TP 1
Prise en main

I Introduction au mode interactif

Dans le mode interactif, on écrit les commandes **une ligne à la fois** à la suite de **l'invite de commande** `>>>`. À chaque ligne, Python affiche le résultat du calcul, et enregistre au fur et à mesure les variables.

Il est donc possible de recopier et tester ligne par ligne les morceaux de programmes présentés. Pour progresser, **ne pas hésiter à prendre des initiatives**, **tester avec d'autres valeurs** que celles proposées ici et **lire les messages d'erreur** !

Les espaces sont optionnels, mais rendent le code plus lisible. Par contre la différence entre minuscules et majuscules est importante.

Un raccourci clavier utile : les flèches haut ↑ et bas ↓ permettent de faire réapparaître les entrées précédentes, pour éviter d'avoir à tout recopier à chaque fois, surtout en cas d'erreur.

I.1 Utilisation comme calculatrice

Dans un premier temps, on peut utiliser Python comme une simple calculatrice avec les nombres et les opérations `+`, `-`, `*`, `/`. Le programme répond par le résultat du calcul.

```
>>> 3 + 4
7
>>> 8 - 10
-2
>>> 10 / 3
3.3333333333333335
```

Ces opérations vérifient les règles de priorité habituelles. Des parenthèses peuvent être nécessaires pour forcer la priorité.

```
>>> 2 + 3 * 5
17
>>> (2 + 3) * 5
25
```

L'opération puissance s'écrit `**` : ainsi 10^3 est

```
>>> 10 ** 3
1000
```

Exercice 1
Calculer les nombres suivants :

$$2^{16} \qquad 3^{\frac{1}{2}} \qquad 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5} \qquad 1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}} \qquad \left(1+\frac{1}{100}\right)^{100}$$

I.2 Les variables

En informatique, une variable est une case de la mémoire capable d'enregistrer une valeur ou le résultat d'un calcul. Pour la manipuler il faut lui donner un nom (qui peut être composé de plusieurs lettres). L'opération d'**affectation**, écrite avec le symbole `=`, permet de donner un contenu à la variable.

```
>>> x = 3
>>> x
3
>>> x + 4
7
>>> y = 8
>>> x - y
-5
```

En mathématiques on note parfois ceci $x \leftarrow 3$ pour bien signifier qu'il s'agit de **l'opération** de mettre dans la variable x la valeur 3.

À partir de ceci on peut vouloir mettre à jour la valeur de `x` :

```
>>> x = 3
>>> x
3
>>> x = x + 1
>>> x
4
```

Là encore, la ligne `x = x + 1` ne se comprend pas vraiment comme une égalité mathématique mais plutôt comme l'opération $x \leftarrow x + 1$: remplacer la valeur de x par sa valeur augmentée de 1.

On remarque que rien ne s'affiche immédiatement après l'opération d'affectation. C'est normal, il s'agit d'une instruction, qui dit à l'ordinateur de réaliser une certaine opération en mémoire mais ne produit pas de résultat visible.

Les types de la section suivante peuvent tous être contenus dans une variable.

En mode interactif, le programme enregistre les variables au fur et à mesure et s'en souvient, jusqu'à ce qu'on lui demande de redémarrer.

II Les types

Les expressions et les variables Python ont toutes un **type**, qui indique quel type d'objet on manipule et comment l'ordinateur se les représente. Le type d'une expression peut-être obtenu avec la fonction `type()` :

```
>>> x = 3
>>> type(x)
<class 'int'>
>>> type(3.5)
<class 'float'>
```

Il est important de connaitre le fonctionnement d'un certain nombre de types de base. Les principaux types que nous manipulerons sont les suivants :

II.1 Le type entier `int`

C'est le type des nombres entiers comme `5` ou bien `-2`, en anglais *integer*, en Python `int`.

