Python_4.2

ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ

Функции

Существует множество встроенных в язык программирования функций.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими

Функции в Python

Функции в Python создаются с помощью инструкции **def**. Это действие создает объект функции и присваивает ему имя, которое становится ссылкой на объект-функцию.

```
def countnumbers():
    a = int(input())
    b = int(input())
    print(a+b)
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово **def** сообщает интерпретатору, что перед ним определение функции. За **def** следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена.

Передача параметров в функцию

При описании функции после имени в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми, но они обязательно должны быть. Далее идет двоеточие, обозначающее окончание заголовка функции.

При вызове функции в скобках указывается нужное количество переменных или выражений, значения которых будут переданы функции в качестве параметров.

Определение функции

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена.

Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Пример вызова функции:

countnumbers()

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение NameError)

Функции придают программе структуру

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Чтобы разделить поток выполнения на несколько ветвей, следует использовать оператор **if-elif-else**:

без функций

```
figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")

if figure == '1':
        a = float(input("Ширина: "))
        b = float(input("Высота: "))
        print("Площадь: %.2f" % (a*b))
elif figure == '2':
        a = float(input("Основание: "))
        h = float(input("Высота: "))
        print("Площадь: %.2f" % (0.5 * a * h))
elif figure == '3':
        r = float(input("Радиус: "))
        print("Площадь: %.2f" % (3.14 * r**2))
else:
        print("Ошибка ввода")
```

с функциями

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a*b))

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))

def circle():
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r**2))

figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")
if figure == '1':
        rectangle()
elif figure == '2':
        triangle()
elif figure == '3':
        circle()
else:
        print("Ошибка ввода")
```

Практический смысл

Вариант с функциями кажется сложнее, а каждая из трех функций вызывается всего один раз. Однако из общей логики программы как бы убраны и обособлены инструкции для нахождения площадей. Программа теперь состоит из отдельных "кирпичиков Лего". В основной ветке мы можем комбинировать их как угодно. Она играет роль управляющего механизма.

Если нам когда-нибудь захочется вычислять площадь треугольника по формуле Герона, а не через высоту, то не придется искать код во всей программе (представьте, что она состоит из тысяч строк кода как реальные программы). Мы пойдем к месту определения функций и изменим тело одной из них.

Локальные и глобальные переменные

В программировании особое внимание уделяется концепции о локальных и глобальных переменных, а также связанное с ними представление об областях видимости. Соответственно, локальные переменные видны только в локальной области видимости, которой может выступать отдельно взятая функция. Глобальные переменные видны во всей программе. "Видны" – значит, известны, доступны. К ним можно обратиться по имени и получить связанное с ними значение.

К глобальной переменной можно обратиться из локальной области видимости. К локальной переменной нельзя обратиться из глобальной области видимости, потому что локальная переменная существует только в момент выполнения тела функции. При выходе из нее, локальные переменные исчезают. Компьютерная память, которая под них отводилась, освобождается. Когда функция будет снова вызвана, локальные переменные будут созданы заново.

Локальные и глобальные переменные

Внутри функции можно использовать переменные, объявленные вне этой функции

```
def f():
    print(a)
a = 1
f()
```

1

Глобальные переменные

Здесь переменной **a** присваивается значение **1**, и функция **f()** печатает это значение, несмотря на то, что до объявления функции **f** эта переменная не инициализируется. В момент вызова функции **f()** переменной a уже присвоено значение, поэтому функция **f()** может вывести его на экран.

Такие переменные (объявленные вне функции, но доступные внутри функции) называются глобальными.

Локальные переменные

Но если инициализировать какую-то переменную внутри функции, использовать эту переменную вне функции не удастся. Например:

```
def f():
    a = 1
f()
print(a)
```

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

Такие переменные, объявленные внутри функции, называются локальными. Эти переменные становятся недоступными после выхода из функции.

