

# Programmer en Python

Support de cours

Réf. T463-001





Module 0

# A propos de cette formation

# Votre formateur

- Son nom
- Ses activités
- Ses domaines de compétence
- Ses qualifications et expériences pour ce cours

# Votre formation - Présentation

- Description
  - Dans cette formation vous apprendrez la syntaxe du langage Python ainsi que la mise en œuvre de tous les concepts de la programmation fonctionnelle et orientée objet. Vous apprendrez également à utiliser les composants de la bibliothèque standard et à utiliser un IDE pour le développement et la mise au point de vos applications.
- Profil des stagiaires
  - Développeurs, architectes techniques, chefs de projet, administrateurs système.
- Connaissances préalables
  - Au minimum, avoir des connaissances en algorithmie ;
  - Idéalement, connaître un langage de programmation structuré (C, VB, Java...).
- Objectifs à atteindre
  - Comprendre et écrire des scripts en langage Python ;
  - Comprendre et mettre en œuvre les concepts de la programmation orientée objet avec le langage Python ;
  - Installer et utiliser des modules Python (administration système, ...).

# Votre formation - Programme

- Présentation de Python
  - Historique
  - Cas d'utilisation
  - Caractéristiques du langage
  - Le contenu de Python
  - Exécution d'un programme Python
- Mise en place d'une plateforme Python
  - Installation de Python
  - La console Python
  - Les librairies additionnelles
  - Les environnements virtuels
  - Les IDE pour Python
- Les bases du langage
  - Syntaxe du langage
  - Instructions et délimiteurs
  - Les blocs
  - Les commentaires
  - Les types de données
  - Typage dynamique fort
  - Les types de données simples
  - Traitement des chaînes de caractères
  - Les conversions de types
  - Les types de données évoluées
- Les opérateurs
- Les structures de contrôle
- La structure conditionnelle
- Les structures itératives
- Entrée et sortie standards
- Les fonctions
  - Utilité des fonctions
  - Déclaration et utilisation d'une fonction
  - Les paramètres
  - Le retour d'une fonction
  - Les paramètres optionnels
  - Les paramètres en nombre variable
  - Les paramètres nommés
- Les modules
  - Utilité de la structuration en module
  - Le nommage des modules
  - Organisation en packages
  - Importation de modules
  - Importation de fonctionnalités de modules
  - Le chemin de localisation des modules : PYTHONPATH

# Votre formation – Programme (Suite)

- La programmation orientée objet
  - Historique
  - Les concepts
  - La notation UML
  - Les méthodologies associées à l'objet
  - La notion d'objet
  - Le concept de Classe
  - Déclaration de classe
  - Déclaration des attributs et des méthodes
  - Création d'objet : instanciation
  - La visibilité des objets
  - L'encapsulation
  - L'héritage
  - Les usages de l'héritage
  - L'héritage multiple et ses conséquences
  - Le polymorphisme : redéfinition de méthodes
- Concepts avancés
  - La classe 'object'
  - Les méthodes spéciales des objets
  - Redéfinition des méthodes spéciales
  - La surcharge des opérateurs
  - La gestion des erreurs avec les exceptions
  - Génération d'exceptions
  - Traitement des exceptions : try ... except ...
  - Stratégies de gestion et propagation
  - Cas particulier de la gestion des ressources : with
- La bibliothèque standard
  - Interaction avec le système d'exploitation
  - Collecter des informations sur le système
  - Ecrire des scripts portables entre les systèmes d'exploitation
  - Intéragir avec les processus
  - Travailler avec les chemins d'accès
  - Manipuler les fichiers et les répertoires

# Tour de table – Présentez-vous

- Votre nom
- Votre société
- Votre métier
- Vos compétences dans des domaines en rapport avec cette formation
- Les objectifs et vos attentes vis-à-vis de cette formation

# Logistique

- Horaires de la formation
  - 9h00-12h30
  - 14h00-17h30
- Pauses

Merci d'éteindre vos téléphones portables





# Module 1

## Présentation de Python

# Contenu du module

- Historique
- Cas d'utilisation
- Caractéristiques du langage
- Le contenu de Python
- Exécution d'un programme 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
- La licence libre de Python, oscillant entre la *GPL* et la licence *Apache* est créée 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
- Programmation scientifique
  - Calculs avancés
- Programmation Web
  - Frameworks Web MVC (Django)
- Informatique embarquée
  - Raspberry PI, Arduino, ...
- Programmation graphique
  - Applications de bureau, prototypage de jeux vidéos
- Education
  - Apprentissage et mise en pratique de l'algorithmie

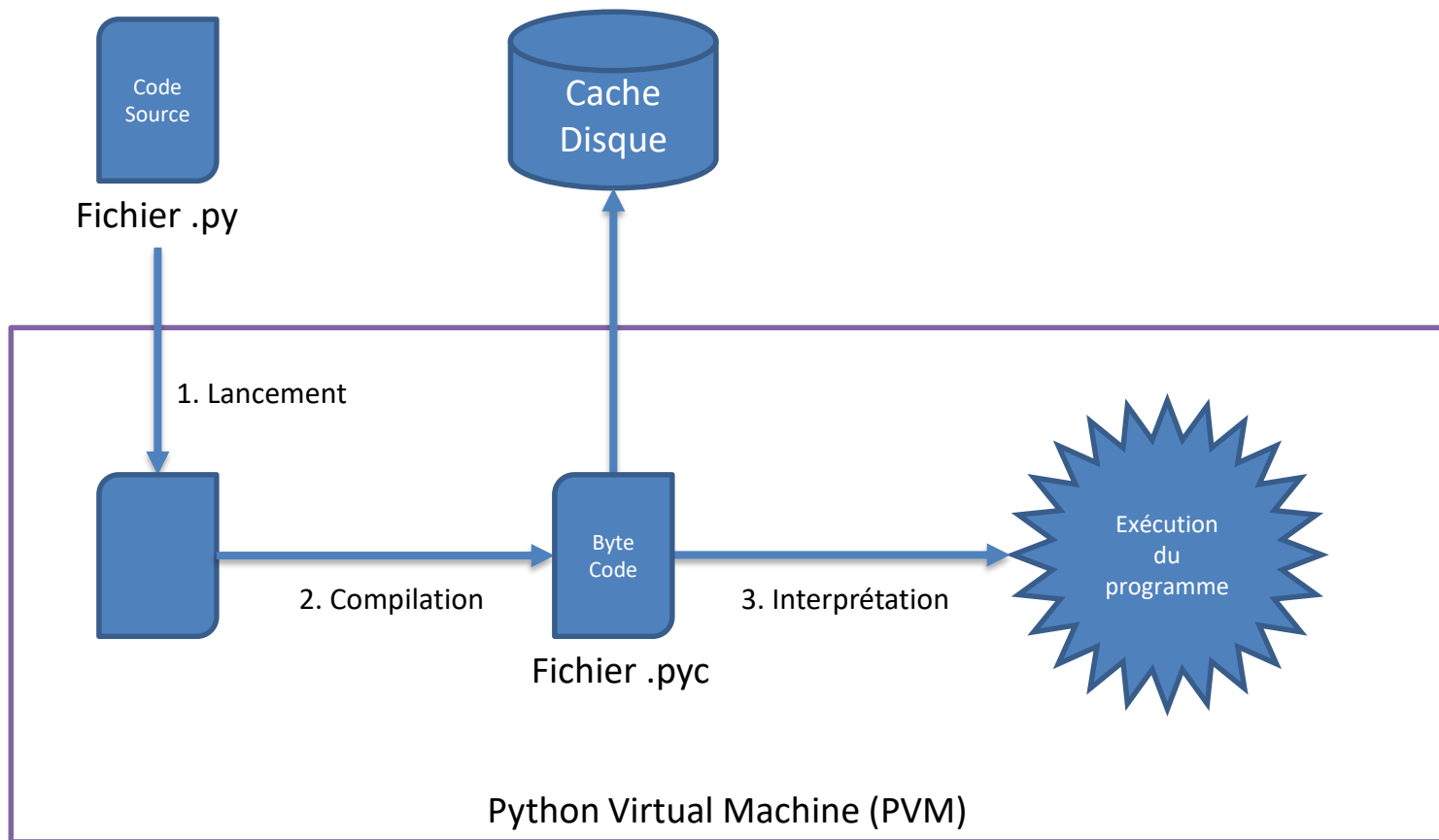
# Caractéristiques du langage

- 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, ...

# 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





## Module 2

# Mise en place d'une plateforme Python

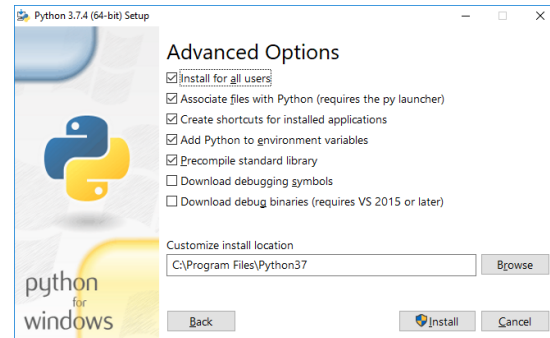
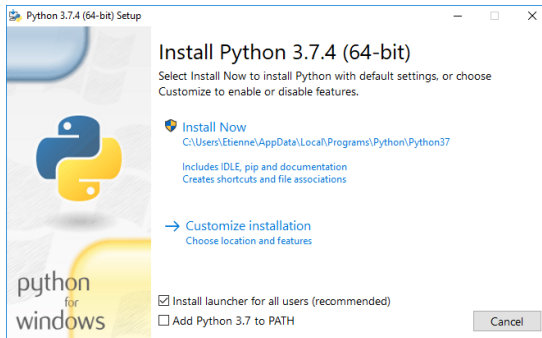


# Contenu du module

- Installation de Python
- La console Python
- Les librairies additionnelles
  - Installation avec PIP
  - Les commandes de PIP
  - Installation à partir des sources
- Les environnements virtuels
- Les IDE pour Python

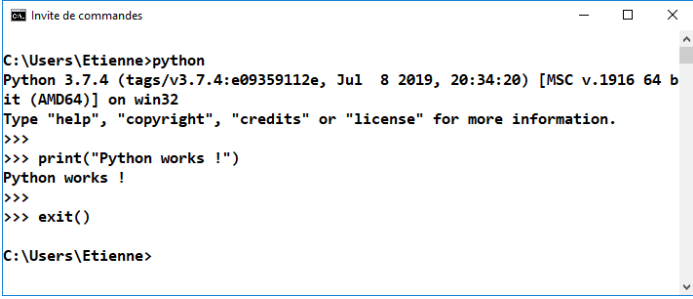
# Installation de Python

- 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>
```

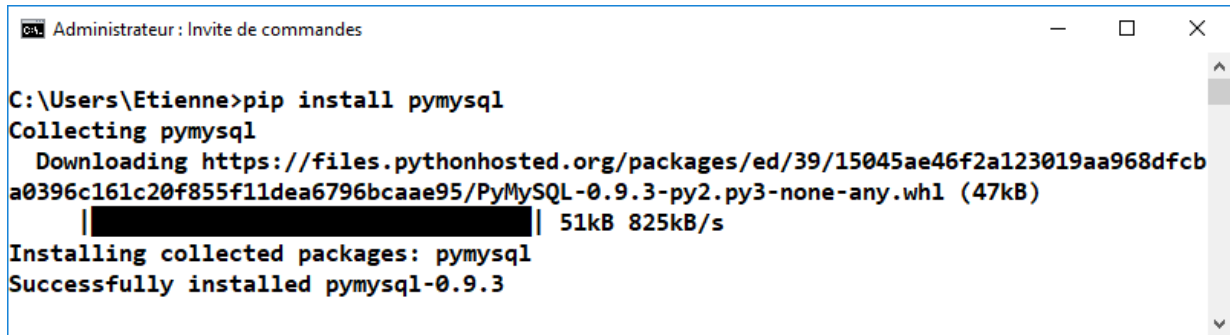
- Cet outil permet d'évaluer rapidement des portions de code
- La console est parfois directement intégrée dans les IDE pour Python

# Les librairies 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 <http://pypi.org> pour effectuer les téléchargements



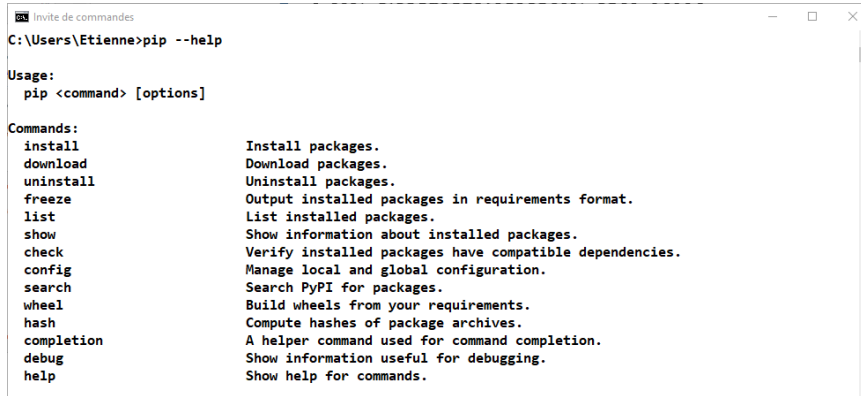
```
Administrateur : Invite de commandes

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

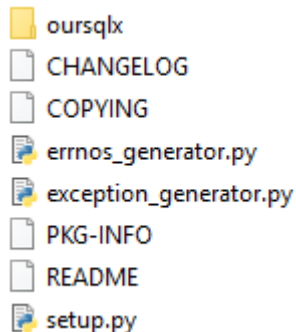
Usage:
  pip <command> [options]

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.

