

Python

Présentation de Python

Historique

- Création en 1989 au Pays-Bas par Guido Van Rossum.
- Première version publique (0.9.0) en 1991.▪
- Objectif :
 - Améliorer et étendre les fonctionnalités du Bourne Shell, l'interface utilisateur des systèmes Unix de l'époque.
- Version 2.0 en 2000
- Version 3.0 en 2008
- Dernière Version 3.11 Novembre 2022
- La licence libre de Python, oscillant entre la GPL et la licence Apache est créé en même temps que la Python Software Foundation
 - Python Software Foundation License

Cas d'utilisation

- Programmation système et réseau
 - Création d'utilitaires complémentaires ou bien spécifique pour les OS
- DataCenter :
 - Manipulation de données
- Programmation scientifique
 - Calculs avancés
- Programmation Web
 - Frameworks Web MVC (Django)
- Informatique embarquée
 - Raspberry PI, Arduino, ...
- Programmation graphique
 - Applications de bureau, prototypage de jeux vidéo
- Education
 - Apprentissage et mise en pratique de l'algorithmie

Caractéristiques du langage

- Simple, concis et lisible
- Bibliothèque standard riche de fonctionnalités
- Multitude de bibliothèques complémentaires disponibles
- Communauté très étendue
- Multiplateforme

- Adapté aux systèmes d'exploitation
- Typage dynamique fort
- Possède plusieurs approches pour la structuration des programmes
 - Fonctionnelle, objet, ...

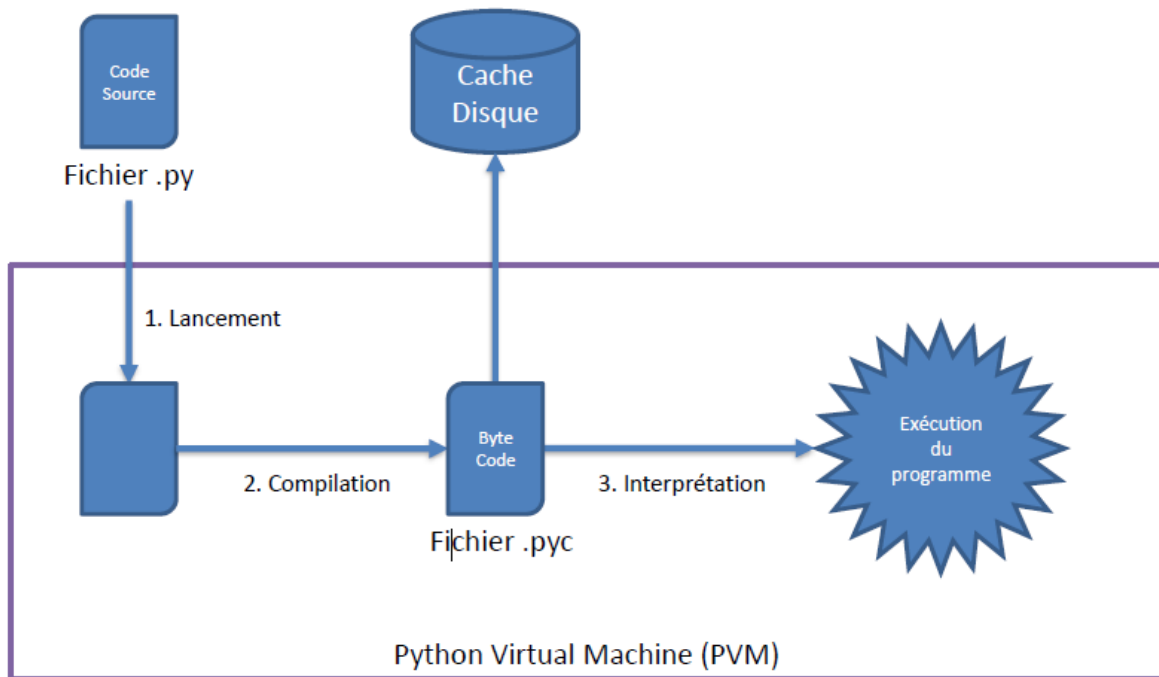
Philosophie Python

- Simplicité d'écriture: le temps d'exécution n'est pas aussi important que le temps nécessaire à l'écriture du code.
 - pour accomplir une certaine tâche, il vous faut une journée pour écrire un programme, qui s'exécute en une seconde.
 - certains utilisateurs préféreront, pour accomplir la même tâche, avoir un programme qui s'exécute en cinq minutes, mais qui s'écrit en une demi-heure.
- Quand la performance est devenue un problème, la communauté a amélioré le langage.
- Basé sur la confiance et la bonne utilisation des développeurs :
 - Pas de constante en python
 - Une constante est une variable qu'on ne doit pas faire évoluer. Pour cela on va simplement déclarer l'identifiant en majuscule.

Le contenu de Python

- Un langage
 - Avec une syntaxe simple
 - Une grammaire étendue
- Des implémentations
 - Pour faire «tourner» un programme Python !
 - CPython (Implémentation de référence)
 - Jython
 - IronPython
- Une bibliothèque standard
 - Elle offre les fonctionnalités de base du langage
- Des bibliothèques complémentaires
 - Mises à disposition par la communauté
 - Elles couvrent des besoins complémentaires à la bibliothèque standard

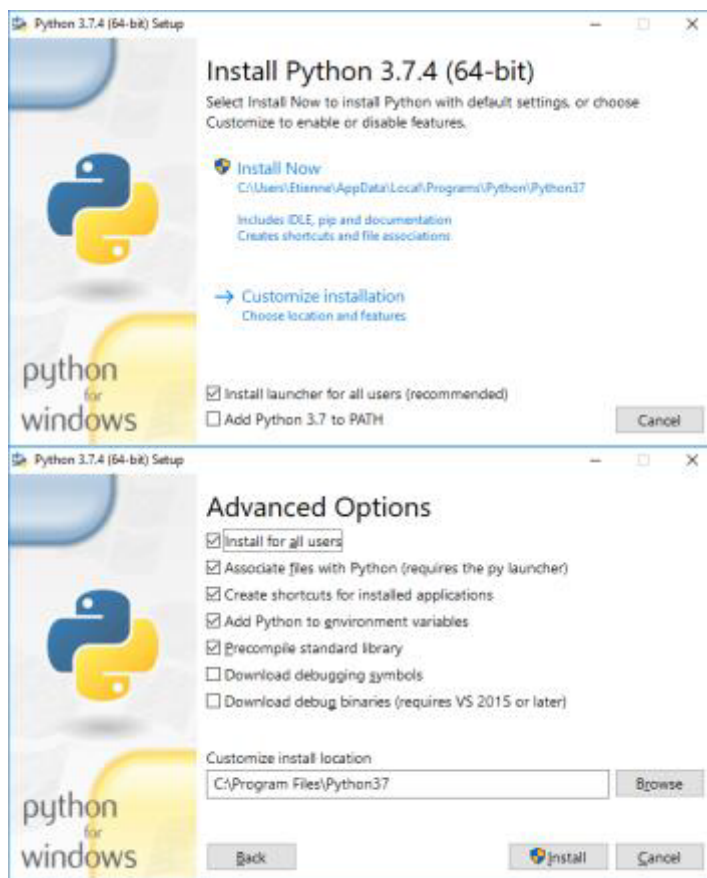
Exécution d'un programme Python



Mise en place d'une plateforme Python

Installation de Python

- Installer une implémentation !
 - Pour commencer, utiliser l'implémentation de référence (CPython) disponible sur le site officiel :
<http://www.python.org>
 - Des versions pour Windows et MacOS sont disponibles
 - Pour Linux, soit Python est déjà installé dans la version souhaitée, soit il faudra utiliser le gestionnaire de logiciels de sa distribution
- L'installateur propose des options de personnalisation
 - Emplacement d'installation
 - Fonctionnalités à installer



La console Python

- La console Python est un outil interactif permettant d'exécuter à la volée des instructions de code
 - A la manière d'un shell Unix !
- La commande **python** dans un terminal ou interpréteur de commande permet de lancer la console
 - L'instruction **exit()** permet d'en sortir

```

C:\Users\Etienne>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print("Python works !")
Python works !
>>>
>>> exit()

C:\Users\Etienne>

```

- Cet outil permet d'évaluer rapidement des portions de code
- La console est parfois directement intégrée dans les IDE pour Python

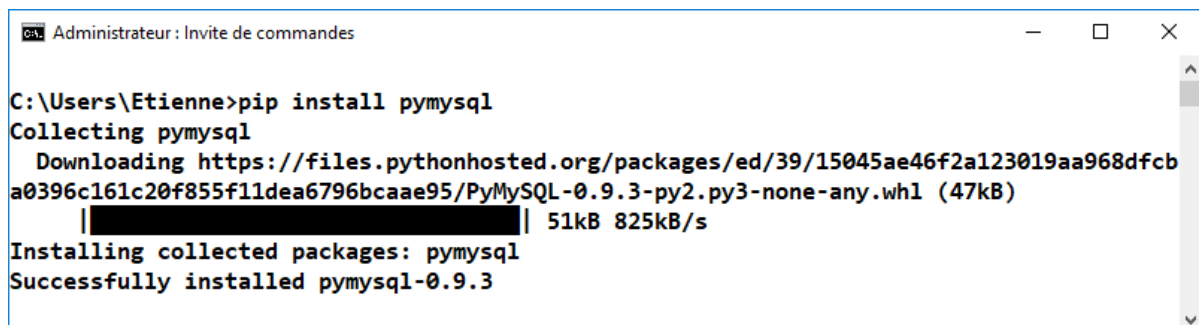
Les bibliothèques additionnelles

- Même si Python vient avec une bibliothèque standard très riche, il est parfois nécessaire d'ajouter des bibliothèques supplémentaires pour couvrir des besoins spécifiques
- 2 options essentiellement :
 - Installation avec PIP

- Installation à partir des sources
- Installation avec PIP
 - Option la plus pratique
 - Une installation de Python vient avec un utilitaire en ligne de commande permettant de télécharger et d'installer des paquets Python

Installation avec PIP

- Option la plus pratique
- Une installation de Python vient avec un utilitaire en ligne de commande permettant de télécharger et d'installer des paquets Python : **pip**
 - **pip** s'appuie sur un référentiel accessible à l'adresse x pour effectuer les téléchargements



```

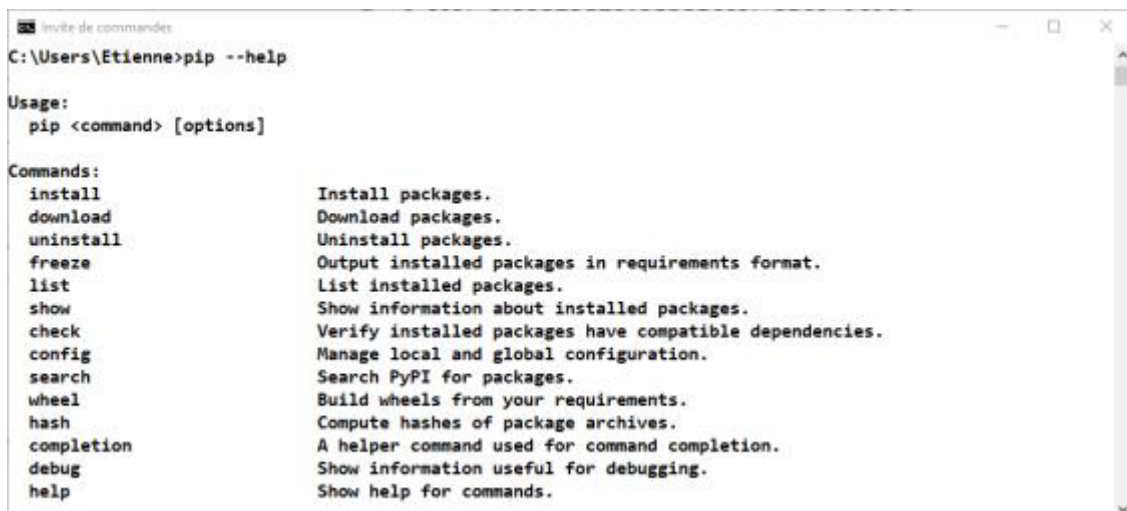
Administrateur : Invite de commandes

C:\Users\Etienne>pip install pymysql
Collecting pymysql
  Downloading https://files.pythonhosted.org/packages/ed/39/15045ae46f2a123019aa968dfcb
a0396c161c20f855f11dea6796bcaae95/PyMySQL-0.9.3-py2.py3-none-any.whl (47kB)
    |████████████████████| 51kB 825kB/s
Installing collected packages: pymysql
Successfully installed pymysql-0.9.3
  
```

- **pip** possède plusieurs commandes pour installer, désinstaller, rechercher, lister les paquets

Les commandes de PIP

- La commande **pip --help** donne la liste des commandes ainsi que des options de pip.



```

Invite de commandes

C:\Users\Etienne>pip --help

Usage:
  pip <command> [options]

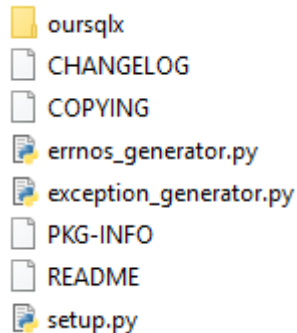
Commands:
  install      Install packages.
  download     Download packages.
  uninstall    Uninstall packages.
  freeze       Output installed packages in requirements format.
  list         List installed packages.
  show         Show information about installed packages.
  check        Verify installed packages have compatible dependencies.
  config       Manage local and global configuration.
  search       Search PyPI for packages.
  wheel        Build wheels from your requirements.
  hash         Compute hashes of package archives.
  completion   A helper command used for command completion.
  debug        Show information useful for debugging.
  help         Show help for commands.
  
```

- Parmi les principales, on notera :
 - **install** : Pour installer un paquet
 - **uninstall** : Pour désinstaller un paquet
 - **freeze** : Pour afficher la liste des paquets installés dans un format « requirements »
 - pip freeze > requirements.txt
 - On dresse la liste des paquets installés, redirigée vers un fichier.
 - pip install -r requirements.txt

- Permet, sur une autre machine par exemple, d'installer les paquets listés dans le fichier.

Installation à partir des sources

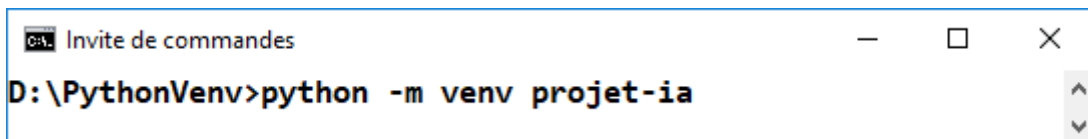
- Certaines librairies Python sont livrées sous forme de code source dans des archives compressée (format ZIP ou TAR.GZ)
- Leur installation se fait à partir de la ligne de commande, grâce à l'interpréteur Python.
 - Un fichier README est fréquemment fournit pour indiquer la procédure d'installation.
- Mais la démarche est souvent la même !
 - Le fichier setup.py permet la configuration et l'installation.



```
python setup.py install
```

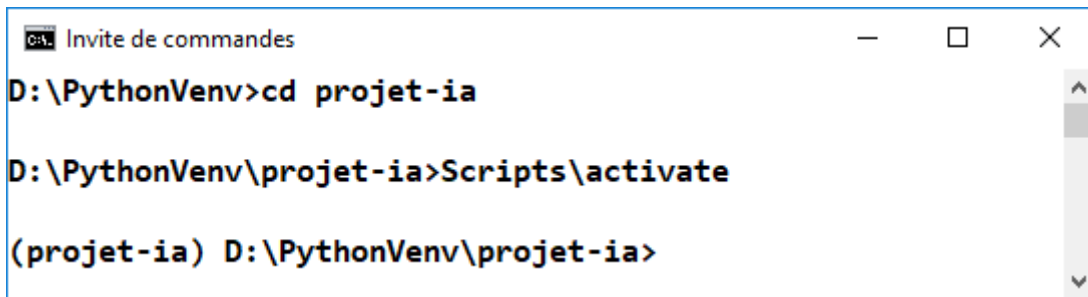
Les environnements virtuels

- Les environnements virtuels de Python permettent de «cloner» une installation de Python dans un dossier dédié.
- Il est ensuite possible d'y installer toutes les librairies nécessaires, sans toucher à l'installation de base de l'interpréteur, évitant ainsi :
 - De surcharger cette dernière d'une multitude de librairies ;
 - De générer des conflits entre les librairies ;
 - De mélanger les spécificités de différents projets.
- La création d'un environnement virtuel se fait en une seule commande. Elle va créer un dossier dédié.



Les environnements virtuels

- Une fois l'environnement virtuel créé, il est nécessaire de l'activer pour pouvoir l'utiliser, le prompt change pour indiquer l'environnement actif.
 - Sous Linux, le script activate se trouve dans le sous-répertoire bin/, sous Windows, dans le sous-répertoire Scripts.



```
Invite de commandes
D:\PythonVenv>cd projet-ia
D:\PythonVenv\projet-ia>Scripts\activate
(projet-ia) D:\PythonVenv\projet-ia>
```

- Le script deactivate permet de désactiver un environnement virtuel.

Les IDE pour Python

- Plusieurs environnements de développement intégré existent pour Python.
- Ils permettent de disposer de fonctionnalités essentielles pour tout développeur, comme par exemple :
 - L'assistance à la création de projet
 - La coloration syntaxique du code
 - L'assistance à la saisie du code
 - Une console Python intégrée
 - Un lancement facilité des programmes
 - Un débogueur pour la mise au point des programmes
- Parmi les IDE les plus populaires, on trouve notamment :
 - PyCharm
 - Référence dans le domaine. Edition «Community» (Gratuite) et «Professional» (Commerciale)
 - <https://www.jetbrains.com/pycharm/>
 - PyDev
 - Base Eclipse associée à un plugin pour la prise en charge de Python
 - <http://www.pydev.org>
 - Visual Studio Code
 - Outil Microsoft multi-plateforme

Les bases du langage

Syntaxe du langage

- Voici quelques principes énoncés par Guido van Rossum :
 - Simple et intuitif
 - Compréhensible, aussi simple à comprendre que de lire l'anglais
 - Approprié pour les tâches quotidiennes
 - Permettant des temps de développement courts

Instructions et délimiteurs

- A la différence d'autres langages, Python n'utilise pas de caractère spécifique pour délimiter les instructions de code.
 - Une instruction étant un ordre simple que le langage doit exécuter
- Le saut de ligne suffit à terminer l'instruction courante
 - Pas d'usage du «;» comme en C, C++, Java, ...
- Il est parfois nécessaire, par soucis de lisibilité, d'écrire des instructions sur plusieurs lignes. Dans ce cas, on utilisera le «\» pour indiquer que l'instruction courante se poursuit sur la ligne suivante.

```
print ("hello world")
chaîne = "Une instruction de code vraiment beaucoup trop longue" \
        "pour être lisible si elle est écrite sur une seule et" \
        "unique ligne !"
print ( chaîne )
```

Les blocs

- En algorithmie, les blocs servent à délimiter une séquence d'instructions.
- L'objectif étant d'exprimer un périmètre de validité à ces instructions.
- En Python, l'usage du «:» permet de démarrer un bloc, ensuite, il est nécessaire d'utiliser une tabulation supplémentaire par rapport au niveau de tabulation courant, pour exprimer le contenu du bloc.
 - Les tabulations ne sont donc pas cosmétiques en Python !
- Exemple :

```
def simple_methode(valeur):
    if valeur < 0 :
        res = "negatif"
```

Structure d'un programme

- La réalisation d'un script python peut se réaliser de plusieurs façons :
 - Ecriture depuis la console Python qui est un outil interactif permettant d'exécuter à la volée des instructions de code

```
C:\Users\TDouchet>python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print ("hello world")
hello world
>>>
```

- Ecriture dans un fichier qui a une extension *.py.

```
print ("hello world")
```

- Python ne propose pas de point d'entrée d'application :
 - Pas de main comme dans certains langages

- L'ensemble des instructions est exécutée
- Aucune fonction n'est appelée automatiquement.
- Ce manque de structure fait partie intégrante de la philosophie mais peut être perturbant pour dans des projets pour la relecture de code.
- Il est donc possible de définir un point d'entrée :

```
if __name__ == '__main__':
    print ("hello world")
    print (repr (__name__))
```

- Un module `__name__` est défini égal à `'__main__'` lorsqu'il est lu à partir d'une entrée standard, d'un script ou d'une invite interactive.
- `__name__` est stocké dans l'espace de noms global du module

Les types de données simples

- Les types de données simples
 - Entiers (**int**)
 - Réels (**float**)
 - Chaines de caractères (**str**)
 - Entre " ou '
 - Booléens (**bool**)
 - Mots clés **True** et **False**

```
if __name__ == '__main__':
    un_entier = 12
    print (type (un_entier))
    un_reel = 3.1415
    print(type(un_reel))
    un_booleen = True
    print(type(un_booleen))
    une_chaine = "hello world"
    print ( type (une_chaine))
```

Typage dynamique

- Typage dynamique
 - Le typage d'une variable est réalisé lors de son affectation.
 - Le type peut changer au cours du déroulement du programme

```
un_entier = 12
print ("valeur de un entier ",un_entier,"type : " , type ( un_entier))
un_entier = "121"
print("valeur de un entier ", un_entier, " type : ", type(un_entier))
```

Typage dynamique fort

- Typage fort
 - Pas de conversion implicite par commodité !
 - Un entier reste un entier

```
un_entier = "121"  
print("valeur de un entier ", un_entier, " type : ", type(un_entier))  
resultat = un_entier + 3
```

Traceback (most recent call last):

File "[C:/TpDev/TpFormation/SupportCours/pbase/mtype](#)", line 45, in <module>
 resultat = un_entier + 3

TypeError: can only concatenate str (not "int") to str

Les fonctions de conversion

- Chaque type de données simple possède une fonction de conversion permettant de convertir une donnée vers ce type.
 - **str(), int(), float()**

```
un_entier = "121"  
print("valeur de un entier ", un_entier, " type : ", type(un_entier))  
resultat = int( un_entier) + 3  
print("resultat : " , resultat )
```

Littéraux numériques

- But :
 - Augmenter la lisibilité de certains nombres
 - Se rapprocher de leur présentation dans la vie courante
- Moyens :
 - Inclusion du caractère «_»
 - Autant de fois que nécessaire
- Exemple

```
mille_milliards= 1_000_000_000_000
```

Les constantes

- Python ne permet pas de déclarer des constantes (pas de mot clé).
- La réalisation de constante est basée sur des conventions :
 - Déclaration globale
 - Identifiant représenté en lettres majuscules
- C'est un moyen d'avertir le programmeur que cette donnée ne doit pas être modifiée :

Les opérateurs

- Un opérateur est un caractère ou une suite de caractères à laquelle la grammaire de Python donne une signification particulière.
- Les opérateurs Python sont regroupés dans les familles suivantes :
 - Opérateurs arithmétiques
 - Opérateurs logiques
 - Opérateurs d'assignation
 - Opérateurs de comparaison
 - Opérateurs d'identité
 - Opérateurs d'inclusion

Opérateurs : Arithmétiques & Logiques

- Opérateurs arithmétiques
 - Les opérateurs arithmétiques sont utilisés avec des valeurs numériques pour effectuer des opérations mathématiques courantes :

Opérateur	Nom	Exemple
+	Addition	$x + y$
-	Soustraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulo	$x \% y$
**	Exponentiel	$x ** y$
//	Partie entière de la division	$x // y$

- Opérateurs logiques
 - Les opérateurs logiques sont utilisés pour combiner les instructions conditionnelles :

Opérateur	Description	Exemple
and	Retourne Vrai si les deux énoncés sont vrais	$x < 5$ and $x < 10$
or	Retourne Vrai si l'un des énoncés est vrai	$x < 5$ or $x < 4$
not	Inverse le résultat, retourne Faux si le résultat est vrai	not($x < 5$ and $x < 10$)

Opérateurs : Assignment

- Les opérateurs d'assignation sont utilisés pour assigner des valeurs aux variables :

Opérateur	Exemple	Equivaut à
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Opérateurs : Comparaison & Identité

- Opérateurs de comparaison Python
 - Les opérateurs de comparaison sont utilisés pour comparer deux valeurs :

Opérateur	Nom	Exemple
==	Egal	x == y
!=	Différent	x != y
>	Supérieur à	x > y
<	Inférieur à	x < y
>=	Supérieur ou égal à	x >= y
<=	Inférieur ou égal à	x <= y

- Opérateurs d'identité
 - Les opérateurs d'identité sont utilisés pour comparer les objets, non pas s'ils sont égaux, mais s'ils sont le même objet, avec le même emplacement mémoire :

Opérateur	Description	Exemple
is	Retourne Vrai si les deux variables sont le même objet	x is y
is not	Retourne Vrai si les deux variables ne sont pas le même objet	x is not y

Opérateurs : Inclusion

- Opérateurs d'inclusion
 - Les opérateurs d'inclusion sont utilisés pour tester si une séquence est présente dans un objet.
 - Un élément dans une liste, un mot dans une chaîne, ...

Opérateur	Description	Exemple
in	Retourne Vrai si une séquence avec la valeur spécifiée est présente dans l'objet	x in y
not in	Retourne Vrai si une séquence avec la valeur spécifiée n'est pas présente dans l'objet	x not in y

Les chaînes de caractères

- Les chaînes de caractères, comme tous les types de données, sont des objets. A ce titre, une chaîne de caractères possède un certain nombre de méthodes internes permettant leur manipulation.
- Les chaînes de caractères s'écrivent entre guillemets ou apostrophe.

```
message = "My name is Bond"
langage = 'Mon langage favori Python'
print ( message )
print ( langage )
```

Les slices

- Les slices sont des expressions qui permettent d'extraire des éléments d'une liste ou d'une chaîne de caractères :

```
def test_slices() :
    message = "Formation Python 06/11/2023"
    print ( message[10:17] )
    print ( message [:9])
    print ( message[-1])
    print (message [-10:])
    res = message[::]
    print ( res )
    print ( message [::-1])
```

La classe Str

- La classe Str est immuable :
 - Une chaîne de caractères ne peut pas être modifiée

```
message = "langage Python"
message_maj = message.upper()
print (f"{message} {message_maj}")
```

- Il existe de nombreuses fonctionnalités de manipulation de chaînes (voir documentation)

```
message = "langage Python"
print ( message.find ( "Python"))
print ( "Python" in message )
print (message.split())
print ('join')
print("-".join(["James" , "Bond"]))
```

Formatage des chaines de caractères

- Le formatage des chaines de caractères se fait en utilisant l'opérateur modulo (%). Voici quelques possibilités :

```
langage = 'Python'
duree = 4
print("Bienvenue à la formation %s de %d jours" % (langage , duree) )
```

- Python pousse à l'usage d'une autre approche, utilisant la méthode format() des chaines de caractères :

```
langage = 'Python'
duree = 4
print("Bienvenue à la formation {0} de {1} jours".format(langage, duree) )
print("Bienvenue à la formation {langage} de {duree} jours" \
      .format(langage=langage, duree=duree))
```

Formatage des chaines de caractères (3.6)

- Python propose un nouveau mécanisme pour le formatage de chaîne de caractères : les f-string.

f"{valeur:formatage}"

- Il est possible de spécifier des options de formatage avancées directement à l'intérieur des accolades.

```
langage = 'Python'
duree = 4
print(f"Bienvenue à la formation {langage} de {duree} jours")
print(f"Bienvenue à la formation {langage:10} de {duree:4} jours")
```

Les chaînes de caractères

- La gestion des chaîne de caractères a évolué entre Python et python 3:
 - en 2.7, les chaînes sont par défaut des arrays d'octets, et il faut les décoder pour obtenir de l'unicode.
 - en 3, les chaînes sont par défaut de type 'unicode', et il faut les encoder pour obtenir de un array d'octets.
- La version 3 prend plusieurs mesures pour éviter les bugs
 - L'encodage par défaut du code est UTF8.
 - L'encodage par défaut de lecture et d'écriture est UTF8.
 - On ne peut plus mélanger 'bytes' et 'unicode'.

```
langage = b"Bienvenue en Python"
print ( langage, type(langage))
str = langage.decode ( "utf8")
print ( str , type(str))
```

Autres types de données

- Les types évolués
 - Listes (list)
 - Mutables
 - Liste non modifiable (tuple)
 - Immutables !
 - Dictionnaires (dict)
 - Tableaux associatifs

```
liste = ["un", "deux", "trois"]
print("type", type(liste))
ensemble = ("un" , "deux" , "trois")
print ( "type" , type (ensemble))
dico = {"un":1, "deux":2, "trois":3}
print("type", type(dico))
```

- Le type spécial
 - None
 - Par exemple un retour nul de fonction

Les listes (list) : création & déclaration

- La classe «list» fournît un ensemble de méthodes :

```

liste = ["un", "deux", "trois", "quatre"]
liste.append("cinq")
print (liste)
liste[2] = "DEUX"
liste.remove ("quatre")
for elem in liste:
    print (elem)

```

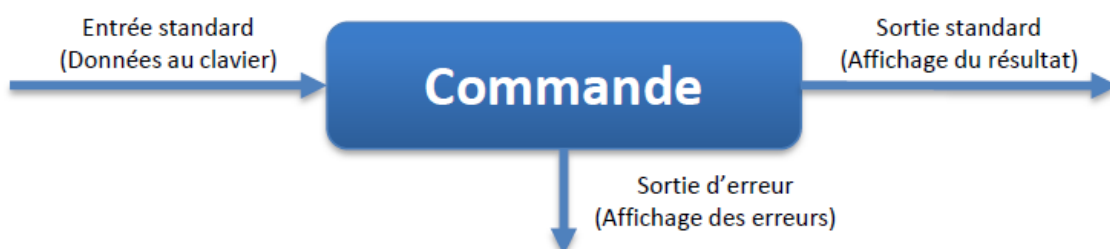
```

['un', 'deux', 'trois', 'quatre', 'cinq']
un
deux
DEUX
cinq

```

Entrée et sortie standards

- Les notions d'entrée et de sorties d'un système d'exploitation sont historiquement liées aux systèmes Unix et à la ligne de commande.
- On considère que pour interagir avec une commande en utilisant un terminal, on utilisera :
 - L'entrée standard (STDIN)
 - Associée au clavier, elle permet à l'utilisateur d'envoyer des données à la commande.
 - La sortie standard (STDOUT)
 - Associée à l'écran, elle offre le résultat nominal de l'exécution de la commande.
 - La sortie d'erreur (STDERR)
 - Associée également à l'écran, elle permet l'affiche des problématiques rencontrées lors de l'exécution de la commande.



Sortie standard & sortie d'erreur

- Pour envoyer des données à l'écran en utilisant ces deux sorties, on utilise la fonction `print()` de Python.
 - `print(*objects, sep=' ', end='n', file=sys.stdout, flush=False) // TODO`
- Cette fonction prend un nombre valeurs variable en premier paramètre (étoilé), les autres paramètres doivent donc être nommés.
 - Note : Ces notions sont évoquées dans le module concernant les fonctions.
- L'affichage se fait par défaut sur la sortie standard (`file=sys.stdout`).


```
langage = 'Python'
duree = 4
print("Bienvenue à la formation", langage, "de :", duree, "jours")
```

- Chaque valeur est séparée par le caractère exprimé par le paramètre sep (l'espace par défaut).
- Un saut de ligne est appliqué par défaut à la fin de l'affichage, comme indiqué dans la signature par le paramètre end.

Sortie Formatées

- Les chaînes de caractères formatées (f-strings) permettent d'inclure directement la valeur d'expressions Python dans des chaînes de caractères en les préfixant avec f ou F
 - L'expression à présenter devra être spécifiée entre {}
 - L'expression peut être suivie d'un spécificateur de format

```
langage = 'Python'
duree = 4
print(f"Bienvenue à la formation {langage} de {duree} jours")
print(f"Bienvenue à la formation {langage:10} de {duree:4} jours")
```

Entrée standard en Python

- La gestion de l'entrée standard en Python permet la saisie d'informations au clavier lors de l'exécution d'un programme.
- La fonction input(), intégrée au langage (builtin), permet cela.

```
langage = input ("entrer votre langage préféré: ")
print ( langage)
duree = input ("entrer la durée de la formation: ")
print ( duree)
```

- Il est important de noter que :
 - Le paramètre de la fonction input() est le «prompt» (message utilisateur) à afficher.
 - Le retour de la fonction est systématiquement une chaîne de caractères.
 - A convertir donc, éventuellement.

Le cas de la sortie d'erreur

- Pour envoyer des données sur la sortie d'erreur, il faut simplement modifier la valeur du paramètre file de la fonction print() !

```
Import sys
print ("ceci est un message d'erreur" , file=sys.stderr)
```

ceci est un message d'erreur|

- NOTE : L'affichage en rouge est provoqué par la console intégré d'un IDE, on l'occurrence ici : PyCharm.

Les structures de contrôle

- Les structures de contrôle permettent d'intervenir sur le déroulement d'une séquence d'instructions.
- L'exécution, habituellement séquentielle, pourra donc se trouver modifiée.
- Il existe essentiellement 2 types de structures de contrôle :
 - Les structures conditionnelles
 - Elles permettent de conditionner l'exécution d'une série d'instruction
 - Les structures itératives
 - Plus communément appelées «boucles», elles permettent d'itérer sur un ensemble de valeurs, ou pour un certain nombre de fois.

La structure conditionnelle

- Dans le langage Python, la structure conditionnelle s'articule autour des mots clés **if**, **elif** et **else**.
- **if** permet de définir une condition pour laquelle, les instructions du bloc suivant seront exécutées. **elif** permet de faire une ou plusieurs conditions alternatives, et enfin **else**, est utilisé si aucune de ces conditions n'est vérifiée.

Condition simple vérifiée

```
 valeur = 5
if valeur > 3 :
    valeur += 1
print ( valeur)
```

Condition simple non vérifiée

```
 valeur = 5
if valeur < 3 :
    valeur += 1
print ( valeur)
```

La structure conditionnelle : Conditions élaborées

Condition et son contraire

```
if valeur > 0 :
    print ( "nombre positif")
else :
    print ( "nombre négatif")
```

Condition et alternative

```

if valeur > 0:
    print("nombre positif")
elif valeur < 0:
    print("nombre négatif")
else :
    print ( "valeur à zéro")

```

L'instruction if else

Condition et son contraire

```

if valeur > 0 :
    print ( "nombre positif")
else :
    print ( "nombre négatif")

```

Condition et alternative

```

if valeur > 0:
    print("nombre positif")
elif valeur < 0:
    print("nombre négatif")
else :
    print ( "valeur à zéro")

```

L'instruction match case (3.10)

```

from random import randint

random_value = randint(0, 5)

match random_value:
    case 0:
        print("Zéro")
    case 1:
        print("Un")
    case 2:
        print("Deux")
    case 3:
        print("Trois")
    case 4:
        print("Quatre")
    case 5:
        print("Cinq")

```

La structure conditionnelle : Expression de la condition

- Python permet une expression très «mathématique» de la condition. Là où, classiquement on écrirait par habitude :

```
if valeur >= 0 and valeur <= 20 :  
    valeur += 1  
print ( valeur)
```

- Python permet la syntaxe suivante, plus claire et lisible :

```
if 0 <= valeur <= 20 :  
    valeur += 1  
print(valeur)
```

Les structures itératives

- Il existe deux structures itératives en Python :
 - `while`
 - On itère tant qu'une condition est vérifiée
 - `for`
 - On parcourt un ensemble d'éléments ou de valeurs
- Contrairement à d'autres langages, pas de structures de type `until` ou bien `do ... while`

La structure itérative `while`

- Elle permet d'exécuter une suite d'instructions tant qu'une condition est vérifiée :

```
i = 0  
while i < 5 :  
    print ( i )  
    i+=1
```

- Attention de bien prendre soin à faire évoluer l'expression de la condition !
 - Sinon -> Boucle infinie !

La structure itérative `for`

- La boucle `for` permet de faire des itérations sur un élément.
 - Une chaîne de caractères, un ensemble, une liste, un dictionnaire, ...

```
langage = "Python"  
for lettre in langage:  
    print ( lettre )
```

```
liste = ["un", "deux", "trois", "quatre"]  
for elem in liste:  
    print (elem)
```

Gestion d'intervalle (range)

- La fonction range() permet de générer des suites arithmétiques :

```
range ( start , stop , step )
```

```
for elem in range (5):  
    print ( elem)  
  
for elem in range ( 2 , 6 , 2):  
    print ( elem)
```

Les ruptures de séquence

- Dans certaines situations, il peut être pertinent d'interrompre complètement une boucle ou bien l'itération en cours.
- Python dispose des mots clés break et continue pour cela.

Interruption de boucle

```
liste = ["un", "deux", "trois" , "quatre"]  
for elem in liste:  
    if elem == "trois" :  
        break  
    print (elem)  
print("sortie")
```

Reprise de boucle

```
for elem in liste:  
    if elem == "deux" :  
        continue  
    print (elem)
```

La clause else dans une boucle

- Il est possible de spécifier une clause else dans une boucle
 - Elle permet de définir un bloc d'instructions qui sera exécuté à la fin seulement si la boucle s'est déroulée complètement sans être interrompue (par une instruction break)

```
taille = 7  
while i < taille :  
    print ( i )  
    i+=1  
    if i == 13 :  
        break  
else :  
    print ( "Traitement iteratif du while complètement réalisé ")
```

Les fonctions

Utilité des fonctions

- En programmation, les fonctions permettent de factoriser un ensemble d'instructions.
 - L'objectif étant la réutilisation !
- Cette séquence d'instruction est le plus souvent configurable par des paramètres d'entrées.
- La fonction peut produire ou non un résultat final qui sera collecté par le code qui appelle la fonction.



Déclaration d'une fonction

- La déclaration d'une fonction est constituée des éléments suivants :
 - Le mot clé `def`
 - L'identificateur de la fonction (son nom !)
 - Une paire de parenthèses avec éventuellement des paramètres sous forme de variables.
- Ceci constitue la **signature** de la fonction.
- Un bloc est ensuite créé pour l'**implémentation** de la fonction (son code interne)

```
def rechercher_minimum ( valeur1 , valeur2 ):
    res = 0
    if valeur1 <= valeur2 :
        res = valeur1
    else:
        res = valeur2
    return res
```

Utilisation d'une fonction

- Pour utiliser une fonction, il suffit de l'appeler par son identificateur tout en spécifiant des valeurs pour les paramètres.
- Le résultat produit en sortie peut être associé à une variable, ou exploité par une autre fonction, ...

```
resultat = rechercher_minimum ( 2 , 5 )
```

- La portée d'une fonction est le module (fichier Python) qui la contient.
 - Donc si une fonction est contenue dans un module différent de celui où elle est utilisée, le module de la fonction doit être importé ! (Notion vue plus loin)

Les paramètres

- Lorsqu'une fonction déclare des paramètres dans sa signature, ils doivent recevoir des valeurs au moment de l'appelle de la fonction !

```
resultat = rechercher_minimum ( 2 , 5 )

def rechercher_minimum ( valeur1 , valeur2):
    res = 0
    if valeur1 <= valeur2 :
        res = valeur1
    else:
        res = valeur2
    return res
```

- Les paramètres correspondent à des variables locales à la fonction.
 - Ces variables ne sont visibles et utilisables qu'à l'intérieur du corps (le bloc) de la fonction.

Le retour d'une fonction

- Le retour d'une fonction est le résultat qu'elle produit à la fin de l'exécution de son code d'implémentation.
 - Certaines fonctions n'ont pas de retour.
- Ce retour met fin à l'exécution de la fonction.
 - Les instructions situées après cette instruction de retour ne seront donc pas exécutées !
- Le retour de la fonction est introduit par le mot clé `return`.

```
def rechercher_minimum ( valeur1 , valeur2):
    res = 0
    if valeur1 <= valeur2 :
        res = valeur1
    else:
        res = valeur2
    return res
```

Les annotations

- Les annotations sont une nouveauté Python qui vont permettre d'ajouter des métadonnées aux fonctions :
 - spécifier le type des paramètres attendus,
 - spécifier le type de la valeur de retour

```
def rechercher_minimum ( val1 : int, val2 : int ) -> int :
    res = 0
    if val1 < val2 :
        res = val1
    else:
        res = val2
    return res
```

- Elles ne sont pas destinées à vérifier au *runtime* le type des paramètres :
 - Améliore la lisibilité du code
 - Utiliser par les outils et pour la documentation

```
def test_annotation ():
    resultat = rechercher_minimum ( 3 , 5 )
    print ( resultat)
    resultat2 = rechercher_minimum( "3" , "5")
    print ( resultat2)
    resultat3 = rechercher_minimum("3", 5)
    print(resultat3)
```

Les paramètres optionnels

- En Python, il est possible de définir des fonctions possédant des paramètres optionnels.
 - Il n'est donc pas obligatoire de leur attribuer de valeurs au moment de l'appel de la fonction.
- Le principe consiste à attribuer une valeur par défaut à ces paramètres.
 - Si une valeur est donnée au moment de l'appel, alors elle est utilisée, sinon, c'est la valeur par défaut.
 - les paramètres optionnels doivent se situer en fin de signature !

```
def ma_fonction ( param1 : int , param2 : int = 0 , param3 : int = 1):
    resultat = param1 + param2 + param3
    return resultat

somme = ma_fonction ( 10 , 20 , 30 )
somme = ma_fonction (10)
somme = ma_fonction ( 10 , 20)
#somme = ma_fonction () #erreur
```

Les paramètres nommés

- Il est possible de nommer explicitement le paramètre que l'on veut renseigner :
 - Permet d'éviter le respect de l'ordre des paramètres notamment quand on a des paramètres optionnels :

```
def ma_fonction ( param1 : int , param2 : int = 0 , param3 : int = 1):
    resultat = param1 + param2 + param3
    return resultat

somme = ma_fonction ( 10 , param3 = 7 )
```

Les paramètres en nombre variable

- Dans la signature d'une fonction, il est possible d'exprimer que celle-ci peut recevoir un nombre variable de paramètres.

- Le *packing* et l'*unpacking* sont deux concepts Python complémentaires qui offrent la possibilité de transformer une succession d'arguments, nommés ou non, en collections (liste/tuple ou dictionnaire et vice-versa).
 - Un seul paramètre est alors exprimé.
 - Son nom est préfixé d'une `*`.
 - Le paramètre est alors exploitable sous forme de la collection.
- Dans notre cas la suite de valeurs est transformée en type Tuple

```
def calcul_moyenne ( *valeurs):
    return sum(valeurs) / len (valeurs)

moyenne1 = calcul_moyenne ( 2 , 4 , 6 , 8 , 10 )
moyenne2 = calcul_moyenne ( 1 , 3 , 5 )
```

La fonction print

- Certaines fonctions, notamment des fonctions intégrées au langage, utilisent des paramètres en nombre variable et des paramètres optionnels à la suite.
 - `print(*objects, sep = ' ', end='\n', file=sys.stdout , flush=False)`

```
print ("un" , "deux" , "trois")
```

- Comment exprimer une valeur pour *end* ou pour *sep* ???
 - La solution consiste à nommer les paramètres au moment de l'appel de la fonction !

```
print ("un" , "deux", "trois" , sep="#" , end=" fin")
```

Les paramètres nommés en nombre variable

- Il est possible de spécifier un nombre de paramètres variables qui sont nommés
 - Un seul paramètre est alors exprimé.
 - Son nom est préfixé de deux `**`.
 - Le paramètre est alors exploitable sous forme d'un dictionnaire.

```
def calcul_moyennev2 ( **valeurs):
    resultat = 0
    for key, value in valeurs.items():
        resultat += value
    return resultat / len ( valeurs )

moyenne1 = calcul_moyennev2( val1=2, val2=4)
moyenne2 = calcul_moyennev2( val1=2, val2=4, val3=6)
```

Valeur de retour multiple

- Un problème très fréquemment rencontré est l'absence de mécanisme pour renvoyer plusieurs valeurs depuis une fonction.
- Il y a bien sûr des solutions de contournement, mais elles sont généralement peu pratiques:
 - Utiliser un tableau ou une liste;
 - Utiliser la classe Tuple;
 - Créer une classe spécifique pour représenter les résultats de la fonction.
- Il est possible de renvoyer plusieurs valeurs dans une fonction. Le langage stocke les différentes valeurs dans une variable de type Tuple

```
def calcul_somme ( values):  
    nbr_valeur = 0  
    somme = 0  
    for val in values:  
        nbr_valeur += 1  
        somme += val  
    return nbr_valeur , somme
```

```
nbr , somme = calcul_somme ( range ( 5))  
print ( nbr , " " , somme )  
  
resultat = calcul_somme ( range ( 5))  
print ( resultat[0] , resultat[1])
```

L'instruction `pass`

- L'instruction « `pass` » est une instruction qui ne fait rien :
 - Elle peut être utilisée lorsqu'une instruction est nécessaire pour fournir une syntaxe correcte, mais qu'aucune action ne doit être effectuée.
 - Elle permet donc de définir la structure de vos programmes sans l'implémentation finale et ceux sans générer d'erreur.

```
def foo ():  
    pass
```

- Cette instruction est applicable à tout type de bloc :

```
if val == 0:  
    pass
```

Les modules

Utilité de la structuration en module

- Les applications complexes doivent faire preuve d'un découpage structurel en divers fichiers pour :
 - Améliorer la lisibilité
 - Favoriser l'évolution des fonctionnalités
 - Faciliter le travail en équipe
 - Réutiliser facilement des fonctionnalités entre plusieurs applications
 - Livrer et re-livrer simplement quelques parties d'applications
- En Python, les **modules** sont ces unités de structuration.

Conception de modules

- Un module peut contenir:
 - des fonctions,
 - des instructions exécutables,
 - des déclarations de variables
- Ces instructions sont destinées à initialiser le module. On les exécute seulement la *première* fois que le module est importé quelque part.

```
#module mmath
print ( "*** Chargement du module math ***")

def rechercher_minimum ( valeur1: int , valeur2 : int ) -> int :
    res = 0
    if valeur1 <= valeur2 :
        res = valeur1
    else:
        res = valeur2
    return res
```

Importation d'un module

- Lorsque des fonctionnalités présentes dans des modules doivent être utilisées, il y a trois manières de les rendre disponibles et accessibles pour les autres modules :
 - Rendre visible un module
 - Créer un alias sur le module
 - Importer une ou plusieurs fonctionnalités
- L'instruction import permet de rendre visible un module :
 - L'instruction place le nom du module dans la table des symboles de l'appelant. Les objets définis dans le module restent dans la table de symboles privée du module.
 - Pour accéder aux objets du module il faut donc les préfixer.

```
#module main
import mmath

if __name__ == '__main__':
    res = mmath.rechercher_minimum ( 10 , 20 )
```

Création d'un alias

- L'utilisation d'une fonctionnalité d'un module nécessite d'importer le module à travers l'instruction « import » :
 - L'instruction place le nom du module dans la table des symboles de l'appelant. Les objets définis dans le module restent dans la table de symboles privée du module.
 - les objets du module ne sont accessibles que lorsqu'ils sont préfixés

```
import mmath as m

if __name__ == '__main__':
    res = m.rechercher_minimum ( 10 , 20 )
    print ( res )
```

Importation de fonctionnalités de modules

- Il est également possible d'importer une ou plusieurs fonctionnalités d'un module à l'aide de l'instruction « from » .
 - La fonctionnalité se trouve donc directement disponible et visible dans le module important la fonctionnalité.
 - La syntaxe est plus concise.
 - Attention cependant aux conflits sur les noms.

```
from mmath import rechercher_minimum

if __name__ == '__main__':
    res = rechercher_minimum ( 10 , 20 )
    print ( res )
```

- Le caractère * permet d'indiquer que l'on veut importer toutes les fonctionnalités du module.
 - Cette technique est déconseillée car elle ne permet pas d'avoir une vision claire des fonctions qui ont été importées.
- Une fois importés, tous les modules possèdent cinq attributs qui contiennent des informations :
 - `__all__` : contient toutes les variables, fonctions, classes du module
 - `__builtins__` : dictionnaire qui contient toutes les fonctions et classes inhérentes au langage utilisé par le module
 - `__doc__` : contient l'aide associé au module
 - `__file__` : contient le nom du fichier qui définit le module

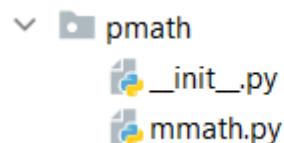
- `__name__` : contient le nom du module sauf si le module est le point d'entrée du programme auquel cas cette variable contient le « `main` ».

```
print (pmath.mmhath.__name__)  
print ( pmath.mmhath.__file__ )  
print ( dir ( pmath ) )
```

- La commande `dir` donne une liste d'attributs valides pour le module spécifié.

Organisation en packages

- Les modules peuvent se trouver stockés dans une arborescence de **packages**:
 - L'objectif est d'assurer l'unicité des noms des modules !
- Un package est un dossier, ni plus, ni moins.
 - Il doit contenir un fichier spécial `__init__.py` qui permet à Python de reconnaître le répertoire comme package.
 - Ce fichier est généralement vide



Accès à un package

- Les packages font partis du nom du module.
- L'accès aux fonctions se feront en mentionnant le nom complet.

```
import pmath.mmhath  
res = pmath.mmhath.rechercher_minimum ( 10 , 20 )  
  
from pmath.mmhath import rechercher_minimum  
res = rechercher_minimum ( 10 , 20 )
```

Le fichier `__init__`

- Il permet à Python de distinguer des répertoires des packages.
 - Ce fichier est généralement vide
- Il permet de spécifier les fonctionnalités du package que l'on veut exporter

```
#fic  
from pmath.mmhath import rechercher_minimum
```

Le chemin de localisation des modules : `PYTHONPATH`

- Le **PYTHONPATH** permet d'indiquer à Python quels dossiers il doit prendre en compte pour sa recherche de modules.
 - Il est visualisable de la manière suivante :

```
import sys
print ( sys.path)
```

- Le **PYTHONPATH** inclue toujours le répertoire courant depuis lequel les commandes sont passées.
 - Il inclut également la bibliothèque standard Python !
- Certains programmes nécessitent l'ajout de répertoires supplémentaires au **PYTHONPATH**.

Ajouter un répertoire au PYTHONPATH

- Il existe deux techniques pour ajouter un répertoire au PYTHONPATH :
 - Avec une variable d'environnement.
 - De manière permanente sur le système d'exploitation, ou bien en la valorisant temporairement via un script système ou avant d'exécuter un programme.
 - UNIX
 - `export PYTHONPATH=$PYTHONPATH:<nouveau répertoire>:<...>`
 - WINDOWS
 - `set PYTHONPATH=%PYTHONPATH%;<nouveau répertoire>;<...>`
 - Dynamiquement dans un programme Python, par programmation.

```
import sys
print ( sys.path.insert ( 0 , "/home/scripts" ))
```

Les modules Python

- Python fournit une bibliothèque standard qui contient un ensemble de modules proposant des fonctions prédéfinies:
 - `math` : traitement mathématique
 - `date` : traitement de date
 - `re` : expression régulière
- Il existe d'autres bibliothèques que l'on doit installer explicitement
 - `numpy` : calcul scientifique. Elle introduit une gestion facilitée des tableaux.

Les classes

Les classes

- Une classe est une entité qui regroupe des attributs et des méthodes

```

class NomClasse :

    #attributs
    def __init__(self) :
        self.__nomAttribut1 = valeurInitiale
        self.__nomAttribut2 = valeurInitiale

    #méthodes
    def nom_fonction(self, paramètre1, paramètre2) :
        # code de la fonction

```

NomClasse
- nomAttribut1 : type - nomAttribute2 : type
+ nomMethode1(arg1 : type) : type + nomMethode2(arg1 : type) : type

Attributs et méthodes

- Membres d'instance
 - Propres à chaque objet !
 - Pas d'ordre dans les déclarations.
- Attributs
 - Définit dans la méthode init avec le mot clé `self`
 - Peut-être un type simple ou complexe
- Méthodes
 - L'équivalent d'une fonction ou d'une procédure dans un contexte objet
 - Une référence à l'objet en cours est obligatoire !
 - Le premier paramètre d'une méthode.
 - Le plus souvent nommé « `self` »
 - Permet l'accès aux membres

Exemple

```

class Voiture :

    def __init__(self):
        self.__kilometrage = 0
        self.__carburant = 20
        self.__immatriculation = "ABC-ZE-44"
        self.__consommation = 5

    def rouler ( self , distance : int ):
        self.__kilometrage = self.__kilometrage + distance
        self.__carburant = (self.__carburant) - \
            (distance * self.__consommation / 100 )

    #méthode qui va convertir un objet en chaine

```

```
def __str__ ( self) -> str :
    return f"{self.__immatriculation} {self.__kilometrage} " \
           f"{self.__carburant} {self.__consommation} "
```

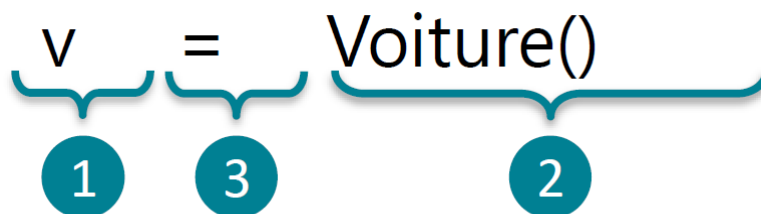
Voiture

immatriculation : str
kilométrage : float
carburant : float
consommation : float

rouler(distance : float)

L'instanciation

- Une classe est un type à partir duquel on va créer des objets.
- La création d'un objet à partir d'une classe est appelée une instanciation. Elle se réalise sous la forme suivante :



1. Création d'une "variable de manipulation d'une Voiture". Pas de création de Voiture. Valorisée à None
2. Création d'une nouvelle instance de `voiture`. Cela se traduit par une réservation de mémoire à une adresse (par ex: @100)
3. Affectation de la Voiture nouvellement créée à la variable `v1` par recopie de l'adresse

Manipulation d'un objet

- Toute manipulation d'objet s'effectue par "envoi de message".
- L'envoi d'un message est composé de trois parties :
 - une référence permettant de désigner l'objet récepteur,
 - le nom de la méthode à exécuter,

- les paramètres éventuels de la méthode
- La mise en oeuvre se réalise avec l'opérateur "."
 - s'applique à la référence sur l'objet
 - n'est possible que si la référence n'est pas nulle
 - Envoi d'un message à un objet destinataire et en réponse, l'objet déclenche un comportement

```
def test_voiture() :
    v = Voiture()
    v.rouler(20)
    print ( v )
```

L'encapsulation

L'objet doit se protéger

- Expression de la visibilité des membres
 - Privés : invisible à l'extérieur de la classe, même en lecture
 - Publics : manipulable de l'extérieur de la classe, même en écriture
- Démarche : sauf bonne raison, on cache les données qui ne doivent être accessibles que par les méthodes.
- En Python
 - Tout est considéré **public par défaut** !
 - La notion d'éléments privés peut être exprimée en préfixant les membres avec `__` (double underscore)

Les propriétés

- Un objet n'est manipulable que par les méthodes spécifiées dans sa classe.
- Toute tentative de manipulation de la partie privée entraîne une erreur.

```
def test_voiture() :
    v = Voiture()
    v.rouler(20)
    print ( v.__carburant ) #erreur
```

- Pour accéder aux données il faut donc prévoir des méthodes d'accès aux données : les propriétés .
 - Elles peuvent être en lecture et/ou en écriture et doivent contrôler l'intégrité des données.

