Тема 5. ПРИКЛАДНЫЕ ЗАДАЧИ *PYTHON*

Оглавление

ТЕМА 5. ПРИКЛАДНЫЕ ЗАДАЧИ *PYTHON*
5.1 ФУНКЦИИ

Функции позволяют определять и повторно использовать определенную функциональность в компактной форме. Вызов функции подразумевает передачу ей входных данных, необходимых для выполнения и возвращения результата.

В *Python* функции являются объектами первого класса (англ. *first-class object*, *first-class entity*, *first-class citizen*), обладают свойствами и методами, а также могут быть присвоены переменной, переданы в качестве аргумента, сохранены в структуре данных и возвращены в качестве результата работы другой функции.

5.1.1 Определение и вызов функций

В *Python* функции делятся на встроенные – заранее написанные процедуры преобразования данных – и пользовательские – обособленные блоки кода, создаваемые по правилам языка в ходе написания программы.

Формальное определение функции имеет вид:

```
def имя_функции(параметры):
    '''строка документации'''
    инструкция_1
    инструкция_2
    ...
    инструкция_n
    return результат
```

Функция состоит из двух частей: заголовка и тела. Заголовок включает ключевое слово *def*, имя функции и список параметров в круглых скобках. Концом заголовка является символ ':'. Тело функции состоит из инструкций, необходимых для решения поставленной задачи, включая условные операторы, циклы и вызовы других функций. Ключевое слово *return* используется в том случае, если результат вычислений требуется для дальнейшего использования.

Строка документации содержит описание назначения функции и является не обязательной, но очень хорошей практикой.

Обычно используют утроенные одинарные или двойные кавычки для возможности размещения многострочной документации.

Выполнение инструкций, записанных в теле функции, начинается после ее вызова. Для этого необходимо написать ее имя или присвоить значению переменной:

```
имя_функции (аргументы)
```

или

```
имя_переменной = имя_функции (аргументы)
```

Вызов с присваиванием используется для функции, возвращающей значение.

Замечание. Параметры – переменные, используемые при создании функции. Аргументы – фактические значения (данные), передаваемые функции при вызове.

Функции обмениваются данными посредством передачи аргументов и возвращения значений. На рис. 5.1 изображена схема принципа подобного взаимодействия в коде программы.

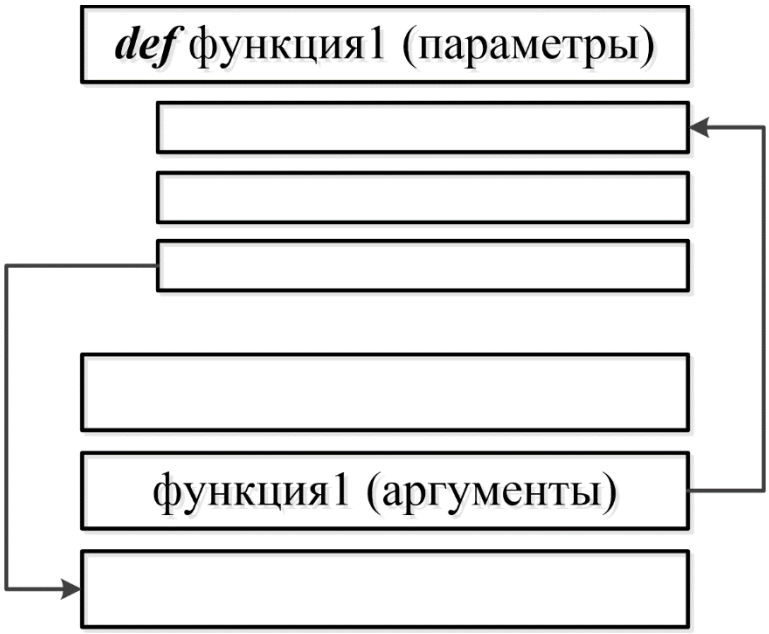

Рис. 5.1. Определение и вызов функции

Пример. Написать функцию, выполняющую приветствие пользователя, имя которого передается при вызове:

```
def user_greeting(person):
    '''
    Функция реализует приветствие
    пользователя, имя которого
    хранится в параметре person.
```

```
    '''
    print('Привет, ' + person + '. Доброе утро!')

name = input('Как Вас зовут? ')
user_greeting(name)
```

В примере присутствует строка документации, получить к ней доступ через атрибут __doc__:

```
print(user_greeting.__doc__)
```

Во время вызова функции происходит связывание переменных-параметров и передаваемых значений-аргументов. Имена параметров и аргументов могут не совпадать: *person* – параметр функции *user_greeting*, а *name* – аргумент с конкретным значением.

5.1.1.1 Параметры и аргументы функции

Язык *Python* позволяет создавать функции с фиксированным и переменным количеством параметров.

В случае *фиксированного числа параметров* их идентификаторы перечисляются через запятую в заголовке функции, а затем используются в ее теле.

Пример 1. Написать функцию, выполняющую приветствие пользователя, имя которого и дополнительное сообщение передаются при вызове.

Код функции:

```
def user_greeting(person, message):
    print(f'Здравствуйте, {person}. {message}')
```

Вызов:

```
name = input('Как Вас зовут? ')
msg = 'Добро пожаловать!'
user_greeting(name, msg)
```

В примере выше сообщение для пользователя не считывалось с клавиатуры, а было задано строковым литералом, присвоенным переменной *msg*.

Если попытаться вызвать функцию с другим количеством параметров, будет выдано сообщение об ошибке *TypeError*:

```
user_greeting() missing 2 required positional
arguments: 'person' and 'message'
```

Создание функции с *переменным количеством параметров* может быть выполнено по средствам определения значений по умолчанию или конструкции произвольного списка параметров.

С помощью оператора присваивания можно выполнить инициализацию параметра уже в процессе создания функции. Такой механизм дает возможность изменять количество передаваемых аргументов при вызове функции за счет заранее определенных значений по умолчанию:

```
def user_greeting(person, message = ' '):
    print(f'Здравствуйте, {person}. {message}')
```

В данном случае у параметра *message* есть значение по умолчанию (пустая строка), поэтому при вызове функции аргумент может отсутствовать. Если он будет указан при вызове, *Python* изменит значение по умолчанию.

Если в функции указан хотя бы один параметр со значением по умолчанию, то все последующие также должны иметь значения по умолчанию. Так, при попытке определить функцию с заголовком вида `def user_greeting(message = ' ', person)`, будет выдана синтаксическая ошибка.

В *Python* есть возможность передавать переменное количество аргументов в функцию с помощью специальных синтаксических конструкций ***args* и ****kwargs*, задающих кортеж и словарь, соответственно. Способ передачи аргументов определяют именно символы * и **, а идентификаторы *args* и *kwargs* могут быть заменены на другие названия.

