Python

Ken CHEN

Télécom Paris

18/08-01/09 (2021)

Préambule

- Python est un langage *interprété*
- Programme *source* : une succession d'instructions (en texte ASCII) dans un fichier (avec l'extension `.py`)
- *Interpréteur* (`python`, `python3`, *etc.*) traduit le programme source en instructions machine.
- Etant un langage interprété, il est disponible sous de nombreux systèmes, dont notamment Windows et Unix.
- Python est devenu ces dernières années un langage de programmation très populaire
- Il est utilisé par de plus en plus de métiers (en particulier dans **Data Science** grâce à un nombreux phénoménal de **bibliothèques** (`package`) spécialisés

Caractéristiques de Python

- Langage interprété : simplicité, large disponibilité
- Modularité : existence de nombreux `module` et `package`
- Langage orienté **objet** (*OOP=Object Oriented Programming*).
- Voici une présentation *simpliste* du concept d'*objet* :
  - Un objet est un espace mémoire dûment identifié qui possède ses **propres** *data* et les traitements **spécifiques** à ces *data*.
  - La forme générique est généralement appelée `class` qui est constituée
    - des paramètres et variables appelées généralement `attributes`
    - des fonctions appelées généralement `method`
  - Exemple : On peut définir une `class Student` qui possède deux *attributes* : `nom, formation` et une *method* : `display` qui affiche, selon un certain format, les 2 attributes.

Objectif pédagogique de la présente série

- Il s'agit d'une **brève** introduction à Python
  - Il **NE** s'agit surtout **PAS** d'une présentation détaillée de Python
  - La partie dominante du volet "informatique" de la session est bien le langage C
- Objectifs visés
  - Une "révision" et de "prise de recul" des concepts et pratiques communs dans l'*art* et la technique de la programmation à travers la présentation de ce *yet another language*
    - Attention : Hypothèse : Langage C correctement assimilé
    - Présentation **rapide** avec un regard plus ciblé sur des points particuliers à Python
  - Avoir une première introduction à ce langage dont la popularité ne cesse de croître
  - Dans l'*idéal*, présenter le concept d'**Objet** (qui est absent dans C)

1er contact avec Python

- La séquence suivante est réalisée sous un *shell* Python invoqué par IDLE (*Integrated DeveLopment Environnement*) pour Python.

```
>>> print("Bonjour")
Bonjour
```

- On consigne généralement les codes Python dans un script. Voici un script “minimaliste” en Python
  - On écrit les lignes suivantes dans un fichier nommé `Bonjour.py`

```
# Ceci est un scrit elementaire en Python
print("Bonjour") # afficher "Bonjour"
```

  - Puis, on lance l'exécution de ce fichier
    - Soit à l'aide d'un IDE (par exemple IDLE) qui l'exécute dans un `shell`
    - Soit en invoquant l'interpréteur Python directement dans une `console` du système avec `python Bonjour.py`
  - Le résultat est l'affichage de "Bonjour" sous un `shell` ou `console`

Structure d'un programme en Python

- Remarque : Ce transparent est préparé pour être **relu** *plus tard*, lorsque les différents concepts et règles auront été présentes et illustrés
- Un script élémentaire en Python se présente comme un fichier avec l'extension `py` dans lequel se trouve une suite d'instructions.
- La plupart du temps, un programme en python s'organise comme suit
  - Les codes sont repartis dans plusieurs fichiers appelés `module` ;
  - Un script "fédérateur" implémente les traitements au niveau le plus élevé en s'appuyant sur les modules existants qu'il incorpore via `import` ;
    - Les modules "importés" sont placés en début du script.
    - Le **point d'entrée**, c'est-à-dire la 1ère fonction à être exécutée, est souvent identifié comme `main()`, même s'il n'y a aucune obligation.
- Tout module peut "importer" d'autres modules.

Commentaire, espace et indentation

- Les commentaires peuvent se rajouter sous 2 formes
  - Une chaîne de caractères qui débute avec # :
    - c'est très souvent une ligne *entière* ;
    - Il est également possible de la placer après toute instruction
  - Le couple """ (3 guillemets) qui encadre un commentaire qui est généralement **multi-lignes**.
- Les **espaces** sont utilisés pour tout besoin de délimitation : opérateurs, expressions,*etc.*.
- Une ligne vide est généralement insérée entre deux blocs d'instructions.
- Attention : Une particularité de Python est l'**indentation** qui se fait avec **quatre** espaces. C'est le moyen de réaliser l'imbrication des blocs d'instructions à différents niveaux

Identifiants, instructions

- Un *identifiant* (*identifier*) est une suite de caractères alphanumériques plus "_".
  - Attention : Les chiffres NE peuvent PAS être la 1ère lettre. Il faut éviter d'utiliser `"_"` comme 1ère lettre
  - Exemples : `monEcole, is_Good`
- Les séquences réservées ou particulières
  - Les mots-clés : `int, elif, while`, *etc.*.
  - Les opérateurs : `>, !=, ==`, *etc.*.
  - les séparateurs : espace, *tabulation*, *etc.*.
  - les valeurs numériques (`1.23`)
- Les chaîne de caractères sont encadrées par un couple de `'` **ou** de `"`
  - Exemple : `'abc', ''abc''`
- Une *instruction* (**statement**) est une unité logique d'exécution qui pourrait seulement être suivie par un commentaire (`# abc...`).
  - Exemple : `print(" Bonjour ") # affiche Bonjour`

