TP 1 INFO — Les bases de Python

Partie I Utiliser Python

Le langage de programmation Python permet d'interagir avec la machine à l'aide d'un interpréteur Python. Ce dernier peut être utilisé selon deux modes : le **mode interactif** et le **mode script**.

I.A - Le mode interactif

Le mode interactif de l'interpréteur Python s'approche du fonctionnement d'une calculatrice. Ce dernier se présente comme une **invite de commandes**, c'est-à-dire une fenêtre dans laquelle le système est en attente de vos commandes après le symbole `>>>`. Vous pouvez écrire une instruction, puis la valider avec la touche entrée du clavier. L'instruction est alors exécutée : le système affiche une réponse, puis l'interpréteur est prêt à recevoir de nouvelles instructions.

Question 1 : Exécuter les expressions suivantes dans le mode interactif de Python.

(i) `18 + 13`	(ii) `18 - 13`	(iii) `18 * 13`	(iv) `2**5`
(v) `73 / 7`	(vi) `73 // 7`	(vii) `73 % 7`	$(viii)$ `2**(1/2)`

I.B - Le mode script

Le mode interactif est idéal pour l'exécution de quelques commandes mais ne convient pas à la rédaction des codes plus complexes. Dans ce cas, on utilise le mode script qui consiste à écrire une suite d'instructions dans un fichier texte, puis à faire exécuter ses instructions à l'interpréteur Python. Cette suite d'instruction est appelée un **programme**. L'usage est de donner l'extension .py aux fichiers contenant des programmes Python.

ATTENTION : En mode script, les résultats ne s'affichent plus automatiquement : il faut utiliser une instruction explicite d'affichage : la fonction `print` en Python.

La fonction `print` permet d'afficher n'importe quel objet Python dans la console.

```
print("Hello world")
```

Il est possible de passer plusieurs objets à la fonction `print` en les séparant par une virgule.

```
print("Hello world", 5, [1, 2], 3.0, "test")
```

Question 2 : Exécuter les deux instructions précédentes dans l'interpréteur Python en mode interactif.

I.C - Utilisation d'un environnement de développement intégré

En pratique, nous utiliserons un **environnement de développement intégré (IDE)** tel que **Idle**, **Spyder** ou **Pyzo**. Ces outils comportent notamment :

- une partie avec un éditeur de texte destiné à la rédaction des programmes ;
- une partie avec un interpréteur permettant d'exécuter directement le programme rédigé dans l'éditeur.

Remarque 1 : Dans la suite, nous travaillerons toujours dans un IDE.

Question 3 : Recopier successivement les deux programmes ci-dessous dans votre éditeur de texte, puis exécuter les et comparer les résultats affichés par l'interpréteur.

```
# Programme 1
print(3 * (4 + 5) + 2**3)
```

```
# Programme 2
3 * (4 + 5) + 2**3
```

I.D - Les commentaires

Les commentaires sont des notes que les programmeurs ajoutent à leur code pour expliquer ce qu'il est censé faire. Ils permettent également de séparer son programme en différentes parties avec des titres. L' interpréteur ignore les commentaires à l'exécution du programme.

En Python, une ligne est un commentaire lorsqu'elle commence par le symbole #. Lorsque l'interpréteur Python rencontre # dans votre code, il ignore tout ce qui suit ce symbole et ne produit aucune erreur.

```
# Voici un commentaire. Cette ligne est ignorée par l'interpréteur.
print(3 + 5) # Affichage du résultat.
```

Afin de rendre vos codes compréhensibles et pour vous souvenir de ce qu'ils font, il est vivement conseillé de les commenter en ajoutant des explications, des précisions ou des titres.

I.E - Instruction et expression

En programmation, on distingue les notions d'expression et d'instruction. Une expression est une suite de symboles qui est réduite à une valeur après exécution (et affichée si on travaille dans le mode interactif).

Exemple 1 : Voici quelques exemples d'expressions.

```
>>> print("Bonjour")
Bonjour
>>> 3*2
6
```

Une instruction exprime un ordre ; son exécution va modifier l'état de la machine (son espace mémoire). Dans la suite de ce TP, nous réviserons plusieurs instructions de base en Python :

- l'instruction d'affectation ;
- l'instruction pour définir une fonction ;
- l'instruction conditionnelle ;
- les instructions itératives.

Partie II Les variables

Une **variable** permet de stocker en mémoire une donnée pour la réutiliser à plusieurs reprises en la désignant par un nom. L'instruction `=` sert à définir ou à modifier la donnée contenue dans une variable. Cette opération s'appelle une **affectation** : elle affecte une valeur à une variable. Une affectation crée une liaison entre un nom et une donnée stockée en mémoire.

