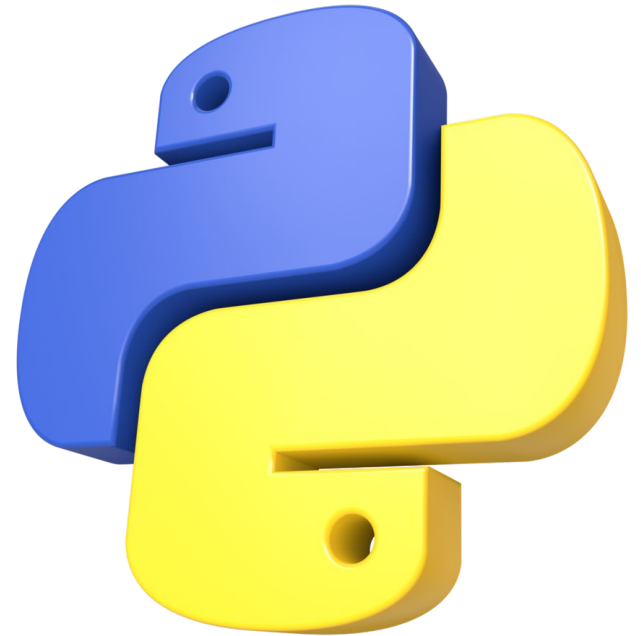


L'essentiel de Python.



Tour de table des stagiaires.



- Nom & prénom
- Expériences
- Précisez vos attentes pour cette formation

Présentation du formateur – **Didier Curvier MALEMBE.**

- **Ingénieur Logiciel en ESN**
 - > Capgemini Technology Services
 - > *Projet SNCF Connect*
 - > *Projet RM (Revenue Management) , etc...*
- **Consultant Freelance**
 - > Secteur public / TPE-PME
- **Formateur partenaire**
 - > ORSYS Formation
 - > FITEC
 - > AFPA
 - > PMN
 - > Cyborg Intelligence – IB Cegos...
- **Co-founder ELITIS CONSULTING**

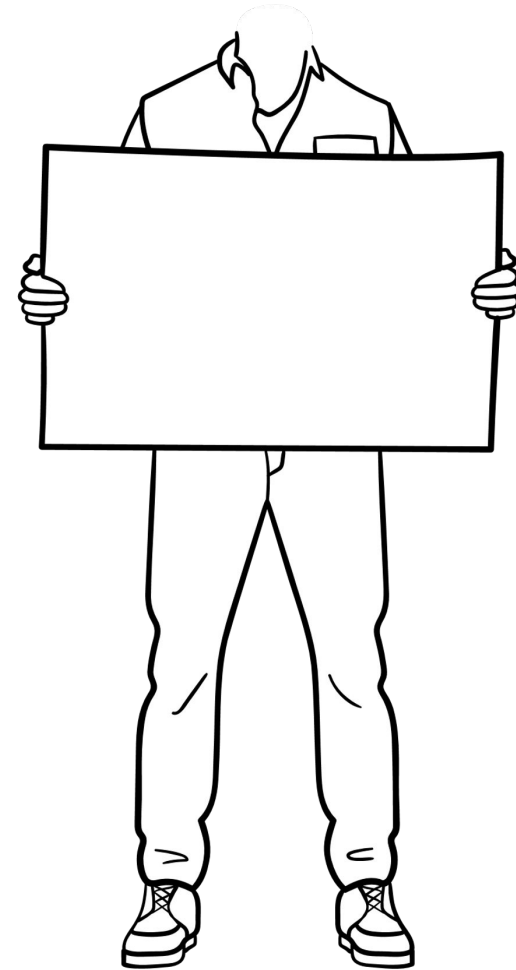
Les objectifs pédagogiques

À l'issue de la formation, le participant sera en mesure de :

- Maîtriser la syntaxe du langage **Python**
- Acquérir les notions essentielles de la programmation objet
- Connaître et mettre en œuvre les différents modules **Python**
- Concevoir des interfaces graphiques
- Mettre en œuvre les outils de test et d'évaluation de la qualité d'un programme **Python**

Table des matières

- I.** Syntaxe du langage Python
- II.** Approche Orientée Objet
- III.** Programmation Objet en Python
- IV.** Utilisation StdLib
- V.** Outils QA
- VI.** Création IHM TkInter
- VII.** Interfaçage avec C
- VIII.** Conclusion





Le langage Python : Historique & concepts de base.

Caractéristiques du Python – Implémentations – Mode de production... .

Python KESAKO ?

Python est un langage de programmation **interprété**, **multiparadigme** et **multiplateforme**.

Il est particulièrement apprécié pour le calcul scientifique avec **Numpy** et plus généralement pour les sciences et techniques avec **SciPy** et aussi à cause de sa facilité à être interfacé avec **C** et **Java**.

Python en quelques mots :

- 1991 : Première version du langage.
- Auteur : **Guido van Rossum**
- Paradigme : Objet, impératif et fonctionnel.
- OS : Multiplateforme.
- Packages : **196.000** packages en 2019 et plus de **453.000** en 2023.

Quelques caractéristiques du langage Python.

- ❑ **Portabilité** : Python est multiplateforme – compatible avec tous les OS (UNIX – MacOS – Windows).
- ❑ **Gratuit** : Utilisable sans restriction dans des projets commerciaux.
- ❑ **Simplicité du langage** : Permet la rédaction rapide de programmes compacts et lisibles.
- ❑ **Multiparadigme** : Python prend en charge plusieurs paradigmes de programmation – orienté objet – fonctionnel – procédural.
- ❑ **Prise en charge du multithreading**
- ❑ **Milliers de bibliothèques disponibles** : Plus de 200.000 packages disponibles.

Implémentations du langage Python.

- ❑ **CPython** : Classic Python, codé en C, portable sur différents systèmes
- ❑ **Python3000** : Python 3, la nouvelle implémentation de CPython
- ❑ **Jython** : ciblé pour la JVM (utilise le bytecode de JAVA)
- ❑ **IronPython** : Python.NET, écrit en C#, utilise le MSIL (MicroSoft Intermediate Language)
- ❑ **Stackless Python** : élimine l'utilisation de la pile du langage C (permet de récurser tant que l'on veut)
- ❑ **Pypy** : projet de recherche européen d'un interpréteur Python écrit en Python

Les modes d'exécution.

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes :

Le mode Interactif :

Permet de tester un grand nombre de fonctionnalités grâce à l'interpréteur python embarqué dans IDLE qui exécute la boucle d'évaluation.

```
>>> 5 + 3
8
>>>
```

Affichage d'une invite (*prompt*)

read : l'utilisateur tape une expression

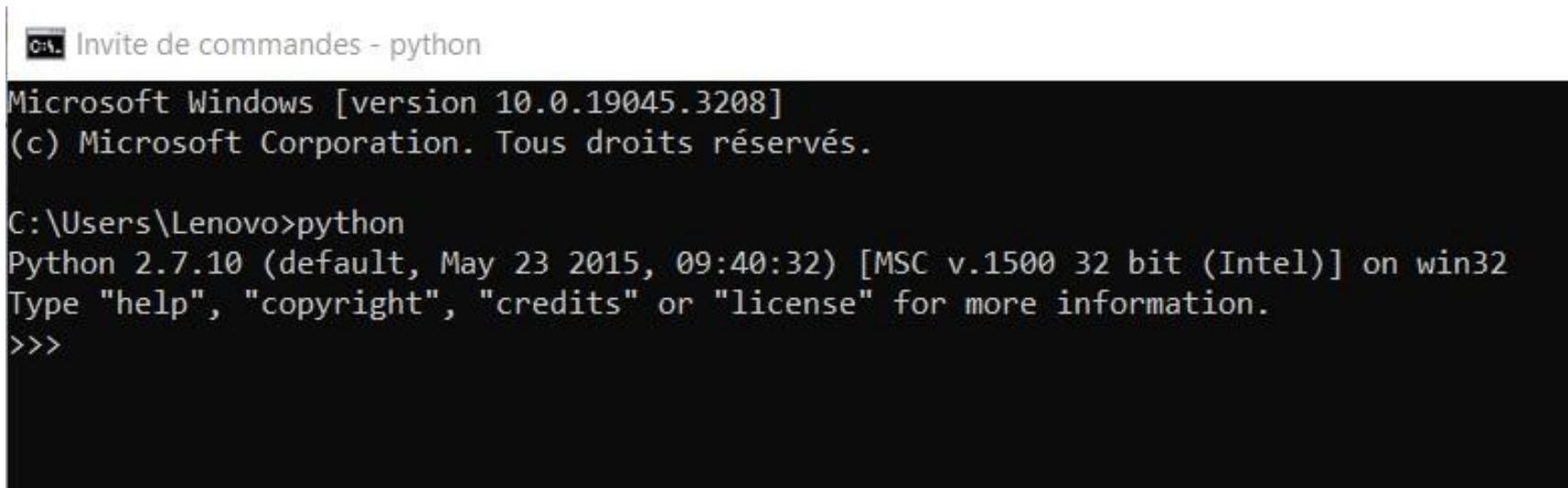
eval et **print** : calcul et affichage du résultat

Réaffichage d'une invite

The diagram shows the interactive Python shell loop. It consists of three lines of text: a prompt followed by an expression, the result of the expression, and a new prompt. Arrows point from descriptive text to each part of the loop. The first arrow points to the prompt '>>>' and is labeled 'Affichage d'une invite (prompt)'. The second arrow points to the expression '5 + 3' and is labeled 'read : l'utilisateur tape une expression'. The third arrow points to the result '8' and is labeled 'eval et print : calcul et affichage du résultat'. The fourth arrow points to the second prompt '>>>' and is labeled 'Réaffichage d'une invite'.

Les modes d'exécution – Mode Interactif.

L'interpréteur peut-être lancé depuis la ligne de commande (dans un shell linux ou dans une fenêtre DOS sous Windows). Il suffit de taper la commande python comme illustré ci-dessous.



```
C:\Users\Lenovo>python
Microsoft Windows [version 10.0.19045.3208]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\Lenovo>python
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Cet interpréteur peut tout de suite être utilisé comme une calculatrice – on parle aussi du mode calcul de python.

Les modes d'exécution – Mode script Python.

On enregistre un ensemble de commande python dans un fichier grâce à un éditeur que l'on exécute au moyen d'une commande.

```
Lenovo@DESKTOP-C4G64N9 MINGW64 ~/Documents/Python-scripts (master)
$ python3 exercice1.py
7*1 = 7
7*2 = 14
7*3 = 21
7*4 = 28
7*5 = 35
7*6 = 42
7*7 = 49
7*8 = 56
7*9 = 63
7*10 = 70
7*11 = 77
7*12 = 84
7*13 = 91
7*14 = 98
7*15 = 105
7*16 = 112
7*17 = 119
7*18 = 126
7*19 = 133
7*20 = 140
```



Syntaxe du langage Python.

Identifiant et mots clés – types de données – variables et affectation ...

Les identifiants et les références en python.

En Python, les notions d'identifiant (identifier) et de référence (reference) sont essentielles pour comprendre comment les objets sont manipulés et stockés en mémoire.

Identifiant (Identifier)

Un **identifiant** en Python est un nom donné à une variable, une fonction, une classe ou tout autre objet.

Les **identifiants** sont utilisés pour accéder et manipuler les objets en mémoire.

Les **identifiants** doivent suivre certaines règles :

- Ils peuvent contenir des lettres, des chiffres et le caractère souligné (_) (mais ne peuvent pas commencer par un chiffre).
- Ils sont sensibles à la casse, ce qui signifie que **"maVariable"** et **"mavARIABLE"** sont considérés comme deux identifiants distincts.
- Certains mots réservés, tels que **"if"**, **"else"**, **"while"**, ne peuvent pas être utilisés comme identifiants, car ils ont une signification spéciale dans le langage.

Les identifiants et les références en python.

Exemple d'identifiant en Python :

```
ma_variable = 42
```

Dans cet exemple, "**ma_variable**" est l'**identifiant** qui est utilisé pour faire référence à l'objet entier (42) en mémoire.

Référence (Reference) : En Python, la plupart des variables ne stockent pas directement la valeur d'un objet, mais plutôt une référence vers l'emplacement mémoire où cet objet est stocké.

Lorsque vous affectez une variable à un objet, vous créez en réalité une référence vers cet objet.

Cela signifie que plusieurs variables peuvent faire référence au même objet en mémoire, ce qui permet de partager des données entre différentes parties de votre code.

Les identifiants et les références en python.

Exemple de référence en Python :

```
liste1 = [1, 2, 3]
```

```
liste2 = liste1 # Les deux variables font référence au même objet liste en mémoire
```

```
liste1.append(4)
```

```
print(liste2) # Affichera [1, 2, 3, 4] car les deux variables référencent le même objet
```


Les identifiants et les références en python.

Dans cet exemple, "**liste1**" et "**liste2**" font référence au même objet de liste en mémoire. Lorsque nous modifions "**liste1**", ces modifications sont également visibles à travers "liste2" car les deux variables pointent vers le même emplacement mémoire.

En résumé, les **identifiants** sont des noms donnés aux objets en Python, tandis que les **références** sont des liens vers l'emplacement mémoire où ces objets sont stockés.

Si l'on souhaite copier liste 1 sans garder la référence on utilise la fonction **copy()**

```
liste2 = liste1.copy()
```

Les commentaires en python.

Les commentaires sont essentiels pour documenter le code python.

Commentaires en Python :

Les **commentaires** sont des annotations dans le code source qui ne sont pas exécutées par l'interpréteur Python. Ils sont utilisés pour expliquer le code, ajouter des notes ou désactiver temporairement certaines parties du code.

En Python, les **commentaires** commencent par le caractère dièse (**#**) et s'étendent jusqu'à la fin de la ligne. Tout ce qui suit le **#** sur la même ligne est ignoré par l'interpréteur.

Les **commentaires** sont utiles pour rendre votre code plus lisible et compréhensible, pour documenter votre code afin que d'autres développeurs (ou vous-même à l'avenir) puissent comprendre son fonctionnement, et pour désactiver temporairement des parties de code lors du débogage.

Exemple de commentaires en Python

Ceci est un commentaire sur une seule ligne

"""

Ceci est un commentaire

sur plusieurs lignes.

Il peut s'étendre sur plusieurs

lignes sans avoir à utiliser # sur chaque ligne.

"""

Exemple de désactivation temporaire de code

print("Cette ligne de code n'est pas exécutée pour le moment")

Les commentaires en python.

Les **commentaires** sont un outil précieux pour améliorer la qualité et la maintenance de votre code Python. Cependant, il est également important de les utiliser avec parcimonie et de maintenir votre code aussi propre et lisible que possible.

Les variables.

Une variable est un **identifiant** associé à une **valeur**. Elle se caractérise donc par son nom et sa valeur.

En informatique, la variable est une référence située à une adresse mémoire.

On affecte une variable par une valeur en utilisant le signe = (*qui n'a rien à voir avec l'égalité en math !*)

```
a = 2 # prononcez : a "reçoit" 2
b = 7.2 * math.log(math.e / 45.12) - 2*math.pi
c = b ** a
```

Les variables.

Sur l'exemple proposé :

a = 2 (a reçoit la valeur 2)

Python a “*deviné*” que la variable était un entier. On dit alors que Python est un langage au **typage dynamique**.

Python a alloué (***réservé***) l'espace en mémoire pour y accueillir un entier. Chaque type de variable prend plus ou moins d'espace en mémoire.

Python a aussi fait en sorte qu'on puisse retrouver la variable sous le nom x.

Enfin, **Python** a assigné la valeur 2 à la variable x.

Autres variantes d'affectation de variables.

affectation simple

$v = 4$

affectation augmentée

$v += 2$ *# idem à : $v = v + 2$ si v est déjà référencé*

affectation de droite à gauche

$c = d = 8$ *# cibles multiples*

affectations parallèles d'une séquence

$e, f = 2.7, 5.1$ *# tuple*

Nommage de variable et mots réservés.

Les noms des variables sont assez librement choisis en python.

Il est cependant important d'observer certaines règles :

- Le nom doit clairement exprimer ce que la variable est censée contenir.
- Un nom de variable est une séquence de lettres (**a-z, A-Z**) et de chiffre (0-9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées.
- Pas de lettres accentuées.
- Aucun caractère spécial à l'exception de underscore (_).
- La casse est significative (**TOTO - Toto - toto sont 3 variables différentes**).
- Vous ne pouvez pas utiliser comme nom de variables les 30 mots suivants.

Nommage de variable et mots réservés.

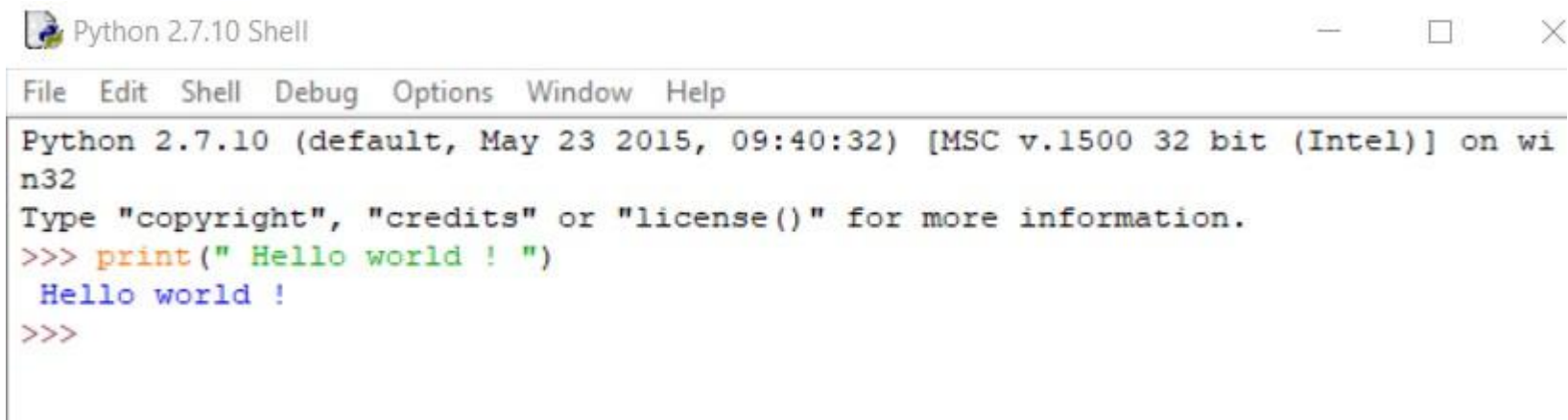
- La casse est significative (TOTO – Toto – toto sont 3 variables différentes).
- Vous ne pouvez pas utiliser comme nom de variables les 30 mots suivants.

| | | | | |
|----------|---------|--------|----------|-------|
| 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 | |

Affichage – La fonction `print()`

La fonction `print()` permet l'affichage d'une chaîne de caractère.

La fonction `print()` affiche l'argument qui lui est passé entre parenthèse et un retour à la ligne.

A screenshot of a Python 2.7.10 Shell window. The window has a title bar that says "Python 2.7.10 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area of the window shows the following text:

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on wi
n32
Type "copyright", "credits" or "license()" for more information.
>>> print(" Hello world ! ")
Hello world !
>>>
```

Affichage formaté

La méthode **format()** permet de contrôler finement toutes sortes d'affichages.