Les variables de base

- Rappel : Une **variable** est un symbole correspondant à un segment de mémoire où l'on peut placer, à tout moment, un contenu sous forme binaire (valeur numérique, chaîne de caractères, *etc.*).
- Dans Python, il n'y a pas de *typage explicite* des variables
  - En réalité, chaque variable est gérée comme un *objet*.
- La déclaration d'une variable et son *initialisation* se font **en même temps**.
- C'est la nature de la *valeur initiale* qui détermine le *type* de la variable
- Les trois types de base sont
  - Entier (`int`) : Exemple : `i = 2`
  - Réel (`float`) : Exemple : `r = 3.4`
  - Chaîne de caractères (`str`) : Exemple : `s = "Voici"`
- Attention : Il n'y a pas (moyen) de déclaration **seule** d'une variable

Conversion de type (*type casting*)

- la fonction `int()` produit un entier
  - `print(int(3.9))` donne `3`
  - `print(int("3"))` donne `3`
  - Remarque : Dans le 2nd exemple, c'est un chiffre sous la forme d'une chaîne de caractères qui est converti en un (vrai) chiffre
    - Attention : Il faut que cette chaîne corresponde à un entier
- la fonction `float()` produit un réel
  - `print(float(3))` donne `3.0`
  - `print(float("3.9"))` donne `3.9`
  - Remarque : Dans le 2nd exemple, la chaîne de caractères peut représenter un réel ou même un *entier*
- La fonction `str()` produit une chaîne de caractère correspondant à la valeur numérique
  - `print(str(3))` donne `3`
  - `print(float(3.9))` donne `3.9`
- Il bien sûr possible de prendre une variable comme argument

Type de base : Compléments

- Les entiers peuvent être de longueur quelconque
- La précision des réels dépendent de l'implémentation.
- Il y a aussi des variables **booléennes** (traitée en interne comme des entiers)
  - Deux valeurs possible : `True` ou bien `False`
  - Exemple : `estUnEtreHumain = True`, `enHiver = False`
    - Il est possible d'utiliser `None` à la place de `False`
    - Exemple : `en2019 = None`
- La fonction `type` donne le `type` d'une variable
  - Exemple : L'exemple qui suit en donne une illustration avec une variable `int`

```
>>> a = 5
>>> type(a)
<class 'int'>
```

  - On remarque qu'il s'agit de la `class int`.
- On détaillera la notion de la `class` plus loin. A ce stade, disons que toute variable est en réalité un objet.

Type de base : Compléments

- Attention : Il n'y a pas de déclaration au préalable d'une variable. Son type est déterminée par la nature de sa valeur. Il est possible qu'une *même* variable change de *type* suite à l'affectation d'une autre valeur

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> x = 3.0
>>> type(x)
<class 'float'>
>>> x = "3.0"
>>> type(x)
<class 'str'>
```

- Il y a aussi le type `complex` que nous ne traitons pas
  - Exemple : `c = 1 + 3j`

Les listes

- Une **liste** dans Python est un type de données combinées qui regroupe une séquence de valeurs encadrée par `[ ]`.
  - Exemple : `weekend = ["samedi", "dimanche"]`, `a = [1, 2, 3]`
- Les éléments d'une liste n'ont pas besoin d'être homogènes
  - Exemple : `we2 = ["samedi", 6, "dimanche", 3.5]`
  - De ce fait, par rapport à C, ça remplit les rôles joués par respectivement les *tableaux* et les *structures*.
- On peut effectuer diverses opérations sur les listes, dont notamment la **concaténation**
  - `weekend + a` donne `["samedi", "dimanche", 1, 2, 3]`

la fonction `print`

- L'affichage se fait à travers la fonction `print`
- Nous en donnons quelques unes parmi les plus usuelles, en partant des valeurs `i=3`, `r=3.9` et `s= "Voici"`
- La forme la plus simple consiste à l'affichage d'une simple séquence, avec possibilité de mélanger des chaînes de caractère et des variables

```
>>> print("i=", i, "r=", r, "s=", s)
i= 3 r= 3.9 s= Voici
```

- Il y a de multipes options d'affichage dont les deux principales sont
  - le **f-string** qui affiche formellement une chaîne de caractères encadrée par `f"` et `"`.
  - La méthode `str.format`

La méthode `str.format`

- Forme générique : `fs.format(va, vb, ...)` où
  - `fs` est une chaîne de caracètres de la forme `"aaaa{}bbbb{}..."`
  - les `va, vb, ...` sont des variables. Leurs valeur s'afficheront séquentiellement à l'endroit précis indiqué par les accolades {}
- Exemple :

```
>>> print("i={}, r={}, s={}".format(i, r, s))
i=3, r=3.9, s=Voici
```

- Il est possible de préciser entre {} le numéro d'ordre (avec `0` pour la 1ère) de la variable à afficher, et avec répétition si besoin.