Les propriétés en lecture

- La réalisation d'une propriété se réalise à l'aide d'un décorateur que l'on va appliquer sur la méthode :

```
class Voiture :
    def __init__(self) :
        self.__carburant = 20

    @property
    def carburant(self) -> int :
        return self.__carburant
```

```
def test_voiture() :
    v = Voiture()
    v.rouler(20)
    print ( v.carburant )
```

Les propriétés en écriture

- Les propriétés en écriture se réalise à l'aide d'un «setter».

```
@carburant.setter
def carburant(self, carburant : int) :
    if carburant <= 50:
        self.__carburant = carburant
```

```
def test_voiture() :
    v = Voiture()
    v.carburant = 20
    v.rouler(20)
    print ( v)
```

Initialisation des attributs d'une classe

- Il est souvent nécessaire d'initialiser les attributs pour donner un sens aux objets.
 - Dans ce cas il faut prévoir des paramètres dans la méthode `__init__`

```
def __init__(self, immat : str , kilo : float ,
             carbu : float , conso : float ) :
    self.__immatriculation = immat
    self.__kilometrage = kilo
    self.__carburant = carbu
    self.__consommation = conso
```

- La méthode `__init__` est appelé implicitement lors de la création
- La création d'objet doit donc respecter le mode de création imposé par la classe en fournissant les valeurs attendues.

```
def test_voiture() :
    v1 = Voiture("ABC-ZE-44" , 20000 , 10 , 0.7 )
    print(v1)
```

Utilisation des valeurs par défaut

- Il est possible de fournir des valeurs par défaut à la méthode `__init__`

```
def __init__(self, immatriculation : str , kilometrage : int ,
             carburant : int = 0 , consommation : int = 5 ):
    self.__immatriculation = immatriculation
    self.__kilometrage = kilometrage
    self.__carburant = carburant
    self.__consommation = consommation
```

```
def test_voiture() :
    v1 = Voiture("ABC-ZE-44" , 20000 , 10 , 0.7)
    v2 = Voiture("DEF-YB-35" , 500)
    print(v1)
    print(v2)
```

Le garbage collector

- Une fois initialisé par la «PVM», le "garbage collector" alloue un segment de mémoire pour stocker et gérer des objets.
- Cette mémoire est appelée tas managé, par opposition à un tas natif dans le système d'exploitation.
- Tout objet est créé en spécifiant le nom de la classe à instancier
- Un objet conserve son emplacement mémoire tant qu'il est référencé.
- La récupération de la mémoire (libération) est assurée par le garbage collector (ramasse miettes).
 - C'est un thread qui s'exécute au sein de la PVM et qui libère la mémoire quand celle-ci n'est plus utilisée.
- Son fonctionnement est basé sur deux algorithmes
 - Comptage de références
 - Un algorithme de détection de cycle (références circulaires)
- Le passage du «garbage» est réalisé en fonction d'un seuil d'allocations d'objets et de «désallocations» d'objets.
 - Lorsque le nombre d'allocations moins le nombre de désallocations est supérieur au nombre de seuil, le garbage collector est exécuté.
- On peut inspecter le seuil à l'aide de la méthode `get_treshold` de la classe `gc`

```
print("Garbage collection thresholds:", gc.get_threshold())
```

- Le seuil par défaut sur le système est de 700.
 - Cela signifie que lorsque le nombre d'allocations par rapport au nombre de désallocations est supérieur à 700, le ramasse-miettes s'exécutera.
- Le GC classe les objets en trois générations en fonction du nombre de balayages de collection auxquels ils ont survécu.
 - Les nouveaux objets sont placés dans la plus jeune génération (génération 0).

- Si un objet survit à une collection, il est déplacé dans la prochaine génération plus ancienne.
- La collecte démarre. Au départ, seule la génération 0 est examinée.
- Si la génération 0 a été examinée plus d'un seuil 1 fois depuis la génération 1 a été examinée, puis la génération 1 est également examinée. De même, threshold2 contrôle le nombre de collections
- Il est possible d'invoquer explicitement le garbage collector à l'aide de la méthode collect :

```
collect = gc.collect()
```

- Attention, lorsque le garbage collector s'exécute, tous les threads en cours d'exécution sont interrompus.
- La classe GC propose d'autres méthodes :
 - `gc.stats()` Renvoie une liste de trois dictionnaires par génération contenant des statistiques de collection depuis le démarrage de l'interpréteur.
 - `collections` est le nombre de fois où cette génération a été collectée;
 - `collected` est le nombre total d'objets collectés au sein de cette génération;
 - un `collectable` est le nombre total d'objets qui se sont avérés irrécupérables (et ont donc été déplacés vers la garbage liste) au sein de cette génération.

Le destructeur

- Il est possible de prévoir un traitement avant la libération des objets : le DESTRUCTEUR.
- Le destructeur est appelé par le garbage collector juste avant la destruction effective.
- Un destructeur est un représenté par la méthode `__del__`

Suppression d'une référence

- Il est également possible de détruire un objet avant la sortie de sa portée avec la méthode del :
 - `del obj`

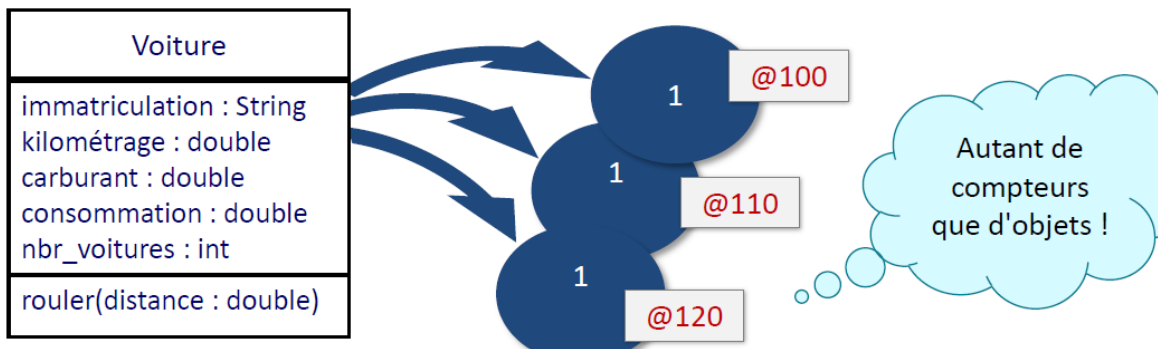
```
def test_voiture ():
    v1 = voiture("ABC-ZE-44", 20000, 10, 0.7 )
    print (v1)
    del v1
    print ( v1 )
```

UnboundLocalError: local variable 'v1' referenced before assignment

- Attention la méthode del détruit uniquement la référence (pas l'objet)

Membres de classe

- Question : comment mettre en place un système de comptage de voitures ?
- Quel problème rencontre-t-on ?



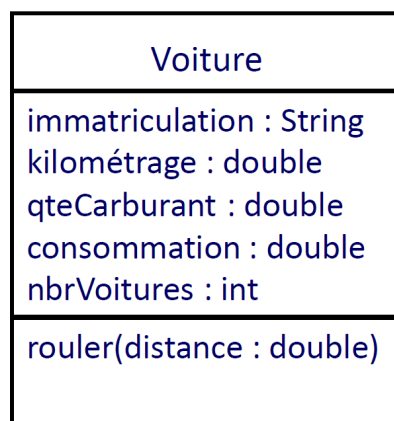
```
class Voiture :
    def __init__(self, immatriculation : str , kilometrage : int ,
                  carburant : int , consommation : int ):
        self.__immatriculation = immatriculation
        self.__kilometrage = kilometrage
        self.__carburant = carburant
        self.__consommation = consommation
        self.__nbr_voitures += 1
```

Parfois des données ne relèvent pas de l'instance mais de la classe

- Membre de classe ≠ membre d'instance
 - Attributs de classe
 - Méthodes de classe
- Les méthodes d'instance peuvent accéder aux membres de classe
- Les méthodes de classe ne peuvent pas accéder aux membres d'instance
- La déclaration d'une variable de classe se fait au sein de la classe (en dehors de toutes méthodes).
- La manipulation d'une variable de classe se réalise à travers le nom de la classe :

`NomClasse.__attribut`

- Il faudra prévoir des méthodes de classes pour y accéder :
 - On utilise le décorateur @classmethod sur les méthodes pour indiquer que ce sont des méthodes de classe
 - Un décorateur est une méta-données positionnée sur un élément de code.
 - Les décorateurs sont toujours préfixés par le caractère @



```
class Voiture :
```

```

__nbr_voitures = 0

@classmethod
def get_nbr_voitures(cls):
    return cls.__nbr_voitures

def __init__(self, immatriculation : str, kilometrage : int,
              carburant: int =0, consommation : int =5):
    self.__immatriculation = immatriculation
    self.__kilometrage = kilometrage
    self.__carburant = carburant
    self.__consommation = consommation
    Voiture.__nbr_voitures += 1

```

- Accès
 - Les membres sont accédés à partir du nom de la classe et non plus à partir du nom de l'instance.

```

def test_voiture() :
    v1 = Voiture("ABC-ZE-44" , 20000 , 10 , 0.7)
    v2 = Voiture("DEF-YB-35", 500)
    print(v1)
    print(v2)
    print(f" nombre de voiture : {Voiture.get_nbr_voitures()}")

```

L'héritage

Héritage et classe dérivée

- L'héritage permet de créer une classe, par extension, ou spécialisation d'une classe existante.
- Chaque niveau (classe) amène un complément ou une spécialisation des caractéristiques du niveau supérieur auquel il est rattaché.
- Dès lors qu'une caractéristique (attribut ou méthode) est définie à un niveau, elle reste valable pour tous les sous-niveaux affiliés.
- Ce mécanisme est appelé l'héritage. Si une classe B hérite d'une classe A, la classe B est appelée classe dérivée et A classe de base.

Exemple

```

class Personne (object):
    def __init__ ( self, nom : str , age : int ):
        self.__nom = nom
        self.__age = age

    @property
    def nom ( self) -> str :
        return self.__nom

    @property def age(self) -> int :
        return self.__age

    def anniversaire (self):

```

```

self.__age = self.__age + 1

def __str__(self) -> str :
    return f"nom: {self.__nom} age: {self.__age}"

```

Héritage

- On veut créer une classe Stagiaire.
- Un Stagiaire est une personne qui suit un cours.
- La mise en oeuvre de l'héritage se réalise au moment de la conception de la nouvelle classe en spécifiant la classe dont on veut hériter :

```

class Stagiaire(Personne):
    pass

```

- La classe Stagiaire hérite de Personne.
- Elle possède donc les attributs et les méthodes de la classe Personne.

La classe Stagiaire

- La classe Stagiaire complète la classe Personne en intégrant la notion de cours :

```

class Stagiaire(Personne):
    def __init__(self, cours : str ):
        self.__cours = cours

    @property
    def cours(self) -> str :
        return self.__cours

    @cours.setter
    def cours(self, cours : str ):
        self.__cours = cours

```

Les constructeurs

- Un constructeur d'une classe dérivée doit toujours invoquer un constructeur de la classe de base
 - Contrairement à ce qui se passe dans certains langages orientés objet, la délégation n'est pas implicite en Python. Il faut y procéder explicitement :
- Il existe deux possibilités pour appeler le constructeur de la classe de base:
 - `SuperClasse.__init__(self, ...)` #Python2
 - `super().__init__(...)` #Python3

```

def __init__(self, nom : str, age : int, cours : str ):
    super().__init__( nom, age ) #Python 3
    self.__cours = cours

```

Héritage

- La création d'un objet s'effectue à travers le nom de la classe

```
stag1 = Stagiaire ( "Dupont" , 40 , "Python")
```

- Un objet stagiaire peut solliciter les méthodes définies dans sa classe :

```
stag1.cours="Java"
```

- Un objet stagiaire peut solliciter les méthodes définies dans sa hiérarchie (méthodes de la classe Personne) :

```
stag1.anniversaire()
```

Redéfinition de méthode

- Que se passe-t-il lorsque l'on tente d'afficher un objet Stagiaire :

```
stag1 = Stagiaire ( "Dupont" , 40 , "Python")  
print(stag1)
```

- C'est la méthode «str» de la classe Personne qui est appelée. Le cours n'est donc pas affiché.
- On peut donc considérer que cette méthode ne remplit pas son rôle pour un objet stagiaire :
 - il faut donc la redéfinir.
- Lorsque l'on redéfinit une méthode, elle vient masquer celle du niveau supérieur qui devient donc inaccessible.

```
def __str__(self) -> str :  
    return super().__str__() + f" Cours: {self.__cours}"
```

La classe dérivée "Stagiaire"

- Les caractéristiques héritées sont tributaires des attributs d'accès des champs de la classe de base.
- Les champs publics de la classe "Personne" sont bien-sûr accessibles pour les méthodes de la classe "Stagiaire" :

```
def __str__(self) -> str :  
    return super().__str__() + f" Cours: {self.__cours}"
```

- Les champs privés de la classe "Personne" sont inaccessibles pour les méthodes de la classe "Stagiaire" :

```
def __str__(self) -> str :  
    return f" Nom:{ self.__nom } Cours: {self.__cours}"
```


L'attribut d'accès `protected`

- L'héritage n'est pas un moyen de contourner l'encapsulation.
 - Les attributs privés d'une classe sont donc inaccessible pour la classe dérivée.
- Python met à disposition un attribut d'accès qui permet de concéder un droit d'accès à une classe dérivée : `protected`
 - Basé sur des règles de nommage
 - Doit commencer par un simple `_`
 - Simple indication que cet attribut ne doit être accessible que depuis une classe dérivée (pas de contrôle)

```
class Personne :  
    def __init__ ( self, nom : str , age : int ):  
        self.__nom = nom  
        self.__age = age
```

La classe `Object`

- Toute classe hérite implicitement de la classe `object`.
- L'ensemble des classes Python se présente donc sous la forme d'une hiérarchie unique.
- En Python, la classe `object` est la super-classe de toutes les classes.
 - `class Personne` : identique à `class Personne (object)` :
- Elle possède un ensemble de méthodes qui sont donc héritées par toutes les classes Python.

```
>>> dir(object)  
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',  
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',  
 '__sizeof__', '__str__', '__subclasshook__']
```

La méthode `str`

- La méthode `str` permet une conversion de l'objet en chaîne de caractères.
- Elle renvoie par défaut le nom de la classe et l'adresse de l'objet

```
def test_personne():  
    p1 = Personne ("Dupont" , 40 )  
    print(p1)
```

```
<__main__.Personne object at 0x000001E81DCE8400>
```

La méthode `str ()`

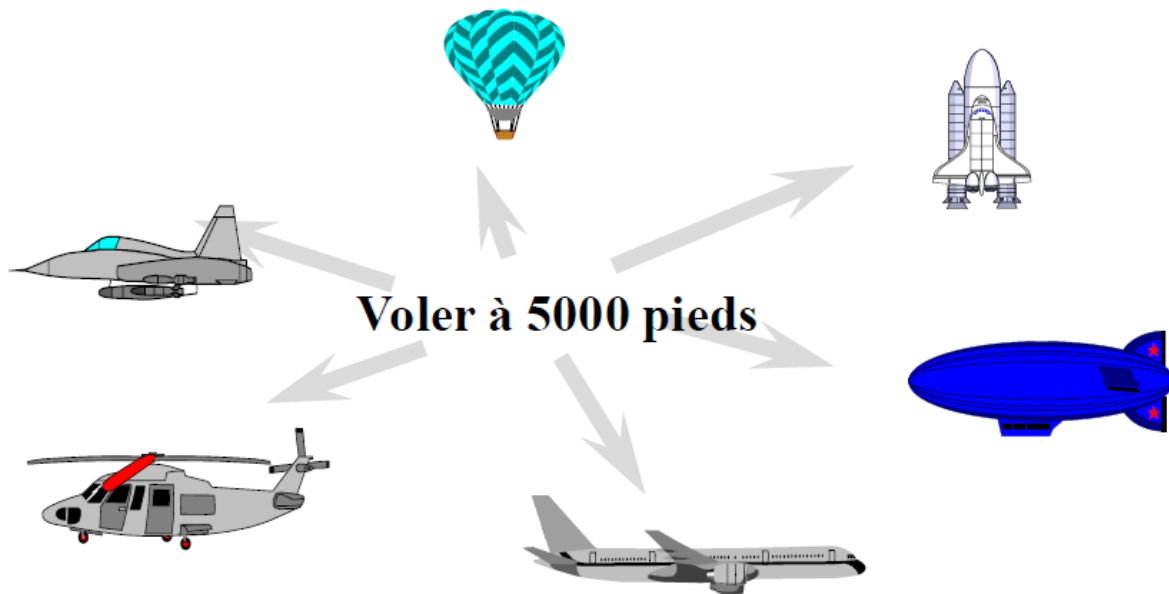
- Les classes métiers doivent redéfinir la méthode `str` pour assurer l'affichage des objets :

```
def test_personne():  
    p1 = Personne ("Dupont" , 40 )  
    print(p1)
```

```
class Personne:
    def __init__( self, nom : str , age : int ):
        self.__nom = nom
        self.__age = age
    def __str__(self) -> str :
        return f"nom: {self.__nom} age: {self.__age}"
```

Le polymorphisme

- Le polymorphisme est la faculté pour un même message de prendre des formes différentes :



Compatibilité d'instance

- La classe `stagiaire` peut être vu comme un "sous type" de la classe `Personne`
- `stag1` est une instance de la classe `stagiaire` mais `stag1` est aussi une instance de la classe `Personne`.
- Il est donc possible de regrouper des objets d'une même hiérarchie dans une collection.

```
tab_personnes = []
tab_personnes.append(Personne("Dupont", 40))
tab_personnes.append(Stagiaire("Martin", 45, "Python"))
for p in tab_personnes :
    print (p)
```

Gestion des exceptions

Les exceptions

- Besoin de gérer les erreurs

```
def diviser ( valeur1 : int , valeur2 : int ) -> float :
    res = valeur1 / valeur2
    return res
```

- Historiquement : code retour d'une fonction
 - Détournement du fondement mathématique de la syntaxe

```
CODE_ERREUR = -1

def diviser ( valeur1 : int , valeur2 : int ) -> float :
    if valeur2 != 0 :
        res = valeur1 / valeur2
        return res
    else :
        return CODE_ERREUR
```

- Besoin d'un autre canal de sortie : les erreurs
 - Séparation du flux de données et d'erreurs
 - La syntaxe d'appel redevient naturelle
 - Syntaxe appropriée en cas d'erreur
- En Python, les erreurs sont des objets :
 - Classe Exception
 - Il faut les créer
- 2 parties
 - La détection du cas d'erreur = émission de l'erreur
 - Le traitement de l'erreur

Les exceptions : différents types

- La bibliothèque standard Python contient un certain nombre de classes d'exception prédéfinies.
- Elles peuvent être utilisées si leur nom est suffisamment représentatif du type d'erreur à exprimer !
- Elles peuvent également servir de super-classe pour des exceptions utilisateur.
 - Dans ce cas, il est préconisé d'hériter de Exception ou de l'une de ses sous-classes.

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |   |   +-- BrokenPipeError
    |   |   +-- ConnectionAbortedError
    |   |   +-- ConnectionRefusedError
    |   |   +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
    |   +-- InterruptedError
    |   +-- IsADirectoryError
    |   +-- NotADirectoryError
    |   +-- PermissionError
    |   +-- ProcessLookupError
    |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
    |   +-- NotImplementedError
    |   +-- RecursionError
    +-- SyntaxError
    |   +-- IndentationError
    |   +-- TabError
    +-- SystemError
    +-- TypeError
    +-- ValueError
    |   +-- UnicodeError
    |   |   +-- UnicodeDecodeError
    |   |   +-- UnicodeEncodeError
    |   |   +-- UnicodeTranslateError
    +-- Warning
    |   +-- DeprecationWarning
    |   +-- PendingDeprecationWarning
    |   +-- RuntimeWarning
    |   +-- SyntaxWarning
    |   +-- UserWarning
    |   +-- FutureWarning
    |   +-- ImportWarning
    |   +-- UnicodeWarning
    |   +-- BytesWarning
    |   +-- ResourceWarning

```

Les exceptions : origine de l'erreur

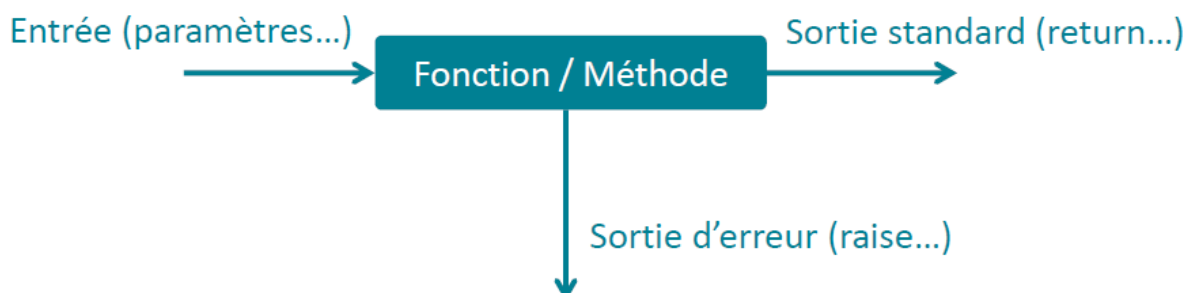
- La mise en oeuvre des exceptions s'effectue avec le mot clé `raise` :

```

def diviser ( valeur1 : int , valeur2 : int ) -> float :
    if valeur2 != 0 :
        res = valeur1 / valeur2
        return res
    else :
        raise Exception ( "division par zéro impossible " )

```

- La définition du déclenchement d'exception laisse «sortir» l'objet d'erreur par un autre canal de sortie.



Les exceptions : traitement de l'erreur

- Gérer les exceptions est presque obligatoire
 - Toute exception non gérée correctement conduira à la fin de l'exécution de l'interpréteur Python !

