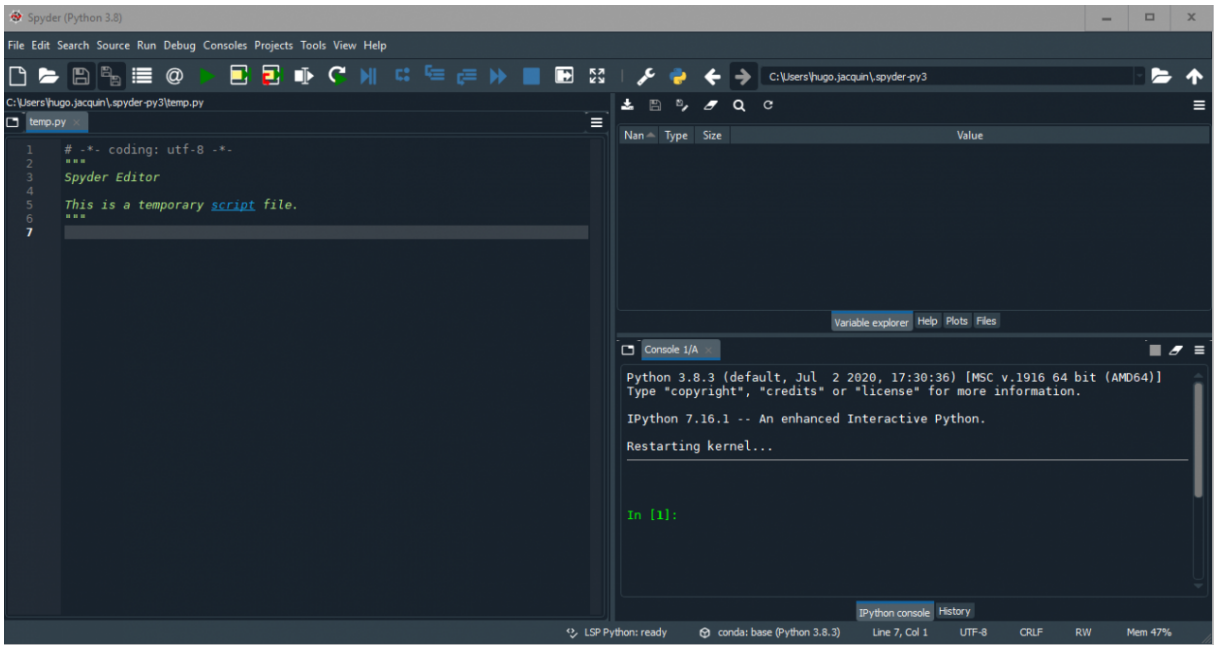
TP 1 : Python, variables, affectations, fonctions

1 Introduction

Pour écrire un programme informatique, on écrit un texte, qu'on appelle **code source**. L'ordinateur étant dépourvu de capacité de déduction et de raisonnement, il se contente d'exécuter fidèlement les **instructions** contenues dans ce texte, **dans l'ordre dans lequel elles apparaissent**. La plus grande rigueur est donc nécessaire, en un sens bien plus fortement qu'en mathématiques : en effet une **preuve** mathématique n'est rien d'autre qu'une tentative de **convaincre** le lecteur de la justesse de ce que l'on écrit. Si un caractère est mal orthographié, si une légère erreur se glisse dans la composition mathématique, erreur sans gravité ou bien qui est compensée par une autre erreur un peu plus loin, le lecteur comprendra tout de même la démonstration proposée. Un ordinateur est quant à lui incapable d'opérer ces corrections (on peut néanmoins inventer des programmes qui tentent de faire des corrections de programmes, mais c'est d'une incroyable complexité).

Il faut donc, au caractère près, rédiger **parfaitement** ses codes sources en Python comme dans les autres langages. C'est un formidable exercice d'entraînement à la rigueur dans l'écriture, bénéfique aussi en mathématiques. Nous verrons néanmoins que certaines notations adoptées en informatique peuvent être en conflit avec les notations mathématiques, il faudra donc prendre bien garde de ne pas mélanger les deux types de rédactions.