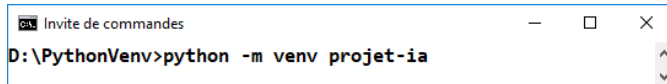


A file explorer view showing the contents of a source archive. The files listed are: oursqlx (folder), CHANGELOG (text file), COPYING (text file), errnos\_generator.py (Python script), exception\_generator.py (Python script), PKG-INFO (text file), README (text file), and setup.py (Python script).

```
python setup.py install
```

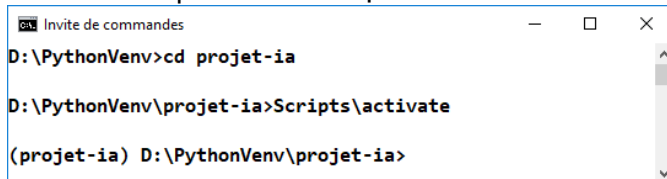
# Les environnements virtuels

- 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.



```
Invite de commandes
D:\PythonVenv>python -m venv projet-ia
```

- 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 environnement 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 associé à un plugin pour la prise en charge de Python
    - <http://www.pydev.org>

# Travaux Pratiques



[www.eni-service.fr](http://www.eni-service.fr)

# Travaux pratiques

- Exercice 1 : Installation de Python
  - Installer Python en prenant soin de faire en sorte qu'il soit utilisable par l'ensemble des utilisateurs de la machine. Veillez également à ce que la commande Python soit référencée dans le PATH
- Exercice 2 : Créer un environnement virtuel
  - Créer un environnement virtuel nommé «python-3-rf» dans le répertoire C:\PythonEnv.
  - Installer la librairie pymysql dans cet environnement.
- Exercice 3 : Installer un IDE
  - Télécharger et installer **PyCharm Community Edition**



# Module 3

## Les bases du langage

# Contenu du module

- Syntaxe du langage
- Instructions et délimiteurs
- Les blocs
- Les commentaires
- Les types de données
- Typage dynamique fort
- Les types de données simples
- Traitement des chaînes de caractères
- Les conversions de types
- Les types de données évoluées
- Les opérateurs
- Les structures de contrôle
- La structure conditionnelle
- Les structures itératives
- Entrée et sortie standards

# 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.

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

# 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 :

```
class A:  
  
    attribut = 0  
  
    def methode(self, param):  
        if param < self.attribut:  
            self.attribut = param + 1
```



# Les commentaires

- Le caractère « # » introduit une ligne de commentaire.
- Par ailleurs, il existe des commentaires spécifiquement dédié à la génération de la documentation du code.
  - Ils sont appelés **docstring**.
  - Ils sont exprimés entre `"""` (3 guillemets) ou `'''` (3 apostrophes)
- Ces commentaires se placent au tout début d'un bloc de module, de classe, de méthode ou de fonction.

# Les commentaires de documentation

```
class A:
    """Documentation de la classe.
    Il est recommandé de préciser le rôle et les objectifs
    de cette classe dans ce type de commentaire
    """

    attribut = 0

    def methode(self, param):
        """Documentation de la méthode.
        On décrit également ses paramètres et le retour produit le cas échéant.
        :param param: Signification du paramètre.
        :return: Type de retour de la méthode ou fonction.
        """
        if param < self.attribut:
            self.attribut = param + 1
```

```
>>> help(A)
Help on class A in module Module3:

class A(builtins.object)
 | Documentation de la classe.
 | Il est recommandé de préciser le rôle et les objectifs de cette classe dans ce type de commentaire.
 |
 | Methods defined here:
 |
 | methode(self, param)
 |     Documentation de la méthode.
 |     On décrit également ses paramètres et le retour produit le cas échéant.
 |     :param param: Signification du paramètre.
 |     :return: Type de retour de la méthode ou fonction.
 |
 | -----
 | Data descriptors defined here:
 |
 | __dict__
 |     dictionary for instance variables (if defined)
 |
 | __weakref__
 |     list of weak references to the object (if defined)
 |
 | -----
 | Data and other attributes defined here:
 |
 | attribut = 0
```

# 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**

```
>>> entier = 12
>>> type(entier)
<class 'int'>
>>> reel = 56.36
>>> type(reel)
<class 'float'>
>>> booleen = True
>>> type(booleen)
<class 'bool'>
>>> chaine = "Une chaine de caractères"
>>> type(chaine)
<class 'str'>
```

# Autres types de données

- Les types évolués

- Ensembles (tuple)

- Immutables !

- Listes (list)

- Mutables

- Dictionnaires (dict)

- Tableaux associatifs

```
>>> ensemble = ("un", "deux", "trois")
>>> type(ensemble)
<class 'tuple'>
>>> liste = ["un", "deux", "trois"]
>>> type(liste)
<class 'list'>
>>> dico = {"un": 1, "deux": 2, "trois": 3}
>>> type(dico)
<class 'dict'>
```

- Le type spécial

- None

- Par exemple un retour nul de fonction

```
>>> def fonction(a, b):
...     if b != 0:
...         return a / b
...
>>> resultat = fonction(10, 0)
>>> print(resultat)
None
```

# Typage dynamique fort

- Typage dynamique
  - Le typage d'une variable est réalisé lors de son affectation.
  - Le type peut donc changer.

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> a = "Texte"
>>> type(a)
<class 'str'>
```

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

```
>>> a = 10
>>> b = "Texte"
>>> c = a + b
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Traitement des chaines de caractères

- Les chaines de caractères, comme tous les types de données, sont des objets. A ce titre, une chaine de caractères possède un certain nombre de méthodes internes permettant leur manipulation.
- Les chaines de caractères s'écrivent entre guillemets ou apostrophe.
- Elles peuvent aussi être exprimées sur plusieurs lignes ! Ce sont les docstring évoquées précédemment.

```
>>> chaine1 = "Une chaine de caractères"  
>>> chaine2 = "Une autre chaine de caractères"  
>>> chaine3 = """  
... Ceci est une chaine  
... sur plusieurs lignes  
... """
```

# Manipulation des chaines de caractères

- Longueur d'une chaine
  - `len(chaine)`
- Gestion de la casse
  - `chaine.lower()`
  - `chaine.upper()`
- Appartenance
  - `mot in chaine`
- Remplacement
  - `chaine.replace("a", "A")`
- Découpage
  - `chaine.split()`

# Formatage des chaines de caractères

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

```
>>> "Le prix des %s" % "abricots"
'Le prix des abricots'
>>> "Le prix des %s est de %s euros" % ("abricots", 6)
'Le prix des abricots est de 6 euros'
>>> "Le prix des %(fruit)s est de %(montant)s euros" % {"fruit": "abricots", "montant": 6}
'Le prix des abricots est de 6 euros'
```

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

```
>>> "Le prix des {}".format("abricots")
'Le prix des abricots'
>>> "Le prix des {} est de {} euros".format("abricots", 6)
'Le prix des abricots est de 6 euros'
>>> "Le prix des {1} est de {0} euros".format(6, "abricots")
'Le prix des abricots est de 6 euros'
>>> "Le prix des {fruit} est de {montant} euros".format(montant=6, fruit="abricots")
'Le prix des abricots est de 6 euros'
```



# Les conversions de types

- Avec le typage fort de Python, il est parfois nécessaire de devoir convertir le type d'une donnée dans un autre.
- Exemple sur de la saisie de données au clavier avec la fonction `input()` :

```
>>> nombre1 = input("Saisissez un premier nombre : ")
Saisissez un premier nombre : >? 23
>>> nombre2 = input("Saisissez un second nombre : ")
Saisissez un second nombre : >? 96
>>> resultat = nombre1 + nombre2
>>> print("Le résultat de l'addition est {}".format(resultat))
Le résultat de l'addition est 2396
```

- Les données collectées par `input()` le sont sous forme de chaîne de caractères. L'opérateur `+` agit donc comme opérateur de concaténation

# Les fonctions de conversion

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

```
>>> nombre1 = int(input("Saisissez un premier nombre : "))
Saisissez un premier nombre : >? 23
>>> nombre2 = int(input("Saisissez un second nombre : "))
Saisissez un second nombre : >? 96
>>> resultat = nombre1 + nombre2
>>> print("Le résultat de l'addition est {}".format(resultat))
Le résultat de l'addition est 119
```

# Les types de données évoluées

- Les types de données évoluées de Python permettent de gérer des collections de valeurs, multi types, selon plusieurs approches.
- Les ensembles (Tuples)
  - Collections de valeurs associées à un indice numérique ordonné à partir de zéro. La collection **ne peut pas être modifiée** après sa création.
- Les listes
  - Collections de valeurs associées à un indice numérique ordonné à partir de zéro. La collection **peut être modifiée** après sa création.
- Les dictionnaires
  - Collections de valeurs associées à une clé, on maîtrise le type de la clé. Dans d'autres langages, on parle de tableau associatif ou encore de table de hachage. La collection **peut être modifiée** après sa création.

# Les ensembles (tuple) : création & déclaration

- Création d'un tuple

```
>>> un_tuple = ("un", "deux", "trois")
>>> type(un_tuple)
<class 'tuple'>
```

- Les parenthèses ne sont pas obligatoires (mais elles contribuent à la lisibilité du code)

```
>>> un_tuple = "un", "deux", "trois"
>>> type(un_tuple)
<class 'tuple'>
```

- Les tuples permettent les affectations multiples (en retour de fonction par exemple).

```
>>> a, b, c = ("un", "deux", "trois")
>>> a
'un'
>>> b
'deux'
>>> c
'trois'
```

# Les ensembles (tuple) : accès et manipulation

- Il est possible d'accéder aux éléments individuels d'un tuple en utilisant leurs indices

```
>>> un_tuple[0]
'un'
```

- Il est également possible d'accéder à une plage de valeurs !

```
>>> un_tuple[0:2]
('un', 'deux')
```

```
>>> un_tuple[1:3]
('deux', 'trois')
```

```
>>> un_tuple[:2]
('un', 'deux')
```

```
>>> un_tuple[1:]
('deux', 'trois')
```

- Par contre, un tuple n'est pas modifiable !

```
>>> un_tuple[1] = "DEUX"
```

```
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

# Les listes (list) : création & déclaration

- Création d'une liste

```
>>> une_liste = ["one", "two", "three", "four"]
>>> type(une_liste)
<class 'list'>
```

- Contrairement aux tuples, les listes sont modifiables. Il est possible d'ajouter, de modifier, de supprimer un élément ou bien d'étendre la liste avec une autre.

```
>>> une_liste.append("five")
>>> une_liste
['one', 'two', 'three', 'four', 'five']
>>> une_liste[2] = "THREE"
>>> une_liste
['one', 'two', 'THREE', 'four', 'five']
>>> del une_liste[1]
>>> une_liste
['one', 'THREE', 'four', 'five']
>>> une_liste.remove("four")
>>> une_liste
['one', 'THREE', 'five']
>>> une_liste.extend(["six", "seven", "eight"])
>>> une_liste
['one', 'THREE', 'five', 'six', 'seven', 'eight']
```

# Les listes (list) : accès & manipulation

- Les listes se manipulent tout comme les tuples, en utilisant la même syntaxe.