Remplacements simples :

```
print ("{} {} {}".format("zéro", "un", "deux")) # zéro un deux
```

```
print ("{} {} {}".format("zéro", "un", "deux")) # deux zéro un
```

```
print("Je m'appelle {}".format("Bob")) # Je m'appelle Bob
```

```
print("Je m'appelle {}".format("Bob")) # Je m'appelle {Bob}
```

```
print("{}".format("-"*10))
```

Affichage – Écriture formatée – f-string.

On appelle écriture formatée, un mécanisme qui permet d'afficher des variables dans un format spécifique (par exemple un certain nombre de décimal pour les float) .

La version **3.6 python**, a introduit les f-strings.

f-string : Formated String literals.

Les chaînes de caractères sont représentées par du texte entouré de guillemets.

```
print(" Hello world ")
```

```
print(f " Hello world ")
```

La présence du “f” indique la mise en place de l'écriture formatée contrairement au string normal.

Écriture formatée – Prise en main des f-string.

```
>>> age = 32
```

```
>>> nom = "Jhon"
```

```
>>> print( f " {nom} a {age} ans" )
```

Equivalent à :

```
print ( nom, "a", age , "ans" )
```

La portée des variables en Python.

Le terme “**Portée de variable**” sert à désigner les différents espaces dans le script dans lesquels une variable est accessible c’est à dire utilisable.

En python, une variable peut avoir une portée locale et globale.

Variable locale : Toute variable définie dans une fonction est appelées variables locales. Une variable locale est utilisée localement c’est à dire à l'intérieur de la fonction les a définie.

Variable globale : Les variables globales sont définies dans l’espace global du script c’est à dire en dehors de toutes fonctions. Ces variables sont accessibles à travers l’ensemble du script et accessible en lecture seulement à l'intérieur seulement des fonctions utilisées dans ce script.

Les types de variables.

Le type d'une variable correspond à la nature de celle-ci.

En python, il existe plusieurs types :

- ❑ Les entiers (integer)
- ❑ Les décimaux (float)
- ❑ Les chaîne de caractère (string),
- ❑ Les booleens (bool)
- ❑ Les nombres complexes .

Les entiers – Integer.

Type int :

Opération arithmétique

```
20 + 3    # 23
20 - 3    # 17
20 * 3    # 60
20 ** 3   # 8000
20 / 3    # 6.666666666666667
20 // 3   # 6 (division entière)
20 % 3    # 2 (modulo)
abs(3 - 20) # valeur absolue
```


Bases usuelles des entiers.

Un entier écrit en **base 10** (par exemple 179) peut se représenter en binaire, octal et hexadécimal en utilisant les syntaxes suivantes :

```
>>> 0b10110011 # binaire
179
>>> 0o263 # octal
179
>>> 0xB3 # hexadécimal
179
```

Les décimaux – float.

- ❑ Un float est noté avec un point décimal ou en notation exponentielle :

```
2.718  
.02  
3e8  
6.023e23
```

- ❑ Les flottants supportent les mêmes opérations que les entiers.
- ❑ Les float ont une précision finie indiquée dans `sys.float_info.epsilon`.

Les booleans - bool.

- ❑ Deux valeurs possibles : **False**, **True**.
- ❑ Opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<` et `<=` :

```
2 > 8 # False
2 <= 8 < 15 # True
```

Les nombres complexes.

- ❑ Les complexes sont écrits en notation cartésienne formée de deux flottants.
- ❑ La partie imaginaire est suffixée par j :
- ❑ Un module mathématique spécifique (`cmath`) leur est réservé :

```
print(1j) # 1j
print((2+3j) + (4-7j)) # (6-4j)
print((9+5j).real) # 9.0
print((9+5j).imag) # 5.0
print(abs(3+4j)) # 5.0 : module
```

Les chaînes de caractères.

- ❑ Entre simple quote (') ou double quote (").
- ❑ Affichage au moyen de la fonction print().

len() – Longueur d'une chaîne de caractère :

```
m = "Hello world"
len(m)          # -> 11
```

Les chaînes de caractères.

Extraction des sous chaînes :

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|
| H | e | l | l | o | | w | o | r | l | d |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```
m[:5]      # -> 'Hello'
m[6:8]     # -> 'wo'
m[-3:]     # -> 'rld'
```

Les chaînes de caractères.

Multiplication - Répétition.

```
"a" * 6    # -> "aaaaaa"
```

Concaténation

```
"Cette phrase" + " est en deux morceaux."
```

```
name = "Marius"  
age = 28  
"Je m'appelle " + name + " et j'ai " + str(age) + " ans"
```

Les chaînes de caractères.

Construction à partir des données avec %s.

```
"Je m'appelle %s et j'ai %s ans" % ("Marius", 28)
```


Les chaînes de caractères.

La substitution

```
"Hello world".replace("Hello", "Goodbye") # -> "Goodbye world"
```

Chaînes sur plusieurs lignes

`\n` est une syntaxe spéciale faisant référence au caractère **“nouvelle ligne”**

```
"Hello\nworld" # -> Hello <nouvelle ligne> world
```



Les Listes

Manipulation des listes, opération sur les listes, etc...

Manipulation des listes.

Une liste est une structure de données qui contient une série de valeurs.

Les valeurs contenues dans une liste peuvent être de différents type (entier et chaîne de caractères).

Les éléments d'une liste sont séparés par des virgules, le tout encadré par des crochets.

```
>>> pays = ["FRANCE", "PORTUGAL", "BELGIQUE", "NIGER"]
```

```
>>> pays
```

```
['FRANCE', 'PORTUGAL', 'BELGIQUE', 'NIGER']
```

NB : Lorsqu'on affiche une liste, Python la restitue telle que saisie.

Manipulation des listes.

On peut appeler les éléments d'une liste par leur position.

Soit la liste suivante :

```
pays : ["FRANCE", "PORTUGAL", "BELGIQUE", "NIGER"]
```

```
indices :      0          1          2          3
```

Les indices d'une liste de n éléments commencent par 0 et se terminent à n-1.

```
>>> pays = ["FRANCE", "PORTUGAL", "BELGIQUE", "NIGER"]
```

```
>>> pays[0]
```

```
FRANCE
```

```
>>> pays[3]
```

```
NIGER
```

Opération sur les listes.

Les **listes** supportent l'opérateur **+** de **concaténation** ainsi que ***** pour la **duplication**.

Considérons deux listes :

```
>>> Liste1 = ["FRANCE", " Pays-Bas " ]
```

```
>>> Liste2 = ["CONGO", "NIGER" ]
```

```
>>> Liste1 + Liste2
```

```
["FRANCE", "Pays-Bas", "CONGO", "NIGER" ] #L'opérateur + sert à concaténer les listes.
```

```
>>> Liste1 * 3
```

```
["FRANCE", " Pays-Bas ", "FRANCE", " Pays-Bas ", "FRANCE", " Pays-Bas " ]
```

Tranches

Les listes offrent la possibilité de sélectionner une partie des éléments en utilisant un indiçage sous la forme **[m,n+1]**.

[m,n+1] : On récupère les éléments de la liste à partir de l'élément d'indice m inclu jusqu'à l'élément d'indice n+1 exclu. On parle alors de **tranche** :

```
>>> pays = ["FRANCE", "PAYS-BAS", "FINLANDE", "HONGRIE"]
```

```
>>> pays[0:2]
```

```
["FRANCE", "PAYS-BAS"]
```

```
>>> pays[0:]
```

```
["FRANCE", "PAYS-BAS", "FINLANDE", "HONGRIE"]
```

```
>>> pays[1:-1]
```

```
["PAYS-BAS", "FINLANDE"]
```

La fonctions Len().

L'instruction **len()** vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
>>> pays = ["FRANCE", "PAYS-BAS", "FINLANDE", "HONGRIE"]
```

```
>>> len ( pays)
```

```
4
```

```
>>> len ([1 , 2, 3, 4, 5, 6, 7, 8])
```

```
8
```

Les fonctions **range()** et **List()**.

L'instruction `range()` est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle. Lorsqu'elle est utilisée en combinaison avec la fonction `list()`, on obtient une liste d'entiers. Par exemple :

```
>>> list ( range (10))
```

```
[0 , 1 , 2 , 3 , 4, 5, 6 , 7, 8, 9]
```

La fonction `range()` peut également prendre plusieurs arguments :

```
>>> list ( range (0 , 5))
```

```
[0 , 1 , 2 , 3 , 4]
```

```
>>> list ( range (15 , 20))
```

```
[15 , 16 , 17 , 18 , 19]
```

```
>>> list ( range (0 , 1000 , 200))
```

```
[0 , 200 , 400 , 600 , 800]
```


Quelques méthodes sur les listes.

```
>>> nombres = [17, 38, 10, 25, 72]

>>> nombres.sort()

>>> print(nombres) # [10, 17, 25, 38, 72]

>>> nombres.append(12)

>>> nombres.reverse()

>>> nombres.remove(38)

>>> print(nombres) # [12, 72, 25, 17, 10]

>>> print(nombres.index(17)) # 3
```

```
>>> nombres[0] = 11

>>> nombres[1:3] = [14, 17, 2]

>>> print(nombres.pop()) # 10

>>> print(nombres) # [11, 14, 17, 2, 17]

>>> print(nombres.count(17)) # 2

>>> nombres.extend([1, 2, 3])

>>> print(nombres) # [11, 14, 17, 2, 17, 1, 2, 3]
```

Les tuples.

On appelle **tuple**, une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes.

Syntaxe :

Éléments séparés par des virgules.

```
mon_tuples = ('a' , 2, [1,3] )
```

- Les **tuples** s'utilisent comme les liste mais leur parcours est plus rapide.
- Ils sont utiles pour définir des constantes

Les dictionnaires (dict)

Collection de couples **cle : valeur** entourée d'accolades.

Les **dictionnaires** constituent un type composite mais ils n'appartiennent pas aux séquences.

Comme les listes, les **dictionnaires** sont modifiables, mais les couples enregistrés n'occupent pas un ordre immuable, leur emplacement est géré par un algorithme spécifique.

Une clé pourra être **alphabétique, numérique**. . . en fait tout type hashable. Les valeurs pourront être des valeurs numériques, des séquences, des dictionnaires, mais aussi des fonctions, des classes ou des instances.

Exemples de création

insertion de clés/valeurs une à une

```
d1 = {} # dictionnaire vide
```

```
d1["nom"] = 3
```

```
d1["taille"] = 176
```

```
print(d1) # {'nom': 3, 'taille': 176}
```

définition en extension

```
d2 = {"nom": 3, "taille": 176}
```

```
print(d2) # {'nom': 3, 'taille': 176}
```

définition en intension

```
d3 = {x: x**2 for x in (2, 4, 6)}
```

```
print(d3) # {2: 4, 4: 16, 6: 36}
```

utilisation de paramètres nommés

```
d4 = dict(nom=3, taille=176)
```

```
print(d4) # {'taille': 176, 'nom': 3}
```

utilisation d'une liste de couples clés/valeurs

```
d5 = dict([("nom", 3), ("taille", 176)])
```

```
print(d5) # {'nom': 3, 'taille': 176}
```

Quelques méthodes applicables aux dictionnaires

```
tel = {'jack': 4098, 'sape': 4139}
```

```
tel['guido'] = 4127
```

```
print(tel) # {'sape': 4139, 'jack': 4098, 'guido': 4127}
```

```
print(tel['jack']) # 4098
```

```
del tel['sape']
```

```
tel['irv'] = 4127
```

```
print(tel) # {'jack': 4098, 'irv': 4127, 'guido': 4127}
```

Quelques méthodes applicables aux dictionnaires

```
print(list(tel.keys())) # ['jack', 'irv', 'guido']
```

```
print(sorted(tel.keys())) # ['guido', 'irv', 'jack']
```

```
print(sorted(tel.values())) # [4098, 4127, 4127]
```

```
print('guido' in tel, 'jack' not in tel) # True False
```

Gestion des fichiers – Ouverture & fermeture des fichiers.

Comme la plupart des langages, Python utilise classiquement la notion de fichier. C'est un type pré-défini en Python, qui ne nécessite donc pas d'importer de module externe.

```
f1 = open("monFichier_1", "r", encoding='utf-8') # en lecture  
f2 = open("monFichier_2", "w", encoding='utf-8') # en écriture  
f3 = open("monFichier_3", "a", encoding='utf-8') # en ajout
```

Le paramètre optionnel `encoding` assure les conversions entre les types **byte** et **str**. Les encodages les plus fréquents sont **"utf-8"** (*c'est l'encodage à privilégier en Python 3*), *'latin1'*, *'ascii'...*

Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.

Une seule méthode de fermeture :

```
f1.close()
```

Gestion des fichiers – Ecriture séquentielle.

Méthodes d'écriture :

```
f = open("truc.txt", "w")  
s = 'toto\n'  
f.write(s) # écrit la chaîne s dans f  
l = ['a', 'b', 'c']  
f.writelines(l) # écrit les chaînes de la liste l dans f  
f.close()  
  
# utilisation de l'option file de print  
f2 = open("truc2.txt", "w")  
print("abcd", file=f2)  
f2.close()
```


Gestion des fichiers – Lecture séquentielle.

Méthodes de lecture :

```
f = open("truc.txt", "r")
s = f.read() # lit tout le fichier --> string
s = f.read(3) # lit au plus n octets --> string
s = f.readline() # lit la ligne suivante --> string
s = f.readlines() # lit tout le fichier --> liste de strings
f.close()

# Affichage des lignes d'un fichier une à une
f = open("truc.txt") # mode "r" par défaut
for ligne in f:
    print(ligne[:-1]) # pour sauter le retour à la ligne
f.close()
```



Les structures de contrôle

syntaxes et exemples.

Les structures de contrôle – blocs & indentation.

Python utilise l'indentation pour définir les blocs. Ce qui améliore la lisibilité du code.

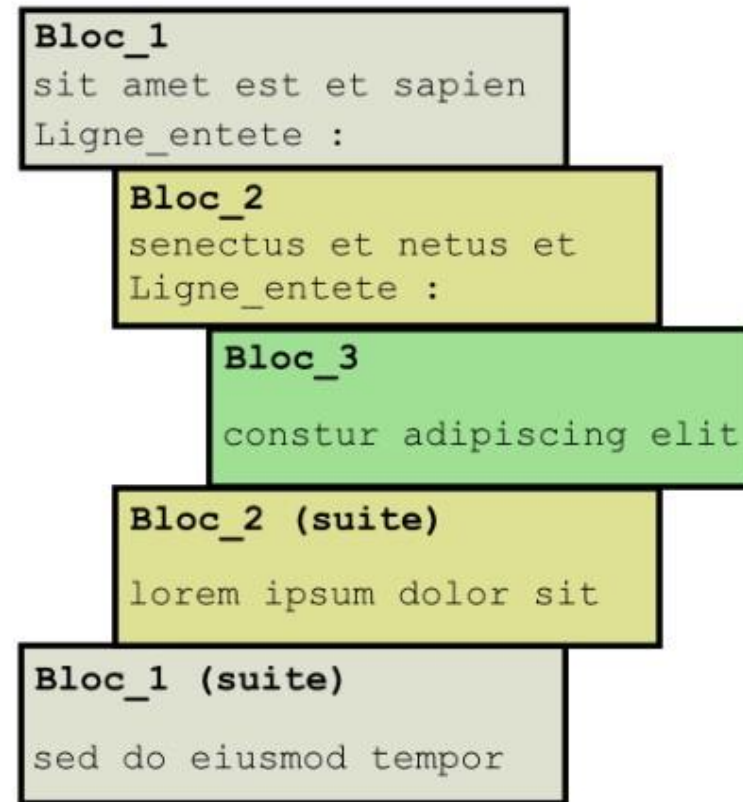


Figure 3.1 – Les instructions composées.

Opérateurs de comparaison & opérateurs logiques

Les opérateurs de comparaison.

== : **Égal à** (*vérifie si les deux valeurs sont égales*).

!= : **Différent de** (*vérifie si les deux valeurs ne sont pas égales*).

< : **Inférieur à** (*vérifie si la valeur de gauche est strictement inférieure à celle de droite*).

