Premiers pas en Python

15 septembre 2020

Programme informatique

Un programme informatique est une suite d'instructions, un texte donc, destiné à être lu pour exécution par une machine.
Ce que fait un programme informatique c'est, en s'inscrivant dans un cadre chronologique cadencé,

- (Entrée) Lire des données qui sont présentes au moment donné en mémoire. Elle peuvent avoir été calculées précédemment, lues d'un disque dur ou d'une clé USB, placées en mémoire par un périphérique d'entrée (clavier, souris, réseau, dispositif d'acquisition numérique) ;
- (Traitement) Transformer ces données suivant un algorithme décrit par les instructions du programme ; instructions écrites dans un certain *langage de programmmation*.
- (Sortie) Ecrire de nouvelles données en mémoire, *i.e.* en RAM, sur un disque, sur un périphérique de sortie (console, écran graphique, imprimante, réseau, ...)

Programme informatique

Cette description en Entrée-Traitement-Sortie est analogue à celle d'une fonction mathématique abstraite (deux notations)

$$f : x \mapsto f(x) \text{ ou } x \overset{f}{\mapsto} f(x).$$

L'argument x décrit l'entrée, la valeur calculée $f(x)$ décrit la sortie, la flèche marque le traitement effectué.
On peut discuter du fait qu'une fonction mathématique puisse écrire quelque chose sur un écran ou lire un clavier, mais oui, on y arrive, pourvu que les ensembles de valeurs possibles pour x et $f(x)$ aient la capacité de décrire l'intégralié physique de la machine...

Programme informatique

Pour composer un programme informatique, et ce sont les instructions du programme qui gèrent cela, on peut faire se succéder–c'est le sens de « cadre chronologique cadencé »– des fonctions (ou programmes) plus élémentaires avec possibilité de répétition ou de choix.

$$x \overset{f}{\mapsto} f(x) \overset{g}{\mapsto} g(f(x)) \ldots \overset{h}{\mapsto} h(\ldots\ldots g(f(x)) \ldots\ldots).$$

Une machine, un langage, des langages

- Une machine physique (*i.e.* un ensemble périphériques d'E/S, CPU et mémoire) lit et exécute de façon native un langage de programmation, qui s'appelle le langage machine. C'est ce langage dont les instructions sont lues et exécutées à la cadence annoncé du CPU.

Une machine, un langage, des langages

- Une machine physique (*i.e.* un ensemble périphériques d'E/S, CPU et mémoire) lit et exécute de façon native un langage de programmation, qui s'appelle le langage machine. C'est ce langage dont les instructions sont lues et exécutées à la cadence annoncé du CPU.
- On peut écrire dans ce langage machine d'autres langages (dits de plus haut niveau) de programmation dont les programmes devront être traduits en langage machine pour être exécutés par la machine.
  Des exemples actuels de tels langages sont C, C++, Python, Perl, Java, Javascript, Lisp, CAML, Pascal, Fortran, ...
  Il existe d'autres langages, qui ne sont pas à proprement parler des langages de programmation mais plutôt des langages de description de données.

Une machine, un langage, des langages

- Une machine physique (*i.e.* un ensemble périphériques d'E/S, CPU et mémoire) lit et exécute de façon native un langage de programmation, qui s'appelle le langage machine. C'est ce langage dont les instructions sont lues et exécutées à la cadence annoncé du CPU.
- On peut écrire dans ce langage machine d'autres langages (dits de plus haut niveau) de programmation dont les programmes devront être traduits en langage machine pour être exécutés par la machine.
- Certaines applications (Firefox, par exemple ou certains moteurs de jeux), ont en leur sein une « machine virtuelle » programmable par un langage propre. Cette machine virtuelle effectue le travail de traduction en langage machine vers la machine réelle.

Langages compilés/Langages interprétés

Les langages de haut niveau se répartissent en deux grandes catégories.

- Langages compilés (p.ex. C/C++) : La traduction du programme en langage machine s'effectue *avant* l'exécution, c'est la compilation. Le flux de production d'un programme est un peu lourd pour un débutant (Ecriture/Compilation/Test/Correction/Compilation/...) mais le programme final est très rapide et autonome ;
- Langages interprétés (p.ex. Python/Javascript) : La traduction du programme en langage machine s'effectue *en même temps* que l'exécution, c'est l'interprétation. Le flux de production d'un programme est plus léger pour un débutant (Ecriture/Test/Correction/Test/) mais le programme final est plus lent et nécessite la présence de l'interpréteur pour fonctionner.

Langages compilés/Langages interprétés

Les langages de haut niveau se caractérisent par :

- un système de repérage des données en mémoire cachant leur adresse réelle (pas d'adresses en hexadécimal à manipuler !) : type de données et variables ;
- un système d'instructions permettant facilement de boucler ou de faire des choix (on oublie le système de déplacement primitif d'une machine de Turing abstraite) : mots-clés du langage ;
- la possibilité de grouper des instructions en des blocs de programme (blocs, définition de fonctions) ;
- un système de fonctions prédéfinies permettant des entrées/sorties faciles ;
- la possibilité d'utiliser des *bibliothèques* de fonctions écrites par d'autres programmeurs (On n'est pas obligé de réinventer la roue à chaque programme.).

Langages interprétés

- La machine virtuelle de Firefox contient un intepréteur Javascript. Un programme Javascript présent dans une page Web est donc interprété en langage propre à la machine virtuelle de Firefox qui elle même le transforme en le langage machine de la machine hôte.
- Nous allons travailler avec un(des ?) interpréteur(s) Python 3, le plus souvent en mode interactif avec une console (shell). Cet interpréteur peut aussi travailler en mode exécution directe d'un fichier de programme (portant l'extension `.py`). Un tel fichier de programme de langage interprété s'appelle un *script*.
- Un script Python est un pur fichier texte qui se travaille avec un *éditeur de texte*. Un tel fichier ne se travaille pas avec un *traitement de textes* ! ! !

Environnement de programmation

Il ne faut pas confondre l'environnement de programmation (IEP, Spyder,...) avec l'interpréteur Python.

- l'environnement de programmation (IDE) est une application qui fédère sous une même interface :
  - un éditeur de texte adapté à la programmation en Python ;
  - un système de consoles qui, chacune, est associée à une instance de l'interpréteur Python ;
  - divers outils permettant de contrôler, débuguer les programmes écrits (Inspecteur de variables, par exemple).

  Il ne faut pas non plus confondre IDE avec Distribution Python (Pyzo, Anaconda, ...) une distribution est un ensemble comprenant un IDE, un interpréteur, des librairies externes. Une distribution est destinée à faciliter la tâche d'installation sur une machine de sorte à avoir un système cohérent.

Environnement de programmation

Il ne faut pas confondre l'environnement de programmation (IEP, Spyder,...) avec l'interpréteur Python.

