Programmation en Python — Rappels

1 Utiliser Python comme une calculette

Pour débuter la programmation en Python, le plus simple pour exécuter des instructions Python est d'utiliser un environnement spécialisé comme `Idle`. Cet environnement se compose d'une fenêtre appelée indifféremment *console*, *shell* ou *terminal* Python.
L'invite de commande se compose de trois chevrons ; il suffit de saisir à la suite une instruction puis d'appuyer sur la touche « Entrée » de votre clavier. La console Python fonctionne comme une simple calculatrice : vous pouvez y saisir une expression dont la valeur est renvoyée dès que vous pressez la touche « Entrée ».

```
>>> 2 * 5 + 6 - (100 + 3)
-87
>>> 7 / 2; 7 / 3    # division décimale
3.5
2.3333333333333335
>>> 34 // 5, 34 % 5 # quotient, reste de la divis° euclidienne de 34 par 5
(6, 4)
>>> 2 ** 7  # pour l'exponentiation (et non pas 2^7 !)
128
```

Au passage, nous avons utilisé le symbole croisillon `#` pour placer des commentaires dans les lignes de commande ; tout ce qui se situe à droite d'un symbole `#` (jusqu'au changement de ligne) est purement et simplement ignoré par l'interpréteur.
Pour naviguer dans l'historique des instructions saisies dans la console Python d'`Idle`, on peut utiliser les raccourcis `Alt+p` (`p` comme *previous*) et `Alt+n` (`n` comme *next*).

2 Variables et affectations

Que se passe-t-il au juste lorsqu'on saisit un nombre (par exemple 128) dans la console Python ? Eh bien, disons, en première approximation, que l'interpréteur crée un nouvel « objet » (sans préciser pour l'instant le sens de ce mot) et le garde en mémoire.

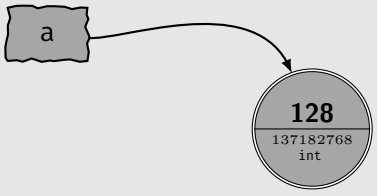
```
>>> 128, id(128), type(128)
(1, 137182768, <class 'int'>)
```

Cet objet possède une *valeur* (ici 128), un *identifiant*, c.-à-d. une « carte d'identité » permettant de savoir où il est gardé en mémoire (ici 137182768), et enfin un *type* (ici le type entier dénommé `int`).

Le moins que l'on puisse dire, c'est que l'identifiant ne nous parle guère... D'où l'intérêt des *affectations*. Au lieu de désigner 128 par son identifiant, on va lui donner un nom commode à manipuler. Au lieu d'appeler l'objet « Monsieur 137180768 », on l'appellera « Monsieur A. », après avoir indiqué à l'interpréteur une fois pour toute l'identification opérée par un message du type : « Monsieur A. *alias* Monsieur 137180768 ».

En pratique, une affectation s'effectue à l'aide du symbole « `=` », comme ceci :

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



Ainsi chaque fois que nous appellerons le nombre 128, il nous suffira d'invoquer la variable `a`. Notez que l'objet désigné par `a` est toujours l'objet 128, rangé encore à la même adresse.
Il faut bien prendre garde au fait que l'instruction d'affectation « `=` » n'a pas la même signification que le symbole d'égalité « = » en mathématiques. Ceci explique que dans les livres d'algorithme, l'affection de `expr` à `x` se note souvent `x` ← `expr`.

Par exemple, le premier n'est pas symétrique, alors que le second l'est : vouloir échanger l'ordre des éléments dans une instruction d'affectation produira immanquablement une erreur dans l'interpréteur :

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

Effectuons à présent la suite d'affectations suivantes :

```
>>> b = a * 2
>>> b                    # rép.: 256
>>> b, id(b), type(b)    # rép.: (256, 137184816, <class 'int'>)
>>> a = 0
>>> a, id(a), type(a)    # rép.: (0, 137180720, <class 'int'>)
>>> b                    # rép.: 256
```

Ici se situe une petite difficulté pour ceux qui débutent la programmation. Contrairement à ce que nos habitudes de calcul algébrique pourraient nous laisser penser, l'instruction `b = a*2` n'affecte pas à `b` le double de la valeur `a` quelle que soit la valeur de `a` au long de la session Python. Au contraire, l'instruction `b = a*2` procède en deux temps :

- l'expression située à droite du signe « = » est évaluée, c.-à-d. calculée en fonction de l'état de la mémoire *à cet instant* : ici l'interpréteur évalue le double de la valeur de `a` ; le résultat est un objet de type entier, de valeur 256, et placé en mémoire avec l'identifiant `137180720`.
- ensuite, et seulement ensuite, l'interpréteur affecte au nom situé à gauche de l'instruction d'affectation (à savoir `b`) l'objet obtenu après évaluation de l'expression de droite.

