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

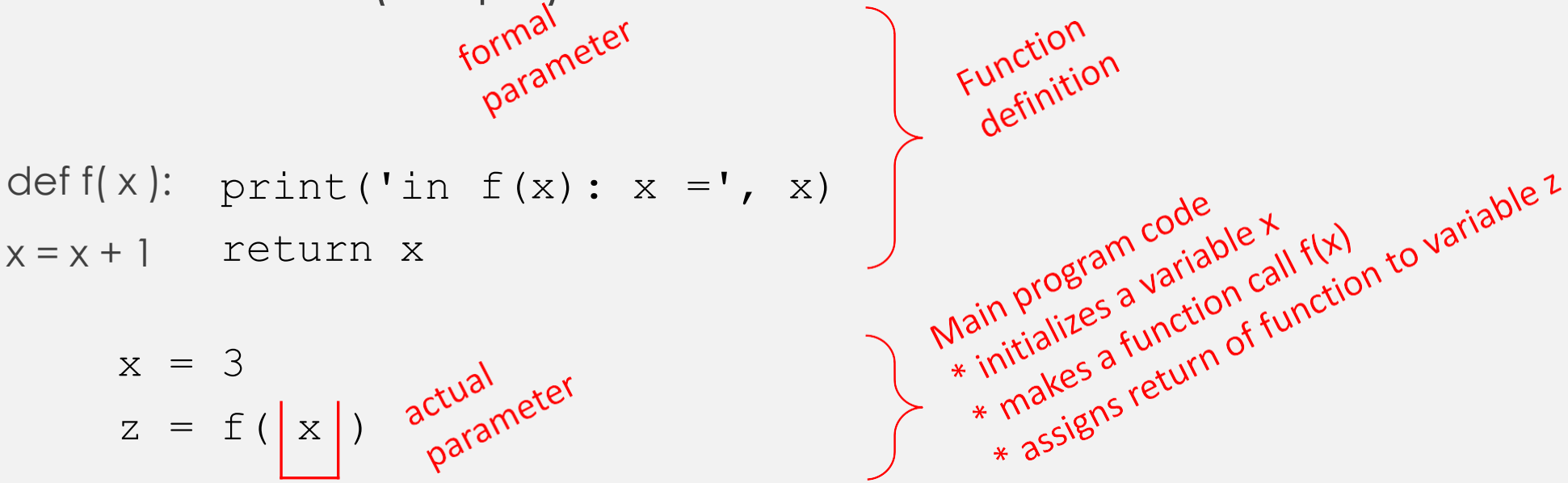
Function Arguments

- 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.

Problem 1: Factoring a Number

Implement as a Python function an algorithm for decomposition of an integer number to prime divisors (factoring).

Problem 2: Efficient Recursive Function Implementation

Implement a recursive Python function calculating efficiently the n-th number in following infinite sequence:

f[0] = 0

f[1] = 1

f[n] = f[n-1] + f[n//2], for n >= 2

The implementation should efectively calculate results for 1000 < n < 10000

Zip, Unzip

```
first_name = ['Joe', 'Earnst', 'Thomas', 'Martin', 'Charles']
last_name = ['Schmoe', 'Ehlmann', 'Fischer', 'Walter', 'Rogan', 'Green']
age = [23, 65, 11, 36, 83]
full_name_list = list(zip(first_name, last_name, age))
print( full_name_list)
```

```
[('Joe', 'Schmoe', 23), ('Earnst', 'Ehlmann', 65), ('Thomas', 'Fischer', 11), ('Martin', 'Walter', 36), ('Charles', 'Rogan', 83)]
```

```
# unzip
first_name, last_name, age = list(zip(*full_name_list))
print(f"first name: {first_name}\nlast name: {last_name} \nage: {age}")
```

```
first name: ('Joe', 'Earnst', 'Thomas', 'Martin', 'Charles')
last name: ('Schmoe', 'Ehlmann', 'Fischer', 'Walter', 'Rogan')
age: (23, 65, 11, 36, 83)
```

Map, Filter, Reduce, Zip

```
books = [
    (1, "Learning Python", "", "Марк Лътз, Дейвид Асър", "O'Reily", 1997, 22.7),
    (2, "Think Python", "An Introduction to Software Design", "Алън Б. Дауни", "O'Reily", 2002, 9.4),
    (3, "Python Cookbook", "Recipes for Mastering Python 3", "Браян Джоунс, Дейвид Баазли", "O'Reily", 2011, 62),
]

def after_year_2000(book: Book) -> bool:
    return book[-2] > 2000

if __name__ == '__main__':
    print(all(map(after_year_2000, books)))
    print(any(map(lambda book: not after_year_2000(book), books)))
    all_books_title_year = map(lambda book: (book[1], book[-2], book[-1]), books) # projection
    new_books_to_buy = filter( lambda book_year: book_year[1] > 2000, all_books_title_year) # selection
    new_books_to_buy_by_column = zip(*new_books_to_buy)
    numbered_books = zip(count(1), *new_books_to_buy_by_column)
    numbered_books_list = list(numbered_books)
    print(*numbered_books_list, sep="\n")
    print("Total: ", sum(map(lambda book: book[-1], numbered_books_list)))
    print("Total: ", reduce(lambda acc, book: acc + book[-1], numbered_books_list, 0.0))
```

Problem 3: Long Number

You are given a long decimal number aa consisting of nn digits from 11 to 99. You also have a function ff that maps every digit from 11 to 99 to some (possibly the same) digit from 11 to 99.