> : **Supérieur à** (*vérifie si la valeur de gauche est strictement supérieure à celle de droite*).

<= : **Inférieur ou égal à** (*vérifie si la valeur de gauche est inférieure ou égale à celle de droite*).

>= : **Supérieur ou égal à** (*vérifie si la valeur de gauche est supérieure ou égale à celle de droite*).

Opérateurs de comparaison & opérateurs logiques

x = 5

y = 10

resultat = x == y **# False**

resultat = x != y **# True**

resultat = x < y **# True**

resultat = x > y **# False**

resultat = x <= y **# True**

resultat = x >= y **# False**

Opérateurs de comparaison & opérateurs logiques

Opérateurs Logiques :

Les opérateurs logiques sont utilisés pour effectuer des opérations logiques sur les valeurs booléennes (**True ou False**).

and : Opérateur logique **ET** (renvoie True si les deux opérandes sont True, sinon False).

or : Opérateur logique **OU** (renvoie True si au moins l'un des opérandes est True, sinon False).

not : Opérateur logique **NON** (renverse la valeur booléenne, True devient False et vice versa).

Opérateurs de comparaison & opérateurs logiques

```
a = True
```

```
b = False
```

```
resultat = a and b # False
```

```
resultat = a or b  # True
```

```
resultat = not a   # False
```

Les structures conditionnelles.

if - [elif] - [else]

Contrôler une alternative

```
if x < 0:  
    print("x est négatif")
```

```
elif x % 2:  
    print("x est positif et impair")  
else:  
    print("x n'est pas négatif et est pair")
```

Tester une valeur booléenne :

```
if x: # mieux que (if x is True:) ou que (if x == True:)  
  
    pass
```


Exemple – Boucle compacte.

```
x, y = 4, 3
```

```
# Ecriture classique :
```

```
if x < y:
```

```
    plus_petit = x
```

```
else:
```

```
    plus_petit = y
```

```
# Utilisation de l'opérateur ternaire :
```

```
plus_petit = x if x < y else y
```

```
print("Plus petit : ", plus_petit) # 3
```

La boucle **while**

```
x, cpt = 257, 0

print("L'approximation de log2 de", x, "est", end=" ")

while x > 1:

    x //= 2 #Division avec troncature.

    cpt += 1 # incrémentation

print(cpt, "\n") # 8
```

Instructions break et continue

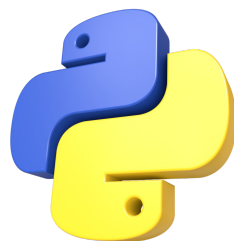
Ces deux instructions permettent de modifier le comportement d'une boucle (for ou while) avec un test.

L'instruction break stoppe la boucle.

```
>>> for i in range (5):  
    if i > 2:  
        break  
    print (i) # 0 1 2
```

L'instruction continue saute à l'itération suivante, sans exécuter la suite du bloc d'instructions de la boucle.

```
>>> for i in range (5):  
    if i == 2:  
        continue  
    print (i) # 0 1 2 3 4
```



Écriture & documentation des fonctions

Définition – syntaxe – passage d'arguments –

Définition et Syntaxe.

Une **fonction** est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande.

Une bonne pratique consiste à définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; c'est la mise en facteur commun.

Le bloc d'instructions est obligatoire. S'il est vide, on emploie l'instruction *pass*. La documentation (facultative) est fortement conseillée.

```
def nomFonction(paramètres):
```

```
    """Documentation de la fonction."""
```

```
    <bloc_instructions>
```

Passage d'arguments.

définition

```
def maFonction(x, y, z):  
    pass
```

x = 7
y = 'k'
z = 2.718

appel

```
maFonction(7, 'k', 2.718):
```

Passage des arguments.

Un ou plusieurs paramètres sans retour

```
def table(base, debut, fin):  
    """Affiche la table des <base> de <debut> à <fin>."""  
    n = debut  
    while n <= fin:  
        print(n, 'x', base, '=', n * base, end=" ")  
        n += 1
```

exemple d'appel :

```
table(7, 2, 11)
```

2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42

7 x 7 = 49 8 x 7 = 56 9 x 7 = 63 10 x 7 = 70 11 x 7 = 77

Passage des arguments

Un ou plusieurs paramètres avec retour

```
from math import pi
def cube(x):
    return x**3
def volumeSphere(r):
    return 4.0 * pi * cube(r) / 3.0

# Saisie du rayon et affichage du volume
rayon = float(input('Rayon : '))
print("Volume de la sphère =", volumeSphere(rayon))
```


Passage d'arguments

Utilisation d'un retour multiple

```
import math
def surfaceVolumeSphere(r):
    surf = 4.0 * math.pi * r**2
    vol = surf * r/3
    return surf, vol
# programme principal
rayon = float(input('Rayon : '))
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface {:g} et de volume {:g}".format(s, v))
```

Les lambdas expressions

En Python, une **lambda expression** (ou fonction lambda) est une fonction anonyme et sans nom qui peut être utilisée pour créer des fonctions simples et courtes. Les lambda expressions sont principalement utilisées lorsqu'une fonction simple doit être définie de manière concise et qu'il n'est pas nécessaire de lui attribuer un nom.

La syntaxe générale d'une lambda expression en Python est la suivante :

lambda arguments: expression

Les éléments clés d'une lambda expression sont les suivants :

- **"lambda"** est le mot-clé utilisé pour définir une lambda expression.
- **"arguments"** est la liste des arguments que la fonction lambda prend en entrée, séparés par des virgules. Cette liste peut être vide ou contenir un ou plusieurs arguments.
- **"expression"** est l'expression qui est évaluée et renvoyée comme résultat de la fonction lambda

Illustration de l'utilisation des lambda expressions.

Fonction lambda sans argument :

```
zero_argument_lambda = lambda: "Hello, World!"
```

```
print(zero_argument_lambda()) # Output: Hello, World!
```

Fonction lambda avec un argument :

```
square = lambda x: x ** 2
```

```
print(square(5)) # Output: 25
```

Fonction lambda avec plusieurs arguments :

```
add = lambda x, y: x + y
```

```
print(add(3, 4)) # Output: 7
```

Les générateurs

Les **générateurs** fournissent un moyen de générer des exécutions paresseuses, ce qui signifie qu'elles ne calculent que les valeurs réellement demandées. Ceci peut s'avérer beaucoup plus efficace (*en termes de mémoire*) que le calcul, par exemple, d'une énorme liste en une seule fois.

Les générateurs sont souvent utilisés avec la boucle for pour parcourir les éléments de la séquence.

Il existe deux façons courantes de créer des générateurs en **Python** : en utilisant des **fonctions génératrices** et en utilisant des **expressions génératrices**.

- **Fonctions génératrices** : Vous pouvez créer un générateur en utilisant une fonction qui contient au moins une instruction **yield**.

Lorsque l'instruction **yield** est rencontrée, la fonction retourne la valeur spécifiée et mémorise son état pour reprendre l'exécution ultérieurement.

Les générateurs

Exemple de fonction génératrice :

```
def quarters(next_quarter=0.0):  
    while True:  
        yield next_quarter  
        next_quarter += 0.25  
if __name__ == "__main__":  
    result = []  
    for x in quarters():  
        result.append(x)  
        if x == 1.0:  
            break  
    print("Liste résultante : ", result)  
# Liste résultante : [0.0, 0.25, 0.5, 0.75, 1.0]
```

Les générateurs

Expressions génératrices :

Les expressions génératrices sont similaires aux list comprehensions, mais elles créent des générateurs au lieu de listes. Elles utilisent la syntaxe (expression for item in iterable).

```
generator = (x * 2 for x in range(5))
```

Utilisation du générateur dans une boucle for

```
for num in generator:
```

```
    print(num)
```

Autres exemples de fonctions génératrices.

```
def uppercase_strings(strings):  
    for s in strings:  
        yield s.upper()
```

```
strings = ["hello", "world", "python"]  
uppercase = uppercase_strings(strings)
```

```
print(list(uppercase)) # Output: ['HELLO', 'WORLD', 'PYTHON']
```

Les générateurs

Les **générateurs** sont particulièrement utiles lorsque vous travaillez avec de grandes quantités de données, car ils permettent d'économiser de la mémoire en générant les éléments un par un, au lieu de tous en une seule fois.

En résumé, les générateurs en Python sont des objets qui permettent de créer des séquences d'éléments de manière paresseuse, ce qui les rend adaptés à la manipulation de grandes quantités de données de manière efficace.



Structuration du code en modules.

Création – utilisation des modules, etc...

Structuration de code en modules

Module : fichier indépendant permettant de scinder un programme en plusieurs scripts. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;
- réalisation de services ou de données partagés ;
- partition de l'espace de noms du système.

Comment créer des modules Python ?

On crée un fichier `module_math.py`.

```
# module_math.py
"""
Votre premier module module_math.py,
contenant des fonctions mathématiques
"""

def add(x,y):
    """La somme de deux nombres"""
    return x + y

def sous(x, y):
    """La soustraction de deux nombres"""
    return x - y
```

On crée ensuite un fichier `test.py`. On utilise le mot clé `import` pour récupérer du code.

Utiliser import module

```
# test.py
```

```
import module_math
```

```
x = 3
```

```
y = 2
```

```
addition = module_math.add(x,y)
```

```
print(addition)
```

Nous avons utilisé `module_math.add()` pour appeler la fonction `add()`, le point indique que la fonction `add()` appartient au module `module_math`.

Utiliser from module import *

Nous pouvons importer toutes les fonctions qui existent dans un module.

```
# test.py
```

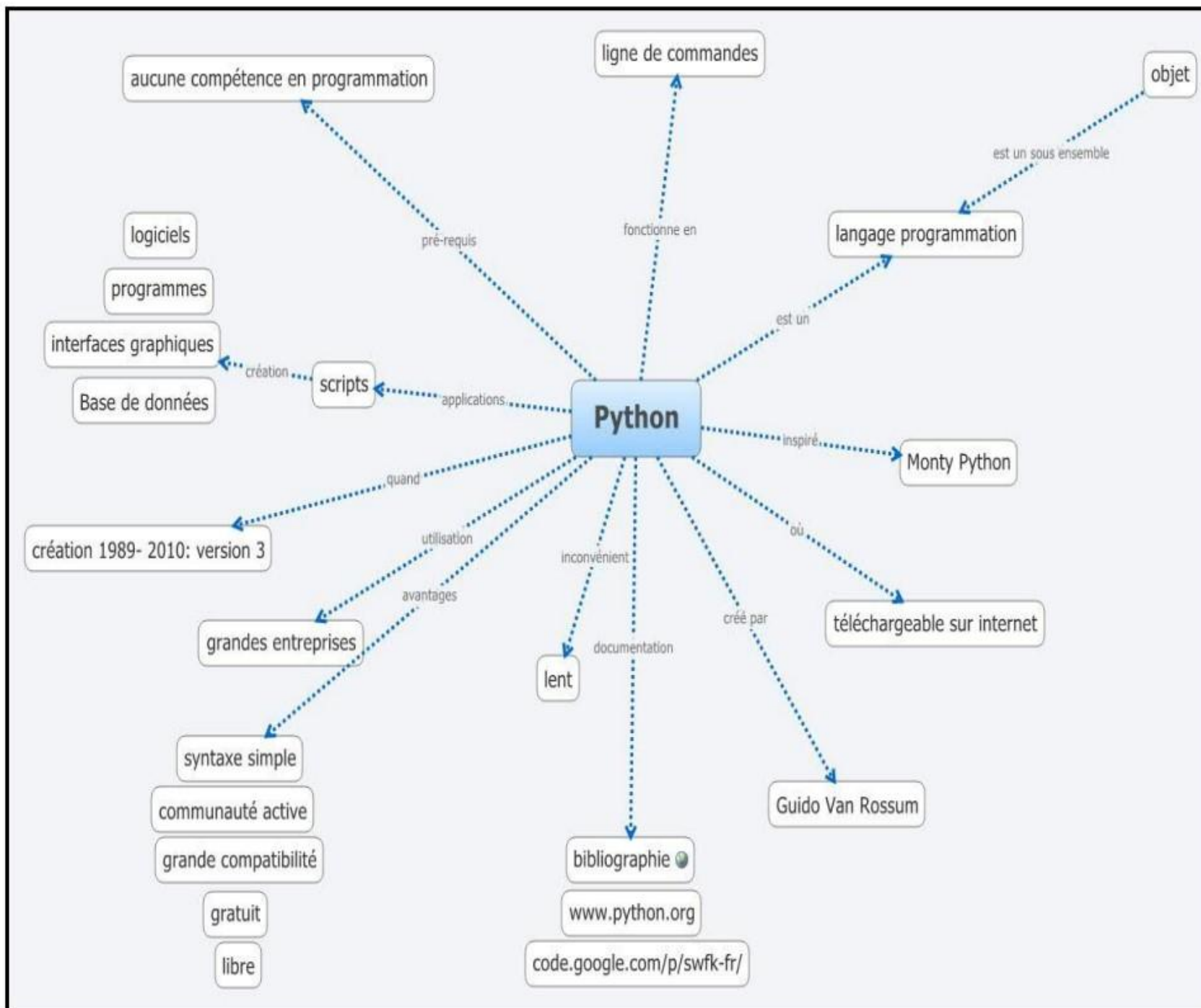
```
from module_math import *
```

```
x = 3
```

```
y = 2
```

```
addition = add(x,y)
```

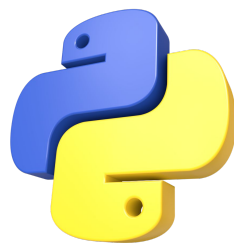
```
print(addition)
```





Travaux Pratiques.

Installation et prise en main de l'interpréteur Python.



Approche orientée objet.

Héritage – Encapsulation – Polymorphisme...

Un mot sur l'approche classique ou programmation procédurale.

Aux tout débuts de l'informatique, le fonctionnement interne des processeurs décidait de la seule manière efficace de programmer un ordinateur.

Alors que l'on acceptait tout programme comme une suite logique d'instructions, il était admis que l'organisation du programme et la nature même de ces instructions ne pouvaient s'éloigner de la façon dont le processeur les exécutait : pour l'essentiel, des **modifications de données mémorisées, des glissements de ces données d'un emplacement mémoire à un autre, des opérations d'arithmétique et de logique élémentaire.**

Un mot sur l'approche classique ou programmation procédurale.

Elle consiste à découper un programme en une série de fonctions lesquelles ont pour but de réaliser un traitement particulier : **programmation procédurale**.

```
#include<iostream.h>
```

Données

const

int taille_maximale = 100;

float vecteur[taille_maximale];

int haut = 0;

Fonctions

bool Pile_vide() { **return** (haut==0); }

bool Pile_pleine() { **return** (haut == taille_maximale); }

void Initialiser_pile() { haut = 0; }

void Insérer_pile(float valeur) { vecteur[haut] = valeur; haut +=1; }

float Accès_pile() { **return** vecteur[haut - 1] ; }

float Enlever_pile() { haut = haut - 1 ; return vecteur[haut] ; }

Utilisation

void main ();

```
{
    Initialiser_pile();
    Insérer_pile(2.3);
    Insérer_pile(3.4);
    Insérer_pile(6.3);
    while (!Pile_vide()) { cout << Enlever_pile() << endl; }
}
```

Avantages & Inconvénients

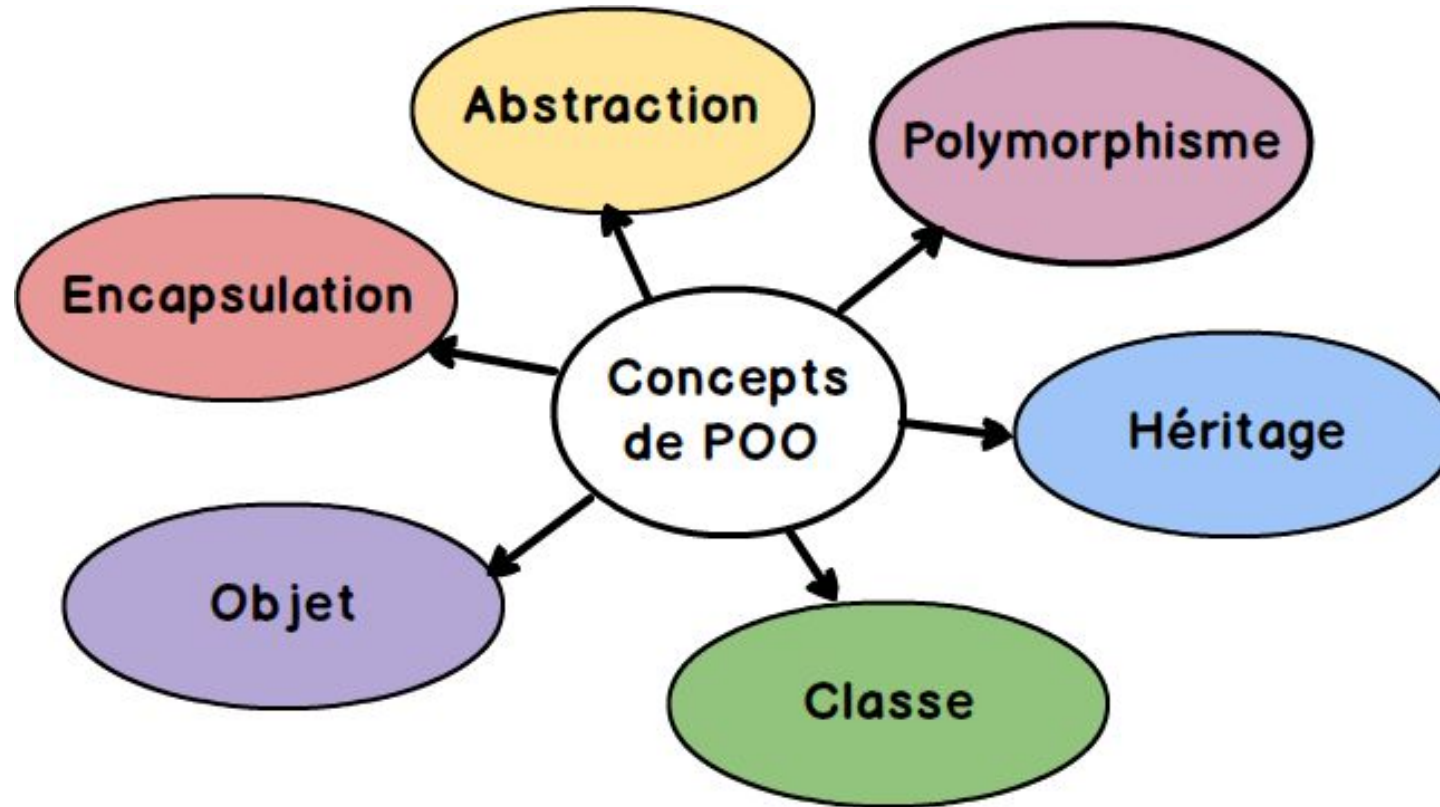
- Facilite la maintenance d'un petit programme.
- Favorise la modularité des programmes, un problème complexe pouvant être découpé en plusieurs sous-problèmes.
- ces fonctions peuvent constituer les éléments d'une boîte à outils qui pourront être réutilisés pour les nouveaux développements.
- Introduction de mécanismes de passage de paramètres.

