


Cours Python

Concepts de base

Par Gilles Chagnon 

Date de publication : 14 juillet 2020

Ce cours a été initialement présenté à des étudiants de licence Projet Web de l'Université Pierre-et-Marie-Curie. Il présente la syntaxe de base de Python, ainsi que l'utilisation de la bibliothèque PyQt.

Pour réagir au contenu de ce tutoriel, un espace de dialogue vous est proposé sur le forum.
Commentez

I - Concepts de base.....	3
I-A - Introduction.....	3
I-A-1 - Utilisation en ligne de commande.....	3
I-A-1-a - Lancer et quitter Python.....	3
I-A-1-b - Entrées/sortie.....	3
I-A-1-c - Utilisation de fichiers.....	3
I-A-2 - Conventions.....	3
I-A-3 - Opérateurs.....	4
I-B - Listes et tableaux.....	4
I-B-1 - Introduction.....	4
I-B-2 - Manipuler une liste.....	4
I-B-2-a - Créer une liste.....	4
I-B-2-b - Ajouter des valeurs.....	4
I-B-2-c - Afficher une valeur.....	4
I-B-2-d - Supprimer des valeurs.....	5
I-B-2-e - Inverser les valeurs.....	5
I-B-2-f - 3. Obtenir des informations.....	5
I-B-2-g - 4. Copier une liste.....	6
I-C - Tests et boucles.....	6
I-C-1 - Tests.....	6
I-C-1-a - Test conditionnel if, if else.....	6
I-C-2 - Boucles.....	7
I-C-2-a - Boucle for.....	7
I-C-2-b - Boucle while.....	8
I-D - Fonctions.....	9
I-D-1 - Définition d'une fonction.....	9
I-D-1-a - Définition de base.....	9
I-D-1-b - Utilisation de valeurs par défaut.....	9
I-D-2 - Portée d'une variable.....	10
II - Modularisation.....	10
II-A - Import de modules.....	10
II-A-1 - Principe.....	10
II-A-2 - Importation avec nommage.....	10
II-B - Classes et objets.....	11
II-B-1 - Classe et objet simple.....	11
II-B-1-a - Définition d'une classe minimale.....	11
II-B-1-b - Ajout de propriétés.....	11
II-B-1-c - Ajout de méthodes.....	11
II-B-2 - Constructeur.....	12
II-B-3 - Encapsulation.....	12
II-B-4 - Héritage.....	12
III - PyQt 5.....	14
III-A - Introduction.....	14
III-B - Pour débiter.....	15
III-B-1 - Une fenêtre vide.....	15
III-C - Créer une application.....	16
III-C-1 - Le squelette.....	16
III-C-2 - Widgets.....	17
III-D - Mise en page.....	19
III-D-1 - Box layout.....	19
III-D-2 - Grille.....	20
III-D-3 - Application d'une mise en page à une fenêtre d'application.....	21
III-E - Gestion des événements.....	21
III-E-1 - Introduction.....	21
III-E-2 - Mise en œuvre.....	21
III-E-2-a - Sans gestionnaire.....	21
III-E-2-b - Avec un gestionnaire.....	22
IV - Remerciements Developpez.com.....	23

I - Concepts de base

Ce chapitre présente les principes de base de Python.

I-A - Introduction

I-A-1 - Utilisation en ligne de commande

I-A-1-a - Lancer et quitter Python

Python permet de produire des applications, mais est aussi disponible en ligne de commande pour tester le fonctionnement d'un script. Il suffit de le lancer en tapant `python` à une invite de commande. Pour quitter Python, il suffit de taper `quit()` ou `exit()` (ne pas oublier les parenthèses !)

I-A-1-b - Entrées/sortie

Il peut être utile d'afficher le résultat d'un calcul, une chaîne de caractères, etc. La fonction `print()` permet de le faire. Attention, en Python 2, les parenthèses n'étaient pas obligatoires. Elles le deviennent en Python3 et on écrira ainsi `print("Bonjour")`. Écrire `print("Bonjour", "tout le monde")` insère automatiquement un espace et produit « Bonjour tout le monde » à l'écran. Il est tout à fait possible de procéder à des calculs et les afficher : `print("Résultat :", 2+3)`.

Une saisie au clavier se fait à l'aide de la fonction `input`. Par exemple, `x=input("Entrez une chaîne de caractères : ")` permet d'affecter une chaîne de caractères saisie par l'utilisateur à la variable `x` à l'invite : « Entrez une chaîne de caractères : ». Attention, cette fonction, comme en JavaScript, renvoie une chaîne de caractères.

I-A-1-c - Utilisation de fichiers

Le code peut être écrit dans un fichier, portant traditionnellement l'extension `py`. Dans ce cas, le script est lancé par la commande `python nomFichier.py`.

Exercice : entrées/sorties.

- Écrire un programme qui demande la saisie de deux nombres, rappelle leurs valeurs puis affiche leur somme.

Correction

```
1. a=input("Saisissez un premier nombre : ")
2. b=input("Saisissez un second nombre : ")
3.
4. print("Voici les deux nombres entrés : ", a,b)
5.
6. print("... et leur somme : ", a+b)
```

I-A-2 - Conventions

Un commentaire commence par le caractère `#`. Tout ce qui suit jusqu'au retour à la ligne n'est pas interprété. Ainsi, pour écrire un commentaire de plusieurs lignes, il est nécessaire de faire débiter chaque ligne par `#`.

L'indentation demande une attention toute particulière en Python, car à la différence de la plupart des langages, les blocs d'instructions ne sont pas délimités par des accolades, mais sont définis par l'indentation. Si on revient à la ligne sans indenter dans un bloc d'instructions, ce dernier est considéré comme terminé. Nous y reviendrons plus loin.

