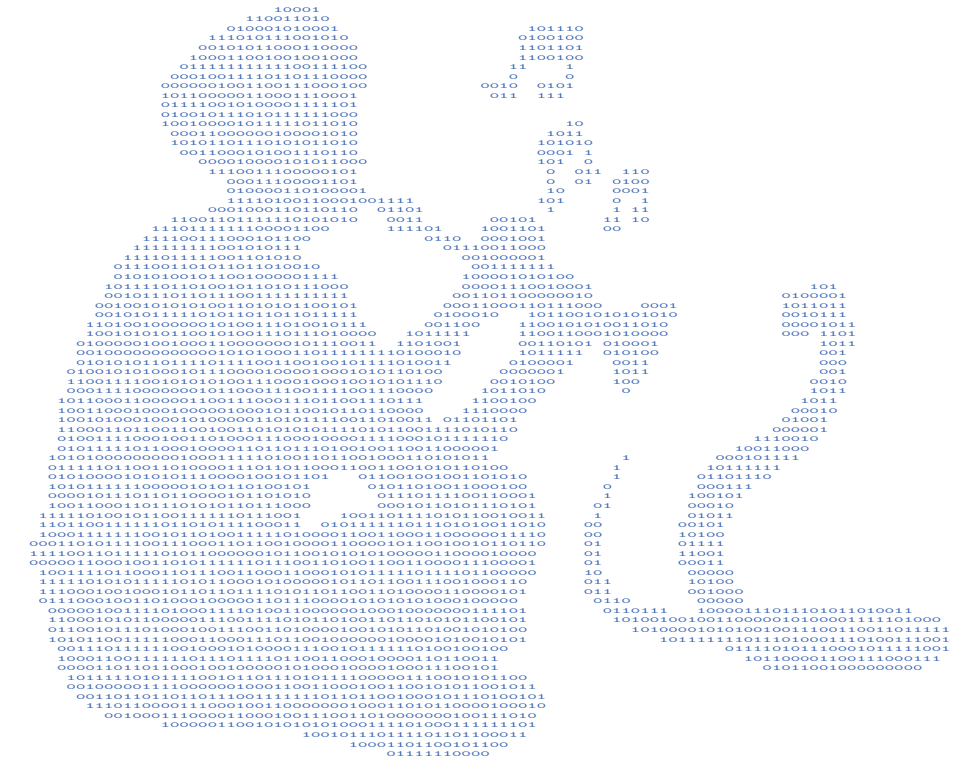
général et technologique
professionnel
métiers d'art
supérieur

Lycée polyvalent
Henri Brisson
25 avenue Henri Brisson
18100 Vierzon

Apprivoiser Python

Pour bien commencer en NSI



R. Casati et P. Violet

Année 2023 – 2024

Avant-propos

Ce document a été rédigé par les enseignants de NSI du lycée Henri Brisson de Vierzon. C'est un support au stage de rentré dédié à l'apprentissage du langage Python.

Document en construction. En raison d'une charge de travail importante pour les professeurs qui conduisent la réforme du lycée, ce document est encore largement incomplet. Rassurez-vous, ce qui n'est pas encore présent dans ce document sera bien traité en classe par les professeurs. Ceci nécessitera alors, de votre part, une attention particulière car le seul support dont vous disposerez sera les notes que vous prendrez.

Se familiariser avec ce document. Pour aider à la lisibilité, certaines parties de ce document sont mises en évidences de la manière suivante. Elles sont de nature diverse.


Info

Un détail qui a son importance mais qui peut souvent être omis en première lecture.


Attention

Un point délicat auquel il convient de prêter attention.


Astuce

Une remarque qui vous fera voir les choses d'un autre point de vue et pourra souvent vous faire gagner du temps.


Exercice

Exercice 1 Un exemple d'exercice

De nombreux exercices pour pratiquer les notions étudiées. La plupart du temps, ils nécessitent d'utiliser Python.

Même si l'intégralité de ce document est destinée à Python, certaines parties vont plus loin encore dans la description du langage et de ses concepts.

Savoir-faire

Bien apprendre son cours

Même s'ils seront rares dans ce document [1], ce sont des points de méthode inscrits au programme et qu'il faut impérativement maîtriser.
Attention, ce ne sont pas les seuls éléments sur lesquels vous serez évalués mais ne pas les avoir acquis vous coûtera probablement cher.

Comment lire ce document ? Dans une large mesure, il est possible de parcourir ce document en autonomie. Vous avez donc la possibilité d'avancer sa lecture en dehors des heures de cours. En classe, certaine parties seront traitées par l'élève seul, face à une machine, les autres seront présentées par l'enseignant.