```
res = diviser ( 5 / 0 ) #ZeroDivisionError: division by zero
```

- La gestion des exceptions se réaliser à l'aide de l'instruction

```
try: ... except: ...
```

```
def test_exception () :  
    try:  
        res = diviser ( 5 / 0 )  
    except Exception as e:  
        print (e)
```

La structure `try: ... except: ...`

- L'idée consiste à écrire les instructions de code critique, celles qui peuvent déclencher une exception, dans un bloc `try`.
- En cas de déclenchement d'exception, le code situé dans le bloc `except` sera exécuté.



Les exceptions utilisateur

- Avoir ses propres classes d'exception
 - Exceptions techniques
 - Exceptions métiers
 - Exceptions de présentation

Hériter de Exception

- Ajouter des attributs et des méthodes
 - ex : code d'erreur
- Reprendre éventuellement le constructeur

Exemple

```
class SosException ( Exception) :  
    def __init__ ( self , message : str = "Bateau en detresse") :  
        super().__init__(message)  
  
class Voilier:  
    def manoeuvre(self) :  
        probleme = True  
        if probleme :  
            raise SosException ( "Erreur de manoeuvre ")
```

```
def test_exception () :
    v = voilier()
    try :
        v.manoevre()
        print ( "manoeuvre effectué on continue")
    except SosException as e :
        print ( e )
    print ("on continue la croisière")
```

Gestion de plusieurs erreurs

- Il est possible de faire suivre le bloc `try` de plusieurs blocs `except`.
 - Ils permettront de faire un traitement approprié en fonction du type d'exception déclenché.
 - Chaque bloc `except` doit alors mentionner le type d'exception qu'il prend en charge.
 - Un dernier bloc `except` «générique» permettra la gestion des erreurs non-prévues.

```
try:
    ...
except NameError as e:
    ...
except TypeError as e:
    ...
except:
```

Les exceptions : finally

- Parfois, il faut avoir la garantie d'exécuter du code même en cas d'exception
 - Exemple : libération de ressources (connexion, référence de fichier, ...)

Le bloc finally

```
ressource = UneRessource()
try:
    ressource.traiter()
except ExceptionTechnique as e:
    raise ExceptionFunctionnelle("message fonctionnel")
finally:
    ressource.close()
```

Les collections

Les collections

- Le langage propose plusieurs types de collections :
 - Les listes: collection de valeurs associées à un indice numérique ordonné à partir de zéro.
 - Les Tuples : collection non modifiable de valeurs associées à un indice numérique ordonné à partir de zéro.
 - Les dictionnaires : collection de valeurs associées à une clé, on maîtrise le type de la clé.

Les Listes

- Une liste est une structure de donnée qui contient un ensemble de valeurs :
 - mis en œuvre se réalise à l'aide de l'opérateur `[]` ou avec le type `list`

```
villes = list() #idem villes = []
villes.append ("Nantes")
villes.append ("Paris")
villes.append ("Lyon")
```

- Il est possible d'initialiser une liste lors de sa création.

```
villes = ["Paris", "Lyon", "Marseille", "Nantes", "Rennes"]
print ( villes )
```

Accès à un élément : les slices

- L'accès à un élément se réalise à travers un index :
 - le premier élément d'une liste se trouve à l'index 0
 - un index peut être négatif, il permet en partant de la fin
 - L'index peut être un intervalle de valeur

```
#accès à un élément d'une liste
print("un élément " , villes[0])
#accès à un élément à partir de la fin
print("un élément à partir de la fin ", villes[-1])
#accès à une suite d'éléments
print ("une suite d'élément : " , villes[0:2])
#Modification d'un element
villes[0] = "PARIS"
```

Parcours d'une liste

- Le parcours d'une liste se fait élément par élément en fonction de la taille de la liste :
- La fonction « `len` » permet de récupérer le nombre d'éléments d'une liste :

```
def affiche_ville1 () :
    for i in range (len (villes )) :
        print ( villes[i] )
```

Les itérateurs

- Un itérateur est un objet permettant de parcourir tout autre objet dit «itérable» :
 - C'est une sorte de curseur qui a pour mission de se déplacer dans une séquence d'objets.
 - Il permet de parcourir chaque objet d'une séquence sans se préoccuper de la structure sous-jacente.
- Les principaux objets itérables sont :

- Les collections
- Les chaîne de caractères
- Les fichiers
- Python possède une fonction faisant partie des Built-In (fonctions de base) nommée « `iter()` ».
- Cette fonction permet de créer un itérateur sur un objet itérable.
- Un itérateur Python doit posséder une méthode `__next__` qui ne prend pas d'argument et renvoie toujours l'élément suivant du flux.
- S'il n'y plus d'élément dans le flux, la méthode doit lever une exception de type `StopIteration`.

```
def affiche_ville2():
    itérateur=iter(villes)
    while True:
        try:
            ville = next(itérateur)
            print(ville)
        except StopIteration as stop:
            break
```

- Python fournit une structure itérative qui s'attend à travailler avec des itérateurs et qui va nous simplifier la syntaxe :
 - La structure for

```
def affiche_ville3 () :
    for ville in villes :
        print ( ville )
```

L'instruction `yield`

- Elle permet de créer des fonctions dont le résultat sera utilisé comme une séquence sans en être une :
 - peut être vu comme l'équivalent d'un return
 - fonction reprend ou elle s'était arrêtée jusqu'au prochain « `yield` »
- Objectif :
 - génération des suites de valeurs infinies
 - gain important en mémoire (peu de consommation)


```
import sympy

def nombres_premiers(borne_max):
    i = 1
    while i <= borne_max :
        if sympy.isprime (i) :
            yield i
        i = i + 1

for i in nombres_premiers ( 20 ) :
    print ( i )
```

Suppression d'éléments

- Il est possible de supprimer des éléments dans une liste :
 - Suppression à partir d'un index : fonction `del`
 - Suppression à partir d'une valeur : fonction `remove`

```
#suppression d'un élément
villes.remove("Marseille")
del villes[2]
print("apres suppression : ", villes)
```

Recherche d'éléments

- Il est possible de rechercher la présence d'un élément dans une liste
 - Recherche de la présence : opérateur `in`
 - Recherche de l'index : fonction `index`
 - Recherche du nombre d'occurrence : fonction `count`

```
v = "Rennes"
if v in villes :
    print ( f"{v} est une grande ville ")
    print (f"{v} se trouve à la position : {villes.index(v)}")
    print (f"nombre d'occurences de {v} dans la liste : {villes.count(v)}")
```

Réalisation de tri

- Python fournît une fonction native permettant de réaliser des tris :

```
villes = ["Paris", "Lyon", "Marseille", "Nantes", "Rennes"]
print (sorted ( villes))
print(villes)
```

- Il existe aussi une méthode `sort` que l'on peut appliquer sur les listes:
- La fonction « `sorted` » génère une nouvelle liste alors que la méthode `sort()` modifie la liste d'origine
- Il est possible de définir la façon de trier avec le paramètre `reverse` qui est un booléen pour déterminer si on veut un tri ascendant ou descendant

```
villes = ["Paris", "Lyon", "Marseille", "Nantes", "Rennes"]
villes.sort()
print ( villes )
```

- Comment faire pour trier sur un critère spécifique :
 - Il faut passer une fonction de comparaison à la méthode `sort`

```
def compare_ville ( ville : str ) :
    return ville [1]
def test_tri () :
    villes = ["Paris", "Nantes" , "Rennes" , "Lyon" , "Marseille"]
    villes.sort ( key = compare_ville)
    print (villes )
```

- Il est dommage de définir des méthodes justes pour le besoin d'un algorithme
 - Elles sont souvent à usage unique et donc peu réutilisable.
- La solution : Les lambdas

Les lambdas

- Une expression lambda est une fonction anonyme. Elle se comporte comme n'importe quelle autre fonction :
 - peut recevoir des arguments,
 - ne contient qu'une seule expression.
 - peut renvoyer une valeur
- Syntaxe :
 - `lambda argument(s): expression`

```
def test_lambda (valeur):
    res = lambda v: v % 2 == 0
    if ( res ( valeur )):
        print ( valeur , "est un nombre pair")
    else:
        print (valeur , "n'est pas un nombre pair")
```

Exemple

- On veut effectuer un tri sur le deuxième caractère des chaînes de caractères :
 - On va passer en paramètre le critère de tri sous forme de lambda :

```
def test_tri () :
    villes = ["Paris", "Nantes" , "Rennes" , "Lyon" , "Marseille"]
    villes.sort ( key = lambda i : i[1])
    print (villes )

print ("Test tri ")
test_tri ()
```

Les fonctions intégrées de Python

- Le langage python fournit un ensemble de fonctions qui exploite les expressions lambdas :
 - `filter` : pour filtrer les éléments d'une collection
 - `map` : pour modifier les éléments d'une collection

```
def test_fonction_lambda():
    liste_entiers = [1 , -2 , -3 , 4 , 5 ]
    liste_positif = list ( filter ( lambda x : x > 0 , liste_entiers ))
    print(liste_positif)
    listeInc = list ( map ( lambda x: x + 1 , liste_entiers ))
    print ( listeInc )
```

Opérations sur les listes

- Il est possible réaliser des traitements sur la liste dans sa totalité :
 - Comparaison : opérateur `==`
 - Concaténation : opérateur `+`

```
villes = ["Paris", "Lyon", "Marseille", "Nantes", "Rennes"]
liste = ["Paris", "Lyon", "Marseille", "Nantes", "Rennes"]
autres_villes = ["Angers" , "Brest"]
if liste == villes :
    print ( "les deux listes sont indentiques")
else :
    print ("les listes sont différentes")
villes += autres_villes
print ( villes )
v2 = autres_villes[:] #copie de listes
print ( v2 )
```

Les Listes

- On peut stocker des objets dans une liste.

```
class Carnet:
    def __init__(self):
        self.__liste_personnes = []
        self.__liste_personnes.append ( Personne ( "nom1" , 30))
        self.__liste_personnes.append ( Personne ( "nom2" , 50))
        self.__liste_personnes.append ( Personne ( "nom3" , 40))
```

Les listes de listes qui ne sont pas des matrices

- On peut stocker des listes dans une liste :

```
matrice = [ [0 , 0 ] , [1 , 1 ] , [ 2 , 2 ] ]
print ( matrice[0][0])
for t in matrice :
    for val in t :
        print ( val)
```

Opération sur les matrices

- On peut faire de l'affectation de matrice
 - `Affectation` : affectation de référence
 - `Shallow Copy` : duplication des références de la liste
 - `Deep Copy` : duplication des éléments de la liste

```
matrice1 = [ [0 , 0 ] , [1 , 1 ] , [ 2 , 2 ] ]
matrice2 = matrice1
matrice2[0][0] = 10
print ( matrice1 , " " , matrice2 )
matrice3 = copy.copy( matrice1)
matrice3[0][0] = 20
print ( matrice1 , " " , matrice3 )
matrice4 = copy.deepcopy ( matrice1)
matrice4[0][0] = 30
print ( matrice1 , " " , matrice4 )
```

Les dictionnaires

- Un dictionnaire permet de stocker des objets en les associant à une clé qui doit être unique.
- La mise en œuvre se réaliser avec le type `dict` ou en utilisant les symboles `{}`

```
def test_dic():
    un_dico = {"un":1 , "deux":2 , "trois":3 , "quatre":4 }
    autre_dico = dict()
    print ( type ( un_dico))
    print ( un_dico["trois"])
    print ( un_dico.get("trois"))
    un_dico["cinq"] = 5
    del un_dico["trois"]
    print(un_dico)
    for elem in un_dico :
        print ( elem)
    for cle , valeur in un_dico.items():
        print ( cle , "=", valeur )
```

La bibliothèque standard

Introduction

- La bibliothèque standard de Python est un élément incontournable de la technologie !
- En effet, c'est elle qui apporte les fonctionnalités à Python.
- Il est bien entendu essentiel de bien connaître le langage et sa syntaxe pour pouvoir utiliser ces fonctionnalités.
- La documentation officielle de Python, propose une référence exhaustive de ces fonctionnalités, agrémentée de nombreux exemples d'utilisation.
 - <https://docs.python.org/3/library/index.html>

La gestion des dates

- Python fournit plusieurs modules pour la gestion de date :
 - `time` : module qui fournit des fonctions et des constantes liées à la date et à l'heure.
 - `datetime` : module objet qui propose différentes classes pour la gestion de la date et de l'heure
 - `calendar` : module qui fournit des fonctions et des classes pour la gestion de calendrier.
 - `Locale` : module qui contient des fonctions qui sont utilisées pour le formatage ou l'analyse de la date et de l'heure en fonction des paramètres régionaux.

Le module `datetime`

- Le module « `datetime` » est conçu sur la base d'objet pour fonctionner avec la date et l'heure. Il contient un ensemble de classes :
 - `date` : représente une date selon le calendrier grégorien
 - `datetime` : représente une date et une heure
 - `time` : représente le temps
 - `tzinfo` : classe abstraite qui sert de la classe de base pour la classe timezone
 - `timezone` : représente le temps universel
 - `timedelta` : représente une durée à savoir le delta entre deux dates.

La classe `Date`

- Elle représente une date sous la forme `aaaa/mm/jj` :

```
def test_date():
    today = date.today()
    d = date(2020, 1, 1)
    annee = d.year
    mois = d.month
    jour = d.day
    print(f"{jour} {mois} {annee}")
    print(d.strftime("%d/%m/%y"))
    print(d.strftime("%A %d %B %Y"))
```

1 1 2020

01/01/20

Wednesday 01 January 2020

- La classe `datetime` représente une date et une heure et propose des méthodes similaires.
- Il est possible d'utiliser les opérateurs sur les dates:

```
def test_manipdate():
    today = date.today()
    d = date(2020, 1, 1)
    print ( today == d)
    print ( today > d)
    print ( today < d )
    duree = (today - d)
    print(type(duree) , " " , duree)
    d2 = timedelta(days=3)
    after = d + d2
    print(after)
```

False

True

False

<class 'datetime.timedelta'> 107 days, 0:00:00

2020-01-04

Les expressions régulières

- Les expressions régulières sont des schémas ou des motifs utilisés pour effectuer des recherches et des remplacements dans des chaînes de caractères.
- Ces schémas ou motifs sont tout simplement des séquences de caractères dont certains vont disposer de significations spéciales et qui vont nous servir de schéma de recherche.
- Les expressions régulières vont nous permettre de:
 - vérifier la présence de certains caractères ou suite de caractères dans une expression.
 - d'extraire des valeurs
 - de substituer des valeurs

Le module `re`

- Le module « `re` » a été spécialement conçu pour travailler avec les expressions régulières.
- Il définit plusieurs fonctions utiles ainsi que des objets propres pour modéliser des expressions.
- La fonction `match` permet de vérifier si une expression correspond à un motif:

```
def test_match():
    motif = r"^\d{2}$"
    chaine = "10"
    res = re.match(motif , chaine)
    print (res.group(0)) #renvoie la chaine correspondant à la re
    if res :
        print ("format numerique ok")
    else :
        print ("format incorrect")
```

Les expressions régulières

- Il existe un grand nombre d'expression :

Symbole	Signification
.	N'importe quel caractère
[xy]	Liste de valeurs possible
[x-y]	Intervalle de valeur possible
[x y]	Choix de valeurs (ou exclusif)
\d	Désigne tout chiffre équivalent à [0-9]
\D	Pas de chiffre équivalent à [^0-9]
\w	Présence alphanumérique idem [a-zA-Z0-9_]
\W	Pas de caractère alphanumérique
\s	Présence d'un espace [\t\n\r\f\v]
\S	Pas d'espace
^	Début ou contraire de
\$	Fin de segment

Les multiplicateurs

- Il permette de définir un nombre de fois ou l'expression est attendue:

Symbole	Signification
*	Présence entre 0 et n fois
+	Présence de 1 à n fois
?	Présence de 0 à 1 fois
{m, n}	Choix de valeurs (ou)
(?!(...))	Absence du groupe désigné par les points de suspension

L'instruction compile

- Une expression régulière produit en interne une machine à état:
 - Cette machine à état évolue au fur et à mesure du traitement de la chaîne de caractères
- Si on utilise une expression régulière fréquemment, il peut être intéressant d'optimiser le traitement en la compilant à l'aide de la méthode `compile`

```
def test_compile():
    regex = re.compile(r"^\d{2}$")
    chaine = "111"
    res = regex.match(chaine)
    if res:
        print('format numerique ok ')
    else:
        print('format incorrect')
```

Les fonctions du module re

- Le module `re` fournit un ensemble d'autres fonctionnalités:
 - `match(s[,pos[,end]])` : Vérifie la correspondance entre l'expression régulière et la chaîne. Il est possible de n'effectuer cette vérification qu'entre les caractères dont les positions sont `pos` et `end`. La fonction retourne `None` s'il n'y a pas de correspondance et sinon un objet de type `Match`.
 - `search(s[,pos[,end]])` idem `match`, au lieu de vérifier la correspondance entre toute la chaîne et l'expression régulière, cherche la première chaîne de caractères extraits correspondant à l'expression régulière.
 - `split(s[,pos[,end]])` : Recherche toutes les chaînes de caractères extraits qui vérifient l'expression régulière puis découpe cette chaîne en fonction des expressions trouvées.
 - `findall(s[,pos[,end]])` : recherche les chaînes correspondant aux motifs.

```
def test_findall():
    res = re.findall("[0-9]+", \
                    "cours Python le 10 janvier et le 17 janvier")
    print(res)  #['10', '17']
```

Interaction avec le système d'exploitation

- Python propose un module de bas niveau pour appréhender son système d'exploitation.
 - Le module `os`
- Les différentes fonctions et constantes présentes dans ces modules permettent de collecter des informations sur son système d'exploitation.


