

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 4

Функции

Содержание

1. Функции и модули.....	4
Что такое функция? Цели и задачи функции	4
Встроенные функции	7
Математические функции и случайные числа	9
Синтаксис объявления функций.....	15
Аргументы и возвращаемые значения	18
2. Расширенные приемы по работе с функциями....	27
Аргументы по умолчанию, аргументы-ключи	30
Область видимости, правило LEGB.....	35
Локальные и глобальные переменные в функциях	38
Функции как объекты первого класса.....	46
Рекурсия.....	51

3. Функциональное программирование.....	56
Что такое функциональное программирование?	56
Анонимные функции <code>lambda</code>	62
Функции высших порядков	66
Функции <code>map()</code> , <code>filter()</code> , <code>zip()</code>	72
Модуль <code>functools</code>	82
Замыкания	93
Карринг	99
4. Декораторы	104

1. Функции и модули

Что такое функция? Цели и задачи функции

Ранее наши программы представляли собой единый, функционально неделимый блок кода, который последовательно обрабатывался интерпретатором соответственно используемым условным блокам, циклам и т.д. Однако в реальных задачах, алгоритмы которых часто достаточно сложные и объемные, решение разбивается на отдельные шаги (каждый из которых имеет часто собственный алгоритм решения). При этом в общем алгоритме эти шаги (логика реализации этой подзадачи) могут повторяться.

Например, у нас есть данные о студенте (допустим в виде списка `[name, age, score]`).

```
student=['Bob', 19, 95]
```

Меню нашей программы предлагает пользователю вывести информацию о студенте, изменить информацию о студенте. Перед тем, как выполнить действия пункта меню, программа запрашивает логин пользователя для проверки прав доступа на это действие (есть ли такой логин в списке пользователей).

```
users=['admin', 'teacher', 'manager']  
  
template = "Name: {}; age: {}; scores: {}."  
  
while True:  
    print("1-print info")
```

```

print("2-mofify info")
print("3-exit")
userChoice = input("Select menu item: ")

if userChoice == "1":
    user = input("Login:")
    if user in users:
        print("Current information:")
        print(template.format(student[0],
                                student[1], student[2]))
elif userChoice == "2":
    if user in users:
        print("Current information:")
        print(template.format(student[0],
                                student[1], student[2]))
        userAnsw = input("Change this info:y-n?")
        if userAnsw == "y":
            name = input("Input new name:")
            age = int(input("Input new age:"))
            scores = input("Input new scores:")
            student[0] = name
            student[1] = age
            student[2] = scores
            print(template.format(student[0],
                                    student[1], student[2]))
elif userChoice == "3":
    print("Bye!")
    break
else:
    print("Error: wrong menu item!")

```

В этом случае блок кода, который осуществляет такую проверку пользователя и вывод текущей информации о студенте, будет повторяться перед каждым блоком кода для реализации пункта меню.

```
if user in users:
    print("Current information:")
    print(template.format(student[0],
                           student[1], student[2]))
```

А что, если пунктов меню не два, а намного больше? Именно для того, чтобы обеспечить повторение логики в основной программе без дублирования строк кода, которые реализуют эту логику, в программировании предусмотрены функции, которые являются неотъемлемой частью большинства языков программирования.

Функция — это именованный блок программного кода, который используется для выполнения некоторого набора действий (согласно алгоритму) и организован таким образом, чтобы многократно использоваться.

Для запуска такого фрагмента кода в нужном месте основной программы используется только имя функции (происходит вызов функции, т.е. запуск нашего, ранее написанного блока кода тела функции). Функции можно вызывать в любом месте программы Python, в том числе вызывать функции внутри других функций.

Использование функций в программировании обеспечивает такие преимущества:

- функции позволяют выполнять один и тот же фрагмент кода несколько раз без дублирования строк кода;
- функции разбивают длинные программы на более мелкие компоненты, что повышает читабельность кода и делает более понятной логику алгоритма;
- функции могут быть общими и использоваться другими программистами, которые будут просто использовать

их как «черный ящик» для решения определенной задачи, не задумываясь об особенностях ее реализации.

Функции обеспечивают лучшую модульность (структурированность) для вашего приложения и эффективность в повторном использовании кода.

Встроенные функции

В Python существует большое количество встроенных функций. Они перечислены ниже в алфавитном порядке.

A <code>abs()</code> <code>aiter()</code> <code>all()</code> <code>any()</code> <code>anext()</code> <code>ascii()</code>	E <code>enumerate()</code> <code>eval()</code> <code>exec()</code>	L <code>len()</code> <code>list()</code> <code>locals()</code>	R <code>range()</code> <code>repr()</code> <code>reversed()</code> <code>round()</code>
B <code>bin()</code> <code>bool()</code> <code>breakpoint()</code> <code>bytearray()</code> <code>bytes()</code>	F <code>filter()</code> <code>float()</code> <code>format()</code> <code>frozenset()</code>	M <code>map()</code> <code>max()</code> <code>memoryview()</code> <code>min()</code>	S <code>set()</code> <code>setattr()</code> <code>slice()</code> <code>sorted()</code> <code>staticmethod()</code> <code>str()</code> <code>sum()</code> <code>super()</code>
C <code>callable()</code> <code>chr()</code> <code>classmethod()</code> <code>compile()</code> <code>complex()</code>	G <code>getattr()</code> <code>globals()</code>	N <code>next()</code>	T <code>tuple()</code> <code>type()</code>
D <code>delattr()</code> <code>dict()</code> <code>dir()</code> <code>divmod()</code>	H <code>hasattr()</code> <code>hash()</code> <code>help()</code> <code>hex()</code>	O <code>object()</code> <code>oct()</code> <code>open()</code> <code>ord()</code>	V <code>vars()</code>
	I <code>id()</code> <code>input()</code> <code>int()</code> <code>isinstance()</code> <code>issubclass()</code>	P <code>pow()</code> <code>print()</code> <code>property()</code>	Z <code>zip()</code>
			—

Рисунок 1

Со многими из них мы уже знакомы. Например, встроенная функция `abs(x)` принимает на вход число `x` и возвращающую его абсолютное значение. Встроенная функция `chr(i)` возвращает строку, представляющую символ, код которого является целым числом `i`, а для преобразования строки в целое число мы использовали функцию `int(s)`.

```
x = int(input("Input Unicode code point: "))

if x<0:
    x = abs(x)
print(chr(x))
```

Как видно из примера, для использования встроенных функций мы просто указываем их имя (т. е. выполняем вызов этой встроенной, нужной нам, функции) и, при необходимости, передаем данные для их работы (в круглых скобках после имени).

Если программисту необходимо узнать, как выглядит общий синтаксис встроенной функции и что она возвращает, то можно вызвать встроенную функцию `help()`, на вход которой нужно передать имя интересующей функции.

Данный фрагмент кода можно написать отдельно от основной программы, например, в консоли или даже в отдельном файле, чтобы просто прочитать описание синтаксиса и назначение интересующей функции.

Например:

```
help(len)
```



```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Рисунок 2

Математические функции и случайные числа

Для решения математических задач в Python есть встроенный модуль `math`, содержащий набор методов и констант.

Для работы со встроенными математическими функциями нужно вначале импортировать модуль `math`.

```
import math
```

После подключения к библиотеке доступ к ее функциям осуществляется следующим образом:

```
math.имя_функции(...)
```

Рассмотрим наиболее распространенные функции модуля `math`.

Одной из самых востребованных функций, является функция округления числа до ближайшего целого. Для этой задачи в модуле `math` предусмотрены две функции:

- `ceil(x)` — возвращает ближайшее целое значение, которое больше или равно числу `x` (округление «вверх»).
- `floor(x)` — возвращает ближайшее целое значение, которое меньше или равно числу `x` (округление «вниз»).

```
import math

x = 0.23
y = 1.87
z = 3

print(math.ceil(x)) #1
print(math.ceil(y)) #2
print(math.ceil(z)) #3

print(math.floor(x)) #0
print(math.floor(y)) #1
print(math.floor(z)) #3
```

Если нужно получить целую или дробную часть числа, то в модуле `math` также есть соответствующие функции: `modf(x)` — возвращает дробную и целую часть `float` числа `x` с сохранением знака числа `x`. Тип возвращаемого результата — также `float`.

```
x = -2.45
y = 1.5

print(math.modf(x)) # (-0.45000000000000002, -2.0)
print(math.modf(y)) # (0.5, 1.0)
```

Для получения только целой части числа можно воспользоваться функцией `trunc(x)`:

```
x=-2.45
y=1.5

print(math.trunc(x)) #-2
print(math.trunc(y)) #1
```

Для получения абсолютного значения (модуля) числа можно воспользоваться функцией `fabs(x)`. В отличие от встроенной функции `abs()`, которая возвращает модуль числа с таким же типом, как и исходное число, функция `fabs(x)` возвращает результат типа `float`.

```
import math
x = -2
y = -2.5
z = 4

print(math.fabs(x)) #2.0
print(math.fabs(y)) #2.5
print(math.fabs(z)) #4.0

print(abs(x)) #2
print(abs(y)) #2.5
print(abs(z)) #4
```

Для вычисления факториала целого числа предусмотрена функция `factorial(x)`. Если число `x` не целое, то возникнет исключение `ValueError`.

```
1 import math
2
3 x=3
4 y=3.5
5
6 print(math.factorial(x)) #6
7 print(math.factorial(y))
```

--
'float' object cannot be interpreted as an integer

Рисунок 3

Для работы со случайными числами в Python используется модуль `random`. Он включает множество полезных функций для генерации, работы со случайными числами и последовательностями (например, случайное перемешивание элементов списка или извлечение случайного элемента из списка).

В начале работы необходимо подключить (импортировать) модуль `random`, чтобы в дальнейшем использовать его функции.

```
import random
```

Рассмотрим функции для генерации случайных чисел и последовательностей случайных чисел.

Для генерации псевдослучайных целых чисел предусмотрены функции `randint()` и `randrange()`. `randint(a, b)` — принимает только два аргумента `a`, `b` — границы диапазона целых чисел, из которого выбирается случайное число. При этом обе границы включаются в диапазон, т. е. случайное число будет выбрано из отрезка `[a; b]`.

Числа могут быть отрицательными, для этого границы диапазона должны быть отрицательными.

Однако первое число всегда должно быть меньше второго (`a < b`).

```
import random

for i in range(3):
    print("step ", i+1)
    print(random.randint(1,5))
    print(random.randint(-3,2))
    print(random.randint(-10,-6))
```

```

step  1
4
0
-6
step  2
5
-2
-9
step  3
3
-3
-6

```

Рисунок 4

Функция `randrange()` может принимать от одного до трех аргументов:

- если указан один аргумент — `randrange(a)`, то результат — случайное число от `0` до указанного. Причем сам аргумент в диапазон генерации не входит: `[0; a)`;

Примечание: в записи `[0; a)` символ «`[`» обозначает, что значение `0` входит в промежуток (диапазон генерации), а символ «`)`» говорит, что значение `a` в диапазон не входит, т.е генерация будет проводиться от `0` (включительно) до `a-1`.

- при двух аргументах — `randrange(a, b)` работает аналогично `randint()`, кроме того, что верхняя граница `b` не входит в диапазон генерации, т. е. `[a; b)`;
- если указано три аргумента — `randrange(a, b, c)`, то первые два являются границами диапазона, а третий — шаг генератора, т. е., например, при вызове `randrange`

(10, 18, 2), случайно» число будет выбираться из последовательности чисел 10, 12, 15, 17.

```
import random

print(random.randrange(10))
print(random.randrange(-4, 3))
print(random.randrange(10, 20, 3))
```



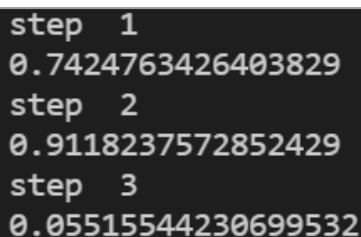
6
0
19

Рисунок 5

Для генерации случайного вещественного числа используется функция `random()`. Данная функция вызывается без аргументов, т.к. случайное число генерируется в диапазоне от 0 до 1, но не включая 1:

```
import random

for i in range(3):
    print("step ", i+1)
    print(random.random())
```



step 1
0.7424763426403829
step 2
0.9118237572852429
step 3
0.05515544230699532

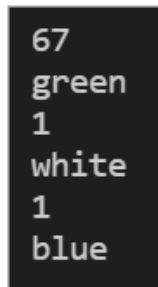
Рисунок 6

Если же нам необходимо получить случайный элемент из последовательности, например, случайный элемент списка, то можно воспользоваться функцией `choice(L)`, где `L` — это последовательность или ее имя:

```
import random

L = [1, 34, 67, 89]

for i in range(3):
    print(random.choice(L))
    print(random.choice(['red', 'blue', 'green',
                        'white']))
```



```
67
green
1
white
1
blue
```

Рисунок 7

Синтаксис объявления функций

Как и в других языках программирования, в Python можно создавать собственные (пользовательские) функции. Работа с пользовательскими функциями происходит в два этапа: вначале функцию нужно создать (определить, т. е. написать блок кода, реализующий ее логику), а потом вызвать в том месте (или нескольких местах) кода основной программы (или в других функциях), где это необходимо.

В Python функции определяются с помощью ключевого слова `def`, после которого указывается имя функции и круглые скобки с входными параметрами (если они есть):

```
def имя_функции (входные_параметры) :  
    тело функции
```

Например:

```
def printMsg() :  
    print("Hello!")  
    print("Welcome!!!")
```

В нашем примере функция не принимает никаких входных данных (поэтому круглые скобки пустые, но они обязательны) и не возвращает никаких результатов, а просто выполняет некоторую последовательность действий.

Имя функции должно соответствовать таким же правилам, как и имена переменных, т. е. начинаться с буквы или символа подчеркивания («`_`») и содержать только буквы, цифры и символ «`_`».

Как мы уже знаем из примеров использования встроенных функций, функции, при необходимости, могут получать данные (аргументы функции) и возвращать результат. При этом входные данные для функций приходят не с клавиатуры, файла и т.д., а из основной (вызывающей их) программы. Сюда же они возвращают результат своей работы, если это предусмотрено этой функцией.

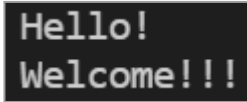
Строки кода, реализующие логику работы функции, называются телом функции и следуют после знака «`:`» с отступом (как, например, тело цикла).

Для того, чтобы вызвать функцию нужно в коде основной программы указать имя функции и скобки (с параметрами, если они необходимы).

