TP Informatique

BCPST1B 2026–2027

L.-C. LEFÈVRE
Lycée Hoche, Versailles

En construction

À propos

Ce document contient l'intégralité des TP d'informatique pour l'année scolaire 2026–2027 de la BCPST1B du Lycée Hoche de Versailles. Il comporte **3 TP** et **36 exercices**.

Il a été rédigé entièrement avec le langage **Typst**. Il s'agit d'un tout récent langage de mise en forme de documents ayant pour ambition d'être une alternative moderne à LaTeX. Encore en plein développement, il fournit déjà de très nombreux outils pour la publication scientifique (mathématiques, bibliographie, coloration syntaxique du code, importation de données CSV, …) supportés nativement avec une grande facilité d'utilisation. Il a, dès sa publication, fédéré de nombreux contributeurs et utilisateurs. Le but de ce document est donc aussi d'expérimenter ce nouveau langage et d'en démontrer les possibilités.

En particulier, la police de caractères s'appelle **New Computer Modern**. Le thème de coloration syntaxique s'appelle **Pastie**. L'intégralité du document rédigé avec l'éditeur de texte **GNU Emacs** pour lequel il existe un mode Typst, avec un clavier en configuration **BÉPO**.

Aucun outil d'Intelligence Artificielle n'a été utilisé pour produire ce document, notamment le contenu, le code informatique, et les illustrations.

Table des matières

Introduction

Le langage Python est un langage de programmation qui permet de donner des instructions à l'ordinateur, écrites de façon lisible par un humain. Par rapport aux nombreux (des milliers !) langages de programmation existants, il a les avantages suivants :

- **Moderne** : utilisé de plus en plus dans de nombreux milieux scientifiques, c'est même le langage phare dans le domaine de l'intelligence artificielle, et il est encore (en 2026) en pleine expansion en terme de nombre d'utilisateurs et d'offres d'emplois. Pour une fois l'Éducation Nationale est vraiment à la pointe…
- **Libre** : n'importe qui peut l'installer sur son ordinateur, le tester, lire la documentation, le modifier et l'améliorer, et contribuer au développement. Le langage Python n'appartient pas à une entreprise mais à la *communauté Python*, qui est particulièrement grande et active.
- **Facile à lire** : l'accent est mis sur la lisibilité et la facilité à programmer. Cela en fait un bon choix pour l'enseignement et pour l'apprentissage d'un premier langage.
- **Dynamique** : en plus du mode script qui exécute tout un programme d'un seul coup, il y a un mode interactif qui exécute les commandes unes par unes et permet de faire des tests très simplement, inspecter le contenu des variables, naviguer dans l'aide, exécuter son programme pas à pas pour trouver des erreurs.
- **Haut niveau** : le langage fournit de très nombreuses fonctions et constructions qui sont « éloignées » du fonctionnement interne de l'ordinateur mais beaucoup plus faciles à manipuler pour un humain. En comparaison d'autres langages, un programme Python est souvent *plus court* et *plus rapide à écrire*, ce qui le rend si apprécié. En contrepartie, un programme Python est aussi *plus lent à fonctionner* : ce n'est pas le langage absolument idéal pour toutes les situations.
- **Extensible** : de très nombreuses librairies sont disponibles, c'est-à-dire des extensions et des fonctions supplémentaires, qui permettent d'interagir avec les fichiers de l'ordinateur, le réseau, tracer des graphiques, traiter des statistiques, des images… et tout est expliqué dans la documentation.

En conséquence, pour un élève, c'est aussi nettement plus difficile qu'un langage de calculatrices ou qu'un langage comme *Scratch* (au collège avec le chat) car ceux-ci ont été conçus spécifiquement pour l'enseignement mais n'ont aucune utilité professionnelle !

Enfin les logiciels viennent avec leur propre documentation, écrite par ceux qui ont créé le logiciel, et qui décrit très précisément leur fonctionnement. *Tout* ne peut donc pas venir de l'enseignant : celui-ci connait avant tout les concepts généraux de la programmation et a appris le langage Python, mais n'est pas responsable du logiciel lui-même et encore moins de ses nombreuses bibilothèques et logiciels liés… Il passe notamment du temps à lire les documentations, en anglais bien entendu ! Alors surtout pour progresser, en séance de TP, vous êtes toujours libres d'expérimenter !