- Dissocie les données des fonctions qui les manipulent.
- Données globales
- Couplage élevé des données
- Aucune technique permettant de spécialiser des fonctions existantes.

Pourquoi l'orienté objet ?

- ❑ Améliorer la réutilisation des outils existants (code ...)
- ❑ Favoriser la spécialisation de ces mêmes outils.
- ❑ S'éloigner de la machine et se rapprocher du problème à résoudre.
- ❑ Développer une approche mieux adaptée à la résolution de problèmes.
- ❑ Nouvelle façon de penser la décomposition de problèmes et l'élaboration de solutions.

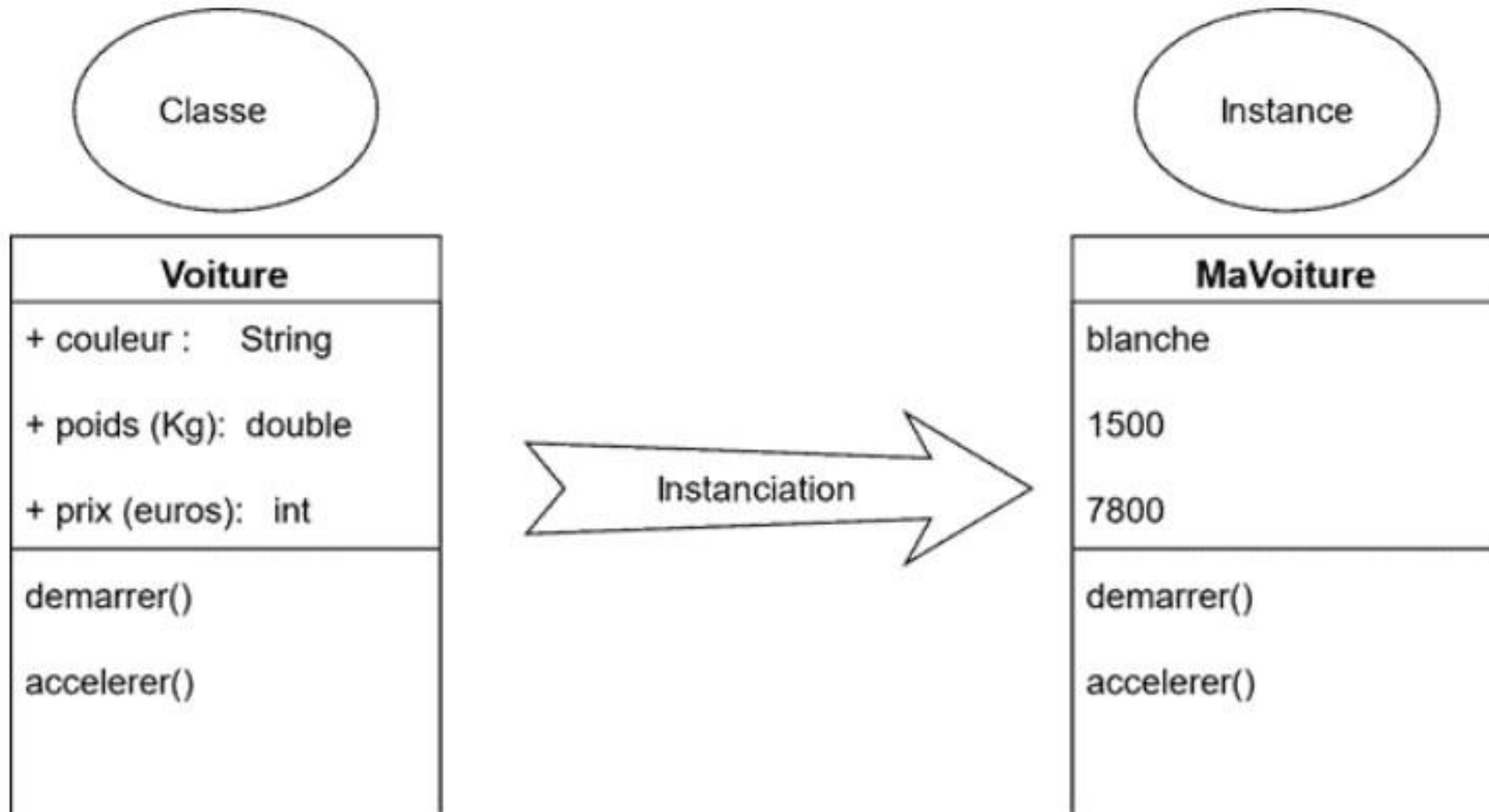
Les concepts de base de l'orientée objet.



Classe, Attribut et Méthode.

- Une **classe** est la définition d'un concept métier. Elle contient des **attributs** (*variables*) et des **méthodes** (*fonctions*).
- Une **classe** définit des **objets** qui sont des **instances** (*des représentants*) de cette classe.
- Une **classe** est une sorte de moule ou de matrice à partir duquel sont engendrés les **objets** réels qui s'appellent des **instances** de la classe considérée.
- Une **classe** est constituée d'attributs (variables associées aux objets de la classe) et des méthodes (qui sont des fonctions associées aux objets et qui peuvent agir sur ces derniers ou encore les utiliser).

Classe, attribut et méthode.



Objet (Instance d'une classe)

Une **classe** est une **abstraction de l'esprit**, elle permet de présenter, d'exposer les données du concept qu'elle manipule.

Toutefois, pour manipuler réellement les données il faut une représentation concrète de cette définition. C'est **l'instance**, ou **l'objet** de la classe considérée.

Une **instance** est un *exemplaire* d'une classe. **L'instanciation**, est le mécanisme qui permet de créer un objet à partir d'une classe.

Encapsulation.

L'encapsulation est l'un des principes fondamentaux de la POO.

Dissimulation d'informations au monde extérieur.

L'encapsulation consiste à restreindre l'accès à certains éléments d'une classe (le plus souvent ses attributs). L'objectif de l'encapsulation est de ne laisser accessible que le strict nécessaire pour que la classe soit utilisable.

Communication entre objets.

La communication entre les objets se fait par envoi de messages :

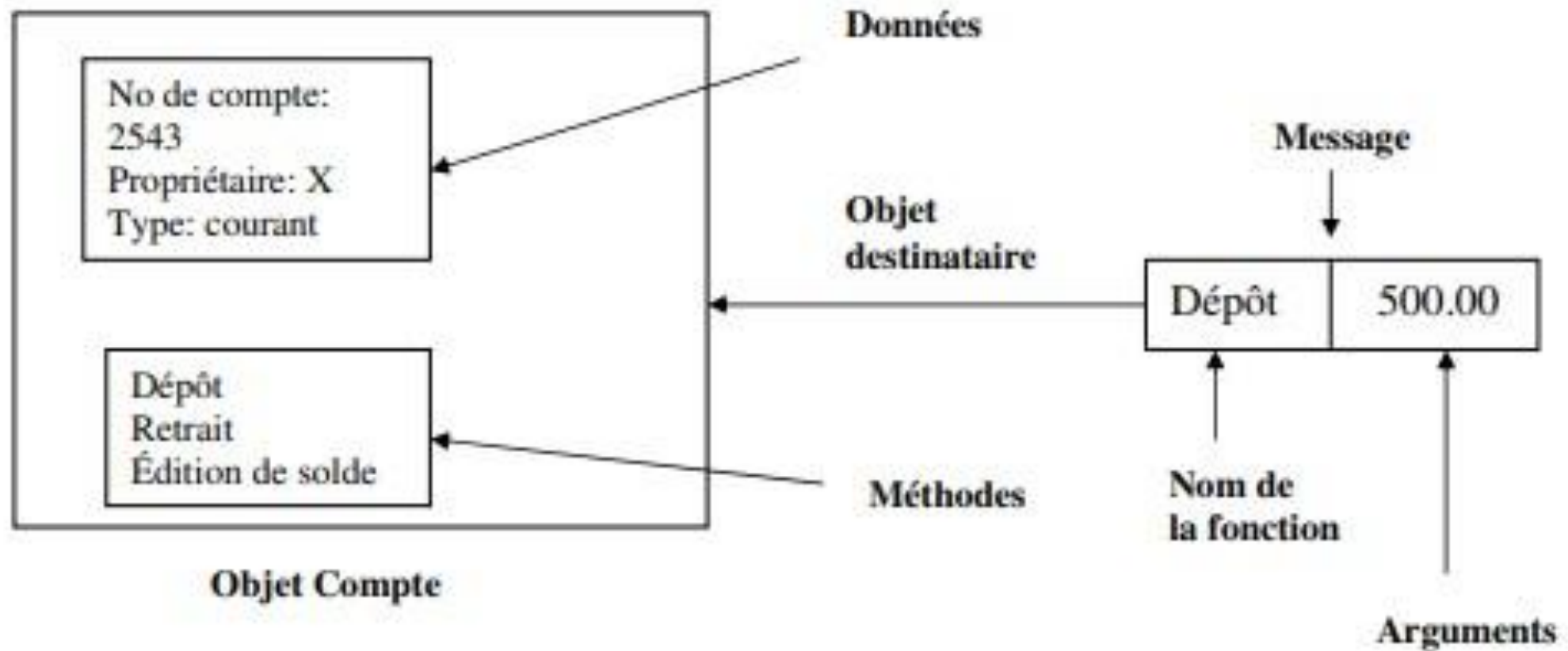
MESSAGE:

- transporte l'information nécessaire aux demandes à satisfaire,
- moyen **UNIQUE** de communication avec les objets (impossible d'accéder directement aux données encapsulées d'un objet).

Le message contient:

- nom de l'objet destinataire,
- énoncé de la demande (exemple: le nom d'une fonction),
- les arguments nécessaires (pour réaliser la demande)

Communication entre objets.



Polymorphisme.

De prime abord, le **polymorphisme** renvoie à la capacité de prendre plusieurs apparences.

En programmation, le **polymorphisme** se définit comme étant la capacité d'une méthode à se comporter différemment en fonction de l'objet qui lui est passé.

Une fonction peut donc avoir plusieurs définitions.

Héritage : Transmission des caractères d'une classe.

En génétique/biologie, **l'héritage** est la capacité qu'ont nos parents à nous transmettre certains traits physiques ou caractères.

En programmation, **l'héritage** est la capacité d'une classe d'hériter des caractéristiques d'une classe pré-existante. On parle alors de classe mère et de classe fille.

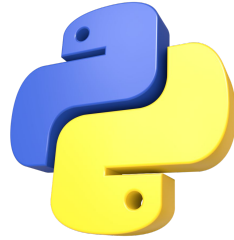
La classe fille hérite des propriétés de la classe mère mais peut les modifier et/ ou ajouter ses propres méthodes.

Les interfaces

Dans l'approche orientée objet, une **interface** est un concept qui définit un contrat ou un ensemble de méthodes (fonctions) que les classes doivent implémenter.

Une interface ne fournit pas d'implémentation réelle de ces méthodes ; elle énonce simplement ce que les classes qui l'implémentent doivent faire.

Les interfaces sont utilisées pour définir un comportement commun que différentes classes peuvent partager.



Programmation orientée objet en Python

Classe et instanciation d'objets

L'instruction class

Instruction composée : en-tête (avec docstring) + corps indenté :

```
class C:  
    """Documentation de la classe."""  
    x = 23
```

L'instanciation et ses attributs

Les classes sont des fabriques d'objets : on construit d'abord l'usine avant de produire des objets !

On instancie un objet (i.e. création, production depuis l'usine) en appelant le nom de sa classe :

```
a = C() # a est un objet de la classe C
print(dir(a)) # affiche les attributs de l'objet a
print(a.x) # affiche 23. x est un attribut de classe
a.x = 12 # modifie l'attribut d'instance (attention...)
print(C.x) # 23, l'attribut de classe est inchangé
a.y = 44 # nouvel attribut d'instance
b = C() # b est un autre objet de la classe C
print(b.x)
```

L'Initialisateur (constructeur)

Lors de l'instanciation d'un objet, la méthode `__init__` est automatiquement invoquée. Elle permet d'effectuer toutes les initialisations nécessaires :

```
class C:
    def __init__(self, n):
        self.x = n # initialisation de l'attribut d'instance x
une_instance = C(42) # paramètre obligatoire, affecté à n
print(une_instance.x) # 42
```

C'est une procédure automatiquement invoquée lors de l'instanciation : elle ne contient jamais l'instruction `return`.

Destructeur.

En Python, le **destructeur** est une méthode spéciale appelée **`__del__`**. Il est invoqué automatiquement lorsqu'un objet n'a plus de références et va être supprimé par le gestionnaire de mémoire.

Le **destructeur** est généralement utilisé pour effectuer des opérations de nettoyage, comme la libération de ressources externes ou la fermeture de fichiers.

Cependant, il n'est pas toujours nécessaire de définir un destructeur. **En général, Python gère automatiquement la libération de la mémoire et la gestion des ressources, de sorte que les destructeurs personnalisés ne sont pas couramment utilisés.**

Exemple de destructeur

```
class MaClasse:  
    def __init__(self, param1):  
        self.param1 = param1  
  
    def __del__(self):  
        print(f"Objet de MaClasse avec param1 = {self.param1} est détruit.")
```

Remarque

En résumé, les **constructeurs** (**`__init__`**) sont utilisés pour initialiser les objets lors de leur création, tandis que les **destructeurs** (**`__del__`**) sont utilisés pour effectuer des opérations de nettoyage lorsque les objets sont détruits.

Cependant, il est rarement nécessaire de définir des destructeurs explicites en Python, sauf dans des cas particuliers où vous avez besoin de gérer des ressources externes spécifiques.

La protection d'accès des attributs et des méthodes en python.

En **Python**, la protection d'accès des attributs et des méthodes est gérée principalement par la convention de nommage et par quelques mécanismes spécifiques.

Contrairement à certains langages, comme **Java** ou **C++**, **Python** ne fournit pas de mots-clés explicites tels que **private**, **protected** ou **public** pour définir la visibilité des membres d'une classe. Au lieu de cela, Python utilise une approche basée sur la convention et repose sur la philosophie du "consenting adults" (c'est-à-dire que les programmeurs sont responsables de leur propre code).

Attributs et méthodes publics :

Par convention, les attributs et les méthodes qui ne commencent pas par un **underscore** (**_**) sont considérés comme publics et peuvent être accessibles depuis l'extérieur de la classe.

La protection d'accès des attributs et des méthodes en python.

```
class MaClasse:  
    def methode_public(self):  
        self.attribut_public = 42
```

Encapsulation et responsabilité du programmeur :

L'encapsulation repose sur un principe conventionnel :

- Si un membre est littéralement privé (ni public, ni protégé), il est possible de préfixer son nom par deux underscores << __ >> .

La nécessité du paramètre self

En **Python**, le paramètre **self** est un concept fondamental et nécessaire pour le bon fonctionnement des méthodes de classe. Le paramètre **self** représente l'instance de la classe elle-même, et il est utilisé pour accéder aux attributs et aux méthodes de cette instance.

Sans le paramètre **self**, les méthodes de classe n'auraient pas de moyen de distinguer entre les différentes instances de la classe, et elles ne pourraient pas manipuler les données spécifiques à chaque instance.

En python, le paramètre 'self' est nécessaire :

- **Accès aux attributs et méthodes d'instance**
- **Gestion des méthodes de classe**
- **Création d'instances**
- **Auto-référence**

Accès aux attributs et aux méthodes.