Les noms de variables respectent les mêmes contraintes que dans les autres langages. **Attention** cependant : en Python, les variables sont en fait des références à des objets. L'opérateur d'affectation `=` permet de définir à quel objet un nom de variable fait référence. Concrètement, si on écrit :

- `a=1;b=a;print(a);print(b);b=2;print(a);print(b)`, alors on obtient en sortie 1 1 1 2 ;
- `a=[1];b=a;print(a);print(b);b[0]=2;print(a);print(b)`, alors on obtient en sortie [1] [1] [2] [2] !

Dans le premier cas en effet, à cause de l'affectation `b=2`, `a` et `b` à la fin de l'opération font référence à deux objets distincts. Dans le second, ils continuent à référencer le même objet, dont on change seulement le premier élément donc quand on modifie l'objet référencé par `b`, le résultat de l'évaluation de `a` change aussi...

I-A-3 - Opérateurs

Les opérateurs classiques restent `+`, `-`, `*` et `/`. En Python2, `/` renvoyait le quotient de la division euclidienne, mais ce n'est plus le cas en Python3.

D'autres opérateurs arithmétiques sont moins courants, comme `//` qui renvoie le quotient de la division euclidienne (par exemple `9/8` renvoie 1), et `**` qui permet d'élever à une puissance donnée (par exemple, `2**3` renvoie 8)

I-B - Listes et tableaux

I-B-1 - Introduction

Les listes en Python vont au-delà de la simple notion de tableau. Si on peut concevoir une liste sous la forme simple et habituelle `["pain", "beurre", "confiture"]`, en fait une liste peut incorporer des éléments de types différents, comme `[1, 2, "trois"]`...

I-B-2 - Manipuler une liste

I-B-2-a - Créer une liste

La manière la plus simple de créer une liste consiste à l'affecter à une variable, comme nous l'avons vu précédemment : `a=[1,4,6]` par exemple. Mais si on souhaite initialiser une liste à une suite arithmétique de nombres, on peut utiliser la fonction `range`. Par exemple, `b=list(range(1,44,7))` va créer une liste comportant tous les nombres compris entre 1 et 44 inclus, par pas de 7, soit `[1, 8, 15, 22, 29, 36, 43]`. Si le dernier argument est omis, le pas vaut 1 par défaut, et `c=list(range(1,44))` renvoie une liste comportant tous les nombres entiers compris entre 1 et 43 inclus. Enfin, `d=list(range(44))` renvoie une liste comportant tous les nombres entiers compris entre 0 et 43 inclus.

I-B-2-b - Ajouter des valeurs

On peut ajouter à une liste déjà créée soit un dernier item unique, à l'aide de la méthode `append()`, soit une autre liste avec la méthode `extend()`. Par exemple, `a=[1,2];a.append(3)` donne comme résultat `[1,2,3]`. Et `a=[1,2];a.extend([3,4])` donne comme résultat `[1,2,3,4]`. On peut d'ailleurs aussi utiliser l'opérateur `+` : `b=a+[3,4]` qui donne le même résultat (`[1,2,3,4]`).

On peut aussi « multiplier » une liste. Par exemple, `a=[1,2]*3` donne finalement pour `a` la valeur `[1,2,1,2,1,2]`.

I-B-2-c - Afficher une valeur

Supposons définie la liste `a=[2,3,4,5,6]`. Classiquement, `a[0]` nous permet d'obtenir le premier élément de la liste, ici 2. Mais Python permet facilement d'afficher tout ou partie de cette liste. Ainsi :

- `a[-1]` renvoie le dernier élément de la liste ;
- plus généralement, `a[-n]`, où `n` est un entier naturel, renvoie le `n`ième élément à partir de la fin de la liste ;
- `a[:2]` renvoie les deux premiers éléments de la liste, ici `[2,3]` ;
- `a[2:]` renvoie les derniers éléments de la liste à partir du deuxième exclu, ici `[4,5,6]`.

Exercice : création et affichage d'une liste.

- Écrire un programme qui :
 - crée une liste vide `a` ;
 - demande la saisie d'un nombre de départ, d'un nombre d'arrivée et d'un pas ;
 - ajoute à la liste `a` les trois nombres ;
 - ajoute à la liste `a` une autre liste, constituée à partir des nombres saisis (nombre de départ, nombre maximal, pas d'augmentation) ;
 - affiche la liste ;
 - affiche les pas derniers éléments de la liste.



Correction

```
1. a = []
2.
3. debut = input("Entrez le nombre entier de départ : ")
4. fin = input("Entrez le nombre entier de fin : ")
5. pas = input("Entrez le pas : ")
6.
7. a.extend([int(debut), int(fin), int(pas)]); print(a)
8.
9. b = list(range(int(debut), int(fin), int(pas)))
10.
11. a.extend(b)
12.
13. print(a)
14. print(a[-1*int(pas):])
```

I-B-2-d - Supprimer des valeurs

On peut supprimer un élément de liste de deux manières :

- par son index avec l'instruction `del`. Par exemple, si `a` vaut `[1,2,3,4]`, alors `del(a[1]);print(a)` renvoie `[1,3,4]` ;
- par sa valeur avec la méthode `remove`. Cela retire la première occurrence de la valeur dans la liste. Par exemple, si `a` vaut `[1,2,3,4]`, alors `a.remove(3);print(a)` renvoie `[1,2,4]` ;
- pour vider complètement la liste, on écrit `a[:]=[]`.

I-B-2-e - Inverser les valeurs

On peut aussi inverser l'ordre des éléments d'une liste avec la méthode `reverse()` : si `a` vaut `[1,2,3,4]`, `a.reverse();print(a)` renvoie `[4,3,2,1]`.