À retenir

Pour progresser en informatique :

- **Expérimentez**, **testez avec vos propres valeurs** et avec vos propres messages les programmes proposés. Essayez avec des valeurs qui marchent bien et aussi avec des valeurs qui pourraient faire échouer les programmes et prenez le temps d'observer le résultat.
- **Lisez les messages d'erreur**. Ne paniquez pas mais apprenez à les lire et à les décoder et à corriger vous-même les erreurs en ré-essayant encore et encore.
- **Lisez les documentations**, les aides-mémoires, les aides interactives (fonction Python `help()`). Testez les fonctions et les options existantes décrites dans ces documentations.

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 1.1
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 1.2

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 1.3
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 1.4
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 1.5
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 1.6
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 1.7
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 1.8
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 1.9
Écrire un programme qui demande à l'utilisateur d'entrer deux nombres `a` et `b` et en donne la moyenne.

Exercice 1.10
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 1.11
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 1.12
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.

TP 2
Conditions et boucles

Aujourd'hui nous écrivons des programmes avant tout dans le mode script (sur plusieurs lignes) ; le mode interactif sert à faire des tests courts ou à inspecter le contenu ou le type des variables. Séparer les programmes et les exercices par des **cellules** délimitées par une ligne commençant par la suite de trois caractères **#%%** permet de tout garder sur une même page (donc d'enregistrer dans un même fichier `.py`), mais de n'exécuter qu'un morceau à la fois et pas tout depuis le début. La présentation du fichier doit donc ressembler à ceci, comme dans les corrections :

```
#%% exercice 1
...

#%% exercice 2
...
```

On utilise alors l'option « exécuter la cellule » (*execute cell*), aussi sur le raccourci clavier `Ctrl` + `Entrée`.

I Conditions simples

Le mot-clé `if` suivi d'une condition introduit un morceau de programme qui va être exécuté **si** la condition est vérifiée. Éventuellement, le mot-clé `else` (**sinon**) introduit un morceau de programme qui va être exécuté dans le cas contraire.

Voici un exemple de bout de programme, qu'on peut recopier tel quel et tester :

```
x = int(input("Entrez un nombre : "))
if x >= 0:
    print("positif")
else:
    print("strictement négatif")
print("FIN")
```

Cela est bien entendu un concept fondamental de la programmation, qui permet de rendre un programme interactif et dont le résultat va dépendre des entrées.

La syntaxe générale en Python est la suivante :

```
if condition:
    instructions
else:
    instructions sinon
```

où :

- `condition` désigne n'importe quelle expression booléenne que le programme va tester. Elle est formée notamment, on le rappelle, avec
  - L'égalité : `==`
  - La différence : `!=`
  - Les comparaisons strictes : `<`, `>`
  - Les comparaisons larges : `<=`, `>=`
  - Les mots-clés `and`, `or`, `not`
- `instructions` est du code Python, qui peut être sur plusieurs lignes, qui va être exécuté seulement si la condition a été évaluée à `True`. La partie du programme qui va être exécutée est tout ensemble décalée vers la droite (on dit **indentée**). En général le décalage est de 4 espaces, ou un seul caractère tabulation ; le logiciel sert notamment à bien aligner les lignes. Elle forme un **bloc d'instructions**.
- `instructions sinon` est un **autre bloc d'instructions** qui va être exécuté dans le cas contraire.

- Ensuite, le code qui n'est plus décalé vers la droite ne fait plus partie des blocs d'instructions ; il est donc exécuté dans tous les cas. Le `else` n'est d'ailleurs pas du tout obligatoire : si la condition est fausse alors le premier bloc d'instruction n'est pas exécuté et le programme passe directement à la suite.

On n'oubliera pas non plus le symbole double points, qui fait partie de la syntaxe, termine la ligne et introduit le bloc d'instructions. Il permet aussi au logiciel de proposer directement d'indenter, le saut de ligne après le double point décale automatiquement à droite et on n'a rarement besoin d'écrire à la main les espaces ou tabulations.

Exercice 2.1
Écrire un programme qui demande à l'utilisateur son âge, et affiche s'il est majeur ou mineur.

Exercice 2.2
Écrire un programme qui demande à l'utilisateur un nombre, et affiche sa valeur absolue.