```
>>> import os
>>> dir(os)
['DirEntry', 'F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINHERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAIT', 'PathLike', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX', 'W_OK', 'X_OK', '_Environ', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_execvpe', '_exists', '_exit', '_fspath', '_get_exports_list', '_putenv', '_unsetenv', '_wrap_close', 'abc', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'close', 'closerange', 'cpu_count', 'curdir', 'defpath', 'device_encoding', 'devnull', 'dup', 'dup2', 'environ', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'extsep', 'fdopen', 'fsdecode', 'fsencode', 'fspath', 'fstat', 'fsync', 'ftruncate', 'get_exec_path', 'get_handle_inheritable', 'get_inheritable', 'get_terminal_size', 'getcwd', 'getcwdb', 'getenv', 'getlogin', 'getpid', 'getppid', 'isatty', 'kill', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'makedirs', 'mkdir', 'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'readlink', 'remove', 'removedirs', 'rename', 'renames', 'replace', 'rmdir', 'scandir', 'sep', 'set_handle_inheritable', 'set_inheritable', 'spawnl', 'spawnle', 'spawnv', 'spawnve', 'st', 'startfile', 'stat', 'stat_result', 'statvfs_result', 'strerror', 'supports_bytes_environ', 'supports_dir_fd', 'supports_effective_ids', 'supports_fd', 'supports_follow_symlink', 'symlink', 'sys', 'system', 'terminal_size', 'times', 'times_result', 'truncate', 'umask', 'uname_result', 'unlink', 'urandom', 'utime', 'waitpid', 'walk', 'write']
>>>
```

Collecter des informations sur le système

- Avec le module `os`, il est donc possible :
- D'identifier les caractéristiques de son système d'exploitation :
 - `os.name`
- De trouver la liste des variables d'environnement :
 - `list(os.environ.keys())`
- De récupérer le répertoire courant
 - `os.getcwd()`
- D'obtenir les valeurs de constantes du système :
 - `os.curdir`, `os.pardir`
 - `os.sep`, `os.pathsep`, `os.linesep`

Interagir avec les processus

- Python permet également, grâce aux modules `os` et `subprocess`, de lancer des commandes du système d'exploitation.
- Lancer une commande et afficher son résultat :

```
os.system("dir")
```

```
subprocess.call("dir /p", shell=True) version historique
```

```
subprocess.run("dir /p", shell=True) version 3.5
```

```
subprocess.Popen(["dir", "/p"], shell=True )
```

```
#plus riche en fonctionnalité
```

- La variable retour contiendra le code de retour de la commande (0 si tout va bien)
- Il faut spécifier l'argument `shell` à `true` pour les commandes internes
- Lancer une commande et récupérer son résultat :
 - `retour, resultat = subprocess.getstatusoutput(['ls', '-l'])`
 - La fonction renvoie un tuple. La variable retour contient le code de retour de la commande, la variable `resultat` contient le résultat produit par la commande sur la sortie standard
- On peut rediriger une commande vers une autre commande à l'aide de l'instruction `popen` :

```
def test_pipe():
    p1 = subprocess.Popen(["dir"], stdout=subprocess.PIPE, shell=True)
    p2 = subprocess.Popen(["findstr", 'py'], stdin=p1.stdout,
                          stdout=subprocess.PIPE, shell=True)

    p1.stdout.close()
    output, err = p2.communicate()
    print(output)
```

Travailler avec les chemins d'accès

- Le module `os` propose un ensemble de méthodes pour manipuler les fichiers à travers le chemin d'accès
 - `os.listdir(rep)`
 - `os.path.exists(fic1)`
 - `os.path.isfile(fic1)`
 - `os.path.isdir(fic1)`
- La manipulation des chemins avec le module de bas niveau `os`, est assez archaïque !
- Dans le cas où l'on veut gérer des chemins multiplateformes, il faut construire les chemins en tenant compte de la spécificité du séparateur de chemin en vigueur sur chaque système !

```
chemin = "rep1" + os.sep + "rep2" + os.sep + "rep3"
```

- Pour construire le chemin « `rep1/rep2/rep3` »
- Peu lisible !
- Le module de haut niveau `pathlib` permet d'apporter ces fonctionnalités.

Le module `pathlib`

- Le module `pathlib` permet une gestion de haut niveau du système de fichiers, permettant ainsi une manipulation aisée des chemins, des répertoires et des fichiers.
 - Il offre des classes représentant le système de fichiers avec la sémantique appropriée pour différents systèmes d'exploitation
- La classe `Path` constitue le principal élément de ce module. Elle permet l'expression de chemins et leur manipulation.

```
def test_path() :
    chemin1 = Path()
    print ( f"nom {chemin1}")
    chemin2 = Path("C:/temp")
    print ( chemin2) # C:\temp sous windows
    chemin3 = Path()/"temp"
    print ( chemin3)
    chemin4 = Path("C:\\")/"temp"
    print(chemin4)
```

- Une fois l'objet de type `Path` créé, des méthodes permettent de manipuler la référence de chemin ainsi créée.

```
def test_path() :  
    path = Path(".")  
    path.is_dir()  
    path.is_file()  
    path.absolute()
```

Manipuler les fichiers et les répertoires

- Python étant à l'origine conçu pour réaliser des opérations système, il est naturellement pourvu d'outils très simples d'utilisation pour manipuler des fichiers.
- Ces fonctionnalités de base font partis des «Buildin» il n'est pas nécessaire de devoir importer un quelconque module pour pouvoir les utiliser.
- La fonction d'ouverture de fichiers possède la signature suivante :
 - `open(file, mode='r')`
 - Le premier paramètre est le fichier à ouvrir,
 - Le second son mode d'ouverture (r étant pour read, par défaut)
- La fonction open renvoie un objet qui représente le fichier
 - L'objet est de type `_io.TextIOWrapper`
- Le fichier doit être fermé à l'aide de la méthode `close()`
 - `idFile.close()`

Les modes d'ouvertures de la fonction open

- Le langage fournit un ensemble de mode d'ouverture :

Caractère	Signification
'r'	Ouverture en lecture (par défaut).
'w'	Ouverture en écriture, le fichier est d'abord vidé.
'x'	Ouverture pour création. Une erreur est générée si le fichier existe.
'a'	Ouverture en écriture, le contenu est ajouté au contenu éventuellement existant.
'b'	Mode binaire.
't'	Mode texte (par défaut).
'+'	Ouverture pour mise à jour (lecture/écriture).

Ecriture dans un fichier

- La bibliothèque fournit différentes fonctions pour écrire dans un fichier :
 - `writelines` : écrit sur une ligne la collection fournit
 - `write` : écrit la chaîne passer en paramètre

```
def ecrire_fichier ():
    langages = ["Java", "C#", "Python", "C++"]
    fic = open("C:\\temp\\langages.txt", 'w')
    fic.writelines(langages)
    fic.close()
```

```
def ecrire_fichier ():
    langages = ["Java", "C#", "Python", "C++"]
    fic = open("C:\\temp\\langages.txt", 'w')
    for l in langages :
        fic.write ( l )
        fic.write ("\n")
    fic.close()
```

Lire et écrire dans un fichier

- La bibliothèque fournit différentes fonctions pour écrire dans un fichier :
 - `readlines` : lit le contenu du fichier ligne par ligne et renvoie une liste
 - `readline` : lit une ligne du fichier
 - `read` : lit le contenu du fichier dans une chaîne de caractère

```
def lire_fichier ():
    fic = open("C:\\temp\\langages.txt")
    lignes = fic.readlines()
    for ligne in lignes:
        print(ligne)
    fic.close()
```

Lecture dans un fichier

- Python nous facilite la lecture de fichier en permettant d'itérer le contenu d'un fichier :

```
def lire_fichier ():
    fic = open("C:\\temp\\langages.txt")
    for ligne in fic:
        print(ligne)
    fic.close()
```

Manipuler les fichiers et les répertoires

- Attention : lors de la manipulation de fichier, il faut toujours penser à fermer le fichier.
- Solution : Python propose de fermer implicitement le fichier en utilisant l'instruction `with` :

```
def ecrire_fichier ():
    langages = ["Java", "C#", "Python", "C++"]
    with open("C:\\temp\\langages.txt", 'w') as fic:
        fic.writelines(langages)
```

Accès direct

- Il est possible d'accéder à des fichiers en mode binaire en utilisant les modes `rb` ou `wb`.
- Dans ce cas vous pouvez réaliser des accès directs à l'aide de la méthode `seek` :
 - `seek (offset , base)`
- Positionne le curseur à l'endroit demandé en fonction de la base qui peut prendre trois `SEEK_SET(0)` : début du fichier
- valeurs :
 - `SEEK_CUR(1)` : position courante
 - `SEEK_END (2)` : fin du fichier
- La fonction `tell()` renvoie la position du curseur de lecture par rapport au début du fichier.

Manipuler des fichiers avec pathlib

- On peut utiliser la classe `Path` pour la manipulation de fichiers :
 - Les méthodes s'appliquent sur une référence de fichier, il n'est donc pas nécessaire de spécifier ce dernier en premier paramètre.

```
def ecrire_fichier():
    chemin = Path("C:/temp")
    fichier = chemin/"langages.txt"
    langages = ["Java", "C#", "Python", "C+"]
    with fichier.open('w') as fic:
        fic.writelines(langages)
```

Manipuler des fichiers csv

- Python fournit le module pour effectuer des traitements sur des fichiers CSV (Comma Separated Values):
 - Fichiers textes pour lesquels chaque ligne est constitué de colonne séparée par un caractère donné;
 - Le caractère de séparation par défaut est la virgule
 - Certains outils type tableur préfèrent utiliser le point-virgule
- Pour créer un fichier csv, vous devez utiliser une instance de `writer`

```
def ecrire_formatcsv():
    lignes = ["Ligne1", "Ligne2", "Ligne3"]
    chemin = Path("C:/temp")
    fichier = chemin/"fic.csv"
    fic = open(fichier, "a", newline="")
    c = csv.writer(fic, delimiter=";")
    for l in lignes:
        c.writerow(l)
    fic.close()
```

Lecture d'un fichiers csv

- Pour lire un fichier csv, vous devez utiliser une instance de « reader »

```
def lire_formatcsv():
    chemin = Path("C:/temp")
    fichier = chemin/"fic.csv"
    fic = open(fichier, "r", newline="")
    lignes = csv.reader(fic, delimiter=";")
    for l in lignes:
        print(l)
```

- Python fournit de la même façon un module `json` permettant de gérer des fichiers aux format json.

Détection du dialect par le Sniffer

```
with open("personnes.csv") as f:
    dialect = csv.Sniffer().sniff(f.readline())

    reader = csv.reader(f, dialect=dialect)
    data = list(reader)
    print(data)
```

Exemple d'utilisation d'un DictReader

```
with open("personnes.csv") as f:
    dialect = csv.Sniffer().sniff(f.readline())
    f.seek(0)

    reader = csv.DictReader(f, dialect=dialect)
    data = list(reader)

    print(data)
```

La sérialisation

- La sérialisation est le processus qui permet de convertir un objet en un flux.
- Il est alors possible de l'enregistrer dans un fichier, ou de le transmettre à une autre application.
- La récupération de ce flux et sa transformation en objet sont appelées la désérialisation.
- Le framework met à disposition trois techniques :
 - Sérialisation binaire : module pickle
 - Sérialisation Json : module json
 - Sérialisation XML :

Exemple

- Utilisation de deux classes : `Pickler` et `Unpickler`

```
def test_serialisation(nomfichier, obj):  
    buffer = Path()/nomfichier  
    f = open(buffer, 'wb')  
    pickler = pickle.Pickler(f, pickle.HIGHEST_PROTOCOL)  
    pickler.dump(obj)  
    f.close()
```

```
def test_deserialisation(nomfichier):  
    buffer = Path()/nomfichier  
    f = open(buffer, 'rb')  
    pickler = pickle.Unpickler(f)  
    obj = pickler.load()  
    return obj
```

Accéder aux bases de données

Présentation

- Python fournit des modules permettant de travailler avec des bases de données :
 - `sqlite3` permet de travailler avec `sqlite`
- `sqlite` est SGBD léger :
 - Utilise un sous ensemble de SQL
 - Les données se trouvent sur le terminal du client et non sur un serveur distant
 - Outil apprécié pour les applications embarqués
 - Intégré nativement à Python

Connexion à la base

- La méthode «connect» permet de nous connecter à une base :

```
import sqlite3
```

```
connection = sqlite3.connect(nameBd)
```

- Si le fichier .db n'existe pas alors il sera créé.
- La fonction renvoie un objet de type `Connection` qui nous permettra de travailler avec la base.
- Pour se déconnecter à la base, il faut utiliser l'instruction `disconnect()`

```
connection.close()
```

Authentification

- Sqlite ne propose pas de mécanisme d'authentification par défaut. Il est possible d'activer ce mécanisme
 - inclure le fichier de code source `ext/userauth/userauth.c` dans la génération
 - ajouter l'option de compilation `DSQLITE_USER_AUTHENTICATION`.
- Il est aussi possible de stocker la BDD directement dans la RAM en utilisant la chaîne clef `":memory:"`.
 - Dans ce cas, il n'y a pas de persistance des données après la déconnexion.
 - Utiliser pour effectuer des tests qui sont ainsi reproductibles et n'altèrent pas d'éventuelles BDD persistantes.

Exécution de requêtes

- L'exécution des requêtes se réalisent à l'aide de la méthode « `execute` » sur un objet `Cursor` :
- Un objet `Cursor` se récupère suite à l'appel de la méthode `cursor` sur l'objet de type `Connection`

```
import sqlite3

def create_table(nameBd):
    connection = sqlite3.connect(nameBd)
    cursor = connection.cursor()
    cursor.execute(""" CREATE TABLE IF NOT EXISTS personnes(
                        id INTEGER PRIMARY KEY,
                        nom TEXT,
                        prenom TEXT,
                        email TEXT ) """)

    connection.commit()
    connection.close()
```

Validation des données

- Lorsque nous effectuons des modifications sur une table (insertion, modification ou encore suppression d'éléments), celles-ci ne sont pas automatiquement validées :
 - La méthode `commit()` permet de valider la requête.
 - La méthode `rollback()` permet d'invalider une requête et de revenir à l'état précédent.

Paramétrage de requêtes

- L'insertion en base se fait par une requête SQL :

```
cursor.execute("""INSERT INTO personnes(nom, prenom , email)
                VALUES( 'nom1', 'prenom1' , 'email1') """)
```

- Pour éviter les injections SQL il faut paramétrer les requêtes:

```
cursor.execute("""INSERT INTO personnes(nom, prenom , email)
                VALUES(?, ?, ?)""", ("nom2", "prenom2", "email2"))
```


Récupération de données

- Pour récupérer des éléments, il faut effectuer une requête Sql puis parcourir le résultat à l'aide de l'objet `Cursor` :
 - `fetchone` permet de récupérer un élément sous forme de tuple ou none si il n'y en a pas.
 - `fetchall` permet de récupérer l'ensemble des résultats.
 - `fetchmany` permet de récupérer plusieurs résultats.
 - Le nombre de résultat par défaut correspond à la valeur de l'attribut `arraysize` du curseur.
 - On peut passer le nombre voulu en paramètre

```
def get_user(nameBd, name):
    connection = sqlite3.connect(nameBd)
    cursor = connection.cursor()
    cursor.execute("""SELECT nom, prenom , email FROM personnes WHERE nom=? """,
(name))
    user = cursor.fetchone()
    connection.close()
    return user
```

Gestion des erreurs

- La manipulation des données en base peut engendrer des erreurs. Il faut les traiter à l'aide des exceptions :

```
def create_table(nameBd):
    try:
        connection = sqlite3.connect(nameBd)
        cursor = connection.cursor()
        cursor.execute(""" CREATE TABLE IF NOT EXISTS personnes(
                                id INTEGER PRIMARY KEY,
                                nom TEXT,
                                prenom TEXT,
                                email TEXT ) """)

        connection.commit()
    except sqlite3.Error as ex:
        connection.rollback()
        print(ex)
    finally:
        if connection:
            connection.close()
```

Utilisation des bases de données

- Il est possible de travailler avec d'autres base de données :
 - Il faut récupérer le driver de la base de données
- Pour utiliser par Mysql il faut réaliser les installations suivantes :
 - `pip install mysql`

- pip install mysql-connector-python

```
def create_tablemysql():
    connection = mysql.connector.connect(host="localhost",
                                         user="root",
                                         password="root",
                                         database="bdUsers")

    auth_plugin="mysql_native_password"
    cursor = connection.cursor()
```

Les ORM

- Il existe plusieurs ORM pour python :
 - SQLAlchemy : le plus connu et le plus utilisé
 - Peewee : simple et efficace
 - Pony ORM
- SQLAlchemy est une ORM codé en python :
 - Il apporte un haut niveau d'abstraction
 - Il n'y a plus besoin de réaliser de requêtes SQL
 - Ce module doit être installé explicitement

La création d'interface graphique avec tkinter

Le graphisme en python

- Python propose un ensemble de bibliothèques pour concevoir des IHM:
 - `tkinter` : bibliothèque la plus utilisée
 - `wxPython` : bibliothèque connectant `wxwidgets` à Python. Elle est une alternative à `TkInter` particulièrement complète, proche du système d'exploitation.
 - `PyQt` : bibliothèque graphique libre connectant Qt à Python.

Conception d'une interface avec tkinter

- Pour créer une interface graphique en python, il faut respecter la démarche suivante :
 - créer la fenêtre principale
 - créer les composants graphiques
 - ajouter les composants à la fenêtre
 - définir la boucle d'évènement

```
import tkinter
def test_fenetre() :
    root = tkinter.Tk() # défini la fenêtre principale
    root.title("Demo Ihm")
    root.geometry ( "200x100")
    lmessage = tkinter.Label(text="Hello world")
    lmessage.pack() # on ajoute l'objet à la fenêtre principale
    root.mainloop() # on lance la boucle d'évènement
```

La boucle d'évènement

- Une application graphique doit pouvoir être réactive :
 - elle se modifie lorsque l'utilisateur génère un évènement clique sur un bouton ou un menu.
- Les applications basées sur `tkinter`, comme beaucoup d'autres applications graphiques, se basent sur l'utilisation d'une boucle des événements
 - La méthode `mainloop`
- La boucle des événements est une boucle qui bloque la sortie du programme. Elle traite continuellement les événements que reçoit l'application graphique.
 - si l'utilisateur appuie sur une touche du clavier ou bouge la souris, la boucle des événements en est alertée et crée des objets pour représenter ces événements.
 - tant que la boucle des événements s'exécute, l'application ne peut pas s'arrêter.
 - Elle attend un événement d'arrêt qui est, par exemple, émis lorsque l'utilisateur clique sur l'icône pour fermer la fenêtre principale.

L'approche objet

- Pour la conception d'interface on va privilégier une approche objet

```
from tkinter import Tk, Label

class MaFenetre(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("200x100")
        self.title("Demo Ihm")
        self.lmessage = Label(self, text="Hello world")
        self.lmessage.pack()

window = MaFenetre()
window.mainloop()
```

Les composants de base

- Le composant `Label` permet d'afficher un texte :

```
self.lmessage = Label(self, text="Hello world")
```

- Ce texte peut être modifié par la propriété `config` :

```
self.lmessage.config ( text="bienvenue en tkinter")
```

Les zones de saisie

- Elle a pour objectif de recevoir des informations provenant de l'utilisateur. La mise en oeuvre se réalise avec la classe `Entry`

```
self.l1langage = Label(self, text="Entrer votre langage")
self.l1langage.pack()
self.e1langage = Entry()
self.e1langage.pack()
```

- La récupération du contenu de la zone texte se fait avec la méthode `get()`

```
contenu = self.l1langage.get()
```

- Les principales autres méthodes sont :
 - `insert` : permet de modifier la zone de contenu
 - `delete` : permet de supprimer le contenu

Les zones de saisie multilignes

- La classe `Text` permet de définir une zone de saisie multi lignes :

```
self.lcommentaire = Label(self, text="Entrer vos commentaires")
self.lcommentaire.pack()
self.ecommentaire = Text()
self.ecommentaire.config(width=50, height=3)
self.ecommentaire.pack()
```

- La méthode `config` permet de définir la taille de la zone (nombre de lignes, nombre de colonnes)
- Les principales autres méthodes sont :
 - `get` : permet de récupérer le contenu de la zone
 - `insert` : permet de modifier la zone de contenu
 - `delete` : permet de supprimer le contenu

Les boutons

- La classe `Button` permet de définir un bouton poussoir.