```
>>> print("i={2}, s={0}, s={0}".format(s, r, i))
i=3, s=Voici, s=Voici
```

- Dans cet exemple, `i` (la dernière dans la liste donnée par `.format`) est affichée en 1er, `s` est affiché 2 fois et `r` non utilisé.

La méthode `str.format`

- Il est possible d'indiquer le format d'affichage à l'intérieur des {} dont voici un exemple avec `i` sur 4 digits, `r` avec 3 chiffres après le virgule, `s` occupe 8 espaces.

```
>>> print("i={2:4d}, r={1:.3f}, s={0:>8s}".format(s, r,
    i))
i=   3, r=3.900, s=   Voici
```

- De multiples options existent pour la mise en page avec `str.format` que nous ne détaillons pas ici.

La *f-string*

- Forme générique : une chaîne de caractères encadrée par `f"` et `"` dans laquelle les variables sont placées entre {}.
- Exemple : `f"i=i, r=r, s=s"`. Ceci donne

```
>>> print(f"i={i}, r={r}, s={s}")
i=3, r=3.9, s=Voici
```

- Il est possible de réaliser le formatage avec les même syntaxes que pour la méthode `str.format`.
  - Exemple : Reprise de l'exemple précédent avec `str.format`

```
>>> print("i={2:4d}, r={1:.3f}, s={0:>8s}".format(s,
    r, i))
i=   3, r=3.900, s=   Voici
>>> print(f"i={i:4d}, r={r:.3f}, s={s:>8s}")
i=   3, r=3.900, s=   Voici
```

f-string : avantages

- Avec *f-string*, il est possible de rentrer une expression dans {}

```
>>> print(f"i={i:4d}, r={r:.2f}, i*r={i*r:.2f}")
i=   3, r=3.90, i*r=11.70
```

- Une autre possibilité est l'affichage multi-lignes. Dans ce cas, le message est encadré par f""" et """
  - Exemple : Nous pouvons par exemple afficher ainsi l'opération $i \times r$

```
>>> print(f"""
          i:{i:6d}
         *r:{r:10.3f}
         ----------------------
         ={i*r:10.3f}
         """)
          i:      3
         *r:      3.900
         ----------------------
         =     11.700
```

La fonction `input`

- Forme générique : `v = input(msg)`
  - `v` est la variable dont on veut récupérer la valeur via l'entrée standard
  - `msg` est une chaîne de caractères *optionnelle* (mais généralement nécessaire pour une question de clarté)
- Opérationnellement, `input` **affiche** le message `msg` *puis* récupère **tout la ligne** de l'entrée standard pour l'affecter à `v`
- Attention : Par défaut, `input` récupère une **chaîne de caractères**. Il faut la **convertir** au bon *type* si la variable est un `int` ou `float`.
- Exemple : Résumé des 3 situations