Exercice 2.3
Choisissez un mot de passe secret ; écrire un programme qui demande à l'utilisateur d'entrer un mot, et qui lui dit si c'est le bon mot de passe ou non.

Le bloc d'instructions peut lui-même contenir d'autres conditions emboitées. L'indentation des blocs est alors cruciale.

Exercice 2.4
Améliorer programme d'exemple ci-dessus pour demander à l'utilisateur un nombre et afficher s'il est positif, négatif ou nul. Sans regarder la suite du TP.

II Conditions en cascade

Il arrive que l'on veuille tester une condition plus complexe qui ne se traduit pas aussi simplement que « si… alors ». Le mot-clé `elif` est la contraction de « else, if » (**sinon, si**) et introduit une condition qui va être testée si la précédente était fausse, ainsi qu'un bloc d'instruction correspondant à exécuter. Par exemple le programme du dernier exercice peut se ré-écrire ainsi :

```
x = int(input("Entrez un nombre : "))
if x > 0:
    print("positif")
elif x < 0:
    print("négatif")
else:
    print("nul")
```

Il est tout à fait possible d'enchainer plusieurs `elif` ; les conditions sont simplement testées les unes à la suite des autres. Le `else` final capture le cas où **aucune** des conditions n'a été vérifiée ; il n'est toujours pas obligatoire, si aucune condition n'est vérifiée le programme passe simplement à la suite. La syntaxe générale ressemble donc à :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif ... :
    ...
else:
    instructions sinon
```

Ainsi lors de l'exécution :
- Le programme teste la `condition1`. Si elle est vraie alors il exécute `instructions1`.
- Sinon, il vérifie la `condition2`. Si elle est vraie alors il exécute `instructions2`.
- Et ainsi de suite.
- À la fin, si aucune condition n'a été vérifiée, le programme exécute le bloc d'instructions du `else`.

Exercice 2.5
Écrire un programme qui demande son âge à l'utilisateur, et affiche s'il est majeur, mineur, ou senior (plus de 65 ans).

Exercice 2.6
Écrire un programme qui demande à l'utilisateur sa note au bac (attention, ce n'est pas forcément un nombre entier) et affiche à quelle mention cela correspond. Vous pouvez l'assortir librement d'un commentaire sur la note…

À retenir

Les conditions en cascade sont testées **successivement** quand la précédente a échoué. À cause de cela, il y a un ordre logique et naturel dans lequel on doit écrire ses conditions. De plus le mot `else` n'est pas suivi par une condition (c'est un « sinon » tout court) et il s'exécute quand aucune des conditions précédentes n'a été vérifiée.

III Boucles

Une **boucle** permet de répéter des instructions automatiquement. C'est à partir de maintenant que les programmes deviennent *vraiment* intéressants : ils automatisent des tâches qui seraient bien pénibles pour un humain.

Dans ce TP on se concentre sur la boucle `while`. Le bloc d'instructions (**corps** de la boucle) est répété **tant que** (traduction de *while*) une condition est vérifiée. La syntaxe est tout simplement la suivante :

```
while condition:
    instructions
```

alors lors de l'exécution :
- Le programme teste si la condition est vraie,
- Si oui, il exécute le bloc d'instructions. Une fois fini, il recommence à évaluer la condition et ainsi de suite.
- Sinon, il sort simplement de la boucle et passe à la suite.

Pour que cela soit intéressant, il faut que la condition puisse varier à chaque passage dans la boucle ! Sinon, si elle est toujours vraie, rien ne l'arrête et on obtient une **boucle infinie**… La méthode de base pour répéter une instruction un certain nombre n de fois est de déclarer une variable appelée **compteur**, que l'on va augmenter de 1 (on dit **incrémenter**) à chaque passage dans la boucle, et de tester la condition `i < n` :

```
i = 0
while i < 3:
    print("i =", i)
    i = i + 1
print("fin avec i =", i)
```

produit le résultat suivant :