```
printMsg()
```

Поскольку в нашем примере в функцию мы не передаем никаких входных данных (у нашей функции нет параметров), скобки при вызове также пустые:

```
def printMsg():  
    print("Hello!")  
    print("Welcome!!!")  
  
printMsg()
```



```
Hello!  
Welcome!!!
```

Рисунок 8

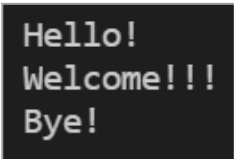
Когда функция вызывается, поток выполнения программы переходит к точке ее определения и начинает исполнять тело функции.

После того, как все строки кода из тела функции выполнены, поток выполнения возвращается в основную программу, в то место, откуда функция вызывалась.

Далее выполняется следующее за вызовом выражение (следующая строка кода основной программы).

```
def printMsg():  
    print("Hello!")  
    print("Welcome!!!")
```

```
printMsg()  
print("Bye!")
```



```
Hello!  
Welcome!!!  
Bye!
```

Рисунок 9

Аргументы и возвращаемые значения

При необходимости и в зависимости от особенностей решаемой задачи функции могут принимать данные для их дальнейшей обработки внутри функции или для реализации логики алгоритма функции на основе их значений.

Такие входные данные называются параметрами функции. Параметры функции — это обычные переменные, которые функция использует внутри своего кода (для внутренних действий согласно некоторому алгоритму). Имя параметра указывается в круглых скобках при объявлении функции:

```
def printMsg1(myMsg):  
    print(myMsg)
```

Если параметров у функции несколько, то их имена перечисляются в круглых скобках через запятую.

```
def printMsg2(myMsg1, myMsg2):  
    print(myMsg1)  
    print(myMsg2)
```

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции.

Конкретные значения, которые передаются в функцию при ее вызове, называются аргументами.

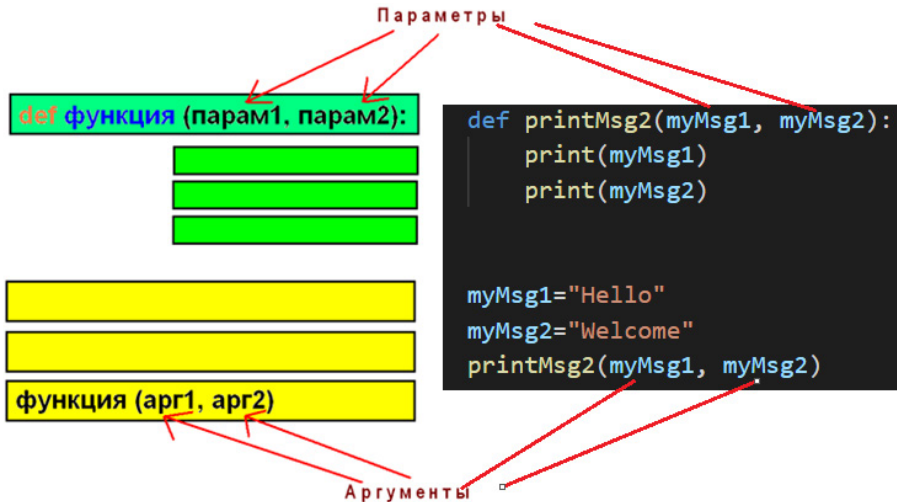
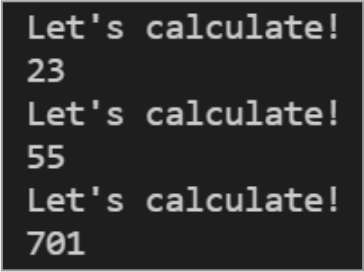


Рисунок 10

Когда функция вызывается, то ей передаются аргументы. В нашем примере указаны переменные `myMsg1` и `myMsg2`. Однако фактически передаются не эти переменные, а их значения, т. е. строки «Hello» и «Welcome». Поэтому аргументы также еще называют фактическими параметрами.

В нашем примере имена аргументов и параметров совпадают, но это совсем не обязательно. Имена могут быть разные, т.к. фактически в момент вызова функции программа копирует значения из переменных-аргументов в только что созданные переменные-параметры.

```
def calculate(a,b,c):  
    print("Let's calculate!")  
    print (a+b*c**2)  
  
x = 5  
y = 2  
z = 3  
calculate(x,y,z)  
calculate(x,y,5)  
calculate(1,7,10)
```



```
Let's calculate!  
23  
Let's calculate!  
55  
Let's calculate!  
701
```

Рисунок 11

Как мы видим, аргументами могут быть как имена переменных, так и литералы (значения).

Функции в наших примерах просто выполняли некоторые действия над параметрами (выводили тексты сообщений в консоль), не возвращая никакого результата в основную программу. Однако часто есть необходимость создавать функции, которые не только выполняют какие-то действия, но и возвращают вычисленный внутри функции результат в основную программу.

В этом случае в основной программе нужно предусмотреть переменную, которая будет принимать результат работы функции.

Например, таким образом мы работали со встроенной функцией `input()`:

```
name = input("Input your name:")
```

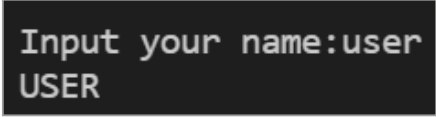
Таким же образом и пользовательские функции могут передавать результат своей работы (возвращать значение).

Выход из функции и передача данных (некоторого вычисленного внутри функции результата) происходит в то место, откуда эта функция была вызвана.

Для выхода из функции в ее теле нужно использовать оператор `return`. Если интерпретатор, выполняя тело функции, встречает оператор `return`, то он «забирает» значение переменной, имя которой указано после этой команды, и «выходит» из функции сразу после этого (даже в случае, если дальше есть еще команды).

Например:

```
def modifyName(userName):  
    newName = userName.upper()  
    return newName  
  
name = input("Input your name:")  
print(modifyName(name))
```



```
Input your name:user  
USER
```

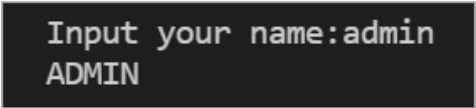
Рисунок 12

Если мы добавим строки кода после команды `return newName` в теле функции `modifyName(userName)`, то они

не будут выполнены, т.к. программы «выйдет» из функции после `return newName`

```
def modifyName(userName):
    newName = userName.upper()
    return newName
    print("Hello!")

name = input("Input your name:")
print(modifyName(name))
```



```
Input your name:admin
ADMIN
```

Рисунок 13

Допустим, что нам нужно вычислить значение выражения, вычисляемое по такой формуле:

$$y = \begin{cases} 5 \cdot x + 2 \cdot z, & \text{если } x + z \leq 0, \\ 10 \cdot x - 3 \cdot z, & \text{если } x + z > 0. \end{cases}$$

Как видно формула для вычисления результата разная и зависит от значения выражения. Поэтому удобно создать отдельную функцию, которая вычисляет это выражение, а потом проверять значение, которое вернет функция, в основной программе и по результатам проверки применять одну или вторую формулу:

```
def calculate(a, b):
    return a + b

x = float(input("Input x:"))
z = float(input("Input z:"))
```

```

if calculate(x,z)<=0:
    y = 5*x+2*z
else:
    y = 10*x-3*z

print("y = ", y)

```

```

Input x:2
Input z:4
y= 8.0

```

Рисунок 14

Предположим, что у нас есть список клиентов и нам для заданного клиента нужно вывести его позиции в списке и подсчитать, сколько раз он встречается в списке, т. е. для частых клиентов (более одного раза встречающихся в списке) действует некоторая скидка.

Создадим функцию, которая выводит позиции элементов (клиентов) в списке и подсчитывает (возвращает) число вхождений.

В основном коде используем результат работы этой функции для определения ситуации, будет ли скидка этому клиенту:

```

def checkCustomer(customer, customers):
    result = 0
    print("Client's positions in the list:")
    for i in range(len(customers)):
        if customers[i] == customer:
            print(i)
            result += 1
    return result

```

```
customerList=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']
myCustomer='Bob'
if checkCustomer(myCustomer, customerList)>1:
    print("Customer", myCustomer, "will get a discount")
```

Полученный результат для клиента «Bob»:

```
Client's positions in the list:
0
3
Customer Bob will get a discount
```

Рисунок 15

В Python функции могут возвращать несколько значений одновременно:

```
def modifyName2(userName):
    newName1=userName.upper()
    newName2=userName.lower()
    return newName1, newName2

name=input("Input your name:")
print(modifyName2(name))
```

Результат работы такой функции — это кортеж:

```
Input your name:Admin
('ADMIN', 'admin')
```

Рисунок 16

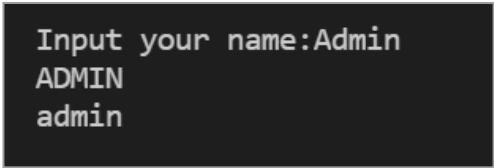
Для дальнейшей работы удобно результаты такой функции присвоить нескольким переменным:


```
def modifyName2(userName):
    newName1=userName.upper()
    newName2=userName.lower()
    return newName1, newName2

name=input("Input your name:")

nameUpper, nameLower=modifyName2(name)

print(nameUpper)
print(nameLower)
```



```
Input your name:Admin
ADMIN
admin
```

Рисунок 17

Переменной, которая указана первой в строке `nameUpper, nameLower = modifyName2(name)` будет присвоен первый результат, возвращаемый функцией, т. е. значение первой переменной после команды `return` в теле функции.

Внутри тела функции может быть несколько операторов `return`, но в процессе ее работы будет выполнен только один из них:

```
def checkLog(userLog):
    if userLog=='admin':
        return userLog.upper()
    elif userLog=='user':
        return userLog.lower()
    else:
        return "Wrong login!"
```

```
print(checkLog("admin"))  
print(checkLog("user"))  
print(checkLog("student"))
```

```
ADMIN  
user  
Wrong login!
```

Рисунок 18

2. Расширенные приемы по работе с функциями

Предположим, нам необходимо написать функцию, которая вычисляет среднее арифметическое произвольного количества чисел (своих аргументов). Получается, что заранее нам неизвестно, сколько входных параметров нужно указать в ее определении: три, четыре, а может десять. В разных ситуациях функция будет обрабатывать разное количество входных данных, т. е. набор (массив параметров), длина которого заранее неизвестна.

Для решения этой проблемы в Python предусмотрены такие подходы как упаковка и распаковка аргументов.

Рассмотрим синтаксис, процесс и примеры упаковки аргументов функции.

Общий синтаксис:

```
def имя_функции(*имя_набора_параметров):  
    тело функции
```

Обработка (процесс упаковки) происходит таким образом: интерпретатор в точке вызова функции «упаковывает» (помещает) все аргументы (указанные при вызове функции) в кортеж, имя которого приведено после символа «*» в строке объявления функции. В теле функции нужно учитывать особенности работы с кортежем (обрабатывать аргументы как элементы последовательности с помощью их позиции-индекса в последовательности).

Поэтому такой параметр называют также параметр с позиционными аргументами переменной длины. Имя параметра (набора аргументов) произвольное (с учетом требований к именам переменных), чаще всего традиционно используют имя `*args`.

Рассмотрим на примере.

```
def meanF(*args):  
    s=0  
    k=0  
    for i in args:  
        s+=i  
        k+=1  
    return s/k  
  
print("(1+2+3)/3=",meanF(1,2,3))  
print("(1+2+3+4+5)/5=",meanF(1,2,3,4,5))
```

```
(1+2+3)/3= 2.0  
(1+2+3+4+5)/5= 3.0
```

Рисунок 19

Итак, у нас есть функция, работающая с произвольным числом аргументов. Однако, перечислять весь набор аргументов при вызове функции (особенно, когда их большое количество) неудобно и это также может стать причиной ошибки в наборе при значительном количестве значений.

Для решения этой проблемы используется распаковка аргументов функции — это способ передачи элементов коллекции (списка, кортежа, словаря) через запятую на вход функции в качестве ее аргументов.

Для этого мы вначале создаем такую коллекцию:

```
numbers=[1,2,3,4,5]
```

А затем при вызове функции передаем ей эту коллекцию как аргумент, поставив перед ее именем символ «*»:

```
print("(1+2+3+4+5)/5=", meanF(*numbers))
```

Рассмотрим задачу: необходимо написать функцию, которая выводит логин и пароль пользователя (каждый из них с новой строки) согласно некоторому шаблону.

Набор исходных данных — это вложенный список (набор логинов и паролей нескольких пользователей):

```
users=[['user1', '111'],
        ['user2', '2222'],
        ['user3', '33333']]
```

Создадим нашу функцию:

```
def printInfo(*clients):
    for i in range(len(clients)):
        if i==0:
            print("login: {}".format(clients[i]))
        elif i==1:
            print("pass: {}".format(clients[i]))
```

Если мы просто обойдем список в цикле, передавая на каждом шаге вложенный список на вход нашей функции, то данные будут выведены некорректно:

```
for user in users:
    printInfo(user)
```

```
login: ['user1', '111']  
login: ['user2', '2222']  
login: ['user3', '33333']
```

Рисунок 20

Использование подхода с распаковкой решит эту проблему:

```
for user in users:  
    printInfo(*user)
```

```
login: user1  
pass: 111  
login: user2  
pass: 2222  
login: user3  
pass: 33333
```

Рисунок 21

Аргументы по умолчанию, аргументы-ключи

Пусть у нас есть функция, которая принимает два аргумента (логин и пароль пользователя) и в зависимости от значения логина выдает разные приветственные сообщения для пользователя.