```
vs = input(" str SVP ")
vi = int(input(" int SVP "))
vr = float(input(" float SVP "))
print(vs, vi, vr)
```

- Attention : La saisie est toujours une opération délicate car elle est tributaire de ce que l'on trouve sur l'entrée standard
  - Voir plus loin pour un moyen élémentaire de sécurisation de saisie

Les opérateurs

- Nous présentons les opérateurs habituels en mettant en avant leur spécificité dans Python
  - Arithmétiques : addition, soustraction, *etc.*.
    - Exemple : `x = 2 + 3.0`
    - Avec combinaison *opération puis affectation* : Exemple : Rajouter 2 à `x` : `(x += 2)`
  - Relationnelles : comparaison (supérieur, égale, *etc.*.)
    - Exemple : `(x != y)`
  - Logiques : ET, OU, NEGATION
    - Exemple : `(x and y)`
- Il y a aussi deux opérateurs particuliers à Python
  - Vérificateur d'appartenance : `in`
  - Vérificateur d'identité : `is`
- Remarque : Nous ne présentons pas ici les opérateurs sur les éléments binaires *bitwise operators*

Opérations arithmétiques pour les entiers et les réels

- Forme générique : `a op b` où
  - `op` est l'un des 7 opérateurs et
  - `a` et `b` sont des opérandes, entier ou réel ou un mélange des 2.
- les 4 opérations de base : `+, -, *, /`, avec l'ordre conventionnel d'exécution.
  - Attention : La division donne par défaut un réel : `4 / 2` donne `2.0`
- Division réduite à la partie entière (`//`) : donne la partie entière du résultat (l'entière immédiatement inférieur)
  - Exemple : `4 // 2` donne `2`, `5 // 2` donne `2`
  - Attention : Si l'un des arguments est un réel, ça donne la valeur de la partie entière, mais sous forme d'un réel : `5.8 // 2` donne `2.0`
- Modulo (`%`) `a % m` : le reste de la division de `a` par `m`
  - Exemple : `7 % 2` donne `1`
  - Exemple : `5.2 % 2` donne `1.2`
- Puissance (`**`) : `5 ** 2` donne `2`, `1.1 ** 2` donne `1.21`

Opération puis affectation

- Comparé au langage C, il n'y pas d'opérateur `++` et `--`
- *A contrario*, la combinaison "opération puis affectation" est possible pour l'ensemble des **7** opérateurs
- Forme générique : `a <op>= b` où `op` est un des 7 opérateurs
- Equivalence opérationnelle : `a = a <op> b`
- Exemple : la séquence suivante débute avec `x=3` pour valeur initiale
  - après `x += 4`, `x` vaut `7`
  - après `x %= 3`, `x` vaut `1`
  - après `x *= 2`, `x` vaut `2`
  - après `x **= 3`, `x` vaut `8`

Comparaison

- Evaluation d'une expression du type `(x operator y)` qui donne `True` (1) ou `False` (0)
- Opérateurs avec notation "habituelles" : `>, >=, <, <=`
- Operateurs "égal à" : `==` et "différent de" : `!=`
  - Attention : Surtout ne pas le confondre avec `=` qui réalise une **affectation de valeur**.
  - Attention : Il faut **s'interdire** de vérifier **l'égalité** pour réels, qui peut donner de faux résultats suivant la manière dont les réels sont mémorisées
- Mélange possible d'entiers et de flottants
- Exemple : avec `i = 2`, `r = 3.5`, `s = "Bon"`
  - `print( 3>4 )` donne `False`
  - `print( i < r )` donne `True`
  - `print( s == "Good" )` donne `False`

Opérations logiques

- Il y a trois opérateurs logiques dans Python : `and`, `or`, `not`
  - `A and B` et `A or B` sont évalués de gauche à droite :
    - Pour `and`, ça renvoie `False` si `A` l'est, sinon, il faut envoyer l'évaluation logique de `B`
    - Pour `or`, ça renvoie `True` si `A` l'est, sinon, il faut envoyer l'évaluation logique de `B`
  - `not A` inverse l'évaluation logique de `A` : en particulier `not True` vaut `False`.
- Exemple : avec `bt = True` et `bf = False`
  - `print( bt and bf)` donne `False`
  - `print( bf or True)` donne `True`
- Les opérations logiques peuvent s'enchaîner
  - Exemple : `print( bf or True and bt )` donne `True`
  - L'ordre d'évaluation est toujours de gauche à droite.

Opération logique et comparaison : mélange et chaînage

- L'opération comparaison donne un résultat **binaire**, il est donc logique de mélanger des opérations de comparaisons et opérations logiques :
  - Exemple : `print( 1 > 2 or 3 <4 and bt )` donne `True`
- Dans Python, il est possible d'*aligner* une séquence de comparaison (*Chaining comparison*)
- Exemple : avec `i = 2`, `r = 3.5`
  - `print( 1 < i < 3 )` donne `True`
  - `print( 2 > r < 20 )` donne `False`
  - `print( 1 <= i < r < 20 )` donne `True`
- En interne, ça revient à effectuer autant de comparaisons élémentaires puis réaliser un `and` de l'ensemble des résultats de ces comparaisons élémentaires

Valeur logique d'un objet

- Dans Python, tout *objet* possède sa valeur **logique** qui s'obtient avec la fonction `bool`.
- `bool` renvoie `False` dans un nombre réduit de cas précis. En dehors de ces cas, `bool` renvoie `True`
  - En particulier, `bool` renvoie `False` s'il s'agit d'une expression qui est `False`, ou d'une valeur qui vaut `0` ou d'une chaîne de caractères *vide*

Opérateurs d'appartenance et d'identité

- L'opérateur d'appartenance (*membership*) : `in`
  - L'opérateur `in` vérifie l'**existence** d'un élément dans un ensemble
    - Exemple : `print( 1 in [1, 2, 3, 4])` donne `True`
    - Exemple : `print( print( "H" in "Bonjour"))` donne `False`
  - L'opérateur *dual* `not in` vérifie l'**absence** d'un élément dans un ensemble
    - Exemple : `print( 3.5 not in [1, 2, 3, 4])` donne `True`
- L'opérateur `is` vérifie si deux variables correspondent au même contenu (même pointeur)
  - Exemple : Le résultat de la séquence suivante donne `True`

```
sa = "Hello"
sb = sa
print(sa is sb)
```

  - Exemple : Dès que `sb` change de valeur, ça donne `False`

```
sb = "Ciao"
print(sa is sb)
```

Les structures de contrôle

- Une **structure de contrôle** (ou structure conditionnelle) permet l'exécution *conditionnelle* d'un bloc d'instructions :
- En Python, il y a deux catégories de structures de contrôle.
  - Décision séquentielle, du genre "IF .. THEN .... ELSE"
  - Boucle, du genre "FOR" ou "WHILE"
- Décision séquentielle :
  - `if` ... `elif` ... `else`
- Boucles :
  - `for`
  - `while`
- instructions particulières : `break`, `continue`

Séquence `if ... elif ... else`

- Forme générique : `if` (condition)`:` {bloc} `else:` {bloc}

```
sa="lundi"
if sa == "lundi":
    print(sa)
else:
    print("mardi")
```

- Avec plusieurs expressions logiques

```
sa = "samedi"
if sa == "lundi" or sa == "jeudi":
    print(sa)
else:
    print("mardi")
```

- Attention : Ne pas oublier les `:` qui terminent chaque instruction de contrôle

Boucle IF ... ELIF ... ELSE : suite et remarques

- Si plusieurs cas à tester, rajouter autant de `elif` :
  - Forme générique : `if` (condition1)`:` {bloc} `elif` (condition2)`:` {bloc} ... `else:` {bloc}