I-B-2-f - 3. Obtenir des informations

Plusieurs fonctions et méthodes permettent d'obtenir des informations sur une liste. Supposons par exemple que `a=list(range(1,5))` (autrement dit, `a` vaut `[2,3,4,5]`) :

- la longueur de la liste est donnée par la fonction `len`. Ici, `len(a)` renvoie 4;
- l'index de la première occurrence d'une valeur est donné par la méthode `index`. Par exemple, `a.index(4)` renvoie 2 ;

- le nombre d'occurrences d'un élément donné peut être obtenu par la méthode `count` : si `b=[1,1,1,3,1,4,1,1,3,6,1,2,3,1,4]`, alors `b.count(1)` renvoie 8 ;
- pour déterminer si un élément est présent dans une liste, on utilise le mot-clef `in`. Par exemple, ici `3 in a` renvoie `true` mais `7 in a` renvoie `false`.

I-B-2-g - 4. Copier une liste

Nous avons vu précédemment que si on écrit `a=[1,2,3,4]` puis `b=a`; `b[1]=6`, alors la valeur renvoyée par `a` est `[1,6,3,4]`. Afin de pouvoir copier une liste, il faut en fait affecter ses **valeurs** dans une nouvelle liste. Ainsi, si on écrit `b=a[:]`; `b[1]=6`, la valeur de `a` n'est pas changée au contraire de celle de `b`.

Exercice : manipulation de liste.

- Initialiser la liste `a` à la valeur `[1,1,1,3,1,4,1,1,3,6,1,2,3,1,4]`
- Puis effectuer les actions suivantes :
 - afficher `a` ;
 - afficher le nombre de 3 présents dans `a` ;
 - afficher la position du dernier 3 ;
 - retirer le dernier 3 de la liste ;
 - afficher `a` ;
 - afficher le nombre de 3 présents dans `a` ;
 - afficher la position du dernier 3.



Correction

```

1. a=[1,1,1,3,1,4,1,1,3,6,1,2,3,1,4]
2.
3. print(a)
4. print("Nombre de 3 :", a.count(3))
5. a.reverse()
6. print("Dernier 3 à la position", len(a)-a.index(3))
7.
8. a.remove(3)
9. a.reverse()
10.
11. print(a)
12. print("Nombre de 3 :", a.count(3))
13. a.reverse()
14. print("Dernier 3 à la position", len(a)-a.index(3))
  
```

I-C - Tests et boucles

I-C-1 - Tests

I-C-1-a - Test conditionnel if, if else

La syntaxe d'un test est simple, mais il faut prendre garde à l'indentation :

```

if test:
    #Instruction 1
    #Instruction 2
    #Instruction 3
#Fin du test...
  
```

Il en est de même pour un test `if... else`, les instructions `if` et `else` devant être alignées :

```
if test:
    #Instruction 1
    #Instruction 2
    #Instruction 3
else:
    #Instruction si test échoué 1
    #Instruction si test échoué 2
#Fin du test...
```

Exercice : tester

- Écrire un programme qui :
 - demande à l'utilisateur de saisir une chaîne de caractères ;
 - affiche un message signalant que la chaîne contient la lettre « e » si c'est le cas et un autre message sinon.



Correction

```
1. liste=input("Entrez une chaîne de caractères : ")
2.
3. if 'e' in liste:
4.     print('Cette chaîne contient au moins un \'e\')
5. else:
6.     print('Cette chaîne ne contient pas de \'e\')
```

I-C-2 - Boucles

I-C-2-a - Boucle for

La boucle **for**, en Python, a une syntaxe légèrement différente de celle des autres langages. Pour faire l'équivalent d'une boucle de 3 itérations, on écrira :

```
for i in range(3):
    print(i)
```

Si on veut parcourir les éléments d'une liste, on peut recourir à la fonction **len** :

```
a = ["pain", "beurre", "confiture", "miel"]
for i in range(len(a)):
    print(a[i])
```

Mais on peut aussi indexer directement la liste (cette syntaxe ressemble à ce qu'il est possible d'utiliser en JavaScript) :

```
a = ["pain", "beurre", "confiture", "miel"]
for i in a:
    print(i)
```

Exercice 1 : boucle for

Écrire un programme qui :

- 1 Crée la liste **a** de tous les nombres entiers inférieurs à 100 ;
- 2 Demande à l'utilisateur de saisir un nombre **diviseur** ;
- 3 Crée à partir de **a** la liste de tous les nombres inférieurs à 100 qui sont multiples de **diviseur**.

Par exemple, si l'utilisateur saisit 19, le programme renvoie [0, 19, 38, 57, 76, 95].



Correction

```
1. liste=list(range(100))
2. listeFinale=[]
3.
4. diviseur=int(input('Diviseur ? '))
5.
6. for i in range(len(liste)):
7.     if liste[i]%diviseur==0:
8.         listeFinale.append(i)
9.
10. print("Voici la liste des multiples de",diviseur,"inférieurs à 100 :",listeFinale)
```

Exercice 2 : boucle for difficile.

Écrire un programme utilisant des boucles for et les fonctions et méthodes de comptage et de détection de présence dans les listes, afin de construire progressivement la liste suivante pour un nombre d'étapes à saisir(c'est une **suite connue**) :

- 1 Le premier terme de la liste est 1. À cette étape, la liste est [1] ;
- 2 On compte ensuite le nombre de 1 de la liste. Il y en a 1. À cette étape, la liste devient [1,1,1] ;
- 3 On compte ensuite le nombre de 1, de 2 et de 3 de la liste. À cette étape, elle devient [1, 1, 1, 3, 1] ;
- 4 On continue... les étapes suivantes sont [1, 1, 1, 3, 1, 4, 1, 1, 3], puis [1, 1, 1, 3, 1, 4, 1, 1, 3, 6, 1, 2, 3, 1, 4], etc.