Пример 2. Написать функцию для определения суммы чисел:

```
def calc_sum(*numbers):
    sum = 0
    for num in numbers:
        sum += num
    return sum
print (calc_sum(1, 2, 4))              # 7
print (calc_sum(15, 45, 2, 88, 8))     # 158
```

Функция *calc_sum* принимает параметр **numbers*, поэтому в качестве аргумента можно передать неопределенное количество значений или набор значений. В теле функции цикл *for* осуществляет перебор, связывая очередное значение из этого набора с переменной *num* и выполняя над ним необходимые действия.

Пример 3. Написать функцию, осуществляющую вывод на экран информации о владельце и его питомцах:

```
def printPetNames(owner, **pets):
    print(f'Владелец: {owner}')
    for pet, name in pets.items():
        print(f'{pet}: {name}')
```

Пример вызова:

```
printPetNames('Артем', Кот = 'Тесла', Птица = 'Кеша',
                    Черепаха = 'Тортилла')
printPetNames('Максим', Кот = 'Пушок')
```

Одним из параметров функции является ***pets*, используемый для создания словаря, в котором содержатся аргументы, полученные при вызове.

В языке *Python* аргументы делятся на два вида: *позиционные* и *именованные* (ключевые).

Во всех рассмотренных выше примерах были продемонстрированы вызовы функции с конкретными значениями или ранее инициализированными переменными, и их связывание с аргументами осуществляется согласно их позиции.

Так, при вызове `printPetNames('Максим', Кот = 'Пушок')` аргументу *owner* присваивается значение 'Максим', а аргументу ***pets* – Кот = 'Пушок'; ***pets* формирует словарь {'Кот': 'Пушок'}. Если при использовании такого способа передачи аргументов изменить их порядок, то будет либо выведено сообщение об ошибке, либо результат выполнения будет неверным.

Использование именованных аргументов позволяет изменять порядок их следования.

Пример 4. Написать функцию, вычисляющую результат одной из арифметических операций (+, -, *, /).

Данная задача была разобрана в разделе «Множественный выбор», преобразуем ранее предложенное решение в функцию:

```
def calc(a, b, operator):
    res = None
    match operator:
        case '+': res = a + b
        case '-': res = a - b
        case '*': res = a * b
        case '/':
            if b != 0:
```

```
                res = a / b
            else:
                res = 'Ошибка! Деление на ноль '
        case _: res = 'Неверный знак '
    return res
```

Все следующие вызовы вышеуказанной функции выполняются без ошибок и возвращают одинаковый результат.

```
# три именованных аргумента
calc(a = 1, b = 8, operator = '+')

# три именованных аргумента (идут не по порядку)
calc(b = 8, a = 1, operator = '+')

# два позиционных и один именованный аргумент
calc(1, 8, operator = '+')
```

В вызове функции можно комбинировать позиционные и именованные аргументы. Однако сначала указываются позиционные аргументы, а затем – именованные.

На этапе создания функции можно указать, какие параметры должны быть позиционными, а какие – именованными. Для этого используются символы / и *:

```
def calc(a, /, b, operator)
```

и

```
def calc(a, b, *, operator)
```

Возможно совместное использование / и * в заголовке функции:

```
def calc(a, /, b, *, operator)
```

Параметры, расположенные слева от /, являются позиционными и могут получать значения *только по позиции*. *Только по имени* параметры получают значение, если они идут справа от символа **.

5.1.1.2 Области видимости переменных

Область видимости переменной (*scope*) представляет собой контекст (часть программы), в пределах которого переменную можно использовать.

Время жизни переменной – период, в течение которого переменная находится в памяти. Время жизни переменной внутри функции длится до тех пор, пока функция выполняется. Пере-

менные «уничтожаются» при выходе из функции, т.е. значения переменных из предыдущих вызовов не запоминаются.

В языке *Python* переменные делятся на глобальные, локальные и нелокальные.

Глобальные переменные создаются вне функций:

```
a = 5
b = 10
def func():
    print('a + b внутри функции:', a + b)

func()
print('a + b вне функции:', a + b)
```

Результат:

```
a + b внутри функции: 15
a + b вне функции: 15
```

Если переменная определена вне функции, то внутри функции можно прочитать ее значение, попытка его изменить вызовет ошибку:

```
a = 5
b = 10
def func():
    print('a + b внутри функции:', a + b)
```

Попытка выполнить приведенный код приведет к ошибке *UnboundLocalError*:

```
local variable 'a' referenced before assignment
```

Исправить ее можно с помощью ключевого слова ***global***:

```
def func():
    global a
    a = a + 3
    print('a + b внутри функции:', a + b)
```

Результат:

```
a + b внутри функции: 18
a + b вне функции: 18
```

Переменная, объявленная внутри тела функции или в локальной области видимости (в цикле или условном операторе), называется *локальной переменной*:

```
def func():
    my_str = 'локальная переменная'
    print('my_str внутри функции:', my_str)
```

```
func()
print('my_str вне функции:', my_str)
```

В приведенном фрагменте кода выполнена попытка получить доступ к локальной переменной из вне. В глобальном контексте переменная *my_str* не существует, поэтому будет сгенерирована ошибка *NameError*.

В программировании часто используют одинаковые имена переменных в разных блоках программы:

```
my_str = 'глобальная переменная'
def func():
    my_str = 'локальная переменная'
    print('Внутри функции:', my_str)

func()
print('Вне функции:', my_str)
```

Результат:

```
Внутри функции: локальная переменная
Вне функции: глобальная переменная
```

В приведенном коде использован один и тот же идентификатор *my_str* как для глобальной переменной, так и для локальной переменной. В процессе интерпретации кода локальная область видимости «перекрывает» глобальную, поэтому в результате на экран будут выведены разные значения переменных *my_str*.

Нелокальные переменные используются во вложенных функциях, локальная область видимости которых не определена:

```
def parent_func():
    my_str = "локальная переменная"

    def child_func():
        nonlocal my_str
        my_str = "нелокальная переменная my_str"
        print("Вложенная функция:", my_str)

    child_func()
    print("Внешняя функция:", my_str)
```

Результат:

```
Вложенная функция: нелокальная переменная my_str
Внешняя функция: нелокальная переменная my_str
```

Ключевое слово ***nonlocal*** прикрепляет идентификатор к переменной из ближайшего окружающего контекста, на глобаль-

ный контекст указанное действие не распространяется. Если изменить значение нелокальной переменной, изменится и значение локальной переменной.

5.1.1.3 Вложенные функции

В языке *Python* тело функции, наряду с условными операторами, циклами и вызовами других встроенных и пользовательских функций, также может содержать и определение последних. В этом случае говорят о вложенных или внутренних функциях:

```
def parent_fun():
    print('Внешняя функция')

    def child_fun():
        print('Внутренняя функция')
    child_fun()

parent_fun()
```

Результат:

```
Внешняя функция
Внутренняя функция
```

Функция *child_fun*() определена внутри *parent_fun*(). Для вызова *child_fun*() сначала нужно вызвать *parent_fun*(). После этого *parent_fun*() начнет выполняться, что приведет к вызову *child_fun*().