```
sa="mardi"
if sa == "jeudi":
    print(sa)
elif sa == "mardi":
    print("OK")
else:
    print("Non")
```

- Pratiquer l'**indentation**
  - Un bloc d'instruction à exécuter doit respecter l'**indentation** par rapport à l'instruction de contrôle à laquelle il est lié.
- On n'utilise généralement pas de parenthèse pour isoler chaque expression logique.
  - Chaque expression s'isole déjà du reste par des espaces
- Une ligne vide marque la fin de la séquence de contrôle
- Ces remarques sont valables également pour les boucles `while` et `for`

Boucle while

- Forme générique :

```
while condition:
    bloc d'instruction
```

- Remarque : Une ligne vide marque la fin de la boucle
- Exemple : l'affichage de la séquence `0, 1, 2`

```
i = 0
while i < 3 :
    print(i)
    i +=1
```

- Attention : Tant que la condition reste inchangée, ça boucle toujours sur le même bloc
  - Il faut toujours s'assurer que la condition est modifiée dans la boucle avec la certitude de pouvoir sortir de la boucle.

Boucle `for`

- La boucle `for` dans Python est assez **particulière** : cela consiste au balayage (avec le mot-clé `in`) d'une **liste** donnée.

```
for a in list:
    bloc d'expressions
```

- Exemple : l'affichage de la séquence `0, 1, 2`

```
my_list = [0, 1, 2]
for i in my_list:
    print(i)
```

- On peut retrouver un `for` "classic" (avec incrémentation itérative) via la fonction `range`
  - Exemple : `range(3)` génère au fait la liste `[0, 1, 2]`

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

  - Remarque : `range` permet de réaliser des incrémentations plus complexes.

Rupture et sortie de boucle avec `break`

- L'instruction `break` permet d'interrompre l'exécution des instructions suivantes dans une boucle (`while` ou `for`) et **quitter** la boucle.
- Exemple : Cette boucle tournera jusqu'à $i = 3$ seulement

```
i = 0
while i < 300:
    print(i)
    if i > 2:
        print(" Break avec i = ", i)
        break
    i += 1

print("Ouf, on sort avec i=", i)
```

- Attention : Il faut prêter grand soin à l'**indentation**

`break` avec boucles emboîtées

- L'effet de `break` porte sur la boucle dans laquelle elle se trouve.
- Si `break` se trouve dans une boucle B_i qui est elle-même incorporée dans une boucle B_o, `break` fait abroger la boucle B_i mais la boucle B_o continue de "tourner".
- Exemple : Dans cette double boucle, l'effet de `break` porte sur la boucle `j`

```
while i < 2:
    print(i)
    j = 0
    while j < 300:
        if j > 1:
            print(" Break boucle j avec (i, j) = ", i, " ", j)
            break
        j += 1
    i += 1
print("On sort de la double boucle  avec i, j=", i, " ", j)
```

- Remarque : Il y a ici trois niveaux d'**indentation**

Saut vers l'itération suivante dans une boucle avec `continue`

- L'instruction `continue` permet d'interrompre l'exécution des instructions suivantes dans une boucle (`while` ou `for`) pour passer à l'**itération suivante** tout en *restant* dans la boucle.
- Exemple : Dans la boucle suivante, un saut est pratiqué chaque fois que l'on a un entier *impaire*

```
for i in range(4):
    if  i % 2:
        continue # saut si IMPAIRE (modulo => 1)
    print(" le i affiche devrait etre PAIRE ", i)
```

- Rappel : On constate de nouveau l'effet d'**indentation**
- Similaire à `break`, l'effet de `continue` est confiné à la boucle dans laquelle elle se trouve, en cas de boucles emboîtées.

Gestion d'exception

- Python offre un puissant moyen de gestion d'exception via le couple `try: except:`
- Ceci est particulièrement utile pour traiter tout problème lié à l'erreur d'exécution (`ValueError`, `NameError`, *etc.*)
- En soi, c'est un procédé évolué pour une programmation avancée.
- Nous nous focalisons ici sur son utilisation dans la gestion des saisies.
  - Le code suivant signale une erreur si la valeur saisie n'est pas un entier, qui provoque une erreur de conversion par `int()`.
  - La variable `i` ne sera pas affectée et conservera sa valeur courante.

```
try:
    i = int(input(" Donner un entier SVP: "))
except ValueError:
    print(" Il faut donner un ENTIER!")
```

Gestion d'exception

- Ce code ne permet toutefois pas la saisie de la bonne valeur. Le code suivant, avec l'aide de `while` et de `break`, permet la saisie sécurisée et garantie d'un entier

```
while True:
    try:
        i = int(input(" Donner un entier SVP: "))
        break
    except ValueError:
        print(" Il faut donner un ENTIER!")
```

Fonction

- une fonction se définit avec `def`. Il faut préciser le nom de la fonction et, le cas échéant, les paramètres. La fonction peut renvoyer (`return`) un résultat.