Рассмотрим программу:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a*b))

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
        rectangle()
elif figure == '2':
        triangle()
```

Сколько здесь переменных? Какие из них являются глобальными, а какие – локальными?

Здесь пять переменных. Глобальной является только **figure**. Переменные **a** и **b** из функции **rectangle()**, а также **a** и **h** из **triangle()** – локальные. При этом локальные переменные с одним и тем же идентификатором a, но объявленные в разных функциях, – разные переменные.

Следует отметить, что идентификаторы **rectangle** и **triangle**, хотя и не являются именами переменных, а представляют собой имена функций, также имеют область видимости. В данном случае она глобальная, так как функции объявлены непосредственно в основной ветке программы.

В приведенной программе к глобальной области видимости относятся заголовки объявлений функций, объявление и присваивание переменной **figure**, конструкция условного оператора.

К локальной области относятся тела функций. Если, находясь в глобальной области видимости, мы попытаемся обратиться к локальной переменной, то возникнет ошибка:

```
…
elif figure == '2':
        triangle()

print(a)
```

Эти функции не совсем идеальны. Они должны вычислять площади фигур, но выводить результат на экран им не следовало бы. Вполне вероятна ситуация, когда результат нужен для внутренних нужд программы, для каких-то дальнейших вычислений, а выводить ли его на экран – вопрос второстепенный.

Если функции не будут выводить, а только вычислять результат, то его надо где-то сохранить для дальнейшего использования. Для этого подошли бы глобальные переменные. В них можно записать результат. Напишем программу вот так:

```python
result = 0

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    result = a*b

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    result = 0.5 * a * h

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
        rectangle()
elif figure == '2':
        triangle()

print("Площадь: %.2f" % result)
```

Листинг

```
result = 0
 def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    result = a*b
 def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    result = 0.5 * a * h
figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
        rectangle()
elif figure == '2':
        triangle()
 print("Площадь: %.2f" % result)
```

Проверка (попробуйте сами запустить)

Итак, мы ввели в программу глобальную переменную **result** и инициировали ее нулем. В функциях ей присваивается результат вычислений. В конце программы ее значение выводится на экран. Мы ожидаем, что программа будет прекрасно работать.

Однако

```
1-прямоугольник, 2-треугольник: 2
Основание: 6
Высота: 4.5
Площадь: 0.00
```

Особенности

Дело в том, что в Python присвоение значения переменной совмещено с ее объявлением. (Во многих других языках это не так.) Поэтому, когда имя **result** впервые упоминается в локальной области видимости, и при этом происходит присваивание ей значения, то создается локальная переменная **result**. Это другая переменная, никак не связанная с глобальной **result.**

Когда функция завершает свою работу, то значение локальной **result** теряется, а глобальная не была изменена.

Когда мы вызывали внутри функции переменную **figure**, то ничего ей не присваивали. Наоборот, мы запрашивали ее значение. Интерпретатор Питона искал ее значение сначала в локальной области видимости и не находил. После этого шел в глобальную и находил.

В случае с **result** он ничего не ищет. Он выполняет вычисления справа от знака присваивания, создает локальную переменную **result**, связывает ее с полученным значением.

global

На самом деле можно принудительно обратиться к глобальной переменной. Для этого существует команда **global**:

```
result = 0

def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    global result
    result = a*b

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    global result
    result = 0.5 * a * h

figure = input("1-прямоугольник, 2-треугольник: ")
if figure == '1':
        rectangle()
elif figure == '2':
        triangle()

print("Площадь: %.2f" % result)
```

Возврат значения

Однако менять значения глобальных переменных в теле функции – плохая практика. В больших программах программисту трудно отследить, где, какая функция и почему изменила их значение. Программист смотрит на исходное значение глобальной переменной и может подумать, что оно остается таким же. Сложно заметить, что какая-то функция поменяла его. Подобное ведет к логическим ошибкам.

Чтобы избавиться от необходимости использовать глобальные переменные, для функций существует возможность возврата результата своей работы в основную ветку программы. И уже это полученное из функции значение можно присвоить глобальной переменной в глобальной области видимости. Это делает программу более понятной.

Возврат значений из функции

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение.

В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором **return**.

Пример

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full

square = cylinder()
print(square)
```

3
7
188,4

Листинг

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full

square = cylinder()
print(square)
```

return

В данной программе в основную ветку из функции возвращается значение локальной переменной **full**. Не сама переменная, а ее значение, в данном случае – какое-либо число, полученное в результате вычисления площади цилиндра.

В основной ветке программы это значение присваивается глобальной переменной **square**. То есть выражение **square = cylinder()** выполняется так:

1. Вызывается функция **cylinder()**.
2. Из нее возвращается значение.
3. Это значение присваивается переменной **square**.

return

Не обязательно присваивать результат переменной, его можно сразу вывести на экран:

...

print(cylinder())

Здесь число, полученное из **cylinder(),** непосредственно передается функции **print()**. Если мы в программе просто напишем **cylinder()**, не присвоив полученные данные переменной или не передав их куда-либо дальше, то эти данные будут потеряны. Но синтаксической ошибки не будет.

return

В функции может быть несколько операторов **return**. Однако всегда выполняется только один из них. Тот, которого первым достигнет поток выполнения. Допустим, мы решили обработать исключение, возникающее на некорректный ввод. Пусть тогда в ветке **except** обработчика исключений происходит выход из функции без всяких вычислений и передачи значения:

return