```
i = 0
i = 1
i = 2
fin avec i = 3
```

Avertissement

Dans chaque boucle, on porte une attention particulière à :
- La valeur à laquelle le compteur est initialisé (essayez avec `i = 1`)
- L'utilisation de la comparaison stricte `<` ou large `<=` (essayez de remplacer par `<=`)
- L'incrémentation au début ou à la fin de la boucle (essayez d'échanger les deux lignes dans le bloc d'instructions),
- Ne pas oublier l'incrémentation (essayez de l'enlever !)

L'exemple ci-dessus est le plus standard et est écrit de telle façon à ce qu'il répète exactement `n` fois.

Exercice 2.7
Écrire un programme qui affiche les nombres x^2 pour $1 \leqslant x \leqslant 10$.

Exercice 2.8
Écrire un programme qui demande à l'utilisateur un nombre `n` et compte à rebours : affiche n puis $n-1$ puis… jusqu'à 0. Il y a deux façons possibles, tester les deux :

1. Une boucle croissante comme ci-dessus, où on incrémente `i`, mais en affichant la quantité `n - i`.
2. Une boucle décroissante, où on décrémente `i` (c'est-à-dire le faire diminuer de 1 avec l'opération `i = i - 1`), mais alors il faut changer la condition dans le `while` ainsi que la valeur initiale de `i`.

IV Application aux suites

Pour calculer les termes successifs d'une suite, on se sert en plus d'une variable `u` qui à chaque passage dans la boucle va devenir le terme suivant. L'exemple de base pour les puissances de 2 est :

```
u = 1
i = 0
while i < 5:
    print(u)
    u = 2 * u
    i = i + 1
```

qui produit l'affichage

```
1
2
4
8
16
```

autrement dit les 5 premières puissances, soit de 2^0 à 2^4.

Exercice 2.9
Écrire un programme qui demande à l'utilisateur un nombre `n` et calcule la somme $1 + 2 + \cdots + n$.

L'intérêt de la boucle `while`, c'est aussi de pouvoir s'arrêter quand une certaine condition sur la suite devient vérifiée — c'est à dire la continuer **tant que** elle n'est **pas** vérifiée.

Exercice 2.10
Au début de l'année 2026, la population mondiale est estimée à environ 8,30 milliard milliard d'habitants. Elle augmente d'environ 0,84 % chaque année. Écrire un programme qui affiche la population mondiale estimée sur les années futures (afficher à la fois l'année et la population, par exemple `Année 2026 : 8300000000`) si le taux de croissance reste le même, et s'arrête quand elle dépasse 10 milliards.

Exercice 2.11 (*)
La **suite de Syracuse** est la suite entière $(S_n)_{n\in\mathbb{N}}$ définie par : le nombre $S_0 \geqslant 1$ est à déterminer par l'utilisateur, et ensuite

$$S_{n+1} = \begin{cases} S_n/2 & \text{si } S_n \text{ est pair} \\ 3 \times S_n + 1 & \text{si } S_n \text{ est impair} \end{cases}$$

1. Que se passe-t-il si $S_0 = 1$?
2. Écrire un programme qui demande à l'utilisateur le nombre S_0 puis affiche tous les termes de la suite jusqu'à ce qu'un terme soit égal à 1.
3. La célèbre **conjecture de Syracuse** est l'énoncé selon lequel quelque soit le nombre S_0, la suite finit par retomber sur 1. Il s'agit d'un problème ouvert célèbre…

   Écrire un programme qui nous aide à vérifier cette conjecture, en demandant à l'utilisateur un entier N puis en testant la conjecture pour tout $1 \leqslant S_0 \leqslant N$, affichant pour chaque valeur testée si la conjecture est bien vraie.
4. En plus, pouvez-vous compter, pour chaque S_0, en combien d'étapes la suite revient à 1 ?

On peut vérifier la conjecture pour des valeurs extrêmement grandes, mais ceci ne constitue malheureusement toujours pas une démonstration mathématique…

TP 3
Fonctions

Aujourd'hui et à partir de maintenant nous écrivons avant tout des **fonctions** dans le mode script. Cela permet de faire varier des paramètres, et de récupérer une valeur calculée.

L'exécution d'un code de déclaration de fonction ne produit aucun affichage, il faut donc tester les fonctions dans le mode interactif ou bien directement à la suite du script. On continue à séparer les exercices par des cellules avec les trois caractères `#%%`. On rappelle que le raccourci clavier Ctrl + Entrée permet d'exécuter la cellule ; dans le mode interactif, jouer avec les flèches ↑ et ↓ pour faire réapparaître les entrées précédentes.

I Notion de fonction

I.1 Déclaration et appel

Une fonction correspond à un morceau de programme ré-utilisable, dans lequel on peut faire varier des paramètres. Elle se déclare avec le mot-clé `def`. Étudions l'exemple suivant :

```
def somme(x, y):
    print("La somme de", x, "et de", y, "est", x+y)
```

Ce code est une **déclaration** de fonction, son exécution ne produit rien mais enregistre la fonction.

Les tests suivants en mode interactif permettent de faire varier les paramètres. On parle d'**appel** de la fonction.