Вложенная функция получает свою собственную локальную область видимости и имеет доступ к переменным, объявленным внутри функции, в которую она вложена (функции-родителя):

```
 def parent_fun(num):
    print(f'Внешняя функция: num = {num}')

    def child_fun():
        print(f'Внутренняя функция: num = {num+1}')
    child_fun()

num = int(input('Введите число: '))
parent_fun(num)
```

Результат:

```
Введите число: 15
Внешняя функция: num = 15
Внутренняя функция: num = 16
```

Идентификаторы, определяемые в локальной области внешней функции, являются нелокальными для *child_fun*().

Вложенная функция существует в качестве локальной переменной только внутри внешней, и ее самостоятельный вызов завершится соответствующей ошибкой: *name 'child_fun' is not defined*. Данный факт демонстрирует механизм инкапсуляции – *parent_fun*(*num*) полностью скрывает *child_fun*(), предотвращая доступ из глобальной области.

Поскольку в *Python* функции являются объектами, внешняя функция может вернуть внутреннюю функцию:

```
def hi_bye(choice):

    def greeting(person):
        return f'Привет, {person}.'

    def farewell(person):
        return f'Пока, {person}.'

    if choice==True:
        return greeting
    else:
        return farewell
```

Функция *hi_bye*(*choice*) выбирает одну из двух внутренних функций на основе значения аргумента *choice* и возвращает объект *function*:

```
<function hi_bye.<locals>.greeting at 0x7f48cba4b910>
```

Результат можно получить, присвоив имя функции переменной:

```
name = input('Как Вас зовут? ')
result = hi_bye(True)
print(result(name))
```

или

```
name = input('Как Вас зовут? ')
print(hi_bye(True)(name))
```

Таким образом, функции могут не только принимать поведение через аргументы, но и возвращать его.

5.1.2 Рекурсивные функции

Рекурсия представляет технику в программировании, при которой функция вызывает сама себя для решения подзадачи, являющейся частью исходной задачи. Данный подход делает код более кратким и понятным, но может привести к проблемам с памятью и производительностью при неправильном использовании.

В каждой рекурсивной функции всегда должны быть предусмотрены два случая:

1. Граничный (базовый или тривиальный), при котором функция завершает работу и возвращает данные в основную программу.
2. Рекурсивный, при котором функция продолжает вызывать себя.

В *Python* предусмотрено ограничение глубины рекурсии. Получить текущее предельное значение можно с помощью функции *getrecursionlimit*() модуля *sys*, а изменить его – функцией *setrecursionlimit*(). Максимально возможный предел зависит от платформы.

Пример 1. Найти *n*-й член в последовательности Фибоначчи с помощью рекурсии, если известно, что первый и второй члены равны единице, а каждый последующий – сумме двух предыдущих.

Код, соответствующий решению задачи:

```
def fib_rec(num):
    if num in (1, 2):
        return 1
    return fib_rec(num-1) + fib_rec(num-2)
```

Следует отметить, по условию задачи нумерация чисел Фибоначчи начинается с единицы.

На рис. 5.2 представлена графическая интерпретация рекурсивного вычисления *n*-го члена ряда Фибоначчи при $n = 5$. Пунктирные линии соответствуют возврату значения граничного случая рекурсии.

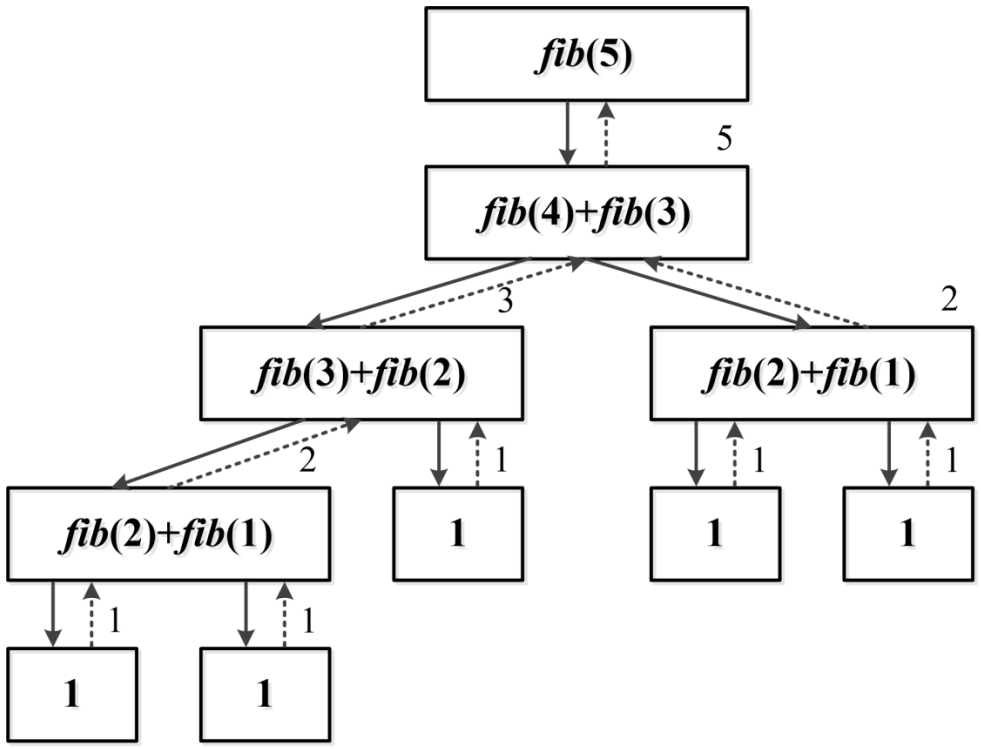

Рис. 5.2. Рекурсивные вызовы при вычислении числа Фибоначчи

Вычисление с помощью рекурсии сопровождается множеством повторных вызовов, и с увеличением числа элементов последовательности количество повторов растет лавинообразно, поэтому данный механизм следует применять только тогда, когда решить задачу иными способами очень сложно.

Пример 2. Сравнить время нахождение n-го члена последовательности Фибоначчи с помощью рекурсии и цикла.

Реализация рекурсивного вычисления остается без изменения.

Решение задачи с помощью цикла в компактном виде:

```
def fib_iter(num):
    fib1 = fib2 = 1
    while num > 2:
        fib1, fib2 = fib2, fib1 + fib2
        num -= 1
    return fib2
```

В приведенном фрагменте *fib*1 = *fib*2 = 1 соответствует групповому присваиванию, а инструкция *fib*1, *fib*2 = *fib*2, *fib*1 + *fib*2 – распаковке кортежа (справа от знака присваивания находится кортеж из двух элементов, слева – переменные, которым будут присвоены по порядку значения элементов кортежа).