```
def userGreetings(login, passw):  
    if login=='admin':  
        print("Welcome, admin")
```

```

elif login=='student':
    print("Welcome, student")
else:
    print("Welcome, guy")

userGreetings()

```

Возникло исключение: `TypeError` ×

`userGreetings()` missing 2 required positional arguments: 'login' and 'passw'

Рисунок 22

Если мы вызовем данную функцию, указав только первый аргумент (логин)

```
userGreetings("admin")
```

Рисунок 23

то тоже будет ошибка:

Возникло исключение: `TypeError` ×

`userGreetings()` missing 1 required positional argument: 'passw'

Рисунок 24

Причина ошибки понятна: в вызове функции `userGreetings()` отсутствуют два необходимых аргумента или один из двух необходимых аргументов.

То есть нам нужно, чтобы одна и та же функция работала и с двумя, и с одним, и вообще без аргументов? Создавать еще две копии этой же функции, но с разным числом параметров при объявлении — не самый рациональный вариант решения этой проблемы.

Например, если в системе есть пароль по умолчанию для всех пользователей «111» и каждый может зайти в систему либо с помощью своего пароля, либо, используя «111», то нам нужно, чтобы наша функция `userGreetings()` работала и с двумя аргументами, и с одним (логином).

Для обработки таких ситуаций предусмотрены параметры (аргументы) по умолчанию. В функциях аргумент может быть обязательным (как в наших предыдущих примерах) и необязательным. У необязательного аргумента должно быть задано по умолчанию.

Обработка таких ситуаций отличается только в строке объявления функции:

```
def userGreetings(login, passw="111"):
    if login=='admin':
        print("Welcome, admin")
    elif login=='student':
        print("Welcome, student")
    else:
        print("Welcome, guy")
```

Первый аргумент (обязательный) приведен стандартным способом, а вот второй (необязательный) — это аргумент со значением по умолчанию, поэтому он приводится в таком формате:

```
имя_аргумента = значение_аргумента
```

Строка с вызовом функции в этом случае может содержать как один, так и два аргумента:

```
userGreetings("admin", "12345")
userGreetings("student")
```



```
Welcome, admin
Welcome, student
```

Рисунок 25

Мы можем создавать аргументы по умолчанию любое количество (в зависимости от условия задачи), но все они должны быть приведены в конце списка аргументов функции, после обязательных аргументов (если такие предусмотрены).

Рассмотрим синтаксически правильные и неправильные строки кода для объявления функции с аргументами по умолчанию:

```
def childParent(childName = 'Bob', parent):
```

неверно

```
def childParent(parent, childName='Bob'):
```

верно

```
def checkUser(userName, userLog = 'student',
               userPass='111'):
```

верно

```
def checkUser(userName, userLog = 'student',
               userPass):
```

неверно

Во всех предыдущих ситуациях мы работали с позиционными аргументами функций (самый распространенный подход), т. е. это такие аргументы, чьи значения в момент вызова функции копируются в соответствующие параметры функции, согласно порядку их следования, в строке объявления функции.

Например:

```
def myCalculation(a,b,c):
    return (a+b)**c

def myCalculation1(a,b,c=50):
    return (a+b)**c

print(myCalculation(10,20,30)) #a=10, b=20, c=30
print(myCalculation1(10,20)) #a=10, b=20, c=50
```

Несмотря на популярность использования позиционных аргументов у них есть следующий недостаток: нужно помнить порядок (позицию) каждого аргумента в объявлении функции, чтобы правильно указать нужные значения (или переменные) при вызове функции.

Например, у нас есть функция, которая вычисляет индекс массы тела:

$$I = \frac{m}{h^2},$$

где:

m — масса тела в килограммах;

h — рост в метрах.

```
def calculationBMI(m,h):
    return m/(h**2)

print(calculationBMI(50,1.6)) #19.53
```

В строке вызова функции мы просто указываем нужные значения (роста, веса), но если мы случайно их перепутаем, то получим неверное значение индекса массы тела.

```
print(calculationBMI(1.6, 50)) #0.00064
```

Во избежание подобной путаницы в Python предусмотрены аргументы-ключи (keyword-аргументы). Их использование происходит в точке вызова функции:

```
print(calculationBMI(h=1.6, m=50)) #19.53
```

При этом нужно использовать такие же имена аргументов, как и в объявлении функции. Однако, порядок их следования может быть любым.

Можно в одной функции использовать и позиционные, и keyword-аргументы. Важно помнить, что первыми (в вызове функции) должны идти позиционные аргументы:

```
def calculationUserBMI(m,h,name):
    print ("{}'s BMI={}".format(name, m/(h**2)))

calculationUserBMI(50,name='Jane',h=1.6) #19.53
```

Область видимости, правило LEGB

Концепция переменных, которая позволяет сохранять значения, а позже извлекать и (при необходимости) менять эти значения, является одной из основных во всех языках программирования.

Когда мы создаем переменные в наших программах, то нужно четко понимать «кто их может видеть», т. е. в какой части программы их можно найти и получить их значение. Эти правила определяют границы для каждой переменной: в каком месте программы она доступна.

Область видимости переменных — это такая часть программного кода, в рамках которой по имени переменной (идентификатору) можно получить (изменить) значение.

Вне области видимости этот же идентификатор может быть свободен или связан абсолютно с другим значением.

Основное назначение области видимости — это предотвращение конфликтов идентификаторов и, как следствие, непредсказуемого поведения программы.

Самыми распространенными и часто используемыми во многих языках программирования являются:

- **глобальная область видимости** — идентификаторы, определяемые в этой области, доступны в любом месте программы;
- **локальная область видимости** — идентификаторы, определяемые в этой области, доступны только коду в этой области (например, внутри конкретной функции).

Так как Python является языком с динамической типизацией, то переменные создаются тогда, когда мы впервые присваиваем им некоторое значение. На основе этой операции присваивания интерпретатор Python определяет область видимости данного идентификатора, т. е. та область кода, где мы создаем переменную (функцию) определяет ее область видимости.

Каждый раз, когда мы в программе используем некоторый идентификатор (имя переменной или функции), интерпретатор Python проходит по всем уровням области видимости для поиска такого идентификатора. Если такой идентификатор существует, то мы получаем результат его первого обнаружения (в случаях, когда один идентификатор используется в разных областях видимости).

Этот процесс поиска следует правилу LEGB (*Local* — локальная, *Enclosing* — вложенная, *Global* — глобальная и *Built-in* — встроенная).

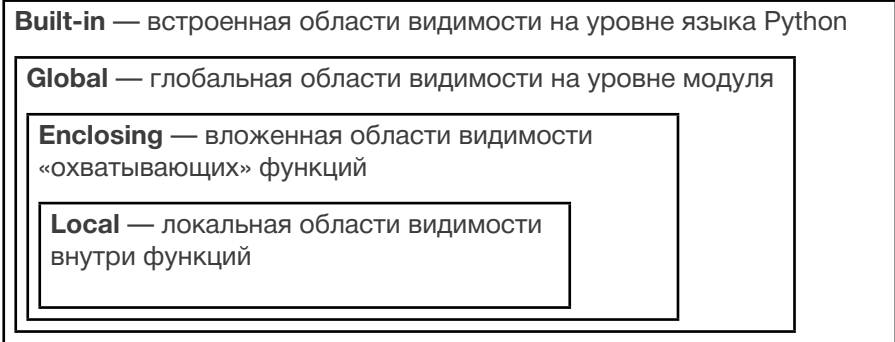


Рисунок 26

Локальная область видимости (*Local*) — это часть кода, которая соответствует телу некоторой функции или лямбда-выражения (будем рассматривать в уроках далее). Идентификаторы, определенные внутри функции, относятся к ее локальной области видимости и существуют только в рамках этой функции или ее локальной области. Эта область создается не при объявлении функции, а при ее вызове (т. е. сколько было вызовов функции, столько и разных, новых, локальных областей).

После завершения работы функции локальная область видимости уничтожается, а идентификаторы «забываются».

Вложенная (нелокальная) область видимости (*Enclosing*) — данная область видимости возникает при работе вложенных функций и содержит идентификаторы, которые мы определяем во вложенной функции. Идентификаторы этой области «видны» из кода внутренних и «охватывающих» функций (внутри которых находятся другие внутренние функции).

Глобальная область видимости (*Global*) — это уровень программы (скрипта, модуля) Python, который содержит

все идентификаторы, которые мы определили на уровне программы (т.е. вне функций, если они есть). Эта область создается в момент запуска нашей программы. Идентификаторы, определенные в этой области, будут доступны в любом месте кода программы. Во время выполнения программы существует единственная глобальная область видимости, которая сохраняется пока наша программа не завершится, после чего все идентификаторы этой области будут забыты.

Встроенная области видимости (*Built-in*) — это уровень языка Python, т. е. все встроенные объекты, функции Python находятся в этой области. Эта область активизируется в момент запуска интерпретатора Python и все встроенные объекты автоматически загружаются в нее. Таким образом, мы можем применять их (например, встроенную функцию `abs()`) без использования операции импорта какого-либо модуля.

Локальные и глобальные переменные в функциях

Как мы уже знаем, в момент вызова функции создается новая локальная области видимости.

Все идентификаторы, назначенные внутри функции, создаются в момент ее вызова, видимы (доступны) только в пределах данной функции (внутри инструкции `def`) и будут уничтожены, когда работа функции закончится.

Рассмотрим на примере:

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
```

```
print(f'The square of {baseVar} is: {resultVar}')
calculateExample1()
```

Функция `calculateExample1()` вычисляет квадрат введенного пользователем целого числа.

После первого вызова этой функции создается локальная область видимости, которая содержит две переменные: `baseVar` со значением 3 (так как пользователь в нашей случае ввел значение 3) и `resultVar`, которая после вычисления получит значение 9.

```
input base variable: 3
The square of 3 is: 9
```

Рисунок 27

Попробуем вызвать нашу функцию еще раз:

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

calculateExample1()
calculateExample1()
```

Вначале создается первая область видимости (после первого вызова функции `calculateExample1()`), в которой переменная `baseVar` равна 4 (предполагаем, что пользователь ввел значение 4), а переменная `resultVar` получает значение 16. После окончания работы функции (после ее первого вызова) эти переменные со своими значениями исчезают (т. е. программа их не запоминает).

При втором вызове функции `calculateExample1()` создастся новая локальная область с другими локальными переменными `baseVar` и `resultVar` (т. е. произойдёт новое выделение новых фрагментов памяти для них) со значениями 5 (предполагаем, что пользователь во второй раз ввел значение 5) и 25.

```
input base variable: 4
The square of 4 is: 16
input base variable: 5
The square of 5 is: 25
```

Рисунок 28

Если мы попытаемся обратиться к переменной `baseVar` (или `resultVar`) вне функции (например, после ее вызова),

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

calculateExample1()

print(baseVar)
```

то получим ошибку `NameError`, т.к. данные переменные существуют только в локальной области, которая, в свою очередь, существует только в момент вызова (работы) функции `calculateExample1()`.

```
name 'baseVar' is not defined
```

Рисунок 29

В сообщении об ошибке `NameError` будет указано имя (в нашем примере `baseVar`), которое интерпретатору не удалось обнаружить.

Так как у нас нет доступа к локальным переменным за пределами функций, которые их создали, то значит в разных функциях можно использоваться одинаковые имена переменных.

Например:

```
def calculateExample1():
    baseVar = int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

def calculateExample2():
    baseVar = int(input("input base variable: "))
    resultVar = baseVar ** 3 + 10
    print(f'The result is: {resultVar}')

calculateExample1()
calculateExample2()
```

Вторая функция `calculateExample2()` использует в своем теле переменные с такими же именами, как и первая функция `calculateExample1()`. Функция `calculateExample2()` не может видеть локальные переменные функции `calculateExample1()` и наоборот. Поэтому обе функции работают корректно и конфликта имен переменных не возникает.

```
input base variable: 4
The square of 4 is: 16
input base variable: 2
The result is: 18
```

Рисунок 30

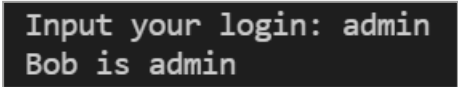
Как было рассмотрено ранее, с момента запуска нашей программы образуется глобальная область видимости и все идентификаторы, определенные в ней (за пределами функций) доступны из любой точки кода, в том числе и внутри функций.

Рассмотрим на примере:

```
userName ="Bob"
def checkUser():
    userLog=input("Input your login: ")
    if userLog=='admin':
        print("{} is admin".format(userName))
    else:
        print("{} is user".format(userName))

checkUser()
```

У нас есть глобальная переменная `userName`, которую мы используем (считываем ее значение) внутри нашей функции `checkUser()` для вывода сообщения пользователю:



```
Input your login: admin
Bob is admin
```

Рисунок 31

Теперь создадим внутри этой функции локальную переменную с таким же именем `userName` и присвоим ей другое значение.

```
userName="Bob"

def checkUser():
    userLog = input("Input your login: ")
```

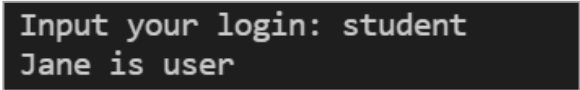
```

userName = "Jane"
if userLog == 'admin':
    print("{} is admin".format(userName))
else:
    print("{} is user".format(userName))

checkUser()

```

В этом случае при обращении к переменной `userName` будет использована локальная переменная с ее значением «Jane».



```

Input your login: student
Jane is user

```

Рисунок 32

Для того, чтобы внутри функции изменить значение глобальной переменной необходимо использовать ключевое слово `global`.

Используя `global` мы можем определить идентификатор (или список идентификаторов), которые наша программа будет воспринимать, как глобальные.

Рассмотрим на примере:

```

userName="Bob"
print("{} – first user".format(userName))

def checkUser():
    global userName
    userName="Joe"
    userLog=input("Input your login: ")
    if userLog=='admin':
        print("{} is admin".format(userName))

```

```
else:  
    print("{} is user".format(userName))  
checkUser()  
print("{} - second user".format(userName))
```

Внутри нашей функции `checkUser()` мы строкой кода `global userName` «говорим» интерпретатору, что данную переменную нужно искать в глобальной области видимости, а не создавать новое локальное имя. В результате существующее значение переменной `userName` было внутри функции изменено на новое.

```
Bob - first user  
Input your login: admin  
Joe is admin  
Joe - second user
```

Рисунок 33

Использование глобальных переменных имеет серьезный недостаток: их значения могут изменяться внутри функций. Это изменение мы можем и не заметить, предполагая, например, что наши переменные сохраняют исходные значения.

Рассмотрим следующий пример: допустим, что нам необходимо написать функцию авторизации пользователя, которая принимает на вход только пароль. Логин является глобальной переменной, значение которой нужно только считывать в теле функции для проверки на соответствие пароля.

При этом программистом в теле функции была вызвана другая функция, выводящая приветствие пользователю.