Exemple 2 : Les instructions ci-dessous définissent deux variables `a` et `message` en leur affectant respectivement les valeurs `3` et `"bonjour"`.

```
>>> a = 3
>>> message = "bonjour"
```

On peut également modifier la valeur d'une variable déjà utilisée avec le symbole d'affectation.

```
>>> b = 8 # Définition de la variable b en lui affectant la valeur 8
>>> b = "abc" # Modification de la variable b en lui affectant la valeur "abc"
```

Dans ce cas, la valeur `8` est perdue après la seconde affectation.

ATTENTION : Le symbole d'affectation ne représente pas une égalité : il n'est pas symétrique. Par exemple, si vous écrivez l'instruction `3 = a`, l'interpréteur vous affichera un message d'erreur.

```
>>> 3 = a
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

Dans le fonctionnement de Python, la création d'une variable établit un lien entre le nom de la variable et un emplacement dans la mémoire de la machine ; ce lien est sauvegardé dans une table de référencement. Pour information (nous ne l'utiliserons plus dans la suite), il est possible de récupérer l'adresse en mémoire d'une variable en utilisant la fonction `id`.

```
>>> a = 512
>>> id(a)
133682634424560
```

Remarque 2 : En Python, l'affectation est uniquement une instruction (ce n'est pas le cas dans tous les langages). En particulier, comme on le remarque sur les exemples ci-dessus, rien ne s'affiche lors d'une affectation dans le mode interactif.

II.A - Les noms de variables

Le programmeur choisit librement le nom de ses variables, mais il doit veiller à choisir des noms significatifs de façon à faciliter la lecture de son code. Il faut néanmoins respecter certaines règles :

- Les noms de variables ne peuvent contenir que des lettres, des chiffres et le caractère underscore _ ; ce nom ne doit pas commencer par un chiffre.
- La **casse** est importante. Par exemple, les noms `var` et `Var` désignent deux variables différentes. Pour éviter les difficultés de lecture, l'usage est de se limiter aux lettres minuscules pour les noms de variables.
- Le nom choisi ne doit pas correspondre à un mot réservé de Python comme `print`, `if`, `and`, etc.

II.B - Accéder à la valeur d'une variable

Après avoir affecté une valeur à une variable, on peut utiliser cette valeur simplement en mentionnant son nom. Lors de l'exécution du code par l'interpréteur, le nom est remplacé par la valeur stockée dans la variable.

Exemple 3 : Poursuivons le code de l'exemple précédent.

```
>>> print(message)
bonjour
>>> a + 2 # L'ordinateur calcule a + 2
5
>>> a # La valeur de a n'a pas été modifiée par le calcul précédent
3
>>> a = a * 10 # Voir les explications ci-dessous
>>> a
30
```

Lorsque l'ordinateur exécute l'instruction `a = a * 10`, il commence par calculer l'expression de droite `a * 10` avec la valeur actuelle affectée à `a`, puis il affecte ce résultat à la variable `a`.

Question 4 : Pour chacun des trois programmes ci-dessous, déterminer le résultat affiché par l'interpréteur lors de son exécution (sans utiliser votre ordinateur dans un premier temps).

```
# Programme 1
>>> a = 2
>>> a = 7
>>> a = 3 * a
>>> a = a - 1
>>> a
```

```
# Programme 2
>>> a = 2
>>> b = a * a
>>> b = a * b
>>> b = b * b
>>> b
```

```
# Programme 3
>>> a = 1
>>> b = 2
>>> c = a
>>> a = b
>>> b = c
>>> print(a, b, c)
```

Question 5 : On suppose que l'on dispose de deux variables `a` et `b`. Écrire un programme permettant d'échanger les valeurs des variables `a` et `b`.

II.C - Affectations multiples

Il est également possible d'effectuer plusieurs affectations en une seule ligne avec la syntaxe suivante.

```
>>> a, b, c = 1, "deux", 3.0
>>> print(a, b, c)
1 deux 3.0
```

Cette syntaxe est très utile pour échanger en une seule ligne les valeurs de deux variables.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> print(a, b)
2 1
```

Partie III Les types de données simples

Un aspect essentiel des langages de programmation est la notion de **type** : tout objet manipulé par Python en possède un, qui caractérise la manière dont il est représenté en mémoire, et dont vont dépendre les fonctions qu'on peut lui appliquer.