- l'environnement de programmation (IDE) est une application qui fédère sous une même interface :
  - un éditeur de texte adapté à la programmation en Python ;
  - un système de consoles qui, chacune, est associée à une instance de l'interpréteur Python ;
  - divers outils permettant de contrôler, débuguer les programmes écrits (Inspecteur de variables, par exemple).
- Un interpréteur Python est un programme qui lit (suivant diverses modalités, d'une traite à l'aveugle ou pas à pas en console) un script ou des instructions et l'exécute en le traduisant en langage machine. La sortie dépend du mode d'utilisation.

Environnement de programmation

Il ne faut pas confondre l'environnement de programmation (IEP, Spyder,...) avec l'interpréteur Python.

- l'environnement de programmation (IDE) est une application qui fédère sous une même interface :
  - un éditeur de texte adapté à la programmation en Python ;
  - un système de consoles qui, chacune, est associée à une instance de l'interpréteur Python ;
  - divers outils permettant de contrôler, débuguer les programmes écrits (Inspecteur de variables, par exemple).
- Un interpréteur Python est un programme qui lit (suivant diverses modalités, d'une traite à l'aveugle ou pas à pas en console) un script ou des instructions et l'exécute en le traduisant en langage machine. La sortie dépend du mode d'utilisation.
- Un script Python écrit dans un IDE, peut se faire exécuter hors de cet environnement par un autre interpréteur.

Affichage élémentaire

Le programme test le plus simple est celui permettant d'afficher un texte. En Python, la fonction permettant ceci est la fonction `print` dont la syntaxe d'appel peut se lire dans le système d'aide en console :

Affichage élémentaire

Dans la console de Spyder, `help(print)` donne

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush

    Prints the values to a stream, or to sys.stdout by default
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newl
    flush: whether to forcibly flush the stream.
```

Affichage élémentaire

On peut alors ouvrir un nouveau script dans l'éditeur intégré à Spyder et y écrire sur une seule ligne `print("Bonjour le monde")`. On sauve, on exécute (trouver le bon bouton ou mieux, le raccourci clavier) et on obtient le résultat en console.

```
Bonjour le monde
```

Affichage élémentaire

On peut se tromper en écrivant `prunt("Bonjour le monde")`. On sauve, on exécute et on obtient le résultat en console.

```
Traceback (most recent call last):
  File "hello-err.py", line 1, in <module>
    prunt("Bonjour le monde")
NameError: name 'prunt' is not defined
```

Il s'agit d'une erreur d'exécution qui doit être lue, comprise, corrigée.

Affichage élémentaire

On peut exécuter le même programme hors IDE :

- dans une ligne de commande ?
- dans un interpréteur en ligne sur le web : `http://pythontutor.com`

Evolution chronologique

On utilise, dans cette séance seulement, le site `http://pythontutor.com` pour montrer le fonctionnement interne de l'interpréteur.
Commençons par comprendre comment fonctionne un programme de plusieurs lignes et comment se déroule l'évolution de la lecture d'un script.
On recopie (copier/coller) le script suivant dans la zone de programme et on exécute ligne à ligne.

Evolution chronologique

Listing 1 – python/hellos.py

```
print("Bonjour le monde")
print("Hello world")
print("salve mundi")
print("Ciao mondo")
print("Hei maailma")
print("Helló Világ")
```

Repérage d'un objet en mémoire, noms de variables

L'interpréteur Python dispose d'une mémoire de travail dans laquelle il stocke tous les « objets » informatiques, *i.e.* les données sur lesquelles il peut travailler.
Lors de la « création » d'un objet, l'interpréteur dégage/ouvre un espace en RAM où il enregistre l'objet en le codant à l'aide d'une suite d'octets.
Cet espace est repéré en interne par l'interpréteur par son adresse, sous forme hexadécimale.
Pour pouvoir utiliser l'objet, on doit le repérer par un *identificateur*. Un *nom de variable* est un tel identificateur. Ce nom doit respecter certaines conventions sur lesquelles on reviendra, la première d'entre elle étant qu'il s'agit d'une suite de caractères commencer par une lettre de l'alphabet ou un _, sans espace.
Un même objet en mémoire peut être repéré par différents identificateurs.
Les objets simples (entiers, chaines) ont seulement l'air dupliqués.

Repérage d'un objet en mémoire, noms de variables

Cette opération de repérage s'appelle l'*affectation* (d'un nom de variable à un objet) et s'effectue en Python à l'aide du signe = isolé.
Dans certains textes (pseudo-code), on utilise le symbole $\leftarrow$ ou $:=$ (qui se lit « devient »), il s'agit d'une opération analogue à la ligne de texte mathématique « Soit $a = 2$ » où l'on pose la valeur de la constante a.

Repérage d'un objet en mémoire, noms de variables

⋆⋆⋆

$\rightsquigarrow$ Dans la zone de programme de PythonTutor ou dans la console de Spyder, taper

```
a = 2
```

observer la zone décrivant l'occupation de la mémoire de travail de l'interpréteur. ⋆

⋆⋆⋆

Cette ligne de programme est interprétée comme :

- faire apparaitre l'entier 2 dans la mémoire de travail (l'*exposer*, il y est sous forme codée en binaire) ;
- placer l'étiquette a sur cet objet pour repérage ultérieur en vue d'utilisation.

Repérage d'un objet en mémoire, noms de variables

Pour utiliser l'objet reperé par un nom de variable, il suffit d'utiliser le nom dans une *expression*.

⋆⋆⋆

$\rightsquigarrow$ Ajouter une ligne `print(a+1,a*a)` dans le programme (ou simplement `a+2` en console), exécuter, observer. Recommencer la séquence en changeant la valeur affectée à `a`. ⋆

⋆⋆⋆

Repérage d'un objet en mémoire, noms de variables

Tout objet non repéré se fait éliminer de la mémoire de travail.
En effet, un objet non reperé n'étant pas utilisable, vu qu'on doit se servir de l'un de ses identifiants pour l'utiliser, il devient inutile et occupe de la place en mémoire. Un mécanisme de l'interpréteur, le *garbage collector*, l'élimine.

⋆⋆⋆

⇝ Dans PythonTutor ou Spyder, faire tourner pas à pas, en observant la mémoire de travail, l'un des blocs de lignes suivants :

```
a = 2
b = 5
a = b
```

```
a = [2,3] #une liste
b = 5
a = b
```

⋆

⋆⋆⋆

Repérage d'un objet en mémoire, noms de variables

Un objet possède, un *type*, une « adresse » en mémoire de travail, une taille, ce qui permet de retrouver sa valeur en décodant les octets de codant.

⋆⋆⋆

⇝ Dans Spyder, taper et exécuter les lignes suivantes :

```
import sys #Librairie externe nécessaire, impossible dans Pytho
a = 5
print(type(a),hex(id(a)),sys.getsizeof(a))
```

⋆

⋆⋆⋆

- `type(a)` : le type de l'objet (désigné par) `a`
- `id(a)` : l'adresse de l'objet `a` dans la mémoire de travail, on l'affiché sous forme hexadécimale en lui appliquant la fonction `hex`.
- `sys.getsizeof(a)` : la taille en octets de l'objet `a`

Repérage d'un objet en mémoire, noms de variables

On voit que l'objet désigné par a est un `int`, qu'il est à l'adresse x?????? et qu'il occupe 28 octets en mémoire (ce qui peut paraître beaucoup pour un simple petit entier...les entiers Python sont un peu plus complexes que de simples blocs de 1, 2 ou 4 octets)

Repérage d'un objet en mémoire, noms de variables

⋆⋆⋆

⇝ Dans Spyder, taper et exécuter les lignes suivantes :

