# intro-to-programming-3

Henri Vandendriessche

05/10/2021

# Reminder

- Bash commands to navigate directories
- Useful commands:
  - **pwd** Print Working Directory. Print the path of of the current directory
  - **ls /folder** list all files of the current directory
  - **cd /folder1/folder2** moving into folder1 and folder2 at once.
  - **cd ..** moving out of a directory
- "Tab" to use the auto-completion
- Many more bash commands to use...

# So far

- Python
- Data types:
  - integer
  - float
  - string
  - boolean
- **If**, **For** and **While** loops:
  - syntax
  - indentation
- Data collections:
  - list
  - tuple
  - set
  - dictionary

# Today

- Exercices from last times
- Functions:
  - Pure functions
  - Procedures
- Excercices

# Functions

A function is a block of instructions that is given a name.

```
#  definition of a function named 'one_two'
def one_two():
    print(1)
    print(2)
    print('...')

one_two() # function calls 1
```

```
## 1
## 2
## ...
```

```
one_two() # function calls 2
```

```
## 1
## 2
## ...
```

Calling a function is like substituting the function call by its body.

If you do not call the function, it will never be executed.

-> Run this code in http://pythontutor.com/

What, in your opinion, is the interest of functions?

# Usefulness of functions

- Using functions avoids to duplicate code (i.e. by cutting and pasting). This facilitates the modification and correction of a program (errors are at a single place!)
- Using functions typically serves to make the code more readable (and maybe shorter).

# Definitions first !

functions must be defined before they are called

```
one_two()  # function call
one_two()
one_two()

def one_two():  # function definition
    print(1)
    print(2)
    print('...')
```

-> Run this code in python.

Remarks:

- A given script can contain several function definitions.
- As a convention, all functions definitions must be at the beginning of the script.

# Arguments

```
def hello(name):
    print('Hello, ' + name)

hello('Alice')
```

```
## Hello, Alice
```

```
hello('Bob')
```

```
## Hello, Bob
```

During the call hello('Alice'), the argument Alice is stored in the variable name.

-> run it in http://pythontutor.com/

Note: the variable name is created only during the execution of the function hello() (it is local to hello())

# Multiple arguments

```
def print_if_divisible(n, div):
  if (n % div == 0):
    print(n, ' is a divisible by ', div)

print_if_divisible(10, 5)
```

```
## 10  is a divisible by  5
```

```
print_if_divisible(11, 5)
```

-> Exercise: using the above function, find the divisors of 10, 15, 27, 33, 64, 100

# Return values

The functions we have seen so far executed actions.

A function can also return the result(s) of a computation

```
def func(x):
  y = 2 * x + 1
  return y

print(func(0.0))
print(func(1.0))
print(func(2.5))

# compute the values of func for x in [-10, _9, -8, ..., 8, 9, 10]
xs = range(-10, 11)
values = []
for x in xs:
  values.append(func(x))

# display them on a graphics
import matplotlib.pyplot as plt
plt.plot(xs, values)
plt.show()
```

# Boolean Functions

Boolean functions return True or False

```
def is_divisible(x, y):
    if (x % y == 0):
        result = True
    else:
        result = False

    return result

print(is_divisible(10, 5))
```

```
## True
```

-> Question: how could one “simplify” (shorten) the function is_divisible ?

# Returning “complex” objets

A function can return a t-uple, a list, a dictionary, . . .

```
def f(x):
    y1 = x + 1
    y2 = x * 3
    y3 = x ** 2 + 3
    return (y1, y2, y3)

f(2.0)
```

```
## (3.0, 6.0, 7.0)
```

Default values for arguments

It is possible to provide defaults values for arguments.

```
def message(name, msg='Hello'):
    print(msg + ' ' + name + '!')

message("Anna")
```

```
## Hello Anna!
```

```
message("Anna", "Gooodbye")
```

```
## Gooodbye Anna!
```

# Use of position or keyword

- In a function call, parameters are typically assigned to arguments based either on the position or on their names.

```
def f(a, b):
  print('a=', a)
  print('b=', b)

f(1 ,2)
```

```
## a= 1
## b= 2
```

```
f(2, 1)
```

```
## a= 2
## b= 1
```

```
f(b=2, a=1)    # but one can also use the names of arguments
```

```
## a= 1
## b= 2
```

# Scope of variables 1/2

-> Try the following code in http://pythontutor.com/

```
name = 'chris'

def hello(name):
    print('Hello, ' + name)

print(name)
```

```
## chris
```

```
hello('Alice')
```

```
## Hello, Alice
```

```
hello('Bob')
```

```
## Hello, Bob
```

```
print(name)
```

```
## chris
```

# Scope of variables 2/2

**local variables**

Arguments or variables defined inside the body of a function only exist while the function is executed. They are destroyed and the associated memory is freed.

**non-local variables**

Variables that have been created in the environment where the variable was called. functions can access them (if they are not shadowed)

Yet, this is bad practice and must be avoided except in a few cases.

Why ? Because one should be able to understand what a function is going to do only based on its call.

Read section "“Local and Global Scope”" in Automate the Boring stuff https://automatetheboringstuff.com/chapter3/

# functions can call other functions

Note that functions can call each other.

```
def func1():
  print(1)

def func2():
  func1()
  print(2)
  func1()

func2()
```

-> Predict the output of this script.

# functions can call other functions

Note that functions can call each other.

```
def func1():
  print(1)

def func2():
  func1()
  print(2)
  func1()

func2()
```

```
## 1
## 2
## 1
```

# Recursive functions

Recursive functions are function that contains calls to themselves:

For example:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

# Modules

Functions defined in a file myfunc.py in the current folder can be called from another python script.

```
### file "mymodule.py"
def hello(name):
  print("Hello ", name, "!")

### file "myscript.py"
import mymodule

mymodule.hello("Chris")
```

- modules (aka libraries) allow to reuse functions.
- Python comes with many modules, e.g. random, math, os.
- Anaconda adds scientific librariesnumpy, scipy

# if name == ‘main’

Many scripts will contain a series of functions and then the line

```
if __name__ == '__main__':
```

The condition is true only if the script is executed as a python script.

The functions in it can be reused with import script

# Exercices:

1- Define a function with two arguments — a string msg and a number nrepetitions — that prints msg, nrepetition times.

2- Read https://en.wikipedia.org/wiki/Fahrenheit and write a function that converts from Fahrenheit to Celsius, and another one that converts from Celsius to Fahrenheit

3- Define a function is_prime(x) which returns True if x is a prime number, else False. Use it to list all prime numbers below 1000.

4- Two taxi companies propose differents pricing schemes: Company A charges 4.80€ plus 1.15€ by km travelled. Company B charges 3.20€ plus 1.20€ by km travelled. Write a first function which, given a distance, returns the costs of both companies, and a second function that returns 'company A' and 'company B', the cheapest company for a given distance.

5- Write a function are_anagrams(word1, word2) that tests if two words are anagrams, that is contain the same letters in different orders.

# Exercice 1:

- Define a function with two arguments:
  - a string msg and a number nrepetitions
  - that prints msg, nrepetition times.

```
def print_n_times(msg, n):
    for x in range(n):
        print(msg)

print_n_times("test0", 0)
print_n_times("test1", 1)
```

```
## test1
```

```
print_n_times("test4", 4)
```

```
## test4
## test4
## test4
## test4
```

## Exercice 2:

- Read https://en.wikipedia.org/wiki/Fahrenheit and write a function that converts from Fahrenheit to Celsius, and another one that converts from Celsius to Fahrenheit

```
def Fahrenheit_to_Celsius(f):
    return (f - 32) * 5.0/9.0

def Celsius_to_Fahrenheit(c):
    return (c * 9.0/5.0) + 32

Fahrenheit_to_Celsius(15)
```

```
## -9.444444444444445
```

```
Celsius_to_Fahrenheit(15)
```

```
## 59.0
```

```
Fahrenheit_to_Celsius(-40)
```

```
## -40.0
```

```
Celsius_to_Fahrenheit(-40)
```

```
## -40.0
```

# Exercice 3:

- Define a function is_prime(x) which returns True if x is a prime number, else False. Use it to list all prime numbers below 1000.

```
def is_prime(n):
    if n < 3:
        return True
    for i in range(2, n):
        if (n % i) == 0:
            return False
    return True

print([x for x in range(1, 1000) if is_prime(x)])
```

```
## [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
```

## Exercie 4:

- Two taxi companies propose differents pricing schemes: Company A charges 4.80€ plus 1.15€ by km travelled. Company B charges 3.20€ plus 1.20€ by km travelled. Write a first function which, given a distance, returns the costs of both companies, and a second function that returns 'company A' and 'company B', the cheapest company for a given distance.

```
def costs(distance):
    price_A = 4.8 + 1.15 * distance
    price_B = 3.2 + 1.20 * distance
    return (price_A, price_B)

def cheapest_company(distance):
    a, b = costs(distance)
    if a < b:
        return 'Company A'
    else:
        return 'Company B'

for d in range(1, 50):
    print(f"{d} km -> " + cheapest_company(d))
```

```
## 1 km -> Company B
## 2 km -> Company B
## 3 km -> Company B
## 4 km -> Company B
## 5 km -> Company B
## 6 km -> Company B
```

# Exercie 5:

- Write a function are_anagrams(word1, word2) that tests if two words are anagrams, that is contain the same letters in different orders.

```
def are_anagrams(w1, w2):
    return sorted(w1) == sorted(w2)

print(are_anagrams('listen', 'silent'))
```

```
## True
```

```
print(are_anagrams('listen', 'speak'))
```

```
## False
```

# Even more exercices:

See

- https://pcbs.readthedocs.io/en/latest/representing-numbers-images-text.html
- https://pcbs.readthedocs.io/en/latest/building_abstractions_with_functions.html