Python est un langage typé dynamiquement, c'est-à-dire qu'on peut changer le type d'une variable au cours de l'exécution d'un programme. En pratique, lorsqu'on définit une variable en Python, on ne précise pas son type en général : ce dernier est déterminé par l'interpréteur lors de l'exécution du code.

Pour déterminer le type d'une variable en Python, on peut utiliser la fonction `type`.

```
>>> a = 1
>>> type(a)
int
```

Dans la suite, on présente les différents types simples de données et les manipulations qui leurs sont associées.

III.A - Les entiers

Les nombres **entiers** (*integer* en anglais) sont représentés en Python par le type `int`. On peut effectuer les opérations binaires suivantes sur les entiers :

- l'addition de `a` et `b` avec l'expression `a + b` ;
- la soustraction de `a` par `b` avec l'expression `a - b` ;
- la multiplication de `a` et `b` avec l'expression `a * b` ;

De plus, si b est un entier positif, on peut également effectuer les opérations binaires suivantes :

- le quotient dans la division euclidienne de `a` par `b` avec l'expression `a // b` ;
- le reste dans la division euclidienne de `a` par `b` avec l'expression `a % b` ;
- l'exponentiation de `a` à la puissance `b` avec l'expression `a**b`.

On peut également utiliser les syntaxes courtes suivantes.

```
a += b # Équivalent à a = a + b
a -= b # Équivalent à a = a - b
a *= b # Équivalent à a = a * b
```

Remarques 3 :

a) Les priorités opératoires sont les mêmes qu'en mathématique. On peut également utiliser des parenthèses dans les expressions à calculer.

b) Python permet de manipuler des entiers de taille arbitrairement grande (les seules limites étant fixées par la capacité de la machine utilisée).

c) Nous étudierons dans l'année la manière dont les nombres entiers sont représentés en mémoire.

Question 6 : Vous souhaitez offrir 2^{50} bonbons à 13 amis en les répartissant de manière équitable dans des paquets individuels.

1) Combien de bonbons contient chaque paquet ?
2) Combien de bonbons ne sont pas utilisés ?

III.B - Les flottants

Les nombres réels sont approchés par les nombres **flottants** qui sont représentés en Python par le type `float`. On peut effectuer les opérations binaires suivantes sur les flottants :

- l'addition de `a` et `b` avec l'expression `a + b` ;
- la soustraction de `a` par `b` avec l'expression `a - b` ;
- la multiplication de `a` et `b` avec l'expression `a * b` ;
- la division de `a` par `b` avec l'expression `a / b` ;
- l'exponentiation de `a` à la puissance `b` avec l'expression `a**b`.

Remarques 4 :

a) Lorsque `a` et `b` sont des entiers, Python effectue automatiquement la conversion en flottant lorsqu'on utilise l'opérateur de division `/`. Par exemple, l'instruction `6 / 2` renvoie le nombre flottant `3.0`.

b) L'opérateur d'exponentiation permet de calculer des racines n-ième. Par exemple, pour calculer la racine carrée de 3, il suffit d'utiliser l'instruction `3**(1/2)`.

c) Les priorités opératoires sont les mêmes qu'en mathématique. On peut également utiliser des parenthèses dans les expressions à calculer.

d) Nous étudierons dans l'année la manière dont les nombres flottants sont représentés en mémoire.

ATTENTION : Ne jamais oublier que les flottants ne font qu'approcher les nombres réels.

```
>>> a = 0.1
>>> a + a + a + a + a + a + a + a + a + a
0.9999999999999999
```

III.C - Les booléens

Un **booléen** correspond à une variable qui ne prend que deux valeurs possibles : « Vrai » ou « Faux ». On les utilise principalement pour faire des tests de comparaison logique. Les booléens sont représentés en Python par le type `bool` ; les valeurs possibles étant notées `True` et `False`.

On peut effectuer les opérations suivantes sur les booléens :

- l'opérateur logique « non » est notée `not` en Python ;
- l'opération logique « et » est notée `and` en Python ;
- l'opération logique « ou » est notée `or` en Python.

Voici les résultats des différentes opérations décrites ci-dessus.

```
>>> not(True)
False
>>> not(False)
True
```

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Remarque 5 : Lorsque l'interpréteur Python évalue une expression booléenne contenant des opérateurs `and` et `or`, il le fait de manière paresseuse, i.e. en court-circuitant l'expression à droite de l'opérateur logique dès qu'il le peut. En effet, il est fréquent que la connaissance de la valeur de l'expression à gauche de l'opérateur soit suffisante pour en déduire le résultat de l'opération.

