Informatique

Rappels

PSI : Lycée Rabelais

Pré-requis

Cours d'informatique de première année

Objectifs

Installer `Edupython`

Maitriser l'utilisation "pratique" de `Python`

Réviser la manipulation des fonctions élémentaires de `Python`

1 Utilisation pratique

`Python` est un langage "interprété", cela signifie que les instructions sont exécutées au fur et à mesure par demande de l'utilisateur. Pour communiquer avec `Python`, on pourra utiliser deux méthodes.

- Par utilisation de la fenêtre *shell* (aussi appelée "console"). Dans ce cas, il faut saisir les instructions une par une et les valider ensuite. Cette solution peut être utile pour tester une fonction ou une instruction. Dès que les instructions deviennent complexes, cette méthode est trop laborieuse pour être efficace.
- Par utilisation d'un *script*. L'utilisation d'un *script* permet simplement d'écrire dans un fichier la totalité des instructions à effectuer. Cela permet donc de regrouper et de structurer les différentes instructions souhaitées. Ce fichier devra **toujours** être enregistré. Il sera également possible, *via* certaines fonctions, d'appeler d'autres fichiers. Il faudra donc créer un répertoire dans lequel seront présents tous les fichiers nécessaires au bon fonctionnement de votre *script*.

L'utilisation de `Python` étant répandue, il existe des *Environnements de Développement Intégrés* (EDI, ou IDE en anglais) qui fournissent tous les outils nécessaires à la programmation sur `Python`. Au lycée, nous utiliserons `Edupython`, `Pyzo` ou `Spyder`. **Vous trouverez, dans le TP n°0, la procédure à suivre pour installer `Python` sur vos ordinateurs. Il est indispensable d'avoir un ordinateur avec une installation de `Python` pour travailler l'informatique.**

Le noyau de `Python` connaît très peu de fonctions (mathématiques, graphiques ou numériques). Il faudra donc, très souvent, demander l'utilisation d'outils supplémentaires qui sont "rangés" dans des bibliothèques (aussi appelées modules).

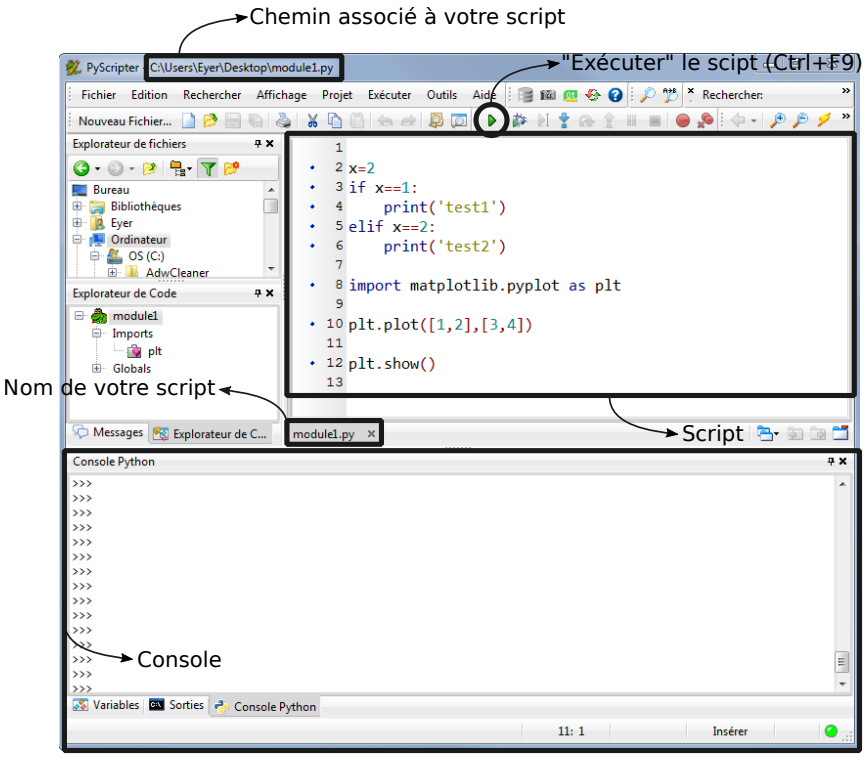
Les bibliothèques les plus utilisées, dans le cadre de la prépa, seront :

- **math**, on y trouvera notamment les fonctions `sqrt` (racine carrée), `exp` (fonction exponentielle), `log` (fonction logarithme népérien), `pi` (la valeur de π), etc...
- **matplotlib**, qui permet de charger les outils graphiques nécessaires aux tracés de courbes.
- **numpy**, qui contient des outils permettant de simplifier les calculs numériques *via* l'utilisation de certaines fonctions utiles.

Deux syntaxes sont possibles pour les importations :

1) Importation d'un module avec un alias : par exemple, `import matplotlib.pyplot as plt` permet d'utiliser les commandes de `pyplot` en les faisant juste précéder de l'alias et d'un point (`plt.plot()`, `plt.show()`, etc...). L'importation sans alias est possible, mais il faut alors recopier le nom complet du module à chaque invocation.