Correction

```
1. liste=[1]
2.
3. nb_iter=int(input("Entrez un nombre d'itérations : "))
4.
5. for i in range(nb_iter):
6.     listetemp=[]
7.     print(liste)
8.     for j in range(nb_iter**2):
9.         if j in liste:
10.            listetemp.append(liste.count(j))
11.            listetemp.append(j)
12.     liste.extend(listetemp)
13.
14. print(liste)
```

I-C-2-b - Boucle while

La boucle **while** a une syntaxe tout à fait similaire à celle d'autres langages...

```
i=0
while i<3 :
    print(i)
    i = i+1
print("Fin de la boucle")
```

Exercice 3 : boucle while

Écrire un programme qui :

- crée une liste des nombres entiers compris entre 3 et 15 inclus ;
- demande à l'utilisateur des nombres et tant que ces nombres sont présents dans la liste, les en retire ;
- arrête le traitement si le nombre n'est pas dans la liste et affiche la liste initiale, puis la liste finalement obtenue.



Correction

```
1. liste=list(range(3,16))
2. listeArchive=liste[:]
3.
4. filtre=int(input("Entrez le nombre à filtrer : "))
5.
6. while (filtre in liste):
7.     liste.remove(int(filtre))
8.     filtre=int(input("Entrez le nombre à filtrer : "))
9.
10. print(listeArchive)
11. print(liste)
```

I-D - Fonctions

I-D-1 - Définition d'une fonction

I-D-1-a - Définition de base

La syntaxe générale est la suivante :

```
def nomFonction (paramètres) :
    #Bloc d'instructions
```

Par exemple :

```
def additionne (x,y) :
    return x+y
```

Il faut noter que Python considère que toute fonction doit renvoyer un résultat.

I-D-1-b - Utilisation de valeurs par défaut

On peut très facilement indiquer des valeurs par défaut pour le jeu de paramètres d'une fonction. Il suffit de les spécifier dans la définition de la fonction :

```
def additionne (x=1,y=2) :
    return x+y
```

Si on appelle par exemple `additionne(3)`, alors le second paramètre est supposé présent et égal à sa valeur par défaut. Le résultat renvoyé finalement dans ce cas est donc 5.

Cette syntaxe permet de plus d'appeler les paramètres par leurs étiquettes. Par exemple, avec la même définition de fonction, `additionne(y=3)` renvoie 4 car x est supposé présent et valoir sa valeur par défaut, 1. Il est alors possible de **changer** l'ordre d'apparition des paramètres, par exemple `additionne(y=3,x=4)`.

Exercice : définition d'une fonction.

Définir une fonction prenant comme paramètres l'année **puis** le numéro de mois (par exemple, 3 pour mars) et qui renvoie le nombre de jours du mois correspondant. Les valeurs par défaut sont 2016 pour l'année et 4 pour le mois.

Tester cette fonction en ne spécifiant aucun paramètre, puis pour le mois de février 2015, enfin pour le mois de février 2016 en utilisant la valeur par défaut pour le paramètre d'année.



Correction

```
1. def nombreJours (an=2016, mois=4):
2.     listeJours=[31,28,31,30,31,30,31,31,30,31,30,31]
3.     if an%4==0:
4.         listeJours[1]=29
5.     return listeJours[mois-1]
6.
7. print ("nombreJours()",nombreJours())
8. print ("nombreJours(2015, 2)",nombreJours(2015, 2))
9. print ("nombreJours(mois=2)",nombreJours(mois=2))
```

I-D-2 - Portée d'une variable

Par défaut, en Python, les variables sont locales. Si on définit une variable nommée `maVariable` dans le corps principal du programme et qu'on utilise une variable nommée également `maVariable` dans une fonction, cette variable sera locale. Autrement dit, sa portée ne dépassera pas celle de la fonction et même si on redéfinit sa valeur à l'intérieur de la fonction, la valeur de la variable globale ne sera pas modifiée. Si l'on souhaite modifier la variable globale à l'intérieur de la fonction, il faut **déclarer** cette variable explicitement, sous la forme `global maVariable`.

On peut donc constater que ce fonctionnement est à l'opposé du JavaScript. Dans ce dernier langage en effet, les variables sont par défaut globales, et on restreint leur portée avec le mot-clef `var`. En Python, les variables sont par défaut locales, et on étend leur portée avec le mot-clef `global`.

II - Modularisation

Ce chapitre présente les bases de la programmation orientée objet en Python.

II-A - Import de modules

II-A-1 - Principe

Il est tout à fait possible de se créer des bibliothèques de fonctions pour éviter de dupliquer un travail déjà fait dans le cadre d'un projet ou bien pour pouvoir réutiliser des fonctions ou des définitions de classes (voir ci-après) dans des projets différents.

Pour cela, il faut simplement écrire la liste des fonctions dans un fichier `.py` (par exemple `mes_fonctions.py`), puis importer ce fichier avant toute autre opération dans le code, avec les mots-clefs `from... import` (dans ce cas, `from mes_fonctions import *`). Cette syntaxe comporte plusieurs variantes :

- `from mes_fonctions import *` importe telles quelles toutes les fonctions définies dans le fichier ;
- `from mes_fonctions import fonction1, fonction2` importe uniquement les fonctions `fonction1` et `fonction2`.

II-A-2 - Importation avec nommage

Afin d'éviter d'éventuels conflits de noms, d'une manière similaire aux espaces de noms XML, on peut expliciter un préfixe pour le module importé :

- `import mes_fonctions`. Dans ce cas, les noms des fonctions importées doivent être préfixés par `mes_fonctions` : `mes_fonctions.fonction1`, `mes_fonctions.fonction2`...
- `import mes_fonctions as mesFonc`. Dans ce cas, les noms des fonctions importées doivent être préfixés par l'alias `mesFonc`, qui peut être défini indépendamment du nom du fichier : `mesFonc.fonction1`, `mesFonc.fonction2`...