Remarque 6 : Python assimile `True` à l'entier `1` et `False` à l'entier `0`. En particulier, il est possible d'utiliser les opérations des entiers sur les booléens, ce qui peut être pratique dans certaines situations.

```
>>> True + True + False + True
3
>>> True * False + 5 * True + 2 * False
5
```

On peut également utiliser les opérateurs de comparaison pour générer des booléens :

- le symbole `==` pour l'égalité ;
- le symbole `!=` pour la différence ;
- les symboles `<` et `>` pour les inégalités strictes ;
- les symboles `<=` et `>=` pour les inégalités larges.

ATTENTION : Il est préférable de comparer des données de même type pour éviter des résultats non souhaités.

Exemple 4 :

```
>>> 1 + 1 == 2
True
>>> 1 == 2
False
>>> 1 + 3 != 2 + 2
False
```

```
>>> 1 + 4 != 3
True
>>> 1 + 5 < 10
True
>>> 1 > 2
False
```

```
>>> 3.14 <= 3.15
True
>>> 10**(1/2) >= 3
True
>>> 2**20 <= 3**10
False
```

III.D - Conversion de type

En Python, il est possible de faire certaines conversions de type. Il faut employer ces conversions avec des précautions : il n'est pas toujours simple d'en prédire le comportement.

Exemple 5 : Voici quelques exemples de conversions de type.

```
>>> a = 2.5
>>> int(a)
2
```

```
>>> a = 7
>>> float(a)
7.0
```

```
>>> int(True)
1
>>> int(False)
0
```

```
>>> float(True)
1.0
>>> float(False)
0.0
```

```
>>> bool(7)
True
>>> bool(0)
False
```

```
>>> bool(7.0)
True
>>> bool(0.0)
False
```

Partie IV Les fonctions en Python

Une **fonction** (en informatique) est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande (l'appel de la fonction). Vous connaissez déjà quelques fonctions comme `print` par exemple.

IV.A - Définir une fonction

La syntaxe Python pour la définition d'une fonction est la suivante.

```
def nom_fonction(arg_1, ..., arg_n):
        bloc_instructions
```

Les contraintes sur le nom d'une fonction sont les mêmes que celles sur le nom d'une variable. La ligne commençant par le mot-clé **`def`** se termine obligatoirement par un symbole « : », qui introduit le bloc des instructions appartenant à la fonction (délimité grâce à l'indentation obtenue en utilisant la touche **tabulation**). Ce bloc d'instructions constitue le **corps de la fonction**.

Pour commencer, on peut définir une fonction sans argument en entrée.

Exemple 6 : On définit ci-dessous une fonction `compteur` sans argument en entrée.

```
>>> def compteur():
...         print(5, 4, 3, 2, 1)
...         print("BOOM")
```

Une fois cette fonction définie, on peut l'utiliser en l'appelant autant de fois que nécessaire.

```
>>> compteur() # Première utilisation
5 4 3 2 1
BOOM
>>> compteur() # Deuxième utilisation
5 4 3 2 1
BOOM
>>> compteur() # Troisième utilisation
5 4 3 2 1
BOOM
```

Nous pouvons également donner des arguments en entrée d'une fonction.

Exemple 7 : Écrivons une fonction `produit_aff` qui affiche le produit de deux nombres donnés comme arguments d'entrée.

```
>>> def produit_aff(x, y):
...         print(x * y)
>>> produit_aff(2, 3)
6
>>> produit_aff(7, 5)
35
```

La fonction définie dans l'exemple précédent ne permet pas récupérer le résultat, car nous lui avons seulement demandé d'afficher le résultat. En général, il est préférable de renvoyer le résultat avec le mot-clé **`return`** : cela permettra de réutiliser ce dernier dans la suite du programme.

Exemple 8 : Écrivons une fonction `produit_ret` qui renvoie le produit de deux nombres donnés comme arguments d'entrée.

```
>>> def produit_ret(x, y):
...         return(x * y)
>>> res = produit_ret(2, 3)
>>> print(res)
6
```

Remarques 7 :

a) Dès que l'interpréteur Python rencontre le mot-clé **`return`**, il retourne la valeur demandée et il arrête l'exécution de la fonction. En particulier, on placera généralement **`return`** en fin de fonction puisque les éventuelles instructions se trouvant après **`return`** ne seront ni lues ni exécutées.

b) Il est possible de placer plusieurs **`return`** dans le corps d'une fonction, mais dans un souci de compréhension du code, on se limite en général à l'utilisation d'un unique **`return`** dans une fonction.

c) Lorsque l'instruction **`return`** n'est pas rencontrée lors de l'appel d'une fonction, Python considère que le résultat renvoyé par la fonction est **`None`**, objet qui indique l'absence de valeur en Python.

```
>>> res = produit_aff(2, 3)
6
>>> print(res)
None
```