2) Importation de certaines fonctions d'un module : par exemple, `from math import pi,sin,cos` permet d'utiliser `pi`, `sin` et `cos` tels quels.



Exemple d'interface : ici, `Edupython`

2 Types prédéfinis en `Python`

2.1 Booléens : `bool`

Les variables de type `bool` peuvent prendre deux valeurs, `True` et `False`. Ces variables sont souvent obtenues comme résultat de l'évaluation d'une expression booléenne, comme `n==0` ou `abs(x)<=e`.

Remarques :

- Ne pas confondre le test `n==0` avec l'affectation `n=0`.
- Le connecteur "différent de" s'écrit `!=`.
- Sur `Python`, l'évaluation d'une expression booléenne (de gauche à droite) est interrompue dès que le résultat est acquis.

2.2 Entiers : `int`

`Python` sait effectuer des opérations élémentaires sur les entiers. Par exemple pour deux entiers `a` et `b` (où `b` est non nul), `a//b` et `a%b` donnent respectivement le quotient et le reste de la division euclidienne de `a` par `b`.

Exemple : Que renvoient `11//4` et `11%4` ?

2.3 Flottants : `float`

Pour faire du calcul numérique, on utilisera essentiellement des objets de type flottants. Pour effectuer ses calculs, `Python` va coder les valeurs sur 64 bits. La précision dans le système décimal sera d'environ 16 chiffres significatifs.

Le calcul étant effectué en base 2, les calculs resteront approchés. La calcul de `0.1 + 0.2 - 0.3` renverra ...

... `5.551115123125783e-17` !

2.4 Listes : `list`

`Python` gère bien les listes : il s'agit d'un object contenant des éléments ordonnés. Ces éléments sont spécifiés entre crochets et séparés par des virgules. Les éléments d'une liste ne sont pas nécessairement distincts (contrairement aux éléments d'un ensemble). Ils ne sont pas nécessairement du même type.

Les listes Python s'apparentent aux tableaux des autres langages et nous les appellerons parfois ainsi. **La maitrise de la manipulation des listes est essentielle en informatique !**

2.5 Chaînes de caractères : `str`

Les chaînes de caractères peuvent être considérées comme des "listes de caractères", au sens où l'on a les mêmes règles d'indexation que pour les listes.

La grande différence réside dans le fait que les chaînes de caractères ne sont pas modifiables.

Par exemple, `s1='scrab'` et `s2="ble"` définissent deux chaînes de caractères. Pour écrire *scrable*, il faudra donc créer une nouvelle chaîne de caractère : `s3 = s1 + s2` (on concatène `s1` et `s2`).

On remarquera que `s1[3]` renvoie bien `'a'` mais `s1[3]='u'` déclenche une erreur (la chaîne n'est pas modifiable !).

3 Éléments de programmation

3.1 Indentation

De nombreuses instructions "encapsulent" d'autres instructions, notamment les tests, les boucles, les définitions de fonctions, etc... Dans ce cas, l'instruction "mère" est écrite sur une ligne terminée par deux points (':') et les instructions encapsulées sont regroupées dans un bloc, formé des lignes suivantes, écrites avec un niveau d'indentation supplémentaire. La fin du bloc est marquée par le retour au niveau d'indentation de l'instruction "mère". On peut ainsi imbriquer plusieurs niveaux de blocs.

```
if n%2==0:
  return 'pair'
else:
  return 'impair'
```

3.2 Définition d'une fonction

On définit la fonction f par l'instruction suivante en en-tête `def f(x,y...):`, celle-ci sera suivie du bloc formé par les instructions à exécuter lors de l'appel de ladite fonction (le corps de la fonction).

x et y sont ici les paramètres de la fonction.

On peut par exemple définir la fonction carrée qui renvoie le carré d'un flottant :

```
def carre(x):
  return x**2
```

Pour obtenir le carré de 2, il faudra donc écrire `carre(2)`. L'instruction `return`, rencontrée n'importe où dans le corps d'une fonction, interrompt l'exécution de l'appel de ladite fonction et renvoie le résultat demandé.

3.3 Instructions conditionnelles

Pour imposer une condition, on utilisera toujours la structure suivante :

```
if expr_bool_1:
  bloc_1
elif expr_bool_2:
  bloc_2
elif ...
else:
  bloc_sinon
```

Les clauses `elif` et `else` sont facultatives.

Les test sont évalués de haut en bas. Si l'un des blocs sous condition est exécuté, on se branche aussitôt après le dernier bloc de l'instruction conditionnelle.

Petit exemple...

Que renvoie les instructions suivantes ?

```
a=6
b=15
if a>5:
    b=b-4
if b>=10:
    b=b+1
print(b)
```

et...

```
a=6
b=15
if a>5:
    b=b-4
elif b>=10:
    b=b+1
print(b)
```

3.4 Boucle `for`

Ce type de boucle sera structurée de la manière suivante :

```
for i in listei:
  bloc
```

Ce type de boucle permettra de réaliser l'ensemble des instructions définies dans `bloc` pour toutes les valeurs de `i` proposées dans la liste `listei`.