You can perform the following operation no more than once: choose a (possibly empty) contiguous subsegment of digits in aa, and replace each digit xx from this segment with f(x)f(x). For example, if a=1337, f(1)=1,f(3)=5, f(7)=3, and you choose the segment consisting of three rightmost digits, you get 15531553 as the result.

What is the maximum possible number you can obtain applying this operation no more than once?

Hint: You can exercise higher order functions: map, filter, reduce, etc. : https://book.pythontips.com/en/latest/map_filter.html

Input
The first line contains one integer nn (1≤n≤2·1051≤n≤2·105) -- the number of digits in aa.
The second line contains a string of nn characters, denoting the number aa. Each character is a decimal digit from 11 to 99.
The third line contains exactly 99 integers f(1)f(1), f(2)f(2), ..., f(9)f(9) (1≤f(i)≤91≤f(i)≤9).

Output
Print the maximum number you can get after applying the operation described in the statement no more than once.

Examples:
input
4
1337
1 2 5 4 6 6 3 1 9
output
1557

input
5
11111
9 8 7 6 5 4 3 2 1
output
99999

input
2
33
1 1 1 1 1 1 1 1 1
output
33

Iterators and Generators -I

```
class Repository:
    def __init__(self):
        self._items = {}
…

def __iter__(self):
    # self._values = self._items.values().__iter__()
    return RepositoryIterator(list(self._items.values()))
    # return iter(self._items.values())

def __iter__(self):
    for item in list(self._items.values()):
        yield item
```

Iterators and Generators - II

```
class RepositoryIterator:
    def __init__(self, values: list):
        self._values = values
        self._next_index = -1

    def __next__(self):
        self._next_index += 1
        if self._next_index < len(self._values):
            return self._values[self._next_index]
        raise StopIteration()
```

Fibonacci Generator

```
def fib_gen(count: int = 10) -> Iterator[int]:
    a = 0
    b = 1
    yield a
    while count:
        yield b
        a, b = b, a + b
        count -= 1

if __name__ == "__main__":
    print("Using while:")
    fg_instance = iter(fib_gen(20))
    try:
        while True:
            print(next(fg_instance))
    except StopIteration:  pass
    print("Using for:")
    for i, fib in enumerate(fib_gen(20)):
        print(i, ":", fib)
    print("Demo end.")
```

Decorators

```
from functools import wraps, update_wrapper

def my_decorator(f):
    @wraps(f)
    def wrapper(*args, **kwds):
        print('Calling decorated function')
        return f(*args, **kwds)

    return  wrapper
    # return update_wrapper(wrapper, f)


@my_decorator
def example(*args, **kwargs):
    """Docstring"""
    return f"Called example function: {args}, {kwargs}"

if __name__ == '__main__':
    print(example(1, 2, 3, title="abc"), ", function name:", example.__name__)
```

```
Calling decorated function
Called example function: (1, 2, 3), {'title': 'abc'} ,
function name: example
```

Decorator Exaple - @profile

```python
import cProfile
import io
import pstats
from functools import wraps

def profile(fnc):
    @wraps(fnc)
    def wrapper(*args, **kwargs):
        pr = cProfile.Profile()
        pr.enable()
        ret_val = fnc(*args, **kwargs)
        pr.disable()
        s = io.StringIO()
        sortby = 'cumulative'
        ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
        ps.print_stats()
        print(s.getvalue())
        return ret_val
    return wrapper
```

```python
def factor_r2(n: int) -> list[int]:
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return [i] + (factor_r2(n // i))  # recursion step
    return [n]  # recursion bottom

@profile
def test(func, *args, count=1000, **kwargs):
    while count:
        func(*args, **kwargs)
        count -= 1


if __name__ == '__main__':
    sys.setrecursionlimit(10 ** 6)
    n = 1000000
    iterations = 100000
    factors = []
    test(factor_r1, n, count=iterations)
    # print(factor_r1(n, factors))
    # print("Factors:", factors)
    test(factor_r2, n, count=iterations)
    test(factor_r3, n, count=iterations)
    test(factor_i1, n, count=iterations)
```

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