Le paramètre **self** permet d'accéder aux attributs et aux méthodes propres à l'instance de la classe. Chaque instance de la classe peut avoir ses propres valeurs d'attributs, et **self** permet de les référencer de manière unique.

```
class MaClasse:  
    def __init__(self, valeur):  
        self.attribut = valeur  
    def afficher_attribut(self):  
        print(self.attribut)
```

```
objet1 = MaClasse(42)
```

```
objet2 = MaClasse(99)
```

```
objet1.afficher_attribut() # Affiche : 42
```

```
objet2.afficher_attribut() # Affiche : 99
```

Auto-référence

```
class MaClasse:  
    def __init__(self, valeur):  
        self.attribut = valeur  
    def doubler_attribut(self):  
        self.attribut *= 2  
        self.afficher_attribut()  
    def afficher_attribut(self):  
        print(self.attribut)  
  
objet = MaClasse(5)  
objet.doubler_attribut() # Affiche : 10
```

Les méthodes

Une méthode s'écrit comme une fonction du corps de la classe avec un premier paramètre **self** obligatoire, où **self** représente l'objet sur lequel la méthode sera appliquée. Autrement dit **self** est la référence d'instance.

```
class C:
    x = 23 # x et y : attributs de classe
    y = x + 5
    def affiche(self): # méthode affiche()
        self.z = 42 # attribut d'instance
        print(C.y) # dans une méthode, on qualifie un attribut de classe,
        print(self.z) # mais pas un attribut d'instance
ob = C() # instantiation de l'objet ob
ob.affiche() # 28 42 (à l'appel, on affecte self)
```

Héritage simple, Héritage multiple & Polymorphisme

L'héritage est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités différentes ou supplémentaires.

Le **polymorphisme** est la faculté pour une méthode portant le même nom mais appartenant à des classes distinctes héritées d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

Héritage simple

```
class Rectangle:
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l, self.nom = longueur, largeur, "rectangle"

class Carre(Rectangle):
    def __init__(self, cote=10):
        Rectangle.__init__(self, cote, cote)
        self.nom = "carré"

r = Rectangle()
print(r.nom) # 'rectangle'

c = Carre()
print(c.nom) # 'carré'
```

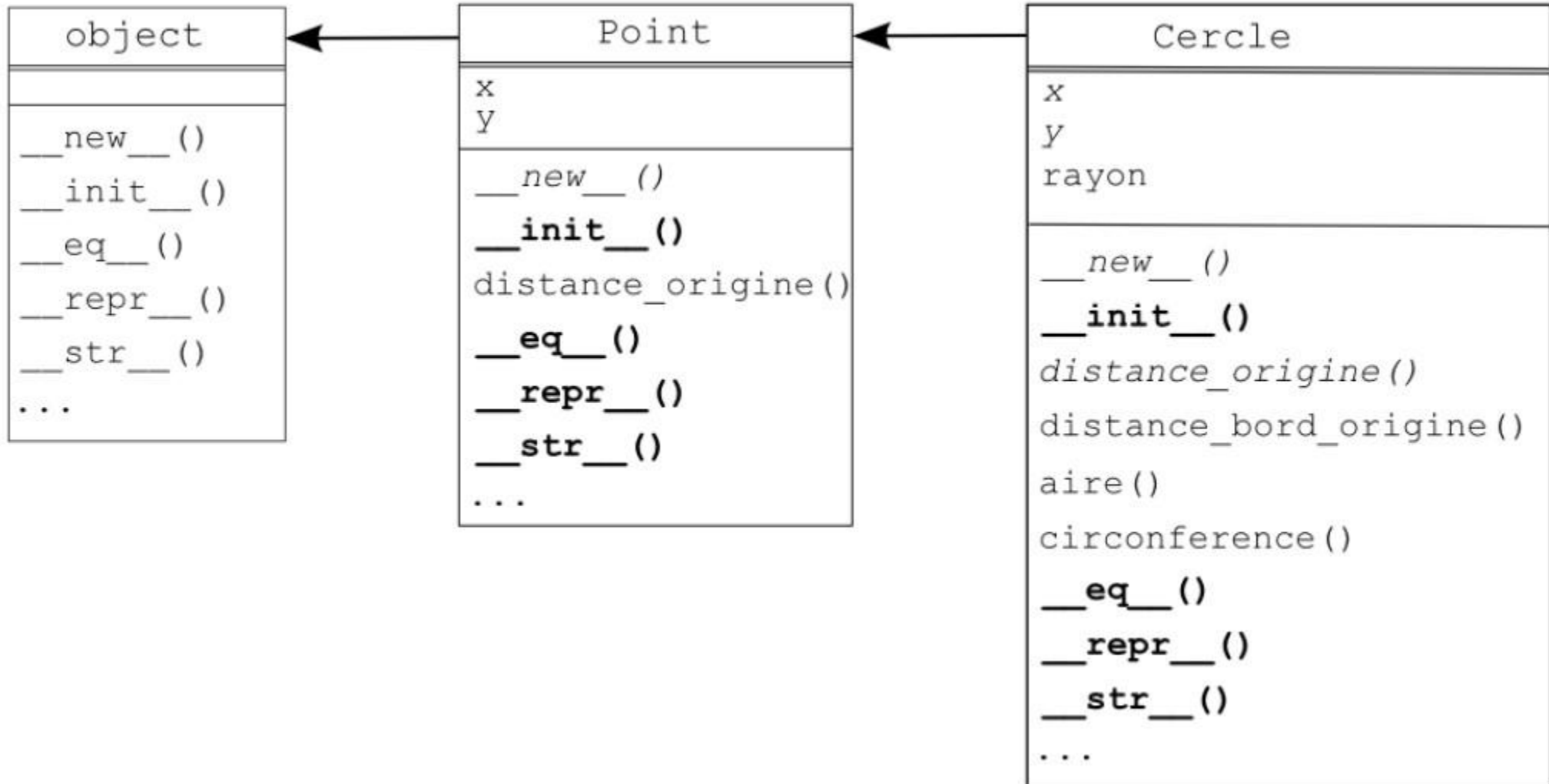
Mise en application

Nous allons tout d'abord concevoir une classe Point héritant de la classe mère object.

Puis nous pourrons l'utiliser comme classe de base de la classe Cercle.

Dans les schémas UML 1 ci-dessous, les attributs en italiques sont hérités, ceux en casse normale sont nouveaux et ceux en gras sont redéfinis (surchargés).

Diagramme de classe



La classe Point

```
class Point:

    def __init__(self, x=0, y=0):

        self.x, self.y = x, y

    @property

    def distance_origine(self):

        return math.hypot(self.x, self.y)
```

La classe Point

```
def __eq__(self, other):  
    return self.x == other.x and self.y == other.y  
  
def __str__(self):  
    return "({0.x!s}, {0.y!s})".format(self)
```

Classe Point

L'utilisation du décorateur **property** permet un accès en lecture seule au résultat de la méthode **distance_origine()** considérée alors comme un simple attribut (car il n'y a pas de parenthèse) :

```
if __name__ == "__main__":  
    p1, p2 = Point(), Point(3, 4)  
    print(p1 == p2) # False  
    print(p2, p2.distance_origine) # (3, 4) 5.0
```

La classe Cercle héritant de la super classe Point

```
class Cercle(Point):  
    def __init__(self, rayon, x=0, y=0):  
        super().__init__(x, y)  
        self.rayon = rayon  
  
    @property  
    def aire(self):  
        return math.pi * (self.rayon ** 2)  
  
    @property  
    def circonference(self):  
        return 2 * math.pi * self.rayon  
  
    @property  
    def distance_bord_origine(self):  
        return abs(self.distance_origine - self.rayon)
```

Attribut en lecture seule.

Voici la syntaxe permettant d'utiliser la méthode rayon comme un attribut en lecture-écriture. Remarquez que la méthode **rayon()** retourne l'attribut protégé : `__rayon` qui sera modifié par le setter (la méthode modificatrice) :

```
@property
def rayon(self):
    return self.__rayon

@rayon.setter
def rayon(self, rayon):
    assert rayon > 0, "rayon strictement positif"
    self.__rayon = rayon
```

Classe Cercle (suite)

```
def __eq__(self, other):  
    return (self.rayon == other.rayon  
            and super().__eq__(other))  
  
def __str__(self):  
    return ("{{0.__class__.__name__}}({0.rayon!s}, {0.x!s}, "  
            "{0.y!s})".format(self))  
  
if __name__ == "__main__":  
    c1 = Cercle(2, 3, 4)  
    print(c1, c1.aire, c1.circonference)  
    print(c1.distance_bord_origine, c1.rayon) # 3.0 2  
    c1.rayon = 1 # modification du rayon  
    print(c1.distance_bord_origine, c1.rayon) # 4.0 1
```

Héritage multiple

```
class ClasseParent1:  
    def methode_parent1(self):  
        print("Méthode de la ClasseParent1")
```

```
class ClasseParent2:  
    def methode_parent2(self):  
        print("Méthode de la ClasseParent2")
```

```
class ClasseEnfant(ClasseParent1, ClasseParent2):  
    def methode_enfant(self):  
        print("Méthode de la ClasseEnfant")
```

Héritage multiple

Instanciation de la classe enfant

```
enfant = ClasseEnfant()
```

Appel de méthodes héritées des classes parentes

```
enfant.methode_parent1() # Affiche : Méthode de la ClasseParent1
```

```
enfant.methode_parent2() # Affiche : Méthode de la ClasseParent2
```

Appel de méthode propre à la classe enfant

```
enfant.methode_enfant() # Affiche : Méthode de la ClasseEnfant
```


Résumé : Définition et utilisation d'une classe

```
#####  
# Programme Python type #  
# auteur : G.Swinnen, Liège, 2003 #  
# licence : GPL #  
#####
```

```
class Point:  
    """point mathématique"""  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
class Rectangle:  
    """rectangle"""  
    def __init__(self, ang, lar, hau):  
        self.ang = ang  
        self.lar = lar  
        self.hau = hau
```

```
    def trouveCentre(self):  
        xc = self.ang.x + self.lar /2  
        yc = self.ang.y + self.hau /2  
        return Point(xc, yc)
```

```
class Carre(Rectangle):  
    """carré = rectangle particulier"""  
    def __init__(self, coin, cote):  
        Rectangle.__init__(self,  
            coin, cote, cote)  
        self.cote = cote
```

```
    def surface(self):  
        return self.cote**2
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une **instance** de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui posséderont seulement un attribut 'x' et un attribut 'y' ; ils ne seront dotés d'aucune méthode.

Le paramètre **self** désigne toutes les instances qui seront produites par cette classe

Les instances de la classe Rectangle() posséderont 3 attributs : le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

```
#####  
## Programme principal : ##
```

```
# coord. de 2 coins sup. gauches :  
csgR = Point(40,30)  
csgC = Point(10,25)
```

*Pour créer (ou instancier) un objet, il suffit d'affecter une classe à une variable.
Les instructions ci-contre créent donc deux objets de la classe Point()...*

```
# "boîtes" rectangulaire et carrée :  
boiteR = Rectangle(csgR, 100, 50)  
boiteC = Carre(csgC, 40)
```

*... et celles-ci, encore deux autres objets.
Note : par convention, le nom d'une classe commence par une lettre majuscule*

```
# Coordonnées du centre pour chacune :  
cR = boiteR.trouveCentre()  
cC = boiteC.trouveCentre()
```

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de classe Rectangle().

```
print "centre du rect. :", cR.x, cR.y  
print "centre du carré :", cC.x, cC.y
```

```
print "surf. du carré :",  
print boiteC.surface()
```

Par contre, la méthode surface() ne peut être invoquée que pour les objets carrés.

Mise en oeuvre du polymorphisme

Utilisation de méthodes communes :

Pour que le polymorphisme fonctionne, les classes doivent avoir des méthodes avec le même nom (et la même signature) mais une implémentation différente. Par exemple, supposons que vous ayez deux classes Chien et Chat avec une méthode **parler()** :

```
class Chien:
```

```
    def parler(self):
```

```
        print("Le chien aboie")
```

```
class Chat:
```

```
    def parler(self):
```

```
        print("Le chat miaule")
```

Mise en oeuvre du polymorphisme

```
animal = Chien()
```

```
animal.parler() # Output: Le chien aboie
```

```
animal = Chat()
```

```
animal.parler() # Output: Le chat miaule
```

Mise en oeuvre du polymorphisme avec héritage.

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def parler(self):
        pass
class Chien(Animal):
    def parler(self):
        print("Le chien aboie")
class Chat(Animal):
    def parler(self):
        print("Le chat miaule")
```

Les méthodes spéciales

Ces méthodes portent des noms *prédéfinis*, précédés et suivis de deux caractères de soulignement. Elles permettent de personnaliser le comportement des objets de classe standard en utilisant des opérateurs ou en simulant des comportements spécifiques.

Elles servent :

- ❑ à initialiser l'objet instancié ;
- ❑ à modifier son affichage ;
- ❑ à surcharger ses opérateurs ;

Introspection

L'introspection en Python fait référence à la capacité d'un programme à examiner ou à interroger ses propres structures de données, types et objets à l'exécution.

Cela signifie que Python offre des mécanismes pour inspecter dynamiquement le code, les objets et les modules à des fins de débogage, de génération de documentation, de réflexion ou d'autres tâches de manipulation de données.

Principales techniques d'introspection en Python :

La fonction ***type()*** : Vous pouvez utiliser la fonction ***type(objet)*** pour déterminer le type d'un objet.

```
x = 42
```

```
print(type(x)) # Output: <class 'int'>
```

Principales techniques d'introspection en Python

La fonction `dir()` : La fonction `dir(objet)` renvoie une liste des attributs (méthodes et variables) d'un objet. Cela permet de découvrir ce que vous pouvez faire avec un objet particulier.

```
x = [1, 2, 3]
print(dir(x)) # Liste des attributs de la liste
```

La fonction `help()` : Vous pouvez utiliser la fonction `help(objet)` pour obtenir une documentation détaillée sur un objet, y compris ses méthodes et ses attributs.

```
x = "Hello"
help(x) # Affiche la documentation de la chaîne de caractères
```


Principales techniques d'introspection en Python

Vous pouvez également inspecter dynamiquement les classes, leurs méthodes et leurs attributs. Par exemple, vous pouvez utiliser **dir(Classe)** pour obtenir la liste des attributs de la classe.

```
class MaClasse:  
    def __init__(self):  
        self.attribut = "Valeur"  
  
print(dir(MaClasse)) # Liste des attributs de la classe
```

Implémentation des interfaces

Python ne dispose pas d'un concept natif d'interface comme certains autres langages de programmation orientée objet tels que *Java* ou *C#*. Cependant, **Python** offre une approche plus souple et dynamique pour définir et implémenter des interfaces en utilisant des classes abstraites, des conventions de nommage et des vérifications à l'exécution.

Utilisation de classes abstraites (module abc) :

Le module **abc (Abstract Base Classes)** permet de créer des classes abstraites en **Python**.

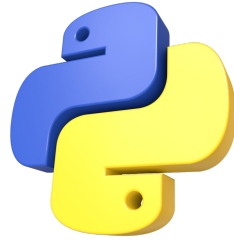
Une **classe abstraite** peut servir de modèle pour définir des méthodes que les classes dérivées doivent implémenter.

Implementation des interfaces.

```
from abc import ABC, abstractmethod
class Interface(ABC):
    @abstractmethod
    def methode_abstraite(self):
        pass
class ClasseConcrete(Interface):
    def methode_abstraite(self):
        print("Implémentation de la méthode abstraite")

objet = ClasseConcrete()
objet.methode_abstraite() # Appel de la méthode concrète
```

Dans cet exemple, Interface est une classe abstraite avec une méthode abstraite **methode_abstraite()**. **ClasseConcrete** hérite de cette interface et implémente la méthode abstraite.



Mécanisme d'exceptions pour la gestion des erreurs.

Mécanisme d'exception pour la gestion des erreurs en python.

En **Python**, les exceptions sont utilisées pour gérer les erreurs et les situations exceptionnelles qui peuvent survenir lors de l'exécution d'un programme. Le mécanisme des exceptions permet de détecter, de signaler et de traiter ces erreurs de manière élégante, plutôt que de laisser le programme planter avec un message d'erreur.

Déclenchement d'une exception :

Pour déclencher une exception, vous pouvez utiliser le mot-clé **raise** suivi du type d'exception. Par exemple, pour générer une exception de type **ValueError**, vous pouvez faire :

```
raise ValueError("Ceci est une exception personnalisée")
```

Mécanisme d'exception pour la gestion des erreurs en python.

Capture et gestion des exceptions :

Pour gérer une exception, vous utilisez un bloc **try** suivi d'un ou plusieurs blocs **except**. Le code dans le bloc **try** est susceptible de générer une exception, et les blocs **except** spécifient comment traiter ces exceptions.

Mécanisme d'exception pour la gestion des erreurs en python.

try:

```
    resultat = 10 / 0 # Division par zéro    # Code qui peut générer une exception
```

except ZeroDivisionError as erreur:

```
    print(f"Erreur : {erreur}") # Gestion de l'exception de division par zéro
```

except Exception as autre_erreur:

```
    print(f"Une autre exception s'est produite : {autre_erreur}") # Gestion d'autres exceptions
```

else:

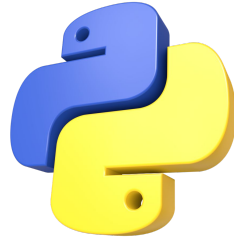
```
    print("Aucune exception n'a été levée.")    # Ce bloc est exécuté si aucune exception n'est levée
```

finally:

```
    print("Le bloc finally est exécuté.") # Ce bloc est toujours exécuté, qu'une exception soit levée ou non
```

Exceptions personnalisées.

```
class MonException(Exception):  
    pass  
  
try:  
    raise MonException("Ceci est ma propre exception")  
except MonException as erreur:  
    print(f"Mon exception personnalisée : {erreur}")
```

Utilisation de la bibliothèque Standard.

Les bibliothèques en python - La bibliothèque standard.

Dans **Python**, une bibliothèque est un ensemble logiciel de modules (classes (types d'objets), fonctions, constantes, ...) ajoutant des possibilités étendues à **Python** : calcul numérique, graphisme, programmation internet ou réseau, formatage de texte, génération de documents, etc

La bibliothèque standard de **Python** contient des modules natifs (écrits en C) exposant les fonctionnalités du système telles que les interactions avec les fichiers qui autrement ne seraient pas accessibles aux développeurs **Python**, ainsi que des modules écrits en **Python** exposant des solutions standardisées à de nombreux problèmes du quotidien du développeur.

