Python Programming Language

BY:

MOHAMED AZIZ TOUSLI

ZIED JARRAYA

About

- Interpreted language: Implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions: no pre-runtime translation.
- Compiled language: Implementations are compilers: translators that generate machine code from source code.
- Python is **dynamically typed language** → Identify data types itself + variables can change types
- Python is **strongly typed language** → Not possible to make operations between ≠ data types
- PS: Python 2 & 3 are different
- Be careful with indentation!!

```
Instruction1
Instruction2
        Instruction3
```

Operators & Variables

- #This is a comment
- +, -, *, /(division), #(Euclidian division), %(modulo)
- PS: 5 = int ; 5.0 = float
- string = “string” or ‘string’ or “””string””” or ‘’’string’’’
  - “””string””” let you write in several lines
  - ‘\’ undo the effect of a special character
  - ‘\n’ return to line
  - a,b=b,a #Permutation
  - a+=1 ⇔ a=a+1
  - x,y=5,6 ⇔ x=5 and y=6
  - x=y=5 ⇔ x=5 and y=5
  - longEquation=x+y \ #Write in several lines
    +z

Keywords to avoid for variables

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Conditions

- Comparison operators: <,<=,>,>=,==,!=; and, or, not

```
if """condition""":
  """code here"""

elif """condition""":
  """code here"""

else:
  """code here"""
```

- PS: Condition=Predicate=Boolean variable (**True** & **False**)

Loops

while """condition""":
  """code here"""

for element in sequence:
  """code here"""

- break #Interrupt a loop
- continue #Return to loop without executing next lines

Predefined Functions

- type(variableName) #Give type of variable
- a=input(“Write SPARTA!”) #Read string from screen , (no ‘\n’ by default)
- x=int(input(“Write a number”)) #Read integer from screen
- print(“This is”,a) #Print/Write on screen, (‘\n’ by default)
  - print (sep=“ “, end=“ “) #Choose specific separator and end element for print function
- import random; randrange(MIN, MAX); #Return a random number between MIN and MAX-1
- import random; randrange(MAX); #Return a random number between 0 and MAX-1

Functions

- def functionName(parameter1, arg2=value): #if arg2 not precised in main, arg2=value
  - #code here #predefined arguments must be in the end of arguments
  - return x,y
- PS: def fN(a=1,b=2); main fn(b=3,a=4) #We can do this if some arguments are predefined
- PS: We can overwrite a function (even a predefined one) by defining another one with the same name
- f=lambda arg1, arg2,… : instruction here #Another way to create functions
- Modules:
  - import moduleName → moduleName.functionName()
  - import moduleName as abbrev → abbrev.functionName()
  - from moduleName import functionName → functionName()
  - from moduleName import * → functionName()
  - from moduleName import functionName as abbrev → abbrev()
- help("moduleName"); help("moduleName.functionName") #Get more information
- def function(*normalParameters,**namedParameters): #Define function with ∞ parameters (Can be a list, a dictionary)
- function2 = function1 #Create function2 object that will take function1

Linux vs Windows

- #!/usr/bin/python3 #Specify python's directory in a python program in Linux
- # -*-coding:Type -* #Specify the type of encoding (for accents) ("Latin-1", Windows and "utf-8", Linux)
- import os; os.system("pause"); #Pause the system in the end of the program (For Windows)
- PS: Modules must exist in the same folder as the file that calls for it or in python folder
- __name__ is the name of the file where the instruction was done (« __main__ » for current file)
- If __name__=« __main __ »: #Do something if you are in current file
- Package: Simple set of packages, modules and functions (folder)

Try.. Except..