Допустим, что программист не видел ее код (внутри которого, возможно, случайно открыт доступ на изменение этой переменной `userLog` с помощью оператора `global` и далее ей присвоено значение «*admin*») и просто воспользовался ей для вывода приветствия в процессе авторизации.

```
def greetingUser():
    global userLog
    userLog="admin"
    print("Welcome!")

def checkPassw(passw):
    greetingUser()

    if userLog=="admin" and passw=='111':
        print("{} , access is allowed. You are admin!".
            format(userLog))
    elif userLog=="student" and passw=='222':
        print("{} , access is allowed. Welcome!".
            format(userLog))
    else:
        print("{} , access isn't allowed. Wrong password!".
            format(userLog))

userLog=input("Input your login: ")
userPassw=input("Input your password: ")

checkPassw(userPassw)
```

В результате, когда пользователь ввел значение логина «*student*» и оно попало в глобальную переменную `userLog`, то далее, после вызова функции `checkPassw()`, введенное значение «*student*» было внутри функции изменено на «*admin*». После этого процесс авторизации

прошел некорректно несмотря на то, что пользователь ввел корректные значения логина и пароля.

```
Input your login: student
Input your password: 222
Welcome!
admin, access isn't allowed. Wrong password!
```

Рисунок 34

Таким образом, глобальные переменные потенциально позволяют функциям иметь неявные, не всегда легко и сразу обнаруживаемые побочные эффекты.

Функции как объекты первого класса

Во многих языках программирования существует понятие объект первого класса.

Объект первого класса — это такой объект, который создается динамически (во время выполнения программы). Его можно передавать функции в качестве аргумента и возвращать как значение, получаемое в результате работы функции. Также объект первого класса можно присвоить некоторой переменной в качестве значения. Таким образом, функции в Python являются объектами первого класса.

В Python функции являются объектами. Используем встроенную функции `type()`, чтобы вывести тип данных нашей функции `sayHello()`:

```
def sayHello():
    name=input("What's is your name? ")
    print("Hello, {}".format(name))

print(type(sayHello)) #<class 'function'>
```

Теперь присвоим функцию в качестве значения переменной и вызовем ее с помощью этой переменной:

```
def sayHello():
    name=input("What's is your name? ")
    print("Hello, {}".format(name))

greeting=sayHello
greeting()
```

```
What's is your name? Student
Hello, Student
```

Рисунок 35

В нашем примере функция `sayHello` присваивается переменной `greeting` в качестве значения. Обратите внимание, что присваивается только имя `sayHello` без последующих круглых скобок, т. к. переменная `greeting` должна получить ссылку на объект функции. Далее, указав после имени переменной `greeting` круглые скобки, мы запустили ту функцию, на которую эта переменная ссылается.

Теперь рассмотрим, как можно передать функцию в качестве аргумента другой функции. Допустим у нас есть список клиентов и есть функция, которая выводит разные приветствия для одного клиента, в зависимости от того, является ли он администратором, студентом или просто клиентом.

```
customers=['AdminJane', 'adminBob', 'STUDENTbob',
           'studentAlice', 'Kate']

def sayHello(customer):
    if customer.find("admin")!=-1:
```

```

    print("Hello, admin - {}".format(customer))
elif customer.find("student") != -1:
    print("Hello, student - {}".format(customer))
else:
    print("Hello, {}".format(customer))

```

Для корректной работы нам необходимо перед запуском нашей функции `sayHello(customer)` перевести логин клиента в нижний регистр (т.к. внутри функции в проверках логинов используется нижний регистр).

Реализуем это в отдельной функции, которая будет принимать на вход список логинов клиентов и для каждого из них (после предварительного перевода в нижний регистр) запускать функцию `sayHello(customer)`:

```

customers=['AdminJane', 'adminBob', 'STUDENTbob',
           'studentAlice', 'Kate']

def sayHello(customer):
    if customer.find("admin") != -1:
        print("Hello, admin - {}".format(customer))
    elif customer.find("student") != -1:
        print("Hello, student - {}".format(customer))
    else:
        print("Hello, {}".format(customer))

def greetings(customList, greetF):
    for c in customList:
        greetF(c.lower())

greetings(customers, sayHello)

```


Результат:

```
Hello, admin - adminjane
Hello, admin - adminbob
Hello, student - studentbob
Hello, student - studentalice
Hello, kate
```

Рисунок 36

Следующий наш вопрос — как возвращать функцию в качестве значения и когда нам это может понадобиться?

Допустим, что у нас есть простые три функции для вычисления разных выражений:

```
def calculateExample1(a,b):
    return a**b+2*a-b/0.23

def calculateExample2(a,b):
    return a*b-5**a+b

def calculateExample3(a,b):
    return b**4+a**2-a/b
```

Пользователь выбирает, какое из выражений ему надо вычислить, и мы должны запустить нужную функцию. Реализуем проверку выбора пользователя с помощью отдельной функции `userChoice()`, которая, в свою очередь, будет возвращать функцию, выполняющую нужные вычисления.

```
def userChoice(c):
    if c=="1":
        return calculateExample1
```

```
elif c=="2":
    return calculateExample2
elif c=="3":
    return calculateExample3
```

Теперь нужно вызвать нашу функцию `userChoice()` с аргументом, который соответствует выбору пользователя и присвоить результат ее работы (одну из функций-вычислителей) в отдельную переменную `myCalculation`.

Затем, используя переменную `myCalculation`, которая содержит ссылку на объект нужной функции-вычислителя, запустим вычисление с аргументами `x` и `y`.

```
x=2
y=3

for i in range(3):
    userAncw=input("Your choice:")
    myCalculation = userChoice(userAncw)

    print(myCalculation(x,y))
```

Результат:

```
Your choice:1
-1.0434782608695645
Your choice:3
84.33333333333333
Your choice:2
-16
```

Рисунок 37

Рекурсия

Как мы уже знаем, функции могут вызывать другие функции (содержать в своем теле их вызов). Также функции могут вызывать сами себя. Такое свойство называется рекурсией, а функции — рекурсивными.

В программировании рекурсия позволяет описывать (реализовывать) процесс, который повторяется без использования операторов цикла.

Классическим примером, когда полезно использовать рекурсию в программировании, является задача вычисления факториала числа $n!$.

Факториал числа n — это произведение всех чисел от 1 до n включительно.

Формула для вычисления $n!$:

$$n! = 1 * 2 * 3 * \dots * n.$$

Например,

$$3! = 1 * 2 * 3$$

$$4! = 1 * 2 * 3 * 4 = 3! * 4$$

$$5! = 1 * 2 * 3 * 4 = 4! * 5$$

При этом $0! = 1$. Таким образом, можно описать процесс вычисления факториала так:

$$n! = \begin{cases} 1 & n = 0 \\ n * (n - 1)! & n > 0 \end{cases}$$

Например,

$3! = 3 * 2!$ — нам нужно вычислить $2!$

$2! = 2 * 1!$ — нам нужно вычислить $1!$

$1! = 1 * 0!$ — нам нужно вычислить $0!$

$0! = 1$ — получен результат.

Теперь начинаем «подниматься»: передаем значение **1** на «уровень» выше: $1 * 1 = 1$. Передаем это вычисленное значение **1** на следующий «уровень» вверх: $2 * 1 = 2$. Опять передаем это вычисленное значение **2** на следующий «уровень» вверх: $3 * 2 = 6$ — итоговый результат.

То есть, чтобы вычислить **3!**, нам надо вначале вычислить **2!**, и еще ранее вычислить **1!** и т.д.

Удобно процедуру вычисления факториала представить в виде функции, которая при условии **n>0**, будет вызывать саму себя с аргументом **(n-1)**, а при **n=0** вернет значение **1**.

```
def factorialCalculation(n):
    if n==0:
        return 1
    else:
        return n*factorialCalculation(n-1)

print(factorialCalculation(3))
```

После запуска нашей функции будет происходить следующее:

```
3* factorialCalculation(2)
  2* factorialCalculation(1)
    1* factorialCalculation(0)
      1
```

Рекурсивная функция должна обязательно содержать хотя бы один оператор **return**, возвращающий обычное значение (точка завершения рекурсии), чтобы не допустить бесконечное выполнение программы (зацикливание).

Рассмотрим следующую задачу: необходимо проверить, является ли слово палиндромом (т. е. читается в прямом и обратном направлении одинаково, например, слово «мадам» — палиндром).

```
def isStrPalind(myStr):
    if len(myStr) == 0:
        return True
    else:
        if myStr[0] == myStr[-1]:
            return isStrPalind(myStr[1:-1])
        else:
            return False

userStr = input("Your word: ")

if (isStrPalind(userStr.lower()) == True):
    print("{} is palindrome".format(userStr))
else:
    print("{} isn't palindrome".format(userStr))
```

Результат:

```
Your word: madam
madam is palindrome
```

Рисунок 38

```
Your word: Level
Level is palindrome
```

Рисунок 39

```
Your word: music
music isn't palindrome
```

Рисунок 40

Решение заключается в последовательном сравнении первого и последнего символа строки `myStr`, потом второго и предпоследнего символов (т. е. опять первого и последнего символа подстроки, полученной из исходной срезом `myStr[1:-1]`) и т. д. При этом дальнейшая проверка подстроки, получаемой из текущей, может запускаться, только если сравниваемые символы совпадают, иначе рекурсия останавливается, и мы получаем результат `False`.

В качестве условия для остановки рекурсии используем значение длины строки равной нулю (все символы строки проверены). В этом случае получаем результат `True`.

Рассмотрим еще один пример использования рекурсии. Допустим, что есть список чисел и нам необходимо написать функцию, которая найдет минимальное число в этом списке. Мы знаем, что в Python есть встроенная функция `min()`, которая находит минимум среди двух чисел, т.е можно процесс поиска представить в виде повторяющегося процесса сравнения двух чисел.

Начинаем с конца списка. Отделяем последний элемент и запускаем поиск на срезе без последнего элемента. На втором шаге отделяем предпоследний элемент и запускаем поиск на срезе уже без двух элементов и т.д.

На последнем шаге у нас в срезе останется только один элемент (первый элемент исходного списка). Известно, что если в списке одно число, то результат (минимум) — это число. Это будет точкой останова нашей рекурсии, и функция начнет «подниматься вверх», передавая полученный результат на верхний уровень.

В нашем примере первый результат мы получим на пятом уровне (это число `2`), который будет передан

на четвертый, где будет найден минимум между этим число 2 и «ожидающим» возврата рекурсии числом 4. Найденный минимум (2) будет передан на третий уровень, где будет сравниваться с очередным «ожидающим» возврата рекурсии числом 1 и т.д.

2	4	1	8	3		
					min(1, 3) ?	1
	2	4	1	8		
					min(1, 8) ?	1
		2	4	1		
					min(2, 1) ?	1
			2	4		
					min(2, 4) ?	2
				2		

Рисунок 41

```
def findMin(numberList):
    if len(numberList)>1:
        return min(findMin(numberList[:-1]),
                    numberList[-1])
    else:
        return numberList[0]

myNumbers=[2,4,1,8,3]

print("min={}".format(findMin(myNumbers))) #min=1
```

Основным недостатком рекурсии является то, что при каждом запуске функции (рекурсивном вызове) создается копия параметров функции и ее локальных переменных, что вызывает дополнительные затраты памяти и процессорного времени.

3. Функциональное программирование

Что такое функциональное программирование?

Функциональное программирование — это такой подход к разработке программного обеспечения, при котором основная логика задачи (алгоритм) представляется в виде набора функций, которые реализуют отдельные шаги общего алгоритма.

Например, задачу нахождения успешных студентов в группе (со средним баллом более 60) удобно разбить на такие шаги (отдельные подзадачи):

- ввод данных об оценках студентов группы;
- подсчет среднего балла каждого студента по всем предметам;
- нахождение студентов со средним баллом более 60 и формирование списка успешных студентов;
- вывод списка успешных студентов с их средним баллом.

Каждую из этих задач легко реализовать и просто тестировать в виде отдельной функции.

Как мы уже знаем, функции могут быть легко вызваны и повторно использованы при необходимости в любой точке общей программы. Это обеспечивает легкость в управлении кодом. Также работу над проектом можно распараллелить между несколькими разработчиками, каждый из которых будет реализовывать свою часть логики (в виде отдельных функций).

Python использует концепцию функционального программирования, предоставляя нам большой набор встроенных функций. Также у нас есть возможность применять особенности функции, как объектов первого класса, передавать их в другие функции в качестве параметров.

Функциональное программирование предполагает разработку с использованием чистых функций.

Чистая функция — это такая функция, которая реализует связь между входными и выходными данными (обеспечивает преобразование входных значений в выходные по некоторому алгоритму).

Чистая функция:

- всегда выдает одинаковый ответ при одних и тех же входных параметрах;
- не имеет скрытого (или неявного) ввода-вывода или других побочных эффектов, т. е. работает только с входными значениями (параметрами) без участия внешних данных.

Благодаря этим свойствам в программах, созданных с использованием методик функционального программирования, легко выявляются и исправляются ошибки.

Помимо этого, преимущества функциональная парадигма популярна в программировании также из-за высокого уровня абстракции: мы описываем с помощью вызовов функций нужный результат, а не шаги по его достижению. Таким образом, код основной программы становится короче, но смысл каждой строки более прозрачный.

Дополнительную прозрачность кода также обеспечивают и сами чистые функции, которые исключают возможность побочных эффектов и промежуточных значений.

Значимыми в функциональном программировании являются возможности функций как объектов первого класса (с которыми мы уже знакомы): функция может использовать другую функцию в качестве своего аргумента и может вернуть другую функцию, как результат своей работы (вызывающей стороне, например, в основную программу).

Давайте вспомним, как это работает: присвоим функцию переменной (создав новую ссылку на это же код функции, но с другим именем) и используем эту переменную для вызова функции:

```
def myGreeting1():  
    print("Good morning! Have a nice day!")  
  
sayGoodMorning=myGreeting1  
sayGoodMorning()
```

Мы можем «увеличить масштаб» этой идеи: поместим имена функций в список и в цикле вызовем каждую:

```
def myGreeting1():  
    print("Good morning! Have a nice day!")  
  
def myGreeting2():  
    print("Good day! Nice to see you!")  
  
def myGreeting3():  
    print("Hey! Long-time no see.")  
  
def myGreeting4():  
    print("Good night! See you tomorrow.")
```

```
myGreetingsList=(myGreeting1, myGreeting2,
                 myGreeting3, myGreeting4)

for myGreeting in myGreetingsList:
    myGreeting()
```

Результат:

```
Good morning! Have a nice day!
Good day! Nice to see you!
Hey! Long-time no see.
Good night! See you tomorrow.
```

Рисунок 42

Теперь передадим функцию, как аргумент другой функции, чтобы спросить имя пользователя, которому предназначено приветствие:

```
def greetingRecipient(greetFunction):
    greetRecipient=input("name?")
    print("Dear, ", greetRecipient)

greetFunction()
```

И поприветствуем каждого пользователя по-разному, запустив нашу функцию `greetingRecipient()` в цикле над списком функций — приветствий:

```
for myGreeting in myGreetingsList:
    greetingRecipient(myGreeting)
```

Результат:

```
name?Hanna
Dear,  Hanna
Good morning! Have a nice day!
name?Joe
Dear,  Joe
Good day! Nice to see you!
name?Bob
Dear,  Bob
Hey! Long-time no see.
name?Jane
Dear,  Jane
Good night! See you tomorrow.
```

Рисунок 43

Такой подход называется функциональной композицией.