Arguments passés sur la ligne de commande.

En Python, vous pouvez utiliser la bibliothèque standard (**Standard Library**) pour accéder aux arguments passés sur la ligne de commande via le module `sys` ou le module `argparse`. Voici comment faire avec ces deux approches : **Utilisation du module 'sys'**.

```
import sys
```

```
# Le premier élément de sys.argv est le nom du script lui-même
```

```
nom_du_script = sys.argv[0]
```

```
# Les arguments suivants sont les arguments passés sur la ligne de commande
```

```
arguments = sys.argv[1:]
```

```
print(f"Nom du script : {nom_du_script}")
```

```
print(f"Arguments : {arguments}")
```

Arguments passés sur la ligne de commande.

Pour exécuter ce script, vous pouvez l'appeler depuis la ligne de commande avec des arguments :

```
python mon_script.py arg1 arg2 arg3
```

Utilisation du module `argparse` :

Le module **`argparse`** est une bibliothèque plus puissante pour gérer les arguments de ligne de commande. Il vous permet de spécifier des arguments attendus, de documenter votre script et d'effectuer une validation avancée des arguments. Voici un exemple d'utilisation :

Arguments passés sur la ligne de commande.

```
import argparse
```

```
# Créez un objet ArgumentParser pour définir les arguments attendus
```

```
parser = argparse.ArgumentParser(description='Un exemple de script avec argparse')
```

```
# Ajoutez des arguments attendus
```

```
parser.add_argument('arg1', type=int, help='Premier argument entier')
```

```
parser.add_argument('arg2', type=float, help='Deuxième argument flottant')
```

```
parser.add_argument('--option', dest='option', action='store_true', help='Une option booléenne')
```

```
# Analysez les arguments de la ligne de commande
```

```
args = parser.parse_args()
```

Arguments passés sur la ligne de commande.

Accédez aux arguments analysés

```
arg1 = args.arg1
```

```
arg2 = args.arg2
```

```
option = args.option
```

```
print(f"arg1 : {arg1}")
```

```
print(f"arg2 : {arg2}")
```

```
print(f"option : {option}")
```

Pour exécuter ce script, vous pouvez l'appeler depuis la ligne de commande de la manière suivante :

```
python mon_script.py 42 3.14 --option
```

L'utilisation du moteur d'expressions régulières Python avec le module "re".

Une expression rationnelle (**regular expression ou RE**) spécifie un ensemble de chaînes de caractères qui lui correspondent ; les fonctions de ce module vous permettent de vérifier si une chaîne particulière correspond à une expression rationnelle donnée (ou si un expression rationnelle donnée correspond à une chaîne particulière, ce qui revient à la même chose).

Quand utiliser les expressions régulières ?

Les expressions régulières s'utilisent dans des domaines divers et variés. En effet, on peut s'en servir pour :

- Des recherches de correspondances ;
- Le remplacement d'expression ;
- L'analyse de données ;
- La validation de données ;
- Et le changement de formats

Utilisation des caractères spéciaux.

Les métacaractères.

Il existe quelques caractères qui, lorsque nous formulons nos expressions régulières, présentent un sens spécifique autre que le caractère lui-même. Voici ces caractères et leurs significations :

- ^** : précise que le résultat doit commencer par la chaîne de caractères qui le suit ;
- \$** : précise que le résultat doit se terminer par la chaîne de caractère qui le précède ;
- .** : correspond à n'importe quel caractère sauf le saut de ligne.

Utilisation des caractères spéciaux.

Les quantificateurs

Il existe également certains caractères qui sont considérés comme des quantificateurs. Ces caractères sont :

+ : qui signifie que le résultat peut contenir une ou plusieurs occurrences du caractère qui le précède

***** : qui veut dire que le résultat peut contenir zéro ou plusieurs occurrences du caractère qui le précède ;

? : qui signifie que le résultat peut inclure zéro ou une occurrence du caractère qui le précède ;

{ } : qui permet de préciser le nombre de nombres de répétitions des chaînes de caractères qui le précède ;

[] : qui permet d'obtenir le résultat sur plusieurs occurrences.

Le module 're'.

Le module **re** permet d'utiliser les expressions régulières en python.

En réalité, c'est un module d'extension du langage C qui a été inclus dans Python. Les expressions régulières vont être compilées en bytecode puis exécutées par un moteur de correspondance en C.

Grâce au module **re**, Python rend possible la manipulation des chaînes Unicode et des chaînes 8-bits. Vous pouvez effectuer plusieurs tâches telles que la recherche de correspondance, le remplacement de chaînes de caractères ou bien le découpage des chaînes en plusieurs morceaux.

La fonction de compilation d'une regex en Python

Pour initialiser une expression régulière avec **Python**, vous pouvez le compiler, surtout si vous serez amené à l'utiliser plusieurs fois tout au long de votre programme. Pour ce faire, il faut utiliser la fonction **compile()** comme suit :

```
x = re.compile(expression)
```

Vous pouvez spécifier un deuxième paramètre à la fonction `compile()` qui fera changer le comportement de l'expression suivant ce dernier. Ces paramètres sont appelés **Flags**. Parmi eux, nous avons : « **ASCII, A** », « **DOTALL, S** », « **IGNORECASE, I** », « **LOCALE, L** », « **MULTILINE, M** » et « **VERBOSE, X** ».

Reprenons notre exemple précédent, à savoir compiler l'expression « data », et ajoutons-y le paramètre **IGNORECASE**.

```
x = re.compile(expression, re.IGNORECASE)
```

Ici, vous obtiendrez une expression qui sera insensible à la casse. Le résultat des opérations sur cette expression pourra donc être data, Data, DATA, etc.

Les fonctions de recherche de correspondance.

Pour effectuer une recherche de correspondance, il existe plusieurs méthodes.

Vous pouvez, par exemple utiliser la méthode **match()** pour connaître si une chaîne de caractère peut correspondre à une expression donnée :

```
import re  
  
chaine = "Bonjour tous le monde"  
  
req = re.match(r"bonjour", strTest, re.I)  
  
print(req.group())
```

Utilisation de la fonction `re.match()`

```
import re

pattern = r"hello"
texte = "hello, world!"
resultat = re.match(pattern, texte)

if resultat:
    print("Correspondance trouvée:", resultat.group())
else:
    print("Aucune correspondance trouvée")
```

La fonction **`re.match(pattern, texte)`** est utilisée pour rechercher une correspondance au début du texte.

Elle renvoie un objet `Match` si une correspondance est trouvée, sinon elle renvoie `None`.

Les fonctions de recherche de correspondance.

Vous pouvez également utiliser **search()** pour effectuer des recherches sur une chaîne de caractère.

Contrairement à **match()**, **search()** va rechercher sur tout l'ensemble de la chaîne mais pas seulement au début.

```
import re

chaine = "Bonjour tous le monde"

req = re.search(r"tous", chaine, re.I)

print(req.group())
```

Utilisation de la fonction `re.search()`

```
>>> texte = "1,2,3 Je bois 2 litres d'eau par  
jour."  
  
>>> re.search(r"[a-zA-Z]+", texte).group()  
  
'Je'
```

La fonction **`search()`** recherche le modèle dans toute la chaîne, contrairement à **`match()`** qui vérifie uniquement le premier élément d'une chaîne.

La fonction de remplacement d'une chaîne de caractère

sub() qui vous permettra de remplacer la chaîne de caractère initiale par un autre ou bien de supprimer certains éléments de cette chaîne. Si nous revenons à notre exemple précédent :

```
import re
```

```
texte = """"Vous êtes au numéro 123 le suivant sera donc 124"""
```

```
regex = '\d+'
```

```
res = re.sub(r'\d','',texte)
```

```
print(res)
```

```
'Vous êtes au numéro le suivant sera donc'
```


Utilisation de la fonction re.sub()

Substituer un texte dans une chaîne avec la fonction sub().

```
>>> texte = "Je bois 2 litres d'eau par jour."  
>>> modele = re.sub(r"litres", "verres", texte)  
>>> modele  
"Je bois 2 verres d'eau par jour."
```

Manipulation du système de fichiers en python.

La manipulation du système de fichiers en **Python** se fait principalement à l'aide du module **"os"**.

Module "os" :

Le module **"os"** fournit des fonctionnalités pour interagir avec le système de fichiers, telles que la création, la suppression, le déplacement, la copie et la navigation dans les répertoires.

Obtenir le répertoire de travail actuel :

```
import os  
repertoire_actuel = os.getcwd()
```

Manipulation du système de fichiers en python.

Lister les fichiers et répertoires d'un répertoire :

```
fichiers_et_repertoires = os.listdir('/chemin/du/repertoire')
```

Créer un répertoire :

```
os.mkdir('/chemin/nouveau_repertoire')
```

Créer plusieurs répertoires :

```
os.makedirs('/chemin/nouveau_repertoire/intermediaire', exist_ok=True)
```

Manipulation du système de fichiers en python.

Supprimer un fichier :

```
os.remove('/chemin/vers/le/fichier')
```

Supprimer un répertoire vide

```
os.rmdir('/chemin/vers/le/repertoire')
```

Packaging et installation d'une bibliothèque Python.

Empaqueter et installer une bibliothèque Python implique de créer un package Python qui peut être facilement distribué et installé par d'autres utilisateurs. Pour ce faire, vous pouvez suivre les étapes suivantes :

Étape 1 : Structurer votre projet

Organisez votre code source en utilisant une structure de répertoires appropriée. Vous pouvez suivre une structure de projet typique comme celle-ci :

Structuration du projet.

```
mon_projet/  
    mon_module/  
        __init__.py  
        module.py  
    setup.py  
    README.md
```

mon_module est le nom de votre bibliothèque.

__init__.py est un fichier spécial qui indique à Python que ce répertoire est un package.

module.py contient le code de votre bibliothèque.

setup.py est un script Python qui décrit comment votre bibliothèque doit être empaquetée et installée.

README.md est la documentation de votre bibliothèque.

Créer le fichier setup.py

```
from setuptools import setup, find_packages

setup(
    name='mon_module',
    version='0.1',
    description='Ma super bibliothèque Python',
    author='Votre nom',
    author_email='votre@email.com',
    url='https://github.com/votre_utilisateur/mon_module',
    packages=find_packages(),
    install_requires=[
        # Liste des dépendances requises par votre bibliothèque
    ],)
```

Créer un fichier README.md

Le fichier **README.md** doit contenir des informations sur l'utilisation de votre bibliothèque, des exemples, des instructions d'installation, etc.

Il est important de fournir une documentation claire pour les utilisateurs de votre bibliothèque.

Créer un package source (sdist)

Pour créer un package source, exécutez la commande suivante dans le répertoire racine de votre projet :

```
python setup.py sdist
```

Cela générera un fichier **.tar.gz** dans un répertoire appelé dist. Ce fichier est prêt à être distribué.

Distribution d'une bibliothèque.

Vous pouvez distribuer votre bibliothèque de différentes manières, telles que sur **PyPI** (Python Package Index), via GitHub, en tant que paquet Debian/Ubuntu, etc. Le choix dépend de vos besoins.

Pour distribuer sur **PyPI**, vous devez créer un compte sur **PyPI**, puis utiliser des outils comme **twine** pour envoyer votre package. Voici comment le faire :

Installez **twine**

```
pip install twine
```

Uploadez le package sur **PyPI** en utilisant '**twine**'

```
twine upload dist/*
```

Installation d'une bibliothèque.

Les utilisateurs peuvent installer votre bibliothèque en utilisant **pip** :

```
pip install mon_module
```

Accès aux bases de données relationnelles.

L'accès aux bases de données relationnelles en Python se fait généralement en utilisant la **DB API (Python Database API)**, qui est une spécification standard pour l'interaction entre les applications Python et les bases de données relationnelles. La DB API fournit une interface unifiée pour interagir avec divers systèmes de gestion de bases de données (SGBD) tels que **MySQL, PostgreSQL, SQLite, Oracle, etc.**

Installation du pilote de base de données.

Avant de pouvoir interagir avec une base de données particulière, vous devez installer le pilote de base de données correspondant. Chaque **SCBD** a son propre pilote.

Par exemple, pour **SQLite**, vous pouvez utiliser le pilote **sqlite3**, qui est généralement inclus dans l'installation de Python. Pour d'autres SCBD, vous devrez peut-être installer un pilote spécifique. On peut le faire à l'aide de pip. Par exemple, pour **MySQL** :

```
pip install mysql-connector-python
```

Installation du pilote de base de données.

Après avoir installé le pilote, vous devez l'importer dans votre script Python en fonction du **SGBD** que vous utilisez. Par exemple, pour **MySQL**:

```
import mysql.connector
```

Connexion à la base de données.

On établit une connexion à votre base de données en utilisant les informations de connexion appropriées, telles que le nom d'utilisateur, le mot de passe et l'URL de la base de données.

Chaque SGBD peut avoir ses propres paramètres de connexion.

```
conn = mysql.connector.connect(  
    host = "localhost",  
    user = "root",  
    password = ""  
)  
print(conn)
```

Création d'un curseur.

Un curseur est un objet qui vous permet d'exécuter des requêtes SQL sur votre base de données. Vous pouvez créer un curseur à partir de la connexion que vous avez établie :

```
cur = conn.cursor()
```


Création d'une base de données.

En utilisant la méthode `execute()` du curseur, nous pouvons exécuter une opération de base de données ou une requête à partir de Python. La méthode `cursor.execute()` prend une requête MySQL comme paramètre et retourne le `resultSet`, c'est-à-dire une ligne de base de données.

```
cur.execute ("CREATE DATABASE my_db")
```

Exécution de requêtes SQL.

Vous pouvez exécuter des requêtes SQL à l'aide de méthodes du curseur, telles que `execute()`.

Par exemple, pour créer une table :

```
cur.execute ("CREATE TABLE IF NOT EXISTS users (  
    id INTEGER PRIMARY KEY,  
    username TEXT NOT NULL,  
    email TEXT NOT NULL  
)");
```

Insertion des données.

Pour insérer les données :

```
cur.execute("INSERT INTO users (username, email) VALUES (?, ?)",  
( 'john_doe', 'john@example.com' ))
```

Validation des transactions et fermeture de la connexion

Il est important de valider les transactions (commit) après avoir effectué des opérations de modification de la base de données, et de fermer la connexion lorsque vous avez terminé :

```
conn.commit() # Valider les modifications  
conn.close()  # Fermer la connexion
```

Récupération des résultats.

Si vous exécutez une requête `SELECT`, vous pouvez récupérer les résultats à l'aide de méthodes comme `fetchone()` ou `fetchall()` :

```
cur.execute("SELECT * FROM users")  
resultats = cur.fetchall()  
for row in resultats:  
    print(row)
```

Remarques :

Chaque SGBD peut avoir des spécificités dans la façon dont les connexions sont gérées, les requêtes SQL sont écrites, etc. Il est recommandé de consulter la documentation du pilote spécifique à votre SGBD pour plus de détails sur l'utilisation de la DB API avec ce SGBD particulier.



Outils QA (Quality Assurance)

QA – Quality assurance

L'assurance qualité se traduit comme étant l'aptitude à satisfaire le niveau de qualité désiré.

L'assurance qualité désigne toutes les actions planifiées et systématiques pour fournir une assurance adéquate qu'un produit, un service ou un résultat satisfera les exigences de qualité données et sera apte à l'utilisation.

Analyse statique de code (Pylint –Pychecker).

L'analyse de code statique est un terme générique utilisé pour décrire plusieurs types d'analyses.

L'analyse de code statique ne nécessite pas l'exécution de code pour fonctionner contrairement à l'analyse dite “**dynamique**”.

De nombreux outils permettent de procéder à l'analyse statique. Ces outils servent à s'assurer du respect de bonnes pratiques de codage : complexité cyclomatique – le contrôle du nombre de paramètre d'une fonction – la vérification du nommage de variable.

Pylint et **Pychecker** sont des outils d'analyse statique de code sources <<Python>>.

Présentation de Pylint.

Pylint est un outil de vérification statique du code **Python**. Il s'agit d'un outil très populaire qui analyse votre code source Python pour **détecter des erreurs de style, des problèmes de conformité aux conventions de codage PEP 8, des avertissements de qualité du code, des erreurs potentielles** et bien plus encore.

Pylint attribue des notes aux modules, classes, fonctions et variables de votre code, ce qui vous permet de connaître la qualité de votre code et d'améliorer sa lisibilité, sa maintenabilité et sa qualité globale. Voici quelques-unes des fonctionnalités et des avantages de **Pylint** :

Fonctionnalités et avantages de Pylint.

Détection d'Erreurs : Pylint identifie les erreurs de syntaxe et les problèmes qui pourraient entraîner une exécution incorrecte de votre programme Python.

Conventions de Codage : Il vérifie si votre code suit les conventions de codage Python PEP 8, ce qui facilite la lecture et la maintenance du code.

Complexité du Code : Pylint mesure la complexité cyclomatique de votre code, ce qui peut aider à identifier les fonctions ou les méthodes trop complexes.

Avertissements de Qualité : Il génère des avertissements pour des pratiques de codage potentiellement risquées ou des constructions de code inutiles.

Fonctionnalités et avantages de Pylint.

Avertissements de Qualité : Il génère des avertissements pour des pratiques de codage potentiellement risquées ou des constructions de code inutiles.

Nommage Conforme : Il vérifie que les noms de variables, de fonctions et de classes suivent les conventions de nommage PEP 8.

Documentation : Il vérifie si votre code est correctement documenté en utilisant docstrings et attribue des notes en fonction de la qualité de la documentation.

Compatibilité : Il vérifie la compatibilité entre les versions de Python, en signalant les utilisations de fonctionnalités obsolètes ou non prises en charge.