- try:
- #Code to try
- except errorName as returnedException:
- #Code to do in case of error (returnedException may be used)
- pass #Pass in case of error
- else:
- #Code to do in case of no error
- finally:
- #Code to do anyway

  - NameError: Variable was not defined before
  - ValueError: Operation on wrong variable
  - ZeroDivisionError: a/b and b=0

- assert test
  - If test=True → return True
  - If test=False → return AssertionError
- PS: **assert** is put into **try** code
- raise errorType("message to print")

String

- Functions = **methods**, variables = **attributes** → object.method();
- str = str() #Create a string
- str2 = str1.lower() #Turn string to lower case
- str2 = str1.upper() #Turn string to upper case
- print("message {} {}".format(variable0, variable1)) #Format a message respecting order
- print("message {1} {0}".format(variable0, variable1)) #Format a message with ≠ orders
- print("message {key}".format(key=variable0) #Format a message respecting order
- len(str) #Return length of string
- str[-1] #Return last character
- str[a:b] #Get substring from index a to index b-1
- PS: Other functions: count, find, replace…

List

- **List** = Sequence that can contain other objects of ≠ types
- List = **Mutable** → Elements can be replaced within it
- L = list(); L=[]; L=[element1,element2,…]; #Create a list
- L.append(newElement) #Add element in the end of a list (Return None)
- L.insert(index,newElement) #Add to specific position in the list
- L1.extend(L2); L1+=L2; #Add L2 to L1 ⇔ Concatenate L2 to L1
- del variableToDelete #Delete variable from memory
- L.remove(element) #Remove first occurrence of element in list
- for element in L: #Go through elements in list
- for i,element in enumerate(L): #Go through indexes and elements in list
- str.split(" ") #String to list
- " ".join(L) #List to string
- L2 = [operationOnElement for element in L1] #List comprehension with operator on element
- L2 = [element for element in L1 if conditionOnElement] #List comprehension with condition on element

- Tuple = Sequence non mutable
- T=(); T=tuple(); T=(1,); T=1,; T=(1,2); #Create tuple
  - L.sort() #Sort list
  - L=sorted(L) #Sort list

Dict

- **Dictionaries** are objects that can contain others with keys referring to them, not having an ordered structure
- D=dict(); D={}; D={key1:value1, key2:value2…}; #Create a dictionary
- D[newKey]=newValue #Add element do dictionary (If key doesn't exist, it'll be created else it'll be replaced
  - PS: Dictionary can have a tuple as a key → D[key1,key2]=value
  - PS: Dictionary is useful to create a map of ≠ function → D[keyFunction]() to call function
- value=D.pop(key) #Delete key and return corresponding value
- for key in D: #Go through keys in dictionary
- for key in D.keys(): #Go through keys in dictionary
- for key in D.values(): #Go through values in dictionary
- for key,value in D.items(): #Go through keys and values in dictionary

- Set = Sequence without repetition
- S=set(); S={element1,element2…};

File

- import os
  - os.chdir("C:/...") #Change working directory
  - os.getcwd() #Get current working directory
  - **Absolute path**: C:/BlaBla/BlaBla/... (Work globally)
  - **Relative path**: BlaBla/BlaBla (Work from current directory)
- fi = open("file.txt", "rwa") #Open file to read/write/append
- fi.close() #Close file
- fileContent = fi.read() #Read whole file (including '\n')
- fi.write("stringToWrite") #Write in file
- with open("file.txt", "rwa") as fi: #Open file as a function (to avoid errors)
- import pickle #Module to save an object in a file
  - pi = pickle.Pickler(fi); pi.dump(object); #Add object in file
  - pi = pickle.Unpickler(fi); object=pi.load(); #Load object from file
  - PS: Open file binary "rb", "wb" or "ab"