```
import sys #Librairie externe nécessaire
a = 5
print(type(a),hex(id(a)),sys.getsizeof(a))
b = 5
print(type(b),hex(id(b)),sys.getsizeof(b))
```

⋆

⋆⋆⋆

On voit que l'objet (5) désigné par `a` et `b` est en fait le même objet en mémoire de travail.

Repérage d'un objet en mémoire, noms de variables

⋆⋆⋆

⇝ Dans PythonTutor, taper et exécuter pas à pas les lignes suivantes :

```
a = [2,3]
print(type(a),hex(id(a)))
b = [2,3]
print(type(b),hex(id(b)))
b = a
print(type(b),hex(id(b)))
```

⋆

⋆⋆⋆

On voit que les objets désignés par `a` et `b` sont :

- dans un premier temps distincts, quand bien même ils ont même valeur ;
- dans un second temps, identiques au sens où ils sont codés au même endroit dans la mémoire.

Règles sur les identificateurs

Un identificateur/un nom de variable doit répondre à certaines contraintes.

- Il débute par une lettre majus. ou minus. ou le caractère _.
- Il ne contient que des lettres, chiffres et le caractère _.
- Il ne doit pas être l'un des mots réservés (*keyword*) de Python 3 dans la liste alphabétique suivante.

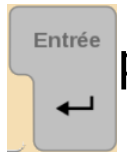
```
False      None     True     and      as
assert     async    await    break    class
continue   def      del      elif     else
except     finally  for      from     global
if         import   in       is       lambda
nonlocal   not      or       pass     raise
return     try      while    with     yield
```

Règles sur les identificateurs

Un identificateur/un nom de variable doit répondre à certaines contraintes.

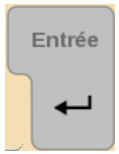
- Il débute par une lettre majus. ou minus. ou le caractère _.
- Il ne contient que des lettres, chiffres et le caractère _.
- *Nous ne parlerons dans la suite du cours que des mots clés colorés qui sont à notre programme, les couleurs données correpondant à un groupement des mots clés par catégorie syntaxique (constantes, instructions de contrôles,...)*
- Il y a possibilité d'utiliser le caractère . dans un « identificateur ». Cela a un sens très particulier. Il s'agit de la construction d'identificateurs « composés » utilisés pour des objets définis dans bibliothèques externes ou des objets référencés « à l'intérieur » d'un autre objet.

Nombres entiers, `int`

Un nombre entier explicite, une *constante* entière s'écrit en décimal à l'aide de son écriture usuelle, s'il est négatif, on précède l'écriture d'un `-`. On peut aussi l'écrire en binaire ou en hexadécimal à l'aide des préfixes `0b` et `0x` : Taper dans la console Python `0b11000011` ou `0xAB10` puis 

pour observer l'entier, affiché en décimal.

Nous avons déja vu l'affectation d'un identificateur/nom de variable à un nombre entier.

On peut, à l'aide de constantes entières et/ou d'identificateurs d'entiers, de symboles d'*opérateurs*, et de parenthèses, composer des expressions d'une manière très proche de l'usage mathématique. Si on tape une telle expression dans la console et que l'on appuie sur 

, cette expression est *évaluée* et le résultat affiché.

Nombres entiers, `int`

Les symboles d'opérateurs arithmétiques sont - (opposé, opérateur *unaire*), +, -, *, **, //, %, (respectivement addition, soustraction, multiplication, exponentiation, division entière et reste dans la divisio entière, opérateurs *binaires*).
Un opérateur est arithmétique si ses(son) arguments s'évaluent en des nombres entiers et que le résultat de l'opération est lui-même un nombre entier.

Expressions

D'une manière générale, dans l'écriture [1] d'une expression

- un opérateur est *unaire* si il agit sur *un* argument. Dans l'écriture de l'expression, l'argument suit le symbole de l'opérateur. Lors de l'évaluation de l'expression, on applique l'opérateur à l'évaluation de son argument.

  p.ex. `- (expression)`

  donne à l'évaluation, l'opposé de l'évaluation de `expression` ;
- un opérateur est *binaire* si il agit sur *deux* arguments. Dans l'écriture de l'expression, les arguments sont placés à gauche et à droite du symbole de l'opération. Lors de l'évaluation de l'expression, on applique l'opérateur aux évaluations de ses deux arguments.

  p.ex. `(expression1) + (expression2)`

  donne à l'évaluation, la somme des évaluations de `expression1` et `expression2` ;

Expressions

- Ces évaluations s'effectuent en respectant un système de priorités (ou précédences), p.ex. la multiplication est prioritaire sur l'addition. Ce système de priorités permet de s'affranchir de l'écriture de parenthèses. On peut, dans un premier temps, négliger ce point simplificateur pour ne s'intéresser qu'aux expressions totalement parenthésées.
- Comment trouver les mots corrects pour décrire ce qu'est une expression (totalement parenthésée) syntaxiquement correcte ? Il nous faudrait quelques notions de grammaire/syntaxe...

Expressions

1. Une expression (totalement parenthésée) correcte est composée d'une suite finie de caractères ;
2. Les expressions fondamentales que sont les noms de variables et les écritures de constantes sont correctes ;
3. Une expression qui s'écrit

   `expression = u(expression1)`

   où `expression1` est une expression correcte, `u` est un symbole d'opérateur unaire, est correcte ;
4. Une expression qui s'écrit

   `expression = (expression1)b(expression2)`

   où `expression1` et `expression2` sont correctes, `b` est un symbole d'opérateur binaire, est correcte ;
5. Il n'y a pas d'autres expressions correctes que celles décrites dans les trois points précédents.

Expressions

Une expression (totalement parenthésée) correcte est correctement parenthésée. Une telle expression est l'écriture en ligne d'une structure arborescente. (Dessin au tableau)
On étendra cette définition pour y inclure l'utilisation de fonctions, de listes, etc...
Si une expression est correcte, la machine peut tenter de l'évaluer (à décrire sur un exemple) et donc de calculer le résultat de l'évaluation. Il peut y avoir des problèmes d'évaluation d'une expression correcte (par exemple une division par une expression qui s'évalue à 0). Ce type de problème se traduit (en général) par l'arrêt du programme avec message approprié.

Expressions

On peut affecter le résultat de l'évaluation d'une expression à un nom de variable par une ligne de programme du type :
`a = expression`
Dans une telle ligne, l'expression à droite est évaluée et son résultat *exposé* est affecté de l'identificateur de gauche.
Une ligne du type `expression1 = expression2` n'a pas de sens si `expression1` n'est pas un identificateur.

- ⇝ Des exemples en console et en script.
- ⇝ Ce qui se passe pour `a = a+1` : exposition, puis affectation.

Réels et flottants, type `float`

Les calculs machine sur les nombres entiers sont *exacts*, aux dépassements de capacité près. La machine calcule sur les représentations binaires des nombres entiers à l'aide d'algorithmes proches de ceux de l'école primaire.
On va bien sûr vouloir faire effectuer à la machine des calculs en nombres réels. Un nombre réel générique n'admet pas de description finie et on calcule de fait sur des approximations des nombres réels que l'on appelle les flottants (`float`). On expliquera plus tard comment coder un tel nombre sur une suite d'octets mais ce qu'il faut retenir, c'est que tous les calculs en nombres réels faits sur machine sont par essence *approximatifs*.
Une constante flottante s'écrit le plus souvent à l'aide de son écriture décimale « à virgule » (Un `.`)!!!

Réels et flottants, type `float`

Les opérateurs agissant sur les nombres flottants sont les mêmes que pour les nombres entiers à l'exception de `//` et `%` qui n'ont pas de sens. `/` désigne l'opération de division.

⋆⋆⋆

⇝ Taper dans la console et observer