On remarque que l'identifiant de l'objet auquel renvoie la variable `b` n'a plus rien à voir avec `a`. Autrement dit, l'objet nommé `b` n'a plus aucune relation avec l'objet nommé `a`. Ainsi, une réaffectation ultérieure de la variable `a` n'entraînera aucun changement pour la variable `b`.
Avez-vous bien compris ? Alors exercez-vous en lisant les suites d'instructions suivantes et en notant sur un papier le contenu de chacune des variables à chaque étape ; puis exécutez chacune de ces suites d'instructions dans la console et vérifiez que ce que vous avez noté concorde avec ce qu'affiche l'interpréteur.

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

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

Allons un peu plus loin ; il est fréquent qu'en programmation, on se serve d'une variable comme d'un compteur et que l'on ait donc besoin d'incrémenter (c.-à-d. augmenter) ou de décrémenter (c.-à-d. diminuer) la valeur de la variable d'une certaine quantité. On procède alors de la manière suivante.

```
>>> x = 0
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (1, 137180736, <class 'int'>)
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (2, 137180752, <class 'int'>)
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (3, 137180768, <class 'int'>)
```

Encore une fois, notez bien la différence avec le calcul algébrique : alors que l'équation $x = x + 1$ n'a pas de solution, l'instruction `x = x + 1` est parfaitement licite, et même utilisée couramment en programmation.
Détaillons la première instruction `x = x + 1` ci-dessus ; cette instruction procède en deux temps :

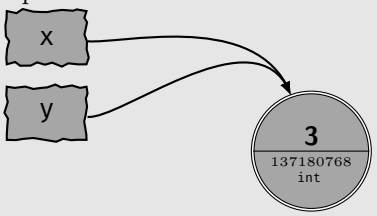
- l'interpréteur évalue la valeur de `x + 1` à l'instant donné ; le résultat est un objet de type entier, de valeur 1, et placé en mémoire avec l'identifiant `137180736`.
- ensuite, et seulement ensuite, l'interpréteur affecte au nom qui se trouve à gauche de l'instruction d'affectation (à savoir `x`) l'objet obtenu après évaluation de l'expression de droite.

Signalons un raccourci propre à Python très utile en pratique et qui a l'avantage d'éviter la confusion avec la manipulation d'équations en algèbre.

```
>>> x += 1  # remplace x par x+1
>>> x -= 3  # remplace x par x-3
>>> x *= 3  # remplace x par x*3
>>> x /= 2  # remplace x par x/2
```

Autre raccourci intéressant, on peut assigner un même objet à plusieurs variables simultanément. Ces deux variables renvoient alors au même objet (on dit parfois que ce sont deux *alias* du même objet).

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



On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur « = ». Toutes les expressions sont alors évaluées *avant* la première affectation.

```
>>> x, y = 128, 256
```

Encore une fois, il faut bien comprendre qu'une affectation se déroule en deux temps : évaluation de l'expression de droite puis création de l'alias avec le nom du terme de gauche.

Exercice n° 1.

a) Quel est le contenu des variables `a` et `b` au terme des trois instructions suivantes ? Contrôler ensuite le résultat en exécutant ces instructions dans l'interpréteur.

```
>>> a, b = 3, 7
>>> b = (a + b) ** 2
>>> a = 5 * a + 1
```

b) Même question avec les instructions suivantes.

```
>>> a, b = 3, 7
>>> a = 5 * a + 1
>>> b = (a + b) ** 2
```

Exercice n° 2. *Échange des contenus de deux variables*

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

a) *Première méthode :* Proposer une méthode qui utilise une variable auxiliaire `tmp`.

b) *Deuxième méthode :* On exécute la séquence d'instructions suivante :

```
>>> x = x + y
>>> y = x - y
>>> x = x - y
```

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

c) *Troisième méthode (propre au langage Python) :* Utiliser une affectation parallèle.

3 Fonctions

Parallèlement à la manipulation de variables, un programme informatique nécessite généralement la définition de fonctions. Une fonction au sens informatique reprend la notion mathématique de fonction.
En Python, la définition de la fonction se fait à l'aide du mot `def` suivi du nom de la fonction, des arguments de la fonction entre parenthèses et de deux-points. Elle se poursuit sur les lignes suivantes par les instructions qui doivent être effectuées et se termine par l'instruction `return` suivi de l'expression à renvoyer en fin de programme. À partir de la deuxième ligne, toutes les lignes doivent être indentées, c'est-à-dire qu'elles doivent commencer par quatre espaces.

