком с параметром reverse (False — в порядке возрастания, True — в порядке убывания) (рис. 9.6).

```
# реализация алгоритма сортировки пузырьком с параметром reverse
# (False – в порядке возрастания, True – в порядке убывания)
def bubble_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if not reverse:
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
            else:
                if arr[j] < arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список:\n\t{arr}')
print(f'\nОтсортированный список:\n\t{bubble_sort(arr)} – в порядке возрастания')
print(f'\t{bubble_sort(arr, reverse=True)} – в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] – в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] – в порядке убывания
```

Рисунок 9.6 — Реализовать алгоритма сортировки пузырьком с параметром reverse

Шейкерная сортировка (Cocktail Shaker Sort) — это усовершенствованный алгоритм сортировки пузырьком является усовершенствованным алгоритмом пузырьковой сортировки. Он работает следующим образом:

- Задаем начальный и конечный индексы для массива, которые соответствуют первому и последнему элементам.
- Создаем переменную (флаг), которая будет использоваться для определения того, были ли произведены какие-либо перестановки на данной итерации.
- Пока начальный индекс меньше конечного индекса, повторяем следующие шаги:
    - Проходимся по массиву от начального до конечного индекса и сравниваем соседние элементы. Если текущий элемент больше следующего, меняем их местами и устанавливаем флаг в значение True.

- Если флаг остался в значении False, то выходим из цикла, так как массив уже отсортирован.
- Уменьшаем конечный индекс на 1 и повторяем шаги a и b.

- Пока начальный индекс меньше конечного индекса, повторяем следующие шаги:
  - Проходимся по массиву от конечного до начального индекса и сравниваем соседние элементы. Если текущий элемент меньше предыдущего, меняем их местами и устанавливаем флаг в значение True.
  - Если флаг остался в значении False, то выходим из цикла, так как массив уже отсортирован.
  - Увеличиваем начальный индекс на 1 и повторяем шаги a и b.
- Массив отсортирован.

Шейкерная сортировка более эффективна, чем сортировка пузырьком, так как она может обработать список с меньшим количеством итераций. Однако для больших списков, эта сортировка все еще является довольно медленной.

```
# реализация алгоритма шейкерной сортировки
def cocktail_sort(arr):
    n = len(arr)
    start = 0
    end = n - 1
    swapped = True
    while swapped:
        swapped = False
        for i in range(start, end):
            if (arr[i] > arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end = end - 1
        for i in range(end - 1, start - 1, -1):
            if (arr[i] > arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start = start + 1
    return arr
```

Рисунок 9.7 — Реализация алгоритма шейкерной сортировки

Задаются начальные значения переменных: n — длина массива, start — начальный индекс, end — конечный индекс, swapped — флаг, указывающий на то, были ли произведены перестановки на текущей итерации. Запускается цикл while, который продолжается до тех пор, пока на текущей итерации были произведены перестановки (т.е. пока флаг swapped равен True). Внутри цикла while запускается два цикла for:

- первый цикл проходит по элементам массива от индекса start до индекса end и проверяет, нужно ли производить перестановку между текущим и следующим элементами (если текущий элемент больше следующего);
- если на текущей итерации была произведена хотя бы одна перестановка, флаг swapped становится равен True;
- после первого цикла проверяется значение флага swapped: если он равен False, то это означает, что на текущей итерации не было произведено ни одной перестановки, и цикл while прерывается;
- если на текущей итерации были произведены перестановки, флаг swapped сбрасывается в False, и запускается второй цикл for;
- второй цикл проходит по элементам массива от индекса end до индекса start и проверяет, нужно ли производить перестановку между текущим и следующим элементами (если текущий элемент больше следующего);
- если на текущей итерации была произведена хотя бы одна перестановка, флаг swapped становится равен True;
- после второго цикла обновляются значения переменных start и end: start увеличивается на 1, end уменьшается на 1.

После завершения цикла while возвращается отсортированный массив arr.

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {cocktail_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.8 — Использование шейкерной сортировки

Реализация алгоритма шейкерной сортировки с параметром reverse (False — в порядке возрастания, True — в порядке убывания) показана на рис. 9.9.

```python
# реализация алгоритма шейкерной сортировки с параметром reverse
# (False – в порядке возрастания, True – в порядке убывания)
def cocktail_sort(arr, reverse=False):
    n = len(arr)
    start = 0
    end = n - 1
    swapped = True
    while swapped:
        swapped = False
        for i in range(start, end):
            if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break
        swapped = False
        end = end - 1
        for i in range(end - 1, start - 1, -1):
            if (not reverse and arr[i] > arr[i + 1]) or (reverse and arr[i] < arr[i + 1]):
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        start = start + 1
    return arr
```

Рисунок 9.9 — Реализация алгоритма шейкерной сортировки с параметром reverse

```python
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список:\n\t{arr}\n')
print(f'Отсортированный список:\n\t{cocktail_sort(arr)} – в порядке возрастания')
print(f'\t{cocktail_sort(arr, reverse=True)} – в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] – в порядке возрастания
        [90, 64, 34, 25, 22, 12, 11] – в порядке убывания
```

Рисунок 9.10 — Использование шейкерной сортировки с параметром reverse

Сортировка расческой (Comb Sort) — это алгоритм сортировки, который является улучшенной версией сортировки пузырьком. Он был разработан в 1980 году в команде сотрудников компании Hewlett-Packard.

Основная идея этого алгоритма заключается в том, что он использует большие промежутки для сравнения элементов массива, а затем постепенно уменьшает их до минимального значения (обычно 1). Это позволяет ему быстро перемещать большие элементы в конец массива и уменьшить количество обменов. Алгоритм работает следующим образом:

- Задаем начальный размер промежутка равный длине массива.
- Сравниваем элементы на расстоянии равном заданному промежутку и меняем их местами, если необходимо.
- Уменьшаем промежуток на определенный коэффициент (обычно 1.3).
- Повторяем шаги 2–3 до тех пор, пока промежуток не станет равным 1.
- Выполняем обычную сортировку пузырьком для окончательной сортировки.

Сложность алгоритма составляет $O(n^2)$, но в среднем он работает быстрее, чем сортировка пузырьком. Если коэффициент уменьшения промежутка выбран неправильно, то может возникнуть проблема «зубчатости» массива, когда элементы не могут перемещаться на достаточно большие расстояния, что приводит к ухудшению производительности алгоритма.

На рис. 9.11 показана реализация алгоритма сортировки расческой. Сначала задаются начальные значения переменных: n определяет длину массива, gap задает начальный размер промежутка между элементами, которые будут сравниваться и меняться местами, shrink задает коэффициент сжатия, который определяет, насколько уменьшается gap на каждой итерации, swapped отвечает за то, был ли выполнен обмен элементов в последней итерации.

```
# реализация алгоритма сортировки расческой
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True
    while swapped:
        gap = int(gap/shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = False
        while i + gap < n:
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True
            i += 1
    return arr
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {comb_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.11 — Реализация алгоритма сортировки расческой

Цикл while выполняется, пока swapped равно True. На каждой итерации уменьшается gap и проверяется, не стало ли оно меньше 1. Если gap меньше 1, то оно устанавливается в 1. Переменная i устанавливается в 0, а swapped — в False. Вложенный цикл while выполняется до тех пор, пока i + gap меньше n. Если элемент arr[i] больше элемента arr[i + gap], то они меняются местами, а переменная swapped устанавливается в True. После выполнения вложенного цикла while функция возвращает отсортированный массив arr.

На рис. 9.12 показана реализация алгоритма сортировки расческой с параметром reverse (False — в порядке возрастания, True — в порядке убывания).

```python
# реализация алгоритма сортировки расческой с параметром reverse
# (False – в порядке возрастания, True – в порядке убывания)
def comb_sort(arr, reverse=False):
    n = len(arr)
    gap = n
    shrink = 1.3
    swapped = True
    while swapped:
        gap = int(gap/shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = False
        while i+gap < n:
            if (not reverse and arr[i] > arr[i+gap]) or (reverse and arr[i] < arr[i+gap]):
                arr[i], arr[i+gap] = arr[i+gap], arr[i]
                swapped = True
            i += 1
    return arr
```

Рисунок 9.12 — Реализация алгоритма сортировки расческой с параметром reverse

```python
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список:\n\t{arr}\n')
print(f'Отсортированный список:\n\t{comb_sort(arr)} – в порядке возрастания')
print(f'\t{comb_sort(arr, reverse=True)} – в порядке убывания')
```

```
Исходный список:
        [64, 34, 25, 12, 22, 11, 90]

Отсортированный список:
        [11, 12, 22, 25, 34, 64, 90] – в порядке возрастания
        [64, 90, 22, 25, 34, 11, 12] – в порядке убывания
```

Рисунок 9.13 — Использование сортировки расческой с параметром reverse

Сортировка выбором или извлечением (Selection Sort) — это алгоритм сортировки, который на каждой итерации находит максимальный (или минимальный) элемент в неотсортированной части массива и перемещает его в конец (или начало) отсортированной части.

Алгоритм сортировки выбором можно реализовать следующим образом:

- Найти наибольший элемент в неотсортированной части массива.
- Обменять его с последним элементом в неотсортированной части массива.
- Пометить последний элемент как отсортированный.
- Повторять шаги 1–3 для оставшейся части массива, пока все элементы не будут отсортированы.

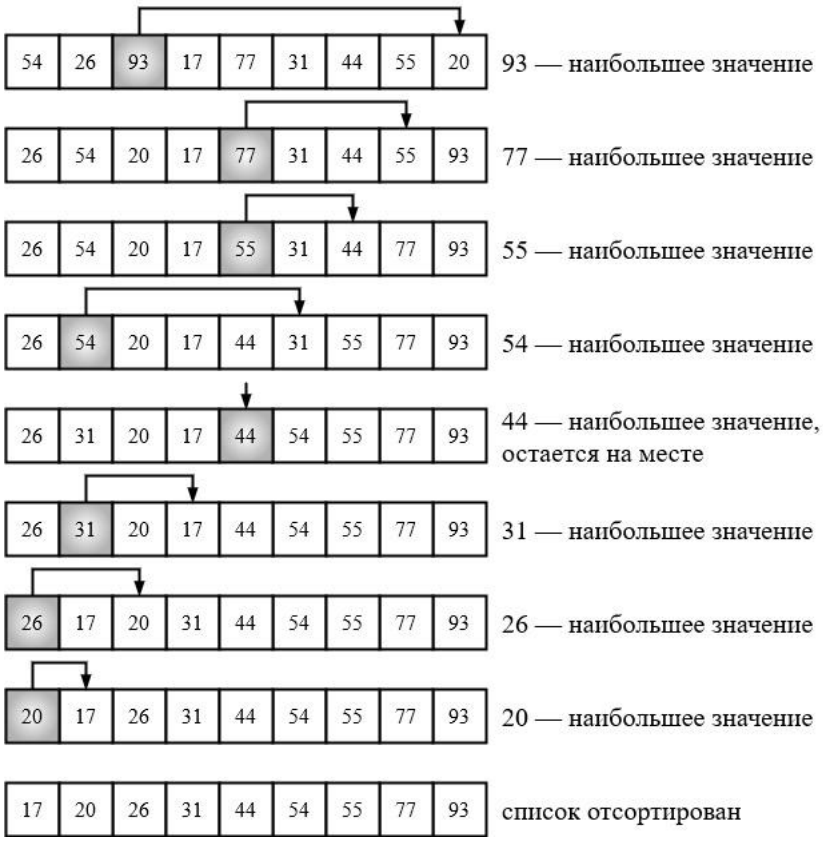


Рисунок 9.13 — Пример работы алгоритма сортировки выбором

Аналогичным образом можно искать наименьший элемент в неотсортированной части массива. После нахождения наименьшего элемента он будет обмениваться с первым элементом в неотсортированной части массива.

С учетом того, что количество рассматриваемых на очередном шаге элементов уменьшается на единицу, общее количество операций:

$$(n-1)+(n-2)+(n-3)+\ldots+1=\frac{1}{2(n-1)(n-1+1)}=\frac{1}{2(n^2-n)}=O(n^2).$$

По сравнению с обменной сортировкой существенно меньше перестановок элементов $O(n)$ по сравнению $O(n^2)$, но нет возможности быстро отсортировать почти отсортированный массив.

Естественной идеей улучшения алгоритма выбором является идея использования информации, полученной при сравнении элементов при поиске максимального (минимального) элемента на предыдущих шагах.

В общем случае, если n — точный квадрат, можно разделить массив на $\sqrt{n}$ групп по $\sqrt{n}$ элементов и находить максимальный элемент в каждой подгруппе. Любой выбор, кроме первого, требует не более чем $\sqrt{n-2}$ сравнений внутри группы ранее выбранного элемента плюс $\sqrt{n-1}$ сравнений среди «лидеров групп». Этот метод получил название квадратичный выбор, общее время его работы составляет порядка $O(n\sqrt{n})$, что сущест-

венно лучше, чем $O(n^2)$. Также сортировка выбором является неустойчивой. Это означает, что если в исходном массиве имеются равные элементы, то после сортировки выбором их порядок может быть изменен. Например, если в массиве есть два элемента со значением 5, то после сортировки выбором они могут поменяться местами, т.е. порядок равных элементов не гарантируется. В отличие от устойчивых сортировок, где порядок равных элементов сохраняется.

```
# реализация алгоритма сортировки выбором
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {selection_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.14 — Реализация алгоритма сортировки выбором

На рис. 9.14 переменная n определяет длину массива arr. Затем начинается цикл по переменной i, который пробегает все значения от 0 до n – 1. В каждой итерации цикла определяется индекс элемента с минимальным значением (min_idx) в подмассиве, начинающемся с i и заканчивающемся в конце массива. Для этого используется вложенный цикл по переменной j, который пробегает все значения от i + 1 до n – 1. Если значение элемента с индексом j меньше значения элемента с индексом min_idx, то значение min_idx обновляется. После того, как найден индекс элемента с минимальным значением в подмассиве, происходит обмен местами элементов с индексами i и min_idx. После завершения всех итераций основного цикла возвращается отсортированный массив arr. Также можно реализовать алгоритм сортировки выбором с параметром reverse (False — в порядке возрастания, True — в порядке убывания) (рис. 9.15).

```
def selection_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if reverse:
                if arr[j] > arr[min_idx]:
                    min_idx = j
            else:
                if arr[j] < arr[min_idx]:
                    min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Рисунок 9.15 — Реализация алгоритма сортировки выбором с параметром reverse

Сортировка включением или вставками (Insertion Sort) — это алгоритм сортировки, который работает путем постепенного построения отсортированного массива. На каждом шаге алгоритм берет очередной элемент массива и вставляет его в правильную позицию в уже отсортированном массиве.

Алгоритм сортировки включением имеет следующий шаги:

- Начинаем с пустого массива, который считается отсортированным.
- Берем первый элемент неотсортированного массива и вставляем его в правильную позицию в отсортированном массиве.
- Берем следующий элемент неотсортированного массива и вставляем его в правильную позицию в отсортированном массиве.
- Продолжаем этот процесс до тех пор, пока не все элементы неотсортированного массива не будут вставлены в отсортированный массив.

Алгоритм имеет сложность $O(n^2)$, но в случае исходно отсортированного массива внутренний цикл не будет выполняться ни разу, поэтому метод имеет в этом случае временную сложность $O(n)$. Сортировка вставками является эффективным алгоритмом для маленьких наборов данных, кроме того, на практике он более эффективен, чем остальные простые квадратичные сортировки. На рис. 9.16 и 9.17 приведены примеры работы алгоритма.

Рисунок 9.16 — Пример работы алгоритма сортировки вставками

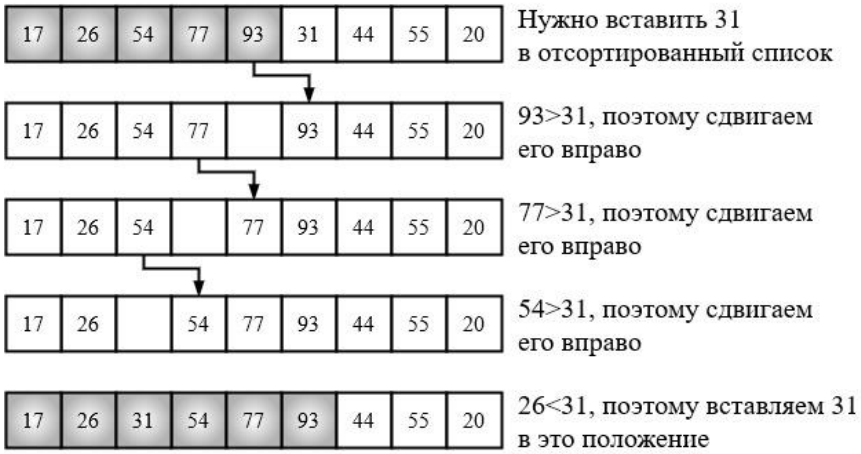


Рисунок 9.17 — Пример работы пятого прохода алгоритма сортировки вставками

На рис. 9.18 функция insertion_sort() принимает на вход массив arr, который нужно отсортировать. Далее начинается цикл for, который проходит по всем элементам массива arr, начиная со второго (индекс 1). Внутри цикла определяется переменная key, которая содержит значение текущего элемента массива arr. Затем определяется переменная j, которая указывает на предыдущий элемент массива arr (т.е. на элемент с индексом i – 1). Далее начинается цикл while, который выполняется до тех пор, пока j не станет меньше нуля или значение элемента arr[j] не станет меньше или равно key. Внутри цикла while происходит сдвиг элементов массива arr вправо, начиная с индекса j + 1 и заканчивая индексом i. После того как цикл while завершается, элемент key вставляется на свое место в отсортированную часть массива arr путем присваивания его значения элементу arr[j + 1]. Цикл for продолжает свою работу, переходя к следующему элементу массива arr, который также становится частью отсортированной последовательности. После завершения цикла for, отсортированный массив arr возвращается из функции.

```
# реализация алгоритма сортировки вставками
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {insertion_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.18 — Реализация алгоритма вставками

На рис. 9.19 представлена реализация алгоритма сортировки вставками с параметром reverse (False — в порядке возрастания, True — в порядке убывания).

```
def insertion_sort(arr, reverse=False):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and ((not reverse and arr[j] > key) or (reverse and arr[j] < key)):
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Рисунок 9.19 — Реализация алгоритма сортировки вставками с параметром reverse

9.3. Эффективные методы сортировки

Быстрая сортировка (Quick Sort) — это алгоритм сортировки, основанный на принципе «разделяй и властвуй». Он был разработан Тони Хоаром в 1960 году и с тех пор стал одним из самых широко используемых алгоритмов сортировки.

Алгоритм быстрой сортировки можно разбить на несколько шагов:

- Выбрать разделяющий (опорный) элемент из массива.
- Разделить массив на две части: элементы, меньшие опорного, и элементы, большие разделяющего.
- Рекурсивно применить алгоритм быстрой сортировки к обеим частям массива.
- Объединить отсортированные части массива в один отсортированный массив.

Оптимальный выбор разделяющего элемента — это медиана массива, но такой подход может быть затратным. Вместо этого часто выбирают первый или последний элемент в массиве, либо случайный элемент.

Сложность алгоритма в худшем случае составляет $O(n^2)$, но в среднем и лучшем случаях его сложность равна $O(n\log n)$. Быстрая сортировка обладает высокой эффективностью и широко применяется в различных областях, например, в компьютерной графике, базах данных и машинном обучении. На рис. 9.20–9.22 представлены примеры работы алгоритма.

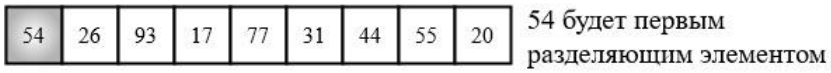


Рисунок 9.20 — Пример работы быстрой сортировки (выбор элемента для разделения массива)

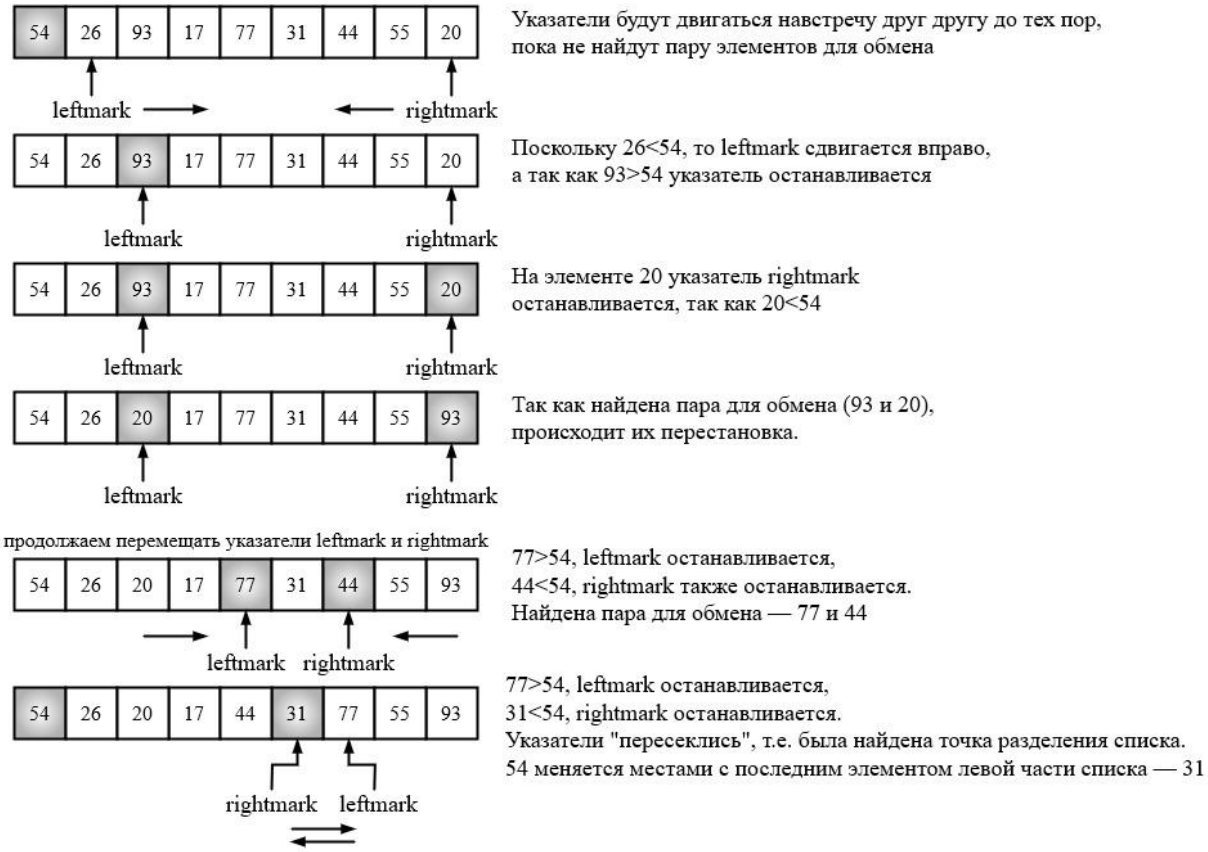


Рисунок 9.21 — Пример работы быстрой сортировки (разделения массива на две части: со значениями меньшими и большими, чем у разделяющего элемента)

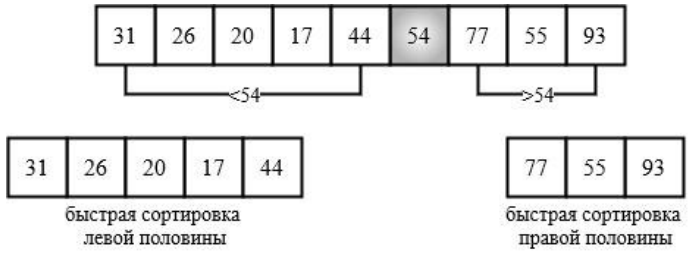


Рисунок 9.22 — Пример работы быстрой сортировки (подготовка к рекурсивному вызову сортировки двух подмассивов)

На рис. 9.23 показан пример реализации алгоритма быстрой сортировки.

```
def quick_sort(arr):
    quick_sort_helper(arr, 0, len(arr) - 1)
    return arr

def quick_sort_helper(arr, first, last):
    if first < last:
        split_point = partition(arr, first, last)
        quick_sort_helper(arr, first, split_point - 1)
        quick_sort_helper(arr, split_point + 1, last)

def partition(arr, first, last):
    pivot_value = arr[first]
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        while left_mark <= right_mark and arr[left_mark] <= pivot_value:
            left_mark += 1
        while arr[right_mark] >= pivot_value and right_mark >= left_mark:
            right_mark -= 1

        if right_mark < left_mark:
            done = True
        else:
            arr[left_mark], arr[right_mark] = arr[right_mark], arr[left_mark]

    arr[first], arr[right_mark] = arr[right_mark], arr[first]

    return right_mark
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {quick_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.23 — Реализация алгоритма быстрой сортировки

Функция quick_sort() вызывает вспомогательную функцию quick_sort_helper() для выполнения сортировки. Она принимает список arr, а также первый и последний индексы списка. Функция quick_sort_helper() рекурсивно вызывает себя для подсписков слева и справа от разделительного элемента (split_point). Функция partition() выбирает первый элемент списка (pivot_value) в качестве разделительного элемента. Затем она использует два указателя (left_mark и right_mark), чтобы перемещаться по списку и переставлять элементы так, чтобы все элементы, меньшие или равные pivot_value, находились слева от него, а все элементы, большие или равные ему, находились справа от него. После завершения цикла while в функции partition(), разделительный элемент правильно располагается на своей позиции в отсортированном списке. Затем разделительный элемент меняется местами с элементом на позиции right_mark. Функция возвращает позицию разделительного элемента, чтобы его можно было использовать для разделения списка на две части при рекурсивных вызовах.

Ожидаемое число обменов в быстром алгоритме — $(m-1)/6$, общее число сравнений $n\log n$. Наихудший случай — в качестве элемента для разбиения x выбирается наибольшее из всех значений в указанной области, т.е. левая часть состоит из $n-1$ элементов, а правая из 1, тогда временная сложность становится пропорциональна n^2.

На рис. 9.24 представлена реализация алгоритма быстрой сортировки с параметром reverse (False — в порядке возрастания, True — в порядке убывания).

```
def quick_sort(arr, reverse=False):
    quick_sort_helper(arr, 0, len(arr) - 1, reverse)
    return arr

def quick_sort_helper(arr, first, last, reverse):
    if first < last:
        split_point = partition(arr, first, last, reverse)
        quick_sort_helper(arr, first, split_point - 1, reverse)
        quick_sort_helper(arr, split_point + 1, last, reverse)

def partition(arr, first, last, reverse):
    pivot_value = arr[first]
    left_mark = first + 1
    right_mark = last
    done = False

    while not done:
        if not reverse:
            while left_mark <= right_mark and arr[left_mark] <= pivot_value:
                left_mark += 1
            while arr[right_mark] >= pivot_value and right_mark >= left_mark:
                right_mark -= 1
        else:
            while left_mark <= right_mark and arr[left_mark] >= pivot_value:
                left_mark += 1
            while arr[right_mark] <= pivot_value and right_mark >= left_mark:
                right_mark -= 1

        if right_mark < left_mark:
            done = True
        else:
            arr[left_mark], arr[right_mark] = arr[right_mark], arr[left_mark]

    arr[first], arr[right_mark] = arr[right_mark], arr[first]

    return right_mark
```

Рисунок 9.24 — Реализация алгоритма быстрой сортировки с параметром reverse

Сортировка Шелла (Shell Sort) — это алгоритм сортировки, который основан на идеи сравнения элементов, находящихся на определенном расстоянии друг от друга. Алгоритм был разработан Дональдом Шеллом в 1959 году.

Этот алгоритм является модификацией сортировки вставками, при которой сначала происходит сортировка элементов, находящихся на некотором расстоянии друг от друга, а затем это расстояние уменьшается и процесс повторяется до тех пор, пока расстояние не станет равным 1.

Принцип работы алгоритма:

- Задается расстояние между элементами, которое будет изменяться на каждой итерации алгоритма.
- Исходный массив разбивается на подмассивы длиной, равной заданному расстоянию.
- В каждом подмассиве выполняется сортировка вставками.
- Расстояние между элементами уменьшается на каждой итерации до тех пор, пока не будет достигнуто значение 1.
- Наконец, выполняется полная сортировка вставками.

Сложность алгоритма сортировки Шелла зависит от выбранного значения gap. В среднем, время выполнения алгоритма составляет $O(n\log n)$, но может достигать $O(n^2)$, если выбрать неправильный размер шага. Сортировка Шелла может использоваться в тех случаях, когда нужно отсортировать большой список элементов, но нет возможности загрузить все элементы в память одновременно. Это может быть полезно при работе с большими базами данных или при обработке больших файлов. Также сортировка Шелла может использоваться вместо сортировки вставками в тех случаях, когда нужно отсортировать список с несколькими повторяющимися значениями. В этом случае сортировка Шелла работает быстрее, чем сортировка вставками. На следующих рисунках приведен пример работы алгоритма.