```
def cylinder():
    try:
        r = float(input())
        h = float(input())
    except ValueError:
        return
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return full

print(cylinder())
```

Если попытаться вместо цифр ввести буквы, то сработает **return**, вложенный в **except**. Он завершит выполнение функции, так что все нижеследующие вычисления, в том числе **return full**, будут опущены. Пример выполнения:

```
r
None
```

None

None – это ничего, такой объект – "ничто". Он принадлежит классу **NoneType**.

Когда после **return** ничего не указывается, то по умолчанию считается, что там стоит объект **None**. Но никто вам не мешает явно написать **return None**.

Более того. Ранее мы рассматривали функции, которые вроде бы не возвращали никакого значения, потому что в них не было оператора **return**. На самом деле возвращали, просто мы не обращали на него внимание, не присваивали никакой переменной и не выводили на экран. В Python всякая функция что-либо возвращает. Если в ней нет оператора **return**, то она возвращает None. То же самое, как если в ней имеется "пустой" **return**.

Возврат нескольких значений

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды **return**:

```
def cylinder():
    r = float(input())
    h = float(input())
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return side, full

sCyl, fCyl = cylinder()
print("Площадь боковой поверхности %.2f" % sCyl)
print("Полная площадь %.2f" % fCyl)
```

Возврат нескольких значений

Из функции **cylinder()** возвращаются два значения. Первое из них присваивается переменной **sCyl**, второе **– fCyl**. Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков:

```
>>> a, b, c = 10, 15, 19
>>> a
10
>>> b
15
>>> c
19
```

Несколько значений

Фокус здесь в том, что перечисление значений через запятую (например, 10, 15, 19) создает объект типа кортеж.

Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса **tuple**. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора **return** стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Пример выполнения:

4

3

(75.36, 175.84)

На экран выводится кортеж, о чем говорят круглые скобки. Его также можно присвоить одной переменной, а потом вывести ее значение на экран.

Практика 2

1. Напишите функцию, которая проверяет, содержится ли число в указанном диапазоне (включая верхнюю и нижнюю границы)

def ran_check(num,low,high):

2. Напишите функцию, которая принимает на вход строку, и вычисляет количество букв в верхнем регистре и в нижнем регистре

3. Напишите функцию Python, которая получает на входе список, и возвращает новый список, содержащий уникальные элементы из первого списка.

Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]

Unique List : [1, 2, 3, 4, 5]

Практика 2

4. Напишите функцию Python, которая проверяет входную строку, является ли эта строка палиндромом или нет.

Палиндром - это слово или фраза, которые одинаково читаются слева направо и справа налево, например madam или nurses run.

pass

pass – это оператор-заглушка, равноценный отсутствию операции.

В ходе исполнения данного оператора ничего не происходит, поэтому он может использоваться в качестве заглушки в тех местах, где это синтаксически необходимо, например: в инструкциях, где тело является обязательным, таких как def, except и пр.

Зачастую pass используется там, где код пока ещё не появился, но планируется. Кроме этого, иногда, его используют при отладке, разместив на строчке с ним точку остановки.

pass

Мы можем использовать оператор pass для определения пустой функции.

```
def foo():
  pass
```

Допустим, нам нужно написать функцию для удаления всех четных чисел из списка. В этом случае мы будем использовать цикл for для обхода чисел в списке. Если число делится на 2, то ничего не делаем. В противном случае мы добавляем его во временный список.

Python не поддерживает пустые блоки кода. Таким образом, мы можем использовать здесь оператор pass для отсутствия операции в блоке if-condition.

Удалить все четные числа