Voici par exemple une fonction qui prend en argument un nombre et renvoie ce nombre élevé au carré.

```
>>> def carré(x):
...     return x**2
...
>>> carré(2.5) # le séparateur décimal est le point et non la virgule
6.25
```

Dans l'exercice suivant, nous aurons du nombre π. Le nombre π est fourni par le module `math` de Python, qu'il faut importer au préalable :

```
>>> import math
>>> math.pi  # le nom du module math doit précéder le nom du nombre π.
3.141592653589793
```

Plus généralement, le module `math` de Python fournit les fonctions mathématiques usuelles, notamment :

commande Python	analogue mathématique
`math.pi`	valeur approchée de π
`math.sin(x)`	$\sin x$
`math.cos(x)`	$\cos x$
`math.tan(x)`	$\tan x$

commande Python	analogue mathématique
`math.abs(x)`	$\lvert x\rvert$
`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	$\exp(x)$
`math.log(x)`	$\ln x$

Exercice n° 3. Écrire une fonction qui calcule l'aire d'un disque en fonction de son rayon.

Il est fréquent que l'on ait besoin de définir une fonction prenant plusieurs arguments. On énumère alors les différents arguments de la fonction en les séparant par une virgule. Voici par exemple une fonction qui calcule l'aire d'un triangle, connaissant une base et la hauteur associée :

```
>>> def aire_triangle(base, hauteur):
...     return base * hauteur / 2
...
>>> aire_triangle(5, 6)
15.0
```

Exercice n° 4. Écrire une fonction qui calcule l'aire d'un trapèze à partir de ses bases et de sa hauteur.

Exercice n° 5. Écrire une fonction `hms2s` qui prend en argument une durée exprimée en heures, minutes et secondes et qui renvoie cette durée convertie en secondes.

Parfois, une fonction doit renvoyer plusieurs arguments. On les énumère alors en les séparant par une virgule. Voici une fonction qui renvoie le carré, le cube et la puissance quatrième d'un nombre :

```
>>> def f(x):
...     return x ** 2, x ** 3, x ** 4
...
>>> f(10)
(100, 1000, 10000)
```

Le résultat renvoyé est ce que l'on appelle un triplet, qui est un cas particulier de n-uplet (ou `tuple` dans la terminologie du langage Python).

Pour récupérer un élément d'un n-uplet, on utilise l'indice de l'élément à extraire placé entre crochets, en prenant garde au fait que la numérotation des éléments commence à 0 et non à 1 :

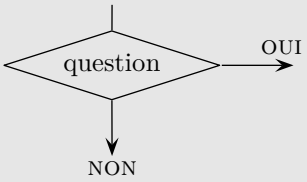
```
>>> f(10)[0]   # on extrait le premier élément, qui est d'indice 0
100
>>> f(10)[2]   # on extrait le troisième élément, qui est d'indice 2
10000
```

Exercice n° 6. Écrire une fonction `s2hms` qui prend en argument une durée exprimée en secondes et qui renvoie cette durée convertie en heures, minutes et secondes. En utilisant la fonction `s2hms` et la fonction `hms2s` de l'exercice 5, expliquer comment ajouter deux durées exprimées en heures, minutes et secondes.

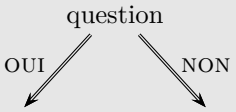
4 Conditionnelles

On a souvent besoin de demander à l'ordinateur d'effectuer une action ou bien une autre suivant le résultat d'un calcul. Pour cela, on doit utiliser ce que l'on appelle une *conditionnelle*.
Une telle conditionnelle représente une alternative, ce qui signifie qu'il n'y a que deux possibilités en réponse à la question posée : VRAI ou FAUX (OUI ou NON).

On convient, dans certains schémas décrivant des algorithmes, de représenter une conditionnelle par un losange (avec la question posée) avec une entrée (représentée ci-contre par le trait du haut) et deux branches de sortie, correspondant aux réponses OUI et NON.



Si on préfère que les deux branches de sortie occupent des positions graphiquement plus symétriques, on pourra utiliser une représentation comme celle de droite.