Personnalisable : Pylint est hautement personnalisable. Vous pouvez configurer ses règles en fonction des besoins spécifiques de votre projet en utilisant un fichier de configuration.

Installation & utilisation de Pylint.

Installation

Utilisez pip, l'outil d'installation des packages python.

pip install pylint

On peut ensuite vérifier l'installation en exécutant :

pip show pylint

Utilisation

On peut alors exécuter Pylint depuis l'invite de commande. Il est possible d'analyser le code d'un module ou de tout un package Python (pour être considéré comme un package, un dossier doit contenir le fichier `__init__.py`).

Pour analyser un module : **pylint my_module**

OU

pylint directory/my_module.py

Pour analyser tous les modules d'un package :

pylint my_package

Analyse des comptes rendus d'analyse.

Lorsque vous exécutez **Pylint** sur votre code **Python**, il génère des comptes rendus d'analyse qui contiennent différents types de messages, avertissements et erreurs. Comprendre ces messages est essentiel pour améliorer la qualité de votre code. Voici une explication des types de messages que **Pylint** peut générer :

Convention (C) : Les messages de ce type signalent des violations des conventions de **codage PEP 8**, telles que des problèmes de mise en forme, des noms de variables non conformes, des lignes trop longues, etc. Bien que ce ne soient généralement pas des erreurs critiques, il est recommandé de les corriger pour maintenir un code propre et lisible.

Analyse des comptes rendus d'analyse.

Avertissement (W) : Les avertissements indiquent des problèmes potentiels ou des constructions de code risquées, bien que le code puisse être techniquement correct. Ils attirent l'attention sur des pratiques potentiellement problématiques, mais le code peut toujours être exécuté avec succès.

Erreur (E) : Les erreurs indiquent des problèmes de programmation critiques qui empêchent généralement l'exécution du code. Vous devez corriger ces erreurs pour que votre code fonctionne correctement.

Message fatale (F) : Les messages fatals indiquent des problèmes très graves qui empêchent l'exécution du code. Il s'agit généralement d'erreurs de programmation majeures, et vous devez les résoudre immédiatement.

Analyse des comptes rendus d'analyse.

Conventions non disponibles (R) : Les messages de ce type indiquent des violations des conventions de **codage PEP 8** qui ne sont pas disponibles dans le fichier de configuration actuel.

Lorsque **Pylint** génère un rapport d'analyse, il affiche chaque message avec un code d'erreur unique et une description détaillée.

Par exemple :

C011: Missing docstring

Dans cet exemple, **C011** est le code d'erreur spécifique à la violation des conventions de codage, et "**Missing docstring**" est la description du problème

Extraction automatique de la documentation – Les docstrings.

Un **docstring** (abréviation de "documentation string") est une chaîne de caractères (string) placée au début d'un module, d'une classe, d'une fonction ou d'une méthode Python. Le docstring sert de documentation pour décrire ce que fait le module, la classe, la fonction ou la méthode, ainsi que la manière dont elle doit être utilisée. Les docstrings sont principalement utilisés pour générer de la documentation automatique et pour aider les développeurs à comprendre et à utiliser le code.

Les docstrings sont spéciaux en Python parce qu'ils sont accessibles à l'aide de l'attribut **__doc__** des objets correspondants. Voici comment un docstring est généralement structuré

Structuration docstrings

```
def ma_fonction(parametre1, parametre2):  
    """  
  
    Ceci est un docstring.  
  
    Il peut contenir plusieurs lignes de texte pour expliquer la fonction.  
  
    Args:  
  
    parametre1 (type): Description du paramètre 1.  
    parametre2 (type): Description du paramètre 2.  
  
    Returns:  
  
    type_de_retour: Description de la valeur de retour.  
    """  
  
    pass    # Corps de la fonction
```

Les docstrings.

Un **docstring** commence et se termine par des triples guillemets (simples ou doubles) pour permettre plusieurs lignes de texte. À l'intérieur du docstring, vous pouvez fournir des informations telles que :

- Une description générale de la fonction ou de la méthode.
- Une liste des paramètres, indiquant leur nom, leur type et leur description.
- Une description de la valeur de retour.
- Des exemples d'utilisation.
- Toute autre information pertinente pour aider les développeurs à comprendre le code.

Les docstrings.

Les **docstrings** sont souvent utilisés avec des outils de documentation tels que Sphinx pour générer de la documentation lisible par les humains à partir du code source. L'utilisation de **docstrings** pour documenter votre code **Python** est une bonne pratique, car elle rend votre code plus compréhensible et facilite la collaboration avec d'autres développeurs.

Par exemple, en utilisant un **docstring** approprié, un autre développeur peut obtenir des informations essentielles sur l'utilisation d'une fonction sans avoir à examiner le code source de la fonction.

Le débogueur de Python

Python est livré avec un débogueur intégré appelé **pdb (Python Debugger)** qui vous permet d'effectuer le débogage de votre code en exécution pas à pas et d'effectuer une analyse post-mortem en cas d'exception.

Exécution Pas à Pas avec pdb :

Pour utiliser **pdb**, vous devez d'abord importer le module `pdb` dans votre script Python.

```
import pdb
```

Le débogueur de Python.

Ensuite, vous pouvez insérer un point d'arrêt (breakpoint) dans votre code en utilisant la fonction `pdb.set_trace()`. Placez cette fonction à l'endroit où vous souhaitez commencer le débogage.

```
def ma_fonction():  
    # Code avant le point d'arrêt  
    pdb.set_trace() # Point d'arrêt  
    # Code après le point d'arrêt
```

Exécutez votre script Python normalement. Lorsque le point d'arrêt est atteint, l'exécution sera suspendue, et vous entrerez dans l'interpréteur **pdb**.

Les commandes couramment utilisées :

- c** : Continue l'exécution jusqu'au prochain point d'arrêt.
- n** : Exécute la ligne actuelle et se déplace à la suivante.
- s** : Exécute la ligne actuelle et entre dans les fonctions appelées.
- q** : Quitte le débogueur pdb et arrête l'exécution.
- p** variable : Affiche la valeur de la variable.
- l** : Affiche les lignes de code environnantes.
- bt** : Affiche la pile d'appels.
- h** : Affiche l'aide sur les commandes pdb.

Analyse Post-Mortem avec pdb :

En cas d'exception non gérée, Python peut automatiquement lancer le débogueur pdb en mode analyse post-mortem. Vous verrez une trace d'exception dans laquelle vous pouvez inspecter les variables et la pile d'appels.

Voici comment cela fonctionne :

```
import pdb
def division(a, b):
    return a / b

try:
    resultat = division(10, 0)
except ZeroDivisionError:
    pdb.post_mortem()
```

Dans cet exemple, une exception **ZeroDivisionError** est levée, et **pdb.post_mortem()** est appelé pour lancer le débogueur en mode analyse post-mortem.

On peut alors utiliser les commandes pdb pour inspecter l'état du programme au moment de l'exception.

Développement piloté par les tests.

Le développement piloté par les tests (**Test-Driven Development, TDD**) est une approche de développement logiciel qui consiste à écrire des tests unitaires pour votre code avant d'écrire le code lui-même. L'objectif principal du **TDD** est d'améliorer la qualité du code en s'assurant qu'il est correct, fiable et maintenable dès le début du développement.

Les modules de tests unitaires (Unittest...)

En Python, le module de tests unitaires standard est appelé **unittest**. **unittest** fait partie de la bibliothèque standard de Python et fournit un framework complet pour écrire, organiser et exécuter des tests unitaires dans vos projets Python. Il s'inspire en grande partie du framework de tests unitaires de Java, JUnit.

Le module Python unittest fournit l'outil **PyUnit**, outil que l'on retrouve dans d'autres langages : **JUnit** (Java), **NUnit** (.Net), **JSUnit** (Javascript), tous dérivés d'un outil initialement développé pour le langage SmallTalk : **SUnit**.

Les modules de tests unitaires (Unitest...)

PyUnit propose une classe de base, TestCase. Chaque méthode implémentée dans une classe dérivée de TestCase, et préfixée de test_, sera considérée comme un test unitaire.

```
"""Module de calculs."""  
# fonctions  
def moyenne(*args):  
    """Renvoie la moyenne."""  
    length = len (args)  
    sum = 0  
    for arg in args:  
        sum += arg  
    return sum/length  
def division(a, b):  
    """Renvoie la division."""  
    return a/b
```

Le module de test du module de calculs

```
"""Module de test du module de calculs."""  
  
# imports  
  
import unittest  
  
import os  
  
import sys  
  
dirName = os.path.dirname(__file__)  
if dirName == "":
```

Le module de test du module de calculs

```
    dirName = "."
dirName = os.path.realpath(dirName)
upDir = os.path.split(dirName)[0]
if upDir not in sys.path:
    sys.path.append(upDir)
from calculs import moyenne, division
# classes
class CalculTest(unittest.TestCase):
    def test_moyenne(self):
        self.assertEqual(moyenne(1, 2, 3), 2)
        self.assertEqual(moyenne(2, 4, 6), 4)
```

Le module de test du module de calculs

```
def test_division(self):
    self.assertEqual(division(10, 5), 2)
    self.assertRaises(ZeroDivisionError, division, 10, 0)
def test_suite():
    tests = [unittest.makeSuite(CalculTest)]
    return unittest.TestSuite(tests)
if __name__ == '__main__':
    unittest.main()
```

Remarques

Pour effectuer une « campagne de tests », il reste à créer un script qui :

- **recherche tous les modules de test : leurs noms commencent par test_ et ils sont contenus dans un répertoire tests ;**
- **récupère la suite, renvoyée par la fonction globale test_suite ;**
- **crée une suite de suites et lance la campagne.**

Exemple :

```
import unittest
# Répertoire où se trouvent les modules de test
directory = 'tests'
# Charger tous les modules de test dont les noms commencent par "test_"
loader = unittest.TestLoader()
test_suite = loader.discover(directory, pattern='test_*.py')
# Créer une suite de suites
suite_globale = unittest.TestSuite()
suite_globale.addTests(test_suite)
# Exécuter les tests
test_runner = unittest.TextTestRunner()
result = test_runner.run(suite_globale)
```


Exemple :

Afficher le rapport de test

```
print("\nRapport de tests :")
```

```
print(f"Tests exécutés : {result.testsRun}")
```

```
print(f"Échecs : {len(result.failures)}")
```

```
print(f"Erreurs : {len(result.errors)}")
```

```
print(f"Passés : {result.wasSuccessful()}")
```

L'automatisation des tests, l'agrégation de tests en python.

L'automatisation des tests et l'agrégation de tests en Python sont essentielles pour garantir la qualité de votre code et faciliter la détection précoce des problèmes. Vous pouvez automatiser l'exécution de plusieurs tests unitaires et les regrouper en suites de tests pour une gestion plus efficace.

Utilisez un Runner de Tests : Python propose plusieurs runners de tests pour automatiser l'exécution des tests, notamment unittest, nose, et pytest. Choisissez celui qui convient le mieux à votre projet.

Écrivez des Scripts d'Exécution : Créez des scripts ou des fichiers de configuration pour exécuter automatiquement vos tests. Ces scripts spécifient quels tests doivent être exécutés, comment les exécuter, et où enregistrer les résultats.

Intégration Continue : Utilisez des services d'intégration continue tels que Jenkins, Travis CI, CircleCI ou GitHub Actions pour automatiser l'exécution de vos tests à chaque fois que vous effectuez un commit ou une fusion de code.

L'automatisation des tests, l'agrégation de tests en python.

Agrégation de Tests en Suites de Tests :

L'agrégation de tests en suites de tests vous permet d'organiser vos tests en groupes logiques. Vous pouvez exécuter l'ensemble des tests d'une suite ou des suites spécifiques en fonction de vos besoins.

Aggrégation avec unittest

```
import unittest

class TestAddition(unittest.TestCase):
    def test_addition_positives(self):
        self.assertEqual(1 + 2, 3)

    def test_addition_negatives(self):
        self.assertEqual(-1 + (-2), -3)

class TestMultiplication(unittest.TestCase):
    def test_multiplication_positives(self):
        self.assertEqual(2 * 3, 6)
```

Aggrégation avec unittest

```
if __name__ == '__main__':  
    # Agréger les suites de tests  
    suite_addition = unittest.TestLoader().loadTestsFromTestCase(TestAddition)  
    suite_multiplication=unittest.TestLoader().loadTestsFromTestCase(TestM  
ultiplication)  
  
    # Créer une suite de tests globale  
    suite_globale = unittest.TestSuite([suite_addition, suite_multiplication])  
  
    # Exécuter tous les tests  
    unittest.TextTestRunner().run(suite_globale)
```

Test de couverture de code.

Les tests de couverture de code et le **profiling (ou profilage)** sont deux pratiques importantes dans le développement logiciel en Python. Ils permettent respectivement de mesurer la qualité de votre suite de tests et d'identifier les performances de votre code.

Tests de Couverture de Code (Code Coverage) :

La couverture de code mesure quelle proportion de votre code source est testée par vos tests unitaires. Une bonne couverture de code garantit que la majorité de votre code est soumise à des tests, ce qui réduit les risques d'erreurs non détectées.

Vous pouvez utiliser des outils de couverture de code tels que **coverage.py** pour générer des rapports de couverture de code.

Etapes pour utiliser 'coverage.py'

Installation : Installez coverage.py en utilisant pip :

pip install coverage

Exécution des Tests : Exécutez vos tests unitaires avec coverage run :

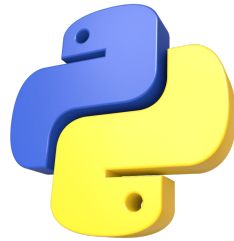
coverage run -m unittest discover -s tests -p 'test_*.py'

Génération du Rapport de Couverture : Générez un rapport de couverture de code en utilisant coverage report :

coverage report

HTML Report (optionnel) :

coverage html



Création IHM TKinter

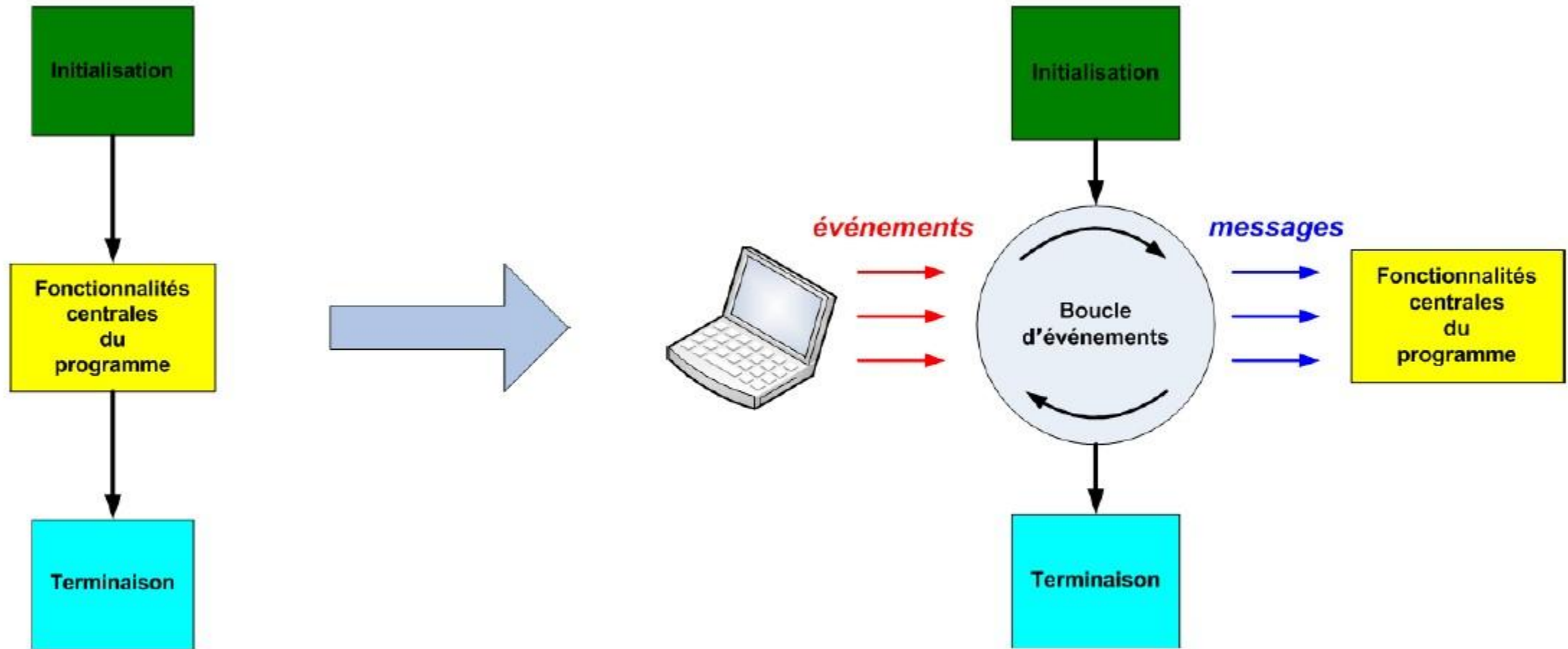
Présentation de la bibliothèque tKinter.

C'est une bibliothèque assez simple qui provient de l'extention graphique, **Tk**, du langage Tcl développé en 1988 par **John K. Ousterhout** de **l'Université de Berkeley**. Cette extention a largement essaimé hors de **Tcl/Tk** et on peut l'utiliser en **Perl**, **Python**, **Ruby**, etc. Dans le cas de Python, l'extension a été renommée tkinter.

Parallèlement à **Tk**, des extensions ont été développées dont certaines sont utilisées en Python. Par exemple le module standard Tix met une quarantaine de widgets à la disposition du développeur.