```
pi = 3.141592 ; type(pi)
pi2019 = 3.14159**2019
pi20 = 3.14159**20 ; pi10 = 3.14159**10 ; pi20/pi10 - pi10
```

⋆

⋆⋆⋆

On peut aussi écrire une constante flottante explicite avec la notation scientifique. `A = 6.23E23`

Le caractère approximatif des calculs en nombres flottants peut mener, à l'issue d'un calcul, à des erreurs dans l'interprétation des résultats finaux. C'est un point sur lequel il faut être vigilant ! ! (Notamment quand on construit un pont....) :

Réels et flottants, type `float`

L'interaction entre nombres entiers et nombres flottants présente quelques singularités. Python3 considère que lorsque des `int` et des `float` cohabitent dans une même expression, lors de l'évaluation, on transforme les `int` en `float`.
Attention, si mathématiquement, les entiers sont des réels parmi les autres, informatiquement, les `int` ne sont pas des `float`.
On peut cependant avoir besoin (questions d'indiçage) de transformer un `float` représentant un nombre entier en `int`. Cela s'effectue par une opération de type `n_i = int(n_f)`

- On peut diviser deux `int` avec l'opération `/`, cela donne alors un `float`.

  ⋆⋆⋆

  ⇝ Essayer en console `q = 1/3 ; type(q)` ⋆

  ⋆⋆⋆

- `1.` est le nombre 1 représenté en `float`, `1` est le nombre 1 représenté en `int`.

⋆⋆⋆

Booléens, `bool`

Les deux constantes de type booléen (`bool`) sont `True` et `False` qui représentent respectivement le Vrai et le Faux.
L'opération fondamentale de *test d'égalité* == s'évalue en un booléen. Plus précisémment == est un opérateur binaire avec une syntaxe du type

`expression1` == `expression2`

qui s'évalue

- à `True` si `expression1` et `expression2` s'évaluent en le même [1] objet ;
- à `False` sinon.

De façon contraire, l'opération binaire de *test de différence* `!=` s'évalue aussi en un booléen.
Pour ces deux opérateurs le type évalué des expressions `expression1` et `expression2` peut être quelconque.

Booléens, `bool`

On peut, sur les modèle des expressions numériques, composer des expressions booléennes à l'aide des opérateurs `not`, (unaire) `and`, `or`(binaires) représentant respectivement la négation (non), la conjonction (et) et l'alternative (ou).

- → On peut affecter un nom de variable au résultat de l'évaluation d'une expression booléenne.
- → Donner les tables de vérité au tableau, des exemples en console.
- → Les opérateurs binaires <, <=, >, >= de comparaison entre nombres sont chacun à valeur booléenne.

⋆⋆⋆

⇝ Essayer en console `print(3*0.2-0.6 == 0.0)`. Pourquoi obtient-on cela ? Comment tester l'égalité à zéro d'une expression flottante ?

⋆

⋆⋆⋆

Booléens, bool

```
epsilon = abs(7./3-4./3-1.) #epsilon-machine!!!
print(abs(3*0.2-0.6) <= epsilon)
print(abs(3*0.5-1.5) <= epsilon)
```

Listes, type `list`

On peut grouper des objets Python *quelconques* évalués par des expressions en une liste suivant la syntaxe élémentaire suivante :
`ident_liste = [expression0,...,expressionN]`
Il s'agit toujours de la même syntaxe d'affectation de l'identificateur `ident_liste` à l'évaluation de l'expression dans le membre de droite. Il faut comprendre que le membre de droite s'évalue en une *liste* des évaluations des expressions `expression*`.

Listes, type `list`

- Une liste est l'équivalent informatique d'une *famille* d'objets mathématiques, indexée par les entiers naturels entre 0 et $N-1$ où N est le nombre d'éléments de la liste ;
- `L = []` affecte l'identificateur `L` à la liste vide
- Une liste possède une *longueur*, son nombre d'éléments, que l'on retrouve grâce à la fonction `len`.
- Les éléments de la liste `L` se retrouvent individuellement à l'aide la notation `L[i]` où `i` est un entier entre `0` et `len(L)-1` ; Cette notation se lit « `L` indice `i` »
- Plus généralement, une écriture du type `exp_liste[exp_entiere]` où `exp_liste` s'évalue en une liste L et `exp_entiere` s'évalue en un nombre entier i est un identificateur d'objet Python.

Listes, type `list`

- Une liste est l'équivalent informatique d'une *famille* d'objets mathématiques, indexée par les entiers naturels entre 0 et $N-1$ où N est le nombre d'éléments de la liste ;
- `L = []` affecte l'identificateur `L` à la liste vide
- Une liste possède une *longueur*, son nombre d'éléments, que l'on retrouve grâce à la fonction `len`.
- Les éléments de la liste `L` se retrouvent individuellement à l'aide la notation `L[i]` où `i` est un entier entre `0` et `len(L)-1` ; Cette notation se lit « `L` indice `i` »
- Plus généralement,
  - On peut donc le placer à gauche d'un symbole d'affectation = pour modifier l'élément référencé en position i de la liste L
  - Dans une expression, il s'évalue en la valeur de l'objet référencé en position i liste L.

Listes, type `list`

Pour manipuler les listes (on suppose que `L` est une liste) :

- `L.append(expression)` place le résultat de l'évaluation de `expression` en fin de la liste `L`, celle-ci comporte un élément de plus ;
- `L.extend(expression_liste)` étend la liste `L` avec l'évaluation de `expression_liste` (qui doit s'évaluer en une liste) ;
- `L.pop()` élimine le dernier élément de la liste `L`, celle-ci comporte un élément de moins ; permet aussi de récupérer cet élément dans la syntaxe d'affectation `a = L.pop()`

⋆⋆⋆

⇝ Exemples dans PythonTutor. ⋆

⋆⋆⋆

Listes, type `list`

Les listes sont d'usage constant en Python, nous y reviendrons beaucoup plus longuement. On peut faire des listes de listes, les trier, s'en servir pour coder des structures algorithmiques complexes, etc..On peut bien sûr combiner les listes avec des structures de boucles ! !

Caractères et chaînes de caractères, type `str`

Une chaîne de caractères, de type `str`, est une suite de caractères. Les constantes se déclarent à l'aide de `'` en début et fin de chaîne. On peut utiliser `"`, `""` ou `"""` à la place de `'` avec quelques nuances de signification. L'affectation d'un identificateur à une chaîne de caractères fonctionne sur les modèles vus précédemment :
`s = "Bonjour"` ou `L[1] = 'Truc truc'`

Caractères et chaînes de caractères, type `str`

- `len(s)` donne la longueur de la chaîne `s`,
- On peut accéder à un caractère particulier ou à une sous-chaîne grâce à syntaxe proche de celle des listes : p.ex. `s[5]` ou `s[2:7]` ;
- On peut tester l'égalité de deux chaînes grâce à `==` ;
- On peut tester l'ordre alphabétique gräce aux opérateurs de comparaison `<=`, `<`, `>=`, `>` ;
- On peut sommer, opération `+`, deux chaînes pour composer la chaîne concaténée, p.ex.
  `s = 'Bonjour' + "le monde"`
- On peut multiplier, opération `*`, une chaîne par un entier pour répéter la chaîne, p.ex.
  `s = 'Bonjour '*32`

Caractères et chaînes de caractères, type `str`

Une chaîne de caractères n'est pas une liste `list`, on ne peut par exemple pas changer directement un caractère de la chaîne par une opération du type `s[2]='z'`.
Pour réaliser une telle opération, on doit avoir recours à l'extraction de sous-chaîne et à l'opération + :