Без использования распаковки кортежа решение имеет следующий вид:

```
def fib_iter(num):
    fib1 = fib2 = 1
    i = 0
    while i < num - 2:
        fib = fib1 + fib2
        fib1 = fib2
        fib2 = fib
        i = i + 1
    return fib2
```

В *Python* доступы разные способы для измерения времени выполнения кода, например, можно использовать модуль *time* и функцию *perf_counter_ns*(), возвращающую время в наносекундах:

```
start = time.perf_counter_ns()
print(f'Значение -> {fib_rec(num)}')
finish = time.perf_counter_ns()
print('Время работы рекурсии: ', (finish - start))
```

Переменные *finish* и *start* фиксируют время до начала выполнения фрагмента кода и после его завершения, их разность определяет общую длительность.

Результат для *num* = 15:

```
Значение -> 610
Время работы рекурсии:  136511
Значение -> 610
Время работы цикла:  9909
```

Полученные результаты свидетельствуют о преимуществе итерационного решение по времени. Несмотря на компактность записи кода, рекурсия работает гораздо медленнее

5.1.3 Функциональное программирование

Функциональное программирование является одной из парадигм, поддерживаемых *Python*. Основными предпосылками для полноценного функционального программирования в этом языке являются: функции высших порядков, развитые средства обработки списков, рекурсия, возможность организации ленивых вычислений.

Функциональная программа – совокупность определений функций. Функции содержат вызовы других функций, а также инструкции, которые управляют последовательностью этих вызовов.

Вычисления начинаются с вызова некоторой функции. Она, в свою очередь, тоже вызывает функции, которые входят в её определение в соответствии с внутренней иерархией. Каждый вызов возвращает значение, но помещается оно не в переменную, а в саму функцию, которая этот вызов совершила.

Функциональный дизайн кода может показаться странным и ограниченным, однако у этого стиля программирования есть теоретические и практические преимущества:

– математическое доказательство правильности программы;
– модульность;
– лаконичность;
– параллелизм;
– легкость отладки и тестирования.

Далее будут рассмотрены некоторые конструкции языка, поддерживающие функциональный стиль.

5.1.3.1 Анонимные lambda-функции

Анонимные *lambda*-функции в *Python* являются мощным инструментом, который может сделать ваш код более компактным и читаемым.

Синтаксис объявление имеет вид:

```
lambda аргументы: выражение
```

У анонимной функции может быть любое число аргументов, но лишь одно выражение. Функция вычисляется и возвращается ее результат.

Технология *lambda*-функций позволяет объявлять функции прямо в том месте, где они используются, что упрощает код и повышает его читаемость:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x ** 2, numbers)
```

В *squared_numbers* хранятся результаты вычисления квадратов элементов списка. С использованием ключевого слова *def* код будет менее лаконичным:

```
def square(x):
    return x ** 2
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
```

Анонимные *lambda*-функции особенно полезны в тех случаях, когда нужны безымянные функции (объекты функций) на небольшой промежуток времени. Обычно в *Python* они используются в качестве аргумента для функций более высокого порядка, принимающих в качестве аргументов другие функции (*map*() в приведенных строках кода).

5.1.3.2 Функции высшего порядка

В программировании и математике функциями высшего порядка называются функции, которые выполняют одно или оба из этих действий:

1. Принимают одну и более функций в качестве аргументов.
2. Возвращают функцию в качестве результата.

В Python, функции являются объектами первого класса, что означает, что они могут быть переданы и возвращены, как и любые другие объекты. В *Python* есть встроенные функции, одним из аргументов которых являются другие функции: *map*() и *filter*().

Функция *map*() принимает функцию и итерируемый объект (например, список) и применяет данную функцию ко всем элементам итерируемого объекта. В результате получаем объект-итератор, который содержит результаты применения функции:

```
def square(x):
    return x * x
numbers = [1, 2, 3, 4, 5]
squares = map(square, numbers)
print(list(squares))          # [1, 4, 9, 16, 25]
```

Функция *filter*() принимает функцию-предикат и итерируемый объект, проходит по каждому элементу итерируемого объекта и возвращает новый итератор, состоящий только из тех элементов, для которых функция-предикат вернула *True*:

```
def is_even(x):
```

```
    return x % 2 == 0
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(is_even, numbers)
print(list(even_numbers))      # [2, 4, 6]
```

В общем случае функции высшего порядка позволяют создавать абстракции, которые без них были бы очень громоздки или вообще невозможны.

Библиотека *functools* содержит дополнительные функции высших порядков: *reduce*(), *partial*(), *cmp_to_key*() и *update_wrapper*().

С помощью *reduce*() запускается цепь вычислений, в которой функция применяется ко всем элементам последовательности.

Функция *partial*() служит для частичного назначения аргументов.

Результатов *cmp_to_key*() является ключевая функция для компарации объектов.

Функция *update_wrapper*() используется для обновления метаданных функции-обертки, данными из некоторых атрибутов оборачиваемой функции. Обеспечивает лучшую читаемость и возможность повторного использования кода.

5.1.3.3 Замыкания

В основе концепции замыкания лежит возможность сохранять значения и состояния между вызовами функций.

Замыкания реализуются через функцию, которая запоминает значения из своей внешней области видимости, даже если эта область уже недоступна:

```
def parent_func(x):
    def child_func(y):
        return x + y
    return child_func
closure = parent_func(45)
print(closure(5))  #  50
```

В этом примере создана функция *parent_func* , которая принимает параметр *x* и возвращает внутреннюю функцию *child_func*. Внутренняя функция также принимает параметр *y* и возвращает сумму *x* и *y*.

Затем создается переменная *closure*, которой присваивается результат вызова *parent_func* с аргументом 45. По завершению данной инструкции переменная ссылается на *child_func* и хранит значение *x* как 45.

В заключении *closure* вызывается с аргументом 5, и программа завершается выводом числа 50. Замыкание *closure* сохраняет значение *x* как 10 между вызовами, поэтому оно может быть использовано внутри *child_fun* даже после того, как *parent_func* уже завершила свою работу.

На практике замыкания используются для инкапсуляции кода и скрытия важных данных. С помощью замыканий также можно избежать использования глобальных переменных.

Важно отметить, не каждая вложенная функция автоматически выступает в качестве замыкания. Если в ней нет ссылок на какие-то переменные из функции высшего порядка, они не будут сохранены, и в этом случае вложенная функция считается просто вложенной функцией, а не замыканием:

```
def upp_level():
    msg1 = 'Привет'
    msg2 = 'Учишь Python?'
    def low_level():
        print(msg1)
        print(msg2)
    return low_level
my_function = upp_level()
print(my_function())
```

Во вложенной функции отсутствует сохранение значений переменных, попытка вызвать замыкание и получить значение завершится *None*.

5.1.3.4 Декораторы

Декораторы в *Python* представляют функции, принимающие другую функцию в качестве параметра, добавляют к ней некоторую дополнительную функциональность и возвращают функцию с измененным поведением.

Декораторы используются для изменения работы существующих функций или классов, добавления новых возможностей и обеспечение безопасности.

Следующий фрагмент кода содержит определение функции, вычисляющей произведение переданных ей чисел:

```
def mul_numbers(x1, x2):
    return x1*x2
```

Можно создать декоратор, который добавит к этой функции функциональность для отладки:

```
def debug_decorator(func):
    def wrapper(*args, **kwargs):
        print("Вызов функции:", func.__name__)
        print("Аргументы:", args, kwargs)
        result = func(*args, **kwargs)
        print("Результат:", result)
        return result
    return wrapper
```

Этот декоратор принимает функцию в качестве аргумента и возвращает новую функцию-обертку, которая добавляет отладочные сообщения в процесс выполнения исходной функции. Для применения декоратора к функции *mul_numbers* нужно его к ней «привязать»:

```
@debug_decorator
def mul_numbers(x1, x2):
    return x1*x2
```

Вызов функции:

```
mul_numbers(2, 3)
```

Результат:

```
Вызов функции: mul_numbers
Аргументы: (2, 3) {}
Результат: 6
```

Следует отметить, декораторы могут использовать замыкания, чтобы сохранять состояния между вызовами функции. Например, декоратор может создать замыкание, которое сохраняет информацию о том, сколько раз функция была вызвана, и возвращать это значение при каждом вызове функции.

5.2 МОДУЛИ И ПАКЕТЫ

5.2.1 Подключение модуля из стандартной библиотеки

Для подключения используется ключевое слово ***import***, после которого указывается имя нужного модуля. Одной инструкцией можно подключить несколько модулей, однако так делать не рекомендуется из-за снижения читаемости кода:

```
import os
print(os.getcwd())
```

В приведенном фрагменте выполняется подключение модуль *os* для получения текущей директории. После импортирования модуля его название становится переменной, и с ее помощью можно получить доступ к атрибутам модуля.

Стоит отметить, что если указанный атрибут модуля не будет найден, возникнет исключение *AttributeError*. А если не удастся найти модуль для импортирования, то *ImportError*.

Если название модуля слишком длинное, для него можно создать псевдоним, с помощью ключевого слова ***as***:

```
import math as m
print(m.pi)      # 3.141592653589793
```

Теперь доступ ко всем атрибутам модуля *math* осуществляется только с помощью переменной *m*, а переменной *math* в этой программе уже не будет.

Подключить определенные атрибуты модуля можно с помощью инструкции *from*. Она имеет несколько форматов:

```
from <Название модуля> import
     <Атрибут 1> [ as <Псевдоним 1> ],
     [<Атрибут 2> [ as <Псевдоним 2> ] ...]
from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова *as*:

```
from math import e, ceil as c
print(e)         # 2.718281828459045
print(c(4.6))    # 5
```

Импортируемые атрибуты можно разместить на нескольких строках для лучшей читаемости кода.

Второй формат инструкции *from* позволяет подключить все (точнее, почти все) переменные из модуля. Если в модуле определена переменная *__all__* (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. В противном случае будут подключены атрибуты, идентификаторы которых не начинаются с нижнего подчеркивания.

5.2.2 Создание собственного модуля

Для формирования модуля необходимо создать обычный текстовый файл с расширением **.py* и записать в него целевые программные инструкции. Название файла при этом будет представлять имя модуля, а сам модуль после создания станет доступным для использования либо в качестве независимого сценария, либо в виде расширения, подключаемого к другим модулям, позволяя тем самым связывать отдельные файлы в крупные программные системы. Все имена, которым будет выполнено присваивание на верхнем уровне модуля (т.е. внутри файла модуля вне функций, классов и т.д.), становятся атрибутами объекта модуля.

Собственные и библиотечные модули подключаются одинаково с помощью *import*. По умолчанию интерпретатор *Python* ищет модули по ряду стандартных путей, один из которых – директория главного, запускаемого скрипта.

Для обращения к функциональности модуля необходимо получить его пространство имен (по умолчанию совпадать с именем модуля) по схеме `пространство_имен.функция`.

Другой вариант настройки предполагает импорт функциональности модуля в глобальное пространство имен текущего модуля с помощью ключевого слова *from*, работа с которым рассмотрена в предыдущем разделе.

Способы подключения собственного модуля показаны на рис. 5.3.

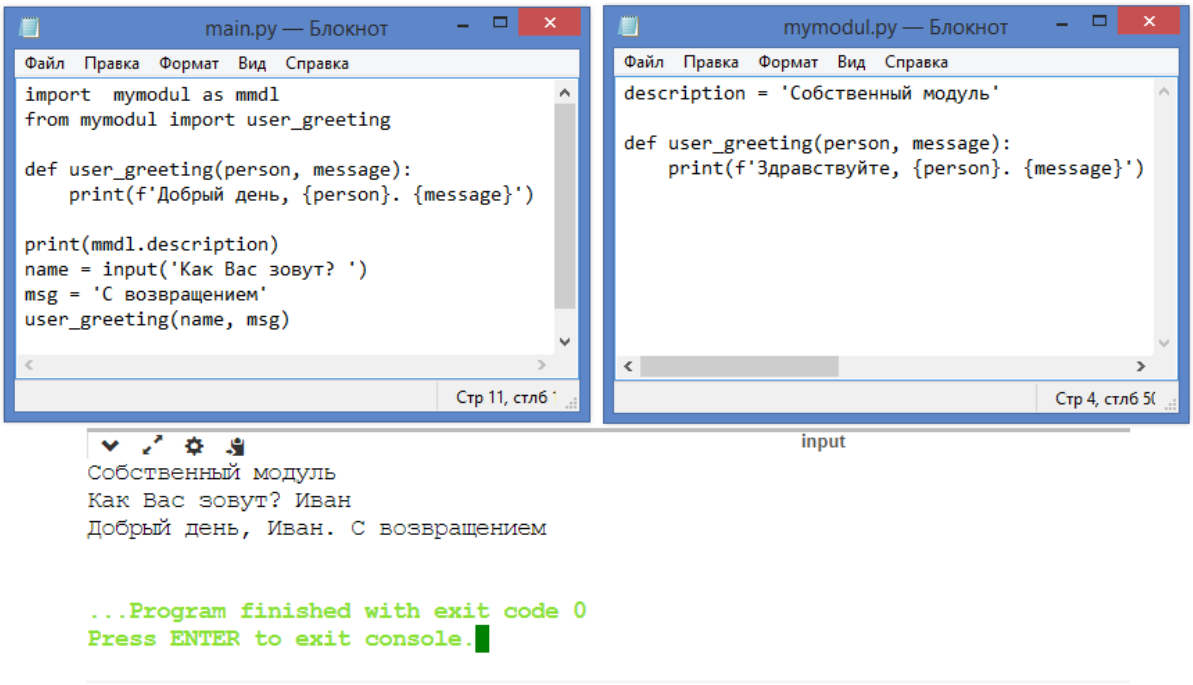

Рис. 5.3. Взаимодействие нескольких модулей кода

Стоит отметить, импорт в глобальное пространство имен чреват коллизиями имен функций. Если определены функции с одинаковыми именами, то будет вызываться функция, которая определена последней. Таким образом, одноименная функция текущего файла скрывает функцию из подключенного модуля.

