

# Python Basic Course

*Part II*

Stefano Alberto Russo

# Outline

- Part I: introduction and basics

- What is Python
- Tools and “hello world”
- Basic syntax and data types
  - assignments, types and operators
  - conditional blocks and loops

- Part II: architecture

- Functions
- Scope
- Built-ins
- Modules

- Part IV: manipulating data

- List operations
- String operations
- Dealing bad data
- Reading and writing files

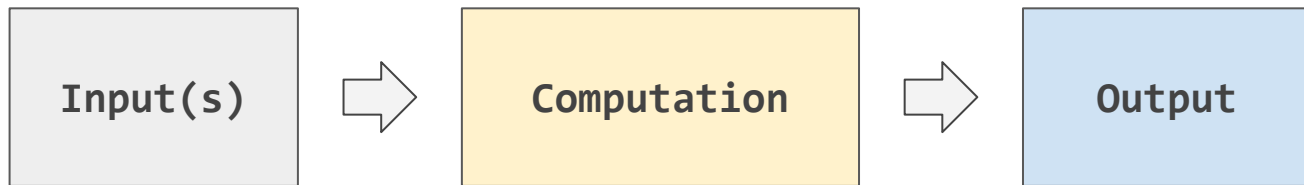
- Part VI: Pandas

- Series and Dataframes
- Common operations
- How to read documentation

# Functions

→ *What are functions*

- Functions are computational units which, given an input, produce an output



# Functions

→ *What are functions*

- Functions are defined in Python with:
  - the *def* keyword:
  - a list of (optional) arguments
  - an indented block
  - the (optional) *return* keyword

```
def square(number):  
    result = number*number  
    return result
```

# Functions

## → *Examples*

- A function with multiple arguments:

```
def rescale_number(number, factor):  
    result = number / factor  
    return result
```

```
print(rescale_number(5,10))  
0.5
```

# Functions

## → Examples

- A function with multiple return values:

```
def string_to_chars(string):  
    chars = []  
    for char in string:  
        chars.append(char)  
    return chars
```

```
print(string_to_chars('hello'))  
['h', 'e', 'l', 'l', 'o']
```

# Functions

## → Examples

- A function with multiple return values:

```
def count_chars(string):  
    chars_count = {}  
    for char in string:  
        if char not in chars_count:  
            chars_count[char] = 1  
        else:  
            chars_count[char] += 1  
    return chars_count
```

```
print(count_chars('hello'))  
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

# Functions

## → *Examples*

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    for number in number_list:  
        number = number/factor  
    return number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
...?
```



# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    for number in number_list:  
        number = number/factor  
    return number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[1,2,3] WRONG
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    for i in range(len(number_list)):  
        number_list[i] = number_list[i]/factor  
    return number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
...?
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    for i in range(len(number_list)):  
        number_list[i] = number_list[i]/factor  
    return number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[0.1, 0.2, 0.3] OK  
print(my_list)  
...?
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    for i in range(len(number_list)):  
        number_list[i] = number_list[i]/factor  
    return number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[0.1, 0.2, 0.3] OK  
print(my_list)  
[0.1, 0.2, 0.3] WRONG
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    rescaled_number_list = []  
    for number in number_list:  
        rescaled_number_list.append(number/factor)  
    return rescaled_number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
...?
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    rescaled_number_list = []  
    for number in number_list:  
        rescaled_number_list.append(number/factor)  
    return rescaled_number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[0.1, 0.2, 0.3] OK
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    rescaled_number_list = []  
    for number in number_list:  
        rescaled_number_list.append(number/factor)  
    return rescaled_number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[0.1, 0.2, 0.3] OK  
print(my_list)  
...?
```

# Functions

## → Examples

- Level up: a function which modifies something

```
def rescale_numbers(number_list, factor):  
    rescaled_number_list = []  
    for number in number_list:  
        rescaled_number_list.append(number/factor)  
    return rescaled_number_list
```