```
s = s[:2]+ 'z' + s[3:]
```

Caractères et chaînes de caractères, type `str`

Les chaînes de caractères offrent le moyen le plus primitif de communiquer avec l'utilisateur à la console :

```
s = input("Votre animal favori ? ")
```

permet de placer une invite (ici `Votre animal favori ?`) à laquelle l'utilisateur peut répondre en finissant sa réponse par un appui sur la touche 

.

Caractères et chaînes de caractères, type `str`

Cette syntaxe ne permet de récupérer que des chaînes de caractères. Si l'on veut récupérer des nombres, il faut transformer la chaine obtenue en nombre. La chaîne doit avoir une syntaxe de nombre correcte.

- `n = int(s)` lit la chaîne `s`, essaie d'en déduire une valeur numérique *entière*, affecte le nom de variable `n` à cette valeur. Essayer le code dans une cellule Spyder :

```
s = input("Quel est votre animal favori ? ")
s2 = input("Combien en avez vous ?")
s3 = input("Combien en attendez vous à votre anniversaire ?")
print("Vous aurez ", s2+s3, " ", s, "dans un an")
print("Vous aurez ", int(s2)+int(s3), " ", s, "dans un an")
```

- `r = float(s)` lit la chaîne `s`, essaie d'en déduire une valeur numérique *flottante*, affecte le nom de variable `r` à cette valeur. Dans le code précédent remplacer `int(..)` par `float(..)`, exécuter.

Blocs

Pour des raisons syntaxiques, un peu comme dans la construction de subordonnées relatives en Français, on va avoir besoin de regrouper des instructions Python en des « blocs ». Dans beaucoup de langages, ces blocs sont signalés par des instructions ou des signes typographiques ouvrant et fermant le bloc, par exemple un couple `BEGIN/END` en `Pascal`, ou un couple `{/}` en `C/C++/Java`.
Python utilise un autre mécanisme typographique : l'indentation des lignes, *i.e.* le numéro de colonne auquel une ligne commence.
Une suite maximale de lignes consécutives (avec possibilité de sous-blocs plus indentés encore) commençant à la même colonne forme un bloc.

Blocs

Par exemple (Les . marquent une famille de lignes à la même indentation, regarder le poly.) :

Listing 2 – python/blocs.py

```
#Groupements en blocs et sous blocs
#Bloc 0, principal
x = "Bonjour"
print(x)
.
    #Sous-bloc 0.1
    y="Ciao"
    print(y)
    .
    #Fin Sous-bloc 0.1
print("rien")
    #Sous-bloc 0.2
    y="Hallo"
    print(y)
    .
```

Boucle `for .... in ...` : suivie d'un bloc

Il s'agit d'une instruction permettant à une variable de parcourir un ensemble de valeurs et d'exécuter le bloc suivant pour *chacune* de ces valeurs. En Français « Pour chaque x dans X, faire .... ».
L'ensemble de valeurs doit être un *itérable*, par exemple :

- une chaine de caractères (on boucle sur chaque caractère tour à tour) ;
- une liste ;
- un objet de type `range`.

Ce dernier type se comporte avec `for` comme une liste d'entiers :

```
for i in range(100) :
```

permet d'exécuter un bloc 100 fois avec le numéro du tour `i` conservé (les numéros commencent à 0 finissent à 99).

Boucle for .... in ... : suivie d'un bloc

Listing 3 – python/exemple-for.py

```
#Exemples simples de boucle for
S=0
for i in range(100):
    #Bloc exécuté dans la boucle
    print("Je frappe au numéro",i+1,", je d'mande mam'ze
    S = S + i*i
    #Fin du bloc
print("La somme des carrés des entiers entre 0 et 99 vau
indice = 0
for c in 'ABRACADRA' :
    #Bloc exécuté dans la boucle
    print(c, " est le caractère en position",indice)
    indice = indice + 1 # ou indice+=1, incrémentation d
    #Fin du bloc
print("Il y a",indice,"caractères en tout")
```

Boucle for .... in ... : suivie d'un bloc

Listing 4 – python/exemple-for-2.py

```
#Exemples simples de boucle for, suite
#Recopie avec modif. d'une liste
D = ['A','B','C','D']
A = []
print("liste D=",D,"liste A=",A)
for elt in D :
    #Bloc exécuté dans la boucle
    A.append(elt*2) #Ajoute elt doublé à la liste A
    #Fin du bloc
print("liste D=",D,"liste A=",A)
A = [] #vidage A
for i in range(len(D)) : #On boucle sur indices admissib
    #Bloc exécuté dans la boucle
    A.append(D[i]*3) #Ajoute la chaine D[i] en fin de A
    #Fin du bloc
print("liste D=",D,"liste A=",A)
```

Boucle `while ....` : suivie d'un bloc

Il s'agit d'une instruction permettant d'exécuter le bloc suivant *tant* qu'une certaine expression booléenne est évaluée à `True`. En Français « Tant que (condition) est vraie, faire .... ».
Deux cas très particuliers :

- `while True` : crée une boucle infinie, à déconseiller sauf si on sait sortir d'une boucle par d'autres moyens
- `while False` : le bloc dépendant n'est jamais exécuté, utile en développement pour contourner une partie de code problématique.

En général, l'expression booléenne fait intervenir des variables qui seront modifiées au cours de la boucle.
Deux instructions d'arrêt prématuré de la boucle, à éviter pour les débutants :

- `continue` : ignore le code dans la suite du bloc, revient au `while` correspondant ;
- `break` : ignore le code dans la suite du bloc, saute en fin de bloc ;

Boucle `while ....` : suivie d'un bloc

Listing 5 – python/exemple-while.py

