Chapitre 1

Programmation Python - 1

1.1 Mise en place

1.1.1 Système informatique du lycée

ordinateurs/terminaux. Utilisation d'Ubuntu. Authentification requise. Identifiants personnels, compte local/local. Passage par un serveur. Changement du mot de passe, pas sur le système Ubuntu mais sur l'intranet, par un navigateur (Firefox).
Serveur Camille. Arborescence, droits d'accès.

1.1.2 Utilisation d'Ubuntu

Aller voir sur https://doc.ubuntu-fr.org/unity.
Parler des "paramètres système".
Proposer l'utilisation des espaces de travail.

1.1.3 Lancer Python

Choix du répertoire courant et configuration

On programmera en Python .
Il existe plusieurs versions de Python . Le passage de(s) version(s) 2 à la version 3 a modifié beaucoup de points, qui ont une influence sur la pratique même simple de Python . Aucune version n'étant privilégiée officiellement, nous détaillerons les différences auxquelles nous devrons prendre garde.
Pour la semaine prochaine, charge à vous d'installer le programme sur votre machine personnelle. Je n'assurerai pas de maintenance concernant l'installation et la pérennité de votre système d'exploitation.
Sur les machines avec Ubuntu, et donc notamment les machines du lycée, pour ouvrir une session Python , il faut ouvrir un terminal et taper `python`. La session que vous aurez ouverte aura alors votre répertoire personnel (`/home`) pour *répertoire courant*, c'est-à-dire le répertoire à partir duquel l'exécuteur Python va lancer ces commandes.
Afin de ne pas forcément travailler dans le répertoire personnel `/home`, on peut

1. ouvrir un terminal,
2. avec les commandes `ls`, `cd`, `cd ..` et le raccourci `*`, on change le répertoire courant,
3. une fois dans le répertoire désiré : `python` qui ouvre une session Python .

Ou alors, on peut ouvrir le navigateur de dossier (`nautilus`) et aller jusqu'au répertoire désiré et faire un clic-droit puis *Ouvrir dans un terminal*.
Ensuite, une fois dans Python :

4. l'invite de commande Python est `>>>`,
5. on peut alors exécuter des commandes Python , mais de préférence des commandes tenant sur une seule ligne.

6. pour exécuter des commandes plus longues, pour éditer des programmes ou des successions de commandes, on lance un éditeur de texte (`gedit`) et on crée dans le répertoire courant un fichier avec l'extension `.py`, pour que l'éditeur colorie la syntaxe PYTHON .
7. pour exécuter le fichier `toto.py` qui se trouve dans le répertoire courant, on tape dans la session PYTHON `execfile ('toto.py')`.
8. pour fermer la session PYTHON , on tape `quit()` ou `exit()`.

Une dernière option est d'utiliser un logiciel dont l'environnement intègre l'interaction en ligne de commande et un éditeur de texte, comme `Idle`, la problématique du répertoire courant étant la même.

Entrée-sortie

Une instruction PYTHON s'écrit à la suite de l'invite (ou *prompt*)`>>>`, et l'interface renvoie le résultat de la commande :

```
>>> 1 + 2
3
```

1.2 Affectation

L'instruction *variable* `=` *valeur* affecte la valeur *valeur* à la variable *variable*, on parle aussi de définition. L'affectation a deux propriétés essentielles :

— la définition est dite *globale* lorsque l'affectation est isolée ;
— la définition est dite *locale* lorsqu'elle a lieu dans une fonction. Une définition locale d'une variable ne change pas son éventuelle définition globale.

On peut changer la valeur liée à un nom en utilisant le nom lui-même :

```
>>> x=1
>>> x
1
>>> x=x+1
>>> x
2
```

1.3 Types

Les données utilisées en PYTHON ont un type. L'instruction `type(`*valeur*`)` renvoie le type de *valeur*. Sur un type donné, on dispose de différents outils : des opérateurs, des fonctions, des méthodes. On détaillera la plupart de ceux-ci plus tard.

