Функции

Сегодня рассмотрим:

- Что такое функция
- Объявление функций и её параметры
- Области видимости
- Анонимные функции

ЧТО ТАКОЕ ФУНКЦИЯ

Если обратиться к математике, то это некоторая зависимость одних элементов от других.

Пример:

$y = x^2+2x+5$ *# Значения y находятся в зависимости от значений x*

В программировании же функцией называют некую часть кода, который можно вызывать для выполнения определённых действий. По сути, это подпрограмма. Кроме того, функция по окончании выполнения возвращает определённое значение. Она позволяет избегать дублирования кода, а также улучшают читаемость и структурированность.

ОБЪЯВЛЕНИЕ ФУНКЦИЙ И ЕЁ ПАРАМЕТРЫ

Общая структура функции выглядит следующим образом:

```
def имя_функции(параметр_функции_1, параметр_функции_2, …,
параметр_функции_N):
        определённые_действия
        return возвращаемое_значение
```

def называют **ключевым словом** для объявления функции, а всё, что находится после запятой – **телом функции**. Поскольку в Python нет ограничителей, как в С-подобных языках (обычно – фигурные скобки), то крайне важно следить за отступами. В противном случае функция может начать работать совсем не так, как ожидается.

Пример:

```
# Простая функция по подсчёту количества имеющихся денег
# salary – месячная зарплата
# bonus – дополнительная выплата
# months – количество месяцев
def my_money(salary, bonus, months):
        result = salary * months + bonus
        return result
```

Чтобы функция выполнилась, её нужно вызвать:

Пример:

```
print(my_money(50000, 20000, 12))                # 620000
```

Кстати, для функций полезно оставлять комментарии по поводу того, что она делает и её особенностей. Чтобы можно было всегда вызывать эту подсказку, можно поместить её в так называемый docstring. Функция выше в таком случае будет выглядеть так:

Пример:

```
def my_money(salary, bonus, months):
      ***
      Простая функция по подсчёту количества имеющихся денег
      salary – месячная зарплата
      bonus – дополнительная выплата
      months – количество месяцев
      ***
      result = salary * months + bonus
      return result
```

Вызвать подсказку можно, написав ?название_функции или help(название_функции)

Пример:

```
?my_money              # Простая функция по подсчёту…
help(my_money)         # То же самое
```

Важным моментом являются параметры функции – переменные, значения, которые передаются в функцию для каких-то действий. Функцию можно определить с любым количеством параметров, в том числе – без параметров. Кроме того, если предполагается, что в определённых случаях функцию следует вызывать без какого-либо параметра, то можно прописать этому опциональному параметру **значение по умолчанию**:

Пример:

```
def my_money(salary, bonus, months = 12):      # по умолчанию – 12 месяцев
      result = salary * months + bonus
      return result

my_money(20000, 10000)                         # 250000
my_money(20000, 10000, 10)                     # 210000
```

Если количество параметров неизвестно, то при объявлении функции можно использовать т.н. args и kwargs. Args (*) предполагает, функция будет запущена с произвольным набором параметров, которые помещены в кортеж. Kwargs (**) действует похожим образом, но помещает набор именованных параметров в словарь.

Пример:

```
def my_func1(*parameters):    # args
      print(parameters)

my_func_1(20000, 10000)                        # (20000, 10000)
```

```
def my_func2(**parameters):   # kwargs
      print(parameters)

my_func_2(a = 20000, b = 10000)                    # {a: 20000, b: 10000}
```

В Python **функция всегда возвращает значение**. Если опустить return, то функция вернёт None – специальный тип данных, говорящий об отсутствии значения. В целом, не всегда нужно, чтобы использовалось возвращаемое значение, но в таком случае нельзя присваивать функцию какой-либо переменной. Это довольно частая ошибка. По этой причине нужно понимать, что происходит внутри функции, и что она вернёт.

Кроме того, нужно помнить, что любой код, написанный после return не выполнится.

Пример:

```
def my_func():
      a = 5
      return a
      b = 10
      print(b)

print(my_func())                                   # 5. Все действия после return потеряны
```

Также стоит упомянуть, что, рассматривая структуры данных, мы упоминали **методы**, с помощью которых выполняются операции. Методы – также являются функциями с тем лишь отличием, что они принадлежат конкретному объекту. Например, метод .keys() есть именно у словарей, а метод .capitalize() – именно у строк.