II-B - Classes et objets

II-B-1 - Classe et objet simple

II-B-1-a - Définition d'une classe minimale

La définition d'une classe est tout à fait similaire à celle d'une fonction. Par convention, on a l'habitude de mettre une majuscule à la première lettre du nom d'une classe.

```
class ExempleClasse:
    """Documentation de la classe""" #Définition de la classe

p = ExempleClasse()
```

Les `"""` après le nom de la classe servent à la documentation. `p` est une instanciation de la classe. Attention à cette étape à la duplication d'objets :

```
p1=ExempleClasse()
p2=ExempleClasse()
p3=p1
print(p1)
<__main__.ExempleClasse object at 0x7ff404091ba8>
print(p3)
<__main__.ExempleClasse object at 0x7ff404091b70>
print(p3)
<__main__.ExempleClasse object at 0x7ff404091ba8>
```

On remarque dans le code précédent que les variables `p1` et `p3` renvoient bien au même objet, à l'adresse mémoire `0x7ff404091ba8`.

II-B-1-b - Ajout de propriétés

On peut ensuite ajouter des propriétés à l'objet ainsi créé :

```
p1.x="valeur de x".
```

II-B-1-c - Ajout de méthodes

La définition de méthodes se fait dans celle de la classe :

```
class ExempleClasse:
    def methode1(self, x, z):
        self.x=2*x
        self.y=2*x*z
```

Le mot-clef `self` est obligatoire et doit apparaître comme premier argument.

On peut tout à fait définir des valeurs par défaut de la même manière que pour les fonctions :

```
class exempleClasse:
    def methode1(self, x=3, z=8):
        self.x=2*x
        self.y=2*x*z
```

II-B-2 - Constructeur

Le constructeur est une méthode portant un nom prédéfini et préservé, `__init__`, commençant et finissant par deux « *underscores* ». Comme toute méthode, le constructeur peut accepter des paramètres, avec ou sans valeur par défaut :

```
class Test:
    def __init__(self, t=4, z=5):
        self.x="varx"
        self.y="vary"
        self.t=t
        self.z=z
```

II-B-3 - Encapsulation

L'encapsulation consiste à masquer certaines propriétés afin qu'elles ne soient plus directement accessibles depuis l'extérieur de l'objet. Pour définir une propriété privée, on préfixe son nom, encore une fois, par deux « *underscores* » dans le constructeur :

class ClasseTest :

```
#Constructeur
def __init__(self):
    self.a=1
    self.b=2
    self.__c=3
    self.__d=4

#getters
def get_a(self):
    return self.a
def get_b(self):
    return self.b
def get_c(self):
    return self.__c
def get_d(self):
    return self.__d

#setters
def set_a(self, x):
    self.a=x
def set_b(self, x):
    self.b=x
def set_c(self, x):
    self.__c=x
def set_d(self, x):
    self.__d=x

test=ClasseTest()
```

Dans l'exemple précédent, les propriétés `test.a` et `test.b` sont de plus accessibles directement.

II-B-4 - Héritage

On peut facilement construire une hiérarchie de classes avec l'héritage. Par exemple :

```
class Parente :
    """Classe parente"""

    def __init__(self):
        self.__prop1=1
        self.__prop2=2
```

```
def get_prop1(self):
    return self.__prop1
def get_prop2(self):
    return self.__prop2

def set_prop1(self, x):
    self.__prop1=x
def set_prop2(self, x):
    self.__prop2=x

class Enfant(Parente) :
    """Classe enfant"""

    def __init__(self):
        Parente.__init__(self)
        self.__prop3=3

    def get_prop3(self):
        return self.__prop3
    def set_prop3(self, x):
        self.__prop3=x
```

Exercice : classes.

Définir une classe `Nombre`. Cette classe possède :

- deux propriétés privées, `__pImaginaire` et `__pReelle`, définissables grâce à un constructeur ayant pour valeurs par défaut 0 et 0 ;
- les getters et setters correspondants ;
- une méthode `additionne` qui prend en argument un objet de type `Nombre` et qui renvoie un autre objet de type `Nombre`, dont la partie réelle est égale à la somme de la partie réelle de l'objet auquel la méthode s'applique et de celle du nombre passé en argument, et dont la partie imaginaire est égale à la somme de la partie imaginaire de l'objet auquel la méthode s'applique et de celle du nombre passé en argument ;
- une méthode `affiche` qui affiche la partie réelle, puis la chaîne de caractères « $i \times$ », puis la partie imaginaire du nombre.

Tester le fonctionnement avec la séquence d'instructions suivante :

```
nb1=Nombre(1,2)
```

```
nb1.affiche()
```

```
nb2=Nombre(-5,8)
```

```
nb2.affiche()
```

```
nb1.additionne(nb2).affiche()
```

Définir une classe `Reel` qui hérite de `Nombre`. Cette classe :

- force `__pImaginaire` à être égale à 0 ;
- définit une méthode nommée `ent` qui renvoie la partie entière de la partie réelle du nombre. Il pourra être nécessaire de recourir à une fonction définie dans un module Python qu'il faudra importer...

Tester le fonctionnement avec la séquence d'instructions suivantes :

```
r1=Reel(5,8)
```

```
print(r1.get_lm())
```



Correction

```

1. import math
2.
3. class Nombre:
4.
5.     """Nombre complexe"""
6.
7.     def __init__(self, re=0, im=0):
8.         self.__pImaginaire=im
9.         self.__pReelle=re
10.
11.     def get_Im(self):
12.         return self.__pImaginaire
13.     def get_Re(self):
14.         return self.__pReelle
15.
16.     def set_Im(self, x):
17.         self.__pImaginaire=x
18.     def set_Re(self, x):
19.         self.__pReelle=x
20.
21.     def additionne(self, nb):
22.         resultat=Nombre()
23.         resultat.set_Im(self.__pImaginaire+nb.get_Im())
24.         resultat.set_Re(self.__pReelle+nb.get_Re())
25.         return resultat
26.
27.     def affiche(self):
28.         print(self.__pReelle, "+i×", self.__pImaginaire)
29.         return 0
30.
31. nb1=Nombre(1,2)
32. nb1.affiche()
33. nb2=Nombre(-5,8)
34. nb2.affiche()
35. nb1.additionne(nb2).affiche()
36.
37. class Reel(Nombre):
38.     """Nombre réel"""
39.
40.     def __init__(self, re=0, im=0):
41.         Nombre.__init__(self)
42.         self.__pReelle=re
43.         self.__pImaginaire=0
44.
45.     def ent(self):
46.         return math.floor(self.__pReelle)
47.
48. r1=Reel(5,8)
49. print(r1.get_Im())

```

III - PyQT 5


Ces cours abordent la réalisation d'interfaces graphiques.