1.3.1 Types simples

`int` entiers compris entre -2^{31} et $2^{31}-1$. Pour des entiers plus grands, le type `long` est utilisé.

```
>>> 2**31
2147483648L
>>> type(2147483648)
<type 'long'>
>>> type(-2147483647)
<type 'int'>
```

On dispose des opérateurs `+ - * // **`. (division euclidienne et puissance).
`float` nombres à virgules flottantes, limites dépendant de la machine (voir cours sur la précision machine).
On dispose des opérateurs `+ - * / **`.

Remarque PYTHON est très arrangeant, lorsqu'un calcul nécessite de changer de type, il le change de lui-même.
Remarque division euclidienne avec `/` en PYTHON 2.
`bool` booléens, qui peut prendre deux valeurs `True` ou `False`.
On dispose des opérateurs `or and not`.
`string` chaînes de caractère, notées avec des apostrophes ou guillemets (pas de double apostrophe).
On dispose de l'opérateur `+` (concaténation).

1.3.2 Produit cartésien

On peut créer des n-uplets composées d'objets de différents types. Le type résultant est le type `tuple`.

```
>>> a=(1,2)
>>> type(a)
<type 'tuple'>
```

On peut mélanger les types dans un `tuple` :

```
>>> b= (1,2,'a',True)
>>> type(b)
<type 'tuple'>
```

On peut parfois négliger les parenthèses, la virgule indiquant seule le type.

```
>>> a= 1,2
>>> type(a)
<type 'tuple'>
```

Attention par contre : `type(1,2)` renvoie une erreur car la fonction `type` attend un seul argument. `type((1,2))` renvoie bien `<type 'tuple'>`.

1.3.3 Type fonctionnel

La syntaxe pour créer une fonction est la suivante
`def` *fonction*`(`*argument1* `,`*argument2*`,...`*argumentn*`):`
    instructions
    `return` *valeur*
Une telle fonction renvoie la valeur *valeur*
La commande `return` peut être utilisée à plusieurs reprises, notamment dans le cas de l'usage d'alternatives.
Le point fondamental de la syntaxe en PYTHON est **le respect de l'indentation**. Pour savoir quand commence et surtout quand finit le corps de la fonction, PYTHON se sert de l'indentation. Il est donc impératif de la respecter.

```
>>> def f(a,b):
        return a+b
>>> type(f)
<type 'function'>
```

On dispose aussi de la commande `lambda` :

```
>>> g = lambda a,b : a*b
>>> g(2,3)
6
```

Le type fonction est un type comme les autres, au sens qu'on peut par exemple s'en servir comme d'un argument dans une fonction

```
>>> def decalage(f):
       lambda x : f(x+1)
>>> def composee(f,g):
       return lambda x:f(g(x))
```

Les variables dans une fonction

— Si on écrit une définition dans le corps d'une fonction, on obtient une définition locale.

```
>>> n=3
>>> def f(x):
       n=1
       m=2
       return x+n
>>> f(5)
6
>>> n
3
>>> m
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'm' is not defined
```

— Si on redéfinit une variable faisant partie des arguments (ce qu'on cherchera à éviter), on a à nouveau une définition locale.

```
>>> def ff(a,b):
       a=1
       return b+a
>>> ff(3,4)
5
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

— L'usage des variables globales dans une fonction est dynamique :

```
>>> m=2
>>> def g(a):
       return a+m
>>> g(1)
3
>>> m = 3
>>> g(1)
4
```

Remarque : Il faut toujours respecter le nombre d'arguments quand on appelle une fonction.

Les remarques dans les fonctions Si on dispose des remarques entre triples guillemets en début de fonction, servant à décrire la fonction et son usage, ces remarques seront affichées dans la fenêtre d'aide de Spyder.

1.3.4 Type liste

Une liste est un tableau dynamique, qui consiste en une suite de cases qui pointent vers un contenu, qui peut être de type quelconque et différent suivant les cases.
La syntaxe consiste en des crochets autour, avec des virgules séparant les éléments.

```
>>> a=[1,3.2,'a',True, lambda x:x**2]
```

On accède à la ième case avec l'instruction `tableau[i]`, l'indexation du tableau commençant à 0.