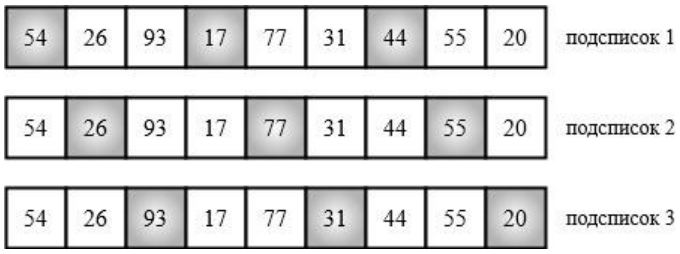


Рисунок 9.25 — Пример работы сортировки Шелла
(разбиение исходного массива с шагом 3)

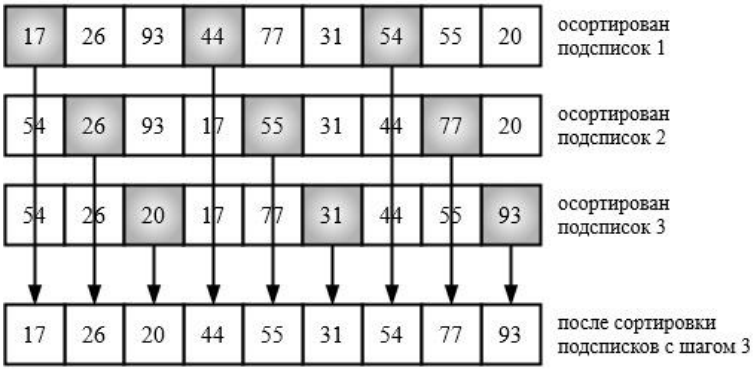


Рисунок 9.26 — Пример работы сортировки Шелла с шагом 3
(после сортировки каждого подсписка)

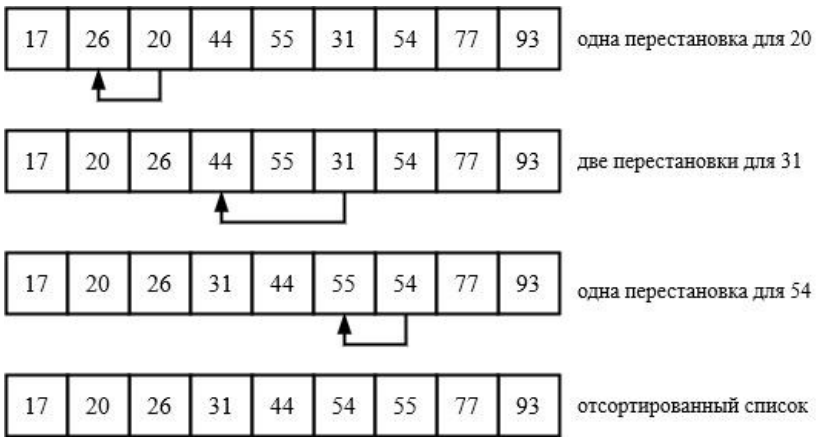


Рисунок 9.27 — Пример работы последней стадии сортировки Шелла с шагом 1 (сортировка вставкой)

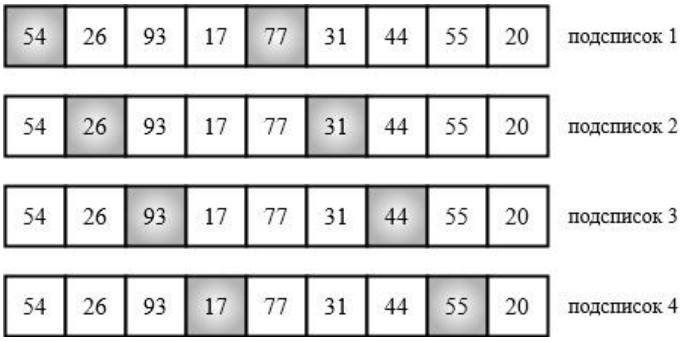


Рисунок 9.28 — Пример работы сортировки Шелла (разбиение исходного массива с шагом 4)

Пример реализации алгоритма сортировки Шелла показан на рис. 9.29.

```
# реализация алгоритма сортировки Шелла
def shell_sort(arr):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {shell_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.29 — Реализация алгоритма сортировки Шелла

Сначала определяется начальное значение расстояния между элементами — это половина длины списка. Затем цикл while выполняется до тех пор, пока расстояние не станет равным 0. Внутри цикла перебираются все элементы списка, начиная с индекса gap. Для

каждого элемента определяется его временное значение в переменной temp, а затем сравниваются элементы, находящиеся на расстоянии gap друг от друга. Если элемент слева больше элемента справа, то они меняются местами. Этот процесс повторяется до тех пор, пока не будут отсортированы все элементы на текущем расстоянии. Затем расстояние уменьшается вдвое и процесс повторяется снова. Наконец, отсортированный список возвращается из функции.

Реализация алгоритма сортировки Шелла с параметром reverse (False — в порядке возрастания, True — в порядке убывания) показана на рис. 9.30.

```
# реализация алгоритма сортировки Шелла с параметром reverse
# (False – в порядке возрастания, True – в порядке убывания)
def shell_sort(arr, reverse=False):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and ((not reverse and arr[j - gap] > temp)
                                or (reverse and arr[j - gap] < temp)):
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

Рисунок 9.30 — Реализация алгоритма сортировки Шелла с параметром reverse

Количество перестановок элементов по результатам экспериментов со случайным массивом иллюстрируется табл. 9.1.

Таблица 9.1 — Сравнение алгоритма сортировки Шелла
и алгоритма сортировки вставками

Размерность	$n = 25$	$n = 1000$	$n = 100000$
Сортировка Шелла	50	7700	2100000
Сортировка простыми вставками	150	240000	2.5 млрд

Многие полезные алгоритмы имеют рекурсивную структуру: для решения данной задачи они рекурсивно вызывают сами себя один или несколько раз, чтобы решить вспомогательную задачу, имеющую непосредственное отношение к поставленной задаче. Такие алгоритмы зачастую разрабатываются с помощью метода декомпозиции, или разбиения:

- сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем;
- далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получить решение исходной задачи.

Парадигма, лежащая в основе метода декомпозиции «разделяй и властвуй», на каждом уровне рекурсии включает в себя три этапа: 1) разделение задачи на несколько подзадач;

2) покорение — рекурсивное решение этих подзадач; когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно; 3) комбинирование решения исходной задачи из решений вспомогательных задач.

Сортировка слиянием (Merge Sort) в большой степени соответствует парадигме метода разбиения. На интуитивном уровне его работу можно описать таким образом.

- Разделение: сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит $n/2$ элементов.
- Покорение: сортировка обеих вспомогательных последовательностей методом слияния.
- Комбинирование: слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность можно считать упорядоченной. Основная операция, которая производится в процессе сортировки по методу слияний, — это объединение двух отсортированных последовательностей в ходе комбинирования (последний этап). Это делается с помощью вспомогательной процедуры слияния. В этой процедуре предполагается, что элементы подмассивов упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы. Для выполнения этой процедуры требуется время в $O(n)$, где n — количество подлежащих слиянию элементов. Временная сложность алгоритма сортировки слиянием можно определить как $O(n\log n)$. Он устойчивый, то есть сохраняет порядок элементов, которые имеют одинаковые значения. Его можно легко распараллелить для ускорения работы на многопроцессорных системах. На рис. 9.31 и 9.32 процессы разбиения и слияния массива в сортировке слиянием.

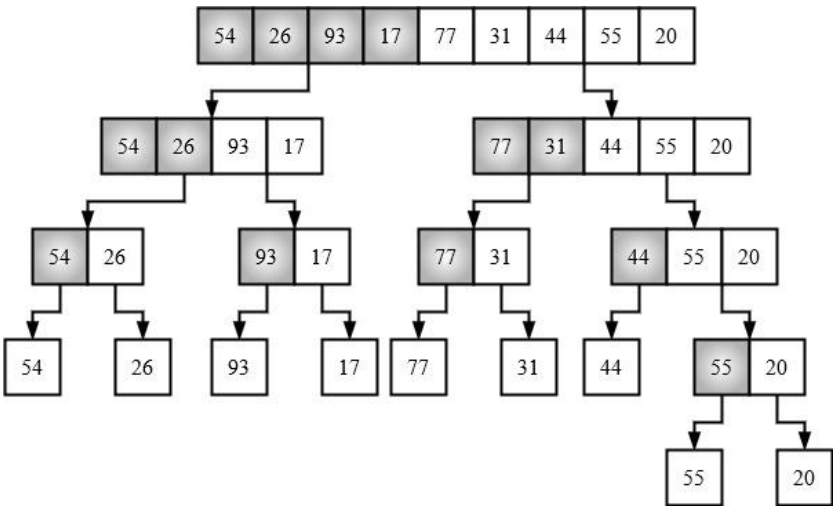


Рисунок 9.31 — Разбиение исходного массива в сортировке слиянием

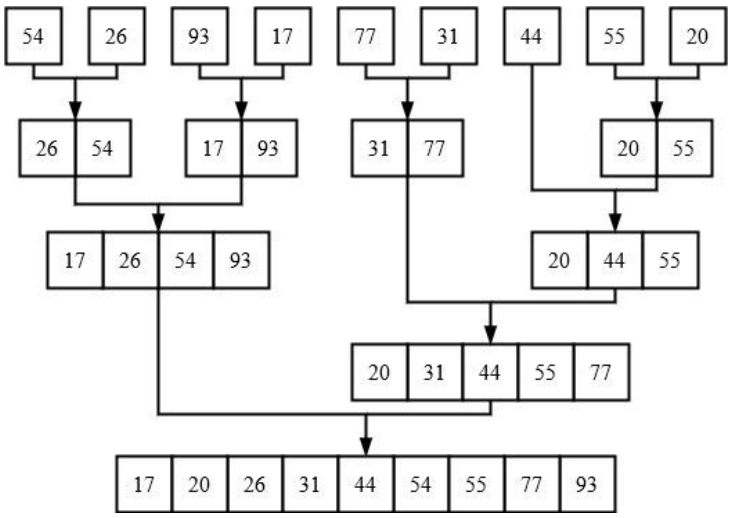


Рисунок 9.32 — Слияние разбитого массива в сортировке слиянием

```python
# реализация алгоритма сортировки слиянием
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

# вспомогательная функция для алгоритма сортировки слиянием
def merge(left_half, right_half):
    result = []
    i = 0
    j = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] <= right_half[j]:
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
            j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
```

```python
arr = [64, 34, 25, 12, 22, 11, 90]
print(f'Исходный список: {arr}\nОтсортированный список: {merge_sort(arr)}')
```

```
Исходный список: [64, 34, 25, 12, 22, 11, 90]
Отсортированный список: [11, 12, 22, 25, 34, 64, 90]
```

Рисунок 9.33 — Реализация алгоритма сортировки слиянием

Функция merge_sort() принимает на вход массив arr. Если длина массива меньше или равна 1, то функция возвращает этот же массив. Иначе функция находит середину массива

(mid), разбивает его на две половины (left_half и right_half) и рекурсивно вызывает себя для каждой половины, чтобы отсортировать их. Затем функция вызывает функцию merge, которая объединяет отсортированные половины в один отсортированный массив.

Вспомогательная функция merge принимает на вход два массива (left_half и right_half). Она создает пустой массив result, а также два счетчика i и j, соответствующие индексам элементов в left_half и right_half соответственно. Затем функция сравнивает элементы на каждой позиции в left_half и right_half, добавляет меньший элемент в result, увеличивает соответствующий счетчик и продолжает сравнение. Когда один из массивов заканчивается, функция добавляет оставшиеся элементы из другого массива в result. В конце функция возвращает отсортированный массив result.

Реализация алгоритма сортировки слиянием с параметром reverse (False — в порядке возрастания, True — в порядке убывания) представлена на рис. 9.34.

```
# реализация алгоритма сортировки слиянием с параметром reverse
# (False – в порядке возрастания, True – в порядке убывания)
def merge_sort(arr, reverse=False):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half, reverse=reverse)
    right_half = merge_sort(right_half, reverse=reverse)

    return merge(left_half, right_half, reverse=reverse)

# вспомогательная функция для алгоритма сортировки слиянием
def merge(left_half, right_half, reverse=False):
    result = []
    i = 0
    j = 0

    while i < len(left_half) and j < len(right_half):
        if not reverse:
            if left_half[i] <= right_half[j]:
                result.append(left_half[i])
                i += 1
            else:
                result.append(right_half[j])
                j += 1
        elif reverse:
            if left_half[i] >= right_half[j]:
                result.append(left_half[i])
                i += 1
            else:
                result.append(right_half[j])
                j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
```

Рисунок 9.33 — Реализация алгоритма сортировки слиянием с параметром reverse

Сравнение различных сортировок. Обменные сортировки (простая обменная сортировка, шейкерная сортировка, сортировка расческой) — это простые алгоритмы сортировки, которые основаны на постоянном обмене элементов массива до тех пор, пока они не будут отсортированы. Они легко реализуются, но медленно работают на больших массивах.

Сортировка выбором (извлечением) — это алгоритм сортировки, который находит минимальный элемент в массиве и перемещает его в начало, затем находит следующий минимальный элемент и перемещает его на следующую позицию, и так далее. Этот алгоритм лучше обменных сортировок на больших массивах, но все еще медленный.

Сортировка включением (вставками) — это алгоритм сортировки, который проходит по массиву и вставляет каждый элемент в отсортированную часть массива. Этот алгоритм быстрее выбора и обменных сортировок на небольших массивах, но медленнее на больших.

Быстрая сортировка — это один из самых быстрых алгоритмов сортировки. Он разделяет массив на две части, меньшую и большую, и рекурсивно сортирует каждую из них. Он быстрее всех других алгоритмов на больших массивах.

Сортировка Шелла — это модификация сортировки вставками, которая сортирует подмассивы с определенным шагом, затем уменьшает шаг и повторяет процесс до тех пор, пока шаг не станет равен 1. Этот алгоритм быстрее сортировки вставками на больших массивах.

Сортировка слиянием — это алгоритм сортировки, который разделяет массив на две части, сортирует каждую из них и затем объединяет их в один отсортированный массив. Этот алгоритм также быстрый на больших массивах.

Таким образом, каждый из этих алгоритмов имеет свои преимущества и недостатки, и выбор зависит от размера массива и требуемой скорости сортировки. Быстрая сортировка и сортировка слиянием являются наиболее эффективными для больших массивов, в то время как сортировка вставками может быть лучшим выбором для небольших массивов.

Для больших массивов данных (например, более 10000 элементов) наиболее быстрыми алгоритмами являются быстрая сортировка и сортировка слиянием. Они имеют сложность $O(n\log n)$, что означает, что время выполнения этих алгоритмов растет логарифмически с увеличением размера массива. Сортировка Шелла и сортировка выбором медленнее, имеют сложность $O(n^2)$.

Для малых массивов данных (например, менее 1000 элементов) наиболее быстрым алгоритмом является сортировка вставками, у которой сложность $O(n^2)$. Быстрая сортировка и сортировка слиянием могут быть медленнее на малых массивах из-за большого количества рекурсивных вызовов, но все еще имеют сложность $O(n\log n)$. Сортировка выбором

и обменные сортировки медленнее всего работают на малых массивах и имеют сложность $O(n^2)$.

O обозначает асимптотическую сложность алгоритма, а n — размер массива, который нужно отсортировать. Например, сортировка выбором имеет сложность $O(n^2)$, что означает, что время ее выполнения будет пропорционально квадрату размера массива в худшем случае. Быстрая сортировка имеет сложность $O(n\log n)$, что означает, что время ее выполнения будет пропорционально произведению размера массива на логарифм от размера массива в худшем случае.

Допустим, у нас есть два массива: один из 10 элементов, а другой из 1000 элементов. Если мы используем сортировку выбором для обоих массивов, то время выполнения для первого будет примерно 100 операций (10^2), а для второго — один миллион операций (10002). Это объясняется тем, что сложность алгоритма сортировки выбором равна $O(n^2)$, то есть время выполнения растет квадратично от размера массива. В то же время, если мы используем быструю сортировку для обоих массивов, то время выполнения для первого будет примерно 30 операций (10log10), а для второго — примерно 30000 операций (1000log1000). Это объясняется тем, что сложность алгоритма быстрой сортировки равна $O(n\log n)$, то есть время выполнения растет логарифмически от размера массива.

Время выполнения сортировки можно измерить с помощью декоратора. Для этого нужно создать декоратор, который будет измерять время выполнения функции и выводить его на экран (рис. 9.34).

```python
import time

# декоратор, измеряющий время выполнения функции и выводящий его на экран
def measure_time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Время выполнения функции {func.__name__}: {end - start:.6f} сек.\n")
        return result
    return wrapper
```

Рисунок 9.34 — Декоратор, измеряющий время выполнения функции

Декоратор принимает функцию func в качестве аргумента и определяет внутреннюю функцию wrapper, которая принимает любое количество позиционных и именованных аргументов (*args и **kwargs). В начале функции wrapper сохраняется текущее время в переменную start. Затем вызывается функция func с переданными аргументами, и ее результат сохраняется в переменную result. После выполнения функции func сохраняется текущее время в переменную end. Затем декоратор выводит на экран строку, содержащую имя функции func, время выполнения (end – start) и единицы измерения (секунды). Время вы-

водится с точностью до шести знаков после запятой. Наконец, декоратор возвращает результат выполнения функции func.

Чтобы использовать этот декоратор, нужно применить его к целевой функции, например, к функции, реализующий некоторый алгоритм сортировки (рис. 9.35).

```
@measure_time
def shell_sort(arr, reverse=False):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and ((not reverse and arr[j - gap] > temp)
                                or (reverse and arr[j - gap] < temp)):
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

```
import random

arr = random.sample(range(50000), 50000)

sorted_arr = shell_sort(arr)
```

```
Время выполнения функции shell_sort: 1.500001 сек.
```

Рисунок 9.35 — Применение декоратора и результат его работы

Рассмотрим другой вариант измерения времени выполнения алгоритма. Для этого воспользуемся магической функцией %timeit.

%timeit — это магическая функция в Jupyter Notebook, которая позволяет измерить время выполнения кода. Она автоматически запускает указанный код несколько раз и вычисляет среднее время выполнения, чтобы получить более точные результаты. %timeit также может автоматически выбирать наилучший способ выполнения кода и выводить дополнительную информацию, такую как количество повторений и среднее время выполнения (рис. 9.36).

```
def bubble_sort(arr, reverse=False):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if not reverse:
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
            else:
                if arr[j] < arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

```
arr = random.sample(range(3000), 3000)

%timeit sorted_arr = bubble_sort(arr)
```

```
1.34 s ± 33.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Рисунок 9.36 — Применение функции %timeit для измерения времени выполнения кода

Результат выполнения кода указывает на среднее время выполнения пузырьковой сортировки для списка arr в 7 запусках программы. В данном случае, среднее время выполнения составляет 1.34 секунды с отклонением в 33.8 миллисекунды.

Это время выполнения может быть репрезентативным для данного списка и конкретного компьютера, но может изменяться при разных условиях выполнения программы. Поэтому, чтобы получить более точные и надежные результаты, рекомендуется провести более детальное исследование времени выполнения алгоритма на разных входных данных и разных компьютерах.

9.4. Пример решения задачи

Пример. Необходимо вывести на экран топ-5 товаров, которые принесли наибольшую выручку за определенный период времени. При решении использовать алгоритмы сортировки: сортировку пузырьком, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Информация о товарах хранится в файле.

Решение. Предположим, что существует CSV-файл (products.csv), который хранит данные о товарах, а именно, наименование товара, его категория, оптовая цена, розничная цена и продажи за год. Определим класс Product, представляющий товар, который имеет следующие характеристики (рис. 9.37):

- name (str) — наименование товара;
- category (str) — категория товара;
- wholesale_price (float) — оптовая цена за единицу товара;
- retail_price (float) — розничная цена за единицу товара;
- sales (list) — список продаж товара за год.

У класса есть несколько методов:

- __init__(self, name, category, wholesale_price, retail_price, sales) — конструктор класса, который принимает на вход данные о товаре и инициализирует атрибуты класса;
- str(self) — метод, который возвращает строку с информацией о товаре в формате, пригодном для чтения;
- total_sales(self) — метод, который суммирует все продажи товара за год;
- total_revenue(self, period=0) — метод, который вычисляет общую выручку от продаж товара за год или за указанный период;
- average_price(self) — метод, который вычисляет среднюю цену за единицу товара;

- add_sales(self, sales) — метод, который добавляет новые продажи в список продаж, удаляя при этом самые старые данные о продажах.

```
# создание класса Product
class Product:
    def __init__(self, name, category, wholesale_price, retail_price, sales):
        self.name = name
        self.category = category
        self.wholesale_price = wholesale_price
        self.retail_price = retail_price
        self.sales = sales

    def __str__(self):
        return (f'Наименование товара: {self.name}\n'
                f'Категория товара: {self.category}\n'
                f'Оптовая цена: {self.wholesale_price}\n'
                f'Розничная цена: {self.retail_price}\n'
                f'Продажи за год: {self.sales}\n')

    def total_sales(self):
        return sum(self.sales)

    def total_revenue(self, period=0):
        if not period:
            return sum(self.sales) * self.retail_price
        else:
            return sum(self.sales[len(self.sales)-period:]) * self.retail_price

    def average_price(self):
        return sum(self.sales) / len(self.sales)

    def add_sales(self, sales):
        del self.sales[-12]
        self.sales.append(sales)
```

Рисунок 9.37 — Класс Product

Теперь откроем файл products.csv в режиме чтения и используем модуль csv для чтения данных из этого файла (рис. 9.38). Создаем пустой список list_of_products. Далее код проходится по каждой строке в файле с помощью цикла for, пропуская первую строку с помощью функции next(reader). Каждая строка файла представляет собой данные о продукте, разделенные запятыми. Код извлекает необходимые данные, создает объект класса Product и добавляет его в список list_of_products.

```
import csv

list_of_products = []

with open('products.csv', 'r') as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        list_of_products.append(Product(row[0], row[1],
                                        float(row[2]), float(row[3]),
                                        [int(sale) for sale in row[4:]]))
```

Рисунок 9.38 — Чтение файла products.csv

Создадим декоратор функции, который позволит измерить время выполнения функции, переданной в качестве аргумента (рис. 9.39).

```
# декоратор, вычсляющий время выполнения функции и выводящий его на экран
def measure_time(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"\nВремя выполнения сортировки: {end - start:.6f} сек.")
        return result
    return wrapper
```

Рисунок 9.39 — Декоратор, измеряющий время выполнения функции

Далее определены три функции (bubble_sort, selection_sort, quick_sort) для трех различных алгоритмов сортировки: пузырьком, выбором и быстрой сортировки (рис. 9.40–9.42). В каждой функции сортировки передаются параметры: список объектов products, period — период, за который нужно вычислить выручку от продаж товара и затем отсортировать объекты по этой выручке, reverse — флаг, указывающий направление сортировки (по возрастанию или убыванию).

```
# реализация алгоритма сортировки пузырьком
def bubble_sort(products, period=0, reverse=False):
    n = len(products)
    for i in range(n):
        for j in range(n - i - 1):
            if not reverse:
                if products[j].total_revenue(period) > products[j + 1].total_revenue(period):
                    products[j], products[j + 1] = products[j + 1], products[j]
            else:
                if products[j].total_revenue(period) < products[j + 1].total_revenue(period):
                    products[j], products[j + 1] = products[j + 1], products[j]
    return products
```

Рисунок 9.40 — Реализация алгоритма сортировки пузырьком

```
# реализация алгоритма сортировки выбором
def selection_sort(products, period=0, reverse=False):
    n = len(products)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if reverse:
                if products[j].total_revenue(period) > products[min_idx].total_revenue(period):
                    min_idx = j
            else:
                if products[j].total_revenue(period) < products[min_idx].total_revenue(period):
                    min_idx = j
        products[i], products[min_idx] = products[min_idx], products[i]
    return products
```

Рисунок 9.41 — Реализация алгоритма сортировки выбором

```python
# реализация алгоритма быстрой сортировки
def quick_sort(products, period=0, reverse=False):
    if len(products) <= 1:
        return products
    else:
        pivot = products[0]
        left = []
        right = []
        for i in range(1, len(products)):
            if products[i].total_revenue(period) < pivot.total_revenue(period):
                left.append(products[i])
            else:
                right.append(products[i])
        if reverse:
            return quick_sort(right, period, reverse=True) + [pivot] + quick_sort(left, period, reverse=True)
        else:
            return quick_sort(left, period) + [pivot] + quick_sort(right, period)
```

Рисунок 9.41 — Реализация алгоритма быстрой сортировки (другой вариант реализации)

Функция sort(products, method='1', period=0, reverse=False) принимает список объектов products, метод сортировки method (1 — пузырьком, 2 — выбором, 3 — быстрая сортировка), период времени period и флаг reverse (рис. 9.42). В зависимости от выбранного метода вызывается соответствующая функция сортировки. Функция также измеряет время выполнения сортировки с помощью декоратора measure_time и возвращает отсортированный список.

```python
# выбор метода сортировки в зависимости от переданного параметра
@measure_time
def sort(products, method='1', period=0, reverse=False):
    if method == '1':
        return bubble_sort(products, period, reverse)
    elif method == '2':
        return selection_sort(products, period, reverse)
    elif method == '3':
        return quick_sort(products, period, reverse)
```

Рисунок 9.42 — Функция выбора алгоритма сортировки

Следующий код импортирует модуль copy и копирует список list_of_products с помощью функции deepcopy. Затем запрашивает у пользователя количество месяцев для расчета выручки и метод сортировки. Далее вызывается функция sort с аргументами скопированный список, выбранный метод сортировки и количество месяцев, за которые нужно посчитать выручку. Результатом является список top_5_products, содержащий топ-5 товаров по выручке за указанный период. На экран выводятся названия и категории товаров, а также их выручка за указанный период (рис. 9.43).

```python
import copy

list_of_products_copy = copy.deepcopy(list_of_products)

months = int(input('За какой период надо посмотреть выручку?\nВведите количество месяцев (0 – год): '))
print('\nМетоды сортировки:\n1. Сортировка пузырьком\n2. Сортировка выбором\n3. Быстрая сортировка')
method = input('Выберете метод сортировки: ')

top_5_products = sort(list_of_products_copy, method, months, reverse=True)[:5]

print(f"\nТоп-5 товаров по выручке за последние {months} мес.:")
for i, product in enumerate(top_5_products[:5]):
    print(f"{i+1:<1}. {product.name:<25} {product.category:<15} {product.total_revenue(months):<10} руб.")
```

```
За какой период надо посмотреть выручку?
Введите количество месяцев (0 – год): 6

Методы сортировки:
1. Сортировка пузырьком
2. Сортировка выбором
3. Быстрая сортировка
Выберете метод сортировки: 1

Время выполнения сортировки: 18.814803 сек.