```
>>> une_liste[2]
'five'
>>> une_liste[1:4]
['THREE', 'five', 'six']
>>> une_liste[2:]
['five', 'six', 'seven', 'eight']
>>> une_liste[:4]
['one', 'THREE', 'five', 'six']
>>> une_liste[-1]
'eight'
```

- Il est évidemment possible de parcourir une liste (tout comme un tuple d'ailleurs !). La syntaxe Python est toute naturelle.

```
>>> for element in une_liste:
...     print(element)
...
one
THREE
five
six
seven
eight
```

# Les dictionnaires (dict) : création & déclaration

- Les dictionnaires sont des collections dans lesquelles les valeurs sont associées à des clés alphanumériques.
- Création d'un dictionnaire

- "un" est la clé du premier élément, et 1 sa valeur.

```
>>> un_dico = {"un": 1, "deux": 2, "trois":3, "quatre":4 }
>>> type(un_dico)
<class 'dict'>
>>> un_dico
{'un': 1, 'deux': 2, 'trois': 3, 'quatre': 4}
```

- Un dictionnaire est modifiable

```
>>> un_dico["cinq"] = 5
>>> un_dico
{'un': 1, 'deux': 2, 'trois': 3, 'quatre': 4, 'cinq': 5}
>>> del un_dico["trois"]
>>> un_dico
{'un': 1, 'deux': 2, 'quatre': 4, 'cinq': 5}
```



# Les dictionnaires (dict) : accès & manipulation

- Accéder à un élément d'un dictionnaire

- Deux méthodes :

```
>>> un_dico["quatre"]
```

```
4
```

```
>>> un_dico.get("quatre")
```

```
4
```

- Le parcours d'un dictionnaire se fait, par défaut, sur les clés.

```
>>> for element in un_dico:
```

```
...     print(element)
```

```
...
```

```
un
```

```
deux
```

```
quatre
```

```
cinq
```

# Les dictionnaires (dict) : accès & manipulation

- Il est également possible de faire un parcours sur les valeurs :

```
>>> for element in un_dico.values():  
...     print(element)  
...  
1  
2  
4  
5
```

- Ou bien un parcours récupérant les paires clés/valeurs sous forme de tuples ! (grâce à la méthode `items()` des dictionnaires). La syntaxe est alors triviale :

```
>>> for cle, valeur in un_dico.items():  
...     print(cle, "=", valeur)  
...  
un = 1  
deux = 2  
quatre = 4  
cinq = 5
```

# 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	$x ** y$
//	Reste 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
==	Égal	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 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
    - Elle 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 non vérifiée

```
>>> a = 5
>>> if a > 7:
...     a = a + 1
...
>>> a
5
```

## Condition simple vérifiée

```
>>> a = 9
>>> if a > 7:
...     a = a + 1
...
>>> a
10
```

# La structure conditionnelle : Conditions élaborées

## Condition et son contraire

```
>>> a = 20
>>> if a > 5:
...     a = a + 1
... else:
...     a = a - 1
...
>>> a
21
```

## Condition et alternative

```
>>> a = 5
>>> if a > 5:
...     a = a + 1
... elif a == 5:
...     a = a + 2
... else:
...     a = a - 1
...
>>> a
7
```

# La structure conditionnelle : Expression de la condition

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

```
>>> if a >= 0 and a <=20:  
...     a = a + 1  
...
```

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

```
>>> if 0 <= a <= 20:  
...     a = a + 1  
...
```

# 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

- En anglais, while signifie « Tant que ».
  - Le principe est donc d'écrire une condition, qui, tant qu'elle est vérifiée, permettra d'itérer.

```
>>> while i < 5:  
...     print("Itération en cours ... i vaut", i)  
...     i = i + 1  
...  
Itération en cours ... i vaut 0  
Itération en cours ... i vaut 1  
Itération en cours ... i vaut 2  
Itération en cours ... i vaut 3  
Itération en cours ... i vaut 4
```

- 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, ...

```
>>> chaine = "Bonjour"
>>> for lettre in chaine:
...     print(lettre)
...
B
o
n
j
o
u
r
```

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

# Les structures itératives : interruption & reprise

- 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

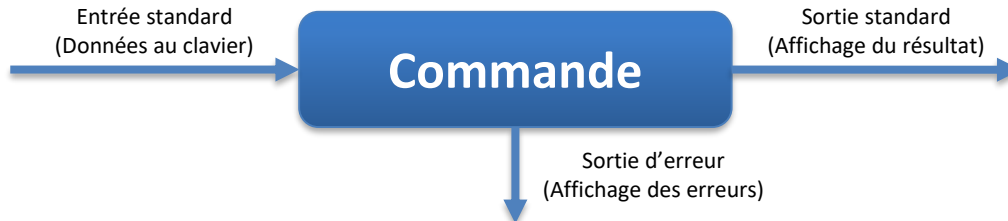
```
>>> liste = ["un", "deux", "trois", "quatre", "cinq", "six"]
>>> for element in liste:
...     if element == "quatre":
...         break
...     print(element)
...
un
deux
trois
```

## Reprise de boucle

```
>>> liste = ["un", "deux", "trois", "quatre", "cinq", "six"]
>>> for element in liste:
...     if element == "quatre":
...         continue
...     print(element)
...
un
deux
trois
cinq
six
```

# 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.





# 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.

```
>>> saisie = input("Merci d'indiquer votre prénom : ")
Merci d'indiquer votre prénom : >? Etienne
>>> print("Bonjour", saisie)
Bonjour Etienne
```

- 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.

# 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)`
- 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`).

```
>>> nom = "DUPONT"
>>> prenom = "Robert"
>>> print("Bonjour", prenom, nom)
Bonjour Robert DUPONT
```

- Chaque valeur est séparé 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`.

# 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.



# Travaux Pratiques



# Travaux pratiques

- Exercice 1 : Un premier programme Python

- L'objectif de ce premier programme est de construire un jeu dans lequel l'utilisateur doit deviner un nombre aléatoirement généré entre 1 et 99. Le jeu se poursuit tant que le nombre n'est pas trouvé. On donnera un indice au joueur pour chaque mauvaise réponse, en précisant si le nombre à deviner est plus grand ou plus petit que sa saisie au clavier.
- AIDE : Pour générer le nombre aléatoirement, on utilisera l'instruction suivante :

```
from random import randint  
nombre = randint(1, 100)
```

- Exercice 2 : Variation sur le jeu

- En reprenant l'exercice précédent, modifier le code de sorte à ce que l'utilisateur ne dispose que de 10 tentatives pour trouver le nombre à deviner.



# Module 4

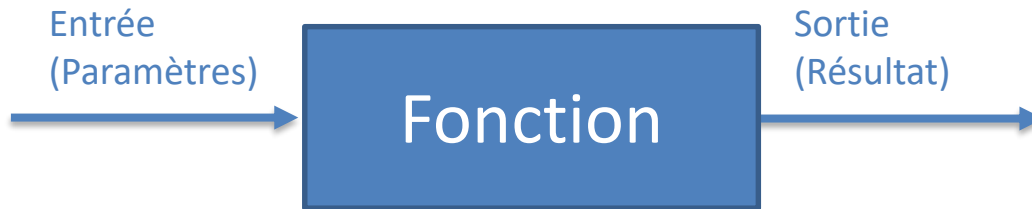
## Les fonctions

# Contenu du module

- Utilité des fonctions
- Déclaration d'une fonction
- Utilisation d'une fonction
- Les paramètres
- Le retour d'une fonction
- Les paramètres optionnels
- Les paramètres en nombre variable
- Les paramètres nommés

# 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
  - Eventuellement des paramètres sous forme de variables. Ils seront exprimés entre les parenthèses.
- Ceci constitue la **signature** de la fonction.
- Un bloc est ensuite créé pour l'**implémentation** de la fonction (son code interne)

The diagram shows a Python function declaration with annotations. The code is: `def addition(operande1, operande2):` followed by an indented block `resultat = operande1 + operande2` and `return resultat`. Annotations with arrows point to parts of the code: 'Identificateur' points to 'addition', 'Paramètres' points to '(operande1, operande2)', and 'Retour de la fonction' points to 'return resultat'.

```
def addition(operande1, operande2):  
    resultat = operande1 + operande2  
    return resultat
```

Identificateur → Paramètres

Retour de la fonction ↑

# 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, ...


```
somme = addition(56, 89)
```

- 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, il doivent recevoir des valeurs au moment de l'appelle de la fonction !

```
somme = addition(56, 89)
```



```
def addition(operande1, operande2):  
    resultat = operande1 + operande2  
    return resultat
```

- 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 addition(operande1, operande2):  
    resultat = operande1 + operande2  
    return resultat
```

# 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.
  - **ATTENTION** : De manière évidente, les paramètres optionnels doivent se situer en fin de signature !

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

```
somme = ma_fonction()           # Erreur ! param1 doit être fourni  
somme = ma_fonction(10)        # Renvoi 10  
somme = ma_fonction(10, 20)    # Renvoi 30  
somme = ma_fonction(10, 20, 30) # Renvoi 60
```

# 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.
  - Un seul paramètre est alors exprimé.
  - Son nom est préfixé d'une \*.
  - Le paramètre est alors exploitable sous forme d'une liste.

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

```
moy = moyenne(15, 10, 12, 17, 8)  
print(moy)          # Affiche 12.4
```

# Les paramètres nommés

- Certaines fonctions, notamment des fonctions intégrées au langage, utilisent des paramètres en nombre variable et des paramètres optionnels à la suite.
- C'est le cas par exemple de la fonction `print()` dont voici la signature (extrait de la documentation officielle de Python) :
  - `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
- Cette fonction prend en effet en paramètre autant de valeurs que nécessaire à afficher (`*objects`), ces valeurs sont séparées par un espace (`sep`) et chaque ligne est terminée par un saut de ligne (`end`).