Un fait remarquable est que Python est capable de gérer des nombres entiers très grands !

```
>>> 2 ** 10
1024
>>> 2 ** 100
1267650600228229401496703205376
>>> 2 ** 1000 # à vous d'essayer !
```

Ce n'est pas si évident : nous verrons qu'un nombre est codé dans l'ordinateur par une suite de 0 et de 1 appelés *bits*, avec un nombre fixe de chiffres (en général 32 ou bien 64) ce qui donne un nombre maximal qu'on peut

représenter. Ici, la place en mémoire pour stocker les entiers Python est agrandie automatiquement selon les besoins.

II.2 Le type flottant `float`

En informatique, on ne peut pas représenter tous les nombres réels avec une infinité de décimales après la virgule, car l'espace nécessaire pour stocker un tel nombre pourrait être infini…

Les nombres à virgule que l'on manipule sont appelés des **nombres à virgules flottantes** (ou plus simplement **flottants**) et forment le type `float`. Le symbole pour la virgule est le point `.` et on peut aussi utiliser la notation scientifique où la lettre `e` désigne la puissance de 10 :

```
>>> 1.23e4
12300.0
>>> 12.34e50
1.234e+51
>>> 1 - 1e-5
0.99999
```

Le nom de *virgule flottante* provient du fait que ces nombres sont représentés par l'ordinateur sous une certaine forme de notation scientifique, dans un emplacement mémoire de taille limitée avec une partie pour stocker l'exposant et une partie pour stocker les décimales. Ainsi la précision d'un nombre est toujours limitée à une quinzaine de chiffres, mais la partie avec exposant peut varier d'environ `e-308` à `e+308`. C'est la plupart du temps bien suffisant pour les sciences !

Mais à cause de cette précision limitée, il y a parfois des erreurs d'arrondis dans les nombres flottants…

Exercice 2

1. Tester :

   ```
   >>> 1e30 + 0.1
   >>> 1e30 + 0.1 == 1e30
   ```

2. Tester :

   ```
   >>> 0.3
   >>> 0.1 + 0.2
   >>> 0.1 + 0.2 == 0.3
   ```

3. Cela est bien visible aussi avec des fonctions spéciales, tester :

   ```
   >>> from math import *
   >>> sin(2*pi)
   >>> sin(2*pi) == 0
   ```

   et

   ```
   >>> from math import *
   >>> exp(log(3))
   >>> exp(log(3)) == 3
   ```

L'opération de division `/` donne toujours un nombre flottant, même entre nombres entiers :

```
>>> 10 / 5
2.0
>>> type(10 / 5)
<class 'float'>
```

Pour travailler uniquement avec des nombres entiers on utilise la division euclidienne `//`, celle qui donne un quotient et un reste :

```
>>> 10 // 5
2
>>> 11 // 5
2
>>> 19 // 5
3
```

Le reste lui est obtenu par l'opération `%`. Cela parlera mieux aux élèves qui ont étudié l'arithmétique…

```
>>> 10 % 5
0
>>> 11 % 5
1
>>> 19 % 5
4
```

C'est la bonne méthode pour tester la parité : un entier n est pair quand `n % 2` vaut 0, et impair si `n % 2` vaut 1.

II.3 Le type des chaines de caractères `str`

Ce que l'on appelle communément du *texte* forme aussi un type, que l'on peut enregistrer dans des variables et que l'on peut vouloir manipuler. En informatique, un tel texte s'appelle une **chaine de caractères**, en anglais *string*. En effet l'ordinateur les considère comme une liste de caractères les uns à la suite des autres, indépendamment de savoir si le texte a du sens ou non. Elles forment le type `str` et sont écrites encadrées préférablement par le guillemet double `"texte"`, même si le guillemet simple (apostrophe `'`) est possible aussi.

```
>>> x = "Bonjour"
>>> type(x)
<class 'str'>
```

L'opération `+` sur les chaines de caractères crée une nouvelle chaine en plaçant les deux bout à bout et s'appelle en informatique la **concaténation**.

```
>>> "Bonjour" + "et bienvenue"
'Bonjouret bienvenue'
```

Corrigeons ceci… premier essai :

```
>>> "Bonjour" + " et bienvenue"
'Bonjour et bienvenue'
```

Deuxième essai :

```
>>> x = "Bonjour"
>>> y = "et bienvenue"
>>> x + " " + y
'Bonjour et bienvenue'
```

Il existe aussi une opération de multiplication entre une chaine et un entier. À votre avis, que fait-elle ?