Le cas le plus fréquent est celui où la liste `listei` est définie par `range(n)` où `n` est un entier. Dans ce cas i prendra donc les valeurs comprises dans l'intervalle $[\![0; n-1]\!]$ (**attention à l'inclusion à gauche et à l'exclusion à droite**) !

Trois exemples

```
L1 = ['a','b','c','d']
L2 = []
for i in L1:
  L2.append(i+i)

print(L2) ## Quel est le résultat renvoyé ?
```

_____

```
L3 = [10,20,30,40,50,60,70]
L4 = [0,len(L3)//2,len(L3)]
L5 = []
for j in L4:
  L5.append(j)

print(L5) ## Quel est le résultat renvoyé ?
```

_____

```
L6 = [4,5,6,7]
res = 0
for j in range(1,len(L6)):
  res = res + L6[j]

print(res) ## Quel est le résultat renvoyé ?
```

3.5 Boucle `while`

Ce type de boucle sera structurée de la manière suivante :

```
while expr_bool:
  bloc
```

Les instructions contenues dans `bloc` seront exécutées tant que l'évaluation de `expr_bool` renvoie `True` ; dès qu'elle renvoie `False`, l'exécution se poursuit à la suite de `bloc`.

On pourra également utiliser les instructions `return ...` ou `break` si la sortie prématurée de la boucle est nécessaire.

Petit exemple...

Complétez le code suivant pour que la fonction `fonct` crée une nouvelle liste `L2` qui ajoute 1 à chaque élément de la liste `L1` tant que l'élément est positif.

```
def fonct(L):
  k = 0
  n = len(L)
  L2 = ..............
  while ....... and ....... :
        ..............
       k +=1
  return L2


```

```
L1 = [1,2,3,4,-1,5]
print(fonct(L1)) ## Quel est le résultat renvoyé ?


L2 = [1,2,3,4,5]
print(fonct(L2)) ## Quel est le résultat renvoyé ?
```

3.6 Construction d'une liste par compréhension

Cette possibilité permet la construction d'une liste de manière très concise à l'aide d'une boucle `for` éventuellement suivie d'un test.

Petit exemple...

On donne la liste suivante construite par compréhension :

```
liste = [i for i in range(101) if i%2==0]
```

Que contient cette liste ?

4 Tracé de courbes : `pyplot`

Les outils nécessaires au tracé de courbes sur `Python` sont disponibles dans la bibliothèque `matplotlib.pyplot`. Pour l'importer, on utilisera donc l'instruction `import matplotlib.pyplot as plt`.

La commande de base est `plt.plot(X,Y)` qui relie les points de coordonnées $(X[k], Y[k])$, X et Y étant deux tableaux de même taille (à une dimension). La répétition de cette instruction permettra de tracer plusieurs courbes sur le même graphique.

La commande `plt.show()` affiche le graphique à l'écran. Par défaut, la fenêtre d'affichage (les intervalles pour les valeurs des abscisses et des ordonnées) est déterminée automatiquement, de même que les unités et graduations sur les axes.

Pour ajuster les différents éléments du graphique, on pourra utiliser les commandes suivantes :

`plt.grid(True)` ajoute une grille en lignes pointillées à chaque graduation.
`plt.title('Titre')` définit le titre du graphique.
`plt.xlabel('Nom axe x')` définit l'étiquette de l'axe des abscisses (même syntaxe pour l'axe des ordonnées avec `plt.ylabel('Nom axe y')`).
`plt.xlim(a,b)` définit l'intervalle $[a, b]$ pour les valeurs de l'axe des abscisses (même syntaxe pour l'axe des ordonnées avec `plt.ylim(c,d)`).
`plt.axis('equal')` impose la même échelle sur l'axe des ordonnées et des abscisses.

Pour sauvegarder le graphique, on utilisera l'instruction `plt.savefig(nom_de_fichier)`. Le fichier devra prendre en compte l'extension `.pdf` (pour obtenir un fichier pdf) ou `.png` (pour obtenir une image au format png).

Il est également possible de rajouter des options sur chaque tracé. D'une manière générale, il faudra utiliser la syntaxe suivante : `plt.plot(X,Y,option1,option2,...)`. Les options les plus courantes sont les suivantes :

- `color='black'` permet d'obtenir un tracé en noir. On pourra aussi utiliser les couleurs suivantes : `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`.
- `linestyle='-'` donne un trait continu, `'--'` des tirets, `':'` des pointillés, `'-.'` des tirets et points alternés.

- `linewidth=2` donne un trait plus épais, la valeur par défaut est 1.
- `marker='x'` matérialise la position de chaque point par une croix, il existe beaucoup d'autres styles de marqueurs : `o`, `+`, `.`, `*`...
- `label='Nom de la courbe'` définit le nom associé à un tracé. Pour afficher le nom, telle une légende, il faudra ajouter la commande `plt.legend(loc=j)` où `j` permettra de choisir la position de la légende sur le cadre du graphique.