```
#Exemples simples de boucle while
#Des for avec des while
i=0
while i < 100 :
    #Bloc exécuté dans la boucle
    print("Je frappe au numéro",i+1,", je d'mande mam'ze
    i +=1 #Incrémentation,
    #la valeur de lacondition de sortie doit être modifi
    #Fin du bloc
s = 'ABRACADRA'
i=len(s)
while i > 0 :
    #Bloc exécuté dans la boucle
    print(s[i-1], " est le caractère en position",i-1)
    i += -1
    #Fin du bloc
```

Boucle `while ....` : suivie d'un bloc

Listing 6 – python/exemple-while-2.py

```
#Exemples simples de boucle while, suite
#On vide une liste pour en remplir une autre
D = ['A','B','C','D']
A = []
print("liste D=",D,"liste A=",A)
while len(D)>0 :
    #Bloc exécuté dans la boucle
    elt = D.pop()
    A.append(elt*2) #Ajoute elt doublé à la liste A
    #Fin du bloc
print("liste D=",D,"liste A=",A)
```

Boucle `while ....` : suivie d'un bloc

Listing 7 – python/exemple-while-3.py

```
#Détermination de l'écriture en base 16 d'un entier
N=125678
position = 0
chiffres = ['0','1','2','3','4','5','6','7',
            '8','9','A','B','C','D','E','F']
Nhex=''
while N >0 :
    r = N % 16
    c = chiffres[r]
    print("Le chiffre en position", position, "est", c)
    position +=1
    Nhex = c + Nhex
    N = N // 16
print(Nhex)
```

Conditionnelle `if ....` : suivie d'un bloc

Il s'agit d'une instruction permettant d'exécuter le bloc suivant *seulement si* une certaine expression booléenne est évaluée à `True`. En Français « Si (condition) est vraie, faire .... ».
Deux cas très particuliers :

- `if True` : le bloc suivant est toujours exécuté ;
- `while False` : le bloc dépendant n'est jamais exécuté, utile en développement pour contourner une partie de code problématique.

En général, l'expression booléenne fait intervenir des variables, et donc, suivant les valeurs de ces variables, s'évaluer tantôt à `True`, tantôt à `False` !

Conditionnelle `if ... :` suivie d'un bloc

Listing 8 – python/exemple-if.py

```
#Exemples simples if
for i in range(100) :
    if i % 3 == 0:
        print(i," est divisible par 3")
```

Conditionnelle if .... : suivie d'un bloc

Listing 9 – python/exemple-if-2.py

```
#Exemples simples if
#Constitution d'une liste sous condition.
premiers = [2]
for k in range(3,100):
    k_premier = True
    for p in premiers :
        if k % p == 0:
            k_premier = False
            print(k," est divisible par ", p)
            break #Stoppe la boucle for p ...
    if k_premier : #on en a trouvé un !
        premiers.append(k)
print("Nombres premiers entre 2 et 99", premiers)
```

Alternative `if .... : blocOui / else : blocNon`

Il s'agit d'une construction permettant d'exécuter le bloc `blocOui` suivant *si* une certaine expression booléenne est évaluée à `True`et qui, *sinon* exécute le bloc `blocNon`. En Français « Si (condition) est vraie, faire .... sinon faire .... ».

Alternative if .... : blocOui / else : blocNon

Listing 10 – python/exemple-if-else.py

```
#Exemples simples if/else
for i in range(100) :
    if i % 3 == 0:
        print(i," est divisible par 3")
    else :
        print(i," n'est pas divisible par 3")
```

Alternative if .... : blocOui / else : blocNon

Listing 11 – python/exemple-if-else-2.py

```
#Exemples simples if
#Constitution d'une liste sous condition.
premiers = [2]
non_premiers = [1]
for k in range(3,100000):
    k_premier = True
    for p in premiers :
        if k % p == 0:
            k_premier = False
            break #Stoppe la boucle for p ...
    if k_premier : #on en a trouvé un !
        premiers.append(k)
    else :
        non_premiers.append(k)
print("Nombres premiers entre 2 et 99", premiers)
print("Nombres non premiers entre 1 et 99", non_premiers
```

Alternative if .... : blocOui / else : blocNon

Listing 12 – python/exemple-if-else-3.py

```
#Exemples simples if/else
#Constitution d'une liste sous condition.
N = 100
diviseurs = []
premiers = [2]
for i in range(N) :
    diviseurs.append([]) #Prépare liste des divisuers pr
for k in range(2,N):
    k_premier = True
    for p in premiers :
        if k % p == 0:
            k_premier = False
            diviseurs[k].append(p)
    if k_premier : #on en a trouvé un !
        premiers.append(k)
continuer = True
while continuer :
```

Alternatives

`if .... : bloc1 / elif ...: bloc2 /else : blocDef`

Il s'agit d'une construction permettant de choisir, dans une alternative exclusive, un bloc à éxécuter. `elif` est la contraction de *else if*. En Français « Si (condition) est vraie, faire .... sinon si ... faire .... sinon faire .... ». C'est pratique pour distinguer des cas en nombre raisonnable (ne pas en enchainer plus de 4, c'est moche), s'il y a plus de cas, préférez une technique basée sur un parcours de tous les cas possible en une boucle `while` (avancé)

Alternatives
if .... : bloc1 / elif ...: bloc2 /else : blocDef

Listing 13 – python/exemple-if-elif.py

```
#Exemple if elif/elif else, dans une boucle while
continuer = True #Pour entrer dans la boucle
while continuer :
    continuer = False #Si réponse OK on arrête
    c = input("Quelle est votre couleur préférée ? ")
    if c == 'rouge' or c == 'Rouge' :
        print("Vous n'êtes pas un taureau!")
    elif c == 'bleu' or c == 'Bleu' :
        print("Vous aimez les beaux jours!")
    elif c == 'vert' or c == 'Vert' :
        print("Vous aimez la campagne")
    else :
        print("Je ne comprends pas")
        continuer = True #On continue la boucle
```

Définition de listes en compréhension

Une syntaxe intéressante propre à Python est la définition de liste en compréhension. Elle est très proche de la définition d'un ensemble en compréhension en mathématiques. Elle mélange la syntaxe de déclaration d'une liste avec `[ ]`, une instruction de boucle `for` et, optionnellement, une conditionnelle `if` :

```
carres = [i**2 for i in range(100)]
```

est l'analogue de

$$C = \left\{ i^2, i \in \{0, \ldots, 99\} \right\}$$

Définition de listes en compréhension

Une syntaxe intéressante propre à Python est la définition de liste en compréhension. Elle est très proche de la définition d'un ensemble en compréhension en mathématiques. Elle mélange la syntaxe de déclaration d'une liste avec `[ ]`, une instruction de boucle `for` et, optionnellement, une conditionnelle `if` :