```
>>> a[0]
1
>>> a[4]
<function <lambda> at 0xa91a374>
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On peut changer la valeur d'un seul élément : `a[0]=5.4`.
On peut changer la longueur d'une liste en rajoutant des éléments avec la concaténation `+` :

```
>>> a=[0,True]+[4.3,(6,8)]
>>> a
[0, True, 4.3, (6, 8)]
```

L'allocation du contenu vers lequel les cases pointent est dynamique. Voici deux exemples :

```
>>> a=[1,2]
>>> b=a
>>> a[1]=3
>>> a
[1, 3]
>>> b
[1, 3]
```

et

```
>>> a=[1,2]
>>> b=[1,2]
>>> a[1]=3
>>> a
[1, 3]
>>> b
[1, 2]
```

Dans le premier cas, a et b ont des cases qui pointent vers le même contenu (les mêmes cases mémoire), alors que dans le deuxième cas, a et b ont des cases qui pointent vers des cases mémoires différentes, mais initialement avec des valeurs identiques.
A l'inverse, contrairement aux fonctions, les variables éventuellement utilisées dans la définition d'une liste sont évaluées au moment de la définition de la liste.

```
>>> x=1
>>> a=[x]
>>> a
[1]
>>> x=2
>>> a
[1]
```

On notera pour la suite plus particulièrement la fonction `range` :

```
>>> range(1,5)
[1, 2, 3, 4]
>>> range(4,11)
[4, 5, 6, 7, 8, 9, 10]
```

1.3.5 Le type type

Les types vus ci-dessus (`int, long, float, bool, string, tuple, function, list` sont également regroupés dans un type, qui se nomme `type`.

```
>>> type(bool)
<type 'type'>
```

1.3.6 Les `TypeError` qu'on peut rencontrer

```
>>> 2+[3]
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> 2(3)
TypeError: 'int' object is not callable
```

Compatibilité entre `float` et `int`.

1.3.7 Typage des fonctions

Python n'impose pas à une fonction que les types des arguments (variables d'entrée) ou des résultats renvoyés soient fixés :

```
>>> let detout(x):
       if type(x)==bool:
          return 1.
       if type(x)==float:
          return 0
       return True
```

Dans le cas où les instructions dans le corps de la fonction imposent le type des arguments, il faut le respecter. Sinon, on dit que la fonction est *polymorphe*.
De manière générale, les fonctions qu'on va créer vont nous servir dans d'autres fonctions, ou dit autrement, les programmes complexes vont souvent nécessiter la création de fonctions intermédiaires qu'on va composer entre elles. Dans ce cadre, le fait d'avoir en sortie de fonction plusieurs types différents peut poser un problème délicat. On essaiera donc autant que possible de créer des fonctions dont le type de sortie est toujours le même. On dira alors qu'on a des fonctions *bien typées*.

1.4 Conditions et boucles

1.4.1 Instruction conditionnelle

On a l'*alternative*
`if` *booléen*`:`
    instructions1
`else:`
    instructions2
Cette instruction exécute *instructions1* si *booléen* vaut `True`, et *instructions2* si *booléen* vaut `False`. La partie `else` est optionnelle, rien n'étant exécuté si *booléen* vaut `False`.
Encore une fois, l'indentation est primordiale.

1.4.2 Boucle inconditionnelle

On dispose de la boucle itérative
`for` *indice* `in` *liste*`:`
    instructions

Cette instruction exécute *instructions*, qui peut ou non dépendre de *indice*, pour chaque valeur de *indice* parcourant *liste*.

```
>>> a=0
>>> for i in range(1,4):
        a=a+i
>>> a
6
```

INDENTATION !

1.4.3 Boucle conditionnelle

On dispose de la boucle conditionnelle
`while` *booléen* `:`
instructions
Cette instruction exécute *instructions* tant que *booléen* vaut `True`, et s'arrête dès que *booléen* vaut `False`. Une telle boucle n'a de sens que si *instructions* peut faire changer la valeur de *booléen* en `False` au bout d'un certain nombre d'exécutions, sinon la boucle tourne sans fin.

```
>>> x=34.6
>>> n=0
>>> while n<x:
        n=n+1
>>> n
34
```

INDENTATION !