Exercice 3
Tester :

```
>>> "Ha" * 5
```

La syntaxe crochets `[i]` permet d'accéder au i-ième caractère de la chaine :

```
>>> x = "Bonjour"
>>> x[1]
'o'
>>> x[2]
'n'
```

… en fait, elle est numérotée à partir de 0 : c'est `x[0]` qui donne `'B'`.

Avec les indices négatifs, on repart de la fin !

```
>>> x = "Bonjour"
>>> x[-1]
'r'
>>> x[-2]
'u'
```

Ces conventions sont peut-être étonnantes au début, mais cela reviendra de nombreuses fois…

Exercice 4
Tester et expliquer la différence entre les deux expressions suivantes :

```
>>> 12 + 34
>>> "12" + "34"
```

À retenir

Les variables Python ont toutes un **type**. Chaque type a ses propriétés. Les opérations ne se comportent pas toutes de la même façon en fonction du type des variables, même si en apparence leur contenu est le même.

II.4 Le type booléen

C'est un type à part entière qui représente une valeur vrai ou faux, en Python `True`, `False`. Le nom provient du mathématicien anglais George Boole. On appelle donc ces valeurs des **booléens**, formant le type `bool`.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

On peut utiliser dessus les opérations

- et : `and`
- ou : `or`
- non : `not`

… et vérifier toutes les propriétés que nous avons vues en cours sur les assertions !

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
```

et ainsi de suite. Si on en combine plusieurs, on peut utiliser des parenthèses.

Exercice 5
Donner le résultat de l'expression suivante :

```
>>> not True and False
```

Cela permet d'en déduire laquelle de ces deux opérations est prioritaire : le `not` ou bien le `and` ?

Les opérations de comparaisons entre nombres donnent une valeur booléenne. Ce sont les suivantes :
- L'égalité : `==`
- La différence : `!=`
- Les comparaisons strictes : `<`, `>`
- Les comparaisons larges $\leqslant$, $\geqslant$: `<=`, `>=`

Quelques exemples :

```
>>> 5 > 8
False
>>> 5 + 3 <= 8
True
>>> 1 == 2
False
>>> 1 != 2
True
```

Insistons encore une fois sur le fait que ce sont des types à part entière, pouvant rentrer dans des variables :

```
>>> résultat = 3 < 4
>>> not résultat
False
```

Les opérations logiques ne sont pas tout à fait *commutatives* comme en mathématiques…

Exercice 6
1. Tester et expliquer la différence entre les deux expressions :

   ```
   >>> 1 + 2 == 5 and 1 / 0 == 2
   >>> 1 / 0 == 2 and 1 + 2 == 5
   ```

2. Pouvez-vous mettre en place un test similaire pour illustrer le comportement du `or` ?

II.5 Les conversions de type

Comme on l'a vu, le nombre `3` n'est pas la même chose que `3.0` car le premier est de type `int` et le second est de type `float`. Et la chaine de caractères `"12"` n'est pas la même chose que le nombre entier `12`.

Aux types correspondent aussi des fonctions de conversion de type `int()`, `float()`, `str()`, `bool()`, qui convertissent **vers** le type :

```
>>> float(5)
5.0
>>> int(3.5)
3
>>> int("12") + int("34")
46
>>> str(12) + str(34)
'1234'
```

Via `bool()`, n'importe quel nombre est converti à `True`, sauf `0`. N'importe quelle chaine est convertie en `True`… sauf la chaine vide `""`. Il y a des bonnes raisons historiques pour cela : puisqu'un entier est codé sur au minimum 8 bits, il a été convenu que le `0` représentait `False` et toute autre valeur représentait `True`.

III Le mode script

En mode script, on peut écrire plusieurs lignes à la suite et l'envoyer d'un coup à l'interpréteur Python. Les instructions sont obligatoirement séparées par un retour à la ligne. Le programme continue à enregistrer les variables au fur et à mesure, mais **il n'affiche rien si on ne le lui demande pas** ! Il faut alors utiliser la commande `print()` pour afficher quelque chose.