III-A - Introduction

Python est un langage qui peut être utilisé pour réaliser des programmes évolués. Pour cela, il faut pouvoir réaliser des interfaces utilisateur. Il existe plusieurs bibliothèques à cet effet ; certaines ont été portées sur Python :

- Tkinter est installé par défaut avec Python. C'est une bibliothèque déjà ancienne, provenant de Tk ;
- PyQt permet de produire des interfaces graphiques utilisant la bibliothèque Qt 2. Elle a été remplacée par PyGObject, qui utilise la bibliothèque GTK 3 ;

- PyQt et PySlide recourent elles à Qt, qui est une bibliothèque disponible sous deux licences différentes en fonction de leur utilisation pour la réalisation d'une application commerciale ou pas.

Nous allons aborder dans ce cours  **PyQt**, pour laquelle il est plus facile de trouver un mode d'installation simple sous Windows. À la date d'écriture de ce cours, la bibliothèque est en version 5.5.1, compatible avec Python 3.4. Au-delà d'une bibliothèque pour la création d'une interface graphique, PyQt embarque aussi de nombreux composants, sous la forme de modules spécialisés. Nous n'utiliserons dans ce cours d'initiation que les modules `QtCore` et `QtGui`, parmi la vingtaine disponible.

III-B - Pour débiter...

III-B-1 - Une fenêtre vide...

PyQt5 permet de se simplifier la vie, et de coder simplement une interface utilisateur. Analysons l'exemple suivant...

fenetreVide.py

```
1. import sys
2. from PyQt5.QtWidgets import QApplication, QWidget
3.
4. monApp=QApplication(sys.argv)
5. w=QWidget()
6. w.resize(500,300)
7. w.move(500, 500)
8. w.setWindowTitle("Titre de fenêtre")
9. w.show()
10.
11. sys.exit(monApp.exec_())
```

Dans cet exemple, on commence par importer les classes `QApplication` et `QWidget` du module `PyQt5.QtWidgets`. On définit ensuite une nouvelle application (`monApp`), puis un « widget », auquel on donne une largeur de 500 pixels, une hauteur de 300 pixels, et que l'on place à 500 pixels du bord gauche de l'écran et 500 pixels à partir du haut. On lui affecte ensuite un titre, puis on le montre. L'instruction `sys.exit(monApp.exec_())` permet de quitter l'application en cliquant sur la croix telle qu'elle est définie par le système d'exploitation sur la fenêtre.

Si l'on veut concevoir de manière un peu plus orientée objet (ce qui permet de réutiliser un widget), on écrira plutôt :

fenetreVideOO.py

```
1. import sys
2. from PyQt5.QtWidgets import QApplication, QWidget
3.
4. class Fen(QWidget):
5.     def __init__(self):
6.         super().__init__()
7.         self.lanceUI()
8.
9.     def lanceUI(self):
10.         self.resize(500,300)
11.         self.move(500, 500)
12.         self.setWindowTitle("Titre de fenêtre")
13.         self.show()
14.
15. monApp=QApplication(sys.argv)
16. w=Fen()
17. sys.exit(monApp.exec_())
```



La méthode `super()` permet de remonter à la classe dont on hérite.

III-C - Créer une application

III-C-1 - Le squelette

Créer un *widget* n'est pas créer une application, mais un composant d'une application. Les *widgets* sont positionnés à l'intérieur d'une fenêtre principale. En voici un exemple.

squeletteappli.py

```

1. import sys
2. from PyQt5.QtWidgets import QMainWindow, QApplication, QAction, qApp
3.
4. class Principale(QMainWindow):
5.     def __init__(self):
6.         super().__init__()
7.         self.setUI()
8.
9.     def setUI(self):
10.        exitAction=QAction('&Exit', self)
11.        exitAction.setShortcut('Ctrl+Q')
12.        exitAction.setStatusTip("Quitter l'application")
13.        exitAction.triggered.connect(qApp.exit)
14.
15.        menu=self.menuBar()
16.        fichierMenu=menu.addMenu("&Fichier")
17.        fichierMenu.addAction(exitAction)
18.
19.        self.barreOutils=self.addToolBar('Quitter')
20.        self.barreOutils.addAction(exitAction)
21.
22.        self.setGeometry(300,300,500,250)
23.        self.setWindowTitle('Fenêtre principale')
24.        self.statusBar().showMessage('Barre de statut')
25.        self.show()
26.
27. monApp=QApplication(sys.argv)
28. fenetre=Principale()
29. sys.exit(monApp.exec_())
  
```




La classe `QAction` permet de définir des **actions**, que l'on peut attacher soit à une barre de menu (activée grâce à la méthode `menuBar`), soit à une barre d'outils (méthode `addToolBar`).

- La définition d'une action peut s'accompagner de son rattachement à un raccourci clavier, d'une information apparaissant dans la barre de statut (`statusTip`), mais surtout d'une action quand elle est activée (« triggered »). Ici, l'activation de l'action est rattachée à la méthode `quit` de l'objet prédéfini `qApp`, d'une manière similaire à l'attachement à un gestionnaire d'événements en JavaScript.
- On définit un menu par la méthode `menuBar`. On ajoute à l'objet ainsi défini des éléments, auxquels on attache un texte (le caractère `&` permet d'indiquer un raccourci clavier) ainsi qu'une action (ici celle définie précédemment)
- La barre d'outils a un fonctionnement similaire.
- Enfin, la méthode `statusBar` permet d'accéder à la barre de statut de l'application

La méthode `setGeometry` permet de spécifier la position et la taille de la fenêtre, en précisant dans cet ordre la position par rapport au coin supérieur gauche, largeur et hauteur.