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

- Dans ce cas, 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='')
un#deux#trois
```

# Travaux Pratiques



[www.eni-service.fr](http://www.eni-service.fr)



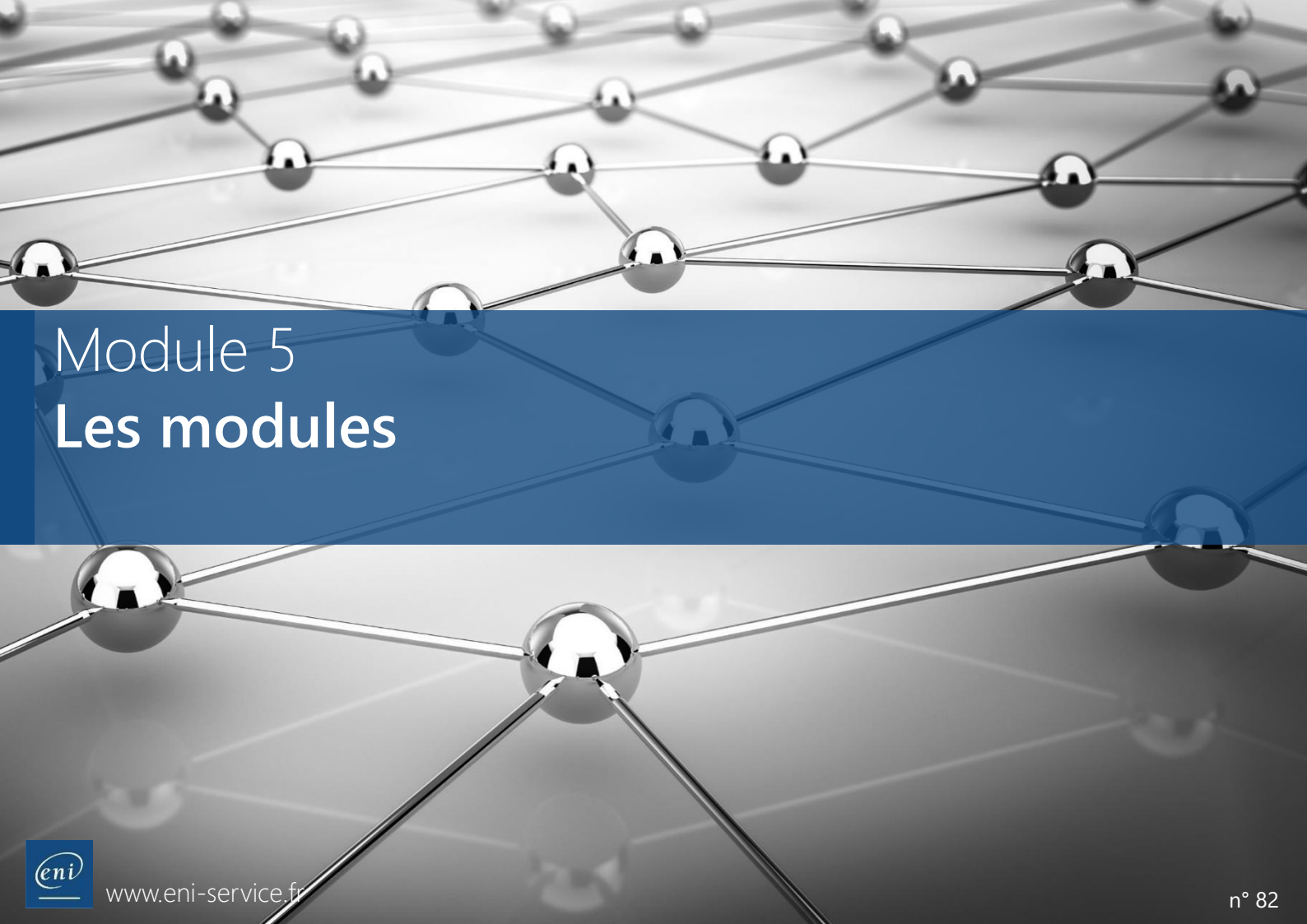
# Travaux pratiques

## ■ Exercice 1 : Calculatrice

- Dans un fichier Python appelé `calculatrice.py`, définir les fonctions suivantes et y implémenter les traitements mathématiques appropriés :
  - `addition(a, b)`
  - `soustraction(a, b)`
  - `multiplication(a, b)`
  - `division(a, b)`
    - Attention une division par zéro est impossible ! Il faut donc gérer ce cas pour éviter une erreur à l'exécution, en affichant un message d'erreur à l'utilisateur par exemple.
  - `moyenne(*notes)`

## ■ Exercice 2 : Utiliser les fonctions de la calculatrice

- Après avoir définis les fonctions il s'agit de les utiliser pour vérifier leur comportement. A la fin du fichier `calculatrice.py`, appeler les fonctions les unes après les autres et vérifiez les résultats produits.



# Module 5

## Les modules

# Contenu du module


- Utilité de la structuration en module
- Le nommage des modules
- Organisation en packages
- Importation de modules
- Importation de fonctionnalités de modules
- Le chemin de localisation des modules : PYTHONPATH

# 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.
- Au-delà de ça, les modules permettent également de spécifier un **espace de nommage** contextuel pour les éléments qu'il contiennent.

# Le nommage des modules

- Les modules Python peuvent prendre deux formes :
  - Un fichier d'extension `.py`
    - Le nom du module est le nom du fichier Python sans son extension.
  - Un dossier contenant un fichier `__init__.py`
    - Le nom du module est le nom du dossier
- Exemples pour un module « calculatrice » :
  - Fichier

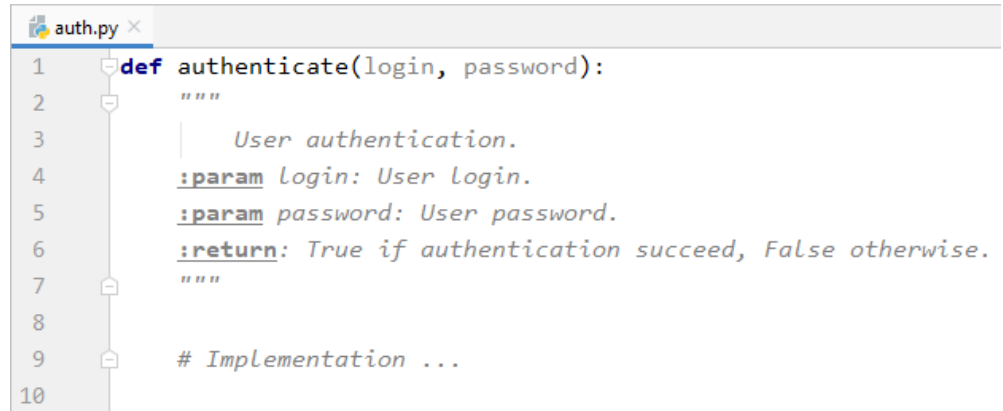
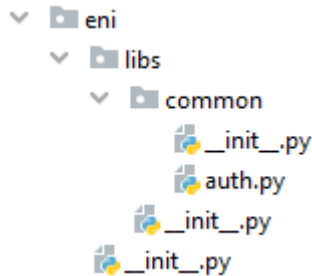
 calculatrice.py

- Dossier

▼  calculatrice  
 \_\_init\_\_.py

# Organisation en packages

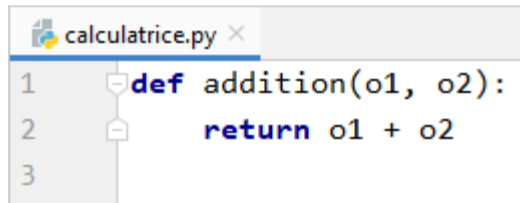
- Les modules peuvent se trouver stockés dans une arborescence de **packages**. Dans ce cas, les packages font partis du nom du module !
  - Un package est un dossier, ni plus, ni moins.
  - L'objectif est d'assurer l'unicité des noms des modules !



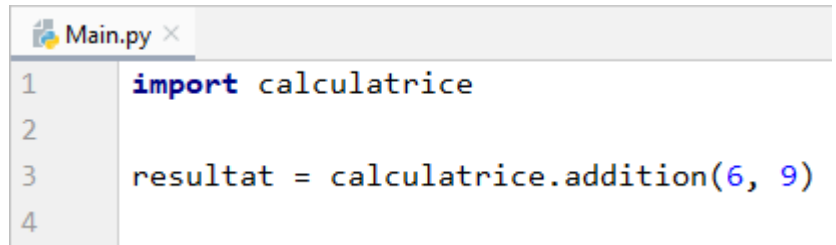
- L'accès à la fonction `authenticate()` se fera par :
  - `eni.libs.common.auth.authenticate(...)`

# Importation de modules

- Lorsque des fonctionnalités présentes dans des modules doivent être utilisées, il y a deux manières de les rendre disponibles et accessibles pour les autres modules :
  - Rendre visible un module
  - Importer une fonctionnalité
- Rendre visible un module se fait simplement en utilisant le mot clé `import`. Il ne permet cependant pas de s'affranchir de spécifier le nom du module lors de l'utilisation d'une de ses fonctionnalités.



```
calculatrice.py x
1  def addition(o1, o2):
2      return o1 + o2
3
```



```
Main.py x
1  import calculatrice
2
3  resultat = calculatrice.addition(6, 9)
4
```

# Importation de fonctionnalités de modules

- Il est également possible d'importer une fonctionnalité unique d'un module.
  - 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.

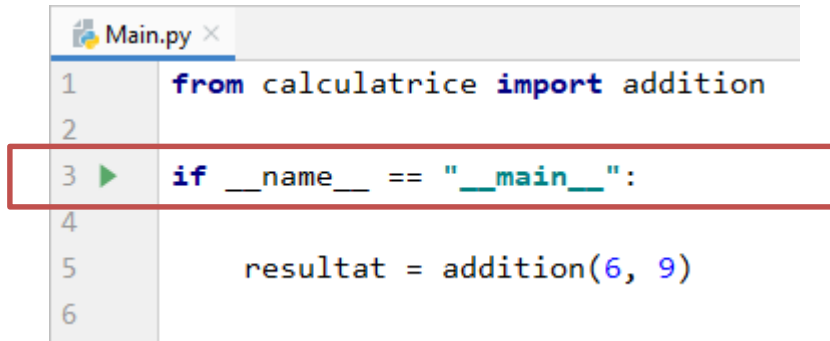
```
calculatrice.py x
1  def addition(o1, o2):
2      return o1 + o2
3
```

```
Main.py x
1  from calculatrice import addition
2
3  resultat = addition(6, 9)
4
```



# Le module principal

- En Python, contrairement à d'autres langages, il n'y a pas spécifiquement de point d'entrée de programme.
  - A l'instar du `main()` du C, du C++ ou du Java !
- Il est cependant courant de trouver une structure de module simulant cette notion.
  - Il s'agit de tester si le module est invoqué directement par l'interpréteur Python.
    - Dans ce cas là, le module porte le nom spécial `__main__`



```
1 from calculatrice import addition
2
3 if __name__ == "__main__":
4
5     resultat = addition(6, 9)
6
```

# 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
>>> 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
>>> sys.path.insert(0, "/home/etienne/scripts")
```



## Module 6

# La programmation orientée objet

# Contenu du module

- Historique
- Les concepts
- La notation UML
- Les méthodologies associées à l'objet
- La notion d'objet
- Le concept de Classe
- Déclaration de classe
- Déclaration des attributs et des méthodes
- Création d'objet : instanciation
- La visibilité des objets
- L'encapsulation
- L'héritage
- Les usages de l'héritage
- L'héritage multiple et ses conséquences
- Le polymorphisme : redéfinition de méthodes

# Historique

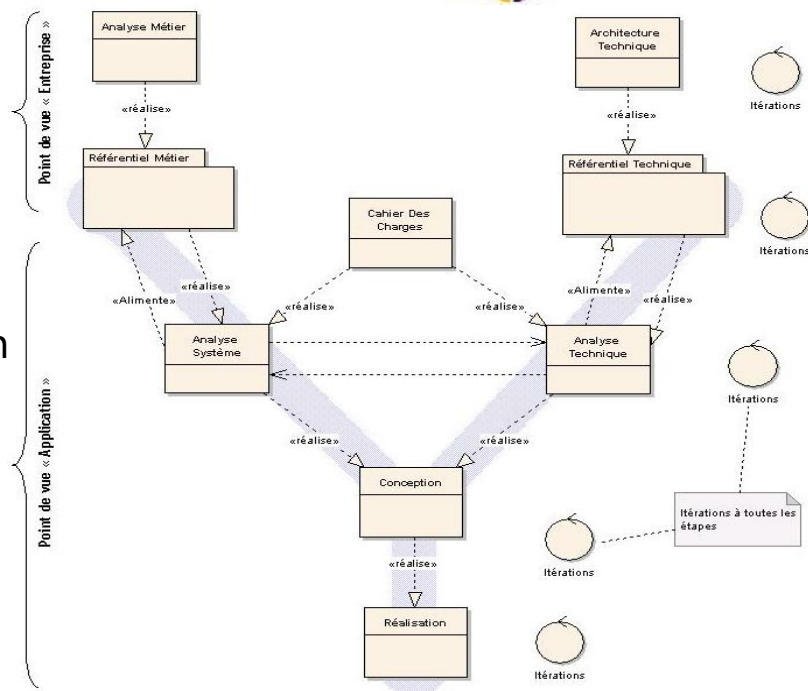
- Vient du monde de la simulation
- Les grammaires procédurales sont finalement impropres à la modélisation des problèmes...
- Besoin d'exprimer un environnement de façon naturelle !
- Décomposition d'un sujet à comprendre, analogue au fonctionnement de l'esprit humain :
  - Diviser pour mieux régner !
  - Le "monde" est composé de "choses", "d'objets« .
  - Ces objets interagissent entre eux.
- Conception orientée objet : une approche descriptive
- Premiers langages objet : SIMULA, Smalltalk, C++, Eiffel

# Les concepts

- Encapsulation
  - L'objet forme un tout.
  - L'objet appartient à une nature (type) qui ne peut changer.
  - L'objet est garant de son état.
- Héritage
  - L'objet peut être une évolution d'un autre, plus général.
- Polymorphisme
  - Des objets de natures différentes peuvent réagir au même message.