Il est indispensable de pratiquer Python chez soi. Ceci constitue une part importante du travail à la maison de ce cours. Reprendre les passages de ce document sur lesquels l'enseignant est passé rapidement ou qui ont mal été compris est une très bonne méthode pour progresser sans lacune.

1. Vous en retrouverez plus encore dans les notes de cours des chapitres à venir.

À propos de la page de couverture. L'image en couverture de ce document a été réalisée avec Python. Le fichier original provient du site Pixabay, il est placé sous la licence Pixabay qui permet une utilisation libre, même pour un usage commercial et ne nécessite pas d'attribution. Elle est reproduite ci-contre mais vous la trouverez à l'adresse : `https://pixabay.com/fr/vectors/serpent-charmeur-silhouette-indien-3652464/`.

Cette image est ensuite traitée de la manière suivante. Elle est d'abord convertie en image binaire (le « vrai » noir et blanc) puis enregistrée au format GIF avec la commande ci-dessous :

```
$ convert charmeur.svg -resize 100x100 -monochrome charmeur.gif
```

C'est alors que Python rentre en action. On balaye les pixels de l'image, s'il est blanc, on affiche un espace, s'il est noir, on tire au hasard un chiffre binaire (0 ou 1) et on l'affiche.

```python
from PIL import Image
from random import randint

img = Image.open("charmeur.gif")
pixels = img.load()

for y in range(img.height):
    for x in range(img.width):
        print(' ' if pixels[x, y] else randint(0, 1), end='')
    print()
```

Remarquez « l'expressivité » de Python : le rapport entre la complexité de l'algorithme et la concision du code. Nous aurons l'occasion de la mettre en évidence de manière plus spectaculaire encore au cours de l'année.

Table des matières

Chapitre 1

Utiliser Python au lycée

I L'environnement CAPYTALE

Afin de pouvoir développer en langage Python, nous utiliserons la plateforme CAPYTALE accessible directement depuis l'Espace Numérique de Travail (ENT). Cet espace permet à votre professeur de distribuer un support d'activité qui sera à compléter, mais vous avez aussi la possibilité de créer votre propre espace de développement.

Puisqu'il est accessible depuis l'ENT, vous avez la possibilité de l'ouvrir depuis chez vous, et de manière générale n'importe quel dispositif communiquant (même le téléphone). Ainsi, pas d'oubli de matériel !

II L'environnement Jupyter

Lors de certaines activités, nous utiliserons l'application générale Jupyter Notebook, application mère de CAPYTALE. Cette application nécessite une installation au préalable sur une machine, mais il n'est pas demandé de l'installer sur votre poste personnel. Dans tous les cas, CAPYTALE et Jupyter utilisent la même extension de fichier. Il sera donc très facile de faire la passerelle de l'un vers l'autre.

Pour exécuter Jupyter depuis les postes du lycée, il suffit d'ouvrir une fenêtre "Invite de commande" et d'y taper "jupyter notebook". Au préalable, il sera intéressant de se positionner dans le bon dossier, afin de retrouver facilement ses fichiers. Nous traiterons plus en détail cette partie en classe.

III L'interprète Python et l'élaboration de scripts

Il existe de nombreuses applications afin de programmer en Python. Au lycée, nous retrouverez globalement partout l'application "EduPython", qui dispose d'un interface sobre, mais suffisament claire pour être utilisé.

On utilisera ce genre d'application pour développer, entre autre, nos propres scripts contenant les fonctions que nous avons réalisé au fur et à mesure de l'année. Une fois le script créé, nous pourrons facilement l'appeler dans CAPYTALE en ajoutant le fichier à l'activité.

Chapitre 2

Mettre au point des programmes

Un *programme* c'est d'abord un texte écrit dans un *langage de programmation*, on parle de *code source* du programme . Il existe beaucoup de langages de programmation mais ils sont le plus souvent organisés autour des mêmes concepts : variables, expressions, affectations, séquences d'instructions et structures de contrôle (conditionnelles, boucles). Ce sont ces concepts qui sont présentés dans ce document, avec comme support, le langage Python.

Dans ce cours, nous utiliserons le langage Python dans sa version 3 (ou supérieure) pour plusieurs raisons. En effet, Python [1] est

- simple d'usage,
- concis,
- libre et gratuit,
- multiplateforme,
- largement répandu,
- dispose de bibliothèques variées et riches (on dit que Python est « piles incluses »).