Точно так же, как мы передавали функцию другой функции в качестве аргумента, мы можем вернуть функцию в качестве возвращаемого значения.

Создадим функцию, которая возвращает нужную функцию из списка в зависимости от кода времени суток:

```
def checkTimeOfDay():
    while True:
        timeOfDay=input("Input time of day (M-morning;
                        D-afternoon;E- evening;N-night):")
        if timeOfDay=="M":
            return myGreetingsList[0]
        elif timeOfDay=="D":
            return myGreetingsList[1]
        elif timeOfDay=="E":
            return myGreetingsList[2]
        elif timeOfDay=="N":
```

```

        return myGreetingsList[3]
    else:
        print("Wrong input!")

```

Теперь можно в цикле запустить тестирование, например, для четырех пользователей запросим их имена и код времени суток, просто вызвав нашу функцию `greetingRecipient()` и передав ей в качестве аргумента вызов функции `checkTimeOfDay()`:

```

for i in range(3):
    greetingRecipient(checkTimeOfDay())

```

Результат:

```

Input time of day (M-morning;D-afternoon;E- evening;N-night):0
Wrong input!
Input time of day (M-morning;D-afternoon;E- evening;N-night):M
name?Hanna
Dear,  Hanna
Good morning! Have a nice day!
Input time of day (M-morning;D-afternoon;E- evening;N-night):D
name?Bob
Dear,  Bob
Good day! Nice to see you!
Input time of day (M-morning;D-afternoon;E- evening;N-night):E
name?Jane
Dear,  Jane
Hey! Long-time no see.

```

Рисунок 44

Таким образом, Python предоставляет нам все необходимое для использования парадигмы функционального программирования.

Анонимные функции `lambda`

Мы уже знаем, как создавать функции с помощью ключевого слова `def`. Но помимо этого подхода Python позволяет создавать функции в виде выражений.

В этом случае используется ключевое слово `lambda` и такой синтаксис объявления `lambda`-функций:

```
lambda аргумент1, аргумент2, ..., аргументN: выражение
```

- `аргумент1, аргумент2, ..., аргументN` — имена переменных-аргументов, которые будут использоваться в выражении
- `выражение` — само выражение, реализующее некоторое вычисление (или действие).

Например, нам нужно создать простую функцию, которая добавляет к некоторому числу значение 10.

Конечно, можно реализовать эту функцию с помощью ключевого слова `def` (как мы делали ранее):

```
def add10(x):
    return x+10
```

Допустим, что у нас есть список чисел, которые нужно преобразовать с помощью этой функции:

```
myNumbers=[2, 2.5, 4.56,23]

for num in myNumbers:
    print(add10(num))
```

Фактически наша функция содержит только одно-единственное выражение внутри.

Удобнее здесь использовать лямбда-функцию:

```
myNumbers=[2, 2.5, 4.56,23]

newNum=lambda x:x+10

for num in myNumbers:
    print(newNum(num))
```

Также можно создавать лямбда-функцию прямо в том же месте, где она будет использована:

```
myNumbers=[2, 2.5, 4.56,23]

for num in myNumbers:
    print((lambda x:x+10)(num))
```

Особенно полезным такой подход является в ситуациях, когда в дальнейшем не предполагается больше использовать эту функцию.

Проанализировав общий синтаксис `lambda`-выражения и примеры выше, можно выделить такие отличия `lambda`-функций от `def`-подхода:

- `lambda`-выражения записываются в одну строку (при попытке разместить выражение во вторую строку мы получим ошибку);
- `lambda`-выражения создают функции, у которых нет имени (анонимные функции), которые можно присвоить переменной (как в примере выше) или использовать прямо в месте создания;
- `lambda`-функция содержит только одно выражение, результат вычисления которого — это результат, воз-

вращаемый `lambda`-функцией, т. е. ключевое слово `return` не нужно использовать;

- `lambda`-функции предназначены для реализации (вычисление) более простых фрагментов кода по сравнению с обычными функциями.

Так как `lambda`-функции являются выражениями, то, значит, их можно использовать там, где использование `def` не допускается — внутри литералов или внутри вызовов функций.

Рассмотрим следующий пример: у нас есть список студентов со средними баллами, причем информация о каждом студенте также представляет собой список, первым элементом которого является имя студента, а вторым — его средний балл.

Нам необходимо отсортировать список студентов по имени. Для сортировки будем использовать встроенную функцию `sorted`, первым аргументом которой является наш список, а вторым — ключ сортировки (имя студента в нашей задаче).

Для задания ключа сортировки нам нужно выделить из внутреннего списка имя, что мы делаем с помощью `lambda`-функции.

```
students = [['Bob', 70],
            ['Jane', 80],
            ['Andy', 50]
            ]
print(students)

sortedSts=sorted(students, key=lambda x: x[1])
print(sortedSts)
```


Результат:

```
[[ 'Bob', 70], [ 'Jane', 80], [ 'Andy', 50]]
[[ 'Andy', 50], [ 'Bob', 70], [ 'Jane', 80]]
```

Рисунок 45

Теперь допустим, что пользователь вводит цены двух товаров в одной валюте, а нам нужно вывести их в другой по курсу и учесть скидку.

```
grnToDollar=28.2
discount=0.15

priceDol=lambda x:x/grnToDollar*(1-discount)

priceGrn1=float(input("Input price1 in grn:"))
print("price: {:.2f}$".format(priceDol(priceGrn1)))

priceGrn2=float(input("Input price2 in grn:"))
print("price: {:.2f}$".format(priceDol(priceGrn2)))
```

Результат:

```
Input price1 in grn:3000
price: 90.43$
Input price2 in grn:1245
price: 37.53$
```

Рисунок 46

Здесь мы используем одну и ту же lambda-функцию два раза, сохранив ее как выражение в переменной `priceDol`, в которую далее передаем разные цены как аргументы.

Наша следующая задача- реализовать сборку строки с именем и фамилией пользователя, которые должны

начинаться с заглавной буквы. Выполним ее также с использованием `lambda`-функции, но уже с двумя аргументами:

```
userName = lambda firstName,
    lastName: "Full user's name: {} {}".
    format(firstName.title(),
    lastName.title())

firstUserName=input("Input your first name:")
lastUserName=input("Input your last name:")
print(userName(firstUserName, lastUserName))
```

Результат:

```
Input your first name:joe
Input your last name:BLACK
Full user's name: Joe Black
```

Рисунок 47

На этих примерах можно заметить еще одно преимущество использования `lambda`-функции — близость программного кода. Анонимные функции (код, реализующий их логику), размещаются рядом в программе, что повышает читабельность кода и не требует использования дополнительных имен функций (как в случае с использованием `def`).

Используя `lambda`-функции мы также уменьшаем общее количество имен в нашей программе, снижая возможные ситуации с конфликтами имен в этом файле кода.

Функции высших порядков

Одной из самых важных и полезных концепций функционального программирования является идея функций высших порядков.

Функция называется функцией высшего порядка, если она принимает другую функцию (функции) в качестве аргументов и (или) возвращает функцию в результате своей работы.

Мы уже знаем, что в Python функции — это объекты первого класса, поэтому возможность принимать в качестве аргумента другую функцию и/или возвращать функцию как результат работы активно используется в Python-программировании.

Функции высшего порядка обеспечивают более гибкий, легкий и компактный способ реализации логики программы с помощью функций.

Допустим, что нам необходимо получить возраст студентов, чьи года рождения хранятся в списке. Используя подход с применением функций высших порядков, эту задачу можно реализовать таким образом:

```
studBirthYear = [2000, 1997, 2002, 1999, 2007]
studAges=list(map(lambda x:2022-x,studBirthYear))

print(studAges) # [22, 25, 20, 23, 15]
```

А без функций высших порядков код будет выглядеть так:

```
studBirthYear = [2000, 1997, 2002, 1999, 2007]
studAges=[]

for year in studBirthYear:
    studAges.append(2022-year)

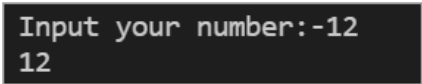
print(studAges) #[22, 25, 20, 23, 15]
```

В первом решении программисту не нужно заранее создавать пустой список для хранения возраста, не нужно вспоминать, какой метод объекта `list` (список) добавляется новый элемент в конец списка, не нужно даже организовывать цикл.

Многие из рассмотренных нами ранее встроенных Python-функций являются функциями высшего порядка. Например, встроенная функция `print()`. Пусть нам нужно вывести модуль числа, вводимого пользователем:

```
userNumber=int(input("Input your number:"))  
print(abs(userNumber))
```

Здесь встроенная функция `print()` принимает как аргумент другую встроенную функцию для получения абсолютного значения числа — `abs()`, которая в свою очередь обрабатывает число пользователя.



```
Input your number:-12  
12
```

Рисунок 48

Встроенная функция `sorted()` также является функцией высокого порядка. Мы можем задавать ключ пользовательской сортировки с помощью функции (например, с помощью той же встроенной функции `abs()` для получения абсолютного значения числа)

```
myNumbers=[1.3, -2.3, -12, 11, 0.45]  
  
print(myNumbers)  
print(sorted(myNumbers, key=abs))
```

Результат:

```
[1.3, -2.3, -12, 11, 0.45]
[0.45, 1.3, -2.3, 11, -12]
```

Рисунок 49

Рассмотрим еще один пример: пусть нам необходимо написать функцию, которая в зависимости от выбора пользователя реализует либо процесс его аутентификации, либо смену пароля, который назначен по умолчанию, на новый. Данную функцию удобно реализовать как функцию высшего порядка, назовем ее `userAction()`, которая будет запускать (возвращать) нужное действие (функцию) для определенного логина и пароля пользователя.

Для реализации каждой из перечисленных возможностей создадим отдельные функции (`userLogIn()`, `changePass()`), которые перед выполнением нужного действия проверяют еще соответствие логина и пароля.

```
adminPassw='111'
studentPassw='222'

def userLogIn(userLog, userPass):
    if (userLog.lower() == 'admin') and
        (userPass == adminPassw):
        print("Welcome, admin")
    elif (userLog.lower() == 'student') and
        (userPass == studentPassw):
        print("Welcome, student")
    else:
        print("Wrong data")

def changePass(userLog, userPass):
    global adminPassw
```

```

global studentPassw
if (userLog.lower() == 'admin') and
    (userPass == adminPassw):
    adminPassw = input("Input new password
                        for admin:")
elif (userLog.lower() == 'student') and
    (userPass == studentPassw):
    studentPassw = input("Input new password
                        for student:")
else:
    print("Wrong data")

def userAction(login, passw, action):
    return action(login, passw)

userL=input("Login:")
userP=input("Password:")

userAnsw=input("1: LogIn; 2: Change password ")

if userAnsw=='1':
    userAction(userL, userP, userLogIn)
elif userAnsw=='2':
    userAction(userL, userP, changePass)
else:
    print("Wrong data")

```

Результаты:

```

Login:Admin
Password:111
1: LogIn; 2: Change password 1
Welcome, admin

```

Рисунок 50

```

Login:STUDENT
Password:222
1: LogIn; 2: Change password 2
Input new password for student:333

```

Рисунок 51

Теперь допустим, что нам нужно реализовать функцию, которая на основе имени пользователя создает его логин (добавляя к имени строку «user»). При этом пользователь может выбрать, в каком регистре добавлять его имя: в верхнем (например, «userkate») или нижнем («userKATE»).

Для этого создадим две функции для формирования первого и второго варианта логина: `nameUpper(name)`, `nameLower(name)`. Также у нас будет функция высшего порядка `makeLog(userName, maker)`, которая принимает имя пользователя и функцию, которая будет его обрабатывать выбранным способом:

```

def nameUpper(name):
    return 'user'+name.upper()

def nameLower(name):
    return 'user'+name.lower()

def makeLog(userName, maker):
    return maker(userName)

user=input("Input your name:")
userAnsw=input("Select login-maker: 1-lower case;
               2 -upper case")

if userAnsw=='1':
    print(makeLog(user,nameLower))

```

```
elif userAnsw=='2':
    print(makeLog(user,nameUpper))
else:
    print("Wrong input!")
```

Результаты:

```
Input your name:Bob
Select login-maker: 1-lower case;2 -upper case1
userbob
```

Рисунок 52

```
Input your name:jane
Select login-maker: 1-lower case;2 -upper case2
userJANE
```

Рисунок 53

Функции `map()`, `filter()`, `zip()`

В предыдущем вопросе мы рассмотрели понятие функций высших порядков, примеры создания собственных функций такого класса и несколько примеров использования встроенных функций высших порядков.

Однако во всех рассмотренных случаях функции запускались только единожды с одним набором параметров. А что, если нам необходимо запустить одну и ту же функцию для каждого элемента некоторого набора (например, выполнить какое-то преобразование каждого элемента списка)? Или нужно выбрать из набора элементы, удовлетворяющие некоторому условию?

Например, у нас есть список логинов пользователей:

```
userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']
```

Нам нужно перевести каждый логин из списка в нижний регистр. Конечно, данную задачу можно выполнить с помощью операторов цикла и ветвления. В нашем случае, например, таким образом:

```
userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']

userLogsLower=[]

for log in userLogs:
    userLogsLower.append(log.lower())

print(userLogsLower)
#['123user45', 'userstudent', '56use3', 'user-23',
  'adminus']
```

Но так как перечисленные задачи являются очень распространенными, то в Python предусмотрели встроенные функции высшего порядка, позволяющие реализовать указанную функциональность простым и компактным способом.