5.2.3 *Python* в научных вычислениях

Для проведения вычислений с действительными числами в *Python* предусмотрена стандартная библиотека (модуль) *math*. Однако для проведения научных расчетов, включающих обработку, анализ и визуализацию данных предоставляемого ею функционала недостаточно.

Достаточно часто применению *Python* в научных вычислениях посвящены отдельные учебные курсы, рассчитанные на студентов, обладающих основными навыками в программировании на данном языке.

Представленная в данном разделе информация может рассматриваться в виде аннотаций к часто используемым специализированным модулям.

5.2.3.1 Библиотека NumPy

NumPy (*Numerical Python*) – библиотека с открытым исходным кодом для языка программирования *Python*. Она содержит многомерные массивы и матричные структуры данных и активно используется в следующих областях:

- научные и инженерные вычисления;
- машинное обучение и искусственный интеллект;
- обработка данных;
- визуализация данных;
- высокопроизводительные вычисления.

Для установки библиотеки необходимо в терминале запустить на выполнение команду *pip3 install numpy*.

Массивы *NumPy* быстрее и компактнее, чем классические списки *Python*. Массив потребляет меньше памяти (что очень важно для оптимизации скорости вычислений или используемой мощности) и удобен в использовании. *NumPy* использует гораздо меньше памяти для хранения данных и предоставляет механизм указания типов данных, что позволяет дополнительно оптимизировать код.

Основным объектом библиотеки является однородный многомерный массив *ndarray*, наиболее важные атрибуты которого представлены в табл. 5.1.

Таблица 5.1

Атрибуты объекта *ndarray*

Название атрибута	Описание
1	2
ndim	Число измерений (чаще их называют «оси») массива
shape	Размеры массива и его форма. Является кортежем натуральных чисел, показывающим длину массива по каждой оси. Для матрицы из *n* строк и *m* столбов, *shape* будет (*n*, *m*). Число элементов кортежа *shape* равно *ndim*
size	Количество элементов массива. Очевидно, равно произведению всех элементов атрибута *shape*
dtype	Объект, описывающий тип элементов массива. Можно определить *dtype*, используя стандартные типы данных *Python* или *NumPy*, а также есть возможность определить собственные типы данных, в том числе и составные

Продолжение табл. 5.1

1	2
itemsize	Размер каждого элемента массива в байтах
data	Буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов

Для создания массива используется метод array():

```
a1 = np.array([1,2,3,4], 'float64')
print(a1)
a2 = np.array([[1, 2], [3, 4], [5, 6]])
print(a2)
print(a2.size) # 6
print(a2.shape) # (3, 2)
```

В результате выполнения приведенного фрагмента кода переменная *a*1 будет связана с массивом вещественных чисел, в *a*2 будет храниться двумерный массив целых чисел (по умолчанию генерируются целые числа).

Доступ к элементам массива осуществляется по целочисленным индексами (нумерация ведется с нуля):

```
a = np.array([[1, 2, 8], [3, 4, 5]])
print(a[1][1]) # 4
```

При использовании неполного набора индексов недостающие компоненты неявно заменяются списком всех возможных индексов вдоль соответствующей оси. С этой целью используется символ ':':

```
a = np.array([[11, -2, -8], [3, -4, 5], [1, 2, 5]])
print(a[1, :]) # [ 3 -4  5]
```

«Пропустить» индекс можно вдоль любой оси или осей. Если после «пропущенной» последуют оси с индексацией, то символ ':' является обязательным:

```
a = np.array([[11, -2, -8], [3, -4, 5], [1, 2, 5]])
print(a[:, 1]) # [-2 -4  2]
```

Класс *ndarray* обладает большим количеством методов, с полным списком которых можно ознакомиться с помощью официальной документации библиотеки.

5.2.3.2 Библиотека Pandas

Ключевым инструментом для анализа данных в *Python* является библиотека *Pandas*. Она построена на базе *NumPy* и позволяет работать с данными, представленными в табличной форме, а также временными рядами.

Для установки библиотеки необходимо в терминале запустить на выполнение команду *pip3 install pandas*.

Общепринятым сокращением для *Pandas* в коде является *pd*.

Данные в *Pandas* представлены в двух видах: *Series* и *DataFrame*.

Series – объект, который похож на одномерный массив и может содержать любые типы данных. Проще всего представить его как столбец таблицы с последовательностью каких-либо значений, у каждого из которых есть индекс – номер строки:

```
import pandas as pd
series_example = pd.Series([4, 7, -5, 3])
```

Результат вывода информации:

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Series отображается в виде таблицы с индексами элементов в первом столбце и значениями во втором.

DataFrame – основной тип данных в *Pandas*, вокруг которого строится вся работа. Его можно представить в виде обычной таблицы с любым количеством столбцов и строк. Внутри ячеек такой «таблицы» могут быть данные самого разного типа: числовые, булевы, строковые и так далее.

У *DataFrame* есть и индексы строк, и индексы столбцов. Это позволяет удобно сортировать и фильтровать данные, а также быстро находить нужные ячейки:

```
import pandas as pd
city = {'Город': ['Москва', 'Санкт-Петербург',
         'Новосибирск', 'Екатеринбург'],
         'Год основания': [1147, 1703, 1893, 1723],
         'Население': [11.9, 4.9, 1.5, 1.4]}
df = pd.DataFrame(city)
```

Результат вывода информации:

	Город	Год основания	Население
0	Москва	1147	11.9
1	Санкт-Петербург	1703	4.9
2	Новосибирск	1893	1.5
3	Екатеринбург	1723	1.4

Переменная *city* инициализируется словарем с нужной информацией о городах. Преобразование словаря в *DataFrame* выполняется одноименным методом библиотеки *Pandas*.

Pandas позволяет импортировать данные разными способами. Например, прочесть их из словаря, списка или кортежа. Однако чаще всего данные считываются из *.csv* файлов, используемых в анализе данных.