```
def remove_evens(list_numbers):
    list_odds = []
    for i in list_numbers:
        if i % 2 == 0:
            pass
        else:
            list_odds.append(i)
    return  list_odds
l_numbers = [1, 2, 3, 4, 5, 6]
l_odds = remove_evens(l_numbers)
print(l_odds)
```

[1, 3, 5]

Параметры

В программировании функции могут не только возвращать данные, но также принимать их, что реализуется с помощью так называемых параметров, которые указываются в скобках в заголовке функции. Количество параметров может быть любым.

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции. Конкретные значения, которые передаются в функцию при ее вызове, будем называть аргументами. Следует иметь в виду, что встречается иная терминология. Например, формальные параметры и фактические параметры. В Python же обычно все называют аргументами.

Аргументы

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

Обратим внимание еще на один момент. Количество аргументов и параметров совпадает. Нельзя передать три аргумента, если функция принимает только два. Нельзя передать один аргумент, если функция требует два обязательных.

Примеры

Определим простейшую функцию:

```
def add(x, y):
    return x + y
```

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму **x** и **y**.

Теперь мы ее можем вызвать:

```
>>>
>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'
```

Пример

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
>>> def newfunc(n):
...     def myfunc(x):
...         return x + n
...     return myfunc
...
>>> new = newfunc(100)  # new - это функция
>>> new(200)
300
```

Позиционные

Значения в позиционных аргументах подставляются согласно позиции имён аргументов:

```
nums = [42, 11, 121, 13, 7]
state = True

#    1-я позиция "nums" -> parameter_1
#    2-я позиция "state" -> parameter_2

def test_params(parameter_1, parameter_2):
    pass
# равнозначные варианты вызова функции
test_params(nums, state)
test_params([42, 11, 121, 13, 7], True)
```

Именованные

Пусть есть функция, принимающая три аргумента, а затем выводящая их на экран. Python позволяет явно задавать соответствия между значениями и именами аргументов.

```
def print_trio(a, b, c):
    print(a, b, c)
```

```
print_trio(c=4, b=5, a=6)
```

6 5 4

При вызове соответствие будет определяться по именам, а не по позициям аргументов.

Значения по умолчанию

В Python у функций бывают параметры, которым уже присвоено значение по-умолчанию. В таком случае, при вызове можно не передавать соответствующие этим параметрам аргументы. Хотя можно и передать. Тогда значение по умолчанию заменится на переданное.

Согласно правилам синтаксиса Python при определении функции параметры, которым присваивается значение по-умолчанию должны следовать (находиться сзади) за параметрами, не имеющими значений по умолчанию.

Необязательные параметры (параметры по умолчанию)

Python позволяет делать отдельные параметры функции необязательными. Если при вызове значение такого аргумента не передается, то ему будет присвоено значение по умолчанию.

```
def not_necessary_arg(x='My', y='love'):
    print(x, y)

# если не передавать в функцию никаких значений, она отработает со значениями по умолчанию
not_necessary_arg()
> My love

# переданные значения заменяют собой значения по умолчанию
not_necessary_arg(2, 1)
> 2 1
```

Пример

```
>>> def func(a, b, c=2): # c - необязательный аргумент
...     return a + b + c
...
>>> func(1, 2)  # a = 1, b = 2, c = 2 (по умолчанию)
5
>>> func(1, 2, 3)  # a = 1, b = 2, c = 3
6
>>> func(a=1, b=3)  # a = 1, b = 3, c = 2
6
>>> func(a=3, c=6)  # a = 3, c = 6, b не определен
Traceback (most recent call last):
  File "", line 1, in
    func(a=3, c=6)
TypeError: func() takes at least 2 arguments (2 given)
```

Аргументы переменной длины (args, kwargs)

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>> def func(*args):
...     return args
...
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
(1,)
```

Когда заранее неизвестно, сколько конкретно аргументов будет передано в функцию, мы пользуемся аргументами переменной длины. Звёздочка "*" перед именем параметра сообщает интерпретатору о том, что количество позиционных аргументов будет переменным

args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Аргументы

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
>>> def func(**imargs): # в переменной imargs хранится словарь
...     return imargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

Пример

Напишем функцию max(), которая принимает два числа и возвращает максимальное из них (на самом деле, такая функция уже встроена).

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
print(max(10, 11))
print(max(11, 10))
print(max(int(input()), int(input())))
```

Пример

Теперь можно написать функцию **max3()**, которая принимает три числа и возвращает максимальное их них.

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
def max3(a, b, c):
    return max(max(a, b), c)
print(max3(5, 6, 7))
```

Переменное число аргументов - *

Встроенная функция **max()** в Питоне может принимать переменное число аргументов и возвращать максимум из них.

Приведём пример того, как такая функция может быть написана.