```
def nom_fonction(parametres)
    bloc d'instruction
```

- Exemple : La fonction suivante réalise l'addition de deux nombres et renvoie le résultat

```
def my_add(a, b):
    return a + b
```

- L'invocation d'une fonction se fait à travers son nom, avec, le cas échéant, le passage des paramètre.
- Exemple : L'invocation de `my_add(3, 4)` dans `print(my_add(3, 4))` permet d'afficher `7`

Fonction (exemple)

- Un autre exemple avec un bloc d'instruction plus complexe

```
def my_add(a, b):
    x = a + b
    if x < 0 :
        x = 'valeur < 0'
    return x
```

  - Remarquer les 2 indentations successives
  - Remarquer aussi que la valeur retournée peut changer de type (à titre illustratif, pas nécessairement une bonne pratique de la programmation)

Documentation

- Comme tout langage évolué, Python permet la **documentation automatique** des codes. Ceci se fait à l'aide des **docstrings**.
- Une **docstrings** est une chaîne de caractères encadrée par deux **triples** ".
- Pour documenter une fonction, il faut insérer une *docstring* comme **la 1ère ligne** après sa définition Exemple : Pour `my_add`,ça pourrait être ceci

```
def my_add(a, b):
    """ addition des 2 parametres a et b,
 renvoyer le resultat
    """
    return a + b
```

- Pour obtenir cette documentation, il faut faire

```
print(my_add.__doc__)
```

- On obtient alors

```
 addition des 2 parametres a et b,
 renvoyer le resultat
```

Variable globale vs Variable locale

- Une **variable locale** est définie et valable au sein de la fonction où elle est définie
- Une **variable globale** est définie et valable au sein de tout le programme
  - Elle peut être définie partout, en dehors des **fonctions**
  - Généralement, elle est définie au début du programme
- Une variable locale peut posséder le même identifiant qu'une variable globale.
  - Dans un tel cas, c'est la variable locale qui prévaut (c'est-à-dire la valeur de l'identifiant sera celle de la variable locale) dans **sa** fonction.
- Remarque : Il est préférable de réduire le nombre de variables **globales** au strict minimum

Ouverture de fichiers (avec `open`)

- Syntaxe : `f = open("filename", "mode")`
  - Le fichier ouvert est représenté dans Python par une variable `f` qui est un **objet** muni des traitements (`method`) appropriés.
  - `mode` spécifie le mode d'ouverture avec les caractères et combinaisons suivants :
    - `r` : mode de lecture seule (mode par défaut)
    - `w` : mode d'écriture. Si le fichier n'existait pas, il sera **créé**. Si le fichier existe, son contenu sera **effacé**.
    - `a` : Similaire à `w`, sauf qu'en termes d'écriture, le contenu sera **rajouté** (*append*) à l'existant.
    - Rajout de la Signe `+` : `w+` ou `a+` permet la lecture également
    - Rajout de la Signe `b` : fichier en **binaire** : Exemple : `rb` : lecture seule en binaire, `w+b` : lecture et écriture en binaire
    - La signe `t` désigne le mode **texte** qui est le mode **par défaut**
    - La signe `x` crée un fichier et renvoie une erreur si le fichier existe déjà.
- Exemple : `f = open("ftest.txt", "r")` : Après cette opération, le fichier `ftest.txt` est ouvert en mode *texte* et *lecture seule* (*read only*).

Lecture depuis un fichier

- Lecture du contenu se fait à travers la *méthode* `.read`
  - Exemple : `content = f.read()`
  - Attention : La "chaîne" de caractères `content` contient alors **tout le contenu** du fichier (`ftest.txt` dans notre exemple)
- Lecture de la ligne courante se fait avec la *méthode* `.readline`
  - Exemple : `s = f.readline()`
  - La chaîne de caractères `s` contient le contenu de **la ligne courante** du fichier `ftest.txt`
  - Attention : Il ne faut pas confondre avec la *méthode* `.readlines` (*cf. infra*)
- La méthode `.readlines` lit **toutes les lignes** pour les renvoyer séquentiellement dans une `list`
  - Exemple : `lns = f.readlines()`
  - `lns` est une liste, `lns[0]` est le contenu de la première ligne plus `\n`, *etc.*.

Création, Ecriture, Fermeture

- L'instruction `fa = open("ftesta.txt", "w+")` crée le fichier `ftesta.txt` s'il n'existait pas. Si le fichier existe déjà, son contenu sera effacé. Le fichier sera prêt pour écriture et lecture. La position d'écriture est la **fin** du fichier.
- L'instruction `fa = open("ftesta.txt", "a+")` produit un effet similaire, mais avec la conservation du contenu si le fichier existe déjà. La position d'écriture est le **début** du fichier.
- L'écriture dans un fichier *texte* se fait à travers la méthode `write(s)` où `s` est une chaîne de caractères.
  - Exemple : `fa.write("ligne 1\n")`
- Il faut rajouter le saut de ligne `\n`.
- Après la fin d'écriture, il faut **fermer** le fichier avec `f.close()`

Fichiers multiples

- Il est possible de travailler avec plusieurs fichiers avec `with`
- Exemple : copie du contenu d'un fichier dans l'autre