I Modifier un programme existant

I.1 La bataille navale

Exécuter le programme ci-dessous.

```
a = 4
b = 7
print("À vous de jouer")
x = int(input("x = "))
y = int(input("y = "))
if x == a and y == b:
    print("Coulé")
else:
    if x == a or y == b:
        print("En vue")
    else:
        print("À l'eau")
```

Si l'on tente de jouer avec ce programme, on se rend vite compte qu'il affiche « À vous de jouer » puis attend que l'on tape deux nombres au clavier. Si ces deux nombres sont 4 et 7, il affiche « Coulé » ; si le premier est 4 ou le second 7, mais pas les deux, il affiche « En vue », sinon il affiche « À l'eau ». C'est une sorte de bataille navale mais avec un seul bateau, toujours placé au même endroit et d'une seule case de long.

1. En toute rigueur, on devrait distinguer le langage Python de son interpréteur (le plus souvent CPython). En effet, la simplicité de la syntaxe est un atout du langage tandis que le caractère multiplateforme est une qualité de l'interpréteur.

Exercice

Exercice 2

Modifier ce programme afin qu'il affiche « À toi de jouer » et non « À vous de jouer ».

Exercice

Exercice 3

Modifier ce programme afin que le bateau soit sur la case de coordonnées $(6\,;9)$.

Astuce

On remarque qu'il n'est pas nécessaire de comprendre le détail du fonctionnement d'un programme pour le modifier. Cependant, on peut ajouter des *commentaires* pour clarifier certaines choses. Ces commentaires servent juste aux humains qui lisent le code du programme, ils ne sont pas exécutés. En Python, un commentaire commence par le symbole `#`. On peut par exemple ajouter le commentaire suivant au programme précédent :

```
# Version simplifiée de la bataille navale
a = 4
b = 7
print("À vous de jouer")
...
```

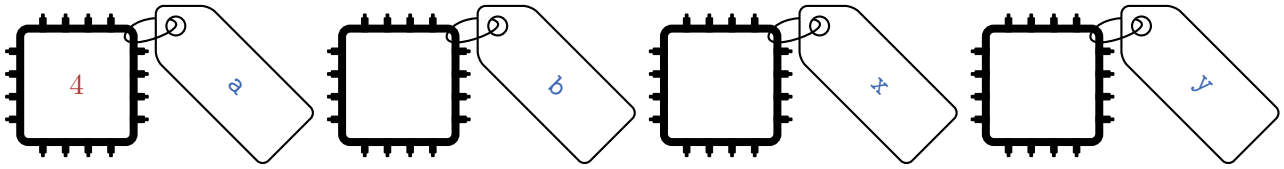
Exercice

Exercice 4

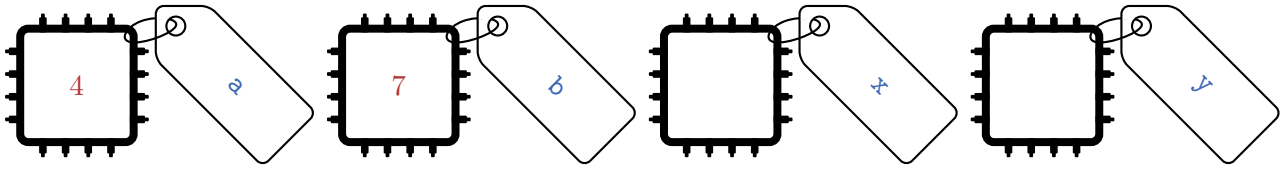
Ajouter les commentaires « abscisse du bateau » et « ordonnée du bateau » au bon endroit pour clarifier le code.

I.2 Déroulement du programme pas-à-pas

On reprend le programme de la section précédente. Observons-le pour essayer d'en comprendre la signification. La première ligne contient l'instruction `a = 4`. Pour comprendre ce qu'il se passe quand on exécute cette instruction, il faut imaginer que la mémoire de l'ordinateur que l'on utilise est composée d'une multitude de petites boîtes. Chacune de ces boîtes porte un nom et peut contenir une valeur. Exécuter l'instruction `a = 4` a pour effet de mettre la valeur `4` dans la boîte de nom `a`, comme sur la figure ci-dessous.



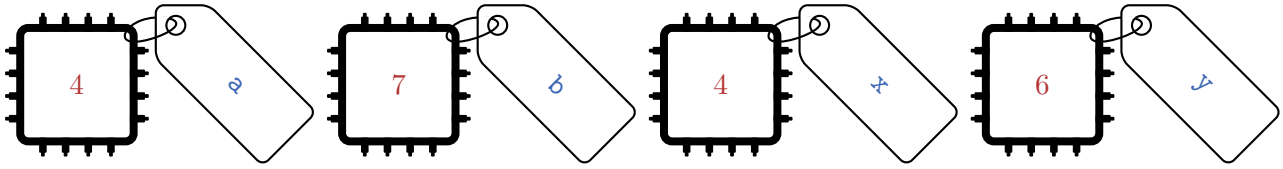
Après avoir exécuté cette instruction, on exécute `b = 7`, ce qui a pour effet de mettre la valeur `7` dans la boîte de nom `b`.