Топ-5 товаров по выручке за последние 6 мес.:
1. Смартфон Арт.306619       Электроника     17703100.0 руб.
2. Планшет Арт.936228        Электроника     16946400.0 руб.
3. Наушники Арт.550660       Электроника     16336800.0 руб.
4. Смартфон Арт.752619       Электроника     15972500.0 руб.
5. Смартфон Арт.060333       Электроника     14940200.0 руб.
```

Рисунок 9.43 — Результат решения задачи

9.5. Задачи для самостоятельного решения

Задача 9.1. При решении использовать указанные алгоритмы сортировки.

1) Необходимо отсортировать список студентов по их среднему баллу за сессию и вывести результат на экран. Использовать алгоритмы сортировки: сортировку выбором, сортировку слиянием и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные о студентах хранятся в файле.

2) Необходимо отсортировать массив объектов по заданному критерию и вывести результат на экран. В зависимости от переданного параметра отсортировать массив объектов класса «Книга» по автору, названию или году издания, используя алгоритмы сортировки: пузырьковую, сортировку вставками и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные о книгах хранятся в файле.

3) Необходимо отсортировать массив строк по алфавиту и вывести результат на экран. Использовать алгоритмы сортировки: сортировку пузырьком, сортировку слиянием и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Строки хранятся в файле.

4) Необходимо отсортировать список товаров по цене и вывести результат на экран. В зависимости от переданного параметра отсортировать список товаров по возрастанию цены или по убыванию цены, используя алгоритмы сортировки: сортировку пузырьком, сортировку слиянием и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные о товарах хранятся в файле.

5) Необходимо отсортировать массив строк по длине и вывести результат на экран. В зависимости от переданного параметра отсортировать массив строк по возрастанию длины или по убыванию длины, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Строки хранятся в файле.

6) Необходимо отсортировать массив чисел по частоте их встречаемости и вывести результат на экран. В зависимости от переданного параметра отсортировать массив чисел по возрастанию частоты их встречаемости или по убыванию частоты их встречаемости, используя алгоритмы сортировки: сортировку пузырьком, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Числа хранятся в файле.

7) Необходимо отсортировать список городов по населению и вывести результат на экран. В зависимости от переданного параметра отсортировать список городов по возрастанию населения или по убыванию населения, используя алгоритмы сортировки: сортировку пузырьком, сортировку вставками и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные о городах хранятся в файле.

8) Необходимо отсортировать массив объектов по определенному полю и вывести результат на экран. В зависимости от переданного параметра отсортировать массив объектов по возрастанию или по убыванию значения определенного поля, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные об объектах хранятся в файле.

9) Необходимо отсортировать список слов по алфавиту и вывести результат на экран. В зависимости от переданного параметра отсортировать список слов по возрастанию или по убыванию алфавитного порядка, используя алгоритмы сортировки: сортировку вставками, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Текст хранится в файле.

10) Необходимо отсортировать массив дат и вывести результат на экран. В зависимости от переданного параметра отсортировать массив дат по возрастанию или по убыванию даты, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и бы-

струю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Даты хранятся в файле.

11) Необходимо отсортировать список товаров по количеству продаж и вывести результат на экран. В зависимости от переданного параметра отсортировать список товаров по возрастанию или по убыванию количества продаж, используя алгоритмы сортировки: сортировку вставками, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Данные о товарах хранятся в файле.

12) Необходимо отсортировать массив чисел по сумме цифр каждого числа и вывести результат на экран. В зависимости от переданного параметра отсортировать массив чисел по возрастанию или по убыванию суммы цифр каждого числа, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Числа хранятся в файле.

13) Необходимо отсортировать массив точек на плоскости по расстоянию до начала координат и вывести результат на экран. В зависимости от переданного параметра отсортировать массив точек по возрастанию или по убыванию расстояния до начала координат, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Координаты точек хранятся в файле.

14) Необходимо отсортировать список фильмов по году выпуска и вывести результат на экран. В зависимости от переданного параметра отсортировать список фильмов по возрастанию или по убыванию года выпуска, используя алгоритмы сортировки: сортировку вставками, сортировку выбором и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Информация о фильмах хранится в файле.

15) Необходимо отсортировать массив звездных величин по яркости и вывести результат на экран. В зависимости от переданного параметра отсортировать массив звездных величин по возрастанию или по убыванию яркости, используя алгоритмы сортировки: сортировку выбором, сортировку пузырьком и быструю сортировку. Сравнить время выполнения алгоритмов сортировки с помощью декоратора. Звездные величины хранятся в файле.

10. Структуры данных: деревья

10.1. Знакомство с бинарными деревьями

Деревья представляют собой особый вид графов, обладающих уникальными свойствами связности и отсутствием циклов. В дереве существует только один путь между любыми двумя узлами, и ни одна пара узлов не может быть соединена более чем одним прямым путем.

Основными элементами в графах и деревьях являются вершины или узлы, а также ребра, которые могут быть названы дугами или связями. Путь в графе представляет собой последовательность узлов, связанных друг с другом.

Узлы могут быть связаны различными типами отношений. Так, узел, получающий входящую связь от другого узла, принято называть дочерним. Узел, отправляющий исходящую связь к другому узлу, обозначается как родительский. Если узлы являются наследниками одного родительского узла, их называют сиблингами или «братьями и сестрами».

Корень дерева — это специальный узел, который не имеет родителя. Он является основой всего дерева. Лист дерева — это узел, который не имеет потомков, он также может быть назван терминальным узлом. Внутренний узел — это любой узел дерева, который имеет потомков и не является листовым узлом.

Уровень узла определяется как количество шагов от корня до этого узла. Корень находится на уровне 0 по определению. Высота дерева — это максимальный уровень среди всех его узлов. Высота дерева позволяет определить его глубину и структуру.

Поддерево представляет собой часть дерева, которая включает в себя родительский узел и все его потомство. Это может быть полезно при анализе и визуализации структуры дерева. Поддеревья помогают разбить большое дерево на более мелкие части для удобства работы с ними.

Важным аспектом работы с деревьями является выбор подходящего представления, которое соответствует требованиям конкретной задачи (рис. 10.1).

Иерархическое представление деревьев основано на идее иерархии или подчиненности элементов. Оно позволяет наглядно отображать структуру дерева, что упрощает его понимание и визуализацию. Каждый элемент дерева имеет одного родителя (за исключением корневого элемента) и может иметь несколько дочерних элементов. Дерево представляется в виде древовидной структуры, где каждый элемент является узлом, а связи между элементами — ребрами.

Представление деревьев с помощью множеств предполагает хранение всех элементов дерева в одном множестве. Отношения между элементами выражаются через связи между членами этого множества. Например, можно применять множество пар (элемент, родитель), где каждая пара отражает связь «элемент является потомком родителя». Такой подход обеспечивает эффективное хранение и обработку структуры дерева.

Линейное представление деревьев подразумевает преобразование иерархической структуры дерева в линейный порядок элементов. Это достигается с помощью алгоритмов обхода дерева, которые преобразуют его в линейный список элементов. Позиция элементов в списке отображает их иерархическую структуру. Такое представление дерева удобно для хранения и обработки данных, так как позволяет применять стандартные алгоритмы работы со списками.

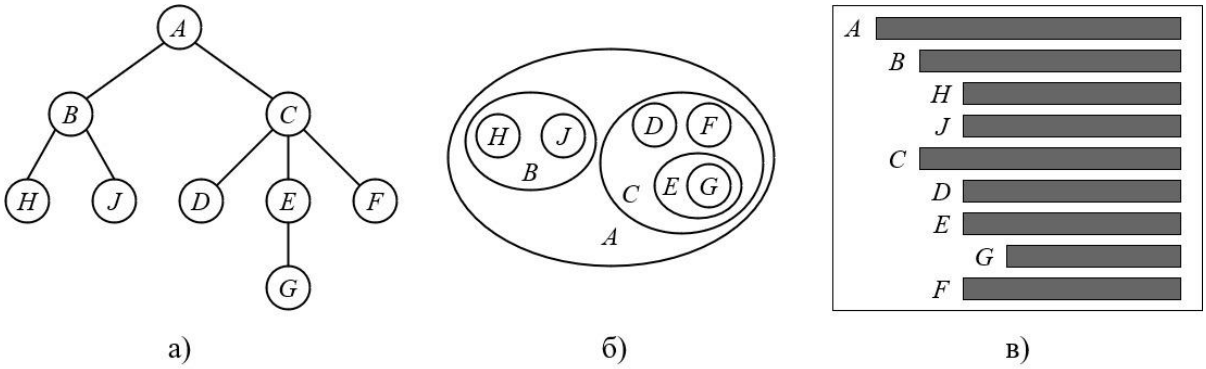


Рисунок 10.1 — Представление деревьев: а) — иерархическая структура, б) — множества, в) — линейное представление

Бинарное (двоичное) дерево — это структура данных, состоящая из узлов, каждый из которых имеет не более двух потомков. Каждый узел в бинарном дереве содержит значение и ссылки на его левого и правого потомка (если они существуют).

Бинарные деревья широко используются в компьютерных науках и информатике для представления и обработки данных, а также для реализации различных алгоритмов, например, поиска, сортировки, вставки и удаления элементов. Бинарные деревья могут быть различных типов в зависимости от специфических требований и ограничений. Некоторые из них включают полные бинарные деревья, сбалансированные бинарные деревья, двоичные деревья поиска и т.д.

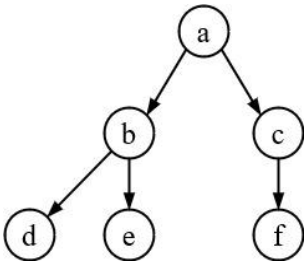


Рисунок 10.2 — Пример небольшого бинарного дерева

Полное бинарное дерево — это тип бинарного дерева, в котором каждый узел имеет либо двух потомков, либо ни одного. Другими словами, каждый уровень дерева полностью заполнен, за исключением, возможно, последнего уровня, который заполняется слева направо.

Основные характеристики полного бинарного дерева:

- Каждый узел имеет либо двух потомков, либо ни одного. Это означает, что каждый узел имеет две ссылки на его левого и правого потомка.
- Последний уровень дерева заполняется слева направо. Если последний уровень не полностью заполнен, то все узлы на этом уровне должны быть расположены слева.
- Количество узлов в полном бинарном дереве можно вычислить по формуле $2^h - 1$, где h — высота дерева. Например, если высота дерева равна 3, то количество узлов будет равно 7.

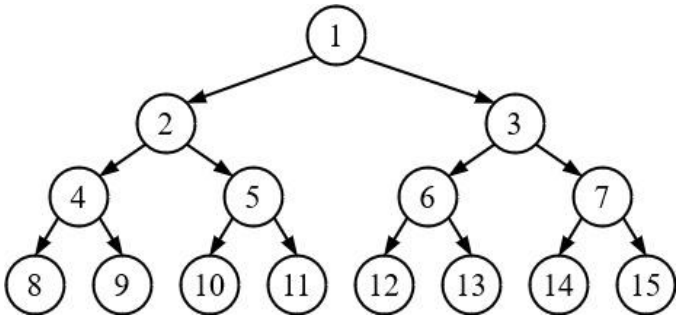


Рисунок 10.3 — Пример полного бинарного дерева

Также полными часто называют бинарные деревья, у которых полностью заполнены все уровни, кроме последнего (рис. 10.4).

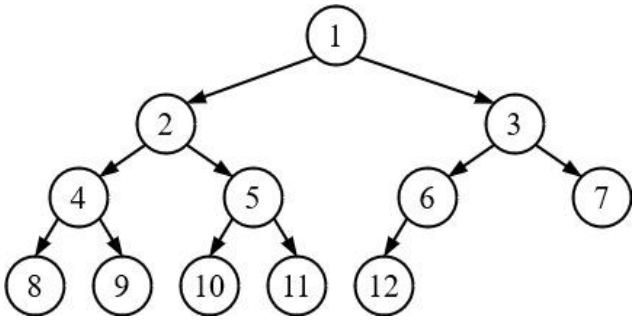


Рисунок 10.4 — Другой пример полного бинарного дерева

Полные бинарные деревья имеют некоторые преимущества в сравнении с другими типами бинарных деревьев. Они обеспечивают эффективное использование памяти, так как не требуют хранения ссылок на несуществующие потомки. Кроме того, полные бинарные деревья обладают определенными свойствами, которые могут быть использованы для оптимизации алгоритмов, работающих с ними. Однако построение и поддержка полного би-

нарного дерева может быть сложным и требовательным по времени процессом, особенно при вставке или удалении узлов.

Вложенные списки позволяют представить структуру бинарного дерева, где каждый узел может иметь свои левое и правое поддеревья. Пустые списки [] используются для обозначения отсутствия поддерева для конкретного узла.

```
my_tree = \
['a', #корень
    ['b', #левое поддерево
        ['d', [], []],
        ['e', [], []] ],
    ['c', #правое поддерево
        ['f', [], []],
        [] ]
]
```

Рисунок 10.5 — Представление бинарного дерева в виде вложенных списков

На рис. 10.5 переменная my_tree представляет собой список, состоящий из трех элементов. Первый элемент списка — это корень дерева, обозначенный символом 'a'. Второй элемент списка — это левое поддерево, также представленное списком. Внутри списка левого поддерева находятся два элемента: 'b' и список, представляющий левое и правое поддеревья для узла 'b'. Аналогично, третий элемент списка my_tree — это правое поддерево, состоящее из узла 'c' и списка, представляющего левое и правое поддеревья для узла 'c'.

API (Application Programming Interface) для работы с бинарными деревьями предоставляет следующие методы:

- BinaryTree() — создание нового экземпляра бинарного дерева.
- get_left_child() — возвращает бинарное дерево, связанное с левым дочерним узлом рассматриваемого узла.
- get_right_child() — возвращает бинарное дерево, связанное с правым дочерним узлом рассматриваемого узла.
- get_root_val() — возвращает объект, хранящийся в данном узле.
- set_root_val(val) — сохраняет объект, хранящийся в данном узле.
- insert_left(val) — создает новое бинарное дерево, связанное с левым дочерним узлом рассматриваемого узла.
- insert_right(val) — создает новое бинарное дерево, связанное с правым дочерним узлом рассматриваемого узла.

Эти методы позволяют работать с бинарными деревьями, добавлять новые узлы и получать доступ к уже существующим.

Код на рис. 10.6 выводит информацию о бинарном дереве, представленном в виде вложенного списка. Он выводит список, представляющий бинарное дерево, а также информацию о левом поддереве, корне и правом поддереве.

```
print(my_tree)
print('left subtree = ', my_tree[1])
print('root = ', my_tree[0])
print('right subtree = ', my_tree[2])
```

```
['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]
left subtree =  ['b', ['d', [], []], ['e', [], []]]
root =  a
right subtree =  ['c', ['f', [], []], []]
```

Рисунок 10.6 — Вывод информации о бинарном дереве

Первая строка кода выводит содержимое всего дерева. Вторая строка выводит информацию о левом поддереве, используя индекс 1 для доступа к соответствующему элементу структуры данных. Третья строка выводит информацию о корне дерева, используя индекс 0. Четвертая строка выводит информацию о правом поддереве, используя индекс 2.

Приведем пример реализации функции insert_left(), которая позволяет добавить новую ветвь слева от указанного узла в представлении бинарного дерева в виде вложенных списков (рис. 10.7).

```
def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root
```

```
insert_left(my_tree[2], 'H')
```

```
['c', ['H', ['f', [], []], []], []]
```

Рисунок 10.7 — Пример реализации функции insert_left()

Сначала из списка root извлекается правая ветвь указанного узла с помощью метода pop(1), который удаляет элемент с индексом 1 из списка root и возвращает его. Полученный элемент сохраняется в переменную t. Затем проверяется длина списка t. Если она больше 1, то это означает, что у указанного узла уже есть правая ветвь. В таком случае, новая ветвь new_branch добавляется перед t в виде списка [new_branch, t, []] с помощью метода insert(1). Если же длина списка t равна 1, то это означает, что у указанного узла нет правой ветви. В таком случае, новая ветвь new_branch добавляется перед пустыми списками [] в виде списка [new_branch, [], []] с помощью метода insert(1). В конце функция возвращает измененный список root.

Бинарное дерево можно представить также в виде узлов и ссылок. Такое представление бинарного дерева обеспечивает эффективное хранение и операции над структурой данных, что делает его широко используемым в различных алгоритмах и приложениях (рис. 10.8).

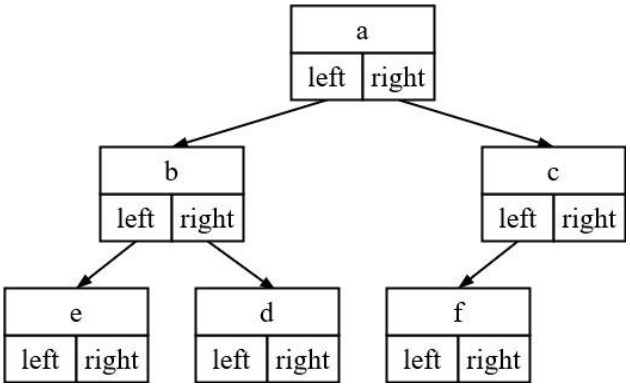


Рисунок 10.8 — Представление бинарного дерева в виде узлов и ссылок

Каждый узел бинарного дерева представляется объектом или структурой данных, которая содержит два поля: поле данных и два поля ссылок. Поле данных содержит значение или ключ, которое хранится в узле, а поля ссылок содержат ссылки на левого и правого потомков этого узла. Преимущество представления бинарного дерева в виде узлов и ссылок заключается в том, что оно позволяет эффективно выполнять операции над деревом, такие как поиск, вставка и удаление элементов.

На рис. 10.9 представлен фрагмент реализации бинарного дерева. Класс BinaryTree представляет собой бинарное дерево, состоящее из узлов. Каждый узел содержит ключ (self.key) и ссылки на левого (self.left_child) и правого (self.right_child) потомков. Метод insert_left(new_node) вставляет новый узел new_node в качестве левого потомка текущего узла. Если левый потомок уже существует, то новый узел становится левым потомком, а предыдущий левый потомок становится левым потомком нового узла. Метод insert_right(new_node) аналогично вставляет новый узел new_node в качестве правого потомка текущего узла. Если правый потомок уже существует, то новый узел становится правым потомком, а предыдущий правый потомок становится правым потомком нового узла. Метод get_right_child() возвращает правого потомка текущего узла. Метод get_left_child() возвращает левого потомка текущего узла. Метод set_root_val(obj) устанавливает значение ключа текущего узла равным obj. Метод get_root_val() возвращает значение ключа текущего узла. Метод __str__() возвращает строковое представление текущего узла и его потомков в формате «ключ (левый_потомок, правый_потомок)».

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None

    def insert_left(self,new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self,new_node):
        if self.right_child == None:
            self.right_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t

    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self,obj):
        self.key = obj

    def get_root_val(self):
        return self.key

    def __str__(self):
        return '{} ({}, {})'.format(self.get_root_val(), str(self.get_left_child()), str(self.get_right_child()))
```

Рисунок 10.9 — Пример реализации бинарного дерева в виде узлов и ссылок

Пример работы с данной реализацией бинарного дерева показан на рис. 10.10.

```
r = BinaryTree('a')
print(r)
```

```
a (None, None)
```

```
r.insert_left('b')
print(r)
print(r.get_left_child())
print(r.get_left_child().get_root_val())
```

```
a (b (None, None), None)
b (None, None)
b
```

```
r.insert_right('c')
print(r)
print(r.get_right_child())
print(r.get_right_child().get_root_val())
```

```
a (b (None, None), c (None, None))
c (None, None)
c
```

```
r.get_right_child().set_root_val('hello')
print(r.get_right_child().get_root_val())
```

```
hello
```

```
print(r)
```

```
a (b (None, None), hello (None, None))
```

```
r.insert_right('f')
print(r)
```

```
a (b (None, None), f (None, hello (None, None)))
```

Рисунок 10.10 — Пример работы с бинарным деревом

Сначала создаем объект r класса BinaryTree, устанавливаем его корневое значение как 'a' и выводим его на экран. Затем вставляем левого потомка со значением 'b' и снова выводим дерево. Далее выводим левого потомка корня ('b') и его значение ('b'). После этого вставляем правого потомка корня со значением 'c' и снова выводим дерево. Выводим правого потомка корня ('c') и его значение ('c'). Изменяем значение правого потомка корня на 'hello' и выводим его значение ('hello'). Последний вывод показывает текущее состояние дерева, которое теперь имеет правого потомка со значением 'hello' и левого потомка со значением 'b'. В конце вставляем нового правого потомка со значением 'f'.

Бинарные деревья могут быть использованы для представления и обработки арифметических, логических и других типов выражений. Каждый узел в дереве представляет операцию или операнд, а дочерние узлы представляют аргументы или подвыражения этой операции.

Процесс разбора выражений с использованием бинарных деревьев обычно включает следующие шаги:

1. Лексический анализ. Исходное выражение разбивается на лексемы (токены), такие как операторы, операнды и скобки.
2. Синтаксический анализ. Лексемы анализируются и проверяются на соответствие грамматике языка. В результате получается дерево разбора, которое представляет собой структуру выражения.
3. Построение бинарного дерева. Дерево разбора преобразуется в бинарное дерево, где каждый узел представляет операцию или операнд, а дочерние узлы представляют аргументы или подвыражения этой операции.
4. Вычисление выражения. Бинарное дерево обходится и выражение вычисляется в соответствии с правилами операций.

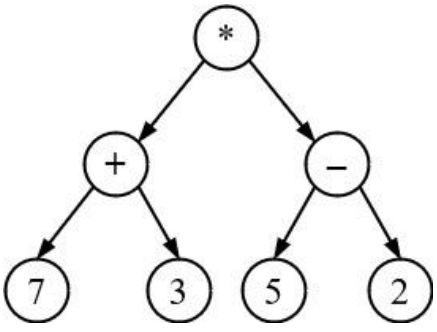


Рисунок 10.11 — Представление математического выражения ((7 + 3) * (5 − 2)) в виде бинарного дерева

Алгоритм разбора математического выражения для получения его древовидного представления:

1. Если текущий токен ' (', то добавляем новый узел в качестве левого дочернего узла текущего узла и спускаемся в новый узел.
2. Если текущий токен содержится в списке ['+', '–', '/', '*'], то устанавливаем значение в текущем узле, соответствующее оператору, представленному в токене. Добавляем новый узел в качестве правого дочернего узла текущего узла и спускаемся в него.
3. Если текущий токен является числом, то устанавливаем значение в текущем узле соответствующее числу в токене и переходим к родительскому узлу.
4. Если текущий токен ') ', то переходим к родителю текущего узла.

Разбор математического выражения (3 + (4 * 5)) или ['(', '3', '+', '(', '4', '*', '5' ,')', ')'] и построение на его основе бинарного дерева показан на рис. 10.12.

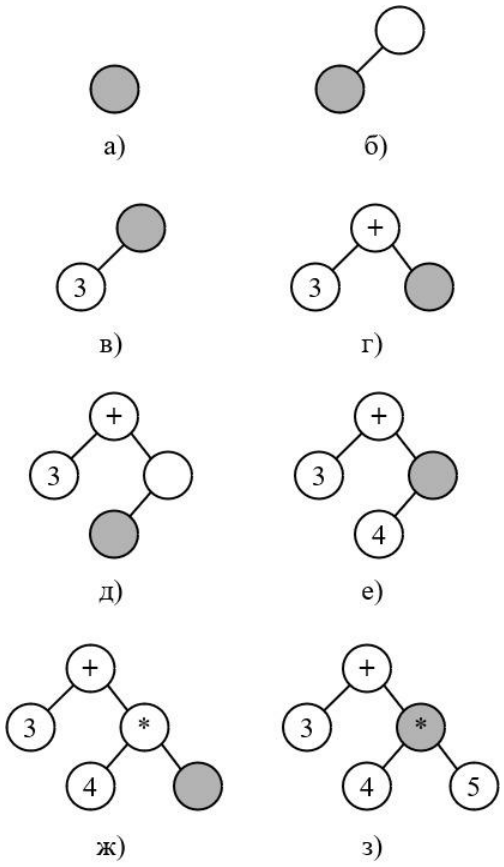


Рисунок 10.12 — Разбор математического выражения (3 + (4 * 5))

В данном случае, можно использовать алгоритм рекурсивного спуска для построения бинарного дерева.

- Начнем с корня дерева, который будет представлять всё выражение.
- Разберем выражение слева направо. Первый символ '(' означает начало подвыражения, поэтому создадим левую ветвь корня и перейдем к ней.
- В левой ветви корня, следующий символ '3' будет представлять число 3. Создадим узел с этим числом и добавим его в левую ветвь.

- Следующий символ '+' означает операцию сложения. Создадим узел с этой операцией и добавим его в правую ветвь корня.
- Перейдем к правой ветви корня.
- В правой ветви корня, следующий символ '(' означает начало подвыражения, поэтому создадим левую ветвь правой ветви корня и перейдем к ней.
- В левой ветви правой ветви корня, следующий символ '4' будет представлять число 4. Создадим узел с этим числом и добавим его в левую ветвь.
- Следующий символ '*' означает операцию умножения. Создадим узел с этой операцией и добавим его в правую ветвь левой ветви правой ветви корня.
- Перейдем к правой ветви левой ветви правой ветви корня.
- В правой ветви левой ветви правой ветви корня, следующий символ '5' будет представлять число 5. Создадим узел с этим числом и добавим его в правую ветвь.
- Закроем подвыражение с помощью символа ')'. Перейдем к родительскому узлу.
- Закроем подвыражение с помощью символа ')'. Перейдем к родительскому узлу.
- Завершим разбор выражения.

Для вычисления значения выражения имеющего древовидное представление достаточно реализовать простую рекурсивную функцию (рис. 10.13).

```
def evaluate_expression(node):
    if isinstance(node, BinaryTree):
        root_value = node.get_root_val()
    else:
        root_value = node
    if isinstance(root_value, str) and root_value.isdigit():
        return int(root_value)

    left_value = evaluate_expression(node.get_left_child())
    right_value = evaluate_expression(node.get_right_child())

    if root_value == '+':
        return left_value + right_value
    elif root_value == '-':
        return left_value - right_value
    elif root_value == '*':
        return left_value * right_value
    elif root_value == '/':
        return left_value / right_value
```

Рисунок 10.13 — Пример реализации рекурсивной функции

Сначала функция evaluate_expression() проверяет, является ли входной узел node экземпляром класса BinaryTree. Если это так, то значение корня узла получается с помощью метода get_root_val() класса BinaryTree. В противном случае значение корня устанавливается равным самому входному узлу. Затем функция проверяет, является ли значение корня строкой и состоит ли оно только из цифр. Если это условие выполняется, функция преобразует значение корня в целое число и возвращает его. Если значение корня не является

строкой из цифр, функция рекурсивно вызывает саму себя для вычисления левого и правого потомков узла. Значения, возвращаемые этими рекурсивными вызовами, сохраняются в переменных left_value и right_value соответственно. Наконец, функция выполняет арифметические операции на основе значения корня.

Еще одним примером практического применения бинарных деревьев является представление документа (книги) в виде дерева. В этой модели каждый узел дерева представляет собой главу или раздел книги, а корневой узел — саму книгу.

Рассмотрим пример (рис. 10.14). Предположим, имеется книга по программированию, состоящая из нескольких глав, разделов и подразделов. Можно представить эту книгу в виде бинарного дерева следующим образом:

- Корневой узел представляет собой всю книгу.
- Первый уровень дочерних узлов представляет собой главы книги.
- Второй уровень дочерних узлов представляет собой разделы внутри каждой главы и т.д.

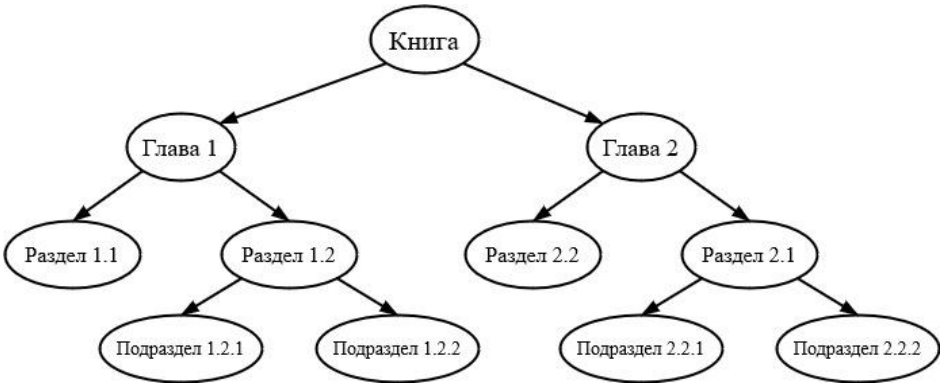


Рисунок 10.14 — Представление документа (книги) как дерева

Такое представление книги в виде бинарного дерева позволяет эффективно организовать и структурировать информацию. Можно легко найти определенную главу или раздел, переходя от корневого узла к нужному узлу посредством навигации по дереву. Кроме того, бинарные деревья позволяют выполнять различные операции над документом. Например, можно добавлять новые главы или разделы в дерево, удалять существующие узлы, изменять порядок глав и разделов, а также выполнять поиск по ключевым словам или темам.

Обход бинарного дерева — это процесс посещения каждого узла в дереве ровно один раз. Во время обхода, можно выполнять определенные операции с данными узла, например, вывод на экран или сохранение в структуру данных.

Общая часть обхода бинарного дерева включает в себя следующие шаги:

1. Проверяем, является ли текущий узел пустым. Если да, то возвращаемся к предыдущему узлу (если такой есть) или завершаем обход.

2. Посещаем текущий узел, выполняя необходимые операции с данными.

3. Рекурсивно обходим левое поддерево, вызывая ту же самую процедуру для левого дочернего узла.

4. Рекурсивно обходим правое поддерево, вызывая ту же самую процедуру для правого дочернего узла.

Порядок выполнения шагов может различаться в зависимости от выбранного порядка обхода (прямой, обратный или симметричный).

Прямой (pre-order) порядок обхода дерева:

- Первым просматривается корневой узел.
- Затем производится рекурсивный прямой обход левого поддерева.
- Затем производится рекурсивный прямой обход правого поддерева.

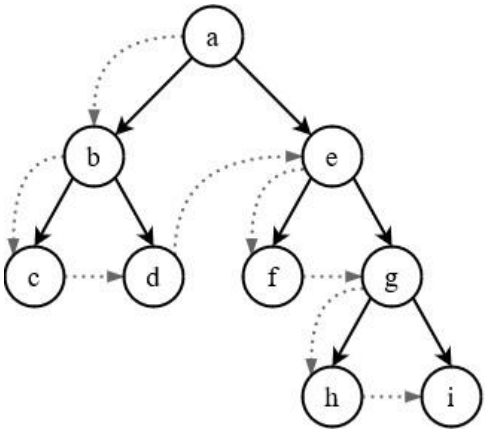


Рисунок 10.15 — Прямой порядок обхода бинарного дерева

При обходе бинарного дерева, изображенного на рисунке 10.15, сначала посещается корень 'a'. Переходим к левому поддереву корня 'a', которое состоит только из узла 'b'. Посещаем корень левого поддерева 'b' и переходим к его левому поддереву, где находим узел 'c'. После посещения корня левого поддерева 'c', не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу.

Возвращаемся к узлу 'b' и переходим к его правому поддереву. Здесь находим узел 'd'. После посещения корня правого поддерева 'd', также не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу.

Возвращаемся к узлу 'a' и переходим к его правому поддереву, где находим узел 'e'. Переходим к левому поддереву корня 'e', где находим узел 'f'. После посещения корня левого поддерева 'f', также не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу. Возвращаемся к узлу 'e' и переходим к его правому поддереву. Здесь

также не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу.

Возвращаемся к узлу 'a' и переходим к его правому поддереву, где находим узел 'g'. Переходим к левому поддереву корня 'g', где находим узел 'h'. После посещения корня левого поддерева 'h', также не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу. Возвращаемся к узлу 'g' и переходим к его правому поддереву, где находим узел 'i'. После посещения корня левого поддерева 'i', также не находим ни левого, ни правого поддерева, поэтому переходим к следующему узлу.

Таким образом, прямой порядок обхода данного бинарного дерева будет следующим: a, b, c, d, e, f, g, h, i.

Обратный (post-order) порядок обхода дерева:

- Первым производится рекурсивный обратный обход левого поддерева.
- Затем производится рекурсивный обратный обход правого поддерева.
- Затем просматривается корневой узел.

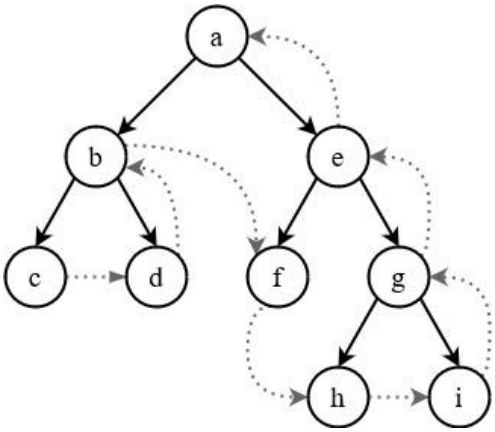


Рисунок 10.16 — Обратный порядок обхода бинарного дерева

Для обратного порядка обхода дерева сначала обрабатывается левое поддерево узла 'b'. Внутри этого поддерева, сначала обрабатывается левое поддерево узла 'c', но так как у него нет поддеревьев, сразу переходим к обработке самого узла 'c'. Затем обрабатывается правое поддерево узла 'c', но так как и у него нет поддеревьев, сразу переходим к его обработке. После этого возвращаемся к узлу 'b' и обрабатываем его.

Затем переходим к правому поддереву узла 'b'. Внутри этого поддерева, сначала обрабатывается левое поддерево узла 'f', но так как у него нет поддеревьев, сразу переходим к его обработке. Затем возвращаемся к узлу 'e' и обрабатываем его.

Далее переходим к правому поддереву узла 'e'. Внутри этого поддерева, сначала обрабатывается левое поддерево узла 'h', но так как у него нет поддеревьев, сразу переходим к его обработке. Затем обрабатывается правое поддерево узла 'h', но так как и у него нет

поддеревьев, сразу переходим к его обработке. После этого возвращаемся к узлу 'g' и обрабатываем его. Наконец, обрабатывается корневой узел 'a'.

Таким образом, обратный порядок обхода данного дерева будет: c, d, b, f, h, i, g, e, a.

Симметричный (in-order) порядок обхода дерева:

- Первым производится рекурсивный симметричный обход левого поддерева.
- Затем просматривается корневой узел.
- Затем производится рекурсивный симметричный обход правого поддерева.