Встроенная функция `map()` позволяет применять нужную функцию к каждому элементу коллекции (итерируемого объекта, например, списка). Возвращаемый результат — новый итератор (объект `map`), который потом можно передать, например, во встроенную функцию `list()` для создания списка результатов.

Общий синтаксис функции `map()`:

```
map (function, iterable1, [iterable2,... iterableN])
```

function — функция-аргумент, в которую `map()` передает на «обработку» каждый элемент итерируемого объекта `iterable1`.

Функция `map()` может принимать более одного итерируемого объекта при необходимости, в этом случае в таком случае наша функция-аргумент должна принимать такое количество аргументов, которое соответствует количеству итерируемых объектов (рассмотрим подобный пример далее).

Давайте выполним сравнение реализаций кода с использованием цикла `for` и с применением функции `map()` для перевода логинов пользователей из списка в нижних регистр.

Первый вариант реализации на основе цикла `for`:

```
userLogs=['Admin','STUDENT','teacher','User']
userLogsLow=[]

print(userLogs)
for userLog in userLogs:
    userLogsLow.append(userLog.lower())
print(userLogsLow)
```

Результат:

```
['Admin', 'STUDENT', 'teacher', 'User']
['admin', 'student', 'teacher', 'user']
```

Рисунок 54

Второй вариант реализации с применением функции `map()`:

```
userLogs=['Admin','STUDENT','teacheR','User']
userLogsLow=[]

print(userLogs)
userLogsLow=list(map(str.lower,userLogs))
print(userLogsLow)
```

Обеспечивает такой же результат, но код при этом более компактный:

```
['Admin', 'STUDENT', 'teacheR', 'User']
['admin', 'student', 'teacher', 'user']
```

Рисунок 55

Примечание: обратите внимание, что в качестве имени функции-аргумента мы передаем `str.lower`, а не просто `lower`, т.к. используемая в примере функция-аргумент, является методом объекта `str`.

Рассмотрим еще один пример использования `map()` со встроенными функциями.

Пусть у нас есть список числовых значений в формате строк (например, полученный от пользователя с помощью функции `input()`). Для дальнейшей работы нам необходим именно список чисел. Выполним преобразование исходного списка значений в числа с помощью `map()`:

```
myValues=['2.3','12','45.67','-3']
print(myValues)
```

```
myNumbers=list(map(float,myValues))
print(myNumbers)
```

Результат:

```
['2.3', '12', '45.67', '-3']
[2.3, 12.0, 45.67, -3.0]
```

Рисунок 56

Также активно используется `map()` совместно с `lambda`-функциями. Например, нам нужно из численных значений в формате строк получить первую цифру числа в целочисленном формате.

Реализация с использованием `lambda` и `map()` будет компактной и понятной:

```
myNumbers=['12.3','12','45.67','30']
print(myNumbers)

myDigits=list(map(lambda x:int(x[0]),myNumbers))
print(myDigits)
```

Результат:

```
['12.3', '12', '45.67', '30']
[1, 1, 4, 3]
```

Рисунок 57

Полезным и удобным является использование `map()` с функциями, определенными пользователем с помощью `def`.

Например, у нас есть список типов пицц, но для формирования меню их нужно преобразовать, добавив к названию типа слово «*Pizza*».

Добавление символа пробела и слова «*Pizza*» реализуем через отдельную функцию, которую передадим как аргумент функции `map()` для формирования списка названий:

```

pizzaTypes=['Neapolitan','Sicilian','Detroit',
            'New York-Style']
print(pizzaTypes)

def modifyPizzaTypes(pizzaType):
    return pizzaType+' Pizza'

pizzaNames=list(map(modifyPizzaTypes,pizzaTypes))
print(pizzaNames)

```

Результат:

```

['Neapolitan', 'Sicilian', 'Detroit', 'New York-Style']
['Neapolitan Pizza', 'Sicilian Pizza', 'Detroit Pizza', 'New York-Style Pizza']

```

Рисунок 58

Как было отмечено выше, `map()` может принимать более одного итерируемого объекта. В такой ситуации мы должны предварительно создать такую функцию-аргумент для `map()`, которая принимает количество аргументов, соответствующее числу итерируемых объектов в `map()`.

Рассмотрим на примере. У нас есть список чисел и список степеней, в которые нужно возвести каждое из чисел первого списка.

```

numbersList1 = [10, 20, 30]
numbersList2 = [1, 2, 3]
result = map(lambda a, b: a**b, numbersList1,
             numbersList2)
print(list(result)) #[10, 400, 27000]

```

С помощью `map()` мы применили `lambda`-функцию, которая принимает два аргумента (число и его степень), для наших двух списков: чисел и их степеней. Таким образом обеспечив, что количество аргументов `lambda`-функции (два: `a, b`) совпало с количеством списков (два: `numbersList1, numbersList2`).

Для задач фильтрации элементов коллекции по некоторому условию и их отбора в новую коллекцию в Python предусмотрена функция `filter()`.

Функция `filter()` принимает два аргумента: функцию, которая реализует (описывает) условие фильтрации и возвращает логические значения (`true` или `false`), и итерируемый объект, каждый элемент которого будет проверяться условием фильтра. Если в результате такой проверки элемента функция-проверка вернет `true`, то он будет включен в результат работы фильтра.

Рассмотрим на примере: у нас есть список с ценами на товары. Допустим, что нужно выбрать цены, значения которых больше 10 долларов.

```
prices=[100.45, 8.56, 5, 234, 45, 87, 567]

expensive=list(filter(lambda x:x>=10, prices))

print(expensive) #[100.45, 234, 45, 87, 567]
```

Теперь нам нужно выбрать такие логины пользователей, которые содержат слово «*user*». Проверку на наличие слова «*user*» в логине реализуем с помощью отдельной функции `checkLog(user)`, которую передадим функции `filter()`.

```

userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']

def checkLog(user):

    if user.lower().find('user')!=-1:
        return True
    else:
        return False

selectedUser=list(filter(checkLog,userLogs))

print(selectedUser) # ['123user45', 'USERstudent',
                     'user-23']

```

Также к очень популярным и полезным функциям относится функция `zip()`, которая активно используется в задачах перебора данных и объединения данных из нескольких коллекций в один набор.

Например, у нас есть два списка: список логинов и паролей пользователей. И нам необходимо вывести их в формате «логин — пароль», т. е. одним циклом пройти по двум спискам. Это легко организовать, используя функцию `zip()`:

```

userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']

userPass=['111', 'abc', '2345', '45fg', 'dffdg']

for log, passw in zip(userLogs, userPass):
    print("login: {} — password: {}".format(log, passw))

```

```
login: 123user45 - password: 111
login: USERstudent - password: abc
login: 56use3 - password: 2345
login: user-23 - password: 45fg
login: adminUs - password: dffdg
```

Рисунок 59

Функция `zip()` принимает итерируемые коллекции в качестве аргументов и возвращает итератор. Этот итератор генерирует серию кортежей, которые содержат элементы из каждого объекта-аргумента функции `zip()`.

Общий синтаксис:

```
zip([iterator1, iterator2,.. iteratorN])
```

Функция `zip()` может не содержать ни одного итератора, один или несколько.

Допустим, что нам просто нужно объединить два списка в набор пар элементов:

```
list1 = [1, 2, 3, 4, 5]
list2 = ['a', 'b', 'c', 'd', 'e']

print(zip(list1, list2))
print(type(zip(list1, list2)))
print(list(zip(list1, list2)))
```

Вначале посмотрим, что фактически возвращает функция `zip()` — это итератор кортежей (объект — *zip object*, *class "zip"*).

Далее с помощью функции `list()` преобразуем его содержимое в список кортежей:


```
<zip object at 0x000001ED896A0DC0>
<class 'zip'>
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Рисунок 60

Рассмотрим детальнее полученный результат: набор кортежей, создаваемый `zip()` формируется по правилу: *i*-й кортеж содержит *i*-й элемент из каждого списка.

Например, первый кортеж `(1, "a")` состоит из первых элементов каждого из наших двух списков. Таким образом, если функции `zip()` передать три списка, то она обработает их следующим образом:

```
list1=[1,2,3,4,5]
list2=['a','b','c','d','e']
list3=['!','*','#','-','+']

print(list(zip(list1,list2,list3)))
#[(1, 'a', '!'), (2, 'b', '*'), (3, 'c', '#'),
  (4, 'd', '-'), (5, 'e', '+')]
```

При одном итерируемом объекте-аргументе `zip()` вернет набор кортежей, в каждом из них будет только один элемент. Также можно использовать `zip()` без аргументов. Результатом будет пустой `zip`-объект.

```
list1=[1,2,3,4,5]

print(list(zip(list1)))
#[(1,), (2,), (3,), (4,), (5,)]

print(list(zip())) #[]
```

`zip()` может принимать любые типы итерируемых объектов, такие как файлы, списки, кортежи, словари и т. д.

Иногда нам приходится обрабатывать неравные по длине наборы. Итератор останавливается, когда самая короткая входная коллекция исчерпана:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c', 'd', 'e']

print(list(zip(list1, list2)))
#[(1, 'a'), (2, 'b'), (3, 'c')]
```

Модуль `functools`

Мы уже знакомы с такими полезными встроенными функциями высшего порядка как `map()`, `reduce()` и `filter()`. В Python есть модуль `functools`, который облегчает и упрощает работу с функциями высшего порядка. Функция `reduce()`, с которой мы работали ранее также входит в `functools`.

Благодаря таким функциям мы можем повторно применять или расширять возможности уже созданных функций, не переписывая их. Это помогает нам повторно использовать код в других задачах довольно просто и удобно.

Одной из наиболее востребованных функций из набора `functools` является функция `partial`, с помощью которой мы можем создать новую версию существующей функции (производную функции).

Частичные функции (*partial functions*) — это такие производные функции, которые имеют некоторые заранее заданные входные параметры. Например, если функция

принимает два параметра («x» и «y»), то из нее можно создать частичную функцию, у которой («x» — предварительно заданный аргумент). Далее мы можем вызывать эту частичную функцию только с одним параметром «y».

Давайте рассмотрим простой пример, чтобы проиллюстрировать все эти нюансы.

Допустим, у нас есть функция, которая дублирует строку указанное количество раз:

```
def myltipleText(myStr, n):
    return myStr*n
```

А что, если нам нужно и не единожды реализовывать повтор строки два раза, потом еще три раза. Например, обеспечить такой вывод на основе строки «Python»:

```
PythonPython
PythonPython
PythonPython
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
```

Можно создать отдельные функции, которые внутри себя будут вызывать `myltipleText()` с нужным параметром (числом повторений):

```
def myltipleText2(myStr):
    return myltipleText(myStr, 2)

def myltipleText3(myStr):
    return myltipleText(myStr, 3)
```

Однако более удобным для таких моментов является использование `partial()`:

```
from functools import partial

def myltipleText(myStr, n):
    return myStr*n

doubleMyltipleText = partial(myltipleText, 2)
tripleMyltipleText = partial(myltipleText, 3)

print(doubleMyltipleText("Python")) #PythonPython
print(tripleMyltipleText("Python")) #PythonPythonPython
```

Как видно из нашего примера, мы с помощью `partial()` устанавливаем для аргумента `n` функции `myltipleText()` значение по умолчанию два (в первом случае) и три во втором.

В этом примере порядок аргументов принципиального значения не имеет. Однако есть ситуации, где изменение порядка повлечет некорректный результат работы функции.

Например:

```
from functools import partial

def SuperCalculation(a,b,c):
    return a**b+c

result = partial(SuperCalculation,b=2,c=5)
print(result(4)) #21
```

В этом примере порядок следования аргументов влияет на правильность вычислений. Поэтому, чтобы

зафиксировать переменные-аргументы мы использовали ключевые слова.

Еще одна полезная возможность, предоставляемая модулем `functools` — это кэширование, значительно ускоряющее работу приложения и снижающее нагрузку на вычислительные ресурсы.

Рассмотрим средства `functools`, которые позволят нам кэшировать результаты наших функций.

`@lru_cache()` «оборачивает» нашу функцию с переданными в нее аргументами и «запоминает» результат, который функция вернула после вызова конкретно с этими аргументам.

Ранее мы уже создавали рекурсивную функцию для вычисления факториала числа:

```
def factorialCalculation(n):  
  
    if n==0:  
        return 1  
    else:  
        return n*factorialCalculation(n-1)
```

Чтобы кэшировать результаты вызова функции `factorialCalculation()` мы вначале импортируем `lru_cache` из модуля `functools` и добавим вызов `@lru_cache()` перед функцией `factorialCalculation()`:

```
from functools import lru_cache  
  
@lru_cache  
  
def factorialCalculation(n):
```

```

    if n==0:
        return 1
    else:
        return n*factorialCalculation(n-1)

factorialCalculation(3)
factorialCalculation(6)
factorialCalculation(10)

```

При первом вызове `factorialCalculation(3)` будет выполнено четыре запуска функции, но уже при втором вызове `factorialCalculation(6)` интерпретатор использует «запомненный» результат от `factorialCalculation(3)` и выполнит не семь, а только три, а третий вызов `factorialCalculation(10)` вместо десяти запусков выполнит только четыре.

Однако, если, у нашей функции `f()` несколько аргументов, заданных с помощью ключевых слов, то следует помнить, что вызовы `f(a = 5, b = 10)` и `f(b = 10, a = 5)` различаются по порядку следования ключевых аргументов и могут иметь две отдельные записи в кэше.

Кэширование — это полезный и эффективный способ, с помощью которого можно ускорить работу приложения и значительно снизить нагрузку на вычислительные ресурсы.

Используя кеширование, мы можем хранить последнюю или часто используемую информацию в тех местах памяти, которые вернут ее при необходимости быстрее или дешевле (с точки зрения вычислительной нагрузки), чем оригинальный источник информации.

Например, мы разрабатываем приложение, которое собирает и отображает последние новости из многих

разных источников. При переходе между новостными источниками в списке наше приложение должно загрузить новости из него и показать их пользователю. Очень часто пользователи повторно переходят из одного источника в другой и обратно. Без кеширования уже ранее загруженных данных из новостного источника нам придется при каждом таком переключении повторно загружать из источника большую порцию одного и того же контента. Такая ситуация не просто серьезно замедлит работу нашего приложения, но и спровоцирует дополнительную нагрузку на новостной сервер, на котором хранятся статьи. Более правильным и эффективным способом будет хранить уже ранее загруженный контент в локальной копии-кеше. И при обращении пользователя (перед загрузкой контента) проверять, есть ли такая новость в кеша. Если есть, то загружать из кеша, нет — обращаться к новостному серверу и выполнять загрузку контента с него.