```
>>> somme(3, 4)
La somme de 3 et de 4 est 7
>>> somme(-6, 2)
La somme de -6 et de 2 est -4
```

Les variables `x` et `y` s'appellent des **arguments** de la fonction. Une fonction `f` peut avoir un ou plusieurs arguments, ou aucun — dans ce dernier cas l'appel s'écrit juste `f()`. Les arguments peuvent être de n'importe lesquels des types manipulés jusque là.

Ce qui suit le mot-clé `def` forme un **bloc d'instructions**, exactement comme dans les conditions et les boucles. On parle du **corps** de la fonction. Il peut donc contenir lui aussi des variables, des conditions et des boucles.

I.2 L'instruction `return`

Dans le bloc d'instructions, lorsque le programme tombe sur une ligne commençant par `return`, **l'exécution de la fonction s'arrête**. La valeur ou l'expression qui suit est dite **renvoyée par la fonction** et on peut alors récupérer sa valeur. Prenons l'exemple

```
def moyenne(x, y):
    return (x + y) / 2
```

alors l'appel

```
>>> m = moyenne(12, 14)
```

- Appelle la fonction, en donnant les valeurs `x` ← `12` et `y` ← `14`,
- Calcule le résultat de `(x + y) / 2` qui donne le nombre `13.0`,
- **renvoie** ce résultat, et le met ici dans la variable `m` ← `13.0`.

On peut le vérifier :

```
>>> m
13.0
```

Ce comportement ne serait pas possible si on écrivait `print((x + y) / 2)` au lieu de `return` : la valeur serait bien affichée mais serait ensuite perdue et on ne pourrait pas la récupérer dans une variable. Or, il est crucial pour la ré-utilisabilité de la fonction de pouvoir ainsi enregistrer la valeur qu'elle calcule.

```
>>> def moyenne(x, y): print((x + y) / 2)
>>> m = moyenne(12, 14)
13.0
# on dirait que ça marche bien
>>> m
# rien ne se passe
>>> print(m)
None
# mais où est donc passé le résultat ???
```

L'instruction `return` n'est pas obligatoire. Si l'exécution de la fonction arrive au bout sans avoir rencontré de `return`, elle ne renvoie pas de valeur. Si `return` est présent sans valeur de retour, l'exécution de la fonction s'arrête mais ne renvoie pas de valeur. L'exemple suivant est donc bien valide

```
def f():
    print("Cette fonction n'a pas de paramètres ni de valeur de retour.")
    print("Est-elle pour autant utile ?")
    return
    print("Ceci, par contre, ne s'affichera jamais.")
```

et son appel ne fait qu'afficher toujours le même texte des deux premières lignes.

Un autre exemple instructif est celui-ci, comme son nom l'indique :

```
def divise_proprement(x, y):
    if y == 0:
        print("Il ne faut JAMAIS diviser par 0 !!!")
        return
    return x / y
```

Alors :

- Si l'argument `y` est `0`, on entre dans le premier `if`. Le message d'avertissement s'affiche. Puis la fonction se termine, à cause du `return`.
- Sinon, le bloc d'instruction du `if` n'est pas exécuté. La fonction calcule `x / y` et renvoie cette valeur.
- Remarquez que du coup le `else` n'est pas nécessaire ici : l'instruction `return x / y` se produit uniquement dans le cas où `y` est non-nul, sinon elle s'est arrêtée avant !

Le test donne :

