PRISE EN MAIN DE PYTHON

Installation du logiciel

- Télécharger Pyzo à l'adresse suivante : http ://www.pyzo.org/start.html et l'installer. Pyzo est un environnement de programmation disponible pour Windows, Linux et OSX.
- Installer miniconda.
- Lancer Pyzo. L'environnement miniconda est généralement détecté automatiquement, il vous suffit alors d'accepter de l'utiliser.
- Pour mettre Pyzo en français, aller dans le menu *Settings, select language* et choisissez *French*. Il faut redémarrer Pyzo pour que cette modification soit prise en compte.

Instructions et fonctions

Python est un langage basé sur les instructions, qui sont des actions qui vont changer l'état du système lorsqu'elles vont être exécutées.
Un exemple fondamental d'instruction est l'**affectation** : le symbole `=` permet d'affecter une valeur, à droite, à une variable, à gauche :

```
>>> x = 2
>>> x
2
>>> y = 1
>>> y = x
>>> y
2
```

Une instruction simple est contenue sur une seule ligne. On peut utiliser le symbole \ en fin de ligne pour poursuivre l'écriture sur la ligne suivante.

La plupart des instructions que nous écrirons dans nos fichiers seront rassemblées dans des **fonctions**, dans l'objectif de factoriser (réutiliser ce qui a déjà été écrit) et de structurer le code. Lors de la définition d'une fonction, on donne son nom, le nom de ses arguments formels (possiblement aucun) et son corps, dans lequel le mot-clé `return` permet de renvoyer un résultat. Par exemple :

```
def moyenne(a,b):
  m = (a+b) / 2
  return m
```

Un `return` termine immédiatement l'exécution d'une fonction, y compris au milieu d'une boucle. Une fonction peut ne pas renvoyer de résultat, on la qualifie alors de **procédure**.

Pour appeler une fonction, on lui donne des arguments effectifs, c'est-à-dire qui ont une valeur au moment de l'exécution :

```
>>> x = 2
>>> moyenne(1,x)
1.5
```

Il est possible et souvent pertinent d'appeler des fonctions déjà écrites dans la définition de nouvelles fonctions. On peut également utiliser les fonctions natives de Python, comme la fonction `abs` donnant la valeur absolue, ou les fonctions de bibliothèques préalablement importées.

Le symbole # permet d'ajouter des **commentaires** dans les programmes :
`# commentaire sur le programme`.
Si on veut ajouter un commentaire sur plusieurs lignes on peut l'entourer avec `'''` :

```
'''
Si on a besoin de raconter sa vie, le temps qu'il fait,
le menu de la cantine, on peut faire comme ca.
'''
```

Pour afficher une valeur, on utilise la procédure `print` :

- Attention à ne pas confondre une variable `a` et une chaîne de caractères `'a'` :

```
>>> a=1
>>> print(a)
1
>>> print('a')
a
```

- Il est possible d'afficher les valeurs de variables au sein même de messages : on alterne alors les chaînes de caractères et les variables :

```
>>> a, b = 1, 10
>>> print('Le nombre', a, 'est plus petite que le nombre', b)
Le nombre 1 est plus petite que le nombre 10
>>> a=5
>>> print('a*a vaut', a**2)
a*a vaut 25
```

- Si on souhaite que notre chaîne de caractères contienne un retour à la ligne, on utilise la commande `\n` :

```
>>>print('ligne1\nligne2')
ligne1
ligne2
```

- Attention, un `print` ne peut pas remplacer un `return` dans une fonction. Afficher une valeur à l'écran ne permet pas de la transmettre à la suite du programme.