Теперь рассмотрим особенности синтаксиса и функции `reduce()` — функции «сокращения», которая в отличие от рассмотренных выше функций возвращает не коллекцию, а одно значение.

Аргументами `reduce()` являются некоторая функция и последовательность (например, список). Результат — одно значение, вычисленное на основании всех элементов последовательности.

Для лучшего понимания особенностей работы и преимуществ использования функции `reduce()` рассмотрим решение задачи нахождения суммы элементов списка

обычным способом, используя цикл `for`. Затем реализуем ее с помощью функции `reduce()`.

Итак, нам требуется вычислить сумму элементов списка $[a_1, a_2, \dots a_n]$:

$$S = a_1 + a_2 + \dots + a_n.$$

Решение сводится к последовательному вычислению промежуточных сумм: $S_0 = 0$ (начальное значение, устанавливается перед вычислениями).

$$S_1 = S_0 + a_1$$

$$S_2 = S_1 + a_2$$

.....

$$S_n = S_{n-1} + a_n = a_1 + a_2 + \dots + a_n.$$

Вычисление значения S_n представляет собой искомую сумму S . Значение промежуточных сумм S_1, \dots, S_{n-1} не требуется сохранять, поэтому последовательность вычислений, представленную выше, можно сформулировать в виде общей формулы:

$$S = S + a_i,$$

где a_i — слагаемое на i — том шаге.

Таким образом, вычисление суммы сводится к ее накоплению в переменной S на каждом шаге цикла.

Например, в нашем списке находится последовательность чисел от 1 до 5 включительно.

Наши слагаемые:

$$a_1 = 1$$

$$a_2 = 2$$

$$a_3 = 3$$

$$a_4 = 4$$

$$a_5 = 5$$

$$S = a_1 + a_2 + a_3 + a_4 + a_5$$

$$S = 1 + 2 + 3 + 4 + 5$$

Перед началом вычисления суммы еще нет, т. е. ее значение равно нулю.

$$S_0 = 0$$

$$S_1 = S_0 + a_1 = 0 + 1 = 1$$

$$S_2 = S_1 + a_2 = 1 + 2 = 3$$

$$S_3 = S_2 + a_3 = 3 + 3 = 6$$

$$S_4 = S_3 + a_4 = 6 + 4 = 10$$

$$S_4 = S_3 + a_4 = 10 + 5 = 15$$

Организуем цикл по накоплению суммы:

```
numbers=[1,2,3,4,5]
sumNum=0
for num in numbers:
    sumNum+=num

print("sum is {}".format(sumNum)) #15
```

Мы создаем переменную `sumNum` и устанавливаем ее равной `0`. Затем мы перебираем наш список чисел `numbers`, используя цикл `for`, и добавляем каждое число к результату предыдущей итерации (аккумулируя результат). После прохода по всему списку чисел сумма (аккумулятор) будет равна `15`.

Мы можем выполнить рассмотренную выше задачу, используя функцию `reduce()` вместо цикла `for`.

Синтаксис:

```
reduce (functionName, iterableObj [, initializer])
```

Функция может принимать три аргумента, два из которых являются обязательными: функция и итерируемый объект (например, список). Третий аргумент, который является инициализатором (начальным значением), является необязательным.

Изначально функция `functionName` вызывается с первыми двумя элементами последовательности `iterableObj` и возвращается результат. Затем `functionName` вызывается снова с результатом, полученным на шаге 1, и следующим значением в последовательности `iterableObj`. Этот процесс повторяется до тех пор, пока в последовательности `iterableObj` есть элементы.

Когда предоставляется третий аргумент (начальное значение, `initializer`), то функция `functionName` вызывается с начальным значением `initializer` и первым элементом из последовательности `iterableObj`.

Наша сумма чисел в списке:

$$1 + 2 + 3 + 4 + 5.$$

может быть записана так:

$$((((0 + 1) + 2) + 3) + 4) + 5).$$

Ноль здесь — то самое начальное значение (`initializer`), которое не добавляет к сумме ничего, но может служить отправной точкой. Также если будет передан пустой список, то функция `reduce()` вернет его значение.

Функция `reduce()` находится в модуле `functools`. Следовательно, чтобы ее использовать, нам нужно либо импортировать весь модуль `functools`,

```
import functools
```

либо мы можем импортировать только функцию `reduce()` из `functools`:

```
from functools import reduce
```

Теперь найдем сумму наших чисел, используя функцию `reduce()` и созданную нами отдельно функцию суммирования двух чисел:

```
import functools
from functools import reduce

numbers=[1, 2, 3, 4, 5]

def mySum(x, y):
    return x+y

result1 = reduce(mySum, numbers)
print("sum is {}".format(result1)) #15

result2 = reduce(mySum, numbers, 0)
print("sum is {}".format(result2)) #15
```

Второе применение функцию `reduce()` прошло с использованием третьего аргумента, начального значения (равного нулю, идея та же, как мы ранее в цикле устанавливали начальное значение `sumNum=0`).

Если мы применим функцию `reduce()` с пустой коллекцией, не указав начальное значение, то возникнет ошибка:

```
numbersEmpty=[]
result3=reduce(mySum, numbersEmpty)
```

`reduce()` of empty iterable with no initial value

Рисунок 61

А вот при заданном начальном значении (`initializer`), результат по работе с пустым списком вернет сам `initializer` (ноль в нашем примере):

```
result4=reduce(mySum, numbersEmpty, 0)
print("sum is {}".format(result4)) #0
```

Мы могли бы и не объявлять функцию `mySum()` отдельно, а использовать `lambda`-функцию просто внутри вызова `reduce()`:

```
numbers=[1,2,3,4,5]

result5=reduce(lambda x, y:x + y, numbers, 0)
print("sum is {}".format(result5)) #15

import functools
from functools import reduce

words=['Python', 'is', 'cool']

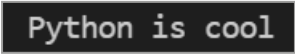
sentence=reduce(lambda x, y:x + " " + y, words, "")
print(sentence)
```

Рассмотрим еще один пример использования функции `reduce()`. Допустим, что у нас есть список слов и нам нужно собрать их в предложение.

```
import functools
from functools import reduce

words=['Python', 'is', 'cool']
sentence=reduce(lambda x, y:x + " " + y, words, "")
print(sentence)
```

Результат:



Python is cool

Рисунок 62

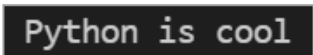
Как мы видим в результате, у нас один лишний пробел в начале троки. Изменим нашу `lambda`-функцию, добавив в ней проверку на то, а является ли первый элемент пустой строкой, если да, то проводить конкатенацию со следующим словом списка без пробела.

```
import functools
from functools import reduce

words = ['Python', 'is', 'cool']

sentence = reduce(lambda x, y: x + y if x == "" else
                  x + " " + y, words, "")
print(sentence)
```

Результат:



Python is cool

Рисунок 63

Замыкания

Мы уже знаем, что в Python существует четыре области видимости:

- Локальная область видимости (*Local*) — это часть кода, которая соответствует телу некоторой функции или лямбда-выражения
- Вложенная (нелокальная) область видимости (*Enclosing*) — данная область видимости возникает при

работе вложенных функций и содержит идентификаторы, которые мы определяем во вложенной функции.

- Глобальная область видимости (*Global*) — это уровень программы (скрипта, модуля) Python, который содержит все идентификаторы, которые мы определили на уровне программы (т. е. вне функций, если они есть).
- Встроенная области видимости (*Built-in*) — это уровень языка Python, т. е. все встроенные объекты, функции Python находятся в этой области.

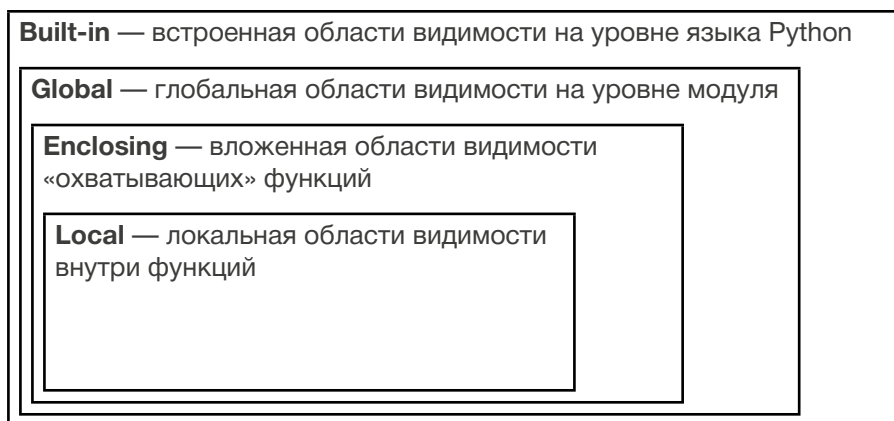


Рисунок 64

Особенности работы с локальной и глобальной областями видимости мы уже изучили детально.

Однако распространенным подходом в Python-программировании, является определение функции внутри другой функции. Такая функция, определение которой находится внутри другой, является вложенной функцией. Функция, содержащая в себе определение других функций, называется охватывающей функцией.

Идентификаторы, определенные внутри охватывающих функций, относятся к области **Enclosing** и «видны» из кода вложенных и охватывающих функций (т. е. локальная переменная охватывающей функции для ее вложенной функции относится к **Enclosing** области видимости). И тот момент, когда внутренняя функция ссылается на переменные **Enclosing** области видимости, называется замыканием (*closure*).

По определению замыкание — это внутренняя функция, которая ссылается на переменные, объявленные в теле охватывающей функции. Такие переменные называются нелокальные (**nonlocal**) переменные.

Рассмотрим на примере:

```
def sayUserHello(user):
    msg = "Hello, " + user
    def showMsf():
        print(msg + "! Let's start...")
```

Для охватывающей функции **sayUserHello()** вложенная функция **showMsf()** в этом примере является замыканием, а переменная **msg** является для нее нелокальной (**nonlocal**). Функция **showMsf()** имеет к ней доступ, чтобы работать с ее значением. Чтобы продемонстрировать это, добавим в тело охватывающей функции **sayUserHello()** вызов функции **showMsf()**:

```
def sayUserHello(user):
    msg = "Hello, " + user
    def showMsf():
        print(msg+"! Let's start...")
```

```
showMsf()
sayUserHello('admin') #Hello, admin! Let's start...
```

А как насчет возможности изменять значения нелокальных переменных внутри функции-замыкания?

Если мы попытаемся это выполнить просто задав новое значение нелокальной переменной `msg`, то фактически создадим только локальную переменную для функции `showMsf()`, которая тоже имеет имя `msg`, используется далее функцией, но фактически не изменяет значение нелокальной версии переменной `msg`. Для демонстраций этого мы после вызова `showMsf()` в теле `sayUserHello()` выводим значение переменной `msg` и видим, что оно не было изменено внутри функции `showMsf()`

```
def sayUserHello(user):
    msg = "Hello, " + user
    def showMsf():
        msg = "Student"
        print(msg + "! Let's start...")

    showMsf()
    print(msg)
```

Результат:

```
Student! Let's start...
Hello, admin
```

Рисунок 65

Для того, чтобы обеспечить желаемый результат (предоставление возможности изменять значения не-

локальных переменных внутри функции-замыкания) нам нужно перед именем такой переменной добавить ключевое слово **nonlocal**

```
def sayUserHello(user):
    msg = "Hello, " + user

    def showMsf():
        nonlocal msg
        msg = "Student"
        print(msg + "! Let's start...")
    showMsf()
    print(msg)
```

Результат:

```
Student! Let's start...
Student
```

Рисунок 66

А теперь рассмотрим ситуацию, когда наша охватывающая функция не вызывает замыкание, а возвращает его, как результат своей работы.

```
def sayUserHello(user):
    msg = "Hello, " + user

    def showMsf():
        print(msg + "! Let's start...")

    return showMsf
```

Так как теперь функция `sayUserHello()` возвращает результат, то строка кода с вызовом функции `sayUserHello()`

также будет изменена: мы создадим переменную, которая примет результат ее работы — объект функции.

```
case1 = sayUserHello('admin')
case1() # Hello, admin! Let's start...
```

Переменная `msg` не просто используется замыканием, она «запоминается». Продемонстрируем этот нюанс на примере: наша функция замыкание имеет собственные параметры:

```
def doExercise1(var1):
    var2 = 5

    def doExercise2(var3):
        return var1**var3
    return doExercise2
```

Функция-замыкание `doExercise2()` использует в своем теле только переменную `var1` (аргумент охватывающей функции `doExercise1()`). Поэтому значение переменной `var1` будет «запомнено» (несмотря на то, что выполнение функции `doExercise1(2)` было завершено).

```
case1=doExercise1(2)

print(case1(5)) #32
print(case1(10)) #1024
```

В строке вызова «`case1(5)`» программа использует для вычисления «запомненное» значение переменной `var1 (2)`, а значение `5` используется как аргумент функции-замыкания `doExercise2()`.

Также и при вызове «`case1(10)`» используется «запомненное» значение переменной `var1` (2) и аргумент для функции-замыкания `doExercise2()` — значение 10.

Рассмотрим еще один пример, демонстрирующий способность замыканий «запоминать»: реализуем программу — счетчик запусков функции.

```
def launchCounter():
    counter = 0
    def incrementCounter():
        nonlocal cout
        counter += 1
        return counter
    return incrementCounter
n = launchCounter()
for i in range(5):
    print(n())
```

Мы запустили наш счетчик в цикле и увидели «запоминание» переменной `counter` после каждого запуска функции.

Примечание: *запоминаются только те из переменных, созданные в теле охватывающей функции, которые использовались внутри функции-замыкания.*

Применение замыканий (помимо «запоминания» нужных данных) также позволяет избежать использования глобальных переменных.

Карринг

Здесь мы познакомимся с еще одной популярной техникой функционального программирования, которая называется карринг (*currying*).

Карринг (также употребляется термин «каррирование») — это техника преобразования функции от нескольких аргументов в последовательность функций, каждая из которых имеет только один аргумент. То есть это такая трансформация функции $f(x, y, z)$, чтобы она принимала аргументы в виде $f(x)(y)(z)$.