On exécute ensuite l'instruction `print("À vous de  jouer")`, ce qui a pour effet d'afficher à l'écran `À vous de jouer`.

On exécute ensuite l'instruction `x = int(input())`, ce qui a pour effet d'interrompre le déroulement du programme jusqu'à ce que l'utilisateur tape un nombre au clavier. Ce nombre est alors mis dans la boîte de nom `x`.

De même, exécuter l'instruction `y = int(input())` a pour effet d'interrompre le déroulement du programme jusqu'à ce que l'utilisateur tape un nombre au clavier. Ce nombre est alors mis dans la boîte de nom `y`. À ce point du programme, les boîtes de nom `a`, `b`, `x` et `y` contiennent chacune un nombre. Les nombres 4 et 7 pour les deux premières et les deux nombres entrés au clavier par l'utilisateur pour les deux dernières.



L'exécution du programme doit alors différer selon que les deux nombres donnés par l'utilisateur sont 4 et 7 ou non. Si c'est le cas, on veut afficher `Coulé`, si ce n'est pas le cas on veut faire autre chose. C'est ce que fait l'instruction :

```
if x == a and y == b:
    print("Coulé")
else:
    if x == a or y == b:
        print("En vue")
    else:
        print("À l'eau")
```

Quand on écrit cette instruction, il est *essentiel* de décaler les lignes 2, 4 et 6 de quatre caractères vers la droite, et les lignes 5 et 7 de huit caractères en commençant chaque ligne par des espaces. Cela s'appelle *indenter* une instruction.

Une instruction mal indentée a souvent des conséquences catastrophiques sur l'exécution du programme. Dans le meilleur des cas, Python va se plaindre d'une mauvaise indentation, dans d'autres, le programme s'exécutera sans vous prévenir que vous n'avez pas écris ce que vous pensiez.

Exécuter cette instruction a pour effet de calculer la valeur de l'expression booléenne `x == a and y == b`, où l'opération `and` est la conjonction et. Cette valeur est `True` (vrai) quand `x` est égal à `a` et `y` est égal à `b`, ou `False` (faux) quand ce n'est pas le cas. En fonction de la valeur de cette expression, on exécute ou bien l'instruction `print("Coulé")` ou bien l'instruction :

```
if x == a or y ==b:
    print("En vue")
else:
    print("À l'eau")
```

Cette instruction étant de la même forme, son exécution a pour effet de calculer la valeur de l'expression booléenne `x == a or y == b`, où l'opération `or` est la conjonction ou, et en fonction de la valeur de cette expression d'exécuter ou bien l'instruction `print("En vue")` ou bien l'instruction `print("À l'eau")`.

Dans bien des cas, comme dans cet exemple, on ne veut pas simplement choisir entre deux instructions mais davantage : ici afficher `Coulé`, `En vue` ou `À l'eau`. Une possibilité est d'utiliser deux tests *imbriqués*. Une autre est d'utiliser une construction spéciale : `elif`. Sa structure est la suivante :

```
if E1:
    I1
elif E2:
    I2
else:
    I3
```

Si l'expression `E1` est évaluée vraie, on exécute l'instruction `I1` uniquement. Sinon, on évalue l'expression `E2`, si elle est vraie, on exécute l'instruction `I2` uniquement. Dans tous les autres cas (`E1` et `E2` sont évaluées fausses), et on exécute uniquement l'instruction `I3`.

Exercice

Exercice 5

Réécrire le programme de la bataille navale sans utiliser de conditionnelles imbriquées mais à l'aide du `elif`.

Exercice

Exercice 6

Le programme suivant permet de calculer le prix toutes taxes comprises d'un article, connaissant son prix hors taxes, dans le cas où le taux de TVA est de 19,6 %.

```
print("Quel est le prix hors taxes ?")
ht = float(input())
ttc = ht + ht * 19.6 / 100.0
print("Le prix toutes taxes comprises est ", ttc)
```

Adapter ce programme pour permettre à l'utilisateur de choisir le taux de TVA.

Exercice

Exercice 7

En général, à la bataille navale, un bateau n'est « en vue » que si la case touchée est immédiatement voisine de celle du bateau. Modifier le premier programme de ce chapitre pour tenir compte de cette règle. On pourra traiter le cas où les cases diagonalement adjacentes au bateau sont « en vue » et le cas où elles ne le sont pas.

II Les variables et les expressions

II.1 Une variable, qu'est-ce que c'est ?