```
with open("ftest.txt", "r") as f_in, \
     open("ftestb.txt", "w+") as f_out:
    # read lines into a list
    c_in = f_in.readlines()
    # write to festb.txt en balayant la liste
    for ligne in c_in:
        f_out.write(ligne)
f_in.close()
f_out.close()
```

- Remarque : Sur la ligne `with`, nous avons utilisé le \ pour "briser" cette longue instruction en 2 lignes

Fichiers multiples

- On obtiendrait le même résultat avec le code suivant, où `read(8)` lit 8 octets et `len(s)` donne la taille de la chaîne `s`

```
with open("ftest.txt", "r") as f_in, \
     open("ftestc.txt", "w+") as f_out:
    s = f_in.read(8) # read 8 bytes
    # write to festc.txt par bout de 8 octets
    # jusqu'a l'epuisement
    while len(s) :
        f_out.write(s)
        s = f_in.read(8)
f_in.close()
f_out.close()
```

Tuples

- Un `tuple` est une `list` avec des valeurs **constantes**.
- Syntaxe : un `tuple` est une suite de valeurs encadrées par `(` et `)` (et non pas par `[` et `]` comme pour une `list`)
- Ses éléments s'identifient à travers `[i]`, comme pour une `list`.
- Exemple : Déclaration et affectation d'un tuple, affichage

```
>>> xt = (1, 2, 3.5)
>>> print(xt)
(1, 2, 3.5)
>>> print(xt[1])
2
```

- Une tentative d'affectation `xt[0] = 3` conduit à une `TypeError`
  - `'tuple' object does not support item assignment`
- la commande `tuple` transforme une `list` en un `tuple`.

```
>>> tuple([1, 2, 3.5])
(1, 2, 3.5)
>>> tuple("Paris")
('P', 'a', 'r', 'i', 's')
```

Dictionnaire

- Un `dictionnary` est une collection de *couples* `key:value` encadrés par { et }
- Exemple : On définit un dictionnaire (`mybase`)avec deux clés (`key`) : `"Name"` et `"Age"` avec les valeurs assiciées `"Alice"` et `23`

```
>>> mybase = {"Name" : "Alice", "Age" : 23}
>>> print(mybase)
{'Name': 'Alice', 'Age': 23}
```

- On peut toujours rajouter un couple à un dictionnaire, en particulier, Un dictionnaire peut être défini sans élément puis être complété.

```
>>> mybase["Taille"] = 1.70
>>> print(mybase)
{'Name': 'Alice', 'Age': 23, 'Taille': 1.7}
>>> mybs = {}
>>> mybs["Name"] = "Bob"
>>> print(mybs)
{'Name': 'Bob'}
```

Dictionnaire

- On accède à la valeur associée à une clé (disons `k`) avec `d[k]`

```
>>> print(mybase["Name"])
Alice
```

- Remarque : Il n'y a pas d'ordre parmi les clés
- Les clés doivent être uniques (à une clé, une et une seule valeur)
- Il est possibles d'avoir de multiples dictionnaires ayant des clés identiques

```
>>> mybs = {"Name" : "Bob", "Age" : 32}
>>> print(mybase, mybs)
{'Name': 'Alice', 'Age': 23, 'Taille': 1.7} {'Name': '
    Bob', 'Age': 32}
```

Dictionnaire

- Les dictionnaires constituent une structure de données très puissantes dont on ne détaille pas toutes les facettes ici. Voici trois possibilités/fonctionnalités
- Balayage des clés

```
>>> for key in mybase:
        print(key, mybase[key])
Name Alice
Age 23
Taille 1.7
```

- Les méthodes `.keys()` et `.values()` renvoie les listes de clés et valeurs respectives

```
>>> mybase.keys()
dict_keys(['Name', 'Age', 'Taille'])
>>> mybase.values()
dict_values(['Alice', 23, 1.7])
```

- Création d'une `list` de *dictionnaires* partageant les mêmes clés pour former une sorte de base de données

Module

- Présentation
  - Un **module** dans Python est un ensemble de codes Python qui a pour vocation d'être utilisé par d'autres programmes Python
  - Ceci renforce la modularité et la re-utilisabilité des codes Python
  - Un des points forts de Python réside dans la disponibilité de **très nombreux modules** et **packages** (collection de modules) spécialisées dans **divers domaines d'applications**.
- Un module est un fichier (script) python qui **s'identifie** par le nom du fichier. Un module regroupe généralement un certain nombre de fonctions et classes autour d'un thème précis.
- Exemple : La fonction `my_add` dans un fichier nommé `module1.py`

```
# L'unique fonction de ce module
def my_add(a, b):
    """ addition des 2 parametres a et b,
 renvoyer le resultat
    """
    return a + b
```

Utilisation d'un Module

- Lorqu'un script Python veut utiliser un module, il faut l'inclure par l'instruction **import**
- Il est coutume de placer les **import** sur les 1ères lignes
- Exemple : Voici l'utilisation du module `module1` dans un script nommé `main.py`

```
import module1 as m1
# il est commode de donner une abbreviation
# utilisation de my_add du module m1
print(m1.my_add(3, 4))
print(m1.my_add.__doc__)
```

  - Nous observons que la fonction se refère *explicitement* au module dans lequel il se trouve
  - Il est commode de donner une abbréviation à un module via `as`

Référence explicite et espace des noms

- Il est important de souligner que les fonctions d'un module se référent **explicitement** à **ce** module.
- Dans le code suivant, une fonction **homonyme** est défini dans le script appelant (`main-b.py`). Les 2 fonctions **ne se confondent** pas, grâce à la référence explicite