Types en Python

Types simples

- nombres entiers (positifs ou négatifs) : `int`
- nombre à virgules flottante (ou nombres flottants) : `float`
  Exemples : `3.1` : $3,1$ `2.4e5` : $2,4 \cdot 10^5$ `2.` : 2 (vu comme un flottant)
- booléens : `bool` ; les valeurs booléennes sont `True` (vrai) et `False` (faux)

Les opérations sur les nombres :

- opérations standards : `+` `-` `*` `/`
- le quotient de la division euclidienne de a par b : `a//b`
- le reste de la division euclidienne de a par b : `a%b`
- a puissance b : `a**b`

Les opérations standards sur les booléens : `or` (ou), `and` (et), `not` (non)
Les comparateurs renvoient un booléen :

- `a == b`, `a != b` : Tests d'égalité et de différence (à éviter sur les flottants)
- `a<b`, `a>b`, `a<=b`, `a>=b` : Tests d'inégalité, stricte ou large

Types itérables

Les objets de ces types peuvent contenir plusieurs éléments. Ils sont dit **itérables** car on peut parcourir leurs éléments à l'aide d'une boucle `for`.

- Tuples : `tuple`
  - On peut créer un tuple avec ou sans parenthèses. `M = (2, 3, 8)`
    On crée le même tuple si on omet les parenthèses : `M = 2, 3, 8`
  - Chaque élément du tuple a un indice (le premier élément étant d'indice 0). La syntaxe `M[i]` donne la valeur de l'élément d'indice i du tuple `M`.
  - `a, b, c =M` : dépaquette un tuple, ici de longueur 3. On récupère dans chaque variable un élément du tuple. Il faut connaître à l'avance le nombre d'éléments du tuple.
  - `len(M)` donne la longueur (le nombre d'éléments) de `M`.
  - `M+T` donne la **concaténation** de `M` et `T`.
    Ainsi `(0, 1) + (2, 3)` vaut `(0, 1, 2, 3)`.
  - `M*n` donne le tuple formé de n concaténations de `M`.
    Ainsi `(2, 3)*3` vaut `(2, 3, 2, 3, 2, 3)`.
  - `x in M` vaut `True` si `x` apparaît dans `M`, et `False` sinon.
  - Une fois un tuple créé, on ne peut plus changer ses éléments.
- Chaînes de caractères : `str`
  - peuvent s'écrire entre apostrophe `'` ou guillemets `"` : `'toto'`, `"tata"`.
  - Les chaînes de caractères fonctionnent comme des tuples dont chaque élément est un caractère.
  - les test d'inégalité se font sur l'**ordre lexicographique** (l'ordre du dictionnaire).
- Listes : `list`
  - Les listes s'écrivent entre crochets : `[4, 5, 8]`.
  - La liste vide s'écrit `[]`.
  - Toutes les opérations décrites sur les tuples sont disponibles sur les listes.
  - Contrairement à un tuple, on peut modifier un élément d'une liste, avec la syntaxe `L[i] = v`. Cela ne marche que s'il existe déjà un élément d'indice i.
  - `L.append(e)` : ajoute l'élément `e` à la fin de la liste `L`.
- Les dictionnaires : `dict`
  - Un dictionnaire associe des valeurs à des clés :
    dans `{'Alice' : 4, 'Bob' : 2}`, à la clé `'Alice'` est associée la valeur 4.
  - Le dictionnaire vide s'écrit `{}`.
  - `D[c]` : valeur associée à la clé `c` dans le dictionnaire `D`.
  - `c in D` : vaut `True` si la clé `c` apparaît dans le dictionnaire `D` et `False` sinon.
  - `D[c] = v` : si la clé `c` est déjà présente dans `D`, sa valeur associée est modifiée à `v`. Sinon, la clé `c` est ajoutée à `D`, et la valeur `v` lui est associée.

Les listes et dictionnaires sont dits **mutables** : leur contenu peut être modifié. Une fonction prenant en argument un objet mutable a la possibilité de modifier la valeur de cet objet pour la suite de l'exécution.