```
def max(*a):
    res = a[0]
    for val in a[1:]:
        if val > res:
            res = val
    return res
print(max(3, 5, 4))
```

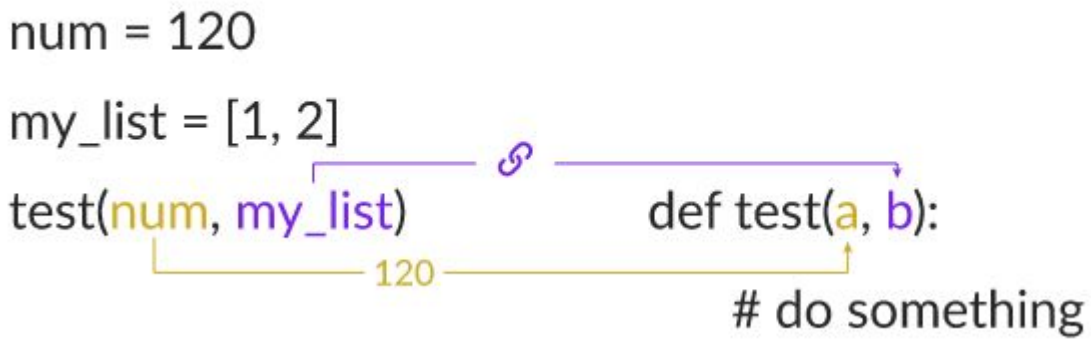
Все переданные в эту функцию параметры соберутся в один кортеж с именем **a**, на что указывает звёздочка в строке объявления функции.

Подробнее про операторы * и ** можно прочитать здесь:

https://kirill-sklyarenko.ru/lenta/zvezdochki-v-python-chto-eto-i-kak-ispolzovat

Передача по значению и по ссылке

В Python аргументы могут быть переданы, как по ссылке, так и по значению. Всё зависит от типа объекта.



По значению

- Если объект **неизменяемый**, то он передаётся в функцию по значению.

Неизменяемые объекты это:

- Числовые типы (int, float, complex)
- Строки (str).
- Кортежи (tuple)

```
num = 42
def some_function(n):
    # в "n" передается значение переменной
num (42)
    n = n + 10
    print(n)

some_function(num)
print(num)
# "num" по прежнему содержит 42
    52
    42
```

По ссылке

Изменяемые объекты передаются в функцию по ссылке. Изменяемыми они называются потому что их содержимое можно менять, при этом ссылка на сам объект остается неизменной.

В Python изменяемые объекты это:

- Списки (list).
- Множества (set).
- Словари (dict).

```
num = [42, 43, 44]

def some_function(n):
    # в "n" передается ссылка на переменную "num".
    # "n" и "num" ссылаются на один и тот же объект
    n[0] = 0
    print(n)


some_function(num)
print(num)  # "num" изменился

>
[0, 43, 44]
[0, 43, 44]
```

Побочные эффекты

Будьте внимательны при передаче изменяемых объектов. Одна из частых проблем новичков.

В функциональном программировании существует понятие "функциями с побочными эффектами" — когда функция в процессе своей работы изменяет значения глобальных переменных. По возможности, избегать таких функций.

lambda - функции

Lambda – выражение – это особая конструкция Python, в результате выполнения которой создается объект-функция, которую принято называть анонимной. Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией **def func()**

lambda функции, в отличие от обычной, не требуется инструкция **return**, а в остальном, ведет себя точно так же

Примеры

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

Примеры

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Практика 3

5. Напишите функцию f(x), которая возвращает значение следующей функции, определённой на всей числовой прямой:

$$f(x) = \begin{cases} 1-(x+2)^2, & \text{при } x \le -2 \\ -\frac{x}{2}, & \text{при } -2 < x \le 2 \\ (x-2)^2+1, & \text{при } 2 < x \end{cases}$$

Требуется реализовать только функцию, решение не должно осуществлять операций ввода-вывода

<u>проверка</u>

4.5 7.25

Практика 3

6. Напишите функцию, которая возвращает меньшее из двух чисел, если оба эти числа чётные. Иначе возвращает большее из двух чисел, если одно или оба числа нечётные.

7. blackjack: На входе три числа от 1 до 11. Если их сумма меньше или равна 21, то вернуть их сумму. Если сумма больше 21 и среди чисел есть 11, то уменьшить общую сумму на 10. И наконец, если сумма (в том числе после уменьшения) превышает 21, вернуть 'WIN'.

8. master_yoda: На входе фраза, на выходе вернуть слова (return) в обратном порядке.¶

master_yoda('I am home') --> 'home am I'