III-C-2 - Widgets

Les widgets sont les composants de base de l'application. En voici quelques-uns...



```

widgets.py
1. import sys
2. from PyQt5.QtWidgets import QMainWindow, QApplication, QAction, qApp, QTextEdit, QPushButton,
   QHBoxLayout, QWidget, QVBoxLayout, QToolTip, QLineEdit, QLabel, QCheckBox, QComboBox
3.
4. class Principale(QMainWindow):
5.     def __init__(self):
6.         super().__init__()
7.         self.setUI()
8.
9.     def setUI(self):
10.        zoneTexte=QTextEdit()
11.        btnOK=QPushButton("OK", self)
12.        zoneLigne=QLineEdit()
13.        label=QLabel("Champ texte")
14.        case=QCheckBox("Case", self)
15.        combo=QComboBox(self)
16.
17.        btnOK.resize(btnOK.sizeHint())
18.        btnOK.setToolTip("Ceci est un bouton <i>OK</i>")
19.
20.        combo.addItem("Choix 1")
21.        combo.addItem("Choix 2")
22.        combo.addItem("Choix 3")
  
```

widgets.py

```

23.     combo.addItem("Choix 4")
24.
25.     hbox=QHBoxLayout()
26.     hbox.addStretch(1)
27.     hbox.addWidget(label)
28.     hbox.addWidget(zoneLigne)
29.     hbox.addWidget(zoneTexte)
30.     hbox.addWidget(btnOK)
31.     hbox.addWidget(case)
32.     hbox.addWidget(combo)
33.     vbox=QVBoxLayout()
34.
35.     w=QWidget()
36.     w.setLayout(hbox)
37.
38.     self.setCentralWidget(w)
39.
40.     #Définition des actions
41.     exitAction=QAction('&Exit', self)
42.     exitAction.setShortcut('Ctrl-Q')
43.     exitAction.setStatusTip("Quitter l'application")
44.     exitAction.triggered.connect(qApp.exit)
45.
46.     menu=self.menuBar()
47.     fichierMenu=menu.addMenu("&Fichier")
48.     fichierMenu.addAction(exitAction)
49.
50.     self.barreOutils=self.addToolBar('Quitter')
51.     self.barreOutils.addAction(exitAction)
52.
53.     self.setGeometry(300,300,500,250)
54.     self.setWindowTitle('Fenêtre principale')
55.     self.statusBar().showMessage('Barre de statut')
56.     self.show()
57.
58. monApp=QApplication(sys.argv)
59. fenetre=Principale()
60. sys.exit(monApp.exec_())
  
```



La plupart des noms sont explicites ; néanmoins voici quelques détails sur certaines méthodes :

- la méthode `resize` appliquée à un objet de type `QPushButton` permet de le redimensionner ;
- la méthode `sizeHint` laisse le soin à `Qt` de déterminer la taille optimale du bouton.

Nous allons revenir sur la mise en page, limitée ici.

III-D - Mise en page

III-D-1 - Box layout

Les classes `QHBoxLayout` et `QVBoxLayout` permettent d'aligner des contenus horizontalement ou verticalement. Combinés, cela permet une grande souplesse. Supposons que l'on veuille par exemple ajouter trois boutons en bas de la fenêtre, répartis régulièrement :



boxlayout.py

```

1. import sys
2. from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QHBoxLayout, QVBoxLayout
3.
4. class Principale(QWidget):
5.     def __init__(self):
6.         super().__init__()
7.         self.setUI()
8.
9.     def setUI(self):
10.
11.         btn1=QPushButton("Bouton1")
12.         btn2=QPushButton("Bouton2")
13.         btn3=QPushButton("Bouton3")
14.
15.         hbox=QHBoxLayout()
16.         hbox.addStretch(1)
17.         hbox.addWidget(btn1)
18.         hbox.addStretch(1)
19.         hbox.addWidget(btn2)
20.         hbox.addStretch(1)
21.         hbox.addWidget(btn3)
22.         hbox.addStretch(1)
23.
24.         vbox=QVBoxLayout()
25.         vbox.addStretch(1)
26.         vbox.addLayout(hbox)
27.
28.         self.setLayout(vbox)
29.
30.         self.setGeometry(300,300,500,250)
31.         self.setWindowTitle('Fenêtre principale')
32.
33.         self.show()
34.
35. monApp=QApplication(sys.argv)
36. fenetre=Principale()
37. sys.exit(monApp.exec_())
  
```



On commence par créer les trois boutons. On crée ensuite un *layout* horizontal avec `QHBoxLayout`. La méthode `addStretch` permet d'ajouter un espace de largeur variable et qui s'ajuste en fonction de la largeur de la fenêtre. Ici, on insère donc un espace variable, un bouton, un espace variable, un deuxième bouton, un espace variable, le dernier bouton et un dernier espace variable.

On crée ensuite un *layout* vertical, auquel on ajoute un espace variable puis le *layout* horizontal que l'on vient de créer. Comme on n'ajoute rien en dessous, le *layout* hbox sera toujours calé sur le bas de la fenêtre.

Une telle mise en page permet de s'assurer que les boutons restent en permanence régulièrement répartis et calés en bas de la fenêtre.