```
congru1mod4 = [i for i in range(100) if i % 4 ==1]
```

est l'analogue de

$$C = \{i \in \{0, \ldots, 99\}, i \equiv 1 \pmod 4\}$$

Structure d'un script

Un programme Python est un assemblage de *fonctions* et d'une partie principale composée essentiellement d'une suite d'appels à ces fonctions. Chacune de ces fonctions est un programme Python « encapsulé » dans un objet Python. La structure de base d'un script Python écrit correctement est la suivante :

Structure d'un script

Listing 14 – python/modele-structure.py

```
#Modele pour l'organisation d'un script Python
#import de modules externes
import ....
import .... as ....
from .... import ....
#Déclaration des fonctions et des "constantes" globales
N = .... #Constante globale, une variable en fait
L = .... #Une liste globale
#Déclaration des fonctions utilisateurs
def ma_fonction_1(....,....):
    .... #Noter l'indentation d'un cran à droite
    return ....
def ....
def ....
#Début partie principale, noter indentation à gauche tou
z = ma_fonction_1(...,...)
ma_fonction_2(z,....)
```

Déclaration d'une fonction

Une fonction se déclare

- à l'aide du mot clé `def`
- suivi d'un *nom de variable* qui sera un identificateur de cette fonction,
- suivi, entre parenthèses, de la liste de ses arguments, *i.e.* une liste de *noms de variables*
- suivi de `:`, qui termine l'*entête* de la fonction.
- Le *corps* de la fonction commence ensuite, il s'agit d'un bloc indenté. Ce corps « finit », la plupart du temps, à l'instruction/mot-clé `return`

Déclaration d'une fonction

On est donc sur le modèle :

Listing 15 – python/declaration-fct.py

```
#Structure déclaration de fonction
def ma_fonction(arg_1,arg_2,...) :
    """
    Ceci est la docstring, documentant entrée et sortie.
    Cette docstring est indentée au niveau bloc/corps.
    """
    #instructions du corps
    ....
    return .... #le return final
```

Déclaration d'une fonction

Ce que le texte/corps d'une fonction décrit c'est comment on utilise les *valeurs* des arguments passés en entrée (sur l'exemple, nommés `arg_1`,...) pour calculer/créer des objets Python. Ces objets Python sont ensuite *retournés* par la fonction. Ce sont les valeurs de sortie de la fonction.
Une fonction Python peut par ailleurs avoir des effets de bord, *i.e.* modifier des zones mémoire qui ne sont pas des valeurs de sortie. Le premier effet de bord bien connu est l'impression (utilisation de la fonction `print`) d'un résultat intermédiaire à la console [1].
D'autres effets de bord sont la modification de variables extérieures à la fonction (des variables globales) ou la modification d'un objet Python référencé indirectement (effet sur un élément d'une liste `list`).
Le nombre d'arguments (voire, la structure de la spécification des arguments) s'appelle l'*arité* de la fonction.
(On peut déclarer des fonctions dont le nombre d'arguments n'est pas prédéfini, c'est du Python avancé)

Déclaration d'une fonction

⋆⋆⋆

⇝ Voici un exemple simple (unaire) pour PythonTutor :

Listing 16 – python/fct1.py

```
#Un exemple de fonction simple
def carre(x):
    """
    Entrée: x : un float
    Sortie: le carré de la valeur de x
    Effet de bord: imprime x et son carré
    """
    x2 = x * x #Calcul du carré de x
    print("Le carré de ",x, " est ", x2 ) #Effet de bord
    return x2 #Valeur retournée par carre
#Partie ppale du script
y = carre(2) #On affecte à y la valeur retournée par car
z = carre(y) #On à z l'évaluation de carre(y)
```

Déclaration d'une fonction

⋆⋆⋆

⇝ Voici un exemple simple (4-aire) pour PythonTutor :

Listing 17 – python/fct4.py

```
#Un exemple de fonction simple 4 arguments
def vectoriel(x,y,z,t):
    """
    Entrée: x,y,z,t : des float
    Sortie: le résultat de x*y + y*z + z*t +t*x
    Effet de bord: aucun
    """
    truc = x*y + y*z + z*t +t*x  #Calcul du truc
    return truc #Valeur retournée par quadratique
#Partie ppale du script
q = quadratique(2,-1,2.0,-3.0) #On affecte à q la valeur
```

⋆

⋆⋆⋆

Déclaration d'une fonction

Listing 18 – python/exemple-while-3-fct.py

```
#D'un script à la fonction
chiffres = ['0','1','2','3','4','5','6','7',
            '8','9','A','B','C','D','E','F']
def int2hex(N) :
    """
    Entrée: N un int positif
    Sortie: chaine donnant l'écriture hexa de N
    """
    global chiffres #Pour récupérer v. externe
    Nhex=''
    while N >0 :
        r = N % 16
        c = chiffres[r]
        Nhex = c + Nhex #Ajout caractère à gauche
        N = N // 16
    return Nhex
print(int2hex(125678))
```

Déclaration d'une fonction

Le mot-clé `return` n'est pas obligatoire et n'est pas forcé d'être en fin de corps. La fin du corps de la fonction est indiquée par un retour à l'indentation la plus à gauche, qui marque la fin de la déclaration de la fonction.
Si le mot-clé `return` n'est pas présent, la fonction retourne au niveau appelant en ne retournant rien...ou plutôt en retournant l'objet Python symbolisant le rien, qui est une constante du langage désignée par le mot-clé `None` .
On peut tester qu'un objet n'est rien par l'expression à valeur booléenne (mot clé `is`) :