ATTENTION : En Python, contrairement à d'autres langages, il n'est pas nécessaire de préciser le type des arguments en entrée. On peut par exemple remarquer que notre fonction `produit_aff` est utilisable avec d'autres types d'arguments que des entiers ou des flottants.

```
>>> produit_aff(True, False)
0
```

Plus généralement, on peut utiliser une fonction avec tous types d'arguments du moment que les instructions du corps de la fonction restent cohérentes avec ces types de données.

```
>>> produit_aff("a", "b")
TypeError: can't multiply sequence by non-int of type 'str'
```

Question 7 : Écrire une fonction `affine(a, b, x)` qui prend comme argument trois nombres `a`, `b` et `x` et qui renvoie le nombre $ax+b$.

Question 8 : Écrire une fonction `cercle(r)` permettant d'afficher le périmètre d'un cercle de rayon `r` et l'aire d'un disque de rayon `r` sous la forme suivante.

```
>>> cercle(1.0)
Un cercle de rayon 1.0 cm a pour périmètre : 6.28 cm.
Un disque de rayon 1.0 cm a pour aire : 3.14 cm2.
```

IV.B - Portée d'une variable

Les variables définies dans une fonction ne peuvent être utilisées que localement, c'est-à-dire qu'à l'intérieur de la fonction qui les a définies. Tenter d'appeler une telle variable depuis l'extérieur de la fonction qui l'a définie provoquera une erreur. De telles variables sont qualifiées de locales, par opposition aux variables globales, qui sont définies en dehors de toutes fonctions et dont le contenu est accessible à tout niveau.

```
>>> a = 1 # Définition d'une variable globale a.
>>> def fonction():
...         print(a) # La variable a est accessible dans la fonction.
...         b = a + 1 # Définition d'une variable locale b.
...         return(b)
>>> fonction()
2
>>> b # La variable b n'existe pas en dehors de la fonction.
NameError: name 'b' is not defined
```

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.

IV.C - Signature d'une fonction

La **signature** d'une fonction est l'ensemble des informations suivantes :

- le nom de la fonction ;
- les arguments de la fonction et leur type ;
- le type de ce qui est retourné par la fonction.

Lorsqu'on définit une fonction en Python, on peut préciser sa signature avec la syntaxe suivante.

```
def nom_fonction(arg_1:type_1, ..., arg_n:type_n) -> type_sortie:
        instruction 1
        ...
        instruction p
```

Exemple 9 : Redéfinissons la fonction `produit_ret` de la partie précédente en précisant que les deux arguments en entrée sont des entiers et qu'elle retourne un entier.

```
>>> def produit_ret(x:int, y:int) -> int:
...         return(x * y)
```

Remarque 8 : Lorsqu'une fonction ne retourne aucune valeur, on note `None` pour `type_sortie`.

Exemple 10 : Redéfinissons la fonction `produit_aff` de la partie précédente en précisant que les deux arguments en entrée sont des entiers et qu'elle ne retourne aucune valeur.

```
>>> def produit_aff(x:int, y:int) -> None:
...         print(x * y)
```

ATTENTION : La signature permet seulement de donner des indications à l'utilisateur, mais Python n'effectue aucun contrôle sur les types à l'entrée et à la sortie de la fonction : rien n'empêche de continuer à appeler notre fonction avec des variables d'autres types que ceux précisés dans la signature.

Question 9 : Écrire une fonction `discriminant(a:float, b:float, c:float) -> float` qui renvoie le discriminant du trinôme $aX^2 + bX + c$.

Partie V Structures conditionnelles

V.A - La structure if

L'instruction conditionnelle **`if`** permet de soumettre l'exécution d'un bloc d'instructions à une condition. La syntaxe Python est la suivante.

```
if condition:
        bloc_instructions
```

La ligne commençant par le mot-clé **`if`** se termine obligatoirement par un symbole « : », qui introduit le bloc des instructions (délimité grâce à l'indentation obtenue en utilisant la touche tabulation) qui sera exécuté ou non en fonction de la valeur de l'expression booléenne `condition`.

On a deux cas :

- si `condition` est évaluée à **`True`**, alors l'interpréteur exécute les lignes de `bloc_instructions` ;
- si `condition` est évaluée à **`False`**, alors l'interpréteur ignore les lignes de `bloc_instructions`.