ОБЛАСТИ ВИДИМОСТИ

Для любого объекта в программе существуют так называемые области видимости (scope), которые определяют, как можно работать с ним. Понимание и правильная работа с ними позволяет повысить безопасность программы, оптимизировать её работу. Выделяют 3 области видимости:

- global – глобальная область видимости
- local – глобальная область видимости
- nonlocal – нелокальная область видимости

Если переменная находится в **глобальной** области видимости (в глобальном контексте), то она доступна из любого места в программе. Она определена вне какой-либо функции и не ограничена ей.

Пример:

```
a = 1
print(a)                # Вызвали переменную вне функции
def func_1():
      print(a)          # Вызвали переменную из функции

func_1()
```

Если переменная находится в **локальной** области видимости, то она определена в какой-либо функции и доступна лишь в её пределах. Чтобы значение переменной стало доступно вне функции, его надо вернуть через return, но саму переменную вызвать нельзя.

Если в ходе выполнения программы в функции запрашивается некая переменная, то Python будет искать её **в первую очередь в локальной области видимости**. В случае, **если не найдёт – обратится к локальной области видимости вышестоящей функции** (если функция определена внутри другой функции). **И так, пока не дойдёт до глобальной области видимости**. Если переменная не будет найдена и там, то возвратит ошибку. В локальных областях нижестоящих функций Python искать переменную не будет.

Пример:

```
c = 1
def func_1():
      b = 2
      def func_2():
            a = 3
            print(a)    # Выполнит. Переменная в локальной области видимости

            print(b)    # Выполнит. Переменная в локальной области видимости
                        # вышестоящей функции

            print(c)    # Выполнит. Переменная в глобальной области видимости

            print(d)    # Не выполнит. Переменная не найдена ни в одной области
                        # видимости
      func_2()
      print(a)          # Не выполнит. Переменная определена в нижестоящей
                        # функции, к которой нельзя обратиться из родительской
                        # функции
func_1()
```

Однако, если необходимо получать доступ к локальным переменным, то существуют специальные операторы **global** и **nonlocal**. Оператор global создаёт в локальном контексте глобальную переменную. Оператор nonlocal разрешает доступ к переменной из вышестоящей области видимости.

Пример:

```
def func_1():
      b = 2
      def func_2():
            nonlocal a = 3
            global b = 2
            print(a)
      func_2()
      print(a)          # Выполнит. Переменная a объявлена как nonlocal
func_1()
print(b)                # Выполнит. Переменная b объявлена как global
```

Использование данных операторов требует внимательности, поскольку при наличии одинаково именованных переменных может происходить перезапись значения.

Пример 1:

```
# Обычное поведение
a = 1
def func_1():
      a = 2
      print(a)
func_1()                # 2
print(a)                # 1
```

Пример 2:

```
# global
a = 1
def func_1():
      global a = 2
      print(a)
func_1()                # 2
print(a)                # 2. Так как произошла перезапись глобальной переменной
```

АНОНИМНЫЕ ФУНКЦИИ

Функции выполняют самые разные действия, в том числе мелкие и рутинные. Многие из них идут в одно действие и нигде более не вызываются. Для таких случаев существуют так называемые **анонимные функции** или **lambda-функции**. Их структура такова:

```
lambda параметр_1, параметр_2, …, параметр_N: действие
```

Пример:

```
result = lambda x, pow: x ** pow              # Возведение в степень
print(result(4, 3))                           # 64
```

У условий есть сокращённая форма записи, которая применяется, если в каждой ветке есть только одно действие. Это удобно именно для lambda-функций. В них обычное if-else условие будет записано так:

```
вариант_1 if условие else вариант_2
```

Пример:

```
bool_var = lambda x: true if x != 0 else false
print(bool_var(1), bool_var(0), bool_var(-1))        # true false true
```

Анонимные функции часто используются при работе с **функциями высшего порядка** – функциями, которые принимают в качестве аргумента другие функции или возвращают другие функции. Рассмотрим применение на примере функции map, которая принимает в качестве параметров список, а также функцию, которая должна этот список обработать, и возвращает изменённый список.

Пример:

```
new_list = [1, 2, 3, 4, 5]
for i in map(new_list, lambda x: x ** 2):
      print(i)          # 1 4 9 16 25
```

Данная функция позволяет проводить различные операции над списками. Тем не менее, для простых задач это не требуется. Например, если необходимо пронумеровать все элементы существует функция enumerate(список), которая выполнит данное действие. Подробнее можно прочесть в дополнительной литературе.