Dans notre programme précédent, nous avons utilisé les lettres `a` et `b` pour définir l'emplacement du bateau. Ainsi, la boite annotée `a` recevait la valeur `4` et la boite annotée `b` recevait la valeur `7`. En programmation, stocker une valeur dans une boite revient à définir une *variable* et lui associer une valeur. Une variable est donc un espace mémoire de l'ordinateur, repéré par un identifiant unique.

```
a = 4
b = 7
```

`a` et `b` sont les noms de variable, qui sont maintenant interprété par le code. Lorsque `a` sera appelé, le programme regardera à l'intérieur de la boite pour y trouver la valeur `4`.

Attention

Le code alpha-numérique à gauche du symbole « = » définit le nom de la variable. L'utilisation du symbole « = » en Python est très différente de celle que l'on a en mathématique. En effet, en mathématique, l'expression « $x = 5$ » est une proposition qui est soit vraie, soit fausse (en fonction de la valeur de x). En Python, c'est une instruction qui demande que la variable `x` contienne la valeur `5`.

La valeur stockée dans une variable peut être modifiée à tout moment. Il suffit pour cela de réécrire une affectation.

```
a = 4
a = 5
print(a)
```

L'affectation de la valeur `4` dans la variable `a` a été écrasée lors de la seconde affectation en ligne 2. En effet, il ne peut pas exister deux variables avec le même nom.

II.2 Opérations sur les variables

Les variables `a` et `b` que nous avons utilisés précédemment vont nous permettre d'effectuer des opérations plus complexe. Ces opérations sont définies par une expression, qui peut faire intervenir des variables et des

opérateurs (+, -, /, *, etc.). Voici un petit programme qui permet de définir l'âge moyen de trois personnes d'après les informations saisies par l'utilisateur.

```
print("Entrez les âges les uns après les autres")
age1 = int(input())
age2 = int(input())
age3 = int(input())
print("La moyenne d'âge est de ", int((age1+age2+age3)/3))
```

Les lignes 2, 3 et 4 permettent de laisser l'utilisateur renseigner les valeurs à associer aux variables. Ensuite, la ligne 5 calcul et affiche le résultat.

Exercice 8

Modifier le programme pour qu'il effectue la somme de quatre âges.

III Les instructions

Une instruction est une opération du langage de programmation (ou du processeur) que le programmeur demande à la machine d'exécuter. Elle se compose toujours de deux éléments : l'action à effectuer et les éléments sur lesquels l'effectuer.

```
c = a + b
```

L'instruction définit ici la création d'une variable `c` qui sera le résultat de l'addition des valeurs des variables `a` et `b`. De même, la ligne suivante est une instruction

```
print("Python, c'est vraiment top")
```

On demande dans ce cas là d'afficher sur l'écran la phrase « Python, c'est vraiment top ». Voici quelques-unes des instructions standards :

- La saisie via le clavier avec la fonction `input()`
- L'affectation avec le signe « = »
- Les conditions, avec les instructions `if`, `elif`, `else`.

Cette liste n'est pas exhaustive, et sera complétée au fur et à mesure de l'année.

IV La structure conditionnelle

Lorsqu'il est nécessaire de vérifier un élément, telle que la valeur rentrée par l'utilisateur ou encore le résultat d'une opération, on réalise une structure conditionnelle. Dans l'exemple de la bataille navale, l'utilisateur renseigne les coordonnées de son tir de canon en précisant la valeur verticale et horizontale. Il est nécessaire de comparer la position du bateau avec la localisation du tir. Pour cela, on réalise un test logique, en comparant dans l'instruction `if` les différentes valeurs.

Exercice 9

Reprendre le programme précédemment modifié et y ajouter un second bateau (par exemple `c = 2` et `d = 5`). Modifier la structure conditionnelle pour que le test prenne en compte les deux jeux de données.

Exercice 10

Modifier maintenant votre programme afin que l'on ne teste plus l'égalité (==) mais la différence (!=).

À faire : point sur l'indentation en Python.

Exercices de fin de chapitre

Exercice

Exercice 11

Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
ma_variable = 3
ma_variable = 4
ma_variable = ma_variable + 2
print(ma_variable)
```

Exercice

Exercice 12

Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
a = 2
b = a * a
b = a * b
b = b * b
print(b)
```

Exercice

Exercice 13

Que se passe-t-il quand on exécute le programme suivant ?

```
nombre = input("saisir un nombre : ")
print("le nombre suivant est", nombre + 1)
```

Exercice

Exercice 14

Que fait la séquence d'instructions suivante ? On supposera qu'à l'origine, les variables `a` et `b` contiennent chacune une valeur.

```
tmp = a
a = b
b = tmp
```

Exercice

Exercice 15

On met deux entiers dans deux variables `a` et `b`, par exemple, 55 et 89. On remplace le contenu de `a` par la somme de `a` et `b`. Puis on remplace le contenu de `b` par la différence `a` moins `b`. Enfin, on remplace le contenu de `a` par la différence `a` moins `b`.
Que contiennent les deux variables à la fin de cette série d'opérations ? Programmer cet algorithme en Python. Quel avantage voyez-vous par rapport à la méthode de l'exercice précédent ?

Exercice

Exercice 16

Écrire un programme qui demande à l'utilisateur de rentrer un nombre de secondes et qui l'affiche au format heures :minutes :secondes.

Exercice

Exercice 17

Modifier le programme précédent pour qu'il n'affiche pas un nombre d'heure nul mais utilise plutôt le format minutes :secondes.

Chapitre 3

Les types

Dans la section II du chapitre II, nous avons vu ce qu'est une variable dans un langage de programmation : une zone dans la mémoire, que l'on symbolise par une boîte, à laquelle on donne un nom. Les variables peuvent contenir des objets de différentes natures : des nombres entiers, des nombres décimaux, du texte, etc. La nature de l'information qu'elle contient est appelé le *type* de la variable.

En Python, le type d'une variable n'est pas connu avant l'exécution du programme. Pire encore, il peut changer au cours du temps ! On dit que le typage est *dynamique* en Python [1].
Pour connaître le type d'une variable, on utilise la fonction `type` de Python.

```
x = 12
print(type(x))
x = -0.4
print(type(x))
x = 'python'
print(type(x))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Dans ce chapitre, nous étudierons les types de base, que l'on manipule souvent en Python, les chaînes de caractères, qui permettent de représenter du texte et enfin les tableaux qui sont utiles pour contenir plusieurs variables.

I Quelques types de base

I.1 Présentation des types, type d'une expression

En Python, on compte 4 types de base [2] :

- `int` : les nombres entiers,
- `float` : les nombres flottants,
- `bool` : les booléens,
- `str` : les chaînes de caractères (pour « string » en anglais) que nous étudierons à la section II.

1. Beaucoup de langages demandent que le type de chaque variable soit connu et fixé pendant l'exécution du programme, on parle de typage *statique*. Par exemple, le C, le C++, Java, etc. sont des langages statiquement typés. Ce typage statique permet en général au compilateur d'accélérer considérablement l'exécution des programmes grâce à des optimisations déduites de la connaissance du type de chaque variable. Python étant dynamiquement typé, les programmes écrits dans ce langage sont considérés comme plutôt lents.

2. En fait, il existe plein d'autres types natif en Python, comme celui des nombres complexes (que vous étudierez peut-être en mathématiques).

Astuce

Pour connaître le type d'une variable ou de n'importe quelle entité en Python, il suffit d'appeler la fonction `type` sur cette variable.

Ainsi, `type(12)` retourne `int` alors que `type(12.0)` retourne `float`.

python

En Python, tout a un type, même une fonction, la valeur nulle `None`, ou un nom de type lui-même !

```
type(print)  # print est une fonction du langage (built-in en anglais)
```

```
builtin_function_or_method
```

```
type(None)    # None est de type NoneType !
```

```
NoneType
```

```
type(int)     # int est un type !
```

```
type
```

I.1.1 Les types numériques, `int` et `float`

Le type `int`. Le type `int` (pour « integer » en anglais) permet de représenter des entiers relatifs de taille arbitraire, seule la mémoire vive de l'ordinateur est une limite. Par exemple, on peut faire

```
7 ** 101
```

```
22641335567373305939412534383701517676000422392332377806537267319741840217458496420007
```

Le type `float`. Le type `float` permet de représenter certains nombres décimaux, de $-1{,}8 \times 10^{308}$ à $1{,}8 \times 10^{308}$. En réalité, beaucoup de nombre décimaux ne sont pas représentables par des nombres flottants. Par exemple, la suite d'instructions suivante donne un résultat surprenant.

```
0.1 + 0.2
```

```
0.30000000000000004
```

Les expressions numériques. Il n'est pas toujours évident de déterminer le type d'une expression numérique. Par exemple

```
mon_calcul = (37 + 0.51) // (18 / 9)
type(mon_calcul)
```

```
float
```

Exercice

Exercice 18

Déterminer le type de la variable `x` après l'affectation `x = (45 / 12) * (13 // 3)`.

I.1.2 Le type des booléens, `bool`

Le type de certaines expression n'est pas numérique car elles expriment une condition qui est soit vraie, soit fausse. Par exemple, `type(1 == 2)` retourne `bool`. C'est comme cela que Python nomme les « booléens », ces variables qui ne peuvent être que dans deux états, `True` pour vrai ou `False` pour faux.