Local & Global Variables

- **Local space**: Contains the parameters that are passed to the function and the variables defined in its body
  - ➔ A function **cannot modify**, by assignment, the value of a variable outside its local space
  - parameter = newValue #The parameter will only be modified in the body of the function
  - parameter.methodToModify() #The object behind the parameter will be well and truly modified
- A variable is an identifier name, pointing to a reference of an object. The reference is a bit of its memory position
- object1 = object2 #Both items have reference on the same object (If object1 is changed with a method, object2 is)
  - ➔ object1 = type(object2) #Here, if object1 is changed with a method, object2 isn't
- id(object) #Return position of object in Python memory
- object1 is object2 #Return true if object1 and object2 have the same ID
- **Global variable**: A variable that can be called by a function, read and **modified** without being a parameter
- variable; def function(): global variable; #Define a global variable in a function

Classes

- **Object** is a data structure, like variables, that can contain other variables and functions (**attributes** and **methods**)
- **Class** is a form of data type, except that it defines functions and variables specific to the type
- class ClassName:

    def __init__(self, value): self.attribute = value #Constructor method

    → objectName = ClassName(value) #Create object in main { dir(objectName) #Return all attributes and methods in objectName }

    → objectName.attribute; objectName.attribute = newValue; #Get and set attribute

    def methodName(**self**, otherArguments): #Create method

    → objectName.methodName(args) ⇔ ClassName.methodName(**objectName**, args) #Call method in main

    def classMethodName(cls): #Create instance method/class method (Can be called even with an object from this class)

    def staticMethodName(): #Create static method

    classMethodName = classmethod(classMethodName); staticMethodName = staticmethod(staticMethodName); #Create class/static method

- **Special method/attribute**: Method/Attribute with "__specialName__" that is created by Python
  - **Attributes** are contained in the object
  - **Methods** are contained in the class that defines the object
- **Self** is used when you have to work in a method of the object on the object itself

Propetries

- **Encapsulation** is a principle that consists of hiding or protecting certain data from object
- **Accessors**: objectName.attributeName → objectName.getAttributeName()
- **Mutators**: objectName.attributeName = newValue → objectName.setAttributeName(newValue)
- attributeName = property(getMethod, setMethod, delMethod, helpMethod) #Deal with attribute while using accessors and mutators instead of touching _attributeName
- **Special methods**:
  - __init__ #Create an object
  - __del__ #Delete an object with 'del'
  - __repr__ #Print an object with 'print'
  - __str__ #Convert object to string with 'str'
  - __getattr__ #Get an attribute with '.'
  - __setattr__ #Set an attribute with '='
  - __delattr__ #Delete an attribute with 'del'
  - __hasattr__ #Tell if an attribute exists or not
  - __getstate__ #Change attributes before Pickler serialization
  - __setstate__ #Change attributes after UnPickler deserialization

- __getitem__ → object[index]
- __setitem__ → object[index] = value
- __delitem__ → del object[index]
- __contains__ → item in object
- __len__ → len(object)
- __add/sub…__ → object1 +/- object 2
- __radd/rsub__ → object 2 +/- object 1
- __iadd/isub__ → object1 +/-= object2
- __eq/ne…__ → object1 ==/!= object2

Sorting

- from operator import itemgetter, attrgetter
- sorted(l, reverse=False, key=itemgetter(i), key=attrgetter("attrName1","attrName2")) #Sort any type of sequence → Return new (i = to sort on)
- l.sort(reverse=False,) #Sort a list → Modify old
- Stability: The order of two elements in the list is not changed if they are equal
- Sorting chaining → Sort first by second criterion and then by first criterion

Heritage

- class daughterClass(motherClass1, motherClass2): #Create class that herits from another class
  - It works in this order: daughterClass → motherClass1 → motherClass2
- motherClass.myMethod(myObject) #Use a method from motherClass
- issubclass(daughterClass, MotherClass) #Verify subclass
- isinstance(object, class) #Verify instance
- class myException(Exception): #Create exception
- def __init __ (self, params): #Build exception
- def __str __ (self): #View exception