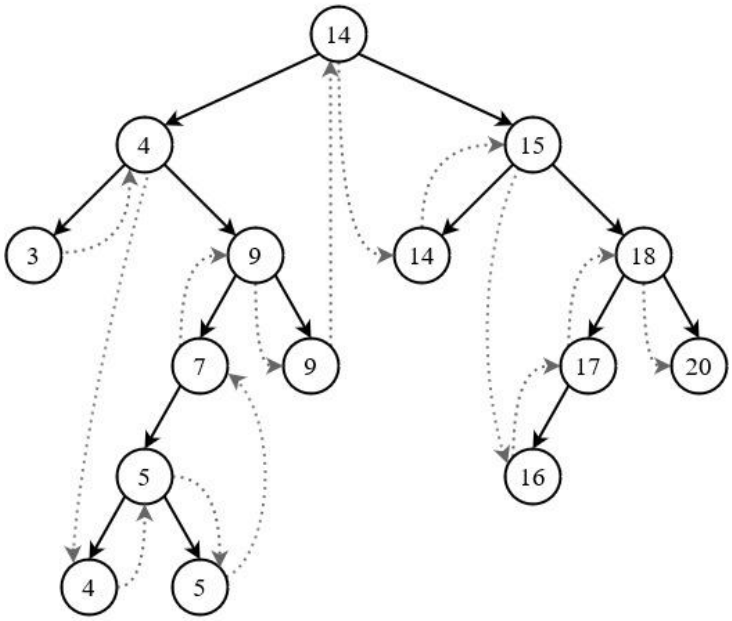


Рисунок 10.17 — Симметричный порядок обхода бинарного дерева

Симметричный порядок обхода бинарного дерева, изображенного на рис. 10.17, начинается со спуска в самую левую ветвь дерева до узла со значением 3. Это становится первым значением в порядке обхода. Затем происходит возвращение к родительскому узлу 4 и его обработка. После этого следует переход к правому поддереву узла 4 и обработка его левого поддерева, представленного узлом со значением 7. Однако перед обработкой этого узла происходит спуск в его левое поддерево, которое является узлом со значением 5. В узле 5 сначала обрабатывается его левое поддерево — узел со значением 4, затем сам узел 5 и его правое поддерево, также представленное узлом со значением 5. После этого можно вернуться к узлу 7 и обработать его. Далее происходит возвращение к узлу 9 и его обработка, а затем переход к правому поддереву узла 9, также представленному узлом со значением 9, и его обработка.

Теперь можно вернуться к корню дерева со значением 14 и обработать его. Затем следует переход к правому поддереву корня и обработка его левого поддерева, представленного узлом со значением 14. Затем следует переход к узлу 15 и его обработка. Далее про-

исходит переход к правому поддереву узла 15 и обработка его левого поддерева, представленного узлом со значением 18. В узле 18 сначала обрабатывается его левое поддерево — узел со значением 17. Затем происходит переход к правому поддереву узла 17, представленному узлом со значением 16, и его обработка. Теперь можно вернуться к узлу 18 и обработать его. Наконец, происходит переход к правому поддереву узла 18, представленному узлом со значением 20, и его обработка.

Таким образом, симметричный порядок обхода данного дерева выглядит следующим образом: 3, 4, 4, 5, 5, 7, 9, 9, 14, 14, 15, 16, 17, 18, 20.

10.2. Бинарное дерево поиска

Бинарное дерево поиска — это двоичное дерево, которое имеет следующие свойства:

- Оба поддерева (левое и правое) являются бинарными деревьями поиска.
- Для каждого узла *X* все значения ключей данных в левом поддереве меньше значения ключа данных узла *X*.
- Для каждого узла *X* все значения ключей данных в правом поддереве больше или равны значению ключа данных узла *X*.

Очевидно, чтобы выполнить эти условия, каждый узел должен иметь ключи данных, на которых можно выполнять операцию сравнения (например, операцию «меньше»).

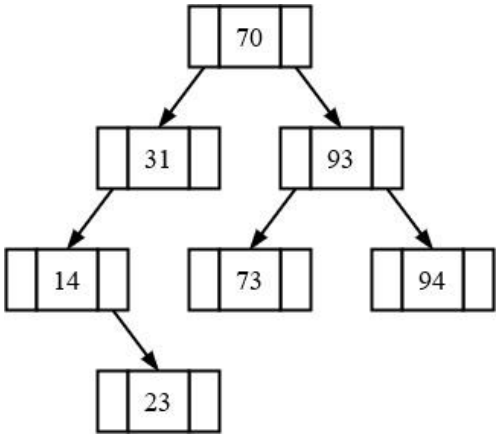


Рисунок 10.17 — Пример бинарного дерева поиска

Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время $O(\log n)$ в наихудшем случае. Математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\log n)$, так что все основные операции динамического множества в таком дереве выполняются в среднем за время $\Theta(\log n)$. Однако на

практике не всегда можно гарантировать случайность построения бинарного дерева поиска. В таких случаях используются сбалансированные по высоте деревья, которые гарантируют хорошее время работы в наихудшем случае. Существуют различные версии сбалансированных деревьев, такие как АВЛ-деревья, 2-3, 3-4 деревья и красно-черные деревья, у которых высота определяется как $O(\log n)$.

Алгоритм вставки нового узла в двоичное дерево поиска следующий:

- Начинаем просмотр с корня дерева (первый текущий узел — корень).
- Сравниваем значение нового узла со значением в текущем узле. Если значение в новом узле меньше, то продолжаем поиск в левом поддереве (текущим узлом становится левый дочерний узел предыдущего текущего узла). Если значение в новом узле больше, чем в текущем узле, то продолжаем поиск в правом поддереве.
- Если левого или правого поддерева не существует, то обнаруживаем место для вставки нового элемента. На это место вставляется новый узел дерева.

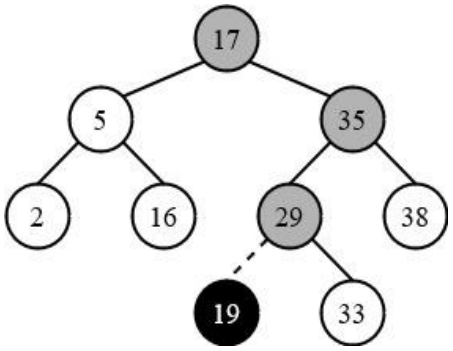


Рисунок 10.18 — Пример вставки узла в бинарное дерево поиска

Вставка узла 19 в бинарное дерево, изображенное на рис. 10.18, происходит по следующему алгоритму:

- Сначала сравниваем значение нового узла 19 с корнем дерева 17. Так как 19 больше 17, идем в правое поддерево.
- В правом поддереве сравниваем значение нового узла 19 с узлом 35. Так как 19 меньше 35, идем в левое поддерево.
- В левом поддереве сравниваем значение нового узла 19 с узлом 29. Так как 19 меньше 29, идем в левое поддерево, которое пусто.
- Вставляем новый узел 19 на место пустого поддерева.

```
def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self, key, current_node):
    if not current_node:
        return None
    elif current_node.key == key:
        return current_node
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)
```

```
def __getitem__(self, key):
    return self.get(key)

def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False
```

Рисунок 10.19 — Пример реализации поиска элемента в двоичном дереве поиска

Код на рис. 10.19 представляет собой реализацию методов для поиска в бинарном дереве поиска.

- Метод get(self, key) используется для получения узла с заданным ключом. Если корень дерева существует, метод вызывает вспомогательный метод _get, передавая ему ключ и корневой узел. Если узел с заданным ключом найден, метод возвращает его полезную нагрузку (payload), иначе возвращает None. Если корня нет, метод также возвращает None. Payload в контексте бинарного дерева поиска означает данные или информацию, которую хранит узел. Каждый узел в бинарном дереве поиска имеет ключ, который используется для определения позиции узла в дереве, и полезную нагрузку, которая представляет собой фактические данные, связанные с этим ключом.
- Вспомогательный метод _get(self, key, current_node) рекурсивно просматривает дерево для поиска узла с заданным ключом. Если текущий узел не существует, метод возвращает None. Если ключ текущего узла равен заданному ключу, метод возвращает текущий узел. Если заданный ключ меньше ключа текущего узла, метод рекурсивно вызывается для левого потомка текущего узла. Если заданный ключ больше ключа текущего узла, метод рекурсивно вызывается для правого потомка текущего узла.
- Метод __getitem__(self, key) позволяет использовать объекты этого класса с индексацией, как если бы они были словарями или списками. Он просто вызывает метод get.

- Метод __contains__(self, key) позволяет использовать оператор in с объектами этого класса. Он вызывает вспомогательный метод _get и возвращает True, если узел с заданным ключом найден, и False в противном случае.

Наиболее сложной операцией является удаление узла из дерева:

Шаг 1. При помощи поиска по дереву найти узел, который нужно удалить.

Шаг 2. После обнаружения узла существует три случая, которые требуют специфической реализации операции удаления узла.

Случай 1. Удаляемый узел не имеет потомков.

В этом случае удаление узла является наиболее простым. Для удаления узла, который не имеет потомков, достаточно просто удалить ссылку на него из его родительского узла.

Шаги для удаления узла без потомков:

1. Найти узел, который нужно удалить, при помощи поиска по дереву.

2. Проверить, является ли узел корневым узлом. Если да, то просто установить ссылку на корень равной None.

3. Если узел не является корневым узлом, найти его родительский узел.

4. Удалить ссылку на удаляемый узел из его родительского узла.

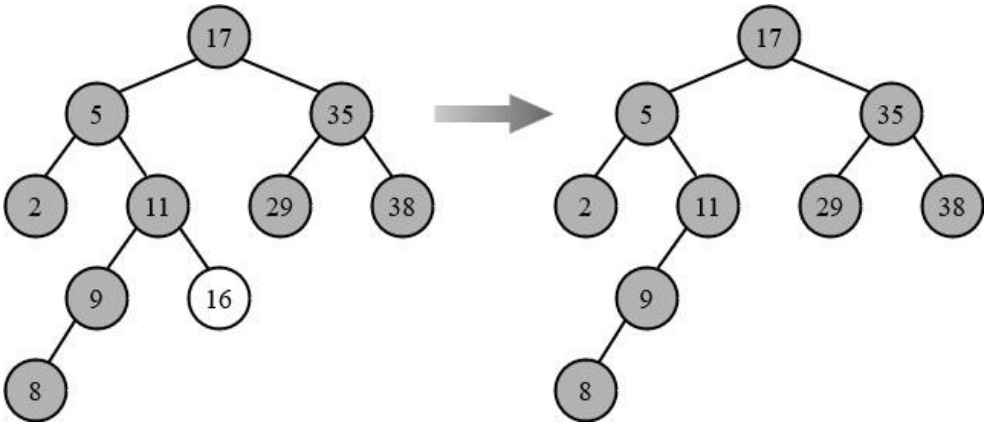


Рисунок 10.19 — Удаление из бинарного дерева поиска узла,
не имеющего потомков

Для удаления узла 16 из бинарного дерева поиска, показанного на рис. 10.19, начинаем с корневого узла 17 и последовательно сравниваем значение узла 16 с каждым узлом. Переходим к левому или правому поддереву в зависимости от результата сравнения. Когда достигаем узла 11, переходим к правому поддереву (16). Достигнув нужного узла, теперь его можно удалить из дерева, удалив также ссылку на него из его родительского узла 11. Таким образом, узел 16 будет успешно удален из бинарного дерева поиска.

Случай 2. Удаляемый узел имеет только одного потомка.

Если текущий узел является левым потомком, то необходимо изменить связь между родительским узлом текущего узла и его левым потомком таким образом, чтобы она указывала на потомка текущего узла. Тогда левый потомок станет прямым потомком родительского узла текущего узла.

Если текущий узел является правым потомком, то необходимо изменить связь между родительским узлом текущего узла и его правым потомком таким образом, чтобы она указывала на потомка текущего узла. Тогда правый потомок станет прямым потомком родительского узла текущего узла.

Если у текущего узла нет родительского узла (т.е. он является корневым узлом), то его единственный потомок станет новым корневым узлом. Это означает, что ссылка на родительский узел у потомка будет удалена, и он сам станет вершиной дерева.

Таким образом, в случае, когда удаляемый узел имеет только одного потомка, необходимо обновить связи между текущим узлом, его потомком и родительским узлом, чтобы правильно сохранить структуру дерева.

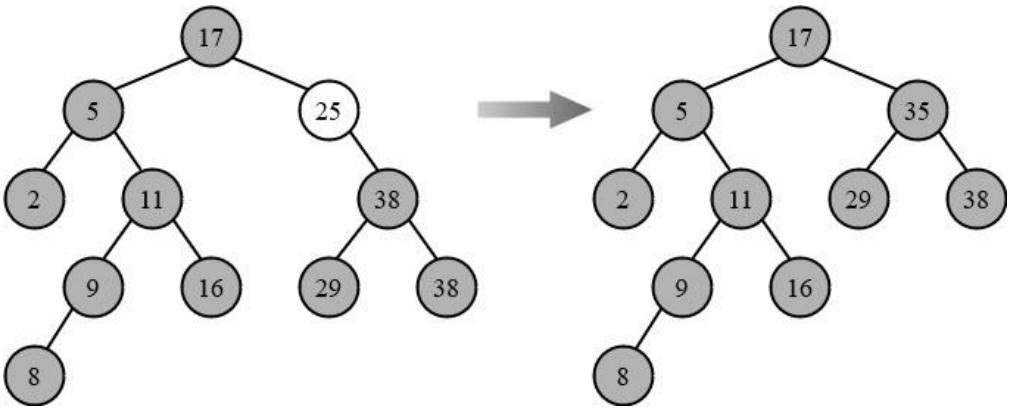


Рисунок 10.20 — Удаление из бинарного дерева поиска узла, имеющего одного потомка

Для удаления узла 25 из бинарного дерева поиска, изображенного на рис. 10.20, сначала необходимо найти его. Узел 25 является правым потомком узла 17. После того как найден узел 25, можно видеть, что у него есть только один потомок — узел 35. В случае, когда у удаляемого узла только один потомок, этот потомок просто занимает место удаляемого узла. Таким образом, узел 35 заменяет узел 25.

Случай 3. Удаляемый узел имеет двух потомков.

Если у удаляемого узла *Z* два потомка, то находим следующий за ним по величине узел *Y*, у которого не более одного потомка и убираем его из позиции, где он находился ранее, путем создания новой связи между его родительским узлом и потомком, и заменяем им узел *Z*.

Поиск *Y* осуществляется следующим образом:

- Переходим из *Z* в его правое поддерево.
- Двигаемся вниз по левому поддерву, пока не встретим узел *Y*, у которого нет левого потомка (у *Y* может быть только один правый потомок, либо может не быть потомков вовсе). *Y* — искомый узел.

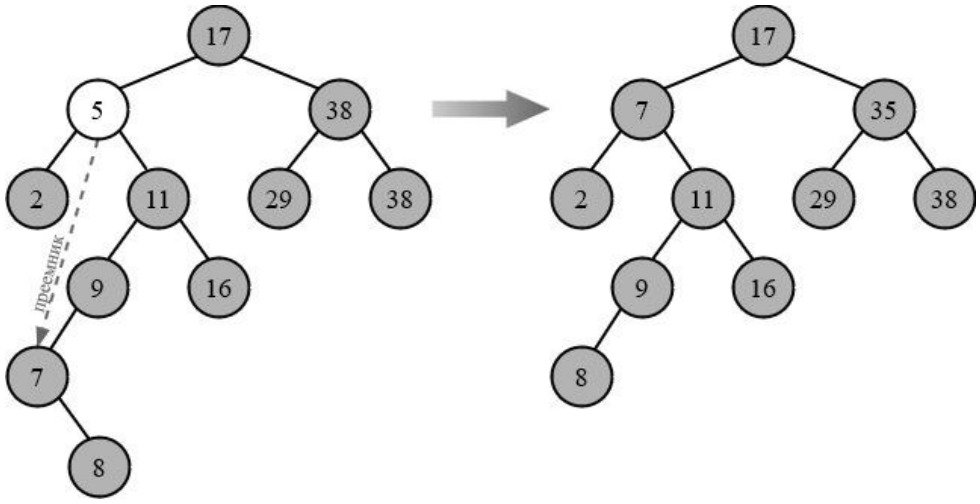


Рисунок 10.21 — Удаление из бинарного дерева поиска узла, имеющего двух потомков

На рис. 10.21 сначала находим узел 5 в дереве. Он является левым потомком корневого узла 17. У узла 5 есть два потомка: 2 и 11. Затем ищем узел, который займет место удаленного. Это будет либо наибольший узел из левого поддерева, либо наименьший узел из правого поддерева. В данном случае выбираем наибольший узел из левого поддерева — узел 7. Заменяем узел 5 на узел 7 в родительском узле 17. Затем обновляем потомков узла 7: его левым потомком становится узел 2 (предыдущий левый потомок узла 5), а правым потомком — узел 11 (предыдущий правый потомок узла 5). После этих операций узел 9 остается без левого потомка, поэтому делаем его левым потомком узел 8 (предыдущий правый потомок узла 7).

Алгоритм вставки в бинарное дерево, который только что рассмотрели, дает хорошие результаты при использовании случайных входных данных, но все же существует неприятная возможность того, что при этом будет построено вырожденное дерево. Можно было бы разработать алгоритм, поддерживающий дерево в оптимальном состоянии все время, где под оптимальностью понимаем сбалансированность дерева.

Идеально сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1. Поддержка идеальной сбалансированности, к сожалению, очень сложная задача. Другая идея заключается в том, чтобы ввести менее жесткие критерии сбалансированности и на их основе предложить достаточно простые алгоритмы обеспечения этих критериев.

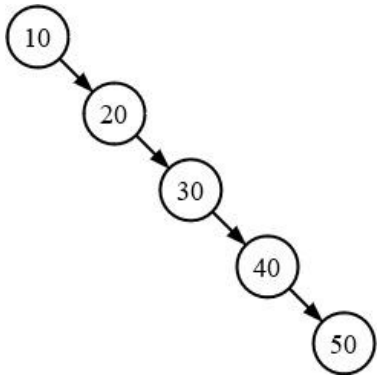


Рисунок 10.22 — Пример неэффективного бинарного дерева поиска

10.3. Двоичные кучи и очереди с приоритетом

Двоичная куча, также известная как пирамида, является особой формой бинарного дерева, которая имеет несколько важных свойств.

Первое свойство состоит в том, что значение в любой вершине двоичной кучи не превышает значения ее потомков. Это означает, что самая большая вершина находится в корне кучи, а все остальные вершины имеют значения, которые не превышают значение их потомков.

Второе свойство заключается в том, что уровень всех листьев, то есть вершин без потомков, отличается не более чем на 1. Это означает, что все листья находятся на самом нижнем уровне или на уровне выше на 1.

Третье свойство заключается в том, что последний уровень двоичной кучи заполняется слева направо без «дырок». Это означает, что все вершины на последнем уровне заполняются последовательно слева направо без пропусков.

Двоичные кучи широко используются в алгоритмах сортировки и приоритетной очереди. Они позволяют эффективно добавлять новые элементы, извлекать наибольший (или наименьший) элемент, а также обновлять значения вершин.

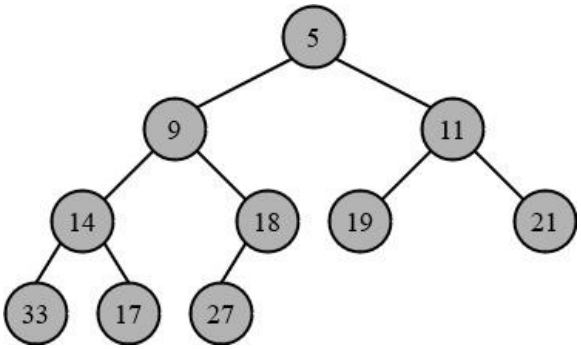


Рисунок 10.23 — Пример двоичной кучи

Для гарантированной логарифмической производительности необходимо поддерживать двоичную кучу в виде сбалансированного дерева. Для этого будем строить двоичную кучу в виде полного бинарного дерева — дерева, в котором каждый уровень кроме последнего содержит все возможные узлы, а последний уровень заполняется слева направо без пропусков.

Важным свойством полного бинарного дерева является то, что такое дерево может быть представлено в виде одного списка. Левый потомок родителя (имеющего индекс p) является элементом списка с индексом $2p$. Аналогично, правый потомок является элементом списка с индексом $2p + 1$. Для того чтобы найти индекс родительского узла нужно взять целую часть от индекса элемента, разделенного на 2.

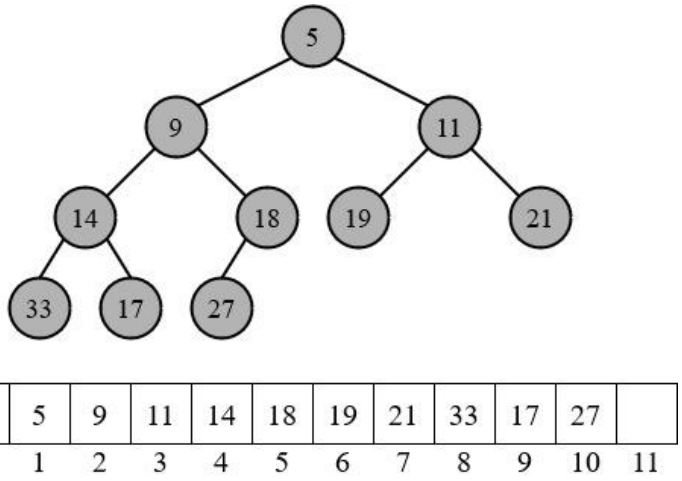


Рисунок 10.24 — Пример двоичной кучи и ее представления в виде списка

Базовые операции для двоичной кучи:

- BinaryHeap() — создает новую пустую бинарную кучу;
- insert(k) — добавить элемент в кучу, сложность $O(\log n)$;
- find_min() — возвратить минимальный элемент в куче, сложность $O(1)$;
- del_min() — возвратить минимальный элемент и исключить его из кучи, сложность $O(\log n)$;
- is_empty() — возвращает True, если куча пуста, и False в обратном случае;

- size() — количество элементов в куче;
- build_heap(list) — создает новую кучу на основе произвольного (неупорядоченного) массива, сложность $O(n)$.

Отсортировать массив можно путем превращения его в кучу, а кучи в отсортированный массив. Время работы $O(n \log n)$.

```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0
```

Рисунок 10.25 — Класс BinaryHeap

Код на рис. 10.25 описывает класс BinaryHeap, который представляет собой двоичную кучу. Метод __init__() инициализирует двоичную кучу. Он создает список heap_list с одним элементом равным нулю, чтобы обеспечить правильное индексирование элементов кучи. Переменная current_size устанавливается в ноль, чтобы отслеживать текущий размер кучи (количество элементов в куче).

Вставка элемента в двоичную кучу — это операция добавления нового элемента в уже существующую двоичную кучу, сохраняя при этом свойства кучи.

Операция вставки в двоичную кучу выполняется следующим образом:

1) Вставьте новый элемент в следующую доступную позицию в куче (то есть в конец массива, который представляет кучу). Это сохраняет свойство полного бинарного дерева.
2) Сравните новый элемент со своим родителем. Если новый элемент меньше своего родителя (в случае минимальной кучи) или больше своего родителя (в случае максимальной кучи), то поменяйте их местами.
3) Продолжайте этот процесс (называемый просачиванием вверх или просачиванием) до тех пор, пока не будет достигнут корень кучи или новый элемент не будет больше (для минимальной кучи) или меньше (для максимальной кучи) его родителя.

Этот процесс гарантирует, что после каждой операции вставки двоичная куча сохраняет свое свойство кучи. Время выполнения операции вставки в двоичную кучу составляет O(log n), где n — количество элементов в куче. В худшем случае нужно сделать столько же обменов, сколько уровней в куче, а количество уровней в полном бинарном дереве равно log(n).

На рис. 12.26 приведен пример вставки элемента в двоичную кучу.

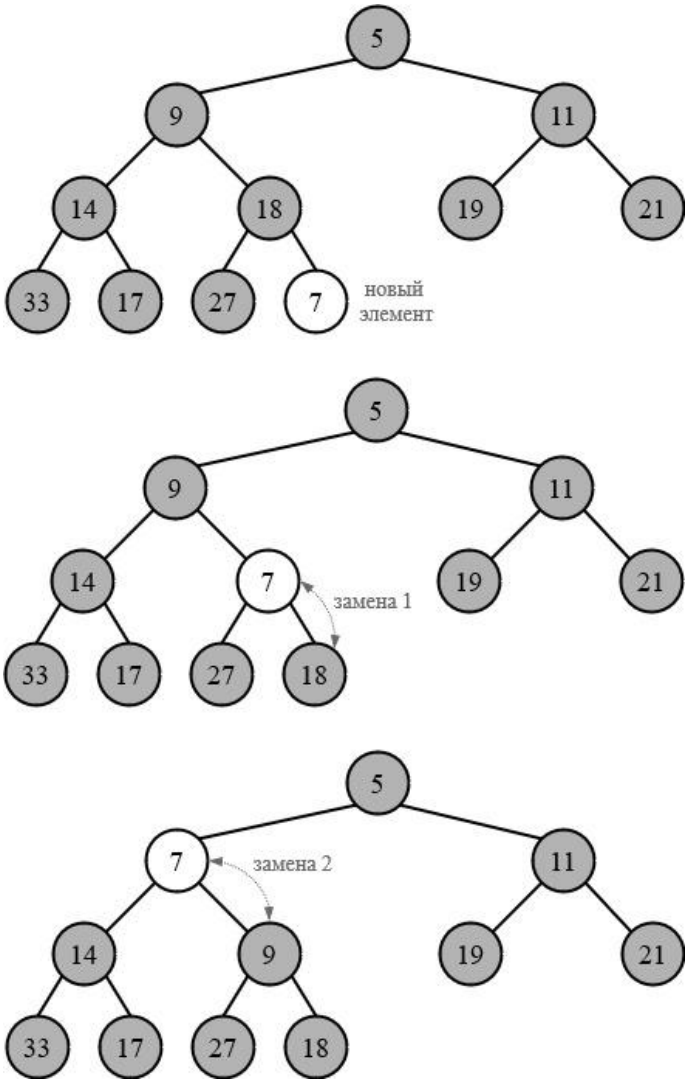


Рисунок 10.26 — Вставка нового узла в двоичную кучу

Сначала добавляем новый элемент в конец кучи. В данном случае добавляем узел 7 как правый потомок узла 18. После добавления нового элемента, необходимо убедиться, что свойство кучи (родительский узел меньше или равен своим потомкам) сохраняется. Если новый элемент меньше его родителя, то они меняются местами. Поскольку 7 меньше 18, то они меняются местами. Продолжаем этот процесс «всплытия» элемента до тех пор, пока не восстановим свойство кучи.

Код на рис. 10.27 предназначен для поддержки операции вставки элементов в кучу и перестройки кучи так, чтобы она соответствовала правилам двоичной кучи.

```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        perc_up(self, self.current_size)

    def perc_up(self, i):
        while i // 2 > 0:
            if self.heap_list[i] < self.heap_list[i // 2]:
                tmp = self.heap_list[i // 2]
                self.heap_list[i // 2] = self.heap_list[i]
                self.heap_list[i] = tmp
            else:
                break
            i = i // 2
```

Рисунок 10.27 — Пример реализации вставки нового элемента в двоичную кучу

Метод __init__() инициализирует пустую кучу с начальным значением [0] и текущим размером 0. Метод insert() используется для вставки нового элемента в кучу. Он добавляет элемент в конец списка heap_list и затем вызывает метод perc_up() для перестройки кучи так, чтобы она соответствовала правилам двоичной кучи.

Метод perc_up() используется для перемещения нового элемента вверх по куче до тех пор, пока он не достигнет своей правильной позиции. Он сравнивает новый элемент с его родителем и, если новый элемент меньше родителя, меняет их местами. Затем он обновляет индекс нового элемента на индекс его родителя и продолжает этот процесс до тех пор, пока новый элемент не будет на своем месте.

Удаление корневого элемента из двоичной кучи — это операция, которая используется для удаления наибольшего или наименьшего элемента из кучи.

Удаление корневого элемента из двоичной кучи в общем случае происходит следующим образом:

1) Сохраняем значение корневого элемента в отдельной переменной, чтобы его можно было вернуть после удаления.
2) Заменяем корневой элемент последним элементом в куче.
3) Удаляем последний элемент из кучи.
4) Просеиваем новый корневой элемент вниз по куче, чтобы восстановить ее свойства (куча с максимальным приоритетом или куча с минимальным приоритетом).
    - Если это куча с максимальным приоритетом, просеиваем элемент вниз, пока он не станет больше обоих своих дочерних элементов или пока не достигнет листового узла.

- Если это куча с минимальным приоритетом, просеиваем элемент вниз, пока он не станет меньше обоих своих дочерних элементов или пока не достигнет листового узла.
- При каждом шаге просеивания сравниваем значение текущего узла с его дочерними элементами и обмениваем его с наибольшим (в случае кучи с максимальным приоритетом) или наименьшим (в случае кучи с минимальным приоритетом) дочерним элементом, если это необходимо.

5) Возвращаем сохраненное значение корневого элемента.

Удаление корневого элемента из двоичной кучи выполняется за время $O(\log n)$, где n — количество элементов в куче. Просеивание элемента вниз по куче занимает время $O(\log n)$, так как на каждом шаге элемент спускается на один уровень вниз по дереву кучи.

Рассмотрим пример удаления корневого элемента 5 из двоичной кучи, изображенной на рис. 10.28. Это процесс включает в себя несколько этапов:

1) Сначала удаляется корневой элемент 5. Это оставляет пустое место в корне дерева. Чтобы заполнить это место, берется последний элемент в дереве (в данном случае 27) и помещается в корень.
2) Затем проверяется, соответствует ли новый корневой элемент свойству двоичной кучи — каждый родительский узел должен быть меньше или равен его потомкам. В данном случае 27 больше, чем оба его потомка (9 и 11), поэтому он меняется местами с наименьшим из потомков, которые меньше его (в данном случае 9).
3) Теперь 27 находится в узле, где раньше было число 9, и повторяется предыдущий шаг. Видно, что 27 все еще больше, чем оба его потомка (14 и 18), поэтому он меняется местами с наименьшим из потомков, которые меньше его (в данном случае 14).
4) Наконец, повторяется этот процесс еще раз и меняются местами 27 и 17, поскольку 27 все еще больше, чем оба его потомка.

Этот процесс продолжается до тех пор, пока не будет восстановлено свойство двоичной кучи — каждый родительский узел должен быть меньше или равен его потомкам.