```
>>> q = divise_proprement(6, 2)
>>> q
3.0
# q contient bien une valeur, le résultat
>>> q = divise_proprement(5, 0)
Il ne faut JAMAIS diviser par 0 !!!
>>> q
# ici rien ne s'affiche ! q existe mais ne contient pas de valeur !
```

En fait, `q` contient la valeur spéciale `None`, qui risque d'apparaitre si on écrit `print(q)`, et indique une variable qui existe mais ne contient pas de valeur.

À retenir

À partir de maintenant, dans les exercices en TP mais aussi à l'écrit, nous écrivons la plupart du temps des **fonctions**, qui ont des arguments, effectuent un certain calcul, et renvoient le résultat avec `return`. Sauf demande explicite, elles ne font pas de `print()` (le résultat calculé est renvoyé, c'est l'utilisateur qui décidera ce qu'il fait avec, ce n'est pas la fonction qui décide de comment elle va l'afficher) ni de `input()` (de même, les arguments sont passés à la fonction par l'utilisateur, ce n'est pas à la fonction de décider avec quel message elle va les demander).

I.3 Notion de variable locale

Une fonction a bien le droit d'utiliser des variables dans son corps, et les arguments eux-mêmes sont des variables à part entière. Cependant, à moins que ces variables soient renvoyées avec un `return`, elles sont **détruites** à la fin de l'exécution de la fonction et donc ne sont pas accessibles au reste du programme. On parle de **variables locales**. Reprenons l'exemple

```
def moyenne(x, y):
    m = (x + y) / 2
    return m
```

alors en mode interactif

```
>>> moyenne(12, 14)
13.0
>>> x
NameError: name 'x' is not defined
>>> m
NameError: name 'm' is not defined
```

Ces variables n'existent plus !

Cela correspond à la notion mathématique de variable non-libre, on dit aussi *liée* ou *muette*.

Cela signifie aussi que l'on peut écrire

```
>>> m = 10
>>> moyenne(12, 14)
13.0
>>> m
10
```

car la variable `m` qui est manipulée à l'intérieure de la fonction n'est pas vraiment la même que celle qui pourrait déjà exister avant.

II À vous de jouer !

Les fonctions suivantes ne font pas intervenir de connaissances en Python autres que celles vues dans les TP précédents, mais la mise en forme est différente : les fonctions ont des arguments, font des calculs, utilisent éventuellement des variables locales, des conditions et des boucles, et renvoient une valeur avec `return`.

Exercice 3.1
Écrire la fonction `perimetre_rectangle(a, b)` qui renvoie le périmètre du rectangle de côtés a et b.

Exercice 3.2
Écrire la fonction `valeur_absolue(x)` qui renvoie la valeur absolue de `x`.

Exercice 3.3
Écrire la fonction `maximum(a, b, c)` qui renvoie le plus grand des trois nombres entre `a`, `b` et `c`.

Exercice 3.4

1. Écrire la fonction `partie_entiere(x)` qui prend en argument un nombre à virgule flottante `x`, qui pour l'instant ne marchera que si `x` est positif, et qui calcule sa partie entière de la façon suivante : une boucle `while` cherche le plus grand entier `n` qui est inférieur ou égal à `x`.
2. Bonus : améliorer la fonction pour traiter séparément le cas où `x` est négatif. Attention à bien la tester dans tous les cas, `x` positif ou négatif, entier ou non.

Exercice 3.5
Écrire la fonction qui calcule la partie entière de la racine carrée sans utiliser la fonction partie entière ni la fonction racine carrée *(Hein ???)*

III L'aide

Toutes les fonctions définies en Python possèdent une aide accessible (en mode interactif) avec la commande `help()` : testez

```
>>> help(abs)
```

Chacun peut créer de la documentation pour sa propre fonction en insérant juste après la déclaration du texte entre trois guillemets doubles successifs *""" documentation """*, ce que Python appelle une **docstring**, contraction de *documentation string* (chaine de documentation), et qui peut même tenir sur plusieurs lignes :

```
def moyenne(x, y):
    """ C'est MA fonction qui calcule la moyenne.
        Entrez deux nombres x et y, et elle vous donne la moyenne. """
    return (x + y) / 2
```

Lancer le programme puis essayer en mode interactif :