Sur le bureau, dans le dossier **Informatique**, ouvrir le logiciel **Spyder**. Une fenêtre apparaît, que vous pouvez mettre en plein écran. Cette fenêtre est séparée en trois grandes zones. À gauche, un **éditeur de texte**, dans lequel on peut écrire du code. En haut à droite, une partie que nous n'utiliserons pas tout de suite qui donne de l'aide ou des informations sur l'état de la mémoire (quelles variables ont déjà été crées, quelles sont leurs valeurs, etc...) c'est ici aussi que s'afficheront les représentations graphiques que nous demanderons à l'ordinateur d'afficher. En bas à droite **l'interpréteur** de commandes. Dans cette zone on peut taper directement des instructions et les faire exécuter instantanément par l'ordinateur.



Par la suite, il faudra toujours choisir si l'on rédige le code qui nous intéresse dans un fichier, afin de l'exécuter plus tard, ou bien si l'on entre les instructions directement dans la console. J'utiliserai deux formats pour indiquer quelle option choisir : une boîte grisée **avec trois chevrons** supérieur strict l'un à la suite de l'autre :

```
>>> print("hello world")
```

indique que l'instruction est à entrer directement dans l'interpréteur. Il faut ensuite appuyer sur la touche Entrée pour exécuter l'instruction. Une boîte grisée **sans chevrons** indique qu'il faut entrer le code dans l'éditeur de texte. Une fois le code écrit dans l'éditeur de texte, pour voir le résultat, il faut **exécuter** votre code. Pour cela, au dessus de l'éditeur de texte, vous trouverez une icône en forme de triangle vert. En cliquant dessus, le code que vous avez entré dans l'éditeur sera **exécuté** et le résultat s'affichera dans l'interpréteur. Par exemple, tapez le code suivant dans l'éditeur

```
1+1
print("well done")
```

puis exécutez le et observez le résultat. Chaque ligne du code a été exécutée, l'une après l'autre. Pour ce premier TP nous utiliserons principalement l'interpréteur. Dans les feuilles de TP, on écrira parfois à la fois le code à entrer dans l'interpréteur (juste après les trois chevrons), et le résultat affiché après calcul (sans chevrons). Par exemple

```
>>> print(2+3)
5
```

signifie que si l'on entre $2+3$ dans l'interpréteur et que l'on exécute en appuyant sur la touche Entrée, la console affichera 5.

2 Utilisation de Python comme d'une calculatrice

Dans un premier temps, on peut utiliser Python comme une calculatrice. Dans la console, on saisit une expression, Python en calcule la valeur et affiche le résultat. Il faut bien comprendre qu'il y a deux étapes du point de vue de l'ordinateur : le calcul et l'affichage. Seul la deuxième étape est visible. Tapez ces quelques commandes dans la console et observez le résultat.

```
>>> print(2+3)
>>> 2*5+6-(100+3)
>>> 7 / 2 ; 7 / 3
>>> 34 // 5; 34 % 5  # quotient et reste de la division euclidienne de 34 par 5
>>> 2 ** 7                  # pour un calcul de puissance (on n'utilise pas ^)
```

Nous avons utilisé le symbole dièse pour placer des commentaires dans les lignes de commande. Tout ce qui se situe à droite du symbole dièse est tout simplement ignoré par l'interpréteur Python (inutile donc de les recopier...). En appuyant sur la flèche du haut, la console fera apparaître à nouveau les commandes tapées précédemment.

3 Variables et affectations

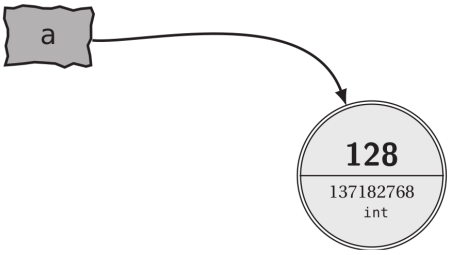
Lorsque l'on saisit un nombre dans la console Python, l'interpréteur créée un objet en mémoire, et lui attribue une **adresse mémoire**. Pour accéder à cette adresse, on peut utiliser la commande **id()**. On accède au type de l'objet en question avec la commande **type()**. Par exemple :

```
>>> 128, id(128), type(128)
(128, 137182768, int)
```

L'objet créé possède une **valeur** (ici 128), un identifiant (son adresse mémoire, ici 137182768) permettant de savoir où il est gardé en mémoire, et enfin un **type** (ici le type entier, **int**). La mémoire étant très grande (plusieurs dizaines de milliards de bits), une adresse dans la mémoire est nécessairement un très grand entier.

