Информационные технологии

Пример определения функции ¶

Например:

In [ ]:

```
from math import sqrt
sqrt(25)+sqrt(9)
```

Функции в программировании похожи на функции в математике, хотя имеют и свою специфику. Давайте напишем какую-нибудь функцию. Для примера рассмотрим *факториал*. Напомним, что факториал от натурального числа n это произведение всех натуральных чисел (начиная с 1) до n.

$$\text{factorial}(n) = 1 \times 2 \times \ldots \times n$$

Давайте для начала напишем программу для вычисления факториала какого-нибудь числа. Она будет выглядеть так:

In [ ]:

```
n = 5

f = 1
for i in range(2, n+1):
    f = f * i
    # эквивалентный синтаксис: f *= i
print(f)
```

В данном случае нам нужно написать функцию, вычисляющую факториал. Она выглядит так:

In [ ]:

```
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Теперь мы можем *вызывать* функцию `factorial` , передавая ей параметр.

In [ ]:

```
factorial(6)
```

И даже использовать её в более сложных выражениях:

In [ ]:

```
factorial(5)+factorial(6)
```

Посмотрим более внимательно на то, что происходит, когда Python вычисляет значение выражения `factorial(6)`. В первую очередь он смотрит на первую строчку определения функции (это так называемая *сигнатура*):

```
def factorial(n):
```

Здесь он видит, что функция `factorial()` имеет аргумент, который называется `n`. Python помнит, что мы вызвали `factorial(6)`, то есть значение аргумента должно быть равно 6. Таким образом, дальше он выполняет строчку (которую мы не писали)

```
n = 6
```

После чего выполняет остальные строчки из *тела функции*:

```
f = 1
for i in range(2, n+1):
    f = f * i
```

Наконец он доходит до строчки

```
return f
```

В этот момент переменная `f` имеет значение 24. Слово `return` означает, что Python должен вернуться к строчке, в которой был вызов `factorial(6)`, и заменить там `factorial(6)` на 24 (то, что написано после `return`). На этом вызов функции завершён.

Возвращаемые значения

Давайте посмотрим на примере, в чём разница между возвращаемым значением и побочным эффектом. Напишем функцию, которую приветствует пользователя.

In [ ]:

```
def hello(name):
    return "Hello, "+name+"!"
```

In [ ]:

```
s = hello("World")
```

In [ ]:

```
s
```

В переменной `s` сейчас — результат выполнения функции `hello()`, которой на вход передали аргумент `name`, равный `"World"`.

А теперь давайте напишем другую функцию, которая не *возвращает* строчку, а печатает её.

In [ ]:

```
def say_hello(name):
    print("Hello, "+name+"!")
```

В этой функции вообще нет команды `return`, но Python поймёт, что из функции надо возвращаться в основную программу в тот момент, когда строчки в функции закончились. В данном случае в функции только одна строка.

In [ ]:

```
s = say_hello("Harry")
```

Обратите внимание: теперь при выполнении `s = say_hello("Harry")` строчка выводится на печать. Это и есть *побочный эффект* выполнения функции `say_hello`. Что лежит в переменной `s`?

In [ ]:

```
s
```

In [ ]:

```
print(s)
```

В ней лежит *ничего*. Это специальный объект `None`, который испоьзуется, когда какое-то значение нужно присвоить переменной, но никакого значения нет. В данном случае его нет, потому что функция ничего не вернула. Возвращаемого значения у `say_hello()` нет. Так бывает.

Более сложные ситуации с функциями

Функции могут вызывать другие функции. Например, вместо того, чтобы копировать строчку `"Hello, "+name+"!"` из функции `hello()` в функцию `say_hello()`, просто вызовем `hello()` из `say_hello()`.

In [ ]:

```
def new_say_hello(name):
    print(hello(name))
```

In [ ]:

```
new_say_hello("Harry")
```

Например, давайте напишем функцию, которая вычисляет *биномиальные коэффициенты*. Напомним, что биномиальным коэффициентом C_n^k (читается «це из эн по ка») называется число, показывающее, сколькими способами можно выбрать k объектов из n. Великая наука комбинаторика учит нас, что это число может быть вычислено следующим образом:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Здесь восклительными знаками обозначаются факториалы. Напишем функцию, которая вычисляет биномиальный коэффициент. Используем для этого написанную ранее функцию `factorial`.

In [ ]:

```
def binom(k, n):
    """
    calculates binomial coeffs: k from n
    k, n are integers
    retuns C_n^k
    """
    return factorial(n)//(factorial(k)*factorial(n-k))
    # можно использовать целочисленное деление, поскольку результат гарантированно целы
й
```

Сколькими способами можно выбрать двух дежурных из трёх участников похода? Тремя — потому что выбрать двух дежурных это то же самое, что и выбрать одного человека, который не дежурит.

In [ ]:

```
binom(2,3)
```

В тройных кавычках сразу после сигнатуры функции обычно приводится её описание (так называемая `docstring`). Это комментарий для людей, которые будут использовать вашу функцию в будущем. Чтобы посмотреть на эту справку, можно набрать название вашей функции, открывающую скобку и нажать *Shift+Tab+Tab*.

После выполнения строчки `return` выполнение функции прекращается. Давайте рассмотрим ещё один пример: вычислим модуль некоторого числа.

In [ ]:

```
def my_abs(x):
    if x > 0:
        return x
    else:
        return -x
```

In [ ]:

```
my_abs(-5)
```

Это самое простое решение: если число положительное, то возвращается оно само, а если отрицательное, то возвращается оно с обратным знаком (`-x`). Можно было бы написать эту функцию и таким образом:

In [ ]:

```
def my_abs(x):
    print("New my_abs")
    # если функция с таким названием уже была, то Python про неё забудет и запишет вмес
то этого новую функция
    # чтобы убедиться в этом я поставил здесь этот print
    if x > 0:
        return x
    return -x
```

In [ ]:

```
my_abs(-6)
```

Здесь происходит следующее: если число положительное, то срабатывает `return x` и после этого выполнение функции прекращается, до строчки `return -x` дело не доходит. А если число отрицательное, то наборот, срабатывает только строчка `return -x` (из-за оператора `if`).

In [ ]:

```
%load_ext tutormagic
```

Локальные и глобальные переменные

Внутри функции могут создаваться и использоваться различные переменные. Чтобы это не создавало проблем, переменные, определенные внутри функции, не видны извне. Давайте рассмотрим пример:

In [ ]:

```
f = 10

def factorial(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    print("In the function, f =", f)
    return f

f = 10
print(f)
print(factorial(8))
print("Out of function")
print(f)
```

Тот же код, запущенный в визуализаторе:

In [ ]:

```
%%tutor lang='python3'
f = 10

def factorial(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    print("In the function, f =", f)
    return f

f = 10
print(f)
print(factorial(8))
print("Out of function")
print(f)
```

Как видно из результата выполнения этого кода, переменная `f` в основной программе и переменная `f` внутри функции — это совсем разные переменные (визуализатор pythontutor рисует их в разных *фреймах*). От того, что мы как-то меняем `f` внутри функции, значение переменной `f` вне её не поменялось, и наоборот. Это очень удобно: если бы функция меняла значение «внешней» переменной, то она могла бы сделать это случайно и это привело бы к непредсказуемым последствиям.

Допустим, мы хотим написать функцию, которая будет приветствовать пользователя, используя язык, указанный им в настройках. Она могла бы выглядеть таким образом:

In [ ]:

```
def hello_i18n(name, lang):
    if lang == 'ru':
        print("Привет,",name)
    else:
        print("Hello,",name)
```

In [ ]:

```
hello_i18n("Ivan", 'ru')
```

In [ ]:

```
hello_i18n("Ivan", 'en')
```

Проблема в том, что функций, которым нужно знать, какой язык выбран, может быть очень много, и каждый раз передавать им вручную значение переменной `lang` отдельным параметром довольно мучительно. Оказывается, можно этого избежать:

In [ ]:

```
def hello_i18n(name):
    if lang == 'ru':
        print("Привет,",name)
    else:
        print("Hello,",name)

lang = 'ru'
print("Hello world")
hello_i18n('Ivan')

lang = 'en'
hello_i18n('John')
```

Как видите, сейчс поведение функции зависит от того, чему равняется переменная `lang`, определенная вне функции. Может быть и в функции `factorial()` можно было обратиться к переменной `f` до того момента, как мы положили в неё число 1? Давайте попробуем:

In [ ]:

```
def factorial(n):
    print("In the function, before assignment, f =", f)
    f = 1
    for i in range(2, n+1):
        f = f * i
    print("In the function, f =", f)
    return f
```

In [ ]:

```
factorial(2)
```

В этом случае Python выдаёт ошибку: локальная переменная `f` использовалась до присвоения значения. В чём разница между этим кодом и предыдущим?

Python прежде, чем выполнить функцию, он анализирует её код и определяет, какая из переменных является локальной, а какая глобальной. В качестве глобальных переменных по умолчанию используются те, которые не меняются в теле функции (то есть такие, к которым не применяются операторы типа приравнивания или `+=`). Иными словами, по умолчанию глобальные переменные доступны только для чтения, но не для модификации изнутри функции.

Ситуация, при которой функция модифицирует глобальную переменную, обычно не очень желательна: функции должны быть изолированы от кода, который из запускает, иначе вы быстро перестанете понимать, что делает ваша программа. Тем не менее, иногда модификация глобальных переменных необходима. Например, мы хотим написать функцию, которая будет устанавливать значение языка пользователя. Она может выглядеть примерно так:

In [ ]:

```
def set_lang():
    useRussian = input("Would you like to speak Russian (Y/N): ")
    if useRussian == 'Y':
        lang = 'ru'
    else:
        lang = 'en'
```

In [ ]:

```
lang = 'en'
print(lang)
set_lang()
print(lang)
```

Как видим, эта функция не работает — собственно, и не должна. Чтобы функция set_lang смогла менять значение переменной `lang`, её необходимо явно объявить как глобальную с помощью ключевого слова `global`.

In [ ]:

```
def set_lang():
    global lang
    useRussian = input("Would you like to speak Russian (Y/N): ")
    if useRussian == 'Y':
        lang = 'ru'
    else:
        lang = 'en'
```

In [ ]:

```
lang = 'en'
print(lang)
set_lang()
print(lang)
```

Теперь всё работает!

Передача аргументов

Есть разные способы передавать аргументы функции. С одним из них мы уже знакомы:

In [ ]:

```
def hello(name, title):
    print("Hello", title, name)
```

In [ ]:

```
hello("Potter", "Mr.")
```

В некоторым случаях мы хотим, чтобы какие-то аргументы можно было не указывать. Скажем, мы хотим иметь возможность вызвать функцию `hello()`, определённую выше, не указывая `title`. В этом случае сейчас нам выдадут ошибку:

In [ ]:

```
hello("Harry")
```

Это неудивительно: мы сказали, что функция `hello()` должна использовать аргумент `title`, но не передали его — какое же значение тогда использовать? Для преодоления этой трудности используютя значения по умолчанию (default values).

In [ ]:

```
def hello(name, title=""):
    print("Hello", title, name)
hello("Harry")
```

In [ ]:

```
hello("Smith", "Mrs.")
```

Аргументы можно передавать, указывая их имена.

In [ ]:

```
hello("Smith", title = "Mr.")
```

In [ ]:

```
hello(name = "Smith", title= "Mr.")
```

В этом случае порядок будет неважен.

In [ ]:

```
hello(title= "Mr.", name = "Smith")
```

Ещё бывают функции, которые принимают неограниченное число аргументов. Например, так ведёт себя функция `print()`.

In [ ]:

```
print(8, 7, 5, 'hello', 8)
```

Как она устроена? Примерно вот так:

In [ ]:

```
def my_print(*args):
    for x in args:
        print(x)
```

In [ ]:

```
my_print(6, 8, 9, 'hello', 88, 55)
```

Обратите внимание на звёздочку перед `args` в сигнатуре функции. Давайте посмотрим повнимательнее, как работает этот код:

In [ ]:

```
def test(*args):
    print(args)
test(1,2,3, 'hello')
```

Оказывается, что в `args` теперь лежит так называемые *кортеж*, состоящий из элементов, которые мы передали функции.

Отступление: кортежи

Кортеж (`tuple`) — это почти то же самое, что и список, только его элементы неизменяемы. Обозначается он круглыми скобками.

In [ ]:

```
t = (2,3, 5, 1)
print(t[1])
print(t[0:2])
```

In [ ]:

```
for x in t:
    print(x)
```

In [ ]:

```
t[0] = 10
```

In [ ]:

```
t.append(1)
```

Тот факт, что нельзя изменять кортеж, не означает, что нельзя переопределить переменную `t` :

In [ ]:

```
t = (8, 1, 2, 3)
```

In [ ]:

```
t
```

Можно также конвертировать списки в кортежи и наоборот.

In [ ]:

```
print( list( (1, 2, 3) ) )
print( tuple( [1, 2, 3] ) )
```

Возвращаясь к функциям

Таким образом, звёздочка в сигнатуре как бы ставит дополнительные скобки вокруг аргументов. Например, следующие два вызова приводят к одинаковым результатам.

In [ ]:

```
def test1(*args):
    print(args)
test1(1, 2, 3)

def test2(args):
    print(args)
test2( (1,2,3) )
```

Можно комбинировать списочные переменные и обычные (при условии, что переменная со звёздочкой только одна):

In [ ]:

```
def my_print(sep, *args):
    for x in args:
        print(x, end = sep)
```

In [ ]:

```
my_print('----', 7, 8, 9, 'hello')
```

На сегодня всё :)