# UML

- Unified Modeling Language
- Langage graphique ≠ méthode
- Boîte à outils
  - Vues
  - Diagrammes
  - Modèles d'éléments
- Reprend, étend plusieurs autres travaux de modélisation : Booch, Rumbaugh et Ivar Jacobson
- Association avec une méthode
  - OMT
  - RUP
  - 2TUP



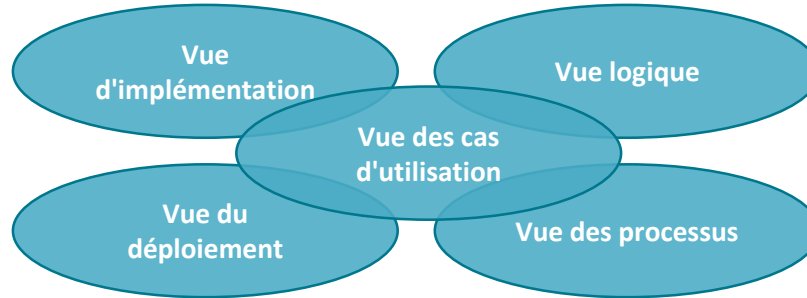


# Les méthodologies associées à UML

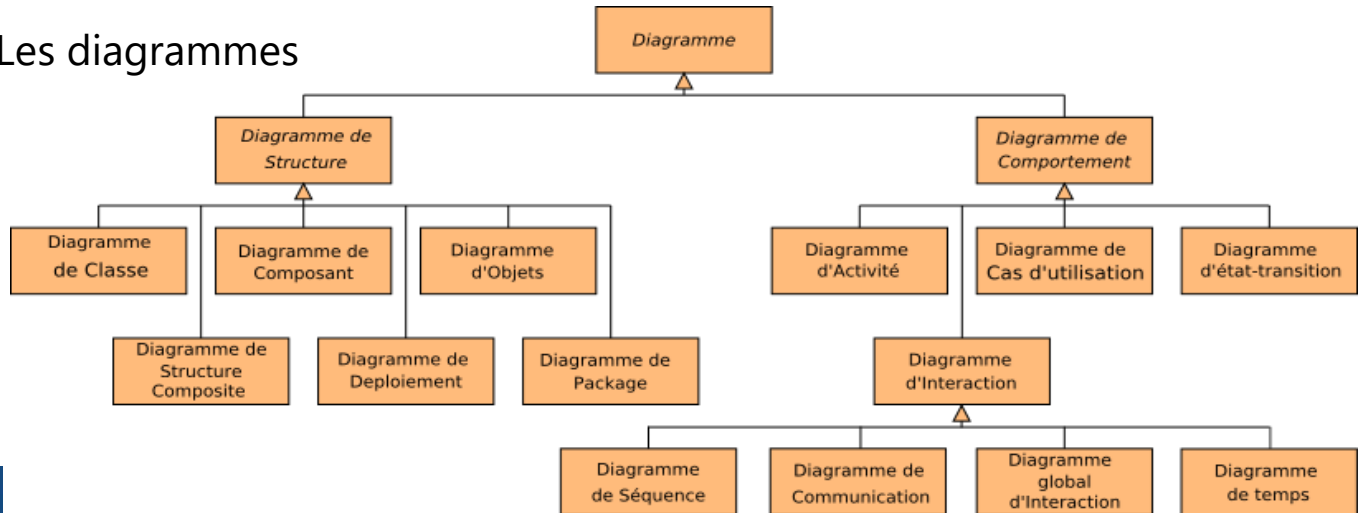
- L'apport de l'objet avait pour objectif de maîtriser la complexité des développements logiciels.
  - Il était fondamental que les méthodologies suivent !
- Le Cycle en V utilisé pendant des années montrait des limites liées à l'absence de livraisons régulières et de retour de la part des clients.
  - Une validation n'intervenant qu'en fin de projet n'est plus suffisante quand les fonctionnalités sont de plus en plus nombreuses !
    - Une livraison régulière s'impose !
- Les méthodologies « Agiles » ont fait leur apparition.
  - Construction itérative et incrémentale des produits
  - On se centre sur les besoins utilisateurs et les fonctionnalités à livrer
  - Les risques sont gérés
- « Unified Process » comme ancêtre...
  - Aujourd'hui, Scrum, Kanban, ...

# UML

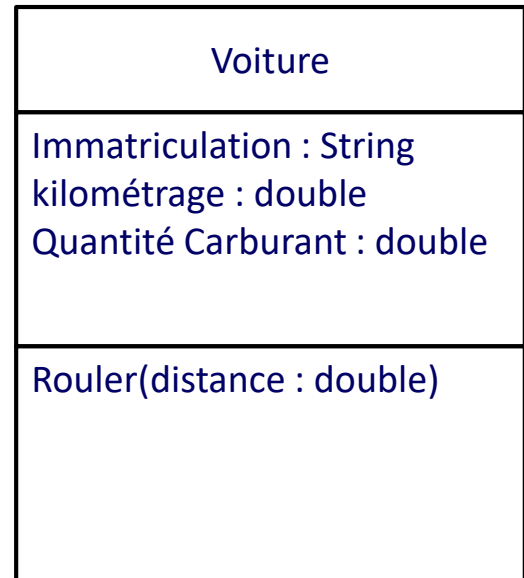
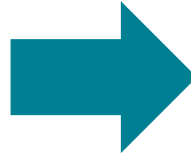
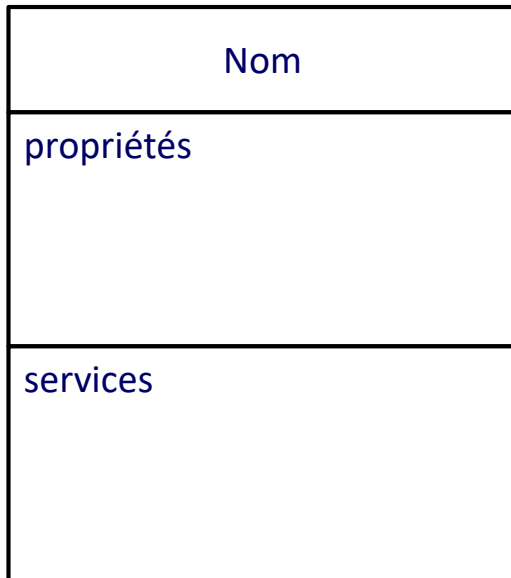
- Les vues



- Les diagrammes



# UML : l'élément classe



# La classe Python

- Mot-clé `class`
- Marque la nature des objets issus de cette classe
- Un objet ne peut changer de nature au cours de sa vie
- La déclaration s'effectue au sein d'un bloc
- Il n'y a pas d'ordre dans la description

```
class Personne:  
    nom = ""  
    prenom = ""
```

# Attributs et méthodes

- Membres d'instance
  - Propres à chaque objets !
- Pas d'ordre dans les déclarations.
- Attributs
  - Une structure de donnée
  - De 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

# Classe : Exemple Python

Personne
nom : str prenom : float
parler(phrase : str)

```
class Personne:  
    nom = ""  
    prenom = ""  
  
    def parler(self, phrase):  
        print(self.prenom + " " + self.nom + " dit " + phrase)
```

# Attributs

- Attributs
  - Déclaration directement dans la classe
  - Variable dont la durée de vie est l'instance
  - Global à l'instance
- `self`
  - Mot-clé de résolution d'espace de nommage
  - Parfois, il y a des ambiguïtés entre paramètre de méthode et attribut d'instance
  - Règle :
    - La variable désignée : celle déclarée dans le bloc le plus interne

```
class Personne:
```

```
    nom = ""
```

```
    prenom = ""
```

```
def set_nom(self, nom):
```

```
    self.nom = nom
```

# Les méthodes

- Factorisation, réutilisation du code
- Modularité
- Comme une fonction !
  - Mais dans le contexte d'un objet !
- Fonction vs. Méthode
  - La simple déclaration de l'objet en cours !
    - `self`



# Les méthodes

- Déclaration

- Fonction

```
def ma_fonction(self, paramètre1, paramètre2):  
    // code source de la fonction  
    return valeur;
```

- Procédure

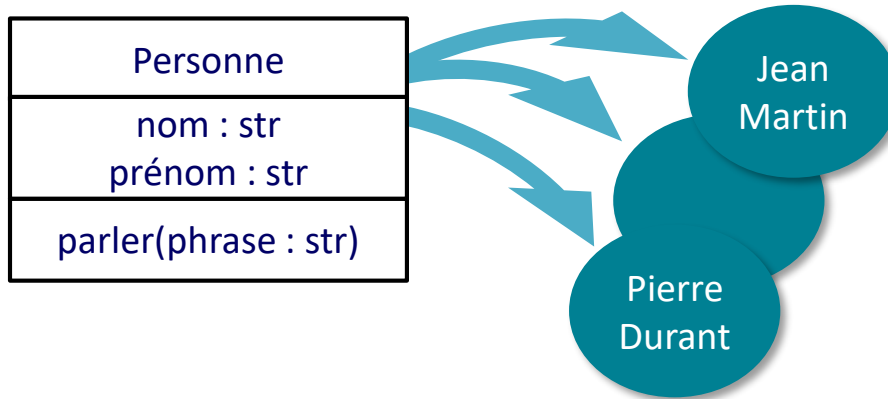
```
def ma_procédure(self, paramètre1, paramètre2):  
    // code source de la procédure
```

# Classe et instance

- La classe
  - Concept
  - Vision statique
  - Ne "fait" rien
  - ⇒ Nécessité de rendre actif tout ce qui est décrit
  - ⇒ Utiliser le point d'entrée du programme qui réalise des traitements
- L'instance
  - Instance = objet = exemplaire  
ex : La voiture immatriculée "123 ABC 45"
  - Vision dynamique
  - Indépendance des "vies" des instances

# Le constructeur

- Mécanisme qui utilise la classe comme une fabrique à objets



- Procédure d'initialisation
  - Même identifiant que la classe
  - Pas de type de retour
  - Construit et initialise des objets (instances)
  - Pont entre le concept et la réalité

# Le constructeur (suite)

```
if __name__ == "__main__":
```

```
    p = Personne()
    p.parler("Bonjour")
```

```
class Personne:
```

```
    nom = ""
    prenom = ""
```

```
    def __init__(self):
        self.nom = "Dupont"
        self.prenom = "Michel"
```

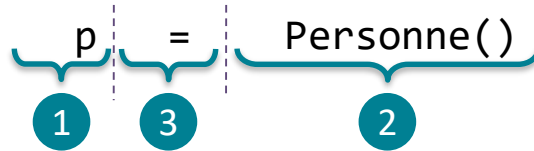
```
    def parler(self, phrase):
        print(self.prenom + " " + self.nom + " dit " + phrase)
```

Appel sur la classe :  
C'est à la classe qu'on demande un  
nouvel exemplaire

Initialisation sur l'instance

# Le constructeur (suite)

- Examen de l'instruction de construction



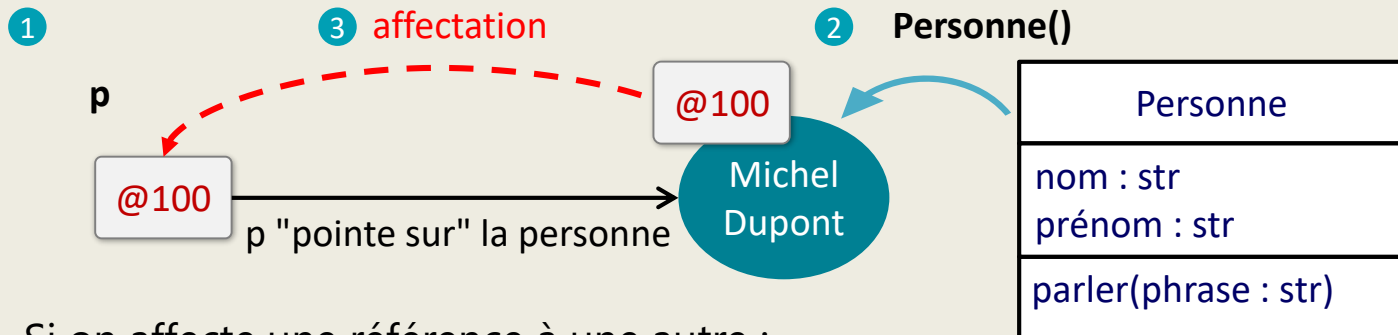
- 1 Création d'une "variable de manipulation d'une Personne"  
Pas de création de Personne. Valorisée à None
- 2 Création d'une nouvelle instance de Personne. Cela se traduit par une réservation de mémoire à une adresse (par ex: @100)
- 3 Affectation de la Personne nouvellement créée à la variable p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



**"p" n'est pas l'objet, mais un moyen de le manipuler**

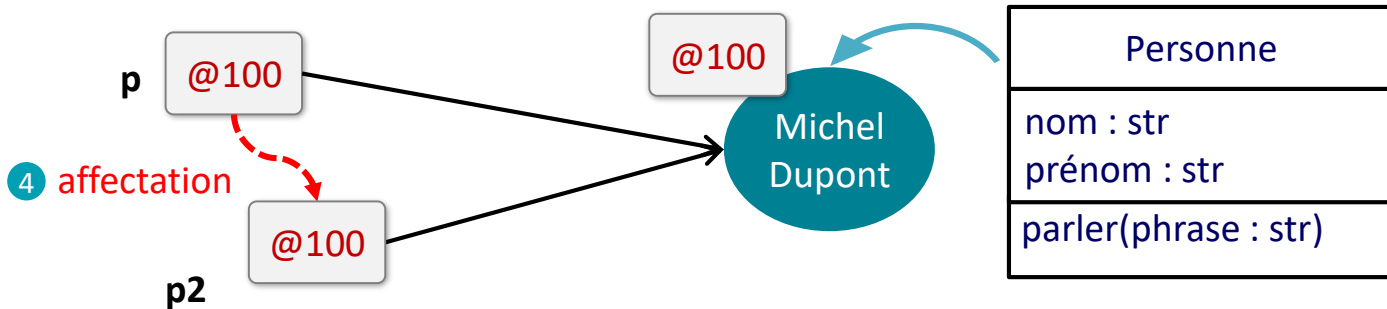
# Le constructeur (suite)