Exemple 11 : On suppose que la variable x contient un entier. L'exécution du code ci-dessous affichera le message uniquement si x est strictement positif.

```
if x > 0:
        print("Le nombre", x, "est strictement positif.")
```

V.B - La structure if ... else

En plus d'un bloc à n'exécuter que lorsque la condition est vérifiée, on peut ajouter un autre bloc à n'exécuter que dans le cas contraire.

```
if condition:
        bloc_instructions_1
else:
        bloc_instructions_2
```

On a deux cas :

- si `condition` est évaluée à **`True`**, alors l'interpréteur exécute les lignes de `bloc_instructions_1` et ignore celles de `bloc_instructions_2` ;
- si `condition` est évaluée à **`False`**, alors l'interpréteur exécute les lignes de `bloc_instructions_2` et ignore celles de `bloc_instructions_1` ;

Exemple 12 : On suppose que la variable x contient un entier. L'exécution du code ci-dessous affichera le message adéquat en fonction du signe de x.

```
if x > 0:
        print("Le nombre", x, "est strictement positif.")
else:
        print("Le nombre", x, "est négatif ou nul.")
```

V.C - La structure if ... elif ... else

```
if condition_1:
        bloc_instructions_1
elif condition_2:
        bloc_instructions_2
else:
        bloc_instructions_3
```

On a plusieurs cas :

- si `condition_1` est évaluée à **`True`**, alors l'interpréteur exécute les lignes de `bloc_instructions_1` et ignore tous les autres blocs d'instructions ;
- si `condition_1` est évaluée à **`False`** et `condition_2` est évaluée à **`True`**, alors l'interpréteur exécute les lignes de `bloc_instructions_2` et ignore tous les autres blocs d'instructions ;
- si `condition_1` et `condition_2` sont évaluées à **`False`**, alors l'interpréteur exécute `bloc_instructions_3` et ignore tous les autres blocs d'instructions.

Exemple 13 : On suppose que la variable `x` contient un entier. L'exécution du code ci-dessous affichera le message adéquat en fonction de la valeur de `x`.

```
if x > 0:
        print("Le nombre est strictement positif.")
elif x < 0:
        print("Le nombre est strictement négatif.")
else:
        print("Le nombre est nul.")
```

Remarques 9 :

a) On peut ajouter autant de **`elif`** qu'on souhaite dans cette structure conditionnelle.

b) La partie de la structure avec **`else`** n'est pas obligatoire.

Question 10 : Écrire une fonction `parité(x:int) -> None` qui affiche un message indiquant la parité de `x` sous la forme suivante.

```
>>> parité(2)
Le nombre 2 est pair.
>>> parité(3)
Le nombre 3 est impair.
```

Question 11 : Écrire une fonction `max2(x:int, y:int) -> int` qui renvoie le maximum entre `x` et `y`.

Question 12 : Écrire une fonction `max3(x:int, y:int, z:int) -> int` qui renvoie le maximum entre `x`, `y` et `z`.

Question 13 : Écrire une fonction `racines(a:float, b:float, c:float) -> None` qui affiche les racines réelles du trinôme $aX^2 + bX + c$.

Question 14 : On rappelle qu'une année est bissextile si elle est un multiple de 400 ou si elle est multiple de quatre mais pas de cent. Écrire une fonction `nb_jours(num:int) -> int` qui renvoie le nombre de jours dans l'année `num`.

Partie VI Structures itératives

VI.A - La boucle for

Rappelons que la fonction `range` peut prendre entre un et trois arguments entiers :

- l'expression `range(b)` énumère les entiers $0, 1, \ldots, b-1$;
- l'expression `range(a, b)` énumère les entiers $a, a+1, a+2, \ldots, b-1$;
- l'expression `range(a, b, p)` énumère les entiers $a, a+p, a+2p, \ldots, a+kp$ où k est le plus grand entier vérifiant l'inégalité $a+kp < b$.

La syntaxe Python pour utiliser une boucle **`for`** est la suivante.

```
for variable in range(...):
        bloc_instructions
```

La ligne commençant par le mot-clé **`for`** se termine obligatoirement par un symbole « : », qui introduit le bloc des instructions délimité grâce à l'indentation obtenue en utilisant la touche tabulation. Ce bloc d'instructions constitue le **corps de la boucle**. La `variable` va prendre les différentes valeurs de l'énumération produite par l'instruction range. Pour chacune de ces valeurs, le code `bloc_instructions` sera exécuté.

Exemple 14 : Voici un exemple simple d'utilisation d'une boucle **`for`**.

```
>>> for k in range(0,3):
...         print("La variable k de la boucle vaut", k)
La variable k de la boucle vaut 0
La variable k de la boucle vaut 1
La variable k de la boucle vaut 2
```

Il est également possible d'imbriquer des boucles dans d'autres boucles. Dans ce cas, il faut faire attention à choisir des noms différents pour les variables et à respecter les règles d'indentation pour délimiter les différents blocs d'instructions.