```
my_list = [1,2,3]  
print(rescale_numbers(my_list, 10))  
[0.1, 0.2, 0.3] OK  
print(my_list)  
[1, 2, 3] OK
```



# Functions

→ *Arguments by value or by reference*

- In most programming languages, arguments in functions can be passed by:
  - *value*, where values are “copied” inside the functions
  - *reference*, where only a reference is passed to the function
- If I change an argument passed by value inside a function, I do not change it outside
- If I instead change an argument passed by reference inside a function, I am actually changing the original and therefore it changes even outside the function

# Functions

→ *Arguments by value or by reference*

- In Python, immutable types are passed by value, all the others by reference.
- In short, this means that:
  - integers, strings, tuples etc., which are immutable types, are passed by value and can be freely manipulated inside the functions
  - lists, dictionaries, sets etc., which are mutable types, are passed by reference and should *never* be changed inside the functions

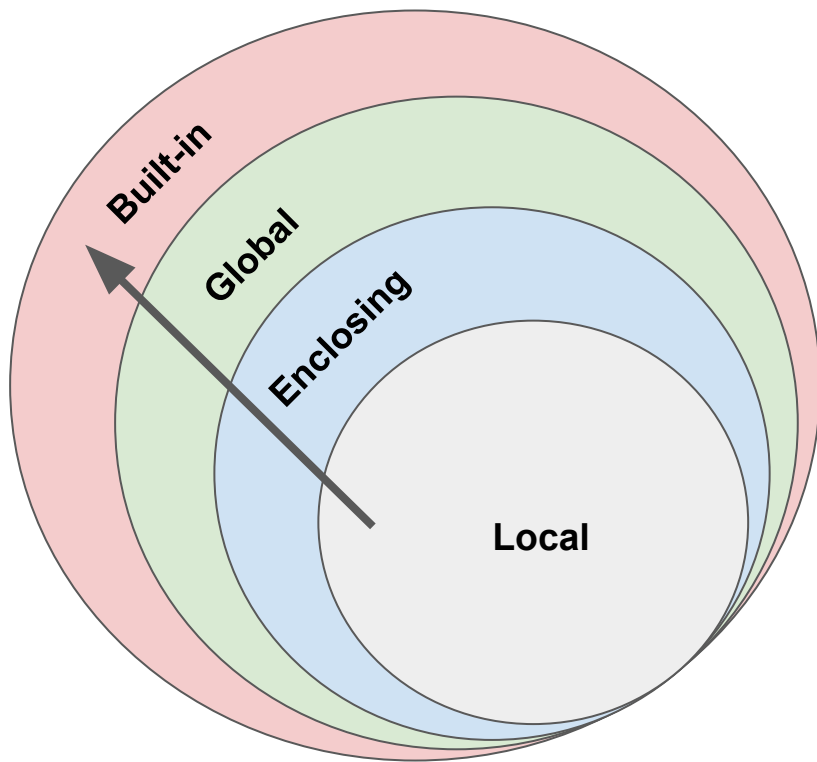
# The scope

## → *The LEGB rule*

- The scope is how variables, functions and in general “names” are resolved.
- The rule is the so called “LEGB”:
  - Local if defined “where you are”, as for example inside a function or code block.
  - Enclosing if defined in the upper levels with respect to “where you are”.
  - Global if defined globally (not covered here)
  - Built-in if defined inside Python itself

# The scope

→ *The LEGB rule*



# The scope

→ *And the functions*

- Explanatory, tricky example:

```
def sum_arg(arg):  
    arg += arg  
    return arg
```

```
arg = 1  
print(sum_arg(arg))  
2  
print(arg)  
1
```

# The scope

→ *And the functions*

- Explanatory, tricky example:

```
def sum_arg(arg):  
    arg += arg  
    return arg
```

```
arg = [1]  
print(sum_arg(arg))  
[1,1]  
print(arg)  
[1,1]
```

# The scope

→ *How to write good functions*

- Always operate on local variables only:

```
number = 5

def square():
    result = number*number
    return result
```

**NO**

```
def square(number):
    result = number*number
    return result
```

**YES!**

# The scope

→ *How to write good functions*

- Always return the result:

```
result = None  
  
def square(number, result):  
    result = number*number
```

**NO**

```
def square(number):  
    result = number*number  
    return result
```

**YES!**



# The built-ins

→ *What are the built-ins*

- The built-ins are the functions, keywords and objects that are always available
- This is because they are defined in Python itself
- They include all the operators and keywords seen so far
- Also all the errors are built-ins, and constants as True or False

# The built-ins

→ *Python built-in functions*

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

# Modules

→ *What are modules*

- Python modules are basically files containing Python definitions and statements.
- They can be organized in a structured, hierarchical way composing a package.
- Packages are the format in which nearly all Python libraries are distributed.
- Python provides a set of “pre-installed”, or built-in modules, which compose the so called “standard library”.

→ however, they need to be imported to be used

# Modules

→ *How to use modules*

- Does Python provide a square root function as a built-in? **NO**
- Does Python provide a square root function as part of the standard library? **YES!**  
→ as part of the **math** module.

```
import math  
math.sqrt(9)
```

OR

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

# End of part II

→ *Questions?*

**Next: exercise 2**

# Exercise 2

Write a function that sums all the numbers of a list.

- Name it “sum\_list” and accept a parameter for the list
- If the list is empty, the function must return “None”
- Think about how to handle non numerical values in the list, or a parameter which is not a list:
  - can you detect them and return “None”?

hint: have a look at the *type()* built-in