```
w = 5
w * 2 == 10
```

```
True
```

```
2 ** 10 <= 10 ** 3
```

```
False
```


Info

Les booléens servent souvent à écrire les tests à effectuer dans les structures conditionnelles. Dans ce cas, inutile d'écrire `if z < 0 == True: ...`, on écrit directement `if z < 0: ...`.


Astuce

On peut même tester si une expression est d'un certain type ! Par exemple, `type(10 ** -3 * 2 ** 10) == float` retourne `True`.


Exercice

Exercice 19

Quel est le type de l'expression « `type(10 ** 3 < 2 ** 10) == float` » ?

I.2 Conversion entre types

En Python, il est possible de convertir un type en autre. Pour cela, on utilise le nom du type comme une fonction que l'on appelle sur l'expression à convertir, comme ceci,

```
int(12.1)
```

```
12
```

ou encore,

```
float("56.8")
```

```
56.8
```


Info

Nous avons déjà rencontré ce genre de conversion après l'utilisation de la fonction `input`. En effet, le type de retour de `input` est `str`. Si l'on veut que l'utilisateur saisisse un entier, il faut convertir cette chaîne de caractère à l'aide de la fonction `int` même si ceci peut causer des erreurs.


Attention

Certaines conversions sont toujours possibles, d'autres peuvent générer des erreurs. Le tableau ci-dessous résume ceci.

↱	`int`	`float`	`bool`	`str`
`int`	OK	OK	OK	OK
`float`	OK	OK	OK	OK
`bool`	OK	OK	OK	OK
`str`	⚠	⚠	OK	OK

II Les chaînes de caractères

Nous avons déjà rencontré les chaînes de caractères de Python. Elles servent à représenter une suite de caractères, on les place entre guillemets, doubles « " » ou simples « ' ».

```
texte = "Ceci est un texte qui illustre la définition d'une chaîne de caractères."
type(texte)
```

```
str
```

Il est possible de demander a Python le nombre de caractères qui constitue une chaîne, autrement dit sa longueur. Ceci se fait avec la fonction `len`.

```
len(texte)
```

```
72
```

On peut aussi aller chercher un caractère en particulier dans la chaîne ! Pour cela, on utilise la notation `[...]`.

```
objet = "écharpe"
objet[5]
```

```
'p'
```

Lorsque l'on utilise la notation `[...]`, la numérotation commence à zéro. Le premier caractère de la chaîne `ma_chaine` est donc `ma_chaine[0]`.

On peut mettre bout-à-bout deux chaînes de caractères, on parle de *concaténation*. Pour cela, on utilise le symbole `+`.

```
"Je fais " + "des phrases"
```

```
'Je fais des phrases'
```

En toute logique, si l'on sait ajouter, on sait multiplier donc on devrait pouvoir utiliser le symbole `*`. C'est bien le cas.

```
"ah" * 3
```

```
'ahahah'
```

Désormais, confiez vos punitions à Python !

```
punition = "À l'avenir, je respecterai le règlement intérieur.\n"
print(20 * punition)
```

```
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
```

```
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
À l'avenir, je respecterai le règlement intérieur.
```

Info

Le caractère `'\n'` est spécial, il permet d'indiquer un saut de ligne dans une chaîne de caractère. Il existe beaucoup d'autres caractères spéciaux.

Attention

On ne peut pas modifier un caractère dans une chaîne.

```
objet = "écharpe"
objet[5] = 'd'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-499a00239096> in <module>()
      1 objet = "écharpe"
----> 2 objet[5] = 'd'

TypeError: 'str' object does not support item assignment
```

Comparaison de chaînes de caractères. Il est possible d'utiliser les opérateurs de comparaison ==, !=, <, >=, etc. sur les chaînes de caractères. Ainsi, `'a' < 'b'` retourne `True`.

Découpes à la chaîne. Il est possible de découper une chaîne de caractères en plusieurs sous-chaînes, préalablement délimitées par un même séparateur. Pour cela, on utilise la méthode de `split`, de la manière suivante.

```
ma_chaine = "A;B;C;D;E;F;G;H"
ma_chaine.split(';')
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

Le résultat ainsi obtenu est un tableau dont chaque élément est une chaîne de caractère. Nous verrons dans la section suivante ce qu'est un tableau.

Pour en savoir plus, consulter la page suivante : `https://docs.python.org/fr/3/library/stdtypes.html#string-methods`

III Les tableaux

Un tableau permet de stocker plusieurs valeurs dans une seule variable. Les tableaux sont assez similaires aux chaînes de caractères sauf qu'ils permettent de rassembler des données plus variées que seulement des caractères. Pour créer un tableau, on utilise les crochet et l'on sépare les éléments par des virgules.

```
mon_premier_tableau = [5, 4, 3, 2 ,1]
type(mon_premier_tableau)
```

```
list
```

On découvre donc que Python appelle ses tableaux des listes. Nous découvrirons pourquoi en classe de terminale.