Structures de contrôle

Branchement conditionnel

On utilise `if`, `else` ainsi éventuellement que `elif` s'il y a plus de deux alternatives :

```
if x == 3:            # si x=3
    y = 0             # on va dans cette branche
elif 3 < x <= 4:      # sinon, si 3<x<=4
    y = x+2           # on va dans cette branche
elif 4 < x < 5:       # sinon, si 4<x<5
    y = x-2           # on va dans cette branche
else:                 # sinon
    y = 3*x           # on va dans cette branche
```

On peut toujours omettre la partie `else` si on n'a aucune instruction à associer à ce cas.

Boucle inconditionnelle

Pour parcourir les entiers de 0 à $n-1$, on utilise un `for` sur un `range(n)` :

```
x=0                  # affecte 0 a la variable x
for i in range(6):   # boucle faisant varier i de 0 inclus a 6 exclu
    x = x + i        # ajoute i a x pour chacun des 6 passages dans la boucle
print(x)             # suite du programme, ici on affiche 15 dans la console
```

On peut également parcourir les éléments d'un tuple, d'une chaîne ou d'une liste :

```
r = 0
for x in L:
    r = r+x
```

Dans le cas d'un dictionnaire, ce sont les clés qui sont parcourues :

```
for c in D:
    corps de la boucle
```

Boucle conditionnelle

Une boucle `while` permet de répéter un corps de boucle tant qu'une condition donnée reste vraie :

```
x = 1
while x < 10**6:
    x = 2*x
```

Contrairement à une boucle `for`, il n'est en général pas évident de prédire combien de passages vont être faits dans le corps d'une boucle `while`.

Bibliothèques usuelles

Pour obtenir de l'**aide** sur une fonction on peut utiliser la fonction `help` :

```
>>>help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Bibliothèque math

Elle contient de nombreuses fonctions et constantes mathématiques

- On peut l'importer à l'aide de l'instruction : `import math`.
- On pourra utiliser les fonctions classiques : `math.cos`, `math.sin`, `math.tan`, `math.asin`, `math.acos`, `math.atant`, `math.exp`, `math.log` (logarithme népérien), `math.log10` (logarithme décimal) ; les constantes `math.pi`, `math.e`...
- La fonction racine carrée s'obtient avec `math.sqrt` (pour *square root*).
- On peut aussi importer la bibliothèque avec la commande : `from math import *`. Dans ce cas, on ne préfixe pas les noms de fonctions et variables par `math`.

  Exemple :

```
>>> from math import *
>>> print(cos(pi/2))
6.123233995736766e-17
```

  On remarque sur cet exemple que les calculs sur les flottants sont soumis à des approximations. Pour cette raison, lorsqu'on veut tester l'égalité de deux flottants `a` et `b`, on n'utilise pas `a == b` mais `abs(a-b) < epsilon`, où `epsilon` est une valeur proche de zéro choisie en fonction du contexte.

Bibliothèque numpy

Elle permet de faire du calcul scientifique et de manipuler des tableaux à plusieurs dimensions :

- Pour importer la bibliothèque `numpy` on peut écrire `import numpy as np`. `np` est un alias : pour faire appel aux fonctions de la bibliothèque on écrira `np.cos` plutôt que `numpy.cos` (c'est plus court).
- `np.array(liste)` : permet de créer une matrice (de type tableau) à partir d'une liste.
- `L1 = np.array([1, 2, 3])` : transforme la liste en un tableau numpy. Ici `L1` est un tableau d'une ligne contenant 3 valeurs.
- Les opérations sur les listes sont disponibles sur les tableaux `numpy`.
- `L2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])` : `L2` est un tableau de 2 lignes et 4 colonnes.
- `np.shape(L2)` : retourne un tuple donnant les dimensions du tableau `L`, soit pour l'exemple précédent le couple `(2,4)`.
- `L[i, j]` : renvoie l'élément de la ligne *i* et de la colonne *j* de la matrice `L`.
- `L[i, :]` : renvoie la ligne d'indice i de `L`.
- `L2[:, i]` : renvoie la colonne d'indice i de `L2`.
- `np.zeros((n,m))` : crée une matrice de *n* lignes et *m* colonnes dont tous les éléments sont égaux à zéro.
- `np.eye(n)` : crée la matrice identité de taille `n`.
- `np.linspace(min, max, nbElements)` : renvoie un tableau de `nbElements` nombres espacés régulièrement entre `min` et `max`.
- Lorsqu'on applique une fonction numérique de `numpy` sur un tableau `numpy`, la fonction est appliquée sur chaque élément du tableau, et on obtient le nouveau tableau résultant.