Iterators & Generators

- Iterator: To navigate the container object → Generator: To manipulate iterator easily
- myIterator = iter(object) #Call __iter__ to create iterator on object
- next(myIterator) #Call __next__ to go to next element
  - def __iter__(self): return myIterator(self) #→ In objectClass
  - def __init__(self, object): #→ In myIteratorClass
  - def __next__(self): #→ In myIteratorClass
    - if conditionToStopIteration: raise StopIteration
    - else treatementOnIterator
- def myGenerator(params):
  yield value #Return a value if next is called on iterator
  valueRecieved = (yield value) #Recieve valueSent
- myIterator = iter(myGenerator(params)) #Create iterator object from generator
- myGenerator = myGenerator(params) #Create generator object from generator
  - generator.close() #Interrupt generator
  - generator.send(valueSent) #Send a value to generator

PEP318

Decorators

- Decorator: Way to change the "default" behavior of a function/method → Metaprogramming
- def myDecorator1(myFunction):

```
    def functionModified(params): #Decorator calls functionModified(), which calls myFunction(), when myFunction() is called
        /* code /*
        return myFunction()
    return functionModified
```

- def anotherFunction(params_):

```
    def myDecorator2(myFunction):
        def functionModified(params):
            /* code /*
            return myFunction(params) #Can return anotherFunction()
        return functionModified
    return myDecorator2
```

- @myDecorator1
- @myDecorator2(params_)
- def myFunction(params):
- PS: @myDecorator ⇔ myFunction = myDecorator(myFunction)

MetaClasses

- **When we create an object:** className(params) → __new__ creates object → __init__(params) constructs object
  - __new__(cls, params) / __init__(self, params)
- PS: After all, a class is an object → Class « **type** »: mother of all calsses
  - className = type("className", (motherClass), {"attributeName": attribute, "methodName": method, …} #Create a new class
- class myMetaClass(type): #Create a metaclass

```
def __new__(metacls, className, (motherClass), {dict}):
        return type.__new__(metacls, className, (motherClass), {dict})
def __init__(cls, className, (motherClass), {dict}):
        type.__init__(cls, className, (motherClass, {dict})
```

- class myClass(metaclass=myMetaClass): #Create a class from metaclass

Regular Expressions (Regex)

- **Object**: Quickly and easily search in strings
  - ^x → str must start with x
  - x$ → str must end with x
  - xy* → str must contain x and, y atleast 0 times
  - xy+ → str must contain x and, y atleast 1 times
  - xy? → str must contain x and, y 0 or 1 time (optionnal)
  - x{1,5} → str must contain x from 1 to 5 times
  - [xy] → str must contain either x or y
  - [a-z] → str must contain a , b … or z
  - (xy) → str must contain x and y (group)

- import re #Module for regular expressions
  - re.search(r"subStr", "str") #Return **None** if subStr not in str, r to avoid '\' problems
  - re.sub(r"(group1)(group2)", r" \1\2", "str") #Find and replace group in str (\1 for first group, \2 for second group)
  - (?P<id>group) #Give name to group
  - re.compile(regex) #Store the returned value in a variable

Time

- **Epoch Unix**: 01/01/1970 at 00:00:00

- import time #import time module
  - time.time() #Return timestamp since Epoch Unix → Useful to substract two ≠ timestamps
  - time.localtime() #Return everything about local time
  - time.mktime(date) #Return timestamp from date
  - time.sleep(float/integerInSeconds) #Stop program for seconds
  - time.strftime("%A %d %B %Y %H:%M:%S") #Format date to string

- import datetime
  - datetime.now() #Return now's date and time
  - datetime.fromtimestamp(timestamp) #Return date and time from timestamp