Exemple 15 : Voici un exemple simple avec deux boucles imbriquées.

```
>>> for i in range(0,4):
...         for j in range(0,2):
...                 print("Le couple (i,j) vaut", i, j)
Le couple (i,j) vaut 0 0
Le couple (i,j) vaut 0 1
Le couple (i,j) vaut 1 0
Le couple (i,j) vaut 1 1
Le couple (i,j) vaut 2 0
Le couple (i,j) vaut 2 1
Le couple (i,j) vaut 3 0
Le couple (i,j) vaut 3 1
```

Question 15 : Écrire une fonction `love() -> None` qui affiche cent fois la phrase "J'adore Python!".

Question 16 : Écrire une fonction `table(n:int) -> None` qui affiche la table de multiplication de `n` sous la forme suivante.

```
>>> table(3)
0 x 3 = 0
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
6 x 3 = 18
7 x 3 = 21
8 x 3 = 24
9 x 3 = 27
10 x 3 = 30
```

```
>>> table(7)
0 x 7 = 0
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Question 17 : Écrire une fonction `nb_div(n:int) -> int` qui renvoie le nombre de diviseurs positifs de l'entier non nul `n`.

Question 18 : Écrire une fonction `decompte(n:int) -> None` qui affiche un compte à rebours sous la forme suivante.

```
>>> decompte(6)
6
5
4
3
2
1
BOOM
```

Question 19 : Écrire une fonction `factorielle(n:int) -> int` qui renvoie l'entier $n! = \prod_{k=1}^{n} k$.

Question 20 : Écrire une fonction `est_premier(n:int) -> bool` qui renvoie un booléen indiquant si `n` est un nombre premier ou non.

Question 21 : Écrire une fonction `triplet(n:int) -> None` qui affiche tous les triplets (i, j, k) de $[\![1, n]\!]^3$ sous la forme suivante.

```
>>> triplet(2)
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
```

VI.B - La boucle while

Une boucle **while** exécute une suite d'instructions tant qu'une certaine condition est réalisée. La syntaxe Python pour utiliser une boucle **while** est la suivante.

```
while condition:
        bloc_instructions
```

La ligne commençant par le mot-clé **while** se termine obligatoirement par un symbole « : », qui introduit le bloc des instructions précisé grâce à l'indentation obtenue en utilisant la touche tabulation. Ce bloc d'instructions constitue le **corps de la boucle**. Ce dernier sera exécuté en boucle tant que la valeur de l'expression booléenne `condition` est évaluée à **True**.

Exemple 16 : Le code ci-dessous permet de déterminer le plus grand entier dont le carré est inférieur ou égal à un entier `n`.

```
k = 0
while (k+1)**2 <= n:
        k += 1
print(k)
```

ATTENTION : Si votre condition est toujours vérifiée, alors votre programme ne s'arrêtera jamais après son lancement : on parle de **boucle infinie**. Si vous avez lancé une boucle infinie, vous pouvez arrêter manuellement l'exécution de votre code en utilisant un bouton de votre `IDE`. Ainsi, en général, la condition dans une boucle **while** dépend d'au moins une variable dont le contenu est susceptible d'être modifié dans le corps de la boucle.

Question 22 : Écrire une fonction `seuil_fact(a:int) -> int` qui renvoie le plus petit entier $n \in \mathbb{N}$ tel que $n! \geqslant$ `a`.

Question 23 : On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = 1$ et

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \frac{1}{2}u_n + 1.$$

On admet que la suite $(u_n)_{n\in\mathbb{N}}$ converge vers 2. Écrire une fonction `seuil_suite(e:float) -> int` qui prend comme argument un flottant `e` > 0 et qui renvoie le plus petit entier $n \in \mathbb{N}$ tel que $|u_n - 2| <$ `e`.

Question 24 : Écrire une fonction `nb_chiffres(a:int) -> int` qui renvoie le nombre de chiffres de l'entier `a`.

Question 25 : Écrire une fonction `pgcd(a:int, b:int) -> int` qui renvoie le pgcd des nombres `a` et `b`.

Question 26 : Étant donné un entier $a \in \mathbb{N}^*$, on considère la suite de Collatz $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = a$ et

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si} \quad u_n \text{ est pair,} \\ 3u_n + 1 & \text{si} \quad u_n \text{ est impair.} \end{cases}$$

La conjecture de Collatz affirme que pour tout $a \in \mathbb{N}^*$, il existe un indice $n \in \mathbb{N}^*$ tel que $u_n = 1$.

1) Écrire une fonction `temps_de_vol(a:int) -> int` qui renvoie le plus petit indice $n \in \mathbb{N}$ tel que $u_n = 1$ pour la suite de Collatz de terme initial $u_0 = a$.
2) Écrire une fonction `altitude_maximale(a:int) -> int` qui renvoie la plus grande valeur de la suite de Collatz de terme initial $u_0 = a$.

Partie VII Quelques nouvelles notions

VII.A - Spécification d'une fonction

La **spécification** d'une fonction est l'ensemble des informations suivantes :

- la signature de la fonction ;
- les éventuels préconditions précisant les contraintes portant sur les arguments de la fonction ;
- une postcondition précisant ce que retourne la fonction.

On peut décrire la spécification d'une fonction Python au sein de la documentation de cette fonction (*docstring* en anglais). Cette documentation apparaît immédiatement après l'entête de la fonction et est encadrée par trois paires de guillemets.

Exemple 17 : Voici un exemple de documentation d'une fonction Python.