## Référence aux objets



Si on affecte une référence à une autre :

4 `p2 = p`



**p et p2 manipulent le même objet !**

# Le constructeur (suite)

- Transmission d'informations d'initialisation des attributs
  - Via les paramètres
  - Possible comme sur les méthodes
  - Permet de proposer des constructeurs plus complets

```
class Personne:

    nom = ""
    prenom = ""

    def __init__(self, nom, prenom): ←
        self.nom = nom
        self.prenom = prenom

    def parler(self, phrase):
        print(self.prenom + " " + self.nom + " dit " + phrase)

if __name__ == "__main__":

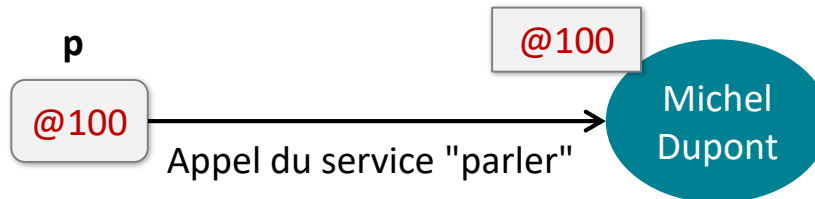
    p = Personne("Dupont", "Michel") ←
    p.parler("Bonjour")
```

# Accès aux membres

- 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
- En réponse, l'objet déclenche un comportement

```
if __name__ == "__main__":
```

```
    p = Personne("Dupont", "Michel")  
    p.parler("Bonjour")
```





# Un tout cohérent

- Exemple, que penser de :

```
if __name__ == "__main__":  
  
    v = Voiture(25000, 23)  
    v.rouler(100)  
    print(v.kilometrage)  
    v.kilometrage = -12  
  
class Voiture:  
  
    kilometrage = 0  
    qteCarburant = 0  
  
    def __init__(self, kilometrage, qteCarburant):  
        self.kilometrage = kilometrage  
        self.qteCarburant = qteCarburant  
  
    def rouler(self, distance):  
        self.kilometrage += distance  
        self.qteCarburant -= distance * 0.07
```

- La voiture doit rouler correctement
  - Modification de l'état de la voiture sans contrôle par celle-ci
- ⇒ Incohérence ⇒ Cas de violation d'encapsulation

# Un tout cohérent : L'encapsulation

⇒ L'objet doit se protéger

- Expression de la visibilité des membres
  - Privés : invisible à l'extérieur de la classe, même en lecture
  - Publiques : manipulable de l'extérieur de la classe, même en écriture
- Démarche : sauf bonne raison, on cache tout
- Si on veut autoriser la lecture, il faut passer par un service dédié : le *getter* (accesseur en lecture)
- Idem pour l'écriture : *setter* (accesseur en écriture).
  - Pour faire des contrôles de validité des valeurs
- En Python
  - Tout est considéré **public par défaut** !
  - La notion d'éléments privés peut être exprimée en préfixant les membres avec `__` (double underscore)

# Un tout cohérent : L'encapsulation

```
Main.py x
2
3 if __name__ == "__main__":
4
5     v = Voiture(25000, 23)
6     v.rouler(100)
7     print(v.__kilometrage)
8     v.__kilometrage = -12 } Erreurs !!!
9
Run: Main x
Traceback (most recent call last):
  File "C:/Users/Etienne/PycharmProjects/Support Python/Main.py", line 7, in <module>
    print(v.__kilometrage)
AttributeError: 'Voiture' object has no attribute '__kilometrage'
Process finished with exit code 1
```

```
class Voiture:

    __kilometrage = 0
    __qteCarburant = 0

    def __init__(self, kilometrage, qteCarburant):
        self.__kilometrage = kilometrage
        self.__qteCarburant = qteCarburant

    def rouler(self, distance):
        self.__kilometrage += distance
        self.__qteCarburant -= distance * 0.07

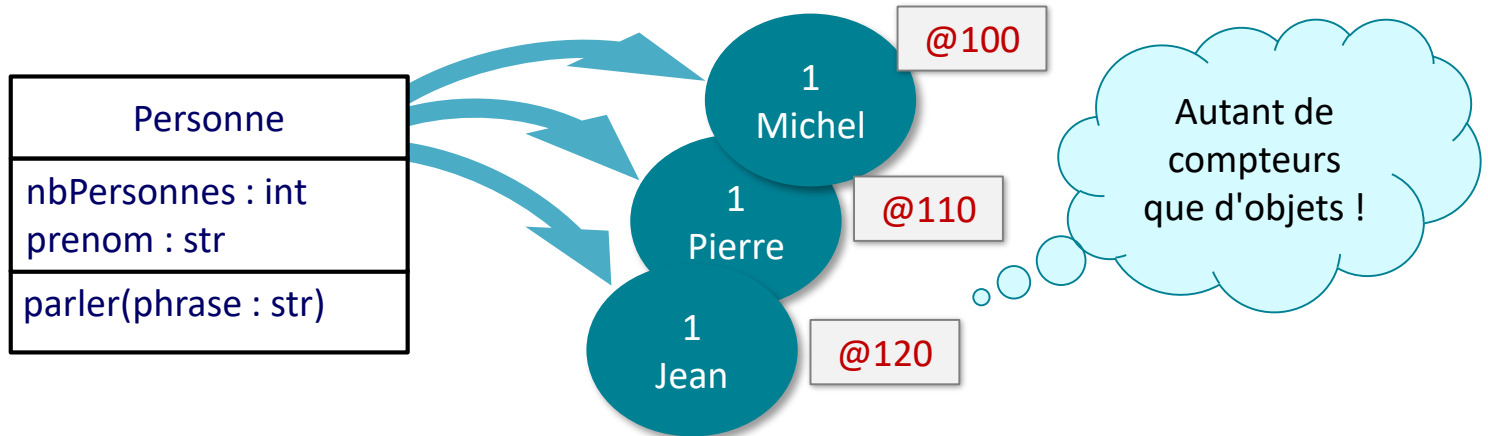
    def getQteCarburant(self):
        return self.__qteCarburant

    def setQteCarburant(self, qteCarburant):
        if 0 < qteCarburant < 35:
            self.__qteCarburant = qteCarburant

    def getKilometrage(self):
        return self.__kilometrage
```

# Membres de classe

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



```
def __init__(self, prenom):  
    self.prenom = prenom  
    self.nbPersonnes += 1
```

# Membres de classe (suite)

- ⇒ Parfois des données ne relèvent pas de l'instance mais de la classe
  - Membre de classe  $\neq$  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

# Membres de classe en Python

- Accès
  - Les membres sont accédés à partir du nom de la classe et non plus à partir du nom de l'instance !
- Déclaration
  - 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 `@`
  - Il n'y pas de syntaxe particulière pour exprimer les attributs de classe !
    - Tout se fait sur la manière dont ils sont accédés !!

# Membres de classe (suite)

```
class Personne:
```

```
    # Attribut de classe : pas de spécificité dans la déclaration !  
    nbPersonnes = 0
```

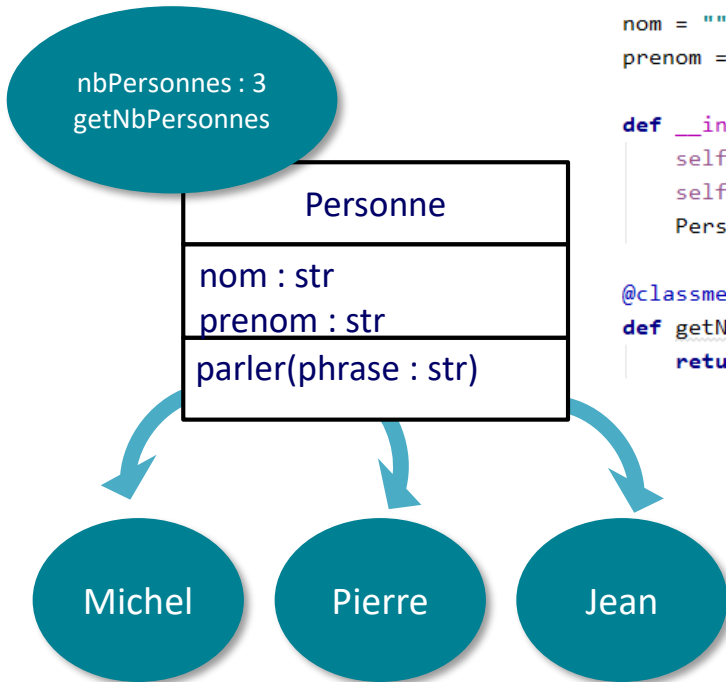
```
    nom = ""  
    prenom = ""
```

```
    def __init__(self, nom, prenom):  
        self.nom = nom  
        self.prenom = prenom  
        Personne.nbPersonnes += 1
```

```
    @classmethod
```

```
    def getNbPersonnes(cls):  
        return cls.nbPersonnes
```

```
    # cls fait référence à la classe !  
    # Ou bien : return Personne.nbPersonnes
```



# Les collaborations

- Association
  - Forme générale de collaboration
  - Forme de collaboration la plus faible
  - Connaissance par importation
  - Pas de référence permanente : pas de relation conteneur-contenu

```
class Personne:
```

```
    nom = ""
```

```
    prenom = ""
```

```
    def __init__(self, nom, prenom):
```

```
        self.nom = nom
```

```
        self.prenom = prenom
```

```
    def conduire(self, voiture):
```

```
        voiture.rouler()
```

La référence à voiture est résolue  
à l'exécution.  
Elle doit être connue ! (importation)



# Les collaborations (suite)

- Relation conteneur-contenu
  - Construire des objets complexes à partir d'objets simples
  - Différentes sortes de collaborations  
Ex : quelle différence sémantique entre "une voiture contient des passagers" et "une voiture contient un moteur"
- Composition
  - Relation "forte"
  - Les cycles de vies des objets sont liés
    - La construction du conteneur implique la construction du contenu
    - La destruction du conteneur provoque la destruction du contenu
    - La destruction du contenu dégrade le conteneur
  - Exemple : Voiture - Moteur
  - Notion de "possession"

# Les collaborations (suite)

- Agrégation
  - Relation conteneur-contenu "faible"
  - Les cycles de vies des objets ne sont pas liés
    - La construction du conteneur est indépendante de celle du contenu
    - La destruction du conteneur ne provoque pas la destruction du contenu
    - Le contenu est facultatif dans le conteneur
  - Exemple : Voiture - Passager
- Précautions relatives à la composition
  - Charge au programmeur de synchroniser les cycles de vie pour la composition, dans le constructeur
  - Le conteneur ne donne pas accès au contenu  $\Rightarrow$  violation d'encapsulation
  - Le problème est caduque pour l'agrégation car le conteneur ne possède pas le contenu

# Les collaborations (suite)

```
class Voiture:
```

```
    __conducteur = None
```

```
    __moteur = None
```

```
def __init__(self, kilometrage, qteCarburant, puissance):
```

```
    self.__kilometrage = kilometrage
```

```
    self.__qteCarburant = qteCarburant
```

```
    self.__moteur = Moteur(puissance)           # Composition
```

```
def setConducteur(self, conducteur):
```

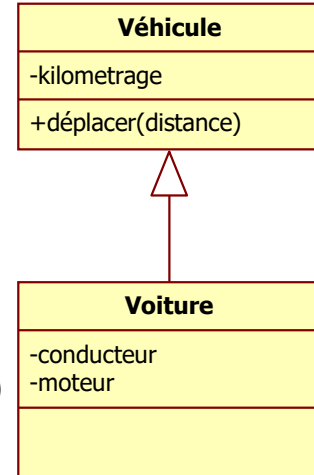
```
    self.__conducteur = conducteur           # Agrégation
```

# L'héritage en Python

- Concept issu d'observations du réel
    - L'esprit humain classe naturellement les concepts : classification du règne animal, etc.
    - Besoin de rattacher un concept précis à un autre plus général  
ex : une voiture est un véhicule, une moto est un véhicule
  - Avantages
    - Gain de temps en ne rappelant pas les qualités déjà admises  
ex : une moto est un véhicule motorisé à 2 roues
    - Abstraction par l'assimilation de plusieurs concepts à un seul  
ex : Tous les véhicules peuvent se déplacer
- ⇒ Permet de se focaliser sur l'essentiel, tout en conservant la nature spécifique des objets.

