

# Functions: Structure of Functions and Applications

By: Tino Cestonaro

# What is a function?

- Mathematically speaking, a function is a mapping  $f$  from an input  $x$  to an output  $y$

$$f(x) = y$$

- E.g.,  $f(x) = 2x$  and  $f(4) = 8, f(12) = 24,$
- In a programming language, a pseudo code looks as follows:

`f(argument1, argument2, ...) = result`

```
def functionName(argument1, argument2, ...):
    do something
    return result
```

# Introduction to functions in Python

- A function is an **executable statement**
- Very useful when, e.g., a **set of operations is applied repeatedly** → key concept **don't repeat yourself (DRY)**
  - Example: verify whether an integer is odd/even for several integers
- A function can have several **arguments** that are evaluated in the **body** of a function
- Results obtained in body can be **returned** that could be used for further calculations
- A function has to be **called** to be executed, i.e., a definition of a function itself does not execute a function's body
- Let's be more specific on the next slides!

# Structure of Functions in Python (1) - Definition

- The keyword **def** in Python defines and initializes a function

```
def double_me(int1):  
    result = 2 * int1  
    return result
```

# Structure of Functions in Python (2) - Name

- The keyword is followed by the **function name**, here: **double\_me**

```
def double_me(int1):  
    result = 2 * int1  
    return result
```

- Convention (PEP8) in Python: always start with lowercase letters and use underscores to define the name of a function

# Structure of Functions in Python (3) - Arguments

- The name is followed by the **arguments** (input parameters) of the function in brackets, here: **(int1)**

```
def double_me(int1):  
    result = 2 * int1  
    return result
```

- A function
  - can have **no, a single or multiple arguments**
  - can have **default argument values**

```
def print_current_time():  
    print(time.time())
```

```
def expo(base, ex=2):  
    result = base**ex  
    return result
```

# Structure of Functions in Python (4) - Body

- The body of the function is defined as **all indented lines after** the line of the **definition statement**
- Note: the variable `result` belongs to the **local namespace** of the function `double_me` and thus, cannot be accessed outside the function's body.
- For instance, having defined `double_me` and executing `print(result)` throws an error because it is defined in a different namespace!

```
def double_me(int1):  
    result = 2 * int1  
    return result  
  
print(result)  
>> NameError: name "result" is not defined
```

**BUT:** `double_me` has a reference to the current global namespace since the global namespace is used when `double_me` is called (see next slide what *calling* means).

# Structure of Functions in Python (5) – Calling and Return Value

- There are several ways to **call** the function `expo`

```
def expo(base, ex=2):  
    result = base**ex # base to the power of ex  
    return result
```



# Structure of Functions in Python (5) – Calling and Return Value

- There are several ways to **call** the function `expo`

```
def expo(base, ex=2):  
    result = base**ex # base to the power of ex  
    return result
```

- Calling `expo`

## + The right way +

```
# 1 positional argument  
res = expo(4)  
print(res)  
>> 16
```

```
# 2 positional args  
res = expo(4, 3)  
print(res)  
>> 64
```

```
# 1 keyword argument  
res = expo(base=5)  
print(res)  
>> 25
```

```
# 2 keyword arguments  
res = expo(ex=3, base=5)  
print(res)  
>> 125
```

## - The wrong way -

```
# required arg missing  
res = expo()
```

```
# pos. arg follows keyword arg  
res = expo(base=6, 2)
```

```
# multiple args for base  
res = expo(2, base=3)
```

# Example: a function using conditional statements

```
def is_positive(number):  
    if number >= 0:  
        return True  
    elif number < 0:  
        return False  
  
print(is_positive(3))  
>> True  
  
print(is_positive(-10))  
>> False
```

# Control for the type of the input argument(s)

```
def double_me(int1):  
    if type(int1) is int:  
        result = 2*int1  
        return result  
    else:  
        print("int1 must be of type int.")  
        return  
  
print(double_me(3))  
>> 6  
  
print(double_me(4.5))  
>> int1 must be of type int.  
>> None
```

# Some exercises for writing functions in Python

- Write a function
  1. `sum_elems` that sums up all elements in a list and returns a scalar
  2. `square_elems` that squares all elements in a list and returns a list
- Assume that all elements in the list are of type numeric and, for the sake of simplicity, we do not control for other types at the moment.
- Afterwards, verify that these functions work properly by testing them on the example list that is provided in the code skeleton:

```
mylist = [1, 6, 20, 12]
```

# Solution

```
def sum_elems(list1):  
    # use a loop  
    sume = 0  
    for i in list1:  
        sume += i  
    return sume  
  
def square_elems(list1):  
    # use a list comprehension  
    sq = [i*i for i in list1]  
    return sq
```

- Does everyone understand the definition of `sq`?

## Cont'd: compute mean and variance

- Write a function `f_mean` that computes the average of a list. This function shall use the function `sum_elems` that you programmed previously:

$$\text{mean}((x_1, x_2, \dots, x_n)') = \frac{1}{n} \sum_{i=1}^n x_i$$

- Write a function `f_variance` that computes the variance of a list. This function shall use the function `f_mean` and `square_elems` that you programmed previously:

$$\text{var}((x_1, x_2, \dots, x_n)') = \frac{1}{n} \sum_i x_i^2 - \text{mean}((x_1, x_2, \dots, x_n)')^2$$

# Solution

```
def f_mean(list1):  
    average = sum_elems(list1) / len(list1)  
    return average  
  
def f_variance(list1):  
    v = f_mean(square_elems(list1)) - f_mean(list1)**2  
    return v
```

# Outlook: tip of the iceberg

- `lambda` operator for **anonymous functions**, i.e., a function that is not called by its name

```
square_me = lambda x: x*x
```

- `map` operator is used to apply a function to multiple elements in parallel instead of iterating over each element via a slower loop

- **Iterating over list elements:**

```
for i in [1, 6, 20]:  
    print(square_me(i))  
>> 1  
>> 36  
>> 400
```

- **Apply map operator to all list elements:**

```
res = list(map(square_me, [1, 6, 20]))  
print(res)  
>> [1, 36, 400]
```

- And many more (that are not part of this course)!



- **Functions**

- <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>
- <https://docs.python.org/3/library/functions.html>

- **Coding Style (PEP8)**

- <https://docs.python.org/3/tutorial/controlflow.html#intermezzo-coding-style>
- <https://www.python.org/dev/peps/pep-0008/>