```
self.bouton1 = Button(text="simple bouton")
```

- Un bouton peut contenir une image

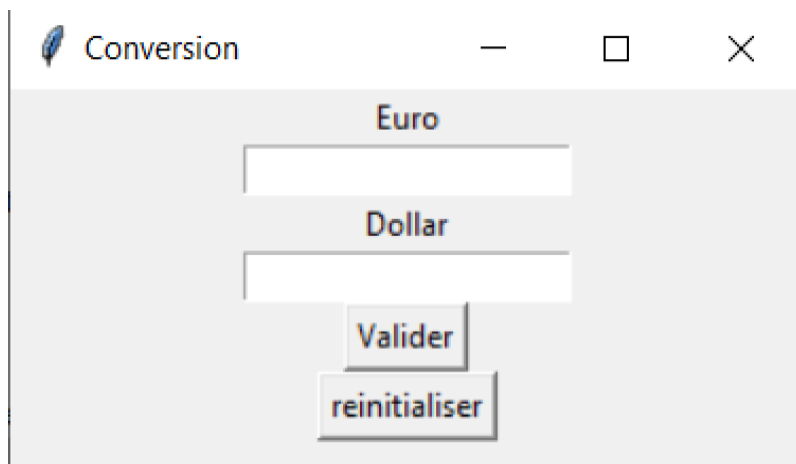
```
self.bouton2 = Button()
self.img = PhotoImage(file="Logo.png").subsample(2,2)
self.bouton2.config(image=self.img)
```

- Un bouton peut contenir un libellé associé à une image.

```
self.bouton3= Button(text="Click Me", image=self.img, compound=LEFT)
```

Un exemple complet

```
class MaFenetre(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x150")
        self.title("Conversion")
        self.label = Label(self, text="Euro")
        self.label.pack()
        self.txt_euro = Entry()
        self.txt_euro.pack()
        self.label = Label(self, text="Dollar")
        self.label.pack()
        self.txt_dollar = Entry()
        self.txt_dollar.pack()
        self.button_valider = Button(text="Valider")
        self.button_valider.pack()
        self.button_init = Button(text="reinitialiser")
        self.button_init.pack()
```



La gestion des évènements

- Tkinter est basé sur un modèle de programmation événementiel. Une interface est donc à l'écoute des différents événements.
- La méthode « bind » permet de lier un événement à un composant graphique

```
composant.bind ( evt , callback)
```

- evt représente le type d'événement qui doit être représenté sous le format suivant :
« <modificateur-type-detail> »
- Callback représente la fonction qui sera appelée lorsque l'événement est déclenché. Elle doit être défini sous le format suivant :

```
def fonc_callback ( event ) :  
    pass
```

Exemple

- On veut réaliser un traitement qui modifie un label lorsque l'on clique sur un bouton :

```
self.bouton1 = Button(text="click me")
self.bouton1.bind("<Button-1>", self.bouton_click )
self.bouton1.pack()
self.lMessage = Label(self, text="tkinter")
self.lMessage.pack()
```

- La fonction de callback est définie sous la forme suivante:

```
def bouton_click ( self, evt :Event ):
    self.lMessage.config(text="Evt reçu")
```

Evènement sur un bouton

- Tkinter propose un attribut command qui permet d'associer un traitement à un évènement click sur un bouton.

```
self.bouton2 = Button(text="test command")
self.bouton2.config( command = self.bouton_command)
self.bouton2.pack()
self.lMessage = Label(self, text="tkinter")
self.lMessage.pack();
```

- Attention la fonction n'attend pas d'évènement en paramètre

```
def bouton_command ( self):
    self.lMessage.config(text="Evt reçu")
```

Les principaux évènements

- Tkinter propose un ensemble d'évènements :

Type évènement	Rôle
<button-i>	Intercepte la pression d'un bouton (i peut prendre les valeurs 1,2,3 correspondant au bouton de la souris)
<buttonRelease-i>	Intercepte le relâchement du bouton
<doubleButton-i>	Intercepte une double pression
<enter>	est généré quand le curseur entre dans la zone de l'objet
<leave>	est généré quand le curseur sort de la zone de l'objet
<key>	Intercepte la pression de n'importe quel touche clavier
<motion>	est généré dès que le curseur bouge

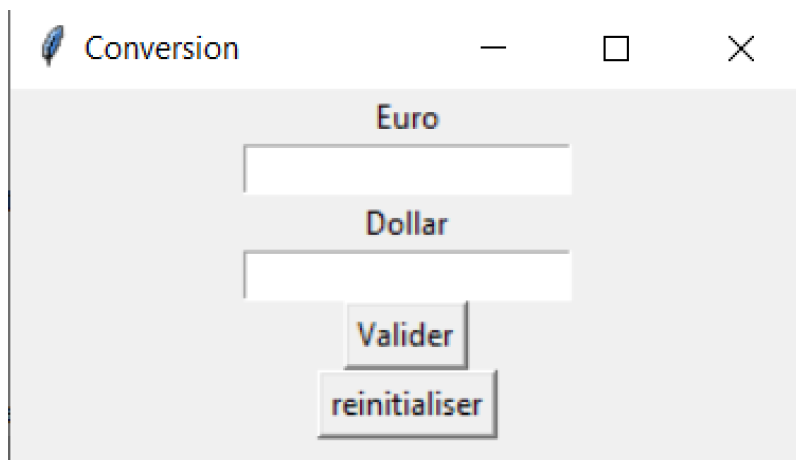
La classe Event

- La classe `Event` propose un ensemble de propriétés permettant de récupérer des informations sur la nature de l'évènement :
 - `widget` : objet qui a déclenché l'évènement
 - `x, y` : coordonnées de la souris par rapport à la fenêtre
 - `x-root, y-root` : position de la souris par rapport à l'écran
 - `char` : touche sélectionnée pour un évènement clavier
 - `num` : le numéro du bouton sélectionné

```
def bouton_click ( self, evt):  
    message = f"{evt.widget} {evt.x} {evt.y}"  
    self.lMessage.config ( text=message)
```

Un exemple complet

```
class MaFenetre(Tk):  
  
    def __init__(self):  
        super().__init__()  
        self.geometry("300x150")  
        self.title("Conversion")  
        self.label = Label(self, text="Euro")  
        self.label.pack()  
        self.txt_euro = Entry()  
        self.txt_euro.pack()  
        self.label = Label(self, text="Dollar")  
        self.label.pack()  
        self.txt_dollar = Entry()  
        self.txt_dollar.pack()  
        self.button_valider = Button(text="Valider")  
        self.button_valider.bind("<Button-1>" , self.conversion_dollar)  
        self.button_valider.pack()  
        self.button_init = Button(text="reinitialiser")  
        self.button_init.bind("<Button-1>", self.init_zone)  
        self.button_init.pack()
```



- Les fonctions de callback

```

def conversion_dollar(self, evt):
    valeur = self.txt_euro.get()
    valeur_dollar = float(valeur) * 1.02
    self.txt_dollar.insert(0, str(valeur_dollar))

def init_zone(self, evt):
    self.txt_dollar.delete("0", "end")
    self.txt_euro.delete("0", "end")

window = MaFenetre()
window.mainloop()

```

Disposition des composants

- Les objets containers comme la fenêtre propose 3 stratégies de disposition des composants :
 - `pack` : dispose les composants les uns en dessous des autres (par défaut). On peut spécifier explicitement le positionnement qui peut prendre 5 valeurs (`top`, `bottom`, `left`, `right`, `center`)
 - `grid` : dispose les composants dans une grille (ligne, colonne)
 - `place` : dispose les composants à un endroit donné (coordonnées `x,y`)

La stratégie `place`

- Cette stratégie permet de positionner les différents éléments en les plaçant dans une position spécifique dans le composant parent.

```

class MaFenetre(Tk):

    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.title("Layout avec place")
        self.label1 = Label(self, text="Label (10, 10)")
        self.label1.place(x=10, y=10)
        self.label2 = Label(self, text="Label (40, 40)")
        self.label2.place(x=40, y=40)
        self.label3 = Label(self, text="Label (70, 70)")
        self.label3.place(x=70, y=70)

```




La stratégie pack

- Cette stratégie permet de positionner les différents éléments dans 5 zones de votre conteneur (haut, bas , droit , gauche et centre).
- Il est possible de spécifier plusieurs composants par zone.

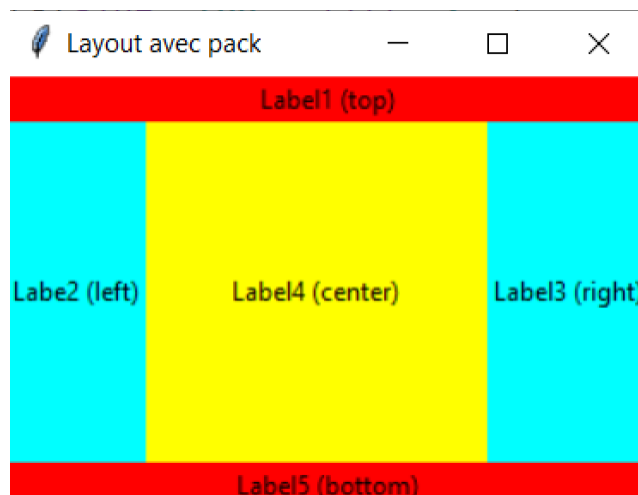
```
class MaFenetre(Tk):  
    def __init__(self):  
        super().__init__()  
        self.geometry("300x200")  
        self.title("Layout avec pack")  
        self.label1 = Label(self, text="Label1 (top)" , bg="red")  
        self.label1.pack(side="top")  
        self.label2 = Label(self, text="Labe2 (left)" , bg = "cyan")  
        self.label2.pack(side="left")  
        self.label3 = Label(self, text="Label3 (right)" , bg = "cyan")  
        self.label3.pack(side="right")  
        self.label4 = Label(self, text="Label4 (center)" , bg="yellow")  
        self.label4.pack()  
        self.label5 = Label(self, text="Label5 (bottom)" , bg="red")  
        self.label5.pack(side="bottom")
```



La propriété « fill »

- Il est possible de remplir tout l'espace du container à l'aide de la propriété `fill`:

```
class MaFenetre(Tk):  
  
    def __init__(self):  
        super().__init__()  
        self.geometry("300x200")  
        self.title("Layout avec pack")  
        self.label1 = Label(self, text="Label1 (top)" , bg="red" )  
        self.label1.pack(side="top" , fill="x" )  
        self.label5 = Label(self, text="Label5 (bottom)", bg="red")  
        self.label5.pack(side="bottom", fill="x")  
        self.label2 = Label(self, text="Labe2 (left)" , bg = "cyan" )  
        self.label2.pack(side="left" , fill = "y")  
        self.label3 = Label(self, text="Label3 (right)" , bg = "cyan" )  
        self.label3.pack(side="right" , fill = "y")  
        self.label4 = Label(self, text="Label4 (center)" , bg="yellow" )  
        self.label4.pack( expand="1" , fill ="both")
```



Les options de la stratégie pack

- La méthode `pack` possède un ensemble de propriétés :
 - `anchor` : définit un point d'ancrage qui peut prendre les valeurs :
 - N , S , E , O , NO , NE , SE , SW , CENTER
 - `padx` , `pady` : marge externe
 - `ipadx` , `ipady` : marge interne
 - `expand` : étend la zone disponible en fonction de la place disponible

Le mode `grid`

- Cette stratégie permet de positionner chacun de vos éléments dans une ou plusieurs cellules d'une grille.
- La grille est organisée en lignes et en colonnes.

```
class MaFenetre(Tk):
```

```
def __init__(self):
    super().__init__()
    self.geometry("300x200")
    self.title("Layout avec pack")
    self.label1 = Label(self, text="cel 0,0 " , bg="red" )
    self.label1.grid(row="0" ,column="0" )
    self.label2 = Label(self, text="cel 0,1" , bg = "cyan" )
    self.label2.grid(row="0" ,column="1" )
    self.label3 = Label(self, text="cel 1,0" , bg = "cyan" )
    self.label3.grid(row="1" ,column="0" )
    self.label4 = Label(self, text="cel 1,1" , bg="yellow" )
    self.label4.grid(row="1" ,column="1" )
```



- La grille peut occuper la totalité de votre classe container. Pour cela il faut la configurer explicitement :

```
class MaFenetre(Tk):
    def __init__(self):
        super().__init__()
        self.grid_rowconfigure(0, weight=1)
        self.grid_rowconfigure(1, weight=1)
        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(1, weight=1)
```



- Par défaut, un élément occupe le minimum de place dans sa cellule et il est centré en plein milieu de cette cellule.

- On peut changer cela grâce au paramètre `sticky` qui peut contenir une chaîne de caractères composés des caractères `n,s,e` et `w` qui correspondent au quatre points cardinaux (`north`, `south`, `east` et `west`).
- La valeur `sticky="nsew"` permettra de prendre toute la place disponible.
- On peut aussi rajouter des marges autour de chaque élément à l'aide des propriétés `padx` et `pady`.

```
self.label1.grid(row="0" ,column="0" , sticky="nsew")
#idem pour les autres
```



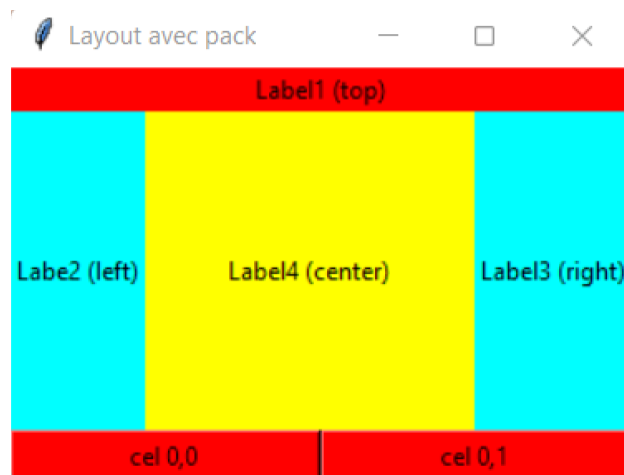
Structurer son application avec les frames

- Vous pouvez structurer votre fenêtre en utilisant des classe containers. Chaque classe container peut avoir son propre modèle de disposition :
- `Frame` : container pouvant contenir d'autres éléments graphiques
- `LabelFrame` : idem `Frame` avec un label sur le bord.
- `Panedwindow` : container pouvant contenir d'autres éléments graphiques. Il possède une barre (ceinture) pour séparer les widgets enfants.

Exemple

```
class MaFenetre(Tk):

    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.label1 = Label(self, text="Label1 (top)" , bg="red" )
        self.label1.pack(side="top" ,fill="x" )
        self.frame = Frame ()
        self.frame.config(width=300, height=100)
        self.frame.grid_rowconfigure(0, weight=1)
        self.frame.grid_columnconfigure(0, weight=1)
        self.frame.grid_columnconfigure(1, weight=1)
        self.button1 = Button(self.frame, text="cel 0,0 " , bg="red")
        self.button1.grid(row="0" , column="0" , sticky="nsew" )
        self.button2 = Button(self.frame, text="cel 0,1 " , bg="red")
        self.button2.grid(row="0" , column="1", sticky="nsew")
        self.frame.pack ( side="bottom",fill="both" )
```



```
self.label2 = Label(self, text="Labe2 (left)" , bg = "cyan" )
self.label2.pack(side="left" , fill = "y")
self.label3 = Label(self, text="Label3 (right)" , bg = "cyan" )
self.label3.pack(side="right" , fill = "y")
self.label4 = Label(self, text="Label4 (center)" , bg="yellow" )
self.label4.pack( expand="1" , fill ="both")
```

Un exemple complet

```
from tkinter import Tk, Button, Label, Entry

class MaFenetre(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("250x150")
        self.title("conversion monnaie")
        self.grid_rowconfigure(0, weight=1)
        self.grid_rowconfigure(1, weight=1)
        self.grid_rowconfigure(2, weight=1)
        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(1, weight=1)
        self.grid_rowconfigure(2, weight=1)
        self.label = Label(self, text="Euro")
        self.label.grid(row="0", column="0" )
        self.txt_euro = Entry()
        self.txt_euro.grid(row="0", column="1", )
        self.label = Label(self, text="Dollar")
        self.label.grid(row="1", column="0" )
        self.txt_dollar = Entry()
        self.txt_dollar.grid(row="1", column="1" )
        self.button_valider = Button(text="valider")
        self.button_valider.bind("<Button-1>" , self.conversion_dollar)
        self.button_valider.grid(row="2", column="0" )
        self.button_init = Button(text="reinitialiser")
        self.button_init.bind("<Button-1>" , self.init_zone)
        self.button_init.grid(row="2", column="1" )
    def conversion_dollar(self, evt):
        valeur = self.txt_euro.get()
        valeur_dollar = float(valeur) * 1.02
        self.txt_dollar.insert(0, str(valeur_dollar))
```

```
def init_zone(self, evt):
    self.txt_dollar.delete("0", "end")
    self.txt_euro.delete("0", "end")

window = MaFenetre()
window.mainloop()
```

Les autres composants

- Tkinter propose un ensemble de composants graphique :
 - Les cases à cocher : `Checkbutton`, `Radiobutton`
 - Les listes : `Listbox`, `Combobox`
 - Les menus : `Menu`
 - Les zones de dessins : `Canvas`
 - Les boites de dialogues : `askyesno`, `showinfo`, `showerror`
 - Zone numérique : `SpinBox`, `Scale`

Les variables de contrôles

- Les variables de contrôle permettent de rendre dynamique certains affichages de tkinter:
 - Elles vont lier une propriété d'un composant à une variable
 - Toute modification de la variable entrainera implicitement une mise à jour de la propriété
 - L'association se réaliser avec la propriété « `textvariable` » du composant
- Tkinter propose plusieurs variables de contrôles :
 - `IntVar`,
 - `DoubleVar`,
 - `StringVar`
- On définit la valeur d'une variable de contrôle avec la méthode `set()`, et on récupère son contenu avec la méthode `get()`.

Exemple

```
class MaFenetre(Tk):
    def __init__(self):
        super().__init__()
        self.geometry("300x200")
        self.title = "Test variable"

        self.nouveaumessage = tkinter.StringVar()
        self.nouveaumessage.set ( "Bienvenue en tkinter")

        self.lmessage = Label( textvariable= self.nouveaumessage )
        self.lmessage.pack()

        self.bouton1 = Button(text="change_label" , command=self.change_label)
        self.bouton1.pack()
```

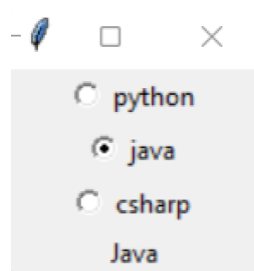
```
def change_label (self) :  
    self.nouveaumessage.set("click recu")
```

Les cases à cocher

- L'utilisation des cases à cocher nécessite de déclarer une variable pour mémoriser l'état du bouton.
 - cette variable sera associée aux différents boutons lors de sa création
- La récupération de la valeur sélectionnée se réalise à travers la méthode `get()`
- La modification de la valeur sélectionnée se réalise à travers la méthode `set()`
- Il est possible d'associer un libellé au bouton avec la méthode `config`.
- On peut associer un traitement à un bouton à l'aide de la propriété `command` ou en utilisant la méthode `bind`

Exemple

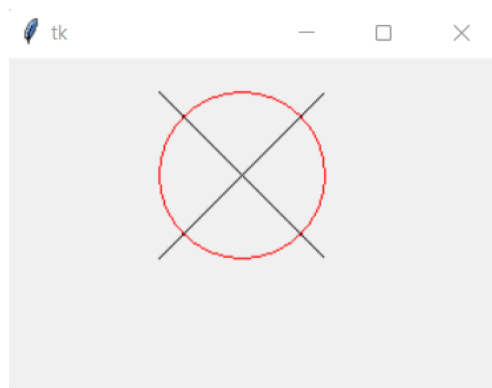
```
def __init__(self):  
    super().__init__()  
    self.v= tkinter.StringVar()  
    self.v.set("python")  
    self.rpython = tkinter.Radiobutton(self, variable=self.v, value="python",  
command=self.choix_formation)  
    self.rpython.config(text="python")  
    self.rpython.pack()  
    self.rjava = tkinter.Radiobutton(self, variable=self.v, value="java",  
command=self.choix_formation)  
    self.rjava.config(text="java")  
    self.rjava.pack()  
    self.rcsharp = tkinter.Radiobutton(self, variable=self.v, value="csharp",  
command=self.choix_formation)  
    self.rcsharp.config(text="csharp")  
    self.rcsharp.pack()  
    self.lformation = tkinter.Label(text="formation choisi")  
    self.lformation.pack()  
  
def choix_formation ( self):  
    self.lformation.config ( text=self.v.get())
```



Les canevas

- Les canevas permettent de définir des zones dans lesquelles nous allons pouvoir dessiner des formes (lignes, ellipses , ...).