```
def racine_carrée(x:float) -> float:
        """
        Calcule la racine carrée d'un nombre.

        Entrée :
                x : type float, le nombre positif dont on veut la racine carrée.

        Sortie :
                type float, la racine carrée de x.
        """

        return(x**(1/2))
```

L'utilisateur peut accéder à cette documentation en utilisant la fonction `help`.

Exemple 18 : On peut récupérer la documentation de l'exemple précédent avec la commande ci-dessous.

```
>>> help(racine_carrée)
Help on function racine_carrée in module __main__:

racine_carrée(x: float) -> float
    Calcule la racine carrée d'un nombre.

    Entrée :
            x : type float, le nombre positif dont on veut la racine carrée.

    Sortie :
            type float, la racine carrée de x.
```

Remarque 10 : Vous pouvez également utiliser la fonction `help` pour obtenir des informations sur les fonctions déjà implémentés dans Python.

Question 27 : Afficher la documentation de la fonction `print` de Python.

Question 28 : Ajouter des spécifications à vos fonctions `nb_div`, `factorielle` et `temps_de_vol`.

VII.B - Assertion

Une **assertion** est une instruction qui vérifie si une condition est vérifiée dans l'état courant du programme. La syntaxe Python pour utiliser une assertion est la suivante.

```
# Syntaxe sans message d'erreur
assert condition

# Syntaxe avec un message d'erreur
assert condition, message_erreur
```

On a deux cas :

- si `condition` est évaluée à `True`, alors l'exécution de l'assertion ne fait rien ;
- si `condition` est évaluée à `False`, alors l'exécution de l'assertion lève une exception de type `AssertionError` qui interrompt l'exécution du programme en affichant `message_erreur` s'il a été précisé.

Rappelons que la signature et la spécification d'une fonction permettent uniquement de donner des indications à l'utilisateur : Python n'effectue aucun contrôle sur les types à l'entrée et à la sortie de la fonction, les préconditions et la postcondition. Ainsi, nous utiliserons principalement les assertions afin d'effectuer ces vérifications.

Exemple 19 : On reprend l'exemple de la partie précédente en ajoutant des assertions afin de contrôler le type de son argument en entrée et la précondition qu'il doit vérifier.

```
def racine_carrée(x:float) -> float:
        """
        Calcule la racine carrée d'un nombre.

        Entrée :
                x : type float, le nombre positif dont on veut la racine carrée.

        Sortie :
                type float, la racine carrée de x.
        """

        assert type(x) == float, "l'argument x doit être un nombre flottant."
        assert x >= 0, "l'argument x doit être positif."

        return(x**(1/2))
```

Si on essaye d'utiliser la fonction `racine_carrée` avec un mauvais argument, l'exécution est interrompue.

```
>>> racine_carrée(True)
AssertionError: l'argument x doit être un nombre flottant.
>>> racine_carrée(-1.0)
AssertionError: l'argument x doit être positif.
```

Remarque 11 : En réalité, les assertions sont un outil utilisé uniquement lors du développement d'un programme afin de détecter d'éventuelles erreurs et de les corriger. Une fois la conception terminée, les assertions sont remplacées par une gestion plus fine des exceptions avec le mot-clé `raise`, mais ce dernier est hors programme.

Question 29 : Ajouter des assertions à vos fonctions `nb_div`, `factorielle` et `temps_de_vol` afin de vérifier le type de leurs arguments d'entrée et les préconditions portant sur ces arguments.