```
x = 3
y = 8
x = x - y
print(x)
```

On peut aussi se servir de `print` pour afficher plusieurs variables, ou nombres ou chaines, à la suite sur une même ligne.

```
x = 5
y = 12
print("x =", x, "y =", y)
```

Toute ligne commençant par le symbole `#` est ignorée : c'est un **commentaire**, qui sert à expliquer ce que fait le programme. Les commentaires sont loin d'être inutiles car ils permettent à d'autres personnes qui lisent le programme de mieux le comprendre.

Dans les logiciels Pyzo ou Spyder, une ligne commençant par `#%%` (dièse, pourcent, pourcent) permet de séparer le code en **cellules** pour garder tout le code sur une même page, mais ne demander à n'exécuter qu'une seule cellule à la fois (sinon, on rajoute du code au fur et à mesure, et à chaque fois, tout est exécuté depuis le début). Une bonne idée est de l'utiliser pour séparer chaque question des exercices. L'option s'appelle « exécuter la cellule » (*execute cell*), qu'on trouve aussi sur le raccourci clavier `Ctrl` + `Entrée`. Toute l'année, les fichiers seront présentés divisés en cellules. Le fichier ressemble à ceci :

```
#%% exercice 1
...

#%% exercice 2
...
```

Enfin la dernière notion utile pour entamer le mode script est la fonction `input()` qui permet de demander une information à l'utilisateur.

```
x = input("Entrez quelque chose : ")
print("Vous avez entré", x)
```

La valeur donnée par `input()` est toujours de type `str` ! Les conversions de type apparaissent donc bien utiles ici. Si on veut demander un nombre (entier) à l'utilisateur, on écrit en général directement `int(input())` :

```
x = int(input("Entrez un nombre entier : "))
print("Son double est :", 2 * x)
```

Si on veut un nombre qui peut avoir une virgule, alors c'est la conversion `float()` qu'il faut utiliser, sinon une erreur se produit : on tente de convertir en nombre entier du texte avec une virgule dedans !

IV D'autres exercices

Dans les exercices suivants, on demande d'écrire des petits bouts de programme : quelques lignes de code dans le mode interactif, séparées en cellules, de façon à pouvoir les exécuter séparément. À ce stade, pour que les programmes soient un minimum intéressants, ils utilisent largement `input()` pour que l'utilisateur puisse varier un paramètre.

Exercice 7
Écrire un programme qui demande à l'utilisateur son nom, puis affiche `Bienvenue [nom] en BCPST1B !` (en remplaçant par son nom).

Pouvez-vous créer une variable contenant tout ce texte d'un coup ?

Exercice 8
Écrire un programme qui demande à l'utilisateur son année de naissance, puis affiche son âge en 2026 (on ne se préoccupera pas du mois de naissance…), sous la forme `Vous avez [ans] ans.`

Idem, pouvez-vous mettre ce texte dans une seule variable ?

Exercice 9
Écrire un programme qui demande à l'utilisateur d'entrer deux nombres `a` et `b` et en donne la moyenne.

Exercice 10
Écrire un programme qui demande à l'utilisateur d'entrer deux nombres `a` et `b`, les affiche, puis les échange, et les affiche encore. En échangeant réellement la valeur des variables, et pas seulement l'affichage…

Exercice 11
On peut utiliser `print()` directement sur une expression booléenne pour afficher simplement `True` or `False`.

Écrire des programmes qui demandent à l'utilisateur des nombres et testent, en affichant le résultat, les conditions suivantes :

1. Le nombre `n` est pair (en utilisant l'opération `%`).
2. Les nombres entiers `n` et `m` sont de même signe.
3. Les nombres entiers `n`, `m`, `p` sont deux à deux distincts.

Exercice 12
Sur un caractère tout seul `x`, la fonction `x.isupper()` donne `True` si `x` est en majuscule et `False` sinon. Essayez-là !

Écrire un programme qui demande à l'utilisateur d'entrer une phrase, et teste si la phrase commence par une majuscule et termine par un point.