```
>>> help(moyenne)
```

La possibilité d'intégrer la documentation dans le corps même de la fonction, de naviguer dans l'aide en mode interactif, et la grande qualité de la documentation Python déjà existante, contribuent pour beaucoup à la diffusion de ce langage et à sa facilité d'apprentissage. C'est une bonne pratique de documenter son programme pour que d'autres puissent l'utiliser.

Exercice 3.6
Remplir soigneusement les documentations des fonctions de la partie II.

IV Les modules

Un **module** (ou aussi : bibliothèque, librairie) est un ensemble de fonctions. Elles sont groupées par thème et permettent de donner des nouvelles possibilités au langage. Les modules Python de base permettent de trouver les fonctions mathématiques, les nombres aléatoires, l'écriture dans des fichiers, mais aussi de connaitre la date, communiquer en réseau, traiter des images… Cela contribue au succès de Python d'avoir des milliers de bibliothèques disponibles et d'interagir dans autant de situations.

Un module se charge (une fois pour toute !) avec la commande `import`, écrite en mode interactif ou avant le code qui va l'utiliser, qui a plusieurs syntaxes. Prenons pour exemple le module `math` qui comporte beaucoup de fonctions mathématiques comme la fonction exponentielle `exp`, la fonction sinus `sin`, et la constante mathématique `pi`. Il n'est pas chargé par défaut, donc on ne peut pas les utiliser tout de suite. On a les possibilités suivantes, avec des différences dans la manière dont sont ensuite nommées les fonctions :

- Importer tout, et y accéder avec le préfixe :

  ```
  import math
  math.exp(x), math.sin(x), math.pi, ...
  ```

- Importer seulement les fonctions dont on a besoin, et y accéder sans préfixe :

  ```
  from math import exp, sin, pi
  exp(x), sin(x), pi
  ```

- Il est courant aussi de donner un **alias** au module, introduit avec le mot-clé `as` :

```
import math as m
m.exp(x), m.sin(x), m.pi, ...
```

Exemple courant : le sous-module `pyplot` du module `matplotlib` est un peu long à écrire.

```
import matplotlib.pyplot as plt
plt.plot(), plt.show(), ...
```

Cela marche aussi avec les fonctions et variables elles-mêmes, bien que l'utilité en soit discutable…

```
from math import pi as plus_beau_nombre_de_l_univers
print(plus_beau_nombre_de_l_univers)
# 3.141592653589793
```

- On trouve aussi parfois la syntaxe

```
from math import *
```

qui importe d'un coup **toutes** les fonctions du module, sans le préfixe. **Cette syntaxe est déconseillée** car le module peut contenir beaucoup de fonctions et on ne sait pas forcément à l'avance lesquelles… Si l'utilisateur a déjà une fonction `f` et que le module contient lui-même une fonction aussi nommée `f`, alors celle de l'utilisateur sera « désactivée » après l'importation et remplacée par celle du module ! Et si on importe plusieurs modules contenant chacun une fonction `f`, on ne sait plus à laquelle on fait référence ! C'est acceptable pour le module `math`, mais certainement pas pour Numpy.

Un module possède aussi une documentation, accessible une fois chargé :

```
>>> import math
>>> help(math)
```

Exercice 3.7
Importer le module `math` comme ci-dessus, lire sa documentation, et éventuellement celle des fonctions contenues dedans, et trouver :

1. La fonction racine carrée,
2. La fonction partie entière,
3. Les PGCD (plus grand commun diviseur) et PPCM (plus petit commun multiple).

… d'ailleurs, il y a plusieurs parties entières ? Quelle différence avec la conversion de type `int(x)` ? Tester avec plusieurs valeurs, entières ou non, positives ou négatives.

Remarque. On a donné ici diverses syntaxes ; le but est d'être capable de les reconnaitre et de les interpréter, pas de tout savoir par cœur. En général, un sujet écrit qui utilise un module précise comment il est importé, par exemple « on suppose qu'on a importé le module `math` avec la commande `import math as m` » et donc il faut savoir que les fonctions s'appellent alors `m.exp(x)`, `m.sin(x)`, etc. Si ce n'est pas précisé, alors cela peut être au candidat de ne pas oublier d'écrire la commande d'importation !

Aux épreuves orales le candidat dispose d'un temps de préparation devant un ordinateur avec les logiciels Pyzo et Spyder. Il n'a bien entendu pas accès librement à tout l'internet, mais utiliser l'aide intégrée de Python est parfaitement légal.

Exercice 3.8
Écrire une fonction Python qui correspond à la fonction mathématique

$$f : x \mapsto \sin^3(2\pi x)e^{\sqrt{x}}$$

Un autre module d'intérêt est le module `random`, lié à tout ce qui est l'aléatoire, qui contient la fonction `randint` générant un nombre aléatoire entier. Nous l'approfondirons en lien avec les chapitres de probabilités.

Exercice 3.9
Importer uniquement cette fonction, lire sa documentation, et écrire une fonction `dé()` qui renvoie le résultat d'un lancer de dé cubique aléatoire.

Exercice 3.10
Écrire une fonction `réponse()` qui, en tirant au hasard un nombre entre 1 et 4, renvoie un choix aléatoire entre les mots `"oui"`, `"non"`, `"peut-être"` ou `"je ne sais pas"`.

V D'autres exercices

La commande spéciale `assert`, suivie d'une condition et éventuellement d'un message d'erreur, permet de faire échouer tout le programme, en provoquant une erreur (en rouge) et en affichant le message, si la condition **n'est pas** vérifiée. On écrit par exemple une ligne