System Programming

- import sys
  - **Standard input** = Keyboard ⇔ sys.stdin = sys.__stdin__ → input() ⇔ sys.stdin.read()
  - **Standard output** = Screen ⇔ sys.stdout = sys.__stdout__ → print() ⇔ sys.stdout.write("msg")
  - **Standard error** = Screen ⇔ sys.stderr = sys.__stderr__ → Traceback of an exception
  - sys.exit(0) #Exit the program
  - sys.stdout.flush() #Display the number right away
- **Signals**: When the system communicates with your program
- import signal
  - def signalFunction(signal, frame):
    /* code */
  - signal.signal(signal.signalName, signalFunction) #Connect signalName to signalFunction
  - sys.argv #List of passed in the command line when program is started; L=[‘program.py’, ‘arg1’, ‘arg 2’, …]
- import argparse #Interpret the arguments of the command line
  - parser = argparse.ArgumentParser() #Useful to configure our options to interpret
  - parser.add_argument("x", type=type, help="msg") #Add positional argument
  - parser.add_argument("-v", "--verbose", action="store_true", help="msg") #Add facultative option #action → args.verbose=True if option is precised
  - parser.parse_args() #Return an object with our arguments/options as attribute → args.x
- import os #Execute system command from Python
  - os.system("command") #Doesn’t capture the return displayed by the command
  - os.popopen("command") #apture the return displayed by the command

Math

- import math
  - pow(x,n) → Float ⇔ x**n → Integer
  - sqrt(x); exp(x); fabs(x);
  - cos(x); sin(x); tan(x); acos(x); asin(x); atan(x);
  - degreees(angleInRads); radians(angleInDegrees);
  - ceil(2.3) → 3; floor(2.3) → 2; trunc(2.3) → 2
  - math.pi=3.14; math.e=2.71
- from fractions import Fraction
  - q = Fraction(num,denum) = Fraction(α*num,α*denum)=Fraction('num/denum') #Create a fraction
  - Fraction.from_float(float) #Convert float to fraction
  - float(fraction) #Convert fraction to float
- import random
  - random.random() #Generate random float between 0 and 1
  - random.randrange(min, max, {gap}) #Generate random integer between min and max-1
  - random.randint(min, max, {gap}) #Generate random integer between min and max
  - random.choice(sequence) #Choose random value from sequence
  - random.shuffle(sequence) #Shuffle sequence
  - random.sample(sequence, n) #Return a sequence with n elements, selected randomly from sequence

Passwords

- from getpass import getpass
  - passWord = getpass({"msg"}) #Input a password
- import hashlib
  - hashlib.algorithms_guaranteed #Algorithms guaranteed by Python
  - hashlib.algorithms_available #Algorithms available on plateform
  - b = str.encode() ⇔ b=b'str' #Convert str to byte; **UTF-8 by default**
  - str = b.decode() #Convert byte to str; **UTF-8 by default**
  - passWord = hashlib.sha1(b) #Generate password with SHA1 algorithm
  - passWord.digest() #Encrypt password and return bytes object
  - passWord.hexdigest() #Encrypt password and return str object