# L'héritage en Python (suite)

- Expression UML :
  - Flèche : sens de généralisation
  - Voiture spécialise Véhicule
  - Véhicule est une généralisation de Voiture
- Héritage au sens patrimonial
  - Ex : une voiture est un véhicule
  - Une instance de Voiture contient tout d'une instance de Véhicule
  - Les membres statiques ne sont pas hérités (ne sont pas dans l'instance)
- Expression Python sur la déclaration de la sous-classe



```
class Vehicule:
```

```
    __kilometrage = 0
```

```
    def deplacer(self, distance):
        |     self.__kilometrage += distance
```

```
class Voiture(Vehicule):
```

```
    __conducteur = None
```

```
    __moteur = None
```

# L'héritage en Python (suite)

- Permet d'avoir une vision générale d'objets spécifiques
  - Ex : les véhicules se déplacent (peu importe leur nature spécifique)
  - Simplification : on ne s'intéresse qu'au minimum nécessaire (abstraction)

```
liste = []  
liste.append(Voiture())  
liste.append(Moto())  
liste.append(Velo())  
for vehicule in liste:  
    vehicule.deplacer(10)
```

- Classe mère de toutes les autres : `object`

# Construction d'objets dans l'héritage

- Constructeur
  - Appel automatique au super-constructeur par défaut
  - Si redéfinition d'un constructeur, appel explicite au super-constructeur nécessaire : `SuperClasse.__init__(self, ...)`
  - On réutilise le bloc d'initialisation

```
class Vehicule:
```

```
    __kilometrage = 0
```

```
    def __init__(self, kilometrage):  
        self.__kilometrage = kilometrage
```

```
    def deplacer(self, distance):  
        self.__kilometrage += distance
```

```
class Voiture(Vehicule):
```

```
    __qteCarburant = 0
```

```
    def __init__(self, kilometrage, qteCarburant):  
        Vehicule.__init__(self, kilometrage)  
        self.__qteCarburant = qteCarburant
```

# L'héritage multiple

- Python, contrairement à d'autres langages, permet l'héritage multiple.
  - Le fait qu'une classe puisse hériter de plusieurs super-classes.

```
>>> class A:
...     def a(self):
...         pass
...
>>> class B:
...     def b(self):
...         pass
...
>>> class C(A,B):
...     pass
...
>>> dir(C)
['__class__', '__delattr__', '__dict__', ... '__weakref__', 'a', 'b']
```

- Le principal risque de l'héritage multiple étant le conflit de méthodes !
  - Des méthodes de même nom dans des super-classes différentes.



# Le polymorphisme

- Partager des méthodes communes n'est pas suffisant  
ex : une voiture ne se déplace pas comme un bateau
- ⇒ Classes spécifiques = comportement spécifique
- ⇒ Mais garder une forme d'appel homogène  
ex : tous les Véhicules savent se déplacer sur une distance, mais pas de la même façon

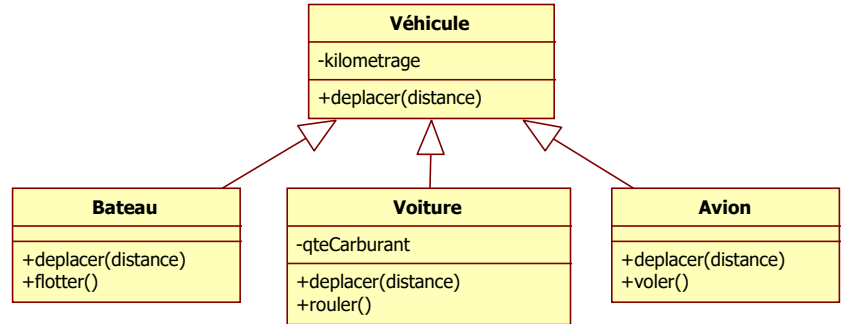


**Polymorphisme**  
**Un message homogène, pour des comportements différenciés**

# Le polymorphisme (suite)

- On veut manipuler un tableau

```
for vehicule in liste:  
    vehicule.deplacer(10)
```



- Mais que `deplacer()` soit spécifique selon que l'instance soit de type `Voiture` ou autre !

```
class Vehicule:  
  
    __kilometrage = 0  
  
    def __init__(self, kilometrage):  
        self.__kilometrage = kilometrage  
  
    def deplacer(self, distance):  
        self.__kilometrage += distance
```

```
class Voiture(Vehicule):  
  
    __qteCarburant = 0  
  
    def __init__(self, kilometrage, qteCarburant):  
        Vehicule.__init__(self, kilometrage)  
        self.__qteCarburant = qteCarburant  
  
    def deplacer(self, distance):  
        Vehicule.deplacer(self, distance)  
        self.__qteCarburant -= distance * 0.07
```

# Le polymorphisme (suite)

- Redéfinition de méthode
  - Réécriture d'une méthode qui possède exactement la même signature que la méthode héritée
  - Les 2 méthodes restent présentes dans l'instance
  - La nouvelle méthode masque l'ancienne
- Lors de l'appel sur une instance, c'est la dernière définie qui est immédiatement disponible
- La méthode redéfinie peut appeler la méthode héritée via la super-classe ⇒ Réutilisation de code

```
class Voiture(Vehicule):  
  
    ...  
  
    def deplacer(self, distance):  
        Vehicule.deplacer(self, distance)  
        self.__qteCarburant -= distance * 0.07
```

# Travaux Pratiques



[www.eni-service.fr](http://www.eni-service.fr)



# Module 7

## Concepts avancés

# Contenu du module

- La classe 'object'
- Les méthodes spéciales des objets
- Redéfinition des méthodes spéciales
- La surcharge des opérateurs
- La gestion des erreurs avec les exceptions
- Génération d'exceptions
- Traitement des exceptions : try ... except ...
- Stratégies de gestion et propagation
- Cas particulier de la gestion des ressources : with

# La classe 'object'

- En Python, la classe object est la super-classe de toutes les classes.

```
class Vehicule:  
    ...
```

- Revient à :

```
class Vehicule(object):  
    ...
```

- C'est d'ailleurs cette deuxième syntaxe qui devait être utilisée en Python 2 !
- 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__']
```

# Les méthodes spéciales des objets

- Les méthodes spéciales, héritées de object, permettent le bon fonctionnement du modèle objet de Python.
- Parmi ces méthodes, `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__` et `__ne__` sont des comparateurs.

Opérateur	Méthode
<code>==</code>	<code>__eq__</code>
<code>&gt;=</code>	<code>__ge__</code>
<code>&gt;</code>	<code>__gt__</code>
<code>&lt;=</code>	<code>__le__</code>
<code>&lt;</code>	<code>__lt__</code>
<code>!=</code>	<code>__ne__</code>

- A noter également la méthode `__str__` permettant de retourner la représentation d'un objet sous forme d'une chaîne de caractères.



# Redéfinition des méthodes spéciales

- La redéfinition de ces méthodes spéciales au sein des classes permet aux objets de ces classes de s'approprier spécifiquement les opérateurs de comparaison.
  - On pourra par exemple définir dans la méthode `__eq__` de la classe `Voiture`, les critères qui font que deux objets de type `Voiture` sont identiques !

```
class Voiture:

    __immatriculation = ""
    __marque = ""
    __modele = ""

    def __init__(self, immatriculation, marque, modele):
        self.__immatriculation = immatriculation
        self.__marque = marque
        self.__modele = modele

    def __eq__(self, other):
        if isinstance(other, Voiture) and self.__immatriculation == other.__immatriculation:
            return True
        return False
```

# La surcharge des opérateurs

- Sur le même modèle que précédemment, les opérateurs arithmétiques sont également associés à des méthodes.
  - Qui peuvent, elle aussi, être redéfinie dans les classes.

Opérateur	Méthode	Signification
+	<code>__add__</code>	Addition
-	<code>__sub__</code>	Soustraction
*	<code>__mul__</code>	Multiplication
/	<code>__truediv__</code>	Division
//	<code>__floordiv__</code>	Division entière
%	<code>__mod__</code>	Modulo
**	<code>__pow__</code>	Puissance

# Les exceptions

- Besoin de gérer les erreurs

```
def diviser(numerateur, denominateur):  
    return numerateur/denominateur
```

← Problème possible !

```
q = diviser(n,d)
```

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

```
int diviser(double numerateur, double denominateur, double *resultat){  
    if(denominateur != 0){  
        *resultat = numerateur/denominateur; //pb écarté  
        return CODE_OK;  
    }  
    else{  
        return CODE_DIV_0;  
    }  
}  
  
code = diviser(n,d,&q);  
if (code==CODE_OK) {...}
```

- Un mal nécessaire...

# Les exceptions (suite)

- 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

```
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
```

- 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.

Schéma tiré de la documentation officielle de Python

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

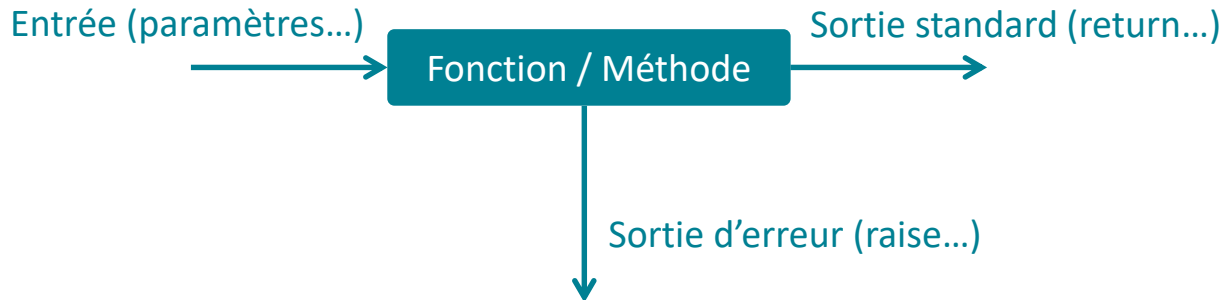
# Les exceptions : origine de l'erreur

```
class Personne:

    __nom = None

    def set_nom(self, nom):
        if nom is None or len(nom) == 0:
            raise Exception("Le nom doit être rempli.")
        self.__nom = nom
```

- Test sur les préconditions
- `raise` : envoi d'un objet `Exception` au premier bloc de traitement possible
- La définition du déclenchement d'exception laisse « sortir » l'objet d'erreur par un autre canal de sortie.





# Les exceptions : traitement de l'erreur

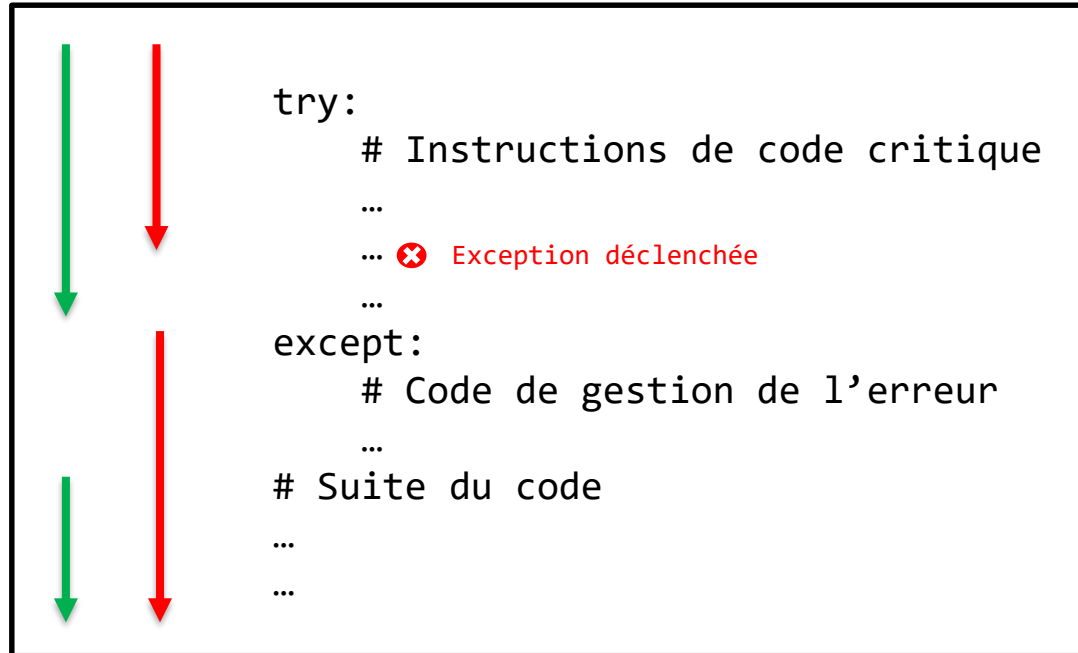
- Gérer les exceptions est presque obligatoire
  - Toute exception non gérée correctement conduira à la fin de l'exécution de l'interpréteur Python !
- Responsabilités du traitement :
  - Réalisation correcte
    - Pas d'erreur
    - Ou interception et traitement alternatif : `try: ... except: ...`
  - Ou alerte via un canal approprié
    - Sortie d'erreur : l'objet d'exception remonte dans le code appelant.
    - Via un message à l'utilisateur

# 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.

 Exécution normale

 Exécution avec exception





# 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:
```