```
>>> np.sin(L2)
array([[ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ],
       [-0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825]])
```

Bibliothèque matplotlib.pyplot

Elle permet de tracer des graphiques :

- On peut l'importer par `import matplotlib.pyplot as plt`.
- `plt.plot(Lx, Ly` permet de tracer la courbe passant par les points dont les abscisses sont dans la liste `Lx` et les ordonnées sont dans la liste `Ly`.
- `plt.show()` : affiche la figure à l'écran.
- Par exemple, pour afficher la courbe de la fonction carrée sur $[-4, 4]$:

```
import matplotlib.pyplot as plt
import numpy as np
Lx = np.linspace(-4,4,10**6)
Ly = [x**2 for x in Lx]
plt.plot(Lx,Ly)
plt.show()
```

- On peut préciser les paramètres du tracé, sur le modèle :
  `plt.plot(Lx, Ly, color ='r', linewidth =0.01, marker ='.', linestyle ='--')`.
  - `color` : choix de la couleur (`'r'` : rouge, `'g'` : vert, `'b'` : bleu, `'black'` : noir.
  - `linewidth` : épaisseur du trait.
  - `marker` : différents symbole pour l'emplacement des points : `'+'`, `'.'`, `'o'`, `'v'`.
  - `linestyle` : style de la ligne : `'-'` ligne continue, `'--'` discontinue, `':'` : pointillés.
- `plt.grid()` : affichage de la grille.
- `plt.title('Titre')` : ajout d'un titre.
- `plt.xlabel('axe x')` : ajout de l'étiquette `axe x` en abscisse.
- `plt.ylabel('axe y')` : ajout de l'étiquette `axe y` en ordonnée.
- `plt.axis([xmin, xmax, ymin, ymax])` : précise les bornes pour les abscisses et les ordonnées.
- `plt.clf()` : efface les tracés précédents.

Compléments

Assertions

Une **assertion** est une instruction permettant de vérifier explicitement une supposition. La syntaxe est **`assert`** `c`, où `c` est à valeur booléenne.
Par exemple, si on définit une fonction `f` prenant un argument entier `n` en supposant qu'il est positif, on pourra commencer la définition de `f` par :

```
def f(n):
    assert n>=0
    corps de la fonction
```

De cette façon, si `f` se retrouve appelée sur un entier strictement négatif, la condition du **`assert`** sera fausse, ce qui provoquera un arrêt de l'exécution :

```
>>> f(-2)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 9, in f
    assert n>0
AssertionError
```

À l'inverse, quand la condition de l'assertion est respectée, l'exécution continue sans autre changement.

Portée des variables

Dans la définition d'une fonction, les variables qui y sont définies (dont les arguments formels) sont qualifiées de **variables locales**. Cela signifie que ces variables n'ont une valeur qu'au sein d'un appel de cette fonction. Dès que l'appel est terminé, ces variables sont inaccessibles.
À l'inverse, une variable définie à l'extérieur d'une fonction est dite **globale**. On peut y accéder partout, y compris à l'intérieur d'une fonction.

```
x = 4              # x est une variable globale
def f(a):          # a est un argument formel, donc local
    b = 3          # b est une variable locale
    return a+b+x   # x est accessible au sein de f
```

Après exécution du fichier :

```
>>> f(5)
12
>>> x
4
>>> a
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'a' is not defined

>>> b
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'b' is not defined
```

Une fonction contribue donc au reste du programme en renvoyant un résultat, avec un `return`, ou en modifiant un argument mutable.