De son côté, le langage Tcl/Tk a largement évolué. La version 8.5 actuelle offre une bibliothèque appelée Ttk qui permet d'« habiller » les widgets avec différents thèmes ou styles. Ce module est également disponible en Python 3.1.1.

Programme pilotés par des événements.



(a) Programmation séquentielle

(b) Boucle d'événements

Principaux conteneurs.

Tkinter, la bibliothèque standard d'interface graphique pour **Python**, offre plusieurs types de conteneurs pour organiser et disposer des widgets (éléments d'interface utilisateur). Les principaux conteneurs Tkinter comprennent :

Frame : Le conteneur le plus basique de Tkinter. Un Frame est une boîte rectangulaire dans laquelle vous pouvez placer d'autres widgets. Il est souvent utilisé pour organiser et regrouper d'autres widgets.

LabelFrame : Une version spéciale du Frame conçue pour contenir des étiquettes (Label) et d'autres widgets. Il peut être utilisé pour créer des groupes de widgets avec des titres.

Canvas : Un conteneur qui permet de dessiner des formes graphiques, des images, et d'interagir avec des éléments graphiques à l'aide de la souris.

Text : Un widget multiligne qui permet d'afficher et d'éditer du texte. Il est souvent utilisé pour créer des éditeurs de texte simples.

Présentation des widgets disponibles.

On appelle widget, mot valise, contraction de window et gadget, les composants graphiques de base d'une bibliothèque.

Liste des principaux widgets de **tkinter** :

- **Tk** fenêtre de plus haut niveau
- **Frame** contenant pour organiser d'autres widgets
- **Label** zone de message
- **Button** bouton avec texte ou image
- **Message** zone d'affichage multi-lignes
- **Entry** zone de saisie
- **Checkbutton** bouton à deux états
- **Radiobutton** bouton à deux états exclusifs
- **Scale** glissière à plusieurs positions
- **PhotoImage** sert à placer des images sur des widgets

Présentation des widgets disponibles.

- **BitmapImage** sert à placer des bitmaps sur des widgets
- **Menu** associé à un Menubutton
- **Menubutton** bouton ouvrant un menu d'options
- **Scrollbar** ascenseur
- **Listbox** liste de noms à sélectionner
- **Text** édition de texte multi-lignes
- **Canvas** zone de dessins graphiques ou de photos
- **OptionMenu** liste déroulante
- **ScrolledText** widget Text avec ascenseur
- **PanedWindow** interface à onglets
- **LabelFrame** contenant avec un cadre et un titre
- **Spinbox** un widget de sélection multiple

Quelques méthodes applicables à tous les widgets.

| Méthode(paramètre) | Description |
|--------------------------------|--|
| <code>grid()</code> | Positionne le widget concerné dans son widget parent selon les numéros de ligne (<i>row = a</i>) et de colonne (<i>column = b</i>) indiqués en paramètres. |
| <code>destroy()</code> | Supprime un widget (<i>et tous ses widgets enfants</i>). |
| <code>configure(para)</code> | Modifie le (<i>ou les</i>) paramètre(s) désignés du widget. |
| <code>bind(evt, fct)</code> | Au déclenchement de l'événement evt dans le widget, la fonction fct , d' unique paramètre event , se lance. |
| <code>after(tps, fct)</code> | Relance l'action de la fonction fct dans le widget toutes les tps millisecondes. |
| <code>winfo_reqheight()</code> | Retourne la hauteur actuelle (<i>en px</i>) du widget. |
| <code>winfo_reqwidth()</code> | Retourne la largeur actuelle (<i>en px</i>) du widget. |

Le gestionnaire de fenêtres tkinter.

Un gestionnaire de fenêtres est un mécanisme qui vous permet de gérer plusieurs fenêtres ou cadres (frames) dans votre application graphique. Les gestionnaires de fenêtres sont utilisés pour afficher des fenêtres multiples, des boîtes de dialogue, des panneaux d'options, etc. Voici comment vous pouvez utiliser le gestionnaire de fenêtres principal **Tk()** ainsi que le gestionnaire de fenêtres supplémentaire **Toplevel()** dans Tkinter :

Gestionnaire de Fenêtres Principal Tk() :

Le gestionnaire de fenêtres principal **Tk()** est utilisé pour créer la fenêtre principale de votre application graphique. Vous ne pouvez avoir qu'une seule instance de **Tk()** dans une application Tkinter, car il représente la fenêtre racine de votre application.

Création d'une fenêtre principale

```
import tkinter as tk
```

```
# Crée la fenêtre principale
```

```
fenetre_principale = tk.Tk()
```

```
# Ajoutez ici des widgets et du contenu à la fenêtre principale
```

```
# Lancez la boucle principale
```

```
fenetre_principale.mainloop()
```


Gestionnaire de Fenêtres Supplémentaire Toplevel()

Le gestionnaire de fenêtres supplémentaire **Toplevel()** est utilisé pour créer des fenêtres de niveau supérieur indépendantes de la fenêtre principale. Vous pouvez avoir plusieurs instances de **Toplevel()** dans une application Tkinter pour créer des fenêtres de dialogue, des fenêtres contextuelles, etc.

Nous allons créer une fenêtre principale (fenetre_principale) et ajouté un bouton. Lorsque le bouton est cliqué, la fonction **afficher_fenetre_supplementaire()** crée une nouvelle fenêtre de niveau supérieur (fenetre_supplementaire) en utilisant **Toplevel()**.

Exemple

```
import tkinter as tk

def afficher_fenetre_supplementaire():

    # Crée une fenêtre de niveau supérieur

    fenetre_supplementaire = tk.Toplevel(fenetre_principale)

    # Ajoutez ici des widgets et du contenu à la fenêtre supplémentaire

fenetre_principale = tk.Tk() # Crée la fenêtre principale

# Crée un bouton pour afficher une fenêtre supplémentaire

bouton = tk.Button(fenetre_principale, text="Afficher Fenêtre",
command=afficher_fenetre_supplementaire)

bouton.pack()

fenetre_principale.mainloop() # Lancez la boucle principale
```

Le placement des composants, les différents layouts

En Tkinter, il existe plusieurs méthodes pour placer et organiser les composants (widgets) dans une fenêtre ou un cadre. Chaque approche a ses avantages et est adaptée à des scénarios spécifiques. Voici les principaux gestionnaires de disposition (layouts) disponibles en Tkinter :

pack() Layout : Le gestionnaire **pack()** permet d'empiler les widgets les uns en dessous des autres ou côte à côte, en fonction des options de placement. Il est simple à utiliser et convient aux mises en page linéaires.

```
widget1.pack()
```

```
widget2.pack()
```

Le placement des composants, les différents layouts

grid() Layout : Le gestionnaire **grid()** organise les widgets dans une grille de lignes et de colonnes. Vous pouvez spécifier la ligne et la colonne pour chaque widget, ainsi que la façon dont ils se comportent lors du redimensionnement de la fenêtre.

```
widget1.grid(row=0, column=0)  
widget2.grid(row=0, column=1)
```

Le placement des composants, les différents layouts

place() Layout : Le gestionnaire place() permet de positionner les widgets de manière précise en spécifiant leurs coordonnées x et y ainsi que leur largeur et leur hauteur. C'est utile pour des mises en page personnalisées.

```
widget1.place(x=10, y=20, width=100, height=30)  
widget2.place(x=50, y=80, width=120, height=40)
```

Le placement des composants, les différents layouts

Gestionnaires de Disposition Géométrique (**Geometry Managers**) : Ces gestionnaires combinent `pack()`, `grid()`, et `place()` pour offrir une plus grande flexibilité dans la disposition des widgets. Les principaux sont `pack()` avec `pack_propagate(0)`, `grid()` avec `grid_propagate(0)`, et `place()` utilisés en conjonction avec des cadres (Frames).

*Exemple (Utilisation de `pack()` avec **`pack_propagate(0)`** pour désactiver le redimensionnement automatique) :*

Le placement des composants, les différents layouts

```
cadre = tk.Frame(fenetre)
cadre.pack()
widget1 = tk.Label(cadre, text="Widget 1")
widget1.pack()
widget2 = tk.Label(cadre, text="Widget 2")
widget2.pack()
cadre.pack_propagate(0)
```

Ces gestionnaires de disposition peuvent être combinés et imbriqués pour créer des mises en page complexes.



Interfaçage Python/C.

Présentation du module Ctypes en python.

Le module **ctypes** en **Python** est une bibliothèque standard qui permet d'interfacer du code écrit en langage C avec **Python**. Il permet d'appeler des fonctions définies en C depuis Python et de manipuler des bibliothèques partagées (**DLLs**) ou des fichiers de bibliothèque dynamique (shared libraries). **ctypes** offre une interface simple pour travailler avec du code en C sans avoir besoin de compiler directement en Python.

Principales fonctionnalités du module ctypes.

Chargement de bibliothèques partagées C :

- Vous pouvez charger des bibliothèques partagées C en utilisant `ctypes.CDLL()` ou `ctypes.WinDLL()` pour les systèmes Windows.
- Par exemple : `mylib = ctypes.CDLL('mylib.so')`.

Types de données :

- ctypes fournit des types de données pour représenter les types C de base, tels que `ctypes.c_int`, `ctypes.c_double`, etc.
- Vous pouvez également définir des types de données personnalisés en utilisant `ctypes.Structure`.

Principales fonctionnalités du module ctypes.

Appel de fonctions C :

- Vous pouvez appeler des fonctions C chargées en utilisant la syntaxe **mylib.ma_fonction()**.
- Vous devez spécifier les types de paramètres et de retour des fonctions C en utilisant les attributs `.argtypes` et `.restype`.

Passage de paramètres :

- ctypes facilite le passage de paramètres Python aux fonctions C en utilisant des types de données appropriés. Par exemple, **ctypes.c_int(42)** représente un entier C.
- Vous pouvez également passer des tableaux ou des chaînes de caractères entre Python et C.

Principales fonctionnalités du module ctypes.

Gestion de la mémoire :

- **ctypes** gère automatiquement la gestion de la mémoire pour les objets C créés à l'intérieur de Python.
- Vous pouvez utiliser **ctypes** pour allouer et libérer manuellement la mémoire si nécessaire.

Utilisation avec des API système :

- **ctypes** est souvent utilisé pour interagir avec des API système, notamment des fonctions Windows, des bibliothèques système POSIX, etc.

Illustration de l'utilisation de la bibliothèque ctypes

```
import ctypes
```

```
# Charge une bibliothèque partagée C
```

```
mylib = ctypes.CDLL('mylib.so')
```

```
# Définit les types de paramètres et de retour de la fonction C
```

```
mylib.ma_fonction.argtypes = [ctypes.c_int, ctypes.c_int]
```

```
mylib.ma_fonction.restype = ctypes.c_int
```

```
# Appelle la fonction C depuis Python
```

```
resultat = mylib.ma_fonction(5, 3)
```

```
print("Résultat de la fonction C :", resultat)
```

Chargement d'une librairie C.

Pour charger une bibliothèque C en Python à l'aide du module `ctypes`, vous pouvez utiliser les fonctions **`ctypes.CDLL()`** ou **`ctypes.WinDLL()`** (pour Windows). Voici comment charger une bibliothèque C en Python :

Utilisation de `ctypes.CDLL()` (pour les systèmes non-Windows) :

```
import ctypes
```

```
# Charge la bibliothèque partagée C (SO ou DLL)
```

```
mylib = ctypes.CDLL('./mylib.so') # Remplacez 'mylib.so' par le nom de votre  
bibliothèque C
```

Chargement d'une librairie C.

Utilisation de `ctypes.WinDLL()` (pour Windows) :

```
import ctypes
```

```
# Charge la bibliothèque partagée C (DLL) sous Windows
```

```
mylib = ctypes.WinDLL('./mylib.dll') # Remplacez 'mylib.dll' par le nom de votre  
bibliothèque C
```

Après avoir chargé la bibliothèque C à l'aide de l'une de ces méthodes, vous pouvez accéder aux fonctions de la bibliothèque C en utilisant l'objet `mylib`. Par exemple, si votre bibliothèque C contient une fonction appelée **ma_fonction**, vous pouvez l'appeler comme ceci :

```
resultat = mylib.ma_fonction(arg1, arg2) # Remplacez 'arg1' et 'arg2' par les  
arguments nécessaires
```

Création de modules C pour Python.

La création de modules C pour Python vous permet d'étendre les fonctionnalités de Python en écrivant du code en C qui peut être utilisé comme un module Python standard. Voici les étapes de base pour créer un module C pour Python :

Étape 1 : Écriture du Code C

Créez un fichier source en C (*par exemple, monmodule.c*) contenant le code C que vous souhaitez exposer en tant que module Python. Assurez-vous que le code contient au moins une fonction qui sera appelée depuis Python.

Exemple en C ('monmodule.c')

```
#include <Python.h>
```

```
// Fonction C que nous voulons appeler depuis Python
```

```
static PyObject* ma_fonction(PyObject* self, PyObject* args) {  
    const char* message;  
    if (!PyArg_ParseTuple(args, "s", &message)) {  
        return NULL;  
    }  
    return Py_BuildValue("s", message);  
}
```

Méthodes exposées par le module

Définissez le tableau des méthodes exposées par le module, y compris la fonction que vous voulez rendre accessible depuis Python.

```
static PyMethodDef module_methods[] = {  
    {"ma_fonction", ma_fonction, METH_VARARGS, "Documentation de ma_fonction"},  
    {NULL, NULL, 0, NULL}  
};
```

Fonction d'initialisation du module

```
static struct PyModuleDef monmodule = {  
    PyModuleDef_HEAD_INIT,  
    "monmodule", // Nom du module  
    NULL,        // Documentation du module  
    -1,  
    module_methods  
};
```

Compilation du Module C

Utilisez un compilateur C pour compiler votre code source en un module Python partagé. La commande de compilation dépend de votre système d'exploitation.

```
gcc -o monmodule.so -shared -fPIC -I/usr/include/python3.8  
monmodule.c
```

Utilisation du Module dans Python

```
import monmodule
```

```
resultat = monmodule.ma_fonction("Bonjour depuis Python!")
```

```
print(resultat) # Affiche "Bonjour depuis Python!"
```

L'utilisation du profileur de code.

Un profileur de code est un outil qui vous permet d'analyser l'exécution de votre programme pour identifier les parties du code qui consomment le plus de temps et de ressources. Cela vous aide à optimiser les performances de votre programme en identifiant les goulets d'étranglement. Voici comment utiliser un profileur de code en Python, en utilisant le module cProfile comme exemple.

Étape 1 : Importation du module cProfile

Tout d'abord, importez le module cProfile dans votre script Python. Le module cProfile est inclus dans la bibliothèque standard Python et vous permet de profiler votre code.

```
import cProfile
```

L'utilisation du profileur de code.

Étape 2 : Définir la Fonction à Profiler

Sélectionnez la fonction ou le bloc de code que vous souhaitez profiler. Vous pouvez profiler une fonction spécifique ou tout le script.

Étape 3 : Création d'un Profiler

Créez un objet profiler en utilisant la fonction `cProfile.Profile()`.

```
profiler = cProfile.Profile()
```

L'utilisation du profileur de code.

Étape 4 : Démarrage et Arrêt du Profilage

Utilisez la méthode **.enable()** pour activer le profiler avant d'exécuter le code à profiler, puis utilisez la méthode **.disable()** pour le désactiver après l'exécution.

```
profiler.enable()  
# Code à profiler  
profiler.disable()
```

Étape 5 : Lancement du Profilage

Exécutez le code que vous souhaitez profiler. Le profileur enregistre les statistiques de performance pendant cette phase.

Exemple concret

```
import cProfile
def fonction_a_profiler():
    total = 0
    for i in range(10000000):
        total += i
    return total
if __name__ == "__main__":
    profiler = cProfile.Profile()
    profiler.enable()

    resultat = fonction_a_profiler()
    print("Résultat :", resultat)
    profiler.disable()
    profiler.print_stats()
```



Conclusion.

Analyse critique de Python.

Python est un langage de programmation populaire et largement utilisé, reconnu pour sa simplicité, sa lisibilité, sa polyvalence et sa grande communauté de développeurs. Cependant, comme tout langage de programmation, Python présente à la fois des avantages et des inconvénients (Problématique du temps réel, problème de performance, Gestion de la mémoire...)

En fin de compte, le choix d'utiliser Python dépend des besoins spécifiques du projet et des compromis entre simplicité, lisibilité, performance et polyvalence. Python est un excellent choix pour de nombreux types de projets, mais il peut ne pas convenir à tous. Il est essentiel de comprendre ses avantages et ses limites pour prendre une décision éclairée.

Les évolutions possibles du langage.

Python 4 : À mesure que Python continue de se développer, une nouvelle version majeure, Python 4, pourrait être envisagée. Cela pourrait introduire des fonctionnalités majeures, des améliorations de performances et des changements importants dans la syntaxe.

Performance Améliorée : L'amélioration des performances est une préoccupation constante. Les futures versions de Python pourraient introduire des optimisations pour rendre le langage plus rapide et plus efficace, notamment grâce à des optimisations de la gestion de la mémoire et des améliorations du GIL.

Sécurité : Avec les préoccupations croissantes en matière de sécurité, Python pourrait se concentrer sur des améliorations de sécurité pour aider les développeurs à écrire un code plus sécurisé par défaut.

Eléments de webographie et de bibliographie.

Gérard Swinnen – Apprendre à programmer avec Python

Hugues Bersini – La programmation orientée objet

<https://docs.python.org/>

<https://supports.uptime-formation.fr/03-python/>

<https://gilles-hunault.leria-info.univ-angers.fr/>

<https://hackernoon.com/fr/>

<https://zestedesavoir.com/>

<https://sametmax2.com/ce-quil-faut-savoir-en-python/index.html>