Afin d'éviter d'avoir à désigner notre objet 128 par son adresse mémoire, nous allons lui donner un nom plus simple à manipuler. On appelle cette opération une **affectation**. Cela se fait à l'aide de l'opérateur égal '=' :

```
>>> a=128
>>> a
128
>>>  a, id(a), type(a)
(128, 137182768, int)
>>> 2 * a
256
```



Ainsi à chaque nécessité d'utiliser le nombre 128, nous pourrons utiliser le la **variable** « a »à la place. Attention : l'opérateur égal en informatique n'a rien à voir avec le égal des mathématiques. Le égal informatique n'est pas symétrique :

```
>>> 128 = a
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

Essayez à présent la suite d'instructions suivante :

```
>>> b = a * 2
>>> b                       # resultat : 256
>>> b, id(b), type(b)       # resultat : (256, 137184616, int)
>>> a=0
>>> a, id(a), type(a)       # resultat : (0, 137180720, int)
>>> b                       # resultat : 256
```

Ici se trouve un point délicat en programmation. L'instruction $b = a * 2$ n'affecte pas à b le double de la valeur de a, et ce quelque soit la valeur de a au cours de la session Python. L'instruction d'affectation "=" procède en deux temps :

— L'expression située à droite du signe égal est **évaluée**, c'est-à-dire calculée en fonction de l'état de la mémoire **à cet instant**. Le résultat est un entier de type int, de valeur 256, et qui est alors placé en mémoire avec l'identifiant 137184616.
— Ensuite seulement, l'interpréteur Python affecte au nom de variable situé à gauche du signe égal l'objet obtenu après évaluation de l'expression à droite du signe égal.

On remarque que l'identifiant de l'objet auquel renvoie la variable b n'a plus rien à voir avec a. L'objet nommé b n'a plus de relation avec l'objet nommé a, et une modification ultérieure de a ne modifiera pas la valeur de b.

Exercice :

Lisez les suites d'instructions suivantes, et notez sur un papier l'état des différentes variables au fur et à mesure de l'exécution. Faites alors une prédiction sur l'état des variables à la fin de cette suite d'instructions. Vérifiez votre résultat en entrant ces instructions dans l'interpréteur.

a = 2 ; b = 17 ; c = 19

1./

```
>>> a = 100
>>> b = 17
>>> c = a - b
>>> a = 2
>>> c = b + a
>>> a, b, c
```

a = 4, b = 3, c = 3

2./

```
>>> a = 3
>>> b = 4
>>> c = a
>>> a = b
>>> b = c
>>> a, b, c
```

Il est fréquent qu'en programmation, on se serve d'une variable comme d'un compteur, que l'on a besoin d'incrémenter (augmenter) ou bien décrémenter (diminuer) d'une certaine quantité. On procède alors ainsi

```
>>> x = 0
>>> x, id(x), type(x)              #res. : (0, 137180712, int)
>>> x = x + 1
>>> x, id(x), type(x)              #res. : (1, 137180736, int)
>>> x = x + 1
>>> x, id(x), type(x)             #res. : (2, 137180752, int)
>>> x = x + 1
>>> x, id(x), type(x)             #res. : (3, 137180768, int)
```

Notons bien à nouveau la différence avec le calcul en mathématiques : l'équation $x = x + 1$ n'a aucune solution mathématique, mais l'instruction informatique $x = x + 1$ est parfaitement valide, et utilisée très couramment. La première instruction $x = x + 1$ procède en deux temps :

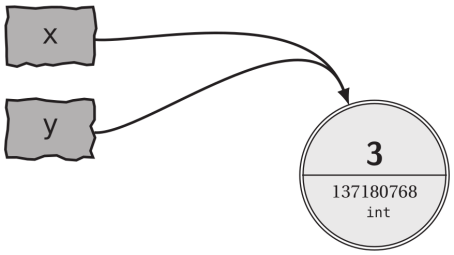
— l'interpréteur évalue la valeur de $x+1$ à l'instant donné. Le résultat est un objet de type int de valeur 1, placé en mémoire avec l'identifiant 137180736.
— ensuite seulement, l'interpréteur affecte au nom x l'objet obtenu. L'identifiant mémoire de x est donc 137180736.

Voici quelques raccourcis pour écrire ce type d'instruction de manière plus compacte :