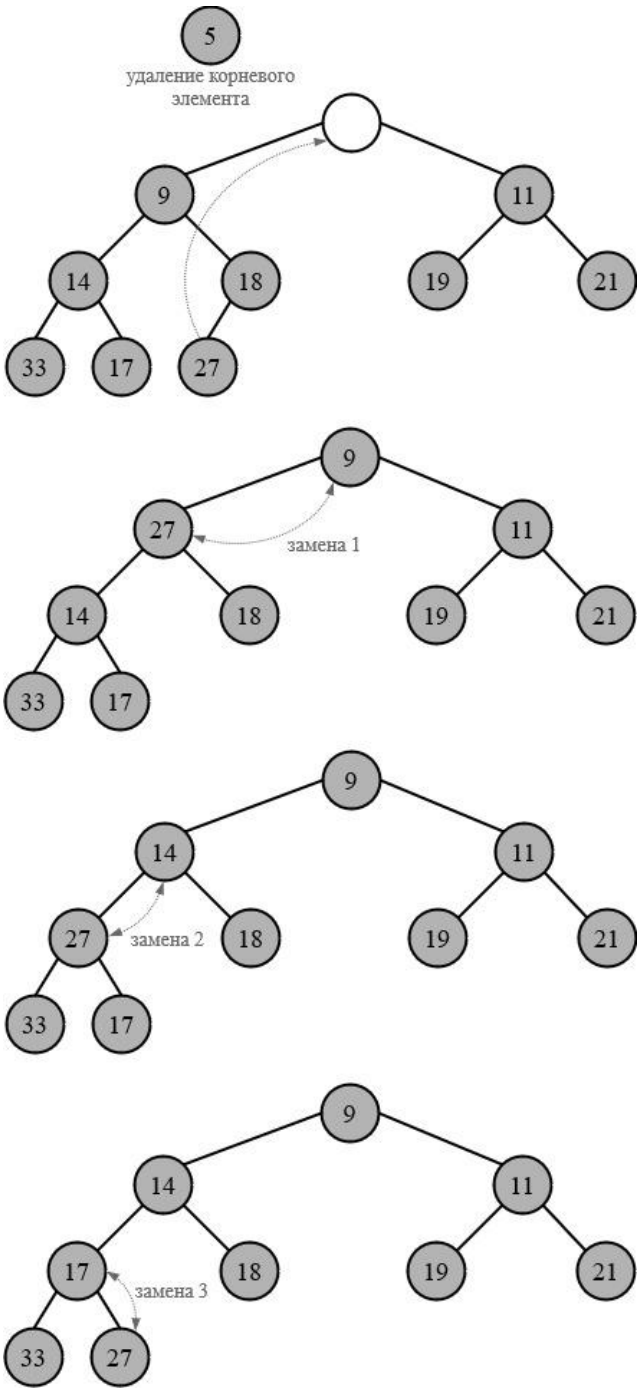


Рисунок 10.28 — Удаление корневого элемента из двоичной кучи

Наконец, рассмотрим код, реализующий бинарную кучу и содержащий методы для вставки элемента, удаления минимального элемента и построения кучи из списка (рис. 10.29).

```
def perc_down(self, i):
    while (i * 2) <= self.current_size:
        mc = min_child(self, i)
        if self.heap_list[i] > self.heap_list[mc]:
            tmp = self.heap_list[i]
            self.heap_list[i] = self.heap_list[mc]
            self.heap_list[mc] = tmp
        else:
            break
        i = mc
```

```
def min_child(self, i):
    if i * 2 + 1 > self.current_size:
        return i * 2
    else:
        if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1
```

```
class BinaryHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        perc_up(self, self.current_size)

    def del_min(self):
        ret_val = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.current_size]
        self.current_size = self.current_size - 1
        self.heap_list.pop()
        perc_down(self, 1)
        return ret_val

    def build_heap(self, a_list):
        i = len(a_list) // 2
        self.current_size = len(a_list)
        self.heap_list = [0] + a_list[:]
        while (i > 0):
            self.perc_down(i)
            i = i - 1
```

Рисунок 10.29 — Пример реализации удаления корневого элемента из двоичной кучи

Метод perc_down() выполняет операцию просеивания вниз для элемента с индексом *i*. Он сравнивает значение элемента с его наименьшим дочерним элементом и, если значение элемента больше, меняет их местами. Затем метод переходит к следующему уровню вниз и продолжает этот процесс до тех пор, пока не достигнет листовых узлов кучи или не найдет дочерний элемент, значение которого больше или равно.

Метод min_child() возвращает индекс наименьшего дочернего элемента для элемента с индексом *i*. Если правого дочернего элемента нет, возвращается индекс левого дочернего элемента. Если оба дочерних элемента есть, возвращается индекс наименьшего из них.

Класс BinaryHeap имеет атрибуты heap_list (список элементов кучи) и current_size (текущий размер кучи). В конструкторе класса устанавливаются начальные значения атрибутов. Метод insert() добавляет новый элемент в кучу, увеличивает размер кучи и вызывает метод perc_up(), чтобы перестроить кучу и удовлетворить свойство кучи.

Метод del_min() возвращает минимальный элемент кучи (корень), заменяет его последним элементом, уменьшает размер кучи, удаляет последний элемент и вызывает метод perc_down(), чтобы перестроить кучу и удовлетворить свойство кучи. Наконец, метод build_heap() строит кучу из заданного списка элементов. Он инициализирует начальное состояние кучи, затем применяет метод perc_down() ко всем элементам, начиная с середины списка и двигаясь вверх по уровням кучи.

Очередь с приоритетом — это структура данных, которая хранит элементы в определенном порядке в соответствии с их приоритетами. Каждый элемент в очереди с приоритетом имеет свой приоритет, который определяет его положение в очереди. Элементы с более высоким приоритетом располагаются ближе к началу очереди, а элементы с более низким приоритетом располагаются дальше от начала.

Для реализации очереди с приоритетом можно использовать двоичную кучу. В этом случае каждый узел двоичной кучи содержит элемент очереди с приоритетом, а приоритет элемента определяется значением в узле.

Операции, которые можно выполнять с очередью с приоритетом, включают:

- Вставка элемента с указанным приоритетом: новый элемент добавляется в очередь с приоритетом, и его положение определяется его приоритетом.
- Извлечение элемента с наивысшим приоритетом: элемент с наивысшим приоритетом удаляется из очереди с приоритетом.
- Получение элемента с наивысшим приоритетом: возвращается элемент с наивысшим приоритетом, но не удаляется из очереди с приоритетом.
- Изменение приоритета элемента: приоритет существующего элемента в очереди с приоритетом может быть изменен.

Реализация очереди с приоритетом с использованием двоичной кучи обеспечивает эффективное выполнение всех этих операций. Вставка нового элемента выполняется путем добавления нового узла в конец двоичной кучи и последующего выполнения операции «подъема» для установки узла в правильное положение. Извлечение элемента с наивысшим приоритетом выполняется путем удаления корневого узла (элемента с наивысшим приоритетом) и замены его последнего узла. Затем выполняется операция «опускания», чтобы восстановить свойство кучи.

Код для реализации очереди с приоритетом на основе двоичной кучи представлен на рис. 10.30.

```
class PriorityQueue:
    def __init__(self):
        self.bh = BinaryHeap()

    def enqueue(self, k):
        self.bh.insert(k)

    def dequeue(self):
        return self.bh.del_min()

    def is_empty(self):
        return self.bh.current_size == 0

    def size(self):
        return self.bh.current_size
```

Рисунок 10.30 — Пример реализации очереди с приоритетом

В этом коде класс PriorityQueue представляет собой обертку над BinaryHeap, предоставляя методы enqueue() и dequeue() для добавления и извлечения элементов с наивысшим приоритетом соответственно. Метод is_empty() проверяет, пуста ли очередь, а метод size() возвращает текущий размер очереди.

10.4. Задачи для самостоятельного решения

Задача 10.1. Решите задачу на работу с бинарным деревом.

1) Реализовать класс бинарного дерева. Написать функцию для поиска элемента в бинарном дереве.
2) Реализовать класс бинарного дерева. Написать функцию для проверки, является ли бинарное дерево сбалансированным.
3) Реализовать класс бинарного дерева. Написать функцию для поиска наименьшего и наибольшего элементов в бинарном дереве.
4) Реализовать класс бинарного дерева. Написать функцию для проверки, является ли бинарное дерево деревом поиска.
5) Реализовать класс бинарного дерева. Написать функцию для нахождения наибольшей глубины листьев в бинарном дереве.
6) Реализовать класс бинарного дерева. Написать функцию для нахождения наименьшего общего предка двух узлов в бинарном дереве.
7) Реализовать класс бинарного дерева. Написать функцию для нахождения суммы всех элементов в бинарном дереве.

8) Реализовать класс бинарного дерева. Написать функцию для нахождения диаметра бинарного дерева (максимального расстояния между двумя узлами).
9) Реализовать класс бинарного дерева. Написать функцию для нахождения всех узлов, которые имеют только одного потомка в бинарном дереве.
10) Реализовать класс бинарного дерева. Написать функцию для нахождения всех узлов, которые имеют двух потомков в бинарном дереве.
11) Реализовать класс бинарного дерева. Написать функцию для нахождения всех узлов, которые являются листьями в бинарном дереве.
12) Реализовать класс бинарного дерева. Написать функцию для проверки, является ли бинарное дерево полным.
13) Реализовать класс бинарного дерева. Написать функцию для проверки, является ли бинарное дерево симметричным (зеркально симметричным).
14) Реализовать класс бинарного дерева. Написать функцию для нахождения всех путей от корня до листьев в бинарном дереве.
15) Реализовать класс бинарного дерева. Написать функцию для нахождения всех узлов на заданной глубине в бинарном дереве.

Задача 10.2. Решите задачу на работу с двоичной кучей.

1) Набор данных представляет собой пары «ключ-значение». Реализуйте структуру данных на основе двоичной кучи, которая будет поддерживать операции вставки пары, удаления пары по ключу и поиска значения по ключу.
2) Работает система, которая отслеживает активность пользователей на веб-сайте. Каждый раз, когда пользователь посещает страницу, система создает запись с временной меткой. Реализовать структуру данных на основе двоичной кучи, которая будет поддерживать операции добавления записи и извлечения записей за определенный период времени.
3) Даны две двоичные кучи. Напишите функцию, которая объединяет эти две кучи в одну.
4) Дан список задач с указанием времени выполнения каждой задачи. Напишите программу, использующую двоичную кучу, для определения порядка выполнения задач таким образом, чтобы минимизировать общее время ожидания.
5) Создайте структуру данных «Top N», которая хранит N наибольших элементов из потока входных данных. Структура должна поддерживать операцию добавления нового элемента в поток и операцию извлечения одного из Top N элементов. Используйте для этого двоичную кучу.

6) Предоставлен набор данных, содержащий информацию о студентах и их оценках. Разработайте систему ранжирования студентов по их среднему баллу с использованием двоичной кучи.
7) Даны две отсортированные последовательности. Напишите функцию, которая объединяет эти две последовательности в одну отсортированную последовательность с использованием двоичной кучи.
8) Дан массив строк. Напишите программу, которая использует двоичную кучу для сортировки массива в лексикографическом порядке.
9) Имеется набор данных, представляющих собой пары «слово-частота». Реализуйте структуру данных на основе двоичной кучи, которая будет поддерживать операции вставки пары, удаления пары по слову и поиска частоты по слову.
10) Имеется система управления задачами. Каждая задача имеет приоритет и дедлайн. Реализовать структуру данных на основе двоичной кучи, которая будет поддерживать операции добавления задачи и извлечения задачи с наивысшим приоритетом и дедлайном до определенного времени.
11) Реализуйте алгоритм поиска медианы в отсортированном массиве с использованием двоичной кучи.
12) Дан массив точек на плоскости. Напишите программу, которая использует двоичную кучу для сортировки точек по их расстоянию от начала координат.
13) Имеется набор данных, представляющих собой пары «имя-возраст». Реализуйте структуру данных на основе двоичной кучи, которая будет поддерживать операции вставки пары, удаления пары по имени и поиска возраста по имени.
14) Разработайте систему управления автобусным расписанием. Каждый автобус имеет номер маршрута и время отправления. Реализовать структуру данных на основе двоичной кучи, которая будет поддерживать операции добавления автобуса и извлечения автобуса с наименьшим временем отправления.
15) Разработайте систему учета книг в библиотеке. Каждая книга имеет название и количество экземпляров. Реализовать структуру данных на основе двоичной кучи, которая будет поддерживать операции добавления книги и извлечения книги с наименьшим количеством экземпляров.

11. Хеш-таблицы

11.1. Ассоциативный массив. Таблица с прямой адресацией

Словарь, также известный как ***ассоциативный массив***, map или dictionary, является абстрактной структурой данных, которая обеспечивает возможность хранения пар «ключ-значение». Он представляет собой некий контейнер, в котором каждый элемент имеет уникальный идентификатор, или ключ, и связанное с ним значение.

Основные операции, которые поддерживаются в словаре, включают добавление новой пары «ключ-значение», поиск пары по ключу и удаление пары по ключу. Важно отметить, что каждый ключ в словаре должен быть уникальным. Это означает, что ассоциативный массив не может содержать две пары с одинаковыми ключами. Если попытаться добавить пару с уже существующим ключом, то новое значение заменит старое.

С точки зрения интерфейса, ассоциативный массив можно рассматривать как обычный массив. Однако отличие состоит в том, что в качестве индексов можно использовать не только целые числа из определенного диапазона, но и значения других типов данных. Например, это могут быть строки, даты или пользовательские объекты.

В языке программирования Python словарь реализуется при помощи встроенного типа dict(). Это позволяет создавать и манипулировать словарями в очень простой и удобной форме. Словари в Python широко используются для хранения и обработки данных в различных областях, начиная от научных вычислений и заканчивая веб-разработкой.

```python
d1 = dict() # создание пустого словаря

d1['abc'] = 42 # put(key, val) добавление пары ключ-значение в словарь;
# если по данному ключу уже есть значение – то оно заменяется новым

v = d1['abc'] # get(key) извлечение значения по ключу;
# в случае отсутсутствия значения по ключу возникает исключительная ситуация KeyError
print(v)

b = 'xyz' in d1 # возвращает True, если ключ содержится в словаре, иначе возвращает False
print(b)

it = iter(d1) # итератор по умолчанию генерирует последовательность всех ключей хранящихся в словаре
print(list(it))

l = len(d1) # возвращает количество пар ключ-значение в словаре
print(l)

del d1['abc'] # удаляет из словаря пару ключ-значение с заданным ключом
```

```
42
False
['abc']
1
```

Рисунок 11.1 — Базовые операции словаря

Код на рис. 11.1 создает словарь d1, добавляет в него пару ключ-значение 'abc': 42, и выводит значение по ключу 'abc'. Затем проверяет наличие ключа 'xyz' в словаре, выводит список всех ключей и количество пар ключ-значение в словаре. В конце удаляет пару с ключом 'abc'.

Рассмотрим различные решения задачи реализации словаря.

Таблица с прямой адресацией представляет собой эффективный способ организации данных в случаях, когда количество ключей невелико, и каждому ключу может быть выделена своя уникальная ячейка в массиве.

Допустим, приложению необходимо динамическое множество. Каждый элемент этого множества имеет ключ из набора $U = \{0,1,\ldots,m-1\}$, где m — это число, которое не слишком велико. Также важно отметить, что в этом предположении учитывается, что никакие два элемента не могут иметь одинаковые ключи.

Для представления данного динамического множества используется специальная структура данных — массив или таблица с прямой адресацией. В такой таблице каждая ячейка соответствует определенному ключу из пространства ключей U.

Одной из основных преимуществ использования прямой адресации является возможность прямого индексирования элементов массива. Это обеспечивает быстрый доступ к любому элементу массива за время $O(1)$, что является очень быстрым временем доступа. Прямое индексирование становится возможным, если есть возможность выделить массив достаточного размера, чтобы каждому возможному значению ключа соответствовала своя ячейка.

Рассмотрим пример, который демонстрирует реализацию функции хеширования для строк, состоящих из строчных латинских символов.

```
def to_key(s=None):
    '''Возвращает неотрицательный ключ по объекту.
       Если объект не задан, то возвращает максимальное значение ключа.
    '''
    if s is None:
        s = 'z'
    return ord(s) - ord('a')
```

```
to_key('a')
```

```
0
```

```
to_key('c')
```

```
2
```

```
to_key()
```

```
25
```

Рисунок 11.2 — Функция to_key() для генерации ключей

Функция to_key(), показанная на рис. 11.2, преобразует каждый символ в числовое значение, равное разности между ASCII-кодом этого символа и ASCII-кодом символа 'a'. Если аргумент не передан, то используется символ 'z', что соответствует максимальному значению ключа.

```
def seq_to_key(seq):
    res = 0
    n = to_key() # количество допустимых символов
    for i, el in enumerate(seq):
        assert 0 <= to_key(el) <= n # проверка корректности рассматриваемого символа
        res += to_key(el) * n ** i
    return res
```

```
seq_to_key('a')
```

```
0
```

```
seq_to_key('z')
```

```
25
```

```
seq_to_key('bb')
```

```
26
```

```
seq_to_key('abc')
```

```
1275
```

```
seq_to_key('aa') # проблема!
```

```
0
```

Рисунок 11.3 — Функция seq_to_key() для генерации ключа из последовательности символов

На рис. 11.3 функция seq_to_key() преобразует всю строку в числовой ключ, используя предыдущую функцию to_key(). Она перебирает все символы строки, преобразует каждый в числовое значение и суммирует их, умножая на степень числа символов (n). Однако возникает проблема с одинаковыми ключами для различных строк. Например, строки 'aa' и 'bb' будут иметь одинаковые ключи. Чтобы исправить это, функцию to_key() нужно изменить таким образом, чтобы возвращать положительные ключи, а не ноль (рис. 11.4).

```
def to_key(s=None):
    '''Возвращает ПОЛОЖИТЕЛЬНЫЙ ключ по объекту.
       Если объект не задан, то возвращает максимальное значение ключа.
    '''
    if s is None:
        s = 'z'
    return ord(s) - ord('a') + 1
```

```
[seq_to_key(x) for x in ['','a', 'z', 'aa', 'ab', 'zz', 'aaa']]
```

```
[0, 1, 26, 27, 53, 702, 703]
```

Рисунок 11.4 — Измененная функция to_key()

Теперь рассмотрим класс DirectStrTable, который представляет собой структуру данных для хранения строковых значений (ассоциативным массив строк) (рис. 11.5).

```
class DirectStrTable:
    def __init__(self, str_len):
        self._str_len = str_len
        self._n = self._to_key()
        self._none = object()
        total = 0
        for i in range(self._str_len + 1):
            total *= self._n
            total += 1
        self._table = [self._none] * total
        self._len = 0

    def _to_key(self, s=None):
        if s is None:
            s = 'z'
        return ord(s) - ord('a') + 1

    def _from_key(self, k):
        return chr(k + ord('a') - 1)

    def _str_to_key(self, seq):
        res = 0
        assert len(seq) <= self._str_len
        for i, el in enumerate(seq):
            assert 0 <= to_key(el) <= self._n
            res += to_key(el) * self._n ** i
        return res

    def _key_to_str(self, k):
        s = ''
        while k > 0:
            k, m = divmod(k, self._n)
            if m == 0:
                s += self._from_key(self._n)
                k -= 1
            else:
                s += self._from_key(m)
        return s
```

Рисунок 11.5 — Описание класса DirectStrTable (начало)

Конструктор класса принимает аргумент str_len — максимальную длину строк, которые могут быть сохранены в таблице. Внутри конструктора инициализируются следующие атрибуты:

- _str_len — максимальная длина строк;
- _n — количество возможных символов в строке (в данном случае от 'a' до 'z');
- _none — объект-заглушка, используется для обозначения пустых ячеек в таблице;
- _table — список, представляющий таблицу, инициализируется пустыми ячейками;
- _len — количество элементов, хранящихся в таблице.

Метод _to_key() преобразует символ строки в числовой ключ, используя кодировку ASCII. Метод _from_key() выполняет обратное преобразование — из числового ключа в символ. Метод _str_to_key() преобразует строку в числовой ключ. При этом проверяется, что длина строки не превышает максимальную длину, а каждый символ имеет корректный ключ. Наконец, метод _key_to_str() выполняет обратное преобразование — из числового ключа в строку.

```
    def __len__(self):
        return self._len

    def __getitem__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            raise KeyError(f'Key Error: {repr(s)}')
        else:
            return self._table[k]

    def __contains__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            return False
        else:
            return True

    def __setitem__(self, str_key, val):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            self._table[k] = val
            self._len += 1
        else:
            self._table[k] = val

    def __delitem__(self, str_key):
        k = self._str_to_key(str_key)
        if self._table[k] is self._none:
            raise KeyError(f'Key Error: {repr(s)}')
        else:
            self._table[k] = self._none
        self._len -= 1

    def _raw_iter(self):
        for rk, val in enumerate(self._table):
            if val is not self._none:
                yield rk, val

    def __iter__(self):
        for rk, val in self._raw_iter():
            yield self._key_to_str(rk)

    def items(self):
        for rk, val in self._raw_iter():
            yield self._key_to_str(rk), val

    def values(self):
        for val in self._table:
            if val is not self._none:
                yield val

    def stat(self):
        return f'''Хранится элементов: {self._len};
размер таблицы: {len(self._table)};
доля используемых элементов таблицы: {self._len/len(self._table)}'''
```

Рисунок 11.6 — Описание класса DirectStrTable (конец)

Класс переопределяет следующие методы (рис. 11.6):

- __len__ — возвращает количество элементов в таблице;
- __getitem__ — возвращает значение по заданному ключу. Если ключ не существует, генерируется исключение KeyError;
- __contains__ — проверяет наличие ключа в таблице;
- __setitem__ — сохраняет значение по заданному ключу. Если ключ уже существует, значение перезаписывается;
- __delitem__ — удаляет значение по заданному ключу. Если ключ не существует, генерируется исключение KeyError;
- __iter__ — возвращает итератор, который перебирает все ключи в таблице;
- items — возвращает итератор, который перебирает все пары ключ-значение в таблице;
- values — возвращает итератор, который перебирает все значения в таблице;
- stat — возвращает строку со статистикой о таблице (количество элементов, размер таблицы, доля используемых элементов).

На рис. 11.7 представлен код, демонстрирующий работу с классом DirectStrTable. В начале создается экземпляр данного класса с размером 3. Для этого используется конструкция dst1 = DirectStrTable(3). Далее проверяется длина созданного экземпляра с помощью функции len(). После этого в таблицу добавляется пара ключ-значение. В качестве ключа используется строка 'ab', в качестве значения — число 11. Это достигается с помощью операции dst1['ab'] = 11. Затем снова проверяется длина таблицы, len(dst1), и значение по ключу 'ab', dst1['ab']. Также производится проверка наличия ключей 'ab' и 'a' в таблице с помощью оператора in.

В таблицу добавляются новые пары ключ-значение: dst1['abc'] = 7 и dst1['zz'] = 77. Затем из нее удаляется пара по ключу 'zz' с помощью операции del dst1['zz']. После этого в таблицу добавляются еще несколько пар ключ-значение, и снова проверяется длина таблицы.

Таблица может быть преобразована в список с помощью функции list(). Можно также получить список из итератора _raw_iter() и список пар ключ-значение из таблицы. Кроме того, можно получить список значений из таблицы, используя метод values().

Для сравнения создается словарь d1 и заполняется двумя парами ключ-значение: d1 = dict([('ab', 11), ('abc', 7)]). Словарь может быть преобразован в список, а также можно получить список значений из словаря. В конце кода выводится статистика по таблице.

```
dst1 = DirectStrTable(3)
```

```
len(dst1)
```

```
0
```

```
dst1['ab'] = 11
```

```
len(dst1)
```

```
1
```

```
dst1['ab']
```

```
11
```

```
'ab' in dst1, 'a' in dst1
```

```
(True, False)
```

```
dst1['abc'] = 7
```

```
len(dst1)
```

```
2
```

```
dst1['zz'] = 77
len(dst1)
```

```
3
```

```
del dst1['zz']
len(dst1)
```

```
2
```

```
dst1['zzz'] = 1
dst1['x'] = 5
dst1['z'] = 6
dst1['aa'] = 7
dst1['zz'] = 8
dst1['aaa'] = 9
```

```
len(dst1)
```

```
8
```

```
list(dst1)
```

```
['x', 'z', 'aa', 'ab', 'zz', 'aaa', 'abc', 'zzz']
```

```
list(dst1._raw_iter())
```

```
[(24, 5),
 (26, 6),
 (27, 7),
 (53, 11),
 (702, 8),
 (703, 9),
 (2081, 7),
 (18278, 1)]
```

```
list(dst1.items())
```

```
[('x', 5),
 ('z', 6),
 ('aa', 7),
 ('ab', 11),
 ('zz', 8),
 ('aaa', 9),
 ('abc', 7),
 ('zzz', 1)]
```

```
list(dst1.values())
```

```
[5, 6, 7, 11, 8, 9, 7, 1]
```

```
d1 = dict([('ab', 11), ('abc', 7)])
```

```
list(d1)
```

```
['ab', 'abc']
```

```
list(d1.values())
```

```
[11, 7]
```

```
print(dst1.stat())
```

```
Хранится элементов: 8;
размер таблицы: 18279;
доля используемых элементов таблицы: 0.00043766070353958096
```

Рисунок 11.7 — Работа с объектом класса DirectStrTable

Каждая из приведенных операций обладает высокой скоростью выполнения. Время их работы составляет всего $O(1)$, что означает, что они выполняются за постоянное количество времени независимо от объема данных. Это особенно полезно в некоторых приложениях, где элементы динамического множества могут храниться непосредственно в таблице с прямой адресацией. Прямая адресация предполагает использование ключа элемента как индекса для его хранения в таблице, что позволяет быстро и просто обращаться к нему.

Однако у прямой адресации есть и свои недостатки. Основной из них заключается в том, что если пространство ключей U велико, то хранение таблицы T размером $|U|$ становится непрактичным или даже невозможным. Это зависит от количества доступной

памяти и размера пространства ключей. Если множество K реально сохраненных ключей мало по сравнению с пространством ключей U, то большая часть памяти, выделенной для таблицы T, будет расходоваться напрасно, то есть большая часть таблицы будет не использована, что является неэффективным использованием ресурсов.

Таким образом, прямая адресация может быть очень эффективной для небольших наборов данных или когда пространство ключей близко к размеру набора данных. Но при больших объемах данных и значительном различии между размером пространства ключей и фактическим количеством ключей, она может стать причиной неэффективного использования памяти.

11.2. Хеш-таблица и хеш-функция

Хеш-таблица — это эффективная структура данных для реализации словарей. В отличие от связанного списка, время поиска элемента в хеш-таблице может быть таким же долгим, как и в связанном списке в наихудшем случае, а именно, $O(n)$. Однако на практике хеширование является очень эффективным. При определенных допущениях математическое ожидание времени поиска элемента в хеш-таблице составляет $O(1)$.

Хеш-таблица представляет собой обобщение обычного массива. Если количество ключей, которые фактически хранятся в массиве, мало по сравнению с количеством возможных значений ключей, то хеш-таблица становится более эффективной альтернативой массиву с прямой индексацией. Хеш-таблица обычно использует массив размером, пропорциональным количеству фактически хранящихся ключей. Вместо использования ключа в качестве индекса массива, индекс вычисляется на основе значения ключа. Идея хеширования заключается в использовании частичной информации из ключа для вычисления хеш-адреса $h(key)$, который используется для индексации в хеш-таблице.

Когда количество ключей, хранящихся в словаре, намного меньше пространства возможных ключей U, хеш-таблица требует гораздо меньше памяти, чем таблица с прямой адресацией. Вместе с тем, время поиска элемента в хеш-таблице остается $O(1)$. Однако стоит отметить, что это среднее время поиска, в то время как для таблицы с прямой адресацией это время относится к наихудшему случаю.

Хеш-таблица представляет собой коллекцию, где элементы (значения) индексируются. Каждая позиция в хеш-таблице, называемая слотом таблицы (slot, bucket), может содержать элемент, адресованный целочисленным неотрицательным индексом внутри таблицы.

В таблице есть слоты с индексами 0, 1 и так далее. При создании хеш-таблицы все слоты пустые. В Python хеш-таблицу можно реализовать в виде списка, заполненного значениями None или их заменителями (если нужно хранить значения None).

При использовании прямой адресации элемент с ключом k хранится в ячейке с индексом k. При хешировании этот элемент хранится в ячейке с индексом $h(k)$, где хеш-функция h используется для вычисления индекса для данного ключа k. Функция h отображает пространство ключей U на ячейки хеш-таблицы $T[0\ldots m-1]$:

$$h: U \to \{0,1,\ldots,m-1\}.$$

Говорится, что элемент с ключом k хешируется в ячейку с индексом $h(k)$), и это значение $h(k)$ называется хеш-значением ключа k.

Рассмотрим пример хеш-таблицы, имеющий размер 11, и хеш-функции, определенной как $k \bmod 11$, где k — ключ элемента:

$$T[0,\ldots,10];\ h(k) = k \bmod 11,$$

то есть для каждого ключа k его хеш-значение (индекс в хеш-таблице) будет равно остатку от деления k на 11.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

Рисунок 11.8 — Пустая хеш-таблица

Код на рис. 11.9 содержит определение хеш-функции h(k), которая принимает на вход значение k и возвращает остаток от деления этого значения на 11. Затем определяется список keys1, содержащий значения [54, 26, 93, 17, 77, 31]. С помощью генератора списка, для каждого значения k из списка keys1 вызывается хеш-функция h(k). Результаты хеш-функции для каждого значения k записываются в новый список, который и выводится на экран.

```
def h(k):
    return k % 11

keys1 = [54, 26, 93, 17, 77, 31]

[h(k) for k in keys1]
```

```
[10, 4, 5, 6, 0, 9]
```

Рисунок 11.9 — Определение хеш-функции.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Рисунок 11.10 — Хеш-таблица с шестью элементами

При построении хеш-таблиц возникает одна значительная проблема. Эта проблема заключается в том, что два различных ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется коллизией, и она может создать проблемы при доступе к данным.

Приведем пример коллизии. Предположим, имеется два ключа: 12 и 23. Используя описанную выше хеш-функцию h, можно получить следующие хеш-значения: h(12) и h(23). Оба эти значения будут равны 1, что означает, что оба ключа будут хешированы в одну и ту же ячейку.

Это происходит потому, что хеш-функция является детерминистической, то есть для одного и того же значения ключа k она всегда дает одно и то же хеш-значение $h(k)$. Таким образом, если размер универсального множества ключей $|U|$ больше размера множества хеш-значений m, то должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Следовательно, полностью избежать коллизий невозможно в принципе.

Однако существуют два основных подхода для борьбы с этой проблемой. Первый подход заключается в выборе такой хеш-функции, которая снижает вероятность возникновения коллизий. Это может быть достигнуто различными способами, например, через использование более сложных или более случайных хеш-функций.

Второй подход заключается в использовании эффективных алгоритмов разрешения коллизий. Такие алгоритмы предлагают способы обработки ситуаций, когда два ключа хешируются в одну и ту же ячейку. Например, одним из таких алгоритмов может быть метод цепочек, при котором в каждой ячейке хранится список всех элементов с соответствующим хеш-значением.