```
def __init__(self):  
    super().__init__()  
    self.geometry("300x200")  
    self.title = "Test canevas"  
  
    self.canevas = tkinter.Canvas()  
    self.canevas.config ( width = 150 , height = 150 )  
    self.canevas.pack()  
  
    self.canevas.create_oval ( 20 , 20 , 120 , 120 , outline = "red")  
    self.canevas.create_line ( 20 , 20 , 120, 120 , fill="black" )  
    self.canevas.create_line( 20 , 120 , 120 , 20 , fill = "black")
```



Les images

- La classe `PhotoImage` permet de représenter une image.
- Elle doit être associée à un autre composant pour l'intégrer dans votre interface :
 - `Label`, `Button` , `Canvas`

```
def __init__(self):  
    super().__init__()  
    self.geometry("200x200")  
    self.title = "Test image"  
    self.img = PhotoImage(file="Logo.png").subsample(3,3)  
    self.canvas = Canvas(self, width=100, height=100)  
    self.canvas.create_image(50, 50, image=self.img)  
    self.canvas.pack()
```

- La classe `PhotoImage` est relativement limité et ne supporte que peu de format (gif, png, pgm) .
- On va donc privilégier la classe `Image` de l'api `Pillow`.
- Le chargement d'une image se réalise à l'aide de la méthode `open` :


```
Import PIL
def affichage_image ( self ) :
    image = PIL.Image.open("./Logo.png")
    image.show()
```

- L'affichage se réaliser dans un outil externe.
- Vous pouvez créer une image `Tkinter` à partir d'une image `Pillow`

```
self.photo = ImageTk.PhotoImage(image)
```

Le timer

- La fenêtre racine de `tkinter` a une méthode `after` qui permet d'appeler une méthode après une période donnée :
 - `after (intervalleTemps , nom_fonction)`

```
def __init__(self):
    super().__init__()
    self.geometry("200x100")
    self.title = "Test timer"
    self.label = Label(text="heure", font=('Helvetica', 24), fg='blue')
    self.label.pack()
    self.update_clock()

def update_clock(self):
    date_jour = datetime.now()
    date_format = date_jour.strftime("%H:%M:%S")
    self.label.configure(text=date_format)
    self.after(1000, self.update_clock)
```

Pour aller plus loin

Documenter le code

D'une manière générale, le code doit être commenté pour chaque fonction. On recommande également de ne pas commenter de façon excessive. En effet, les commentaires du type :

```
# ceci est un commentaire
```

ne doivent être utilisés que pour signaler à un relecteur du code ou rappeler à son créateur une particularité qui ne serait pas explicite à la lecture du code en question.

Le code doit comporter en particulier des lignes pour toutes les fonctions. Ces blocs de documentation sont délimités à l'aide de guillemets :

```
"""Ceci est un bloc de commentaire"""
```

Il est possible d'insérer des commentaires sur une ou plusieurs lignes.

D'une manière générale, un commentaire comporte toujours en première ligne la description du rôle de la fonction.

Les lignes ci-après illustrent la documentation d'une méthode qui ne retourne pas de paramètre et ne prend pas de paramètre. Il y a donc une seule ligne :

```
def augmente_carburant(self):  
    """augmente le volume de carburant d'un litre"""  
    self.vol_carburant += 1
```

Dans un cas où il y a des paramètres, le commentaire qui décrit sur chaque ligne leur type et leur rôle. La valeur par défaut, quand il y en a une, doit être également indiquée. Il en va de même pour la valeur retournée. À titre d'illustration :

```
def maj_val(self, nouv_val=10000):  
    """Mise à jour de la valeur de la voiture en Euros  
    paramètre : nouv_val -- valeur pour la mise à jour (10 000 par défaut)  
    return : entier -- valeur de l'attribut valeur après mise à jour  
    """  
    self.valeur = nouv_val  
    return self.valeur
```

Commentons la classe Voiture suivante :

```
class Voiture(object):  
  
    def __init__(self, modele, marque, annee):  
        """constructeur de la classe voiture  
        paramètres :  
            modele -- modèle de la voiture  
            marque -- marque du véhicule  
            annee -- année de fabrication  
        """  
        self.modele = modele  
        self.marque = marque  
        self.annee = annee  
        self.vol_carburant = 0  
        self.valeur = 0  
  
    def augmente_carburant(self):  
        """augmente le volume de carburant d'un litre"""  
        self.vol_carburant += 1  
  
    def reservoir_plein(self):  
        """Vérifie si le niveau de carburant est à 50L  
        return : booléen -- True si la valeur est 50 False sinon  
        """  
        if self.augmente_carburant == 50:  
            return True  
        else:  
            return False  
  
    def maj_val(self, nouv_val=10000):  
        """Mise à jour de la valeur de la voiture en Euros
```

```

    paramètre : nouv_val -- valeur pour la mise à jour (10 000 par défaut)
    return : entier -- valeur de l'attribut valeur après mise à jour
    """

    self.valeur = nouv_val
    return self.valeur

if __name__ == '__main__':
    ma_voiture = Voiture(modele='406', marque='peugeot', annee=2014)
    ma_voiture.augmente_carburant()

```

L'intérêt des blocs de commentaire, qu'on nomme par ailleurs docstrings, n'est pas seulement d'illustrer le code. Ils permettent aux outils d'autocomplétion de proposer le contenu d'une docstring lorsqu'on utilise une fonction liée à cette docstring.

The screenshot shows a code editor with a Python file named 'classe_voiture.py'. The code defines a class 'Voiture' with several methods and attributes. A docstring is provided for the 'augmente_carburant' method. An autocomplete popup is visible over the 'augmente_carburant' method call, showing a list of attributes and methods with their types.

```

1 augmente le volume de carburant d'un litre
2
3
4 """constructeur de la classe voiture
5
6 parametres :
7     modele -- modèle de la voiture
8     marque -- marque du véhicule
9     annee -- année de fabrication
10 """
11 self.modele = modele
12 self.marque = marque
13 self.annee = annee
14 self.vol_carburant = 0
15 self.valeur = 0
16
17 def augmente_carburant(self, nouv_val):
18     """augmente le volume de carburant d'un litre
19     parametres :
20         nouv_val -- valeur pour la mise à jour (10 000 par défaut)
21     return : entier -- valeur de l'attribut valeur après mise à jour
22     """
23     self.valeur = nouv_val
24     return self.valeur
25
26 def reservoir_plein(self):
27     """vérifie si le réservoir est plein
28     return : bool -- True si le réservoir est plein, False sinon
29     """
30     if self.vol_carburant == 0:
31         return False
32     else:
33         return True
34
35 def maj_val(self, nouv_val):
36     """Mise à jour de la valeur
37     parametres :
38         nouv_val -- valeur pour la mise à jour (10 000 par défaut)
39     return : entier -- valeur de l'attribut valeur après mise à jour
40     """
41     self.valeur = nouv_val
42     return self.valeur
43
44 if __name__ == '__main__':
45     ma_voiture = Voiture(modele='406', marque='peugeot', annee=2014)
46     ma_voiture.augmente_carburant()
47

```

Autocomplete popup for 'augmente_carburant':

annee	attribute: instance
augmente_carburant	attribute: function
self.vol_carburant	attribute: instance
maj_val	attribute: function
marque	attribute: instance
modele	attribute: instance
reservoir_plein	attribute: function
valeur	attribute: instance
vol_carburant	attribute: instance
__init__	attribute: function
__class__	builtin: class
__delattr__	builtin: function
__dir__	builtin: function
__doc__	builtin: instance
__eq__	builtin: function
__format__	builtin: function
__ge__	builtin: function
__getattr__	builtin: function
__gt__	builtin: function
__hash__	builtin: function
__le__	builtin: function
__lt__	builtin: function
__ne__	builtin: function
__new__	builtin: function
__reduce__	builtin: function
__reduce_ex__	builtin: function
__repr__	builtin: function
__setattr__	builtin: function
__sizeof__	builtin: function
__str__	builtin: function
__subclasshook__	builtin: function

Les tests

Il existe plusieurs types de tests :

- tests unitaires
- tests d'intégration

Les tests d'intégration sont réalisés lors de la phase d'assemblage des différentes parties de code du programme final. Ils sont généralement précédés par les tests unitaires qui sont l'objet de cette section.

Les tests unitaires permettent de tester des portions particulières du code afin de vérifier si les spécifications sont suivies. Les tests sont importants même pour le développeur qui travaille seul et dont le travail ne sera pas directement utilisé par un tiers.

En effet, les tests permettent d'éviter la « régression » d'un code ou d'un projet. Imaginons que vous développez une librairie de fonctions et que chaque fonction est couverte par des tests.

Les tests vous permettront en partie d'assurer la compatibilité entre les anciennes utilisations de votre librairie et les utilisations postérieures à vos modifications.

Il existe différentes stratégies pour créer les tests unitaires. Certaines pratiques encouragent à créer les tests lors de la conception du projet ou de l'objet à développer, c'est-à-dire avant d'écrire le code. Ceci peut paraître étrange, mais présente de nombreux avantages, principalement l'absence de biais dans la conception des tests.

En effet, si l'auteur d'un code est également l'auteur des tests (ce qui ne pose pas de problème intrinsèque), et si ce même auteur écrit les tests après avoir écrit le code, alors il peut, de façon volontaire ou complètement involontaire, concevoir des tests qui ne couvrent que les éléments qu'il juge intéressants après le développement du code. Le risque est donc de tester uniquement des cas que le développeur sait qu'ils fonctionneront avec le code implémenté.

Pour réduire ce risque et donc enlever le biais potentiel, il est préférable d'écrire les tests avant le développement du code ou de les faire écrire par un tiers. Cette méthodologie de création des tests postérieure au développement du code est nommée *Test Driven Development* (en français, développement piloté par les tests).

Après ces considérations théoriques sur l'intérêt de tester le code, il est nécessaire de se poser la question sur les éléments à tester : lesquels ? tous ?...

Sur ce point, il n'existe pas de réponse parfaite. Les pratiques sont propres aux équipes de développement : c'est donc le contexte du projet et l'expérience de l'équipe qui va guider les choix.

Nous pouvons tout de même donner quelques éléments généraux de bonnes pratiques :

- Tester toutes les propriétés disponibles dans la spécification.
- Chercher à couvrir 100 % du code, c'est-à-dire que chaque ligne du code est couverte au moins par un test.
- Couvrir uniquement les éléments publics du code. En effet, les éléments privés sont appelés par les éléments publics. Par conséquent, quand on couvre un élément public par un test unitaire, les éléments privés qu'il utilise sont par voie de conséquence couverts eux-mêmes par le test.
- Retirer aucun test qui était déjà présent dans la base de tests existante.

Il existe de nombreux outils de test disponibles pour Python.

- nose
- pytest
- unittest

Les tests avec Unit Test

- Les logiciels deviennent de plus en plus complexes et importants.
- Pas de programme sans bug!
- Important de tester avant de distribuer son code.
- On peut distinguer les tests en fonction de leur :
 - objectif :
 - scénario : vérifier que le comportement fonctionnel est bien celui attendu
 - non-régression : ce qui fonctionnait dans les versions précédentes du code fonctionne toujours dans la nouvelle version
 - performance (bench) ou charge (load) : les temps de réponse à une requête sont conformes aux attentes
 - intégration/fonctionnels : le code s'intègre bien avec les autres éléments du système.
 - cible :
 - scénarios : tester un cas d'utilisation
 - unitaires : tester un composant du système
 - technologie :
 - Web : envoyer des requêtes Web simulant le comportement d'utilisateur(s)
- ◦
- Unittest a été intégré à la bibliothèque standard Python depuis la version 2.1:
 - contient un framework de test
 - lanceur de test.

Création de tests

- Un test est une classe Python qui doit hériter de TestCase :

```
import unittest

class MyTestCase(unittest.TestCase):
    def test_something(self):
        self.assertEqual(True, False)

if __name__ == '__main__':
    unittest.main()
```

- La classe de test doit posséder le nom de la classe métier à tester avec l'extension «Test».

Ecriture de Test

- Une méthode `Test` comprend généralement trois étapes :
 - Initialisation des données requises ou instantiations des objets
 - Invocation des méthodes
 - Vérification des résultats des traitements
- Chacun Test doit être indépendant.

- La vérification des résultats se fait à l'aide des assertions :
 - `assertEqual()`, `assertNotEqual()`
 - `assertTrue()`, `assertNotTrue()`
- Les assertions non vérifiées entraînent une remontée d'erreur

```
def test_rouler(self):
    v = Voiture()
    v.rouler(20)
    self.assertEqual(v.get_kilometrage() , 20 )
```

Initialisation des tests

- La classe `TestCase` fournit un ensemble de méthodes pour l'exécution des tests :
 - `setUp()` : méthode appelée pour réaliser la mise en place du test. Elle est exécutée immédiatement avant l'appel de la méthode de test ;
 - `tearDown()` : méthode appelée immédiatement après l'appel de la méthode de test et l'enregistrement du résultat.
 - `setUpClass()` : méthode de classe appelée avant l'exécution des tests dans la classe en `question.setUpClass` est appelée avec la classe comme seul argument et doit être décorée sous la forme `@classmethod()`

Gestion des exceptions

- Il est possible de tester les erreurs attendues.
 - utiliser `assertRaises()` comme gestionnaire de contexte,
 - à l'intérieur du bloc exécuter les étapes de test
- Les tests peuvent s'exécuter en ligne de commande :

```
python -m unittest test\ProduitTest.py
```

Les tests avec Pytest

Unittest permet de réaliser des tests de manière intéressante, mais il existe un module beaucoup plus pertinent : **pytest**.

Le gros avantage de celui-ci est de nécessiter une écriture des tests moins longue.

Avec pytest il n'est pas nécessaire de surcharger une classe. Il suffit d'écrire des fonctions. Et d'utiliser la fonction `assert` pour tester les valeurs retournées par des méthodes ou les valeurs des attributs.

Avant toute chose, il faut installer le module `pytest` à l'aide de `pip`.

```
pip install pytest
```

Le code qui suit teste la classe `Voiture` avec `pytest` :

```
from classe_voiture import Voiture
def test_cons():

    ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
```

```

assert ma_voiture.modele == 'Mustang'
assert type(ma_voiture.modele) is str
assert type(ma_voiture.marque) is str
assert type(ma_voiture.marque) is int or float

def test_augmente_carburant():

    ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
    ma_voiture.augmente_carburant()
    assert ma_voiture.vol_carburant == 1

```

Pour exécuter pytest, il suffit de saisir la ligne suivante dans un terminal :

```
pytest classe_voiture_test_pytest.py
```

Le résultat obtenu est le suivant :

```

pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$ pytest
classe_voiture_test_pytest.py
=====
===== test session starts
=====
platform linux2 -- Python 2.7.13, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /home/pi/Documents/Redaction/PythonRaspberry/Chap9/script, inifile:
collected 2 items

classe_voiture_test_pytest.py .
.
=====
===== 2 passed in 0.12 seconds
=====
=====

```

Il est également possible de rendre pytest plus verbeux à l'aide de l'option -v. Ce qui donne :

```

pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$ pytest
-v classe_voiture_test_pytest.py
=====
===== test session starts
=====
platform linux2 -- Python 2.7.13, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
-- /usr/bin/python
cachedir: .cache
rootdir: /home/pi/Documents/Redaction/PythonRaspberry/Chap9/script, inifile:
collected 2 items

classe_voiture_test_pytest.py::test_cons PASSED
classe_voiture_test_pytest.py::test_augmente_carburant PASSED

=====
===== 2 passed in 0.04 seconds
=====

```

Éviter la répétition avec les *pytest fixtures*

Pour éviter de réécrire un code utilisé dans plusieurs tests, Pytest dispose d'un [décorateur](#) spécifique appelé fixture qui permet d'accéder facilement aux éléments nécessaires à l'exécution d'un test, de la donnée par exemple.

Pour spécifier qu'une fonction est une [fixture](#), il sera nécessaire d'utiliser le décorateur **@pytest.fixture** avant de la définir.

```
from classe_voiture import voiture
import pytest

@pytest.fixture #test fixture decorator
def voiture():
    return voiture(modele='Mustang', marque='ford', annee='1968')

def test_cons(voiture):
    assert voiture.modele == 'Mustang'
    assert type(voiture.modele) is str
    assert type(voiture.marque) is str
    assert type(voiture.marque) is int or float
```

Le décorateur *parametrize tests* pour tester une suite d'instructions

```
from student import Student
import pytest

@pytest.mark.parametrize("grade_1, grade_2, grade_3, average", [(12, 10, 8, 10),
(20, 18, 16, 18), (3, 9, 9, 7)])
def test_average(grade_1, grade_2, grade_3, average):
    student = Student()
    student.grades = [] #Set an empty list of grades each time the test runs
    student.add_grade(grade_1)
    student.add_grade(grade_2)
    student.add_grade(grade_3)
    assert student.academic_average == average
```