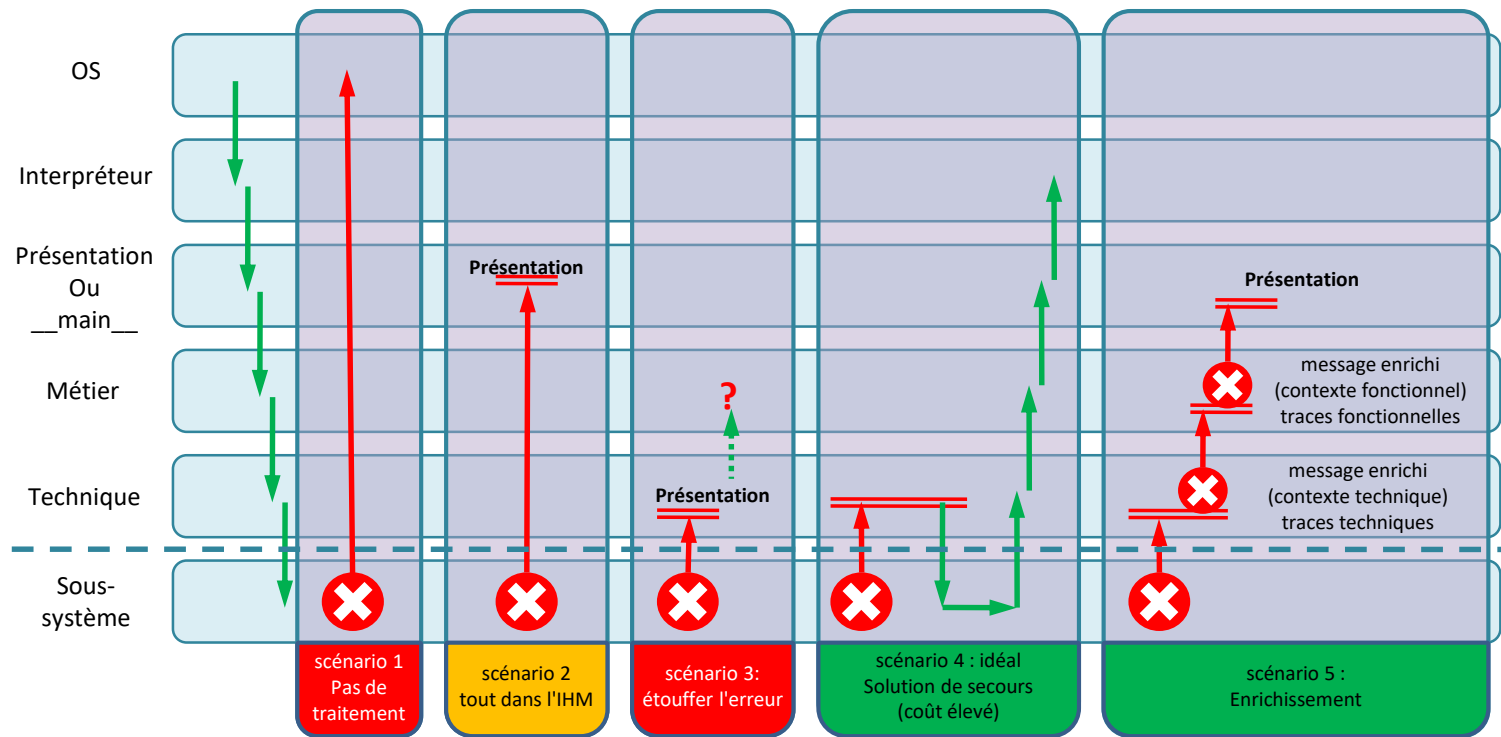
```
    ...  
except TypeError as e:
```

```
    ...  
except:
```

```
    ...
```

« e » est une référence  
sur l'objet d'exception

# Les exceptions : différents scénarios



# 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

# Les exceptions : bonnes pratiques

- Qu'est-ce qu'un cas d'erreur ?
  - Les pré-conditions ne sont pas respectées
  - Le sous-système de la fonctionnalité est défaillant
- Qu'est-ce qui n'est pas un cas d'erreur ?
  - Un scénario applicatif qui doit refuser une fonctionnalité  
ex : Interdiction de connexion suite à un mot de passe erroné
  - Le `try ... except ...` ne doit pas remplacer le `if ... else ...`

# 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 ExceptionFonctionnelle("message fonctionnel");
finally:
    ressource.close()
```

- Dans ce cas, **except** n'est pas obligatoire !

# La structure with

- La structure `with` de Python permet de simplifier l'écriture d'un bloc `try ... finally ...`
  - Dans ce cas, il n'y a pas de gestion de l'erreur et l'exception remonte au code appelant.
  - La fermeture de la ressource allouée dans le `with` est alors automatique !!

- Exemple :

```
ressource = UneRessource()
try:
    ressource.traiter()
finally:
    ressource.close()
```

- Pourra être écrit :

```
with UneRessource() as ressource:
    ressource.traiter()
```

# Travaux Pratiques





# Module 8

## La bibliothèque standard



# Contenu du module

- Introduction
- Interaction avec le système d'exploitation
- Collecter des informations sur le système
- Interagir avec les processus
- Manipuler les fichiers et les répertoires
- Travailler avec les chemins d'accès

# 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>

# 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 ce 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', 'AIT', '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`
  - `os.uname()`
- De trouver la liste des variables d'environnement :
  - `list(os.environ.keys())`
- D'obtenir les valeurs de constantes types du système :
  - `os.curdir`, `os.pardir`
  - `os.sep`, `os.pathsep`, `os.linesep`
    - Les commandes précédentes sous Windows :

```
>>> os.curdir, os.pardir
('.', '..')
>>> os.sep, os.pathsep, os.linesep
('\\', ';', '\r\n')
>>>
```

# Interagir avec les processus

- Python permet également, grâce au module `subprocess`, de lancer des commandes du système d'exploitation.
- Lancer une commande et afficher son résultat :
  - `retour = subprocess.call(['ls', '-l'])`
  - La variable `retour` contiendra le code de retour de la commande (0 si tout va bien)
- 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

# 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 « Builtin » il n'est pas nécessaire de devoir importer un quelconque module pour pouvoir les utiliser.
- Ouverture d'un fichier :

```
>>> with open('fichier.txt') as f:
...     pass    # Travailler avec le fichier
...
>>>
```

- La variable `f` est alors un descripteur sur le fichier. Ce descripteur sera fermé automatiquement fermé à la fin du bloc `with`.
- Par défaut, la fonction `open()` ouvre un fichier en lecture au format texte. Pour l'ouvrir différemment, il faut ajouter une séquence de caractère en deuxième paramètre de la fonction, afin de préciser son mode d'ouverture.

# La fonction open()

- La fonction d'ouverture de fichiers possède la signature (simplifiée) 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)
- Les autres modes d'ouverture sont synthétisés dans le tableau ci-dessous.

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).

# Lire et écrire dans un fichier

- Une fois le fichier ouvert (dans le bon mode !), il est possible d'écrire et de lire à l'intérieur.

- Ecriture :

```
>>> lignes = ['ligne 1', 'ligne 2', 'ligne 3']
>>> with open('fichier.txt', 'w') as f:
...     f.writelines(lignes)
...
>>>
```

- La fonction `write()` permet d'écrire des octets.

- Lecture :

```
>>> with open('fichier.txt') as f:
...     lignes = f.readlines()
...
>>> print(lignes)
['ligne 1\n', 'ligne 2\n', 'ligne 3']
>>>
```

- La fonction `read()` permet d'écrire des octets.



# Travailler avec les chemins d'accès

- La manipulation des chemins avec le module de bas niveau `os`, est assez archaïque !
- Dans le cas où l'on veut gérer des chemins multiplateforme, il faut construire les chemins en tenant compte de la spécificité du séparateur de chemin en vigueur sur chaque système !
- Exemple :

```
>>> chemin = "rep1" + os.pathsep + "rep2" + os.pathsep + "rep3"
```

- Pour construire le chemin « rep1/rep2/rep3 ».
- Peu lisible !
- De plus, il n'est pas possible de vérifier l'existence de ce chemin, ou encore de vérifier que les éléments sont bien des répertoires et non pas des fichiers.
- 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.
- La classe `Path` constitue le principal élément de ce module. Elle permet l'expression de chemins et leur manipulation.

```
>>> from pathlib import Path
>>> ici = Path('.')
```

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

```
>>> ici.exists()
True
>>> ici.is_dir()
True
>>> ici.is_file()
False
>>> ici.absolute()
WindowsPath('C:/Users/Etienne/PycharmProjects/Support Python')
```

# Manipuler des fichiers avec pathlib

- La classe Path du module pathlib offre les mêmes fonctions de manipulation de fichier que les primitives du langage.
  - `open()`, `read()`, `write()`, ...
- La manipulation de fichiers en lecture et écriture se fait donc « presque » comme précédemment.
  - La méthode s'appliquant à une référence de fichier, il n'est donc pas nécessaire de spécifier ce dernier en premier paramètre.

## Lecture de fichier

```
>>> from pathlib import Path
>>> chemin = Path() / 'data' / 'fichiers'
>>> fichier = chemin / 'mesdonnées.txt'
>>> if fichier.exists():
...     with fichier.open('r') as f:
...         lignes = f.readlines()
...         for ligne in lignes:
...             print(ligne, end='')
...
Ligne 1
Ligne 2
Ligne 3
>>>
```

## Écriture de fichier

```
>>> from pathlib import Path
>>> chemin = Path() / 'data' / 'fichiers'
>>> fichier = chemin / 'mesdonnées.txt'
>>> lignes = ["Ligne 1", "Ligne 2", "Ligne 3"]
>>> with fichier.open('w') as f:
...     for ligne in lignes:
...         f.write(ligne + '\n')
...
>>>
```

# Travaux Pratiques





Fin de la formation

# Programmer en Python

# Pour aller plus loin

- Les cours associés
  - T463-002 - Programmation Python - Niveau 2
  - T463-003 - Développement Web en Python avec le framework Django
  - T463-004 - Python - Développement d'IHM avec la librairie PyQt
  - T130-102 - Développer ses applications de hacking et forensic en Python

# Pour aller plus loin

- ENI Service sur Internet
  - Consultez notre site web [www.eni-service.fr](http://www.eni-service.fr)
    - Les actualités
    - Les plans de cours de notre catalogue
    - Les filières thématiques et certifications
  - Abonnez-vous à nos newsletters pour rester informé sur nos nouvelles formations et nos événements en fonction de vos centres d'intérêts.
  - Suivez-nous sur les réseaux sociaux
    -  Twitter : <http://twitter.com/eniservice>
    -  Viadeo : <http://bit.ly/eni-service-viadeo>

# Pour aller plus loin

- Notre accompagnement
  - Tous nos Formateurs sont également Consultants et peuvent :
    - Vous accompagner à l'issue d'une formation sur le démarrage d'un projet.
    - Réaliser un audit de votre système d'information.
    - Vous conseiller, lors de vos phases de réflexion, de migration informatique.
    - Vous guider dans votre veille technologique.
    - Vous assister dans l'intégration d'un logiciel.
    - Réaliser complètement ou partiellement vos projets en assurant un transfert de compétence.



# Votre avis nous intéresse

Nous espérons que vous êtes satisfait de votre formation.

Merci de prendre quelques instants pour nous faire un retour en remplissant le questionnaire de satisfaction.

Merci pour votre attention,  
et au plaisir de vous revoir prochainement.