Écrivons une fonction qui renvoie la valeur absolue d'un nombre : $|x| = \begin{cases} x \text{ si } x \geqslant 0 \\ -x \text{ si } x < 0 \end{cases}$
La syntaxe en Python sera la suivante :

```
>>> def vabs(x):
...     if x >= 0:
...         return x
...     else:
...         return -x
...
>>> vabs(2), vabs(-2)
(2, 2)
```

Observons la syntaxe de la structure de choix `if`. Tout d'abord, le mot `if` (qui signifie « si ») est suivi d'une condition de choix : quel est l'objet renvoyé par l'interpréteur lorsqu'il évalue cette condition ?

```
>>> 2 >= 0
True
```

```
>>> -2 >= 0
False
```

Lorsqu'une condition est évaluée par l'interpréteur, le résultat est un valeur dite booléenne (du nom du mathématicien anglais G. Boole) : `True` (qui signifie « vrai ») ou `False` (qui signifie « faux »).
La condition de choix est vérifiée quand l'évaluation de cette condition renvoie le booléen `True` et l'interpréteur exécute alors la suite d'instructions qui se trouve dans le premier bloc d'instructions. Dans le cas contraire, l'interpréteur saute au bloc d'instructions situé après le mot `else` (qui signifie « sinon ») délimité par l'indentation.

Il faut bien noter le rôle essentiel de l'*indentation* qui permet de délimiter chaque bloc d'instructions et la présence des deux-points après la condition du choix et après le mot clé `else`.

Pour obtenir une variable booléenne, on utilise en général, comme dans l'exemple précédent, un ou plusieurs *opérateurs de comparaison*. Voici les six opérateurs les plus courants :

`x == y`	`x` est égal à `y`	`x != y`	`x` est différent de `y`
`x >= y`	`x` est supérieur ou égal à `y`	`x > y`	`x` est strictement supérieur à `y`
`x <= y`	`x` est inférieur ou égal à `y`	`x < y`	`x` est strictement inférieur à `y`

⇨ *Attention* : En Python, il faut bien distinguer l'instruction d'affectation « `=` » du symbole de comparaison « `==` ».

Pour exprimer des conditions complexes (par exemple $x > -2$ et $x^2 < 5$), on peut combiner des variables booléennes en utilisant les trois *opérateurs booléens* (par ordre de priorité croissante) : `or` (qui signifie « ou »), `and` (qui signifie « et »), et `not` (qui marque la négation).

Exercice n° 7. Dessiner un arbre décisionnel qui, à partir d'un parallélogramme, indique la qualification la plus précise que l'on puisse donner à ce parallélogramme (parallélogramme sans plus de particularités, rectangle, losange ou carré). Il y a plusieurs plusieurs arbres décisionnels qui répondent à cette question.

Exercice n° 8. *Années bissextiles*
On dit souvent que les années bissextiles sont les années multiples de 4. Il existe néanmoins une exception : les années multiples de 100, appelées aussi années *séculaires*, qui ne sont pas bissextiles (par exemple, 1900 n'a pas été une année bissextile). Cette exception possède elle-même une exception : les années multiples de 400, qui sont bissextiles (et donc l'année 2000 a été bissextile).

a) On souhaite résumer les données de l'énoncé à l'aide d'un arbre décisionnel. Quels arbres peut-on proposer ? Modifier l'ordre des tests et donne trois solutions pour cette question.

b) Pour chacun des arbres trouvés à la question précédente, écrire une fonction qui, à partir d'un entier n, indique si oui ou non l'année correspondante est bissextile.

```
>>> bissextile(1900),bissextile(2000),bissextile(2016),bissextile(2017)
False True True False
```

En informatique, on peut simuler des lancers de dés, des jeux de « pile ou face », et un certain nombre d'expériences aléatoires, grâce à des générateurs de nombres (pseudo-)aléatoires. En Python, c'est le module `random` qui fournit diverses fonctions permettant de générer des nombres aléatoires. Par exemple :

`random.randint(a, b)`	choisit un entier aléatoirement dans $[a;b]$
`random.random()`	choisit un décimal aléatoirement dans $[0;1[$
`random.uniform(a, b)`	choisit un décimal aléatoirement dans $[a;b[$

```
>>> import random
>>> random.randint(1, 10)  # entier dans [1, 10]
7
>>> random.random()        # nombre décimal aléatoire dans [0, 1[
0.37444887175646646
>>> random.uniform(1, 10)  # nombre décimal aléatoire dans [1, 10[
1.1800146073117523
```

Exercice n° 9. Écrire une fonction `double_six` qui simule le lancer de deux dés à six faces et qui renvoie la valeur `True` si la somme des points obtenus vaut 12 et `False` sinon.