```
x is None
```

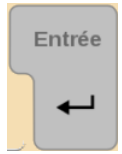
Interactions avec l'utilisateur :`print` et `input`

Nous avons déjà rencontré deux fonctions prédéfinies (`builtin`)

- La fonction `print`–
  - Elle prend une quantité indéfinie de valeurs en argument.
  - Elle écrit en console des représentations textuelles de ces valeurs : un entier est écrit sous forme décimale, une liste est écrite encadrée par des `[]`...
  - Elle n'a pas de valeur de retour : `a = print(...)` affecte à `a` l'objet `None`.

Interactions avec l'utilisateur :`print` et `input`

Nous avons déjà rencontré deux fonctions prédéfinies (`builtin`)

- La fonction `print`–
- La fonction `input`–
  - Elle prend une chaîne de caractères en argument.
  - Elle écrit le texte de cette chaîne en console et attend que l'utilisateur frappe des touches du clavier jusqu'à ce qu'il frappe la touche 

.
  - Sa valeur de retour est la chaîne des caractères frappés par l'utilisateur.

Fonctions de casting

Il s'agit de fonctions qui transforment (en anglais *cast* = fondre/couler/mouler) un objet d'un certain type en un objet d'un autre type.

1. `int(..)` :
   1. `xi = int(xf)` transforme le `float xf` en un entier, affecte la valeur à `xi`
   2. `xi = int(xs)` transforme la `string xs` en un entier, affecte la valeur à `xi`
2. `float(..)` :
   1. `xf = float(xi)` transforme le `int xi` en un flottant `float`, affecte la valeur à `xf`
   2. `xf = float(xs)` transforme la `string xs` en un flottant `float`, affecte la valeur à `xf`
3. `str(..)` : Transforme son argument en une chaîne de caractères.
4. `L = list()` : Fabrique la liste vide (transforme "rien" en la liste vide)

Portée et visibilité des noms de variables

Les noms de variable affectés dans la partie principale du script sont qualifiés de *globaux*.
On affecter des noms de variables à des objets dans le corps d'une fonction : l'identificateur utilisé sera dit *local* à la fonction. Il référence l'objet auquel il est affecté pendant le seul temps de l'éxecution/interprétation du corps de la fonction.
La question de savoir quel objet est référencé par tel identificateur au moment de l'éxecution est la question de la portée (en anglais *scope*) de l'identificateur.
C'est une notion délicate, reliée au concept d'*espace de nommage* (en anglais *namespace*).

Portée et visibilité des noms de variables

Quelques règles de base sur la portée des identificateurs.

- Les identificateurs utilisés pour nommer dans la définition de la fonction les valeurs des arguments passés sont des identificateurs locaux ; l'appel de la fonction affecte à ces identificateurs les valeurs passées en argument.
- Deux mêmes identificateurs locaux utilisés dans deux corps de fonctions distinctes sont indépendants.

Portée et visibilité des noms de variables

- Si, dans le corps d'une fonction on évalue un identificateur qui n'a pas déjà été affecté dans l'éxécution du corps de la fonction, alors une erreur est levée :
  `UnboundLocalError: local variable 'x' referenced before ass`
- Enfin, si on affecte dans le corps d'une fonction une valeur à un identificateur utilisé dans le corps principal du script (un identificateur *global*), un nouvel identificateur *local* est créé, il « masque » l'identificateur global dont la valeur devient inaccessible de la fonction. Si l'on veut affecter l'identificateur *global* (effet de bord), il faut déclarer l'identificateur avec le mot-clé `global`.

Des exemples à exécuter pas à pas dans PythonTutor.

Portée et visibilité des noms de variables

Listing 19 – python/nommage1.py

```
#Démonstration 1 de la portée d'un nom
def f(x):
    print("f appelée avec x=",x)
    return x*x
def g(x):
    print("g appelée avec x=",x)
    return x*x*x
#Corps principal
#A ce point x n'est pas un id. global
print("f(2):",f(2))
print("g(3):",g(3))
x=4 #A ce point x est un id. global
print("f(2):",f(2))
print("g(3):",g(3))
```

Portée et visibilité des noms de variables

Listing 20 – python/nommage2.py

```
#Démonstration 2 de la portée d'un nom
def f(x):
    print("f appelée avec x=",x)
    x = 5
    return x*x
def g(y):
    print("g appelée avec y=",y)
    y = x*x*x ; x = 5
    return y
#Corps principal
#A ce point x n'est pas un id. global
print("f(2):",f(2))
print("f(3):",f(3))
#print("g(3):",g(3)) #Engendre une erreur
x=4 #A ce point x est un id. global
print("f(2):",f(2))
print("g(3):",g(3))
```

Portée et visibilité des noms de variables

Listing 21 – python/nommage3.py

```
#Démonstration 3 de la portée d'un nom
def f(x):
    print("f appelée avec x=",x)
    y = x*x ;  x = 5
    return y
def g(y):
    global x
    print("g appelée avec y=",y)
    y = x*x*x ; x=5
    return y
#Corps principal
#A ce point x n'est pas un id. global
print("f(2):",f(2)) ; print("f(3):",f(3))
#print("g(3):",g(3)) #Engendre une erreur
x=4 #A ce point x est un id. global
print("g(3):",g(3)) ; print("g(3):",g(3))
```

Modules externes : fonctions et constantes

Lors du développement d'un programme, il ne faut pas réinventer la roue à chaque fois ! ! ! ! Des milliers de développeurs expérimentés ont développé des *librairies*, *bibliothèque* ou *modules externes* comportant des fonctions ou autres objets Python que l'on peut réutiliser.
La distribution Anaconda comporte plus de 500 tels modules externes et il n'est pas question de tout connaitre.
Nous allons nous focaliser sur quelques modules, notamment dans le cours sur l'initiation au calcul scientifique :

1. `numpy` : Calcul scientifique : matrices, constantes et fonctions mathématiques usuelles, générateurs de nombres aléatoires, ... ;
2. `matplotlib` : Représentations graphiques 2D et 3D, à coupler avec `numpy` ;
3. `os`, `sys` : Interface avec le système d'exploitation (informations sur le système, comment retrouver ses fichiers, ...) ;
4. `string`, `re` : Manipulation de chaînes de caractères ;
5. `sqlite` ou autre : Interaction avec des bases de données ; (le dernier cours !)

Modules externes : fonctions et constantes

Pour utiliser les fonctions d'une module externe, il faut les importer, il y a plusieurs syntaxes :

Listing 22 – python/exemples-import.py

```
#Exemples d'importation de fonctions ou de modules compl
#import d'une librairie complète
import os
os.chdir('..') #change dossier de travail:..=dossier par
print(os.getcwd())
#import d'une (sous.)librairie complète sous alias
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,np.pi,100) #100 floattants  entre 0 et
y = np.sin(x) #Leurs sinus
plt.plot(x,y) #Le graphe de la fonction sinus sur [0,pi]
#import d'une constante ou fonction particulière
from string import ascii_letters
print(ascii_letters)
```

Modules externes : fonctions et constantes

Ce qu'on voit dans ces syntaxes, ce sont plusieurs façons d'affecter des identificateurs aux objets Python présents dans ces modules externes, la question de l'espace de nommage des objets est en jeu.

1. import complet : un objet est reperé/identifié sous la forme

   `nom_module.nom_objet`

2. import complet avec alias : un objet est reperé/identifié sous la forme

   `alias.nom_objet`

3. import d'un objet particulier : l'objet est reperé/identifié sous la forme

   `nom_objet`

Modules externes : fonctions et constantes

Vous pouvez écrire vos propres modules ! Il suffit de prendre un script standard et de placer la zone principale du script sous une conditionnelle :

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

Si ce script (appelons le `mon_module.py`) est présent dans le répertoire de travail, un autre script peut l'importer et utiliser les objets qui y sont définis.

Modules externes : fonctions et constantes

Listing 23 – python/monmodule.py

```
#Un module personnel
def foo(s):
    """
    Entrée : s une 'string'
    Sortie : la chaine s augmentée d'un espace, répétée
    """
    return (s+' ')*4
#Ceci est exécuté lors de l'importation
print(foo("Ciao"))
print("Je suis", "'"+__name__+"'",",", \
      "Suis-je un module ? ",not(__name__ == '__main__'))
#Zone de test
if __name__ == '__main__' :
    #Ceci n'est pas exécuté lors de l'importation
    print("Tests")
    print(foo("Bonjour"))
```

Modules externes : fonctions et constantes

Listing 24 – python/scriptmaitre.py

```
#Utilisation d'un module perso: script appelant
import monmodule
print(monmodule.foo("Hallo"))
```