5.2.3.3 Библиотека SciPy

SciPy – библиотека *Python* с открытым исходным кодом, предназначенная для решения научных и математических проблем.

Установка библиотеки выполняется с помощью команды *pip3 install scipy*.

Она построена на базе *NumPy* и позволяет управлять данными, а также визуализировать их с помощью разных высокоуровневых команд. Если будет установлена *SciPy*, то *NumPy* отдельно импортировать не нужно.

Данная библиотека расширяет возможности *NumPy*. Некоторые частые действия в ней реализованы как отдельные функции, поэтому библиотека сильно упрощает работу со сложными задачами с использованием продвинутой математики. Однако для более простых задач она избыточна.

Возможности библиотеки распределены по нескольким модулям, или пакетам, которые объединены назначением. Каждый из них необходимо импортировать отдельно. Описание основных модулей *SciPy* представлено в табл. 5.2

Таблица 5.2

Модули библиотеки *SciPy*

Название модуля	Описание
scipy.special	Специальные функции, возможности и понятия из математической физики
scipy.integrate	Функции для численного интегрирования и решения обыкновенных дифференциальных уравнений
scipy.optimize	Алгоритмы оптимизации (в т.ч. минимизации математических функций)
scipy.interpolate	методы для интерполяции, т.е. для приближенного нахождения какой-либо величины по уже известным отдельным ее значениям
scipy.fft	Преобразования Фурье (операции, сопоставляющие одной функции вещественной переменной другую функцию вещественной переменной)
scipy.signal	Методы для обработки и преобразования сигналов
scipy.linalg	Операции линейной алгебры
scipy.sparse.csgraph	Методы для работы с разреженными графами, особыми структурами данных
scipy.spatial	Функции для работы с пространственными структурами данных и алгоритмами
scipy. stats	Операции для статистических расчетов
scipy.ndimage	Обработка многомерных изображений
scipy.io	Ввод и вывод, загрузка и сохранение файлов

Пример. Вычислить интеграл $\int_{x=1}^{x=2}\int_{y=2}^{y=3}\int_{z=0}^{z=1} xyz\ dxdydz$:

```
from scipy import integrate

f = lambda z, y, x: x*y*z
print(integrate.tplquad(f, 1, 2, 2, 3, 0, 1))
```

Результат:

```
(1.8750000000000002, 3.324644794257407e-14)
```

Метод *tplquad*() принимает подынтегральную функцию и пределы интегрирования и выдает кортеж из двух чисел – значения интеграла и величины абсолютной ошибки.

Одним из применений тройного интеграла в физике является расчет массы неоднородного объекта.

5.2.3.4 Библиотека Matplotlib

Библиотека *Matplotlib* предназначена для создания статических, анимированных и интерактивных визуализаций на *Python* и может быть использована в операционных системах *Windows*, *macOS* и *Linux*.

При работе с *Google Colab* или *Jupyter Notebook* устанавливать *Matplotlib* не понадобится. В случае работы с другой *IDE*, например в *Visual Studio Code*, необходимо установить библиотеку через терминал с помощью команды *pip3 install matplotlib*.

Сокращение *plt* для библиотеки является общепринятым и используется в официальной документации.

Matplotlib позволяет строить двумерные визуализации любых типов (линейные, круговые и столбчатые диаграммы, гистограммы, поля градиентов и т.д.), а также поддерживает основные типы трехмерных графиков.

Пример 1. Построить график функции $y = 4 + 2\sin(2x)$:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = 4 + 2*np.sin(2*x)

fig, ax = plt.subplots()
fig = fig.canvas.set_window_title('График 1')
ax.plot(x, y, linewidth = 2.0)
ax.set(xlim = (0, 10), xticks = np.arange(1, 10),
       ylim = (0, 8), yticks = np.arange(1, 8))

plt.grid()#Отображение сетки
plt.xlabel('Ось x') #Подпись для оси x
plt.ylabel('Ось y') #Подпись для оси y
plt.show()
```

Код предлагаемого решения условно можно разделить на три части: работу с данными, настройку области будущего графика и его отображения.

Значения аргумента *x* сгенерированы в виде последовательности данных, равномерно расположенных на числовой прямой в заданном интервале, с помощью метода *linspace*() из библиотеки *NumPy*. Для вычисления значения синуса угла также использует-

ся метод из этой библиотеки, который выполняется быстрее по сравнению с аналогом из встроенного модуля *math*.

Инструкция *fig, ax = plt.subplots*() – метод создания графика, объединяющий инициализацию области для рисунка (*fig*) и осей (*ax*), на которых впоследствии будут отображаться данные. Данный подход значительно упрощает подготовительный этап к визуализации данных, позволяя основное внимание уделить представлению информации.

Метод *plot*() используется для построения и оформления двумерных графиков

Вывод графика функции осуществляется методом *show*() библиотеки *Matplotlib*. Дополнительно отображаются сетка и подписи осей.

Результат визуализации представлен на рис. 5.4.

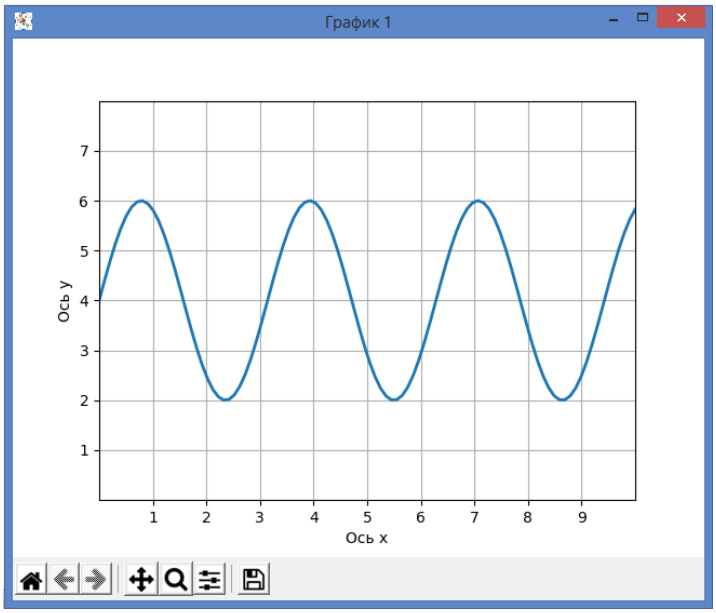

Рис. 5.4. Результат визуализации графика функции

Пример 2. Построить гистограмму случайной выборки из нормального (гауссовского) распределения:

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)
x = 4 + np.random.normal(0, 1.5, 200)

fig, ax = plt.subplots()
fig = fig.canvas.set_window_title('График 2')
```

```
ax.hist(x, bins=8, facecolor='green', linewidth=0.5,
        edgecolor='white')
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
        ylim=(0, 56), yticks=np.linspace(0, 56, 9))

plt.xlabel('Интервалы')
plt.ylabel('Частота')
plt.show()
```

Метод *random.seed*() позволяет изменить число, передаваемое генерации случайного числа.