PS: There is no decryption. Encrypt newPassWord and compare the encryptions

Sockets

- **Goal**: Connect a client to a server and transmit data from one to another
- import socket
- mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #1=Internet @'s; 2=TCP Type
- **PS**: Everything sent and received is in byte (not str)
- **Server**:
  - mySocket.bind((' ', port)) #Bind server to port [1024,65535]
  - mySocket.listen(5) #5 = Maximum number of connections it can receive on this port without accepting them
  - connectionWithClient, connectionInfos = mySocket.accept() #connectionInfos = (IP@, Exit Port)
  - connectionWithClient.send(b"msg") #Send msg
  - connectionWithClient.close()
- Client:
  - mySocket.connect(('IP@', port)) #IP@ can be localhost/127.0.0.0
  - msg = connectionWithServer.recv(1024) #Recieve in 1024 chunks (If > 1024 → Wait in buffer) #mySocket=connectionWithServer
  - connectionWithServer.close()
- mySocket.close()
- import select #Listen on a list of clients and return, after a specified time, the list of clients that have a message to receive
- rlist, wlist, xlist = select.select([mySocket], [], [], 0.05)

```python
import socket
import select

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

serveur_lance = True
clients_connectes = []
while serveur_lance:
    # On va vérifier que de nouveaux clients ne demandent pas à se connecter
    # Pour cela, on écoute la connexion_principale en lecture
    # On attend maximum 50ms
    connexions_demandees, wlist, xlist = select.select([connexion_principale],
        [], [], 0.05)

    for connexion in connexions_demandees:
        connexion_avec_client, infos_connexion = connexion.accept()
        # On ajoute le socket connecté à la liste des clients
        clients_connectes.append(connexion_avec_client)
    # Maintenant, on écoute la liste des clients connectés
    # Les clients renvoyés par select sont ceux devant être lus (recv)
    # On attend là encore 50ms maximum
    # On enferme l'appel à select.select dans un bloc try
    # En effet, si la liste de clients connectés est vide, une exception
    # Peut être levée
    clients_a_lire = []
    try:
        clients_a_lire, wlist, xlist = select.select(clients_connectes,
                [], [], 0.05)
    except select.error:
        pass
    else:
        # On parcourt la liste des clients à lire
        for client in clients_a_lire:
            # Client est de type socket
            msg_recu = client.recv(1024)
            # Peut planter si le message contient des caractères spéciaux
            msg_recu = msg_recu.decode()
            print("Reçu {}".format(msg_recu))
            client.send(b"5 / 5")
            if msg_recu == "fin":
                serveur_lance = False

print("Fermeture des connexions")
for client in clients_connectes:
    client.close()

connexion_principale.close()
```

Unitest

- import unittest
- class ClassTest(unittest.TestCase): #To create unitary test

```
def setUp(self): #Test initialization → Called before functionTest
def functionTest(self):
    function(params)
    assert*(params)
```

Context Manager

```
with self.assertRaises(NameError):
    function(params)
```

⇔ self.assertRaises(NameError, function, params)

- unittest.main() #To execute unittest code → It closes Python console (normal thing to happen)
  - '.' if test is valid; 'F' if test is failed; 'E' if error is occured
- **PS**: An assertion throws an exception that would be considered by unittest as an error
- python -m unittest #Used in command line to execute tests

Méthode	Explications
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`x is True`
`assertFalse(x)`	`x is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`
`assertRaises(exception, fonction, *args, **kwargs)`	Vérifie que la fonction lève l'exception attendue.

Threading

- **Goal**: Execute several instructions at the same time ➔ Parallel programming

```
from threading import Thread, RLock
lock= RLock() #Block the other threads → If another thread wants to use this resource, it must wait until it is released
class MyThread(Thread):
        def __init__(self, attributes):
                Thread.__init__(self)
        def run(self): #Code to execute while the thread is running
                with lock: #This locked part of only one thread at a time
thread_1 = MyThread(attributes); thread_2 = MyThread(attributes) #Create thread
thread_1.start(); thread_2.start() #Start thread and go to next line
thread_1.join(); thread_2.join() #Wait thread to end, in order to end the program
```

- **Problems with threading**:
  - Concurrent operations: An instruction may have unexpected results if it is called at the same time by different threads
  - Simultaneous access to resources: Result can be printed mixed

Tkinter

- **Goal**: Create graphic interface
- from tkinter import *
  - window = Tk() #Create a window
  - variable = StringVar()/IntVar() #Create a Tkinter variable (It has more features than a normal variable)
  - variable.get() #Get variable state/content
  - textLine = Entry(window, textvariable=variable) #Create **entry** widget
  - textLines = Text(window, textvariable=variable) #Create **text** widget
  - label = Label(window, text = "msg") #Create **label** widget
  - button = Button(window, text="msg", command=functionName) #Create **button** widget
  - radioButton = Radiobutton(window, text="msg", variable=variable, value="radioButtonID") #Create **radio button** widget
  - checkButton = Checkbutton(window, text = "msg", variable=variable) #Create **check button** widget
  - list = Listbox(window) #Create **list box** widget
    - list.insert(position, "msg") #Add element to list box (position = END)
    - list.curselection() #Return selected element in list box
  - widget["text"] / widget["text"] = "newMsg" #Get/Change widget text
  - widget.pack(fill=X/Y/BOTH, side="top") #Display label on window
  - frame = Labelframe/Frame(window, width=w, height=h) #Create **frame** = object to place widgets within
  - window.mainloop() #Start Tkinter loop