Хеш-функция выполняет преобразование массива входных данных произвольной длины (ключа, сообщения) в (выходную) битовую строку установленной длины (хеш, хеш-код, хеш-сумму).

Хеш-функции применяются в следующих задачах:

- Построение ассоциативных массивов;
- Поиск дубликатов в сериях наборов данных;
- Построение уникальных идентификаторов для наборов данных;

- Вычисление контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
- Сохранение паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
- Выработка электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ») и многих других.

Для решения различных задач требования к хеш-функциям могут существенно отличаться.

«Хорошая» хеш-функция должна удовлетворять двум свойствам:

- быстрое вычисление;
- минимальное количество коллизий.

Для обеспечения минимального количества коллизий хеш-функция должна приближенно удовлетворять предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из m ячеек, независимо от хеширования остальных ключей. Однако это условие обычно невозможно проверить, так как распределение вероятностей, в соответствии с которым поступают вносимые в таблицу ключи, неизвестно, и вставляемые ключи могут не быть независимыми.

При построении хеш-функции хорошим подходом является подбор функции таким образом, чтобы она никак не коррелировала с закономерностями, которым могут подчиняться существующие данные. Например, можно потребовать, чтобы «близкие» в некотором смысле ключи давали далекие хеш-значения (например, хеш-функция для подряд идущих целых чисел давала далекие хеш-значения). В некоторых приложениях хеш-функций требуется противоположное свойство — непрерывность (близкие ключи должны порождать близкие хеш-значения).

Обычно от хеш-функций ожидается, что значения хеш-функции находятся в диапазоне от 0 до $m-1$. Часто удобно, если $m=2^n$. Таким образом значение хеш-функции может, например, без преобразований храниться в машинном слове.

Метод деления — это один из способов построения хеш-функции. Принцип работы данного метода заключается в отображении ключа k в определенную ячейку путем получения остатка от деления k на m. В математическом выражении это выглядит как $h(k)=k \bmod m$, где $h(k)$ — это хеш-функция, k — ключ, а m — число ячеек.

При использовании метода деления необходимо учесть некоторые особенности выбора значения m. Например, m не должно быть степенью 2, так как в этом случае $h(k)$ будет представлять собой просто p младших битов числа k. Это может привести к тому, что хеш-функция будет не равномерной и, как следствие, неэффективной. Если заранее неизвестно, что все наборы младших p битов ключей равновероятны, то лучше построить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа.

Важно отметить, что зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, которое достаточно далеко от степени двойки. Это связано с тем, что простые числа обладают свойствами, которые позволяют более равномерно распределить ключи по ячейкам. Таким образом, метод деления позволяет создавать эффективные хеш-функции, обеспечивающие быстрый доступ к данным.

Рассмотрим пример хеш-функции, построенной по методу деления (рис. 11.11).

```
def str_h(s, m=701):
    return sum(ord(symb) for symb in s) % m
```

```
s1 ='''Хеш-функция выполняет преобразование массива входных данных произвольной
длины (ключа, сообщения) в (выходную) битовую
строку установленной длины (хеш, хеш-код, хеш-сумму).'''
s1_cod = [(s, str_h(s)) for s in s1.split()]
s1_cod
```

```
[('Хеш-функция', 391),
 ('выполняет', 671),
 ('преобразование', 403),
 ('массива', 550),
 ('входных', 596),
 ('данных', 201),
 ('произвольной', 397),
 ('длины', 516),
 ('(ключа,', 611),
 ('сообщения)', 3),
 ('в', 373),
 ('(выходную)', 375),
 ('битовую', 586),
 ('строку', 217),
 ('установленной', 64),
 ('длины', 516),
 ('(хеш,', 546),
 ('хеш-код,', 290),
 ('хеш-сумму).', 425)]
```

```
print(sorted(s[1] for s in s1_cod))
```

```
[3, 64, 201, 217, 290, 373, 375, 391, 397, 403, 425, 516, 516, 546, 550, 586, 596, 611, 671]
```

Рисунок 11.11 — Пример хеш-функции для строк, построенной по методу деления

Функция str_h() принимает строку s и число m (по умолчанию равное 701) и возвращает остаток от деления суммы числовых значений каждого символа строки на m. Числовые значения символов определяются с помощью встроенной функции ord(), которая возвра-

щает Unicode-код символа. Затем создается список слов из строки s1, к каждому слову применяется функция str_h(), и создается список кортежей, где первый элемент — это исходное слово, а второй — его хеш-значение. Наконец, отсортированный список хеш-кодов слов из списка s1_cod выводится на экран.

Метод MAD (Multiply-Add-and-Divide) представляет собой хеш-функцию, которая преобразует целое число k в соответствии с определенным алгоритмом. Для работы хеш-функции используются следующие параметры: p — большое простое число, $a \in \{1, 2, \ldots, p-1\}$ и $b \in \{0, 1, \ldots, p-1\}$, а также m — количество значений в диапазоне значений хеш-функции. Формула преобразования выглядит следующим образом:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m.$$

Одним из преимуществ данного класса хеш-функций является то, что размер m выходного диапазона может быть произвольным и не обязательно должен быть простым числом. В этом семействе будет содержаться $p(p-1)$ хеш-функций, поскольку число a можно выбрать $p-1$ способами, а число b — p способами.

Хеш-функцию MAD можно использовать для универсального хеширования — техники, при которой не одна конкретная хеш-функция используется, а выбор происходит из заданного семейства по случайному алгоритму. Семейство универсальных хеш-функций считается таковым, если для любых двух допустимых ключей вероятность коллизии минимальна:

$$\Pr_{h \in H}[h(x) = h(y)] = \frac{1}{m}, \; x \neq y.$$

Важно подчеркнуть, что данное условие должно выполняться для любых различных ключей, при этом функции хеширования h случайно выбираются из всего класса универсальных хеш-функций H.

Универсальное хеширование характеризуется низким числом коллизий и широко применяется в различных областях, включая реализацию хеш-таблиц и криптографию. В частности, доказано, что хеш-функции MAD для случайно выбранных a и b образуют класс универсальных хеш-функций.

Рассмотрим класс MadHash, который реализует простой хэш-алгоритм (рис. 11.12). Чтобы уменьшить количество коллизий, используем простое число mad_p. В классе MadHash есть два метода: __init__() и h(). Метод __init__() инициализирует объект класса с заданным числом m, а также случайными числами a и b. Метод h() — это собственно хэш-функция. Она принимает на вход значение k и вычисляет хэш этого значения по формуле ((self.a * k + self.b) % self.p) % self.m. Затем создается объект mh класса MadHash

с параметром m = 2**8. Значения a и b для этого объекта выводятся на экран. Далее вычисляются хэши для чисел от 0 до 127 с помощью этой хэш-функции и выводятся на экран. После этого эти хэши сортируются и проверяются на наличие дубликатов. Результат этой проверки выводится на экран в виде пары (хэш, есть ли дубликат) (рис. 11.13).

```
import random
```

```
mad_p = 4294967311
```

```
class MadHash:
    def __init__(self, m):
        self.m = m
        self.p = mad_p
        assert self.p > self.m
        self.a = random.randint(1, self.p)
        self.b = random.randint(0, self.p)

    def h(self, k):
        return ((self.a * k + self.b) % self.p ) % self.m
```

```
mh = MadHash(2**8)
mh.a, mh.b
```

```
(3410987614, 242347065)
```

```
hr1 = [mh.h(v) for v in range(128)]
```

```
for i, h in enumerate(hr1):
    print(i, h)
```

```
0 67
1 219
2 115
3 26
4 178
5 74
6 226
7 137
8 33
9 185
10 81
```

Рисунок 11.12 — Пример хеш-функции, построенной по методу MAD

```
hr1s = sorted(hr1)
list(zip(hr1s, [x0 == x1 for x0, x1 in zip(hr1s[:-1], hr1s[1:])]))
```

```
[(0, False),
 (1, False),
 (5, False),
 (6, False),
 (7, False),
 (12, False),
 (13, False),
 (14, False),
 (19, False),
 (20, False),
 (21, False),
 (26, True),
 (26, False),
 (27, False),
 (28, False),
 (33, True),
```

Рисунок 11.13 — Вывод информации о дубликатах хеш-значений

Часто расчет хеш-функции $h(k)$ можно представить в форме двух последовательных операций: вычисления хеш-кода, преобразующего ключ k в целое число, и функции компрессии, которая переводит полученный хеш-код в целое число в пределах заданного диапазона $[0, m-1]$ (рис. 11.14).

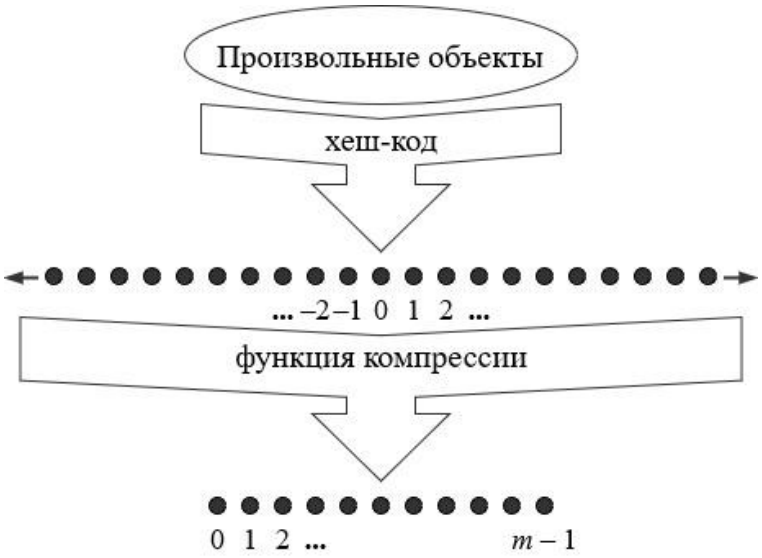


Рисунок 11.14 — Схема построения хеш-функции на базе двух шагов

На первом шаге (hash code) происходит преобразование ключа в целое число. Это может быть простым преобразованием, например, если ключи являются целыми числами, то хеш-кодом может быть сам ключ. Если ключи представляют собой строки или другие сложные структуры данных, то может быть использована более сложная функция для вычисления хеш-кода. Главное требование к функции вычисления хеш-кода — она должна быть детерминированной, то есть для одного и того же ключа всегда должен получаться один и тот же хеш-код.

На втором шаге (compression function) происходит преобразование полученного на первом этапе хеш-кода в индекс в пределах заданного диапазона $[0, m-1]$, где m — размер хеш-таблицы. Это необходимо для того, чтобы можно было сохранить значение по соответствующему ключу в хеш-таблице. Функция компрессии также должна быть детерминированной, то есть для одного и того же хеш-кода всегда должен получаться один и тот же индекс. Однако в отличие от функции вычисления хеш-кода, функция компрессии зависит от размера хеш-таблицы и может меняться вместе с ним.

Одним из преимуществ разделения хеш-функции на две составляющие является то, что вычисление хеш-кода происходит независимо от размера хеш-таблицы, с которой предстоит работать. Это дает возможность разрабатывать функции для вычисления хеш-кодов различных типов данных, не ориентируясь на размер хеш-таблицы, который имеет значе-

ние только для функции компрессии. Это особенно удобно, поскольку размер хеш-таблицы может динамически меняться в зависимости от количества элементов, хранящихся в словаре.

Функция MAD может быть использована как в качестве функции вычисления хеш-кода для целых чисел, так и в качестве функции компрессии для хеш-кодов, созданных с помощью других функций.

Ранее была рассмотрена функция построения хеш-кода, которая основывается на суммировании или операции xor хеш-кодов. Однако она неэффективна при работе с символьными строками и другими объектами разной длины, которые могут быть представлены в виде кортежа $(x_0, x_1, \ldots, x_{n-1})$. В этом кортеже позиция каждого элемента x_i имеет значение. Это связано с тем, что такой подход к созданию хеш-кода может привести к коллизиям для строк (последовательностей) с одинаковым составом элементов.

Возьмем в качестве примера следующие строки: s2 = ["stop", "tops", "pots", "spot"]. При использовании хеш-функции, основанной на суммировании (см. рис. 11.11), для каждой строки получим один и тот же хеш-код: [('stop', 454), ('tops', 454), ('pots', 454), ('spot', 454)].

Такие коллизии можно избежать, если использовать хеш-функцию, которая учитывает положение элементов в массиве входных данных. Такой функцией является ***полиномиальный хеш-код***, который строится с использованием константы a ($a \neq 0$, $a \neq 1$) по формуле:

$$x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-2} a + x_{n-1}.$$

Это полином, где элементы массива входных данных $(x_0, x_1, \ldots, x_{n-1})$ используются в качестве коэффициентов. Для применения этой функции в качестве хеш-функции необходимо добавить функцию компрессии, которая будет приводить полученные значения к требуемому диапазону.

Полиномиальный хеш-код можно эффективно вычислить с помощью схемы Горнера:

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots + a(x_1 + ax_0))\ldots)).$$

Схема Горнера позволяет убрать необходимость в использовании функции компрессии. Вместо этого, можно вычислить хеш-код последовательно, начиная с последнего элемента и перемножая его с константой a, а затем прибавлять следующий элемент. Это позволяет избежать коллизий, так как каждый элемент имеет разное влияние на окончательный хеш-код. Использование полиномиальной хеш-функции позволяет избежать коллизий для строк с одинаковым составом элементов, так как каждый элемент вносит свой вклад в окончательный хеш-код.

11.3. Функция hash в Python

Функция hash в языке Python относится к встроенным функциям и служит для получения хеш-кода объекта. Хеш-код представляет собой целочисленное значение, которое генерируется на основе данных объекта. Такое значение применяется для быстрого сравнения и поиска объектов внутри структур данных, например, в хеш-таблицах.

Встроенная функция hash() в Python принимает один аргумент x и генерирует целочисленное значение, соответствующее хеш-коду данного объекта. Стоит обратить внимание на то, что функция hash может быть использована исключительно для неизменяемых типов данных, таких как числа, строки и кортежи. Это ограничение связано с тем, что хеш-код объекта должен оставаться постоянным на протяжении всего его жизненного цикла. В случае возможности изменения объекта после вычисления его хеш-кода, могут возникнуть непредсказуемые и некорректные результаты при дальнейшем использовании этих хеш-кодов. Неизменность хеш-кода играет ключевую роль для обеспечения корректной работы при использовании хеш-кодов объектов в хеш-таблицах. Примером применения хеш-таблиц в Python является тип данных dict(). В словарях Python ключи подвергаются хешированию для обеспечения быстрого доступа к соответствующим значениям.

```
hash('Hello world!')
```
```
1518382487413275710
```
```
hash(42)
```
```
42
```
```
hash(3.141)
```
```
325123864299130883
```
```
hash((1, 2))
```
```
3713081631934410656
```
```
hash(None)
```
```
8795120784846
```
```
hash(frozenset([1, 2]))
```
```
-1826646154956904602
```
```
# ошибка:
hash([1, 2])
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-80-210e02e9c6e1> in <module>
      1 # ошибка:
----> 2 hash([1, 2])

TypeError: unhashable type: 'list'
```

Рисунок 11.15 — Пример использования функции hash() для вычисления хеш-значений различных объектов

Функция hash является также важным элементом при работе с классами. Она используется для создания уникального идентификатора для объектов, что упрощает поиск и сортировку данных. Однако чтобы функция hash работала корректно, необходимо соблюдать определенные правила.

Одно из ключевых правил заключается в том, что должна быть сохранена *консистентность* между равенством объектов и равенством их хеш-функций. Это означает, что если два объекта равны (x == y), то и их хеш-функции также должны быть равны (hash(x) == hash(y)). Иначе говоря, если два объекта идентичны, их хеш-функции также должны быть идентичными.

Важно помнить, что из-за возможности коллизий у хеш-функций следствие в обратную сторону в общем случае не выполняется. Коллизия в контексте хеш-функций происходит, когда два разных объекта имеют одинаковые хеш-коды. Это может привести к некорректной работе программы, поэтому при реализации функции hash следует учитывать этот фактор.

Сохранение консистентности между равенством объектов и равенством их хеш-функций особенно важно при использовании объекта в качестве ключа в хеш-таблице. В этом случае, если два объекта равны, результат поиска в таблице (который ведется с использованием hash(x), hash(y)) будет идентичен. Таким образом, это правило помогает обеспечить корректность работы хеш-таблицы и избежать возможных ошибок при поиске данных.

```
42 == 42.0
```

```
True
```

```
hash(42), hash(42.0), hash(42.0000001)
```

```
(42, 42, 230584303658)
```

Рисунок 11.16 — Сравнение и хеширование чисел

На рис. 11.16 можно заметить, что хеширование для целого числа 42 и числа с плавающей точкой 42.0 дает одинаковый результат, поскольку они считаются равными в контексте сравнения. Однако значение хеширования для числа с плавающей точкой 42.0000001 отличается и равно 230584303658.

Опишем класс Color, который представляет цвет в формате RGB (рис. 11.17). В методе __init__() производится проверка, что значения r, g и b являются целыми числами и находятся в диапазоне от 0 до 255. Это соответствует формату RGB, где каждый из компонентов цвета может принимать значения от 0 до 255.

Методы с декоратором @property позволяют получить значения отдельных компонентов цвета. Метод __hash__() возвращает хеш-значение для объекта класса Color. Хеш вычисляется как хеш кортежа из трех значений — red, green и blue. Это необходимо для того, чтобы можно было использовать объекты этого класса как ключи в словаре. Наконец, метод __eq__() проверяет равенство двух объектов класса Color. Если значения всех трех компонентов цвета равны у обоих объектов, то они считаются равными.

```
class Color:
    def __init__(self, r, g, b):
        assert type(r) is int
        assert 0 <= r <= 255
        self.__red = r
        assert type(g) is int
        assert 0 <= g <= 255
        self.__green = g
        assert type(b) is int
        assert 0 <= b <= 255
        self.__blue = b

    @property
    def red(self):
        return self.__red

    @property
    def green(self):
        return self.__green

    @property
    def blue(self):
        return self.__blue

    def __hash__(self):
        return hash((self.__red, self.__green, self.__blue))

    def __eq__(self, other):
        return self.__red == other.red and\
    self.__green == other.green and self.__blue == other.blue
```

Рисунок 11.17 — Пример реализации функции hash для пользовательского типа данных

В коде на рис. 11.18 создаются два объекта класса Color с одинаковыми значениями компонентов цвета: 2, 2 и 115. Затем вызываются методы c1.red, c1.green, c1.blue, чтобы вернуть значения этих трех компонентов цвета для объекта. Вызывается функция hash(c1), которая генерирует хеш-значение на основе значений компонентов цвета. Каждый уникальный цвет будет иметь свой уникальный хеш, поэтому это полезный способ проверить, являются ли два цвета одинаковыми.

При сравнении c1 и c2 проверяется, имеют ли эти два объекта одинаковые значения для красного, зеленого и синего. В данном случае они совпадают, поэтому результатом будет True. Аналогично, сравниваются хеш-значения hash(c1) и hash(c2). Они будут равны, если цвета одинаковы, поэтому и результат будет True.

Далее в словарь dc добавляется пара «ключ-значение», где ключом является объект c1 класса Color, а значением — число 110. Наконец, dc[c1] возвращает значение, связанное с ключом c1 в словаре dc, а именно, число 110.

```
c1 = Color(2, 2, 115)
c1.red, c1.green, c1.blue
```

```
(2, 2, 115)
```

```
hash(c1)
```

```
3789703756491517098
```

```
c2 = Color(2, 2, 115)
c1 == c2
```

```
True
```

```
hash(c1) == hash(c2)
```

```
True
```

```
dc = dict()
dc[c1] = 110
```

```
dc[c1]
```

```
110
```

Рисунок 11.18 —Работа с объектами класса Color

11.4. Методы разрешения коллизий

Рассмотрим методы разрешения коллизий — метод цепочек и метод открытой адресации.

Разрешение коллизий при помощи цепочек — это метод, при котором элементы с одинаковым хешем хранятся в связанных списках. В этом случае каждая ячейка j или содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; или если таких элементов нет, то ячейка содержит значение None.

```
def h(k):
    return k % 11
```

```
keys1 = [54, 26, 93, 17, 77, 31, 44, 20, 55]
```

```
[h(k) for k in keys1]
```

```
[10, 4, 5, 6, 0, 9, 0, 9, 0]
```

Рисунок 11.18 — Пример хеш-функции

В примере на рис. 11.18 представлена функция хеширования h(k), которая возвращает остаток от деления ключа k на число 11. Создается список ключей keys1, и затем функция хеширования применяется ко всем ключам в этом списке. Результатом будет список хешей для каждого ключа. Как видно из результата, некоторые ключи хешируются в одну и ту же ячейку (например, ключи 77, 44 и 55 все хешируются в ячейку 0), что демонстрирует возникновение коллизий. Метод цепочек решает данную коллизию, объединяя ключи в связный список внутри ячейки 0. Аналогично, ячейка 9 будет содержать ключи 31 и 20, объединив их в связный список (рис. 11.19).

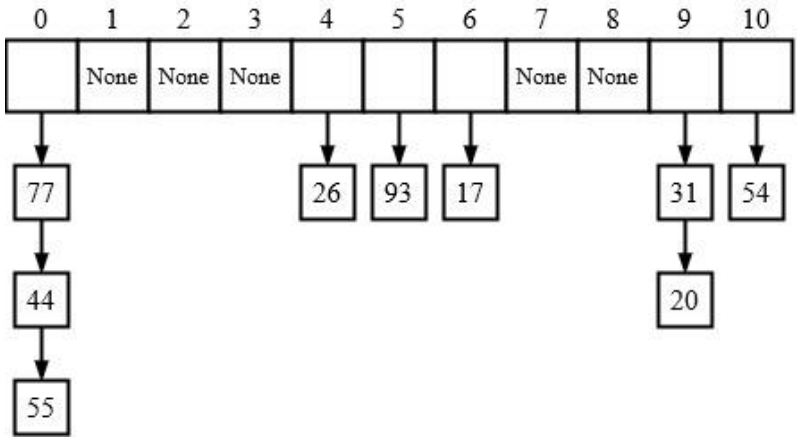


Рисунок 11.19 — Хеш-таблица, использующая цепочки для разрешения коллизий

Время, необходимое для вставки нового элемента в хеш-таблицу в наихудшем случае, равно $O(1)$, что означает, что оно выполняется очень быстро. Это предполагает, что вставляемый элемент отсутствует в таблице. Если нужно убедиться в этом, можно выполнить поиск перед вставкой. Время работы поиска в наихудшем случае пропорционально длине списка (то есть количеству элементов в ячейке). Удаление элемента также может быть выполнено за время $O(1)$, что делает этот метод эффективным для использования в хеш-таблицах.

Метод открытой адресации в хеш-таблицах представляет собой технику, при которой все элементы данных хранятся прямо в самой таблице. Это означает, что каждая запись в таблице либо содержит элемент динамического множества, либо специальное значение, обозначающее пустую ячейку (None или его заменитель). Этот метод отличается от метода цепочек, где элементы могут храниться вне таблицы и связаны между собой списками.

В методе открытой адресации поиск элемента осуществляется путем систематической проверки ячеек таблицы. Процесс продолжается до тех пор, пока не будет найден искомый элемент или пока не будет установлено, что такого элемента в таблице нет. Однако

этот метод имеет ограничение: хеш-таблица может быть полностью заполнена, что делает невозможной вставку новых элементов.

Коэффициент заполнения таблицы (load factor) представляет собой отношение числа элементов в таблице к ее общему размеру. В случае хеш-таблицы с открытой адресацией этот коэффициент не может превышать 1, что означает, что таблица не может быть переполнена.

Одним из преимуществ метода открытой адресации является то, что вместо использования указателей вычисляется последовательность проверяемых ячеек. Это позволяет экономить память и использовать хеш-таблицы большего размера при том же общем количестве памяти. В результате это может привести к меньшему количеству коллизий и более быстрой выборке данных.

Вставка элементов при использовании метода открытой адресации осуществляется путем последовательного исследования ячеек хеш-таблицы до тех пор, пока не будет найдена пустая ячейка. Затем вставляемый ключ помещается в эту ячейку.

Для определения исследуемых ячеек хеш-функция расширяется, включая номер исследования в качестве второго аргумента. Важно, чтобы для каждого ключа k последовательность исследований $h(k,0)$, $h(k,1)$, …, $h(k,m-1)$ представляла собой перестановку множества всех возможных номеров ячеек, чтобы гарантировать, что все ячейки хеш-таблицы могут быть просмотрены.

Алгоритм добавления элемента в хеш-таблицу с открытой адресацией:

1) Инициализируется переменная i со значением 0. Эта переменная будет использоваться для обхода хеш-таблицы в случае коллизий.
2) Начинается цикл, в котором вычисляется значение хеш-функции для ключа k с учетом текущего значения i (h(k, i)). Полученное значение записывается в переменную j.
3) Проверяется, свободна ли ячейка хеш-таблицы с индексом j (T[j] = None). Если да, то в нее записывается ключ k, а индекс j возвращается как результат работы алгоритма.
4) Если ячейка занята, то увеличивается значение i и процесс повторяется с начала цикла.
5) Если все ячейки таблицы просмотрены (i = m), то выдается ошибка «Хеш-таблица переполнена».

Алгоритм поиска элемента в хеш-таблице с открытой адресацией:

1) Инициализируется переменная i со значением 0. Эта переменная будет использоваться для обхода хеш-таблицы.

2) Начинается цикл, в котором вычисляется значение хеш-функции для ключа k с учетом текущего значения i (h(k, i)). Полученное значение записывается в переменную j.
3) Проверяется, содержит ли ячейка хеш-таблицы с индексом j искомый ключ (T[j] = k). Если да, то индекс j возвращается как результат работы алгоритма.
4) Если ячейка не содержит искомый ключ, то увеличивается значение i и процесс повторяется с начала цикла.
5) Если все ячейки таблицы просмотрены (i = m) или встречена пустая ячейка (T[j] = None), то возвращается значение None, указывающее на отсутствие искомого ключа в таблице.

Процедура удаления элемента из хеш-таблицы с открытой адресацией — это процесс, который требует особого внимания и точности. При удалении ключа из определенной ячейки, обозначенной как i, нельзя просто заменить его значением None или «нулевым» значением. Если таким образом поступить, можно сделать невозможным поиск и извлечение ключа k, который был рассмотрен в процессе вставки и оказался в занятой ячейке i.

Одним из возможных решений этой проблемы является пометка таких ячеек специальным значением DELETED или «удалено» вместо None. Это требует небольшой модификации процедуры добавления элемента в хеш-таблицу, которая теперь должна рассматривать такую ячейку как пустую и быть способной вставить в нее новый ключ.

Вместе с тем, процедура поиска элемента в хеш-таблице не требует никаких изменений. Это объясняется тем, что ячейки, помеченные как DELETED, пропускаются при поиске, и исследование следующих ячеек в последовательности продолжается.

Однако стоит отметить, что при использовании специального значения DELETED время поиска перестает зависеть от коэффициента заполнения. Это означает, что время, которое требуется для поиска ключа в хеш-таблице, больше не зависит от того, насколько полностью заполнена таблица. Это может иметь как положительные, так и отрицательные последствия, в зависимости от конкретной ситуации и требований к производительности системы.

Исследование в контексте хеширования — это процесс поиска свободной ячейки в хеш-таблице для размещения элемента в случае возникновения коллизии. Также исследование может быть использовано для поиска конкретного элемента в хеш-таблице. Существуют различные методы исследования, такие как линейное, квадратичное и двойное хеширование, каждый из которых имеет свои преимущества и недостатки.

Линейное исследование — это метод решения коллизий в хеш-таблице, при котором в случае коллизии происходит поиск следующей свободной ячейки путем последовательного просмотра ячеек, начиная с места коллизии.

Рассмотрим обычную хеш-функцию $h' : U \to \{0,1,\ldots,m-1\}$, которую будем называть вспомогательной хеш-функцией. Эта функция преобразует входные данные в числовые значения, которые затем используются для определения местоположения данных в хеш-таблице. Основной принцип работы метода линейного исследования заключается в использовании хеш-функции следующего вида:

$$h(k,i) = (h'(k)+1) \bmod m.$$

Здесь k — это ключ, который ищем, а i — это индекс в последовательности. При этом $\bmod m$ обеспечивает цикличность поиска, то есть когда достигается конец последовательности, поиск продолжается с начала.

Однако у линейного исследования есть одна значительная проблема, которая связана с первичной кластеризацией. Это означает, что возникают длинные последовательности занятых ячеек, что увеличивает среднее время поиска данных. Эти кластеры или группы возникают потому, что вероятность заполнения пустой ячейки, которой предшествуют i заполненных ячеек, равна $(i+1)/m$. Таким образом, длинные серии заполненных ячеек имеют тенденцию к все большему удлинению, что приводит к увеличению среднего времени поиска.

Вернемся к коду на рис. 11.18 и хеш-таблице, представленной на рис. 11.19, и используем метод линейного исследования для разрешения коллизий. В результате получим хеш-таблицу с открытой адресацией и линейным исследованием (рис. 11.20).

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Рисунок 11.20 — Хеш-таблица с открытой адресацией и линейным исследованием

Квадратичное исследование — это другой метод решения коллизий в хеш-таблице, при котором в случае коллизии происходит поиск свободной ячейки путем просмотра ячеек с интервалами, квадратично увеличивающимися (1, 4, 9, 16 и т.д.), начиная с места коллизии.

Квадратичное хеширование применяет хеш-функцию следующего вида:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m.$$

Здесь h' — дополнительная хеш-функция, ненулевые c_1 и c_2 — дополнительные константы, а i принимает значения от 0 до $m-1$ включительно.

Первая исследуемая ячейка — $T[h'(k)]$; остальные исследуемые позиции отстоят от нее на расстояния, которые определяются квадратичной зависимостью от номера исследования i.

Этот метод значительно превосходит линейное исследование, но для того, чтобы исследование покрывало все ячейки, требуется выбор специальных значений c_1, c_2 и m. Также если два ключа имеют одну и ту же начальную позицию исследования, то последовательности исследования в целом будут одинаковыми.

Двойное хеширование — это один из самых эффективных способов использования открытой адресации, так как получаемые при этом перестановки имеют множество свойств случайно выбираемых перестановок. Двойное хеширование применяет хеш-функцию вида:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m,$$

где h_1 и h_2 — дополнительные хеш-функции. Начальное исследование проводится в позиции $T[h_1(k)]$, а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно $h_2(k)$ по модулю m.

В отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа k по двум параметрам — по выбору начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа. Производительность двойного хеширования близка к производительности «идеальной» схемы равномерного хеширования.

11.5. Примеры решения задач

Пример 11.1. Создать класс «Животное» с полями «Вид», «Кличка», «Пол» и «Возраст». Создать хеш-таблицу для хранения объектов класса «Животное» по ключу — номеру чипа.

Решение. Создаем класс «Животное» с полями «Вид», «Кличка», «Пол» и «Возраст» (рис. 11.21).

```
class Animal:
    def __init__(self, species, name, gender, age):
        self.species = species
        self.name = name
        self.gender = gender
        self.age = age

    def __str__(self):
        return f"Кличка: {self.name} ({self.species}), пол: {self.gender}, возраст: {self.age}"
```

Рисунок 11.21 — Класс Animal

Затем создаем класс хеш-таблицы, который используется для хранения объектов класса «Животное». Ключом в этой хеш-таблице является номер чипа животного (рис. 11.22).

Класс HashTable имеет следующие методы:

- __init__(self, size=10) — инициализация хеш-таблицы. По умолчанию размер таблицы равен 10.
- _hash(self, chip_number) — вычисление хеша для номера чипа. Используется строковое представление номера чипа.
- add(self, chip_number, animal) — добавление объекта в хеш-таблицу. Если элемент с таким ключом уже существует, то его значение заменяется на новое.
- remove(self, chip_number) — удаление объекта из хеш-таблицы по номеру чипа.
- get(self, chip_number) — получение объекта из хеш-таблицы по номеру чипа. Если такого элемента нет, то возвращается None.

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, chip_number):
        return hash(str(chip_number)) % self.size

    def add(self, chip_number, animal):
        index = self._hash(chip_number)
        for item in self.table[index]:
            if item[0] == chip_number:
                item[1] = animal
                return
        self.table[index].append([chip_number, animal])

    def remove(self, chip_number):
        index = self._hash(chip_number)
        for i, item in enumerate(self.table[index]):
            if item[0] == chip_number:
                del self.table[index][i]
                return

    def get(self, chip_number):
        index = self._hash(chip_number)
        for item in self.table[index]:
            if item[0] == chip_number:
                return item[1]
        return None
```

Рисунок 11.22 — Класс HashTable

После создания хеш-таблицы и списка объектов класса Animal выполняется добавление этих объектов в хеш-таблицу со случайными номерами чипов. Затем производится получение объектов из хеш-таблицы по номерам чипов и удаление объекта из хеш-таблицы по номеру чипа (рис. 11.23).

```
# создание хеш-таблицы
animals = HashTable()

# создание списка объектов класса Animal
list_of_animals = [Animal("собака", "Рекс", "муж.", 5),
                   Animal("кошка", "Матильда", "жен.", 7),
                   Animal("собака", "Белла", "жен.", 3),
                   Animal("хомяк", "Кузьма", "муж.", 1),
                   Animal("попугай", "Раджа", "муж.", 2)]

# вывод информации о животных
for animal in list_of_animals:
    print(animal)
```

```
Кличка: Рекс (собака), пол: муж., возраст: 5
Кличка: Матильда (кошка), пол: жен., возраст: 7
Кличка: Белла (собака), пол: жен., возраст: 3
Кличка: Кузьма (хомяк), пол: муж., возраст: 1
Кличка: Раджа (попугай), пол: муж., возраст: 2
```

```
import random

# добавление элементов в хеш-таблицу
print('Номера чипов:')
for animal in list_of_animals:
    chip_number = random.randint(100000000, 999999999)
    print(f'{chip_number} — {animal.name}')
    animals.add(chip_number, animal)
```

```
Номера чипов:
754416660 — Рекс
314089471 — Матильда
916339882 — Белла
584617521 — Кузьма
327212941 — Раджа
```

```
# получение объекта из хеш-таблицы по номеру чипа
print(animals.get(754416660))
print(animals.get(584617521))
```

```
Кличка: Рекс (собака), пол: муж., возраст: 5
Кличка: Кузьма (хомяк), пол: муж., возраст: 1
```

```
# удаление объекта из хеш-таблицы по номеру чипа
animals.remove(584617521)

print(animals.get(584617521) if animals.get(584617521) else "Животного с данным номером чипа нет в хеш-таблице")
```

```
Животного с данным номером чипа нет в хеш-таблице
```

Рисунок 11.23 — Работа с хеш-таблицей

Пример 11.2. Написать функцию для нахождения наиболее часто встречающегося значения в хеш-таблице.

Решение. Воспользуемся хеш-таблицей, созданной в примере 11.1. Реализуем функцию most_common_species(hash_table), которая принимает на вход хеш-таблицу и вычисляет наиболее часто встречающееся значение в этой таблице по полю «Вид» (species) (рис. 11.24).

Сначала создается словарь species_count, где ключами являются различные виды животных, а значениями — количество раз, сколько каждый вид встречается в хеш-таблице. Далее происходит перебор всех «слотов» (элементов) хеш-таблицы. Для каждого слота извлекается объект animal. Если вид этого животного уже встречался ранее (то есть он уже присутствует в словаре species_count), то соответствующее значение в словаре увели-

чивается на 1. Если же этот вид встречается впервые, то в словарь добавляется новая пара ключ-значение, где ключ — это вид животного, а значение равно 1.

После того как все элементы хеш-таблицы были обработаны, функция возвращает вид животного, который встречается чаще всего. Это делается с помощью функции max(), которой передается словарь species_count и ключевая функция, которая возвращает значение для каждого ключа. В результате функция max() вернет тот ключ (вид животного), для которого значение (количество раз, сколько этот вид встречается) максимально. После вызова функции most_common_species(animals) выводится сообщение о том, какой вид животного встречается чаще всего в хеш-таблице.

```
# функция для нахождения наиболее часто встречающегося значения в хеш-таблице (по полю species)
def most_common_species(hash_table):
    species_count = {}
    for slot in hash_table.table:
        for _, animal in slot:
            if animal.species in species_count:
                species_count[animal.species] += 1
            else:
                species_count[animal.species] = 1
    return max(species_count, key=species_count.get)
```

```
# нахождение и вывод на экран наиболее часто встречающегося значения в хеш-таблице (по полю species)
print(f'Наиболее часто в хеш-таблице встречается значение "{most_common_species(animals)}"')
```

```
Наиболее часто в хеш-таблице встречается значение "собака"
```

Рисунок 11.24 — Решение задачи

Пример 11.3. Реализуйте хеш-таблицу для хранения информации о пользователях системы. Ключом является логин пользователя, значение — объект, содержащий информацию о пользователе (имя, фамилия, пароль, права доступа и т.д.). Используйте метод разрешения коллизий методом открытой адресации с двойным хешированием и реализуйте возможность добавления, удаления и изменения информации о пользователях.

Решение. Используем класс User для хранения информации о пользователях (рис. 11.25).

```
class User:
    def __init__(self, name, login, password, access_rights):
        self.name = name
        self.login = login
        self.password = password
        self.access_rights = access_rights

    def __str__(self):
        return f'user: {self.name}\nlogin: {self.login}\npassword: {self.password}\naccess_rights: {self.access_rights}'
```

Рисунок 11.25 — Класс User

Класс HashTable используется для создания хеш-таблицы (рис. 11.26).

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.keys = [None] * self.size
        self.values = [None] * self.size

    def hash_function(self, key):
        return hash(key) % self.size

    def double_hash_function(self, key):
        return 1 + (hash(key) % (self.size - 2))

    def add(self, key, value):
        hash_value = self.hash_function(key)
        if self.keys[hash_value] is None:
            self.keys[hash_value] = key
            self.values[hash_value] = value
        elif self.keys[hash_value] == key:
            self.values[hash_value] = value
        else:
            i = 1
            while True:
                new_hash_value = (hash_value + i * self.double_hash_function(key)) % self.size
                if self.keys[new_hash_value] is None:
                    self.keys[new_hash_value] = key
                    self.values[new_hash_value] = value
                    break
                elif self.keys[new_hash_value] == key:
                    self.values[new_hash_value] = value
                    break
                else:
                    i += 1

    def get(self, key):
        hash_value = self.hash_function(key)
        if self.keys[hash_value] == key:
            return self.values[hash_value]
        else:
            i = 1
            while True:
                new_hash_value = (hash_value + *i  self.double_hash_function(key)) % self.size
                if self.keys[new_hash_value] == key:
                    return self.values[new_hash_value]
                elif self.keys[new_hash_value] is None:
                    return None
                else:
                    i += 1

    def remove(self, key):
        hash_value = self.hash_function(key)
        if self.keys[hash_value] == key:
            self.keys[hash_value] = None
            self.values[hash_value] = None
        else:
            i = 1
            while True:
                new_hash_value = (hash_value + i * self.double_hash_function(key)) % self.size
                if self.keys[new_hash_value] == key:
                    self.keys[new_hash_value] = None
                    self.values[new_hash_value] = None
                    break
                elif self.keys[new_hash_value] is None:
                    break
                else:
                    i += 1
```

Рисунок 11.26 — Описание класса HashTable

Данный класс содержит методы для выполнения различных операций в хеш-таблице, таких как добавление, удаление и получение данных. Используется два метода хеширования: основной и дополнительный (двойное хеширование). Основной метод хеширования используется для определения индекса в массиве для хранения ключа и значения. Если по этому индексу уже занят другим ключом (коллизия), то используется второй метод хеширования для определения нового индекса. Этот процесс повторяется до тех пор, пока не будет найден свободный индекс.

Метод add() добавляет пару ключ-значение в таблицу. Если ключ уже существует, значение обновляется. Метод get() возвращает значение по ключу. Если ключа нет, возвращается None. Метод remove() удаляет пару ключ-значение из таблицы по ключу. Если ключа нет, ничего не происходит.

```
# создание хеш-таблицы размера 20
users = HashTable(20)
```

```
# создание списка пользователей
list_of_users = [User("Александр Иванов", "alex_ivanov", "password123", "admin"),
                 User("Екатерина Петрова", "kate_petrova", "secret321", "user"),
                 User("Дмитрий Сидоров", "dima_sidorov", "qwerty", "user"),
                 User("Анна Кузнецова", "anna_kuznetsova", "password", "admin"),
                 User("Илья Шевченко", "ilya_shevchenko", "letmein", "user"),
                 User("Наталья Ковалева", "natalia_kovaleva", "pastry123", "user"),
                 User("Максим Федоров", "max_fedorov", "123456", "user"),
                 User("Ольга Макарова", "olga_makarova", "password123", "admin"),
                 User("Артем Григорьев", "artem_grigorev", "password123", "user"),
                 User("Мария Смирнова", "maria_smirnova", "ilovecats", "user")
                ]
```

```
# добавление объектов класса User в хеш-таблицу
for user in list_of_users:
    users.add(user.login, user)
```

```
# получение информации о пользователе по его логину
user_info = users.get("alex_ivanov")
print(user_info)
```

```
user: Александр Иванов
login: alex_ivanov
password: password123
access_rights: admin
```

```
# изменение прав доступа пользователя
user_info.access_rights = "superadmin"
print(user_info)
```

```
user: Александр Иванов
login: alex_ivanov
password: password123
access_rights: superadmin
```

```
# обновление информации о пользователе в хеш-таблице
users.add(user_info.login, user_info)
```

```
# удаление пользователя из хеш-таблицы
users.remove("maria_smirnova")

print(users.get("maria_smirnova") if users.get("maria_smirnova") else "Такого пользователя нет в хеш-таблице")
```

```
Такого пользователя нет в хеш-таблице
```

Рисунок 11.27 — Работа с хеш-таблицей

11.6. Задачи для самостоятельного решения

Задача 11.1. Решите задачу на создание хеш-таблицы.

1) Создать класс «Студент» с полями «Имя», «Фамилия», «Год рождения» и «Средний балл». Создать хеш-таблицу для хранения объектов класса «Студент» по ключу — номеру зачетной книжки.

2) Создать класс «Товар» с полями «Название», «Цена» и «Количество». Создать хеш-таблицу для хранения объектов класса «Товар» по ключу — артикулу товара.

3) Создать класс «Клиент» с полями «Имя», «Фамилия», «Адрес» и «Номер телефона». Создать хеш-таблицу для хранения объектов класса «Клиент» по ключу — номеру клиентской карты.

4) Создать класс «Заказ» с полями «Номер заказа», «Дата заказа», «Сумма заказа» и «Статус заказа». Создать хеш-таблицу для хранения объектов класса «Заказ» по ключу — номеру заказа.

5) Создать класс «Книга» с полями «Название», «Автор», «Год выпуска» и «Количество страниц». Создать хеш-таблицу для хранения объектов класса «Книга» по ключу — ISBN.

6) Создать класс «Сотрудник» с полями «Имя», «Фамилия», «Должность» и «Зарплата». Создать хеш-таблицу для хранения объектов класса «Сотрудник» по ключу — табельному номеру.

7) Создать класс «Ресторан» с полями «Название», «Адрес», «Телефон» и «Тип кухни». Создать хеш-таблицу для хранения объектов класса «Ресторан» по ключу — названию ресторана.

8) Создать класс «Пользователь» с полями «Имя», «Фамилия», «Логин» и «Пароль». Создать хеш-таблицу для хранения объектов класса «Пользователь» по ключу — логину.

9) Создать класс «Страна» с полями «Название», «Столица», «Население» и «Площадь». Создать хеш-таблицу для хранения объектов класса «Страна» по ключу — названию страны.

10) Создать класс «Фильм» с полями «Название», «Режиссер», «Год выпуска» и «Жанр». Создать хеш-таблицу для хранения объектов класса «Фильм» по ключу — названию фильма.

11) Создать класс «Продукт» с полями «Название», «Цена», «Количество» и «Срок годности». Создать хеш-таблицу для хранения объектов класса «Продукт» по ключу — названию продукта.

12) Создать класс «Адрес» с полями «Улица», «Дом», «Квартира» и «Индекс». Создать хеш-таблицу для хранения объектов класса «Адрес» по ключу — индексу.

13) Создать класс «Курс» с полями «Название», «Преподаватель», «Количество студентов» и «День недели». Создать хеш-таблицу для хранения объектов класса «Курс» по ключу — названию курса.

14) Создать класс «Автомобиль» с полями «Марка», «Модель», «Год выпуска» и «Цвет». Создать хеш-таблицу для хранения объектов класса «Автомобиль» по ключу — номеру государственной регистрации.

15) Создать класс «Квартира» с полями «Адрес», «Количество комнат», «Площадь» и «Цена за квадратный метр». Создать хеш-таблицу для хранения объектов класса «Квартира» по ключу — адресу.

Задача 11.2. Решите задачу на работу с хеш-таблицей.

1) Написать функцию для объединения двух хеш-таблиц.

2) Написать функцию для разделения хеш-таблицы на две, где первая будет содержать элементы с четными ключами, а вторая — с нечетными.

3) Написать функцию для удаления всех элементов из хеш-таблицы, которые не удовлетворяют заданному условию.

4) Написать функцию для удаления всех элементов из хеш-таблицы, которые меньше заданного значения.

5) Написать функцию для нахождения количества элементов из хеш-таблицы, у которых значение больше заданного значения.

6) Написать функцию для удаления всех дубликатов из хеш-таблицы.

7) Написать функцию для проверки, содержит ли хеш-таблица заданный ключ.

8) Написать функцию для сортировки значений в хеш-таблице по возрастанию/убыванию.

9) Написать функцию для нахождения наименьшего значения в хеш-таблице.

10) Написать функцию для нахождения элемента в хеш-таблице, который наиболее близок по значению к заданному числу.

11) Написать функцию для нахождения наиболее часто встречающегося ключа в хеш-таблице.

12) Написать функцию для нахождения среднего значения всех значений в хеш-таблице.

13) Написать функцию для нахождения медианы всех значений в хеш-таблице.

14) Написать функцию для нахождения наиболее часто встречающегося значения в хеш-таблице.

15) Написать функцию для проверки, содержит ли хеш-таблица заданное значение.

Задача 11.3. Решите задачу на создание хеш-таблицы с использованием указанного метода разрешения коллизий.

1) Реализуйте хеш-таблицу для хранения информации о клиентах туристического агентства. Ключом является номер паспорта клиента, значение — объект, содержащий информацию о клиенте (ФИО, контактная информация, список заказанных туров и т.д.). Используйте метод разрешения коллизий методом открытой адресации с линейным пробированием.
2) Реализуйте хеш-таблицу для хранения информации о документах в архиве. Ключом является номер документа, значение — объект, содержащий информацию о документе (название, дата создания, автор и т.д.). Используйте метод разрешения коллизий методом цепочек и реализуйте возможность удаления элементов из таблицы.
3) Реализуйте хеш-таблицу для хранения информации о студентах университета. Ключом является номер студенческого билета, значение — объект, содержащий информацию о студенте (ФИО, группа, оценки и т.д.). Используйте метод разрешения коллизий методом цепочек.
4) Реализуйте хеш-таблицу для хранения информации о сотрудниках компании. Ключом является табельный номер, значение — объект, содержащий информацию о сотруднике (ФИО, должность, зарплата и т.д.). Используйте метод разрешения коллизий методом открытой адресации с двойным хешированием.
5) Реализуйте хеш-таблицу для хранения информации о товарах на складе. Ключом является штрих-код товара, значение — объект, содержащий информацию о товаре (название, количество, цена и т.д.). Используйте метод разрешения коллизий методом цепочек.
6) Реализуйте хеш-таблицу для хранения информации о книгах в библиотеке. Ключом является ISBN книги, значение — объект, содержащий информацию о книге (название, автор, количество экземпляров и т.д.). Используйте метод разрешения коллизий методом открытой адресации с квадратичным пробированием.
7) Реализуйте хеш-таблицу для хранения информации о заказах в интернет-магазине. Ключом является номер заказа, значение — объект, содержащий информацию о заказе (список товаров, стоимость заказа, адрес доставки и т.д.). Используйте метод

разрешения коллизий методом цепочек и реализуйте возможность изменения элементов в таблице.

8) Реализуйте хеш-таблицу для хранения информации о клиентах банка. Ключом является номер счета, значение — объект, содержащий информацию о клиенте (ФИО, адрес, баланс и т.д.). Используйте метод разрешения коллизий методом открытой адресации с линейным пробированием.
9) Реализуйте хеш-таблицу для хранения информации о продуктах в интернет-магазине. Ключом является артикул товара, значение — объект, содержащий информацию о товаре (название, описание, цена и т.д.). Используйте метод разрешения коллизий методом цепочек и реализуйте возможность удаления элементов из таблицы.
10) Реализуйте хеш-таблицу для хранения информации о пациентах в больнице. Ключом является номер медицинской карты, значение — объект, содержащий информацию о пациенте (ФИО, диагноз, лечение и т.д.). Используйте метод разрешения коллизий методом открытой адресации с квадратичным пробированием.
11) Реализуйте хеш-таблицу для хранения информации о заказах в ресторане. Ключом является номер заказа, значение — объект, содержащий информацию о заказе (список блюд, стоимость заказа, адрес доставки и т.д.). Используйте метод разрешения коллизий методом цепочек и реализуйте возможность поиска элементов в таблице.
12) Реализуйте хеш-таблицу для хранения информации о клиентах страховой компании. Ключом является номер полиса, значение — объект, содержащий информацию о клиенте (ФИО, адрес, страховые услуги и т.д.). Используйте метод разрешения коллизий методом открытой адресации с двойным хешированием и реализуйте возможность добавления новых элементов в таблицу.
13) Реализуйте хеш-таблицу для хранения информации о студентах университета. Ключом является номер зачетной книжки, значение — объект, содержащий информацию о студенте (ФИО, факультет, курс и т.д.). Используйте метод разрешения коллизий методом цепочек и реализуйте возможность изменения информации о студентах.
14) Реализуйте хеш-таблицу для хранения информации о товарах на складе. Ключом является штрих-код товара, значение — объект, содержащий информацию о товаре (название, количество, цена и т.д.). Используйте метод разрешения коллизий методом открытой адресации с квадратичным пробированием и реализуйте возможность изменения количества товаров.

15) Реализуйте хеш-таблицу для хранения информации о задачах в проекте. Ключом является номер задачи, значение — объект, содержащий информацию о задаче (описание, статус выполнения, ответственный и т.д.). Используйте метод разрешения коллизий методом цепочек и реализуйте возможность удаления задач из таблицы.

12. Задания для практикума по программированию

12.1. Разработка простых игр

Требуется создать консольное приложение для определенной логической игры с участием одного или более участников. Программа должна обеспечивать интерактивное взаимодействие с пользователем, контролировать правильность хода, отображать текущее состояние игрового поля и определять окончание игры. Все действия игрока осуществляются через консольный ввод. Программа не участвует в игре как противник.

Варианты:

1) Реализовать программу, с которой можно играть в логическую игру «Быки и коровы» (http://робомозг.рф/Articles/BullsAndCowsRules). Программа загадывает число, пользователь вводит очередной вариант отгадываемого числа, программа возвращает количество быков и коров и в случае выигрыша игрока сообщает о победе и завершается. Сама программа НЕ ходит, т.е. не пытается отгадать число загаданное игроком. Взаимодействие с программой производится через консоль, при запросе данных от пользователя программа сообщает, что ожидает от пользователя и проверяет корректность ввода.

2) Реализовать программу, при помощи которой двое пользователей могут играть в «Крестики-нолики» на поле 3 на 3. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде трех текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (в частности, координаты новой отметки на поле) и проверяет корректность ввода. Программа должна уметь автоматически определять, что партия окончена, и сообщать о победе одного из игроков или о ничьей. Сама программа НЕ ходит, т.е. не пытается ставить крестики и нолики с целью заполнить линию.

3) Реализовать программу, при помощи которой двое пользователей могут играть в игру «Супер ним» (https://www.iqfun.ru/articles/super-nim.shtml). Правила игры следующие. На шахматной доске в некоторых клетках случайно разбросаны фишки или пуговицы. Игроки ходят по очереди. За один ход можно снять все фишки с какой-либо горизонтали или вертикали, на которой они есть. Выигрывает тот, кто заберет последние фишки. Взаимодействие с программой производится через кон-

соль. Игровое поле изображается в виде текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (в частности, координаты новой отметки на поле) и проверяет корректность ввода. Программа должна уметь автоматически определять, что партия окончена, и сообщать о победе одного из игроков. Сама программа НЕ ходит, т.е. не пытается выбирать строки или столбцы с целью победить в игре.

4) Реализовать программу, с которой можно играть в игру «19» (http://podelki-fox.ru/igry-dlya-detey-na-bumage-s-chislami/). Правила игры следующие. Нужно выписать подряд числа от 1 до 19: в строчку до 9, а потом начать следующую строку, в каждой клетке по 1 цифре (не числу, см. пример по ссылке). Затем игроку необходимо вычеркнуть парные цифры или дающие в сумме 10. Условие — пары должны находиться рядом или через зачеркнутые цифры по горизонтали или по вертикали. После того как все возможные пары вычеркнуты, оставшиеся цифры переписываются в конец таблицы. Цель — полностью вычеркнуть все цифры. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде трех текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (в частности, координаты очередного хода) и проверяет корректность ввода. Программа должна уметь автоматически определять, что нужно выписать новые строки с цифрами и то, что партия окончена. Сама программа НЕ ходит, т.е. не пытается выбирать пары цифр с целью окончить игру.

5) Реализовать программу, при помощи которой трое пользователей могут играть в «Лоскутное одеяло». Правила игры следующие. На поле, имеющем размер 4 на 5 клеток за один ход каждый игрок должен заполнить одну клетку своим символом. Игрок старается, чтобы его символы были как можно дальше друг от друга. В ходе игры ведется подсчет очков: за каждое соседство клеток с одинаковыми символами игроку, владельцу символа добавляется одно штрафное очко. Соседними считаются клетки, имеющие общую сторону или расположенные наискосок друг от друга. Выигрывает тот, у кого в конце игры меньше всего штрафных очков. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде четырех текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (например, координаты очередного хода) и проверяет корректность

ввода. Программа должна уметь автоматически определять количество штрафных очков и окончание партии и ее победителя. Сама программа НЕ ходит, т.е. не пытается заполнять клетки символами с целью выиграть игру.

6) Реализовать программу, при помощи которой двое пользователей могут играть в «Клондайк» (https://www.iqfun.ru/printable-puzzles/klondike-igra.shtml). Правила игры следующие. Игра ведётся на игровом поле размером 10 на 10 клеток. Игроки по очереди выставляют в любую свободную клетку по отметке, и тот игрок, после чьего хода получилась цепочка длиной хотя бы в три отметки, проигрывает. При этом в цепочке считаются как свои отметки, так и отметки соперника, у игровых фишек как бы нет хозяина. Цепочка — это ряд фишек, следующая фишка в котором примыкает к предыдущей с любого из восьми направлений. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде десяти текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (например, координаты очередного хода) и проверяет корректность ввода. Программа должна уметь автоматически определять окончание партии и ее победителя. Сама программа НЕ ходит, т.е. не пытается ставить в клетки отметки с целью выиграть игру.

7) Реализовать программу, при помощи которой двое пользователей могут играть в «Максит» (https://www.iqfun.ru/articles/maxit.shtml). Правила игры следующие. В клетках квадрата 3 на 3 пишутся случайные числа из диапазона от 1 до 9. Начинающий выбирает любое понравившееся ему число и вычеркивает его, прибавляя к своей сумме. Второй игрок может выбрать любое из оставшихся чисел того столбца, в котором первый игрок делал свой предыдущий ход. Он тоже вычеркивает выбранное число, прибавляя его к своей сумме. Первый игрок далее поступает аналогично, выбирая число-кандидата из той строки, в которой второй игрок ходил перед этим. Может так случиться, что у какого-то игрока не будет хода. Тогда его соперник продолжает игру, делая ход в той же строке (для первого игрока) или в том же столбце (для второго игрока), что и до этого. Игра заканчивается, когда оба играющих не имеют ходов. Результат определяется по набранным суммам, у кого она больше, тот и выиграл. При равенстве сумм фиксируется ничья. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде трех текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает

от пользователя (например, координаты очередного хода) и проверяет корректность ввода. Программа должна уметь автоматически определять сумму очков каждого из игроков и окончание партии и ее победителя. Сама программа НЕ ходит, т.е. не пытается вычеркивать числа с целью выиграть игру.

8) (*) Реализовать программу, при помощи которой двое пользователей могут играть в «Мостики» (https://www.7ya.ru/article/Chem-zanyat-rebenka-13-igr-na-liste-bumagi-so-slovami-kartinkami/). Правила игры следующие. В ходе игры каждый из игроков старается построить мост с одного своего берега на другой по камням, образующим массив 4 на 5 (4 камня вдоль берега игрока и 5 камней между берегами). У первого игрока — крестики в качестве камней и берега крестиков (левый и правый край поля), у второго игрока — нолики и берега ноликов (верхний и нижний край поля). Игру можно начинать в любой точке поля. За один ход игрок может соединить два своих соседних камня вертикальным или горизонтальным мостиком (обозначаются в текстовом режиме символами «-» и «|»). Мосты первого и второго игрока пересекаться не должны. Выигрывает тот, кто построит непрерывный мост с одного своего берега на другой. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде девяти текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (например, координаты очередного хода) и проверяет корректность ввода. Программа должна уметь автоматически определять недопустимые ходы (приводящие к пересечению мостов соперников) и окончание партии и ее победителя. Сама программа НЕ ходит, т.е. не пытается строить мосты с целью выиграть игру.

9) (*) Реализовать программу, с которой можно играть в игру «Морской бой». Программа автоматически случайно расставляет на поле размером 10 на 10 клеток: четыре однопалубных корабля, три двухпалубных корабля, два трехпалубных корабля и один четырехпалубный. Между любыми двумя кораблями по горизонтали и вертикали должна быть как минимум одна незанятая клетка. Программа позволяет игроку ходить, производя выстрелы. Сама программа НЕ ходит, т.е. не пытается топить корабли расставленные игроком. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде десяти текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (в частности, координаты очередного «выстрела») и проверяет корректность ввода. Программа

должна уметь автоматически определять потопление корабля и окончание партии и сообщать об этих событиях.

10) (*) Реализовать программу, с которой можно играть в игру «Пятнашки». Правила игры следующие. Головоломка представляет собой 15 квадратных костяшек с числами от 1 до 15. Все костяшки заключены в квадратную коробку (поле) размером 4 на 4. При размещении костяшек в коробке остается одно пустое место, которое можно использовать для перемещения костяшек внутри коробки. Цель игры — упорядочить размещение чисел в коробке, разместив их по возрастанию слева направо и сверху вниз, начиная с костяшки с номером 1 в левом верхнем углу и заканчивая пустым местом в правом нижнем углу коробки. Взаимодействие с программой производится через консоль. Игровое поле изображается в виде четырех текстовых строк и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (например, координаты очередного хода) и проверяет корректность ввода. Программа должна считать количество сделанных ходов, уметь автоматически определять недопустимые ходы, окончание партии и ее победителя. Сама программа НЕ ходит, т.е. не пытается упорядочить костяшки с целью выиграть игру.

12.2. Текстовый калькулятор

Базовое задание. Необходимо написать функцию, которая будет работать в качестве калькулятора для строковых выражений вида '<число> <операция> <число>'. В этом выражении <число> представляет собой неотрицательное целое число, меньшее 100, записанное словами (например, «тридцать четыре»), а <арифметическая операция> — одна из операций «плюс», «минус», «умножить». Функция должна возвращать результат выполнения операции в виде текстового представления числа. Пример использования функции:

```
calc("двадцать пять плюс тринадцать") -> "тридцать восемь".
```

Дополнительные задания:

1) Реализовать поддержку операции деления и остатка от деления и работу с дробными числами (десятичными дробями). Пример:

```
calc("сорок один и тридцать одна сотая разделить на семнадцать")
                    -> "два и сорок три сотых".
```

Обрабатывать дробную часть до тысячных включительно, если при делении получаются числа с меньшей дробной частью выполнять округление до тысячных.

2) Расширение задания 1. Реализовать поддержку десятичной дробной части до миллионных долей включительно. Реализовать корректный вывод информации о периодической десятичной дроби (период дроби вплоть до четырех десятичных знаков). Пример:

```
calc("девятнадцать и восемьдесят две сотых разделить на девяносто
    девять") -> "ноль и двадцать сотых и ноль два в периоде ".
```

3) Реализовать текстовый калькулятор для выражения из произвольного количества операций с учетом приоритета операций. Пример:

```
calc("пять плюс два умножить на три минус один") -> "десять".
```

Для реализации рекомендуется использовать алгоритмы, основанные на польской инверсной записи (https://ru.wikipedia.org/wiki/Обратная_польская_запись).