```
>>> x += 1            # remplace x par x + 1
>>> x -= 2            # remplace x par x - 2
>>> x *= 10          # remplace x par x * 10
>>> x /= 7            # remplace x par x / 7
```

On peut aussi assigner plusieurs noms de variables au même objet. On procède ainsi :

```
>>> x = y = 3
>>> x, y
(3, 3)
>>> id(x), id(y)
(137180768, 137180768)
```



On peut aussi effectuer des affectation parallèles de plusieurs variables avec un unique symbole égal (c'est une affectation d'un couple. En Python, on dira plutôt un 2-uple, le type général étant dénommé « t-uple »)

```
>>> x, y = 128, 256
>>> (x, y) = (128, 256)    # cette expression est en fait identique a la precedente
>>> type((x,y))
```

Exercice :

Prévoyez le résultat des suites d'instructions suivantes :

x = 21, y = 42

1./

```
>>> x = 19
>>> x = x + 2 ; y = x * 2
>>> x, y
```

x = 21, y = 38

2./

```
>>> x = 19
>>> x, y = x + 2, x * 2
>>> x, y
```

Exercice fondamental :

On suppose que les variables `x` et `y` ont pour valeurs respectives deux entiers.
On souhaite échanger le contenu de ces deux variables.

1. (méthode 1) Proposez une méthode qui utilise une variable auxiliaire `tmp`.

2. (méthode 2) On exécute la suite d'instructions suivante (inutile de l'entrer dans l'interpréteur) :

```
x = x + y; y = x - y; x = x - y
```

   Quels sont les contenus des variables `x` et `y` en fin de séquence ?

3. (méthode 3) Utilisez une affectation parallèle pour résoudre le problème en une seule instruction Python.

Commentaire sur les noms de variables : ils ne peuvent pas commencer par un nombre, mais peuvent contenir des caractères spéciaux comme le tiret bas _ (« underscore »en anglais). C'est une bonne pratique de programmation de nommer ses variables de manière courte mais de façon à illustrer le rôle de cette variable.

4 Fonctions

Supposons que nous cherchions à calculer l'image d'un nombre par une fonction polynomiale donnée. Si la fonction est un peu longue à saisir, par exemple pour $f : x \mapsto x^7 - 6x^6 + 15x^4 - 12x^3 + 3x - 6$, il sera fastidieux de retaper toute cette fonction à chaque fois que l'on veut calculer une image par f. On peut dans un premier temps utiliser l'historique de l'intepréteur (grâce à la flèche du haut). Par exemple :

```
>>> x = 2
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
-112
>>> x = 3
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6   # grace a fleche du haut deux fois
-1293
>>> x = -10
>>> x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6   # grace a fleche du haut deux fois
-15838036
```

On peut bien entendu faire beaucoup mieux grâce à Python. On va définir une **fonction Python** qui ressemble à une fonction mathématique. La **syntaxe** (la manière précise d'écrire ce type d'instruction) est alors la suivante :