```
assert x >= 0, "x doit être positif"
```

Le verbe anglais *assert* (comme dans *assertion*) se traduit ici avec le sens plus fort de « affirmer que », autrement dit on affirme que `x` doit être positif pour pouvoir continuer. Cela lui donne un sens proche du langage mathématique « supposons $x \geqslant 0$ » — si ce n'est pas le cas alors la suite n'a pas de sens et on s'arrête.

Il est courant qu'une fonction démarre par une ou plusieurs assertions qui servent à vérifier si les arguments donnés sont bien valides et évitent de faire des calculs qui provoqueront plus tard une erreur ou donneront des résultats incohérents. Par exemple la fonction `divise_proprement()` de § I.2 peut s'écrire

```
def divise_proprement(x, y):
    assert y != 0, "Il ne faut JAMAIS diviser par 0 !!!"
    return x / y
```

et l'exécution s'affiche comme un message d'erreur Python

```
>>> divise_proprement(5, 0)
AssertionError: Il ne faut JAMAIS diviser par 0 !!!
```

Dans les exercices suivants, vous pouvez utiliser `assert` pour éviter que les programmes donnent des erreurs, des boucles infinies ou des résultats incohérents.

Exercice 3.11
La *moyenne harmonique* de deux nombres x, y tous les deux strictement positifs ou bien tous deux strictement négatifs est l'unique nombre H tel que

$$\frac{1}{H} = \frac{1}{2}\left(\frac{1}{x} + \frac{1}{y}\right)$$

Écrire une fonction `moyenne_harmonique(x, y)` qui renvoie la moyenne harmonique de `x` et de `y`.

Exercice 3.12
Écrire une fonction `seuil(t)` qui prend en argument un nombre réel $t \geqslant 1$ et renvoie le plus petit entier $n > 0$ pour lequel

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \geqslant t.$$

Tester des valeurs de t de plus en plus grandes entre 3 et 20 (doucement, pas à pas !)

Remarque. Il n'est jamais exigé des étudiants d'utiliser `assert`, et c'est même plutôt déconseillé à l'écrit. En effet, en algorithmique on considère que le programme est correct quand il répond correctement à la question qui lui est posée sous les hypothèses données, et c'est l'utilisateur qui est responsable de respecter ces hypothèses. Par exemple une phrase telle que « écrire une fonction qui prend en argument un réel x supposé positif […] »

se comprend comme « la fonction doit donner le bon résultat si x est positif » et ne doit pas essayer de traiter un cas négatif. Ce n'est vraiment pas la même façon de penser que pour la conception d'une interface utilisateur conviviale, où on demanderait à l'utilisateur « veuillez entrer un nombre x positif » et où il faudrait gérer toutes les erreurs possibles, en demandant à l'utilisateur « vous avez entré un nombre négatif, merci de recommencer […] » et cela compliquerait beaucoup le programme alors que nous nous concentrons sur la logique.

Exercice 3.13
On rappelle que la divisibilité se teste à partir de l'opération `%` (`a % b` est le reste dans la division euclidienne de a par b, dont le résultat est nul si et seulement si b divise a).

1. Écrire une fonction `est_premier(n)` qui prend en argument un entier n en supposant $n \geqslant 2$ et qui renvoie `True` si n est premier et `False` sinon, en tentant de diviser n par tous les entiers strictement entre 1 et n.
2. En déduire une fonctions `premiers(n)` qui affiche tous les nombres premiers inférieurs ou égaux à n.