III-D-2 - Grille

Un autre système de mise en page commode est la grille, avec la classe `QGridLayout`.



gridlayout.py

```
1. import sys
2. from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QGridLayout
3.
4. class Principale(QWidget):
5.     def __init__(self):
6.         super().__init__()
7.         self.setUI()
8.
9.     def setUI(self):
10.
11.         btn1=QPushButton("Bouton1")
12.         btn2=QPushButton("Bouton2")
13.         btn3=QPushButton("Bouton3")
14.         btn4=QPushButton("Bouton4")
15.         btn5=QPushButton("Bouton5")
16.         btn6=QPushButton("Bouton6")
17.
18.         grille=QGridLayout()
19.         self.setLayout(grille)
20.         grille.addWidget(btn1, 1,1)
21.         grille.addWidget(btn2, 1,2)
22.         grille.addWidget(btn3, 1,3)
23.         grille.addWidget(btn4, 2,1)
24.         grille.addWidget(btn5, 2,2)
25.         grille.addWidget(btn6, 2,3)
26.
27.         self.setGeometry(300,300,500,250)
28.         self.setWindowTitle('Fenêtre principale')
29.
30.         self.show()
31.
32. monApp=QApplication(sys.argv)
33. fenetre=Principale()
34. sys.exit(monApp.exec_())
```



III-D-3 - Application d'une mise en page à une fenêtre d'application

Les *layouts* ne peuvent être appliqués qu'à des widgets, pas à une classe héritée de `QMainWindow` comme nous l'avons vu précédemment. Pour cela, il faut déclarer qu'un *widget* est le *widget* central de la fenêtre. Si on a défini un *layout* `miseEnPage` (cela pourrait être par exemple l'objet grille de l'exemple précédent), on écrira :

```
centre=QWidget()
centre.setLayout(miseEnPage)

self.setCentralWidget(centre)
```

III-E - Gestion des événements

III-E-1 - Introduction

Il existe une multitude d'événements susceptibles de changer l'état d'une application ou des données qu'elle traite. Ces événements peuvent résulter d'une action de l'utilisateur (comme un clic), ou bien de l'activation d'une connexion Internet, du signal envoyé par une horloge, etc. Dans tous les cas, trois éléments sont concernés :

- la source de l'événement, qui est l'objet changeant d'état (comme un bouton qui passe à l'état cliqué) ;
- l'objet événement lui-même, qui porte des informations sur ce changement d'état ;
- la cible de l'événement, c'est-à-dire l'objet qui va devoir réagir à la survenue de l'événement.

III-E-2 - Mise en œuvre

III-E-2-a - Sans gestionnaire

PyQt5 utilise les concepts de signal et de slot pour gérer les événements. Voici un exemple simple :

```
sliderlcd.py
1. import sys
2. from PyQt5.QtCore import Qt
3. from PyQt5.QtWidgets import QWidget, QApplication, QVBoxLayout, QLCDNumber, QSlider
4.
5. class Principale(QWidget):
6.     def __init__(self):
7.         super().__init__()
8.         self.setUI()
```

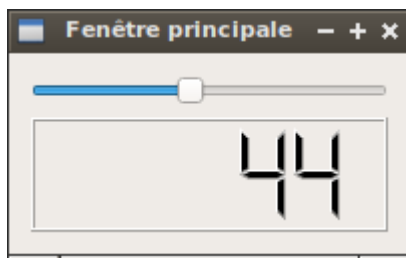


sliderlcd.py

```

9.
10. def setUI(self):
11.     lcd=QLCDNumber(self)
12.     slider=QSlider(Qt.Horizontal, self)
13.
14.     miseEnPage=QVBoxLayout()
15.     miseEnPage.addWidget(slider)
16.     miseEnPage.addWidget(lcd)
17.     self.setLayout(miseEnPage)
18.
19.     slider.valueChanged.connect(lcd.display)
20.
21.     self.setGeometry(300,300,200,100)
22.     self.setWindowTitle('Fenêtre principale')
23.
24.     self.show()
25.
26. monApp=QApplication(sys.argv)
27. fenetre=Principale()
28. sys.exit(monApp.exec_())

```



QLCDNumber et QSlider sont des classes permettant d'afficher un nombre au format LCD et une barre de réglage. L'événement valueChanged du slider est **connecté** à l'affichage du LCD.

III-E-2-b - Avec un gestionnaire

On peut également, tout comme en JavaScript, associer un gestionnaire d'événement :



messagebox.py

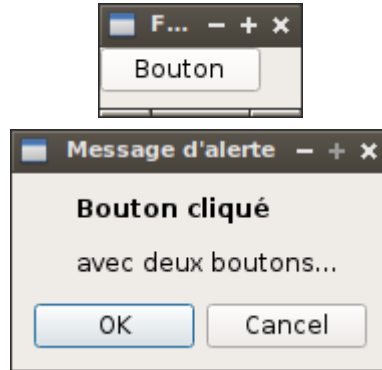
```

1. import sys
2. from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QGridLayout, QMessageBox
3.
4. class Principale(QWidget):
5.     def __init__(self):
6.         super().__init__()
7.         self.setUI()
8.
9.     def afficheMessage(self):
10.         message=QMessageBox()
11.         message.setText("<b>Bouton cliqué</b>")
12.         message.setInformativeText("avec deux boutons...")
13.         message.setWindowTitle("Message d'alerte")
14.         message.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
15.         message.exec()
16.
17.     def setUI(self):
18.
19.         btn=QPushButton("Bouton", self)
20.
21.         btn.clicked.connect(self.afficheMessage)
22.
23.         self.setGeometry(300,300,100,30)
24.         self.setWindowTitle('Fenêtre principale')
25.

```

messagebox.py

```
26.     self.show()
27.
28. monApp=QApplication(sys.argv)
29. fenetre=Principale()
30. sys.exit(monApp.exec_())
```



On associe cette fois-ci au clic sur le bouton le gestionnaire `afficheMessage`, qui affiche des informations diverses dans une boîte...

IV - Remerciements Developpez.com

L'équipe de la Rédaction de Developpez.com tient à remercier Hervé Troadec (alias **naute**) pour la mise au gabarit de ce tutoriel sur le langage Python, **escartefigue** pour la relecture typographique et orthographique et **Malick** pour la mise en ligne ainsi que la publication sur les différentes rubriques dédiées au langage python.