- **PS**: Personnalized commands **can't take parameters** if widgets are not in a class

```python
from tkinter import *

class Interface(Frame):

    """Notre fenêtre principale.
    Tous les widgets sont stockés comme attributs de cette fenêtre."""

    def __init__(self, fenetre, **kwargs):
        Frame.__init__(self, fenetre, width=768, height=576, **kwargs)
        self.pack(fill=BOTH)
        self.nb_clic = 0

        # Création de nos widgets
        self.message = Label(self, text="Vous n'avez pas cliqué sur le bouton.")
        self.message.pack()

        self.bouton_quitter = Button(self, text="Quitter", command=self.quit)
        self.bouton_quitter.pack(side="left")

        self.bouton_cliquer = Button(self, text="Cliquez ici", fg="red",
                command=self.cliquer)
        self.bouton_cliquer.pack(side="right")

    def cliquer(self):
        """Il y a eu un clic sur le bouton.

        On change la valeur du label message."""

        self.nb_clic += 1
        self.message["text"] = "Vous avez cliqué {} fois.".format(self.nb_clic)
```

```python
fenetre = Tk()
interface = Interface(fenetre)

interface.mainloop()
interface.destroy()
```

cx_Freeze

- **Goal**: Convert .py to .exe → Standalone version of program
- cd C:\python34\scripts; python.exe cxfreeze file.py; #Convert .py to .exe using cx_Freeze → 'dist' directory
- C:\python34\python.exe setup.py build #Convert .py to .exe using setup.py code → 'build' directory

```
1  """Fichier d'installation de notre script salut.py."""
2
3  from cx_Freeze import setup, Executable
4
5  # On appelle la fonction setup
6  setup(
7      name = "salut",
8      version = "0.1",
9      description = "Ce programme vous dit bonjour",
10     executables = [Executable("salut.py")],
11 )
```

PEP: Python Enhancement Proposal

- **PEP20** → General advice on development
- **PEP8** → Precise advice on the form of the code
  - Code line length: 79 characters / Text line length: 72 characters → Cut with ({[better than \ + Cut after operator
  - import a; import b NOT import a,b → Seperate between libraries: Standard library / Third-party library / Modules of your project
  - **No space** in: At the heart of parentheses, hooks and braces + Just before a comma, a semicolon or a colon + Just before the opening parenthesis that introduces the list of parameters for a function + Just before the opening hook indicates indexing or selection + More than one space around the assignment operator = (or other) to align with another statement
  - Always surround the following operators with a space (one before the symbol, one after): affectation, comparison, boolean, arithmetic (but don't when calling a function)
  - Comments must be complete sentences, starting with a capital letter. The point ending the sentence may be missing if the comment is short
  - Avoid using l, I, O as variable names
  - Module name: my_module | Package name: mypackage | Class name: MyClass | Exception name: MyError | Variable/Function name: my_function | Constant name: MY_CONSTANT
  - Programming conventions:
    - Object is None vs Object == None
    - isinstance(variable, type) vs type(variable) == type
    - boolean vs boolean == True | not boolean vs boolean == False
- **PEP257** → Documentation via docstrings
  - On one line: def function(params): " " "This function does this. " " "
  - On multiple lines: " " "This does that. \n \n Params : \n … \n \n" " "

Other Libraries

Graphic interface:

- PyQT

- PyGTK

- wx Python

Web developement:

- Django

- CherryPy

Networking:

- Twisted