Случайная выборка из нормального распределения получена с помощью метода *np.random.normal*(0, 1.5, 200) библиотеки *NumPy*, где первый аргумент соответствует среднему («центру») распределения, второй определяет стандартное отклонение (разброс выборки), третий равен размеру выборки.

Построение гистограммы выполнено с помощью метода *hist*() библиотеки *Matplotlib*.

Аргумент *bins* задает количество интервалов гистограммы. По умолчанию используется значение 10. Если вместо целого числа в аргумент *bins* передать кортеж значений, то они будут использованы для задания границ интервалов. Таким образом, можно построить гистограмму с произвольным разбиением.

Результат визуализации представлен на рис. 5.5.

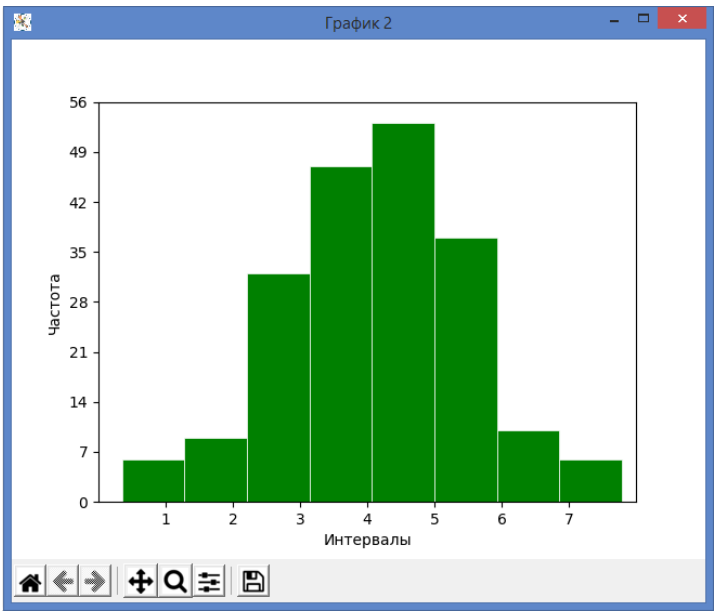

Рис. 5.5. Визуализация гистограммы

5.3 ОБРАБОТКА ИСКЛЮЧЕНИЙ

Любая программа может выполняться с ошибками. Часть из них связана с самим кодом. А другая часть связана с ситуациями, которые возникают довольно редко, но означают, что дальнейшее выполнение программы невозможно, если возникшую проблему никак не решить. Такие ситуации исключительны. Поэтому и механизм языка, предназначенный для работы с исключительными ситуациями, называется системой исключений (*exceptions*).

В *Python* исключения объединяются в *иерархию исключений*, представляющей дерево, корнем которого является *BaseException*, стволом – *Exception*, а дальше происходит ветвление на виды и конкретные исключения.

5.3.1 Конструкция try..except..finally

При возникновении исключения работа программы прерывается. Обеспечить корректное завершение можно с помощью конструкция ***try..except..finally*** со следующимформальным определением:

```
try:
    инструкции
except [Тип_исключения]:
    инструкции
finally:
    инструкции
```

Весь основной код, в котором потенциально может возникнуть исключение, помещается после ключевого слова ***try***. Если в этом коде генерируется исключение, то работа кода в блоке прерывается, и выполнение переходит в блок ***except***.

После ключевого слова ***except*** опционально можно указать, какое исключение будет обрабатываться (например, ValueError или KeyError). После слова *except* на следующей строке идут инструкции, выполняемые при возникновении исключения.

При обработке исключений также можно использовать необязательный блок ***finally***. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение.

Как правило, блок *finally* применяется для освобождения используемых ресурсов, например, для закрытия файлов. Однако он не обрабатывает исключения, поэтому без *except* при возникновении ошибки приложение аварийно завершится.

Блоки типа *try..except..finally* могут быть вложенными для обеспечения гибкого управления исключениями в *Python*.

Пример. В файле записаны целые числа, сформировать из них список.

Для решения задачи необходимо отслеживать два основных исключения: существование файла с исходными данными и значение литералов, преобразуемых в целые числа:

```
title = input('Введите название файла: ')
print(f'Открытие файла {title}')
try:
    f = open(title,'r')
    item = f.readlines()
    num = []
    try:
        for i in item:
            num.append(int(i))
        print(num)
    except ValueError:
        print('Не число. Завершить выполнение')
    except Exception:
        print('Возникло исключение')
    else:
        print('Исключение не возникло')
    finally:
        f.close()
        print('Файл закрыт')
except FileNotFoundError:
        print('Файл не найден')
except Exception:
        print('Возникло исключение')
```

При попытке открыть несуществующий для заданного пути файл, результат работы программы будет иметь вид:

```
Введите название файла: 1.txt
Открытие файла 1.txt
Файл не найден
```

Если файл был успешно открыт, данные из него записываются в список *item*, после чего начинается перебор элементов и

их преобразование в список целых чисел *num*. Если встречается литерал, значение которого не может быть преобразовано, срабатывает вложенное исключение с типом *ValueError*:

```
Введите название файла: data.txt
Открытие файла data.txt
Не число. Завершить выполнение
Файл закрыт
```

В случае отсутствия исключений на всех уровнях будет выдан следующий результат:

```
Открытие файла data.txt
[3, 5, 6]
Исключение не возникло
Файл закрыт
```

Следует отметить, внутренняя инструкция обработки исключений дополнена необязательным блоком *else*, который запускается в случае отсутствия в блоке t*ry* исключений.

5.3.2 Генерация исключений и оператор raise

Иногда возникает необходимость вручную сгенерировать то или иное исключение. Для этого применяется оператор *raise*, который позволяет прервать штатный поток исполнения при помощи возбуждения исключения. Если после инструкции отсутствует выражение (например, не указывается тип исключения), то повторно поднимается отловленное исключение. Если в данной области нет активного исключения (например, raise не находится внутри блока except), возбуждается *RuntimeError*.

Пример. Вычислить факториал числа, выполнив его предварительную проверку.

```
def factorial(num):
    if not isinstance(num, int):
        raise TypeError("Число должно быть целым.")
    if num < 0:
        raise ValueError("Число должно быть неотрица-
тельным.")

    def calc(num):
        if num <= 1:
            return 1
        return num * calc(num - 1)
```

```
return calc(num)
```

В функции *factorial*() сначала проверяются входные данные, чтобы убедиться, что пользователь предоставляет неотрицательное целое число. Затем определяется рекурсивная внутренняя функция с именем *calc*(), выполняющая вычисление факториала. На последнем шаге вызывается с*alc*() и выполняется соответствующий расчет.