```
>>> def f(x) :
...     return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>>f(2), f(3), f(-10)
(-112, -1293, -15838036)
```

Dans cette définition vient d'abord la déclaration d'une nouvelle fonction par le mot-clé `def`. Vient ensuite le **nom** de la fonction, ici `f`, suivi du **paramètre formel** de la fonction, placé entre parenthèses (le paramètre formel est l'équivalent de la variable muette x dans la déclaration mathématique $f : x \mapsto x^7 - 6x^6 + 15x^4 - 12x^3 + 3x - 6$). On conclut cette première ligne par deux-points. Une fois cette ligne saisie, on appuie sur Entrée, et les chevrons sont alors remplacés par trois points. Cela signifie que l'interpréteur attend la suite des instructions. Il faut saisir **quatre espaces** (on dit qu'on fait une **indentation** du code) afin d'indiquer que l'on va désormais préciser le contenu de la fonction. Une fois le contenu de la fonction terminé, on fait Entrée, les trois points apparaissent à nouveau, et si l'on fait Entrée sans rien taper sur cette ligne cela signale la fin de la définition de la fonction. Dans notre exemple, la fonction ne contenait qu'une ligne, la suivante contient donc seulement trois points et la définition de la fonction est alors terminée. Voici quelques erreurs à éviter :

```
>>> def f(x)              # erreur 1 : oubli des deux points
  File "<stdin>", line 1
    def f(x)
           ^
SyntaxError : invalid syntax

>>> def f(x) :       # erreur 2 : non respect de l'indentation
... return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
       ^
IndentationError : expected an indented block

>>> def f(x) :       # erreur 3 : oubli du mot-cle return
...       x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>> f(2), f(3), f(-10)
(None, None, None)
```

Dans la troisième erreur, il n'y a pas de message d'erreur, mais notre fonction ne donne aucun résultat : l'expression $x**7-6*x**6+15*x**4-12*x**3+3*x-6$ est calculée en remplaçant x successivement par $2, 3$ et -10, mais l'interpréteur n'a pas reçu l'instruction de renvoyer le résultat. Il reste muet et se contente de renvoyer comme valeur l'objet `None`. On pourrait être tenté de remplacer `return` par l'instruction `print` que nous avons recontrée plus haut :

```
>>> def f(x) :
...     print(x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6)
...
>>> f(2), f(3), f(-10)
-112
-1293
-15838036
(None, None, None)
```

Pour mieux comprendre ce qui vient de se passer voici un autre exemple. Supposons que nous cherchions à calculer la somme des images de 2 3 et -10 par f. Comparons les résultats avec `print` et avec `return`.

```
>>> def f(x) :
...     return x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6
...
>>> f(2)+ f(3)+f(-10)
-15839441
>>> def f(x) :
...      print(x**7 - 6*x**6 + 15*x**4 - 12*x**3 + 3*x - 6)
...
>>> f(2)+ f(3)+f(-10)
-112
-1293
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Dans le deuxième cas, l'interpréteur affiche à l'écran la valeur de l'expression $x**7-6*x**6+15*x**4-12*x**3+3*x-6$ lorsqu'il rencontre la fonction `print`, mais ensuite il ne renvoie aucun objet réutilisable ultérieurement pour faire un calcul. Il vaut donc mieux éviter de prendre l'habitude d'utiliser `print` à l'intérieur des fonctions, suaf si c'est explicitement requis. On utilisera surtout la fonction `print` en dehors des fonctions, dans des suites d'expressions écrites dans l'éditeur de texte (appelées **scripts**). Une propriété remarquable de `return`, c'est qu'elle interrompt instantanément l'exécution de la fonction. Inutile donc de placer des instruction après un `return`. Ces instructions ne seront jamais lues, c'est du code mort.

```
>>> def mult_7(x) :
...     return 7 * x
...     print("Ceci ne s'affichera jamais")     # du code mort
...     return 0                    # toujours mort
```

Autre fait crucial : les variables définies à l'intérieur d'une fonction ne sont pas visibles depuis l'extérieur de la fonction : ce sont des variables muettes ! Aussitôt l'exécution de la fonction terminée, elles sont effacées par l'interpréteur. On dit qu'une telle variable est **locale** à la fonction.

```
def f(y) :
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Si une variable `x` existait déjà avant l'exécution de la fonction, tout se passe comme si, durant l'exécution de `f`, cette variable était masquée momentanément, puis restituée à la fin de l'exécution de la fonction.

```
>>> x = 0
>>> def f(y) :
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
0
```

Finalement, signalons qu'une fonction peut comporter autant de paramètres formels que souhaité (et éventuellement aucun). Par exemple

```
>>> def une_fonction_a_deux_variables(x, y) :
...     return x * 2**y
...
```