```
import module1 as m1
# Une fonction "locale" homonyme
def my_add(a, b):
    print(" Locale :", a, " ", b)
    return a + b
# utilisation de my_add du module m1
print("from m1: ", m1.my_add(3, 4))
# utilisation de my_add LOCALE
print("Locale : ", my_add(3, 4))
```

Référence explicite et espace des noms

- Il convient de faire attention à ce que toute fonction d'un module soit explicitement et uniquement référencé.
- Il existe une instruction `from moduleX import fonctionY` (resp. `from moduleX import *`) pour importer depuis le module `moduleX` *nominalement* la fonction `fonctionY` (resp. *toutes les fonction* avec `*`).
- Ces fonctions partageront, alors, le **même espace des noms** que le script qui les importent.
- Le script `main-c` reproduit une telle situation et on constate que la fonction *locale homonyme* est *neutralisée*.
- Remarque : Il est prudent de garder son espace de noms propre à chaque module
- Un module se documente selon le même principe qu'une fonction : on place les `docstrings` sur les premières lignes d'un module

Exploitation des `module`

- On obtient la liste des modules sous un `shell` avec la commande `help("modules")`
- Il y a un certain nombre de `built-in module` dont
  - `sys` qui gère l'environnement d'exécution (*runtime environment*) ;
  - `OS` qui offre des fonctionnalités du système d'exploitation, telles création ou changement de répertoire ;
  - `math` qui offre des fonctions mathématiques telles sin ou log.
  - `random` qui offre diverses fonctions liées à la génération de nombres aléatoires.
- l'ensemble des fonctions internes et `built-in modules` constitue la bibliothéque standard de Python.

Les packages

- Les nombreux packages constituent un **atout majeur** de Python
- Un **package** est une collection de modules Python :
  - Un **module** correspond à un seul fichier *homonyme*.
  - Un **package** est un répertoire contenant plusieurs modules et qui se distingue (versus un répertoire ordinaire avec plusieurs fichier `.py`) par l'existence d'un fichier `__init__.py`
- Voici un spécimen de **package** :
  - `NumPy` est dédié au traitement vectoriel et matricielle
  - `SciPy` vise à couvrir le calcul mathématique dans son ensemble
  - `Pandas` est dédié au traitement et à l'analyse de données à structures complexes et hétérogènes
  - `Matplotlib` permet de tracer et visualiser des données sous diverses formes
- Attention : En raison de l'évolutivité de chaque package et les dépendances complexe entre les packages, l'installation d'un package doit être confiée à un utilitaire spécialisé.
- Remarque : L'installation puis l'utilisation des packages ne rentre pas dans les objectifs de cette présentation

Arguments de `main`

- Nous illustrons l'utilisation de modules à travers le passage d'arguments (équivalent aux arguments passés avec `main` de C)
- Cette fonctionnalité est supportée par le module `sys`
- La liste des arguments est donnée par `sys.argv`
- `len(sys.argv)` joue le rôle du compteur `argc` de C
- Exemple : Le code suivant reproduit une situation similaire à celle que vous avez vu en C
  - Il faut l'invoquer dans un terminal avec `python sysarg.py p1 p2`
  - `sysarg.py` est le nom du script, `p1, p2` sont des arguments facultatifs

```
import sys
print(sys.argv, len(sys.argv))
if  len(sys.argv) > 2:
    print("Tu es trop bavard.e, you are boring \n")
elif (len(sys.argv) == 2):
    print(sys.argv[0], " te dit ", sys.argv[1])
else:
    print(sys.argv[0], " te dit Bonjour par defaut")
```

Python et OOP

- Python est un langage orienté objet (OOP)
- Nous présentons ici, **extrêmement sobrement**, une illustration *simplissimo* de la façon dont les objets (`class`)sont gérés et exploités dans Python
- Tout objet correspond à une `class` possèdant son *espace propre* avec
  - l'ensemble des `attributs` (paramètres internes)
  - une collection de `method` (fonctions internes)
- Chaque objet variable correspondant à une classe donnée est instanciée par le `constructor` de la classe
- Dans Python, le constructeur est la methode nommée `__init__(self, p1, p2, ...)`
  - `self` se réfère à l'objet (la class) lui-même
  - Les `attributs` sont définis par le constructeur
  - Ils reçoivent les valeurs initiales `p1, p2, ...`
- Les `method` se définissent comme des fonctions.
  - Les attributs doivent être identifiés par le prefix `self`.
  - `self` est donc généralment passé comme un argument à une methode.

Class : un exemple

- Définition d'une classe `myBase` possédant 2 attributs et 2 méthodes ; puis instantiation de 2 variables `mb, mc` et leur exploitation

```
class myBase:
    # CONSTRUCTEUR
    def __init__(self, n="Alice", a=16):
        self.Name = n
        self.Age  = a
    # methods
    def addAge(self, i):
        self.Age += i
    def printAge(self):
        print(f"{self.Age}")
mb = myBase()
print(mb.Name)
mc = myBase("Bob", 23)
print(mc.Age)
mc.addAge(53)
mc.printAge()
```