Working with Functions in Python

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

Where to Find The Code and Materials?

https://github.com/iproduct/intro-python

Functions

- Многократно изпълнимо множество команди
- Не се изпълняват в програмата докато не бъдат “извикани” (има “обръщение” към тях)
- Свойства:
  - Имат име
  - Имат аргументи (параметри) - 0 или повече
  - Имат docstring (документация)
  - Имат тяло
  - Връщат стойност

Functions Syntax

keyword

name

parameters or arguments

specification, docstring

```
def is_even (i):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

body

later in the code, you call the function using its name and values for parameters

```
is_even(3)
```


5

Defining Functions

Използва се командата **def**:

```
>>> def   foo(bar):
...           return bar
>>>
```

Това е елементарна функция с име **foo** и с един параметър **bar**

Function Objects

Всяка дефиниция създава обект от тип function в текущото пространство на имена

```
>>> foo
<function foo at fac680>
>>>
```

Този обект може да бъде извикан:

```
>>> foo(3)
3
>>>
```

Functions

- Командата **def** създава функция с дадено име
- Командата **return** връща резултат там, където е направено обръщение към функцията
- Аргументите се предават чрез присвояване
- Аргументите и връщаната стойност не се декларират

def <име>(arg1, arg2, ..., argN):

<команди>

return <израз-стойност>

def times(x,y): return x*y

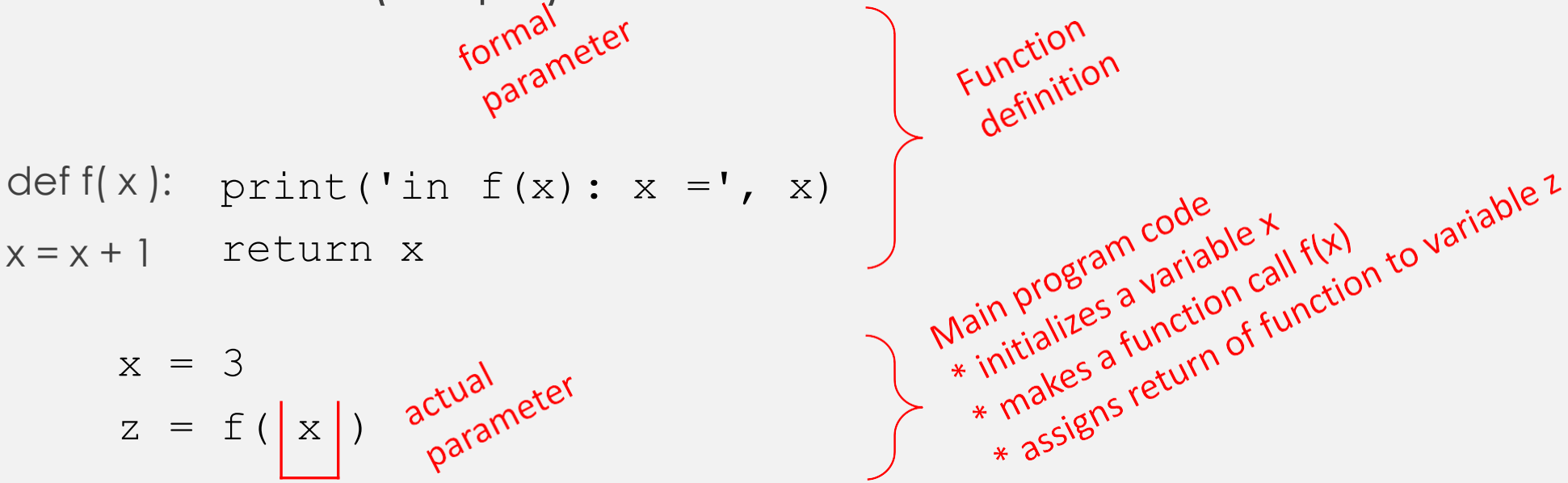
Function Arguments

- Аргументите се предават чрез присвояване
- Предават стойностите си на локални променливи
- Това по никакъв начин не променя оригиналните обекти и променливи
- Промяна на mutable аргумент може да промени и оригиналния обект или променлива
- def changer (x,y):

```
    x = 2           # changes local value of x only
    y[0] = 'hi'     # changes shared object
```

Function Scopes

- Формалният (локален) параметър сочи към стойността на
- реалния параметър при обръщение към функция
- Създава се нова област на имена (scope/frame/environment) при изпълнение на функция
- Тази област (scope) е съпоставяне на имена с обекти



Differtent Types of Arguments

- default arguments
- keyword arguments
- positional arguments
- arbitrary positional arguments
- arbitrary keyword arguments

Default Arguments

- Някои възможни аргументи не се подават при обръщение към функция (незадължителни)
- За тях се дефинира стойност по премълчаване
- Те са след задължителните в дефиницията

```
def func(a, b, c=10, d=100): print (a, b, c, d)

>>> func(1,2) 1 2 10 100

>>> func(1,2,3,4) 1,2,3,4
```

Functions Proprties

- Всички функции в Python връщат стойност
- При липса на команда return се връща обект **None**
- Всяка функция трябва да има уникално име – не се допускат функции с еднакви имена
- Функциите са обекти и могат:
  - Да бъдат аргументи към други функции
  - Да връщат като резултат обект - функция
  - Да бъдат присвоявани на променливи
  - Да бъдат елементи в редици, списъци и други

Functions Argument Types

- # Позиционни

```
def sum(a, b)
    return a + b
```

- # Ключови думи (незадължителни)

```
def shout(vik="hooray!!!")
    print(vik)
```

- # Позиционни и ключови

```
def echo(niz, prefix=" ")
    print(prefix, niz)
```

Variadic Arguments

- Когато стойността на аргументи е в контейнер (редица или списък) пред името се слага *****. Така се задават и произволен брой аргументи
- Ако е в контейнер речник, тогава пред името се поставя ******

```
def some_fun(*a, **b)
    for i in a:
    print(a)
    for key, val in b.items()
    print(key)
some_fun(1, 2, 3, name="Digits")
```

Different Types of Arguments

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

where **/** and ***** are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only.

Keyword parameters are also referred to as named parameters.

Differtent Types of Arguments - II

- Positional-or-Keyword Arguments - if / and * are not present in the function definition, arguments may be passed to a function by position or by keyword.
- Positional-Only Parameters - looking at this in a bit more detail, it is possible to mark certain parameters as positional-only. If positional-only, the parameters' order matters, and the parameters cannot be passed by keyword. The / is used to logically separate the positional-only parameters from the rest of the parameters. If there is no / in the function definition, there are no positional-only parameters.
- Keyword-Only Arguments - to mark parameters as keyword-only, indicating the parameters must be passed by keyword argument, place an * in the arguments list just before the first keyword-only parameter.

Differtent Types of Arguments - Examples

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

Differtent Types of Arguments - Recap

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2)

- Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD