Intro to programming 4

Henri Vandendriessche
henri.vandendriessche@ens.fr

2022-10-11

Terminal cheat sheet reminder

- Bash commands to navigate directories
  - Print Working Directory. Print the path of the current directory

```
pwd
```

  - List all files of the current directory

```
ls folder
```

  - Moving into folder1 and subfolder2 at once.

```
cd folder1/subfolder2
```

  - Moving out of a directory

```
cd ..
```

  - Going back and forth in the directory tree

```
cd ../../folder1/subfolder1
```

  - Going back to the root directory

```
cd ~
```

- "**Tab**" to use the auto-completion
- **Ctrl + C** to stop a program execution
- 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
- Python Standard library
  - Python modules
  - Python built-in functions

Today

- Functions:
  - Definition
  - Parameter and argument
  - Return value
  - Scope of variable
  - Module
  - Pure functions vs function vs procedures
- Exercises

Functions intro 1/2

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

Functions intro 1/2

- A function is a block of instructions that is given a name.
- If you call a function, you execute the code written in that function

```
#  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
## ...
```

Functions intro 2/2

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

Functions intro 2/2

- Calling a function is like substituting the function call by its body.
- If you do not call the function, it will never be executed.

Functions intro 2/2

- 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/

```
#  definition of a function named 'one_two'
def lorem_ipsum():
  print("Lorem ipsum dolor sit amet, consectetur adipiscing elit.")
  print(" Sed non risus.")
        print("Suspendisse lectus tortor, dignissim sit amet, adipiscing nec, ul
  print('...')

lorem_ipsum() # function calls 1
lorem_ipsum() # function calls 2
```

Functions intro 2/2

- 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/

```
#  definition of a function named 'one_two'
def lorem_ipsum():
  print("Lorem ipsum dolor sit amet, consectetur adipiscing elit.")
  print(" Sed non risus.")
        print("Suspendisse lectus tortor, dignissim sit amet, adipiscing nec, ul
  print('...')

lorem_ipsum() # function calls 1
lorem_ipsum() # function calls 2
```

- 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!)

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 !

- Function have a syntax You define a function using **def name():**

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name + ()** as for the built-in functions (ex: len(string))

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name + ()** as for the built-in functions (ex: len(string))
  - **:** Never forget the **:**

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name** + **()** as for the built-in functions (ex: len(string))
  - **:** Never forget the **:**

- Functions must be defined before they are called

```
one_two() # function calls 1
one_two() # function calls 2

def one_two():
  print(1)
  print(2)
  print('...')
```

NameError: name 'one_two' is not defined

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name** + **()** as for the built-in functions (ex: len(string))
  - **:** Never forget the **:**

- Functions must be defined before they are called

```
one_two() # function calls 1
one_two() # function calls 2

def one_two():
  print(1)
  print(2)
  print('...')
```

NameError: name 'one_two' is not defined

- Remarks:

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name** + **()** as for the built-in functions (ex: len(string))
  - **:** Never forget the **:**

- Functions must be defined before they are called

```
one_two() # function calls 1
one_two() # function calls 2

def one_two():
  print(1)
  print(2)
  print('...')
```

NameError: name 'one_two' is not defined

- Remarks:
  - A given script can contain several function definitions.

Definitions first !

- Function have a syntax You define a function using **def name():**
  - **def**
  - **name + ()** as for the built-in functions (ex: len(string))
  - **:** Never forget the **:**

- Functions must be defined before they are called

```
one_two() # function calls 1
one_two() # function calls 2

def one_two():
  print(1)
  print(2)
  print('...')
```

NameError: name 'one_two' is not defined

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

Arguments

- You can also give an input to your function. This input is called a parameter

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

hello('Alice')
```

```
## Hello, Alice
```

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

```
## Hello, Bob
```

Arguments

- You can also give an input to your function. This input is called a parameter

```
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.

Arguments

- You can also give an input to your function. This input is called a parameter

```
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/

Arguments

- You can also give an input to your function. This input is called a parameter

```
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

- If you can pass one, you can pass two or 10

```
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)
```

Multiple arguments

- If you can pass one, you can pass two or 10

```
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, write a script to find the divisors of 10, 15, 27, 33, 64, 100

Return values 1/3

- The functions we have seen so far executed actions.

Return values 1/3

- 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))
```

```
## 1.0
```

```
print(func(1.0))
```

```
## 3.0
```

```
print(func(2.5))
```

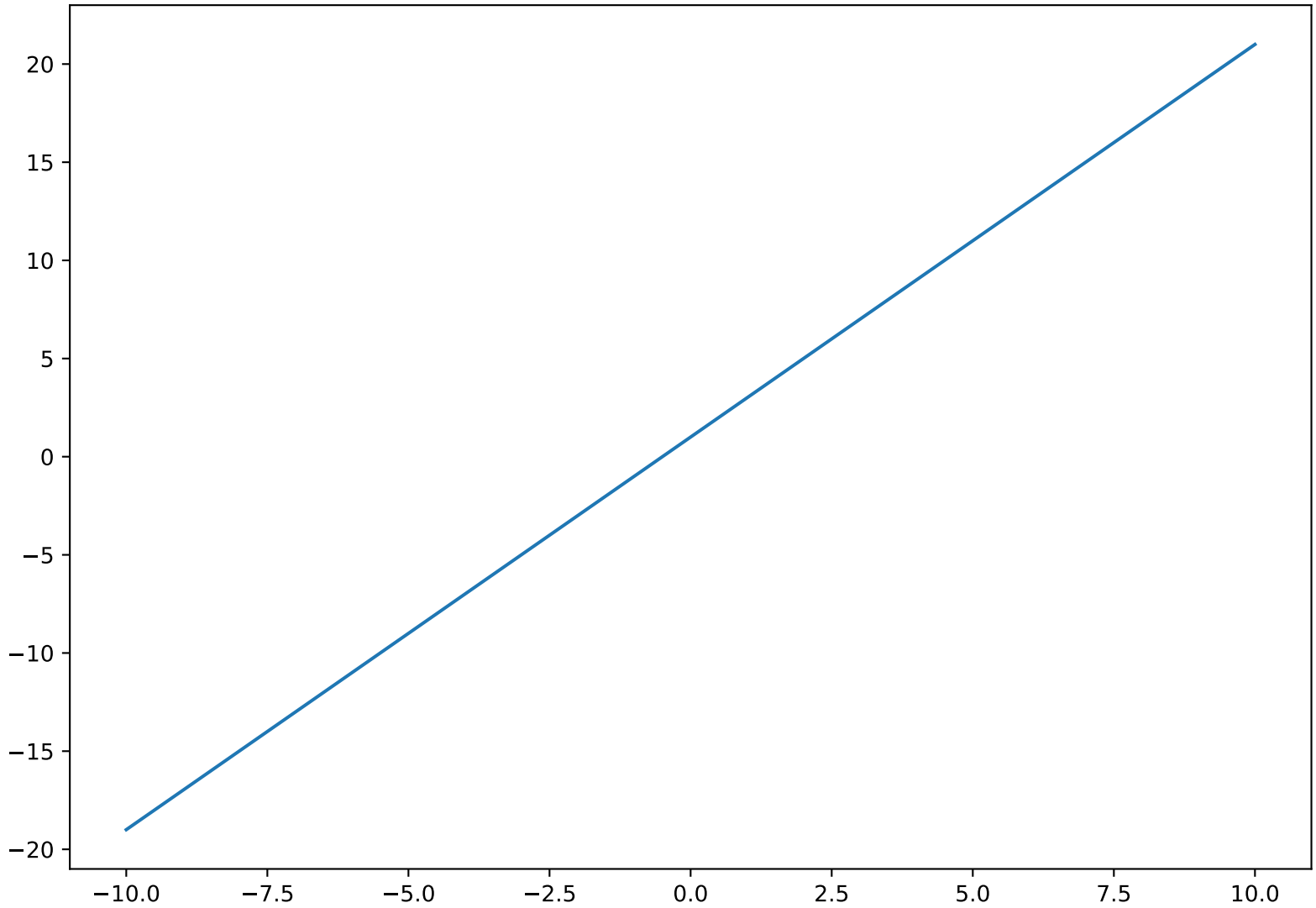
```
## 6.0
```

Return values 2/3

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

Return values 3/3



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” objects

- A function can return a tuple, a list, a dictionary, . . .

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

a,b,c =f(2.0)
print(a,b,c)
```

```
## 3.0 6.0 7.0
```

Returning "complex" objects

- A function can return a tuple, a list, a dictionary, ...

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

a,b,c =f(2.0)
print(a,b,c)
```

```
## 3.0 6.0 7.0
```

- Default values for arguments. You can insert a parameter by default if none are passed.

Returning "complex" objects

- A function can return a tuple, a list, a dictionary, ...

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

a,b,c =f(2.0)
print(a,b,c)
```

```
## 3.0 6.0 7.0
```

- Default values for arguments. You can insert a parameter by default if none are passed.
- 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
```

Methods vs Functions

- Methods are pretty much like functions
- Each data type has its own set of methods
- For example the **list** data type has:
  - **append()**
  - **sort()**
  - **index()**
  - **reverse()**
  - ...
- Contrary to functions, methods are called on a value (lists, dictionaries, sets ...)
- Example of the difference between method and function

```
def appending(list1,new_element):
  list1[-1] = new_element
  return list1

element = "g"
myList = ["a", "b", "c", "d", "e" ]
myList.append(element) # method
print(myList)
```

```
## ['a', 'b', 'c', 'd', 'e', 'g']
```

```
print(appending(myList,element)) # function
```

```
## ['a', 'b', 'c', 'd', 'e', 'g']
```

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 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
```

- Name is used for two different variables with the same name. It's possible (and confusing) because they don't have the same scope

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.

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 by a local variable)

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 by a local variable)
- Yet, this is bad practice and must be avoided except in a few cases.

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 by a local variable)
- 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.

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 by a local variable)
- 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

- We saw in the 3rd class that modules can let you access to function via the command:

```
import random
```

Modules

- We saw in the 3rd class that modules can let you access to function via the command:

```
import random
```

- But you can create your own module with your own functions. Just be sure to not use a name already used by the standard Library of Python

Modules

- We saw in the 3rd class that modules can let you access to function via the command:

```
import random
```

- But you can create your own module with your own functions. Just be sure to not use a name already used by the standard Library of Python

- 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

- We saw in the 3rd class that modules can let you access to function via the command:

```
import random
```

- But you can create your own module with your own functions. Just be sure to not use a name already used by the standard Library of Python

- 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 and have a clearer script.

Modules

- We saw in the 3rd class that modules can let you access to function via the command:

```
import random
```

- But you can create your own module with your own functions. Just be sure to not use a name already used by the standard Library of Python

- 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 and have a clearer script.
- Reminder Python comes with many modules, e.g. random, math, os.

if name == 'main'

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

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

- It will act differently if the script is the main script of if it is imported by another script
- The condition is true only if the script is executed as a python script.
- The functions defined before the **if name == 'main':** can be reused with import script
- Why does it work like that? If you are creating a library or a module, you might want to have some configuration tests or settings. At the same time you want it to be used as a module by others so that they only need the functions.

Functions vs Procedures

- A pure function is returning exactly the same value for the same parameter and has no side effect.

```
def addition(a,b):
  return a+b
x=addition(1,2)
print(x)
```

```
## 3
```

Functions vs Procedures

- A pure function is returning exactly the same value for the same parameter and has no side effect.

```
def addition(a,b):
  return a+b
x=addition(1,2)
print(x)
```

```
## 3
```

- A function return a value and calculate the value based on the input.

```
a=1
def addition(b):
  return a+b
x=addition(2)
print(x)
```

```
## 3
```

Functions vs Procedures

- A pure function is returning exactly the same value for the same parameter and has no side effect.

```
def addition(a,b):
  return a+b
x=addition(1,2)
print(x)
```

```
## 3
```

- A function return a value and calculate the value based on the input.

```
a=1
def addition(b):
  return a+b
x=addition(2)
print(x)
```

```
## 3
```

- A procedure execute commands

```
def addition(a,b):
  print(a+b)
addition(1,2)
```

```
## 3
```

Exercises:

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 taxis companies propose different pricing schemes: Company A charges 4.80€ plus 1.15€ by km traveled. Company B charges 3.20€ plus 1.20€ by km traveled. 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.

Even more exercises:

See

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