5 Boucles indéfinies

On a souvent besoin de faire répéter par l'ordinateur un certain nombre d'instructions similaires. Bien entendu, ces instructions ne doivent pas être effectuées indéfiniment (il faut bien que, au bout d'un certain temps, l'ordinateur arrête de travailler et nous donne la sortie attendue). Il doit donc y avoir une condition qui indique si les instructions en question doivent (encore) être exécutées ou bien si on peut passer à la suite du programme.

Imaginons par exemple que l'on souhaite demander à l'ordinateur d'affecter la valeur `1` à la variable `x` et d'augmenter cette variable de `1` tant que son carré ne dépasse pas le nombre 1000. La valeur finale de `x` contiendra donc le plus petit nombre entier dont le carré dépasse 1000.

En Python, une telle répétition est introduite par le mot `while` suivi de la condition ponctuée d'un deux-points, puis vient le bloc d'instructions à répéter, ce dernier étant délimité par l'indentation.

```
>>> x = 1
>>> while x ** 2 <= 1000:
...     x = x + 1
...
>>> x
32
```

Une telle structure informatique s'appelle une *boucle* (simplement du fait de la boucle qui apparaît graphiquement sur le schéma ci-dessus). Comme les instructions du corps de la boucle sont répétées *tant qu'*une

certaine condition est remplie, une telle boucle est appelée une boucle « tant que » ou boucle « indéfinie » (le nombre d'itérations, c'est-à-dire le nombre d'exécutions du corps de la boucle n'est pas défini avant le début des itérations).

Exercice n° 10. *Série harmonique*
Quand on additionne les inverses des n premiers entiers naturels non nuls, on obtient une quantité

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

qui augmente à mesure que n augmente.
Comment trouver le premier entier n tel que cette somme dépasse 12 ?

Exercice n° 11.

a) Écrire une fonction `somme` qui renvoie la somme des carrés des n premiers entiers : $1^2+2^2+\cdots+n^2$.
b) Modifier la fonction précédente et écris une fonction `depasse` qui, pour tout entier M, renvoie le plus petit entier n tel que $1^2+2^2+\cdots+n^2 \geqslant M$.

Exercice n° 12. *Nombres premiers*
Écrire une fonction `est_premier` qui prend un entier `n` en argument et qui renvoie `True` si `n` est premier et `False` sinon.

6 Boucles définies

À côté des boucles indéfinies, les langages de programmation proposent souvent une structure que l'on appelle *boucle définie*. On peut utiliser une telle structure lorsque l'on sait, avant le début des itérations, le nombre d'itérations qui seront effectuées (quand on dit « on sait », il faut comprendre que « l'ordinateur sait », c'est-à-dire que ce nombre d'itérations peut être lu dans une variable de la mémoire de la machine). En Python, une boucle définie est introduite par le mot `for` :

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

```
>>> for i in range(1, 4):
...     print(i)
...
1
2
3
```

Nous voyons apparaître ici pour la première fois la fonction `range`. Cette fonction crée un distributeur d'entiers consécutifs. Au lieu de créer et garder en mémoire une liste d'entiers, cette fonction génère les entiers au fur et à mesure des besoins.

- L'instruction `range(4)` permet d'obtenir les entiers de `0` inclus à `4` *exclu*, autrement dit les entiers compris entre `0` et `3`.
- L'instruction `range(1, 4)` permet d'obtenir les entiers de `1` inclus à `4` *exclu*, autrement dit les entiers compris entre `1` et `3`.

Comme dans le cas des boucles indéfinies, le bloc d'instructions est délimité par l'indentation.

Exercice n° 13.

a) Écrire une fonction `somme` qui renvoie la somme des `n` premiers entiers non nuls : $1+2+3+\cdots+n$.
b) Écrire une fonction `factorielle` qui renvoie le produit des `n` premiers entiers non nuls : $1\times2\times3\times\cdots\times n$.

Dans chacun des cas, on utilisera un accumulateur qui, après initialisation, sera modifié à chaque itération d'une boucle définie.

Exercice n° 14. En se servant de la fonction `est_premier` écrite dans un exercice précédent, écrire une fonction `compte_prems` qui prend en argument un entier n et qui compte combien il y a de nombres premiers dans $[\![1;n]\!]$.

Exercice n° 15.

a) Écrire une fonction `aléa_1` qui simule `n` fois le lancer d'une pièce de monnaie et qui compte la fréquence d'apparition de « pile ».
b) Écrire une fonction `aléa_2` qui simule `n` fois le lancer de deux dés à six faces et qui compte la fréquence d'apparition de la somme 12.