4) Расширение задания 3. Добавить поддержку приоритета операций с помощью скобок. Пример:

```
calc("скобка открывается пять плюс два скобка закрывается
      умножить на три минус один") -> "двадцать".
```

5) Добавить возможность использования отрицательных чисел. Пример:

```
calc("пять минус минус один") -> "шесть".
```

6) Добавить возможность оперировать с дробями (правильными и смешанными). Реализовать поддержку сложения, вычитания и умножения, дробей. Результат операций не должен представлять неправильную дробь, такие результаты нужно превращать в смешанные дроби. Пример:

```
calc("один и четыре пятых плюс шесть седьмых ")
    -> "два и двадцать три тридцать пятых".
```

7) Расширение задания 6. Добавить автоматическое сокращение дроби в ответе. Пример:

```
calc("одна шестая умножить на две третьих") -> "одна девятая".
```

8) Расширение задания 1. Добавить операции возведения в степень и тригонометрические операции синус, косинус, тангенс и константу пи. Допускается как минимум одна из этих функций в выражении с обычными операциями. Пример:

```
calc("два в степени четыре") -> "шестнадцать".
```

Пример:

```
calc("синус от пи разделить на четыре")
  -> "ноли и семьсот семь тысячных".
```

9) Добавить комбинаторные операции перестановки, размещения и сочетания. Пример:

```
calc("размещений из трех по два") -> "шесть".
```

10) Диагностировать ошибки: неправильную запись числа; неправильную последовательность чисел и операций; (задание 1) деление на ноль; (задание 3) неправильную последовательность чисел и операций; (задание 4) некорректный баланс и вложенность скобок; (задание 6) некорректную запись числа.

12.3. Реализация пакета модулей по манипулированию табличными данными

Базовое задание. На базе модулей csv, pickle и прямой работы с файлами реализовать следующий базовый функционал:

1. Функции load_table и save_table по загрузке/сохранению табличных данных во внутреннее представление модуля/из внутреннего представления модуля:

- файла формата csv (отдельный модуль с функциями load_table и save_table в рамках общего пакета)
- файла формата pickle (отдельный модуль с функциями load_table и save_table в рамках общего пакета), модуль использует структуру данных для представления таблицы, удобную автору работы.
- текстового файла (только функция save_table, сохраняющая в текстовом файле представление таблицы, аналогичное выводу на печать с помощью функции print_table()).

Примечание: внутреннее представление может базироваться на словаре, где по разным ключам хранятся ключевые «атрибуты» таблицы, а значения таблицы хранятся в виде вложенных списков. Можно выбрать другое внутреннее представление таблицы, в том числе можно реализовать собственный класс для таблицы. При определении API модулей максимально полно использовать возможности сигнатур функций на Python (значения по

умолчанию, запаковка/распаковка, в т.ч. именованных параметров, возвращение множественных значений), интенсивно выполнять проверки и возбуждать исключительные ситуации.

2. Модуль с базовыми операциями над таблицами:

- get_rows_by_number(start, [stop], copy_table=False) — получение таблицы из одной строки или из строк из интервала по номеру строки. Функция либо копирует исходные данные, либо создает новое представление таблицы, работающее с исходным набором данных (copy_table=False), таким образом изменения, внесенные через это представление, будут наблюдаться и в исходной таблице.
- get_rows_by_index(val1, … , copy_table=False) — получение новой таблицы из одной строки или из строк со значениями в первом столбце, совпадающими с переданными аргументами val1, … , valN. Функция либо копирует исходные данные, либо создает новое представление таблицы, работающее с исходным набором данных (copy_table=False), таким образом, изменения, внесенные через это представление, будут наблюдаться и в исходной таблице.
- get_column_types(by_number=True) — получение словаря вида «столбец: тип значений». Тип значения: int, float, bool, str (по умолчанию для всех столбцов). Параметр by_number определяет вид значения столбца – целочисленный индекс столбца или его строковое представление.
- set_column_types(types_dict, by_number=True) — задание словаря вида столбец: тип_значений. Тип значения: int, float, bool, str (по умолчанию для всех столбцов). Параметр by_number определяет вид значения столбца — целочисленный индекс столбца или его строковое представление.
- get_values(column=0) — получение списка значений (типизированных согласно типу столбца) таблицы из столбца либо по номеру столбца (целое число, значение по умолчанию 0), либо по имени столбца.
- get_value(column=0) — аналог функции get_values(column=0) для представления таблицы с одной строкой, возвращает не список, а одно значение (типизированное согласно типу столбца).
- set_values(values, column=0) — задание списка значений values для столбца таблицы (типизированных согласно типу столбца) либо по номеру столбца (целое число, значение по умолчанию 0), либо по имени столбца.

- set_value(value, column=0) — аналог функции set_values(values, column=0) для представления таблицы с одной строкой, устанавливает не список значений, а одно значение (типизированное согласно типу столбца).
- print_table() — вывод таблицы на печать.

3. Для каждой функции должна быть реализована генерация не менее одного вида исключительных ситуаций.

Дополнительные задания:

1) В load_table реализовать load_table(file1, …) — поддержку загрузки таблицы, разбитой на несколько файлов (произвольное количество файлов) (для форматов csv и pickle). В случае несоответствия структуры столбцов файлов вызывать исключительную ситуацию.
2) Расширение задания 1. В save_table реализовать поддержку сохранения таблицы в разбитой на несколько файлов (произвольное количество файлов) по параметру max_rows, определяющему максимальное количество строк в файле. Файлы csv и pickle, полученные с помощью save_table должны быть совместимы с load_table из задания 1.
3) Реализовать функцию concat(table1, table2) и split(row_number) склеивающую две таблицы или разбивающую одну таблицу на две по номеру строки.
4) Реализовать автоматическое определение типа столбцов по хранящимся в таблице значениям. Оформить как отдельную функцию и встроить этот функционал как опцию работы функции load_table.
5) Реализовать поддержку дополнительного типа значений «дата и время» на основе модуля datetime.
6) Добавить набор функций add, sub, mul, div, которые обеспечат выполнение арифметических операций для столбцов типа int, float, bool. Продумать сигнатуру функций и изменения в другие функции, которые позволят удобно выполнять арифметические операции со столбцами и присваивать результаты вычислений. Реализовать реагирование на некорректные значения с помощью генерации исключительных ситуаций.
7) По аналогии с п. 6 реализовать функции eq (==), gr (>), ls (<), ge (>=), le (<=), ne (==), которые возвращают список булевских значений длинной в количество строк срав-

ниваемых столбцов. Реализовать функцию filter_rows(bool_list, copy_table=False) — получение новой таблицы из строк, для которых в bool_list (длинной в количество строк в таблице) находится значение True.

8) Реализовать функцию merge_tables(table1, table2, by_number=True): в результате слияния создается таблица с набором столбцов, соответствующих объединенному набору столбцов исходных таблиц. Соответствие строк ищется либо по их номеру (by_number=True) либо по значению индекса (первый столбец). При выполнении слияния возможно множество конфликтных ситуаций. Автор должен их описать и определить допустимый способ реакции на них (в т.ч. через дополнительные параметры функции и инициацию исключительных ситуаций).

9) Реализовать полноценную поддержку значения None в незаполненных ячейках таблицы. Должно работать при загрузке ячеек с пропусками значений, при операциях, приводящих к появлению пустых ячеек, при работе с get и set операциями.

12.4. Шахматный симулятор

Базовое задание. Реализовать программу, которая позволяет играть в шахматы на компьютере. Взаимодействие с программой производится через консоль (базовый вариант). Игровое поле изображается в виде восьми текстовых строк, плюс строки с буквенным обозначением столбцов (рис. 12.1) и перерисовывается при каждом изменении состояния поля. При запросе данных от пользователя программа сообщает, что ожидает от пользователя (например, позицию фигуры для следующего хода белыми; целевую позицию выбранной фигуры) и проверяет корректность ввода (допускаются только ходы соответствующие правилам шахмат; поддержка рокировки, сложных правил для пешек и проверки мата вынесена в отдельные пункты). Программа должна считать количество сделанных ходов.

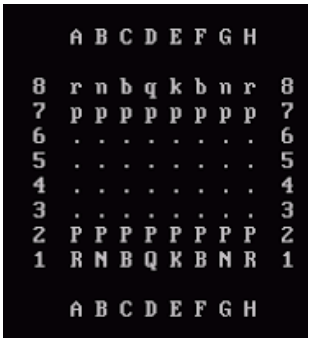


Рисунок 12.1 — Пример изображения шахматного поля в текстовом режиме

Сама программа НЕ ходит, т.е. не пытается выполнить ходы за одну из сторон, а предоставляет поочередно вводить ходы за белых и черных.

Справка о шахматной нотации:

- Общая информация о шахматной нотации записи партий: https://ru.wikipedia.org/wiki/Шахматная_нотация.
- Партии в полной нотации: бесплатная база (для открытия партий нужно зарегистрироваться на ресурсе) записей партий в шахматной нотации (полной): http://www.chessebook.com/openings.php?lan=ru&pa=pa (для получения файлов с записью партий копируйте текст понравившихся партий в текстовый файл, скопированный текст не подвергать дополнительному редактированию и сохранить в файл).
- Партии в сокращенной нотации взять из обсуждений на kasparovchess.crestbook.com, например, из этой ветки: http://kasparovchess.crestbook.com/threads/8210/ (для получения файлов с записью партий копируйте текст понравившихся партий, расположенных справа от блока с доской, в текстовый файл, скопированный текст не подвергать дополнительному редактированию (он во многих нюансах будет отличаться от партий с chessebook.com, так и должно быть) и сохранять файл).

Дополнительные задания:

1) Реализовать чтение записи шахматной партии из выбранного пользователем файла в полной нотации. После чтения должна быть возможность двигаться вперед и назад по записи партии (с соответствующим изменением на поле). Должна быть возможность в выбранной позиции перейти из режима просмотра партии в обычный режим игры. Протестировать не менее чем на 20 реальных партиях с сайта.

2) Реализовать чтение записи шахматной партии из выбранного пользователем файла в сокращенной нотации. После чтения должна быть возможность двигаться вперед и назад по записи партии (с соответствующим изменением на поле). Должна быть возможность в выбранной позиции перейти из режима просмотра партии в обычный режим игры. Протестировать не менее чем на 20 реальных партиях с сайта.

3) Реализовать возможность записи разыгрываемой шахматной партии в текстовый файл в полной (сокращенной, если выполнено задание 2) нотации. Записать партию можно на любом ходу, с историей всей партии с самого начала. Записанная партия должна корректно воспроизводиться в режиме чтения записи партии.

4) Реализовать возможность «отката» ходов. С помощью специальной команды можно возвращаться на ход (или заданное количество ходов) назад вплоть до начала партии.

5) Реализовать функцию подсказки выбора новой позиции фигуры: после выбора фигуры для хода функция визуально на поле показывает поля доступные для хода или фигуры соперника, доступные для взятия, выбранной фигурой.

6) Реализовать функцию подсказки угрожаемых фигур: она возвращает информацию о том, какие фигуры ходящего игрока сейчас находятся под боем (т.е. могут быть взяты соперником на следующий ход) и визуально выделяет их на поле. Функция отдельно указывает на наличие шаха королю.

7) Автоматически определять мат (https://ru.wikipedia.org/wiki/Мат_(шахматы)).

8) Реализовать поддержку выполнения рокировки по всем шахматным правилам (в базовой версии поддержка рокировки не обязательна). Правила рокировки: https://ru.wikipedia.org/wiki/Рокировка.

9) Реализовать поддержку для пешки сложных правил: «взятие на проходе» и замены на других фигуру при достижении крайней горизонтали (в базовой версии их поддержка не обязательна, но возможность первого хода на одну или две горизонтали — обязательно). Подробно: https://ru.wikipedia.org/wiki/Правила_шахмат.

12.5. Шахматный симулятор: объектно-ориентированная версия

Базовое задание. На базе собственной реализации предыдущего задания создать объектно-ориентированную реализацию программы для игры в шахматы. Базовые требования к функциональности программы сохраняются прежними.

Требования к реализации. Основные объекты и абстрактные сущности игры должны быть представлены в виде объектов, представителей соответствующих классов, часть классов должны быть организованы в виде иерархии. В частности: шахматные фигуры — объекты, представители классов, организованных в виде иерархии; доска — объект; ходы фигур — объекты. Вся основная информация должна храниться в атрибутах объектов или классов (например, информация о положении фигур, цвете фигур, символах, используемых для визуализации фигур и т.п.). Основная часть функционала должна программы должна быть организована в виде методов, закрепленных за соответствующими объекта-

ми или классами. Например, это касается методов определяющих допустимые ходы фигур. Организация иерархий классов, атрибутов и методов должна позволять гибко расширять возможности программы с минимальными изменениями в уже созданном коде.

Дополнительные задания:

1) Придумать 3 новых вида фигур с оригинальными правилами перемещения и реализовать их классы. Создать модификацию шахмат с новыми фигурами с минимальным вмешательством в существующий код.

2) На базе игры в шахматы реализовать игру в шашки. Разработать модификацию шахмат с минимальным вмешательством в существующий код.

3) На базе игры в шахматы на классической доске реализовать игру в гексагональные шахматы (https://ru.wikipedia.org/wiki/Гексагональные_шахматы). Выбрать один из трех вариантов: шахматы Глинского; шахматы МакКуэя; шахматы Шафрана. Разработать модификацию шахмат с минимальным вмешательством в существующий код для обычных шахмат.

4) На базе игры в шахматы на классической доске реализовать игру в гексагональные шахматы на троих (https://ru.wikipedia.org/wiki/Шахматы_для_троих). Выбрать один из существующих вариантов. Разработать модификацию шахмат с минимальным вмешательством в существующий код для обычных шахмат.

5) Реализовать возможность «отката» ходов. С помощью специальной команды можно возвращаться на ход (или заданное количество ходов) назад вплоть до начала партии. Информация о ходах в партии должна храниться в объектно-ориентированном виде.

6) Реализовать функцию подсказки выбора новой позиции фигуры: после выбора фигуры для хода функция визуально на поле показывает поля доступные для хода или фигуры соперника, доступные для взятия, выбранной фигурой. Информация о допустимых ходах должна храниться в объектно-ориентированном виде, алгоритм без модификации должен работать при добавлении новых типов фигур (задание берется совместно с заданием 1).

7) Реализовать функцию подсказки угрожаемых фигур: она возвращает информацию о том, какие фигуры ходящего игрока сейчас находятся под боем (т.е. могут быть взяты соперником на следующий ход) и визуально выделяет их на поле. Функция отдельно указывает на наличие шаха королю. Информация о допустимых ходах должна храниться в объектно-ориентированном виде, алгоритм без модификации

должен работать при добавлении новых типов фигур (задание берется совместно с заданием 1).

8) Реализовать поддержку для пешки сложных правил: «взятие на проходе» и замены на другую фигуру при достижении крайней горизонтали (в базовой версии их поддержка не обязательна, но возможность первого хода на одну или две горизонтали — обязательно). Подробно: https://ru.wikipedia.org/wiki/Правила_шахмат. Информация о допустимых ходах должна храниться в объектно-ориентированном виде, алгоритм без модификации должен работать при добавлении новых типов фигур со сложным поведением (задание берется совместно с заданием 1 и как минимум одна из новых фигур должна иметь сложное поведение, т.е. изменение правил хода и взятия фигуры в зависимости от дополнительных условий).

12.6. Реализация пакета модулей для манипулирования плоскими фигурами

Реализовать API, которое позволяет генерировать, преобразовывать и визуализировать последовательность плоских полигонов, представленных в виде картежа картежей (например: ((0,0), (0,1), (1,1), (1,0)) — представление для квадрата). Последовательности представлений полигонов представляют собой итераторы (далее: последовательности полигонов). Решать задачи с использованием функционального стиля программирования, в том числе активно использовать функции из модуля itertools и functools.

1. Реализовать функцию визуализации последовательности полигонов, представленной в виде итератора (например, можно использовать визуализацию с помощью библиотеки matplotlib, см. пример):
https://matplotlib.org/stable/gallery/shapes_and_collections/patch_collection.html#sphx-glr-gallery-shapes-and-collections-patch-collection-py.

2. Реализовать функции, генерирующие бесконечную последовательность непересекающихся полигонов с различающимися координатами (например, «ленту», см. рис. 12.2):

- прямоугольников (gen_rectangle);
- треугольников (gen_triangle);
- правильных шестиугольников (gen_hexagon).
- с помощью данных функций используя функции из модуля itertools сгенерировать семь фигур, включающих как прямоугольники, так и треугольники и шестиугольники, визуализировать результат.

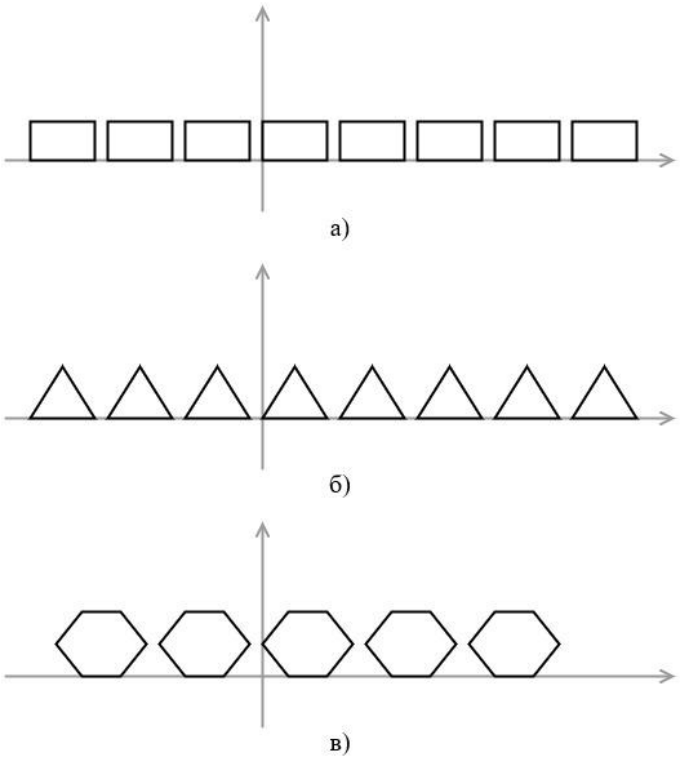


Рисунок 12.2 — Последовательности непересекающихся полигонов

3. Реализовать операции:
   - параллельный перенос (tr_translate);
   - поворот (tr_rotate);
   - симметрия (tr_symmetry);
   - гомотетия (tr_homothety);

которые можно применить к последовательности полигонов с помощью функции map.

4. С помощью данных функций создать и визуализировать (рис. 12.3).:
   - три параллельных «ленты» из последовательностей полигонов, расположенных под острым углом к оси абсцисс;
   - две пересекающихся «ленты» из последовательностей полигонов, пересекающихся не в начале координат;
   - две параллельных ленты треугольников, ориентированных симметрично друг к другу;
   - последовательность четырехугольников в разном масштабе, ограниченных двумя прямыми, пересекающимися в начале координат.

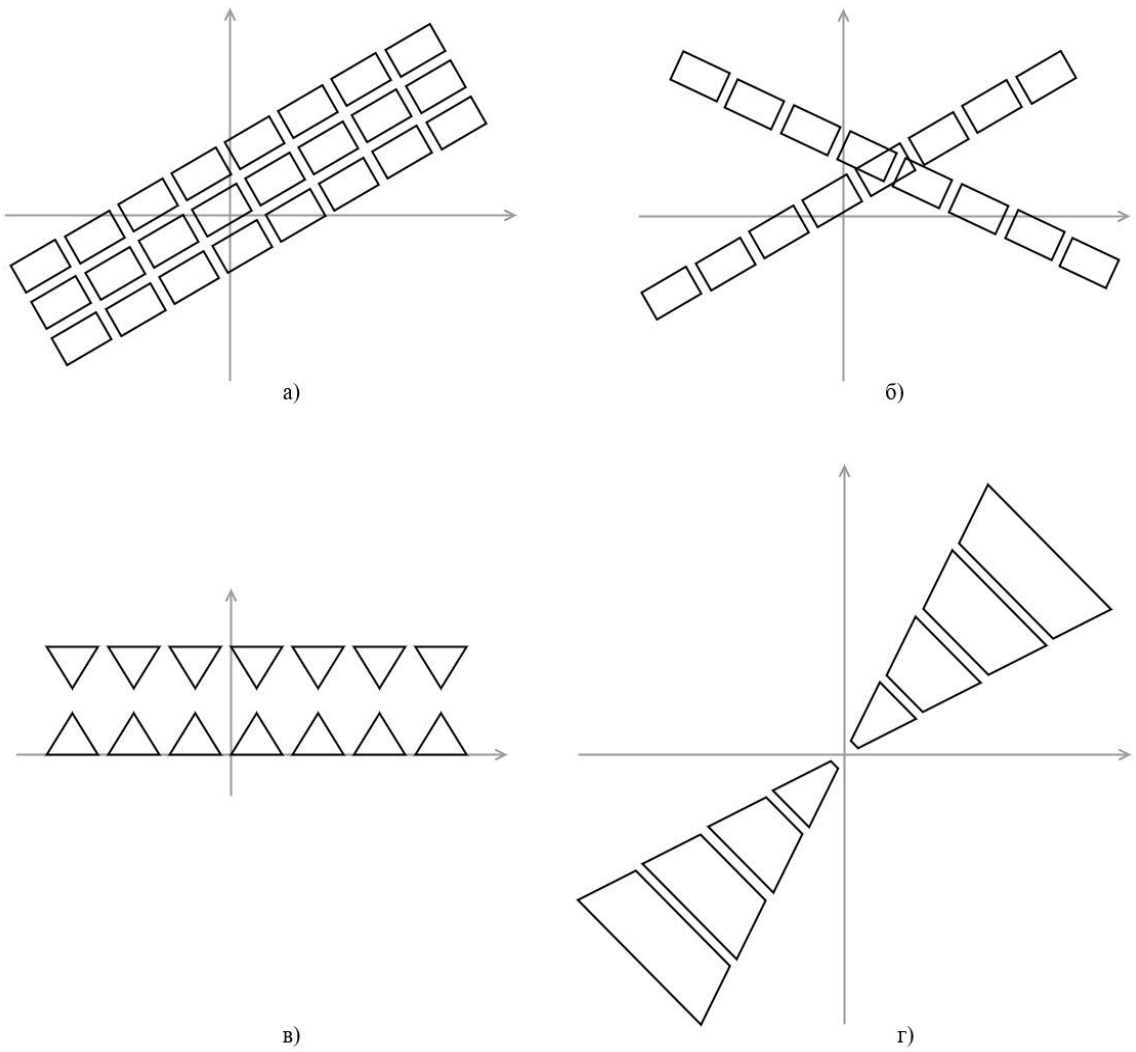


Рисунок 12.3 — Реализация операции параллельного переноса (а), поворота (б), симметрии (в) и гомотетии (г)

5. Реализовать операции:

- фильтрации фигур, являющихся выпуклыми многоугольниками (flt_convex_polygon);
- фильтрации фигур, имеющих хотя бы один угол, совпадающий с заданной точкой (flt_angle_point);
- фильтрации фигур, имеющих площадь меньше заданной (flt_square);
- фильтрации фигур, имеющих кратчайшую сторону меньше заданного значения (flt_short_side);
- фильтрации выпуклых многоугольников, включающих заданную точку (внутри многоугольника) (flt_point_inside);
- фильтрации выпуклых многоугольников, включающих любой из углов заданного многоугольника (flt_polygon_angles_inside);

которые можно применить к последовательности полигонов с помощью функции filter.

6. С помощью данных функций реализовать и визуализировать:
   - фильтрацию фигур, созданных в рамках пункта 4.4; подобрать параметры так, чтобы на выходе было получено 6 фигур;
   - используя функции генерации из п. 2 и операции из п. 3, создать не менее 15 фигур, которые имеют различный масштаб, и выбрать из них (подбором параметра фильтрации) не более четырех фигур, имеющих кратчайшую сторону меньше заданного значения;
   - используя функции генерации из п. 2 и операции из п. 3, создать не менее 15 фигур имеющих множество пересечений и обеспечить фильтрацию пересекающихся фигур.
7. Реализовать декораторы и продемонстрировать корректность их работы:
   - фильтрующие многоугольники в итераторах среди аргументов функции, работающие на основе функций из 5: @flt_convex_polygon, @flt_angle_point, @flt_square, @flt_short_side, @flt_point_inside, @flt_polygon_angles_inside;
   - преобразующие многоугольники в итераторах среди аргументов функции, работающие на основе функций из п. 3: @tr_translate, @tr_rotate, @tr_symmetry, @tr_homothety.
8. Реализовать функции и продемонстрировать их корректность:
   - поиск угла, самого близкого к началу координат (agr_origin_nearest);
   - поиск самого длинной стороны многоугольника (agr_max_side);
   - поиск самой маленькой площади многоугольника (agr_min_area);
   - расчет суммарного периметра (agr_perimeter);
   - расчет суммарной площади (agr_area);

которые можно применить к последовательности полигонов с помощью функции functools.reduce .

9. Реализовать функции и продемонстрировать пример их работы (если возможно, с визуализацией):
   - склейки полигонов в одну последовательность полигонов из нескольких последовательностей полигонов zip_polygons(iterator1, iterator2, [iterator3, …]). Пример: zip_polygons([((1, 1), (2, 2), (3, 1)), ((11, 11), (12, 12), (13, 11))], [((1, –1), (2, –2), (3, –1)), ((11, –11), (12, –12), (13, –11))]) → [((1, 1), (2, 2), (3, 1), (1, –1), (2, –2), (3, –1)), ((11, 11), (12, 12), (13, 11), (11, –11), (12, –12), (13, –11))]. Альтернативный пример (визуализация) на рис. 12.4.

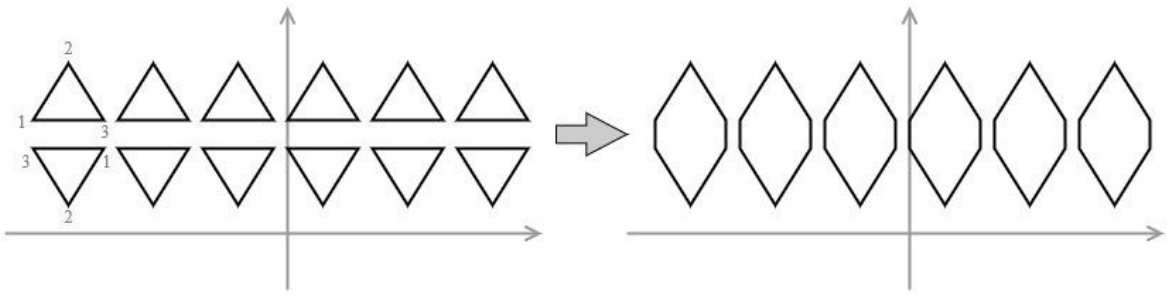


Рисунок 12.4 — Альтернативный пример склейки полигонов

- генерации count_2D() параметры: (start1, start2), [(step1, step2)], результаты: (start1, start2), (start1+step1, start2+step2), (start1+2*step1, start2+2*step2);
- склейки полигонов в одну последовательность полигонов из нескольких последовательностей zip_tuple(iterator1, iterator2). Пример: zip_tuple([(1,1), (2,2), (3,3), (4,4)], [(2,2), (3,3), (4,4), (5,5)], [(3,3), (4,4), (5,5), (6,6)]) → ((1,1), (2,2), (3,3)), ((2,2), (3,3) (4,4)), ((3,3), (4,4), (5,5)), ((5,5), (6,6), (7,7)).

ЛИТЕРАТУРА

1. Программирование на языке Python: Учебно-методическое пособие по дисциплине «Компьютерный практикум» для студентов, обучающихся по направлению подготовки 38.03.05 «Бизнес-информатика» (очная и заочная формы обучения). — М.: Финансовый университет, департамент анализа данных, принятия решений и финансовых технологий, 2018. — 68 с.

2. Программирование на языке Python. Часть 2: Учебное пособие по дисциплинам «Компьютерный практикум» и «Алгоритмы и структуры данных в языке Python» для студентов, обучающихся по направлениям подготовки «Бизнес-информатика» и «Прикладная информатика» профиль «Высокопроизводительные вычисления в цифровой экономике». — М.: Финансовый университет, департамент анализа данных, принятия решений и финансовых технологий, 2019. — 51 с.

3. Саммерфилд М. Программирование на Python 3. Подробное руководство. — Пер. с англ. — СПб.: СимволПлюс, 2009. — 608 с., ил.

4. Доусон М. Программируем на Python. — СПб.: Питер, 2014 -416 с.: ил.

5. Сузи Р.А. Язык программирования Python [Электронный ресурс]: курс / Р.А. Сузи. — 2-е изд., испр. — Москва: Интернет-Университет Информационных Технологий, 2007. — 327 с. — Режим доступа: http://biblioclub.ru/index.php?page=book&id=233288.

6. Северенс Ч. Введение в программирование на Python [Электронный ресурс] / Ч. Северенс. — 2-е изд., испр. — Москва: Национальный Открытый Университет «ИНТУИТ», 2016. — 231 с. [Электронный ресурс]. — Режим доступа: http://biblioclub.ru/index.php?page=book&id=429184.

Учебное издание

Горохова Римма Ивановна
кандидат педагогических наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве Российской Федерации

Догадина Елена Петровна
кандидат технических наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве Российской Федерации

Долгов Виталий Игоревич
кандидат физико-математических наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве Российской Федерации

Макрушин Сергей Вячеславович
кандидат экономических наук, доцент Департамента анализа данных и машинного обучения факультета информационных технологий и анализа больших данных Финансового университета при Правительстве РФ

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON.
УЧЕБНОЕ ПОСОБИЕ

Для студентов,
обучающихся по направлениям
01.03.02 «Прикладная математика и информатика»
09.03.03 «Прикладная информатика»
10.03.01 «Информационная безопасность»,
38.03.05 «Бизнес-информатика»
(программа подготовки бакалавра)

Компьютерный набор, верстка: Е.П. Догадина, Р.И. Горохова, В.И. Долгов, С.В. Макрушин

Вычитка и корректура выполнены авторами

Формат 60x90/16. Гарнитура *Times New Roman.*
Усл. п.л. 20. Изд. № — 2023. Заказ № ____
Электронное издание