Рассмотрим на примере. Допустим, что у нас есть функция, которая принимает два аргумента: логин пользователя, которому адресуется сообщение, и сам текст сообщения:

```
def sendMsg(userTo, msgTxt):
    print("Dear {}, welcome to Python world! {}".
          format(userTo, msgTxt))
```

А это вызовы этой функции в основной программе:

```
sendMsg('admin', 'Have a nice day!')
sendMsg('admin', 'See you!')
sendMsg('admin', 'Good luck!')
sendMsg('student', 'Good luck!')
```

Как мы видим, в этой точке программы первый аргумент часто повторяется. Было бы удобно в этой ситуации иметь возможность вызвать эту функцию с установленным (зафиксированным) первым аргументом «**admin**», указывая только второй аргумент — текст сообщения.

Каррированный вариант нашей функции `sendMsg()` будет выглядеть так:

```
def sendMsg(userTo):
    def setMsg(msgTxt):
```

```
print("Dear {}, welcome to Python world! {}".  
      format(userTo, msgTxt))  
return setMsg
```

Теперь создадим новую функцию для нужного имени пользователя:

```
userAdmin=sendMsg('admin')
```

И вызовем ее с нужными аргументами-сообщениями:

```
userAdmin('Have a nice day!')  
userAdmin('See you!')  
userAdmin('Good luck!')
```

Для пользователя студента (которому нужно отправить только одно приветствие) будем напрямую вызывать функцию `sendMsg()` таким образом (тип пользователя — студент является первым параметром функции):

```
sendMsg('student') ('Good luck!')
```

Результат:

```
Dear admin, welcome to Python world! Have a nice day!  
Dear admin, welcome to Python world! See you!  
Dear admin, welcome to Python world! Good luck!
```

Рисунок 67

Методику карринга можно использовать с любым количеством аргументов. Например, пусть следующая версия нашей функции `sendMsg()` помимо логина пользователя, которому адресуется сообщение, и текста сообщения

принимает логин отправителя и название языка программирования, который начнет изучать пользователь (не только Python, как в предыдущем примере).

```
def sendMsg(userTo):
    def setMsg(msgTxt):
        def setUserFrom(userFrom):
            def setLang(lang):
                print("Dear {}, Hello from {}. Welcome to {} world! {}".format(userTo, userFrom, lang, msgTxt))
            return setLang
        return setUserFrom
    return setMsg
```

Сделаем вызовы для формирования сообщения для администратора и студента от разных отправителей и с разными языками программирования:

```
case1=sendMsg('admin')('Good luck!')
case2=sendMsg('student')('See you!')('admin')

case1('teacher')('Python')
case2('C++')
```

Результат:

```
Dear admin, Hello from teacher. Welcome to Python world! Good luck!
Dear student, Hello from admin. Welcome to C++ world! See you!
```

Рисунок 68

Как следует из рассмотренных примеров, каррирование позволяет нам легко создавать частично примененные

функции, которые позволяют упрощать вызовы при недостаточном или частично повторяющемся наборе аргументов. При таких ситуациях можно просто передать часть аргументов (которые, например, повторяются) в функцию и получить обратно частичную функцию, которая будет принимать остальные аргументы.

4. Декораторы

Ранее мы уже неоднократно использовали возможности функций, как объектов первого класса: сохраняли их в переменные; передавали в качестве аргументов и возвращали как результат работы другой функции. Также мы рассмотрели, насколько удобной и полезной является методика определения функции внутри другой функции. Изучили преимущества работы с частично примененными функциями, которые позволяют упрощать вызовы при недостаточном или повторяющемся наборе аргументов.

И сейчас мы познакомимся с еще одним полезным инструментом в Python — декораторами.

Декоратор — это функция-«обёртки», которые позволяет нам расширить функциональность уже существующей функции без прямого изменения кода в ее теле. Использование декораторов показывает, что функция может работать с другой функцией как с обычными аргументами (данными).

Рассмотрим на примере.

```
def simpleDecorator(myFunction):  
    print("Hello! I'm Decorator!")  
    def simpleWrapper():  
        print("Function starts working...")  
        myFunction()  
        print("See you!")  
    return simpleWrapper
```

В этом примере мы создали декоратор — функцию `simpleDecorator()`, которая (будучи функцией высшего

порядка) в качестве аргумента принимает имя другой функции `myFunction`, чью функциональность нам нужно расширить. Внутри `simpleDecorator()` мы определяем функцию — «обертку» `simpleWrapper()`. Функция `simpleWrapper()` «оборачивает» функцию-аргумент `myFunction`, т. е. в своем теле содержит строки кода с новой функциональностью и вызов «декорируемой» функции `myFunction()`.

В качестве результата декоратор возвращает функцию-«обертку».

Теперь допустим, что у нас есть функция, код которой мы (по разным причинам) не можем изменять:

```
def sayHi():  
    print("Welcome!")
```

Для изменения (дополнения, расширения) ее функциональности мы будем ее «декорировать». Передадим ее декоратору `simpleDecorator()`, который с помощью функции-«обертки» `simpleWrapper()` добавит новое поведение и вернет новую версию нашей базовой функции `sayHi()` с уже расширенной функциональностью.

```
sayHiAdvanced = simpleDecorator(sayHi)  
sayHiAdvanced()
```

Результат:

```
Hello! I'm Decorator!  
Function starts working...  
Welcome!  
See you!
```

Рисунок 69

Записать код предыдущего примера, используя синтаксис декораторов с помощью инструкции «@имя_декоратора» можно так:

```
@simpleDecorator
def sayHi():
    print("Welcome!")
sayHi()
```

Результат будет аналогичный предыдущему:

```
Hello! I'm Decorator!
Function starts working...
Welcome!
See you!
```

Рисунок 70

Таким образом, второй способ «декорирования» функций — это использование инструкции «@имя_декоратора», которую нужно поместить над объявлением «декорируемой» функции.

Однако, фактически мы могли просто переопределить нашу базовую функцию `sayHi()` с помощью первого подхода так:

```
def simpleDecorator(myFunction):
    print("Hello! I'm Decorator!")
    def simpleWrapper():
        print("Function starts working...")
        myFunction()
        print("See you!")
    return simpleWrapper
def sayHi():
    print("Welcome!")
```

```
sayHi = simpleDecorator(sayHi)
sayHi()
```

Результат:

```
Hello! I'm Decorator!
Function starts working...
Welcome!
See you!
```

Рисунок 71

Мы можем использовать один декоратор для любой функции и декорировать функцию любым (или несколькими) декоратором.

Создадим еще одну базовую функцию:

```
def sayBye():
    print("Buy!")
```

И «декорируем» ее созданным ранее декоратором `simpleDecorator`:

```
sayBye = simpleDecorator(sayBye)
sayBye()
```

Результат:

```
Hello! I'm Decorator!
Function1 starts working...
Buy!
See you!
```

Рисунок 72

Теперь «декорируем» нашу первую базовую функцию `sayHi()` двумя разными декораторами. Для этого вначале создадим второй декоратор:

```
def simpleDecorator_v2(myFunction):
    print("Hello! I'm Second Decorator!")
    def simpleWrapper():
        print("Let's start...")
        myFunction()
        print("Good luck!")
    return simpleWrapper
```

И применим оба к функции `sayHi()`:

```
@simpleDecorator
@simpleDecorator_v2
def sayHi():
    print("Welcome!")

sayHi()
```

Результат:

```
Hello! I'm Second Decorator!
Hello! I'm Decorator!
Function starts working...
Let's start...
Welcome!
Good luck!
See you!
```

Рисунок 73

В случае, когда к функции применяется несколько декораторов, они будут срабатывать в порядке обратному тому, в каком они были вызваны.

То есть вызов

```
@simpleDecorator
@simpleDecorator_v2
.....
```

можно для лучшего понимания записать так:

```
sayHi = simpleDecorator(simpleDecorator_v2(sayHi))
sayHi()
```

Пока мы рассмотрели только ситуации, когда базовые функции не возвращали значений. Если же функция, которую нужно «декорировать» должна возвращать значение, то она также должна возвращать и функция-«обертка».

Например:

```
def simpleDecorator_v3(myFunction):
    print("Hello! I'm Third Decorator!")
    def simpleWrapper():
        print("Function starts working...")
        result=myFunction()
        print("See you!")
        return result
    return simpleWrapper

def calculateSum():
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))
    return x+y

calculateSum = simpleDecorator_v3(calculateSum)
print(calculateSum())
```

```

Hello! I'm Third Decorator!
Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
3

```

Рисунок 74

Итак, мы научились создавать декораторы для функций, которые возвращают результат. Но во всех наших предыдущих примерах базовые функции не имели аргументов. Поэтому сейчас нужно рассмотреть, каким же образом можно передать аргументы декорируемой функции.

Аналогично предыдущей ситуации с возвратом результата функция-«обертка» и в этом случае должна обеспечить передачу аргументов в декорируемую функцию:

```

def simpleDecorator_v4(myFunction):
    print("Hello! I'm Fourth Decorator!")
    def simpleWrapper(argX, argY):
        print("I've got {},
              {}. Function starts working...".
              format(argX, argY))
        result1=myFunction(argX, argY)
        print("See you!")
        return result1
    return simpleWrapper

def calculateSum_v1(a,b):
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))
    return x+y+a+b

```

```
calculateSum_v1 = simpleDecorator_v4(calculateSum_v1)
print(calculateSum_v1(3,4))
```

Результат:

```
Hello! I'm Fourth Decorator!
I've got 3, 4. Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
10
```

Рисунок 75

Если можно передавать аргументы декорируемой функции, то возникает вопрос: а можно ли передавать аргументы самому декоратору? Мы ведь знаем, что в качестве аргумента декоратор должен принимать базовую функцию. А что, если нам нужны какие-то еще дополнительные данные, например, для управления логикой работы самого декоратора?

Для решения этой задачи нам нужно добавить еще один уровень абстракции — создать «обертку» для самого декоратора и передать этой функции нужные дополнительные аргументы.

Обязательное условие: эта функция-«обертка» для декоратора должна возвращать декоратор в результате своей работы.

```
def decoratorWrapper(argForDec):
    print("I've got arg={} for decorator!".
          format(argForDec))
```

```
def simpleDecorator_v5(myFunction):
    print("Hello! I'm Decorator with arg={}".format(argForDec))

    def simpleWrapper(argX, argY):
        print("Hi! I am Funcion. I've got {},
              {}".format(argX, argY))
        result=myFunction(argX, argY)+argForDec
        print("See you!")

        return result

    return simpleWrapper

return simpleDecorator_v5
```

Теперь мы вызовем нашу функцию `decoratorWrapper()` с нужным аргументом и получим декоратор, которому они были переданы:

```
decoratorWithArg =decoratorWrapper(10)
```

Далее «декорируем» нашу базовую функцию с двумя аргументами `calculateSum_v1()` и вызовем ее новую «декорированную» версию:

```
def calculateSum_v1(a,b):
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))
    return x+y+a+b

calculateSum_v1 = decoratorWithArg(calculateSum_v1)
print(calculateSum_v1(3,4))
```


Результат:

```
I've got arg=10 for decorator!
Hello! I'm Decorator with arg=10!
Hi! I am Funcion. I've got 3, 4. Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
20
```

Рисунок 76

Рассмотрим пример использования декоратор на практике. Допустим, что у нас есть список с ценами на товары в долларах. И функция, которая переводит цену товара в долларах в соответствующий гривневый эквивалент:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)

def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))
```

Однако сейчас на все товары действует скидка (например, 15%) и нам нужно перевести цены в гривны и еще дополнительно учесть скидку.

Скидка — непостоянная характеристика товара, поэтому изменять код функции `toPriceNew()` нам нет смысла.

Мы создадим декоратор, который «применит» скидку после перевода цены в другую валюту:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)
```

```

def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))

def changePriceDecorator_v1(myFunction):
    print("Hello! Let's change your prices...")
    def simpleWrapper(argList):
        print("I've got list of prices with {} elements.
              Function starts working...".
              format(len(argList)))
        resutl = myFunction(argList)
        resutlwithDisc = list(map(lambda x: x*(1-0.15),
                                   resutl))
        print("Let's set a discount..")
        return resutlwithDisc
    return simpleWrapper

pricesToGRN = changePriceDecorator_v1(toPriceNew)
print(pricesToGRN(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts working...
Let's set a discount..
[2345.4474999999998, 818.125, 1589.26625, 596.0625] _

```

Рисунок 77

Однако наша скидка не всегда может составлять именно 15%. Удобнее размер скидки передать декоратору в качестве его аргумента.

Изменим предыдущий код следующим образом:

```

pricesUSD=[100.34, 35, 67.99, 25.5]
print(pricesUSD)

```

```

def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))
def setDiscountDecoratorWrapper(disc):
    def changePriceDecorator_v1(myFunction):
        print("Hello! Let's change your prices...")
        def simpleWrapper(argList):
            print("I've got list of prices with {}
                  elements.
                  Function starts working...".
                  format(len(argList)))
            resutl = myFunction(argList)
            resutlwithDisc = list(map(lambda x: x*(1-disc),
                                      resutl))
            print("Let's set a discount..")
            return resutlwithDisc
        return simpleWrapper
    return changePriceDecorator_v1

discount=float(input("Discount value:"))
changePriceDecorator_v2 =
    setDiscountDecoratorWrapper(discount)
pricesToGRN = changePriceDecorator_v2(toPriceNew)
print(pricesToGRN(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Discount value:0.25
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts working...
Let's set a discount..
[2069.5125, 721.875, 1402.2937499999998, 525.9375]

```

Рисунок 78

Используя такой подход, мы можем передавать нашим декораторам нужные им аргументы в любом количестве.



Урок 4 Функции

© STEP IT Academy, www.itstep.org

© Анна Егошина

All rights to protected pictures, audio, and video belong to their authors or legal owners.

Fragments of works are used exclusively in illustration purposes to the extent justified by the purpose as part of an educational process and for educational purposes in accordance with Article 1273 Sec. 4 of the Civil Code of the Russian Federation and Articles 21 and 23 of the Law of Ukraine "On Copyright and Related Rights". The extent and method of cited works are in conformity with the standards, do not conflict with a normal exploitation of the work, and do not prejudice the legitimate interests of the authors and rightholders. Cited fragments of works can be replaced with alternative, non-protected analogs, and as such correspond the criteria of fair use.

All rights reserved. Any reproduction, in whole or in part, is prohibited. Agreement of the use of works and their fragments is carried out with the authors and other right owners. Materials from this document can be used only with resource link.

Liability for unauthorized copying and commercial use of materials is defined according to the current legislation of Ukraine.