On peut également accéder à la taille d'un tableau ou encore à ses éléments.

```
len(mon_premier_tableau)
```

```
5
```

```
mon_premier_tableau[1]
```

```
4
```


Attention

Pour les tableaux aussi, la numérotation commence à zéro. D'autre part, on ne peut pas accéder au delà de la fin du tableau.

```
mon_premier_tableau[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-6-81e9a1e5c369> in <module>()
----> 1 mon_premier_tableau[5]

IndexError: list index out of range
```

Contrairement aux chaînes de caractères, il est possible de modifier les éléments d'un tableau.

```
mon_premier_tableau[1] = 0
mon_premier_tableau
```

```
[5, 0, 3, 2, 1]
```


Astuce

Comme avec les chaînes de caractère, on peut concaténer deux tableau avec le symbole `+`.

```
['a', 'b', 'c', 'd'] + ['x', 'y', 'z']
```

```
['a', 'b', 'c', 'd', 'x', 'y', 'z']
```

Enfin, on peut créer de gros tableaux avec le symbole `*`.

```
[False] * 50
```

```
[False, False, False, False, False, False, False, False, False, False]
```


python

En Python, il est possible d'accéder directement au dernier élément d'un tableau. Pour cela, on imagine le tableau comme un ruban fermé sur lui-même de sorte que l'élément qui précède le premier est le dernier. On utilise dont la notation `...[-1]`.

```
tab = [1.2, 12.5, 57.9, 37.2, 8.0]
tab[-1]
```

```
8.0
```


python

À l'inverse de la méthode `split` sur les chaînes de caractères, il est possible d'assembler une liste en une chaîne de caractères séparé par un motif unique. Pour cela, on utilise la méthode `join`.

```
tab = ["c'est", "à", "dire"]
'-'.join(tab)
```

```
"c'est-à-dire"
```

Exercices de fin de chapitre

Exercice

Exercice 20

Que retourne l'instruction suivante : `len(2 * ([3, 4, 6] + 5 * [6, 7]))` ?

Exercice

Exercice 21 (tiré du sujet 0 de NSI)

Avec la variable `alphabet` définie par l'affectation suivante, quelle est l'expression Python permettant d'accéder à la lettre `'E'` ?

```
alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
            'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

(a) `alphabet[4]`
(b) `alphabet[5]`
(c) `alphabet['E']`
(d) `alphabet.E`

Exercice

Exercice 22

On souhaite qu'un utilisateur saisisse une chaîne de caractère au clavier et lui afficher un message dans le cas où cette chaîne est vide.

1) Citer au moins deux manières possibles pour tester si une chaîne de caractères est vide.
2) Écrire le programme demandé.

Exercice

Exercice 23

Nous avons vu qu'en Python, les tableaux ont un type particulier, appelé `list`. On étudie les différentes conversions de type possibles. Dans chacun des cas, bien identifier de comportement de chaque conversion.

1) Est-il possible de convertir un entier en une liste ? Et l'inverse ?
2) Est-il possible de convertir un booléen en une liste ? Et l'inverse ?
3) Est-il possible de convertir une chaîne de caractère en une liste ? Et l'inverse ?

Exercice

Exercice 24

À l'aide des méthodes `split` et `join`, écrire un programme qui demande à l'utilisateur d'écrire une chaîne de caractère contenant une suite de prénoms séparés par des virgules et affiche la même chaîne de caractères séparée par des points-virgule.
Il existe en fait une méthode pour remplacer toutes les sous-chaînes d'une chaîne de caractère par une autre sous-chaîne. C'est la méthode `replace`. Consulter la page `https://docs.python.org/fr/3/library/stdtypes.html#string-methods` et réécrire le programme précédent avec la méthode `replace`.

Chapitre 4

Les boucles

Il est très fréquent que certaines parties d'un algorithme soient répétitives. Pour les implémenter dans un langage de programmation, on a besoin de structures spécifiques : les structures de boucles.

I La boucle `for`

La boucle `for`[1], aussi appelée *boucle bornée* est utilisée lorsque le nombre d'itérations est connu à l'avance. On l'utilise pour parcourir une structure existante, comme un tableau ou une chaîne de caractères, ou encore avec un indice qui compte le nombre de tours de boucles (on parle d'itérations).

I.1 Parcours d'un tableau ou d'une chaîne de caractères

On peut facilement parcourir une chaîne de caractère avec une boucle `for`. Ceci se réalise de la manière suivante.

```
for c in "bonjour":
    print(c)
```

```
b
o
n
j
o
u
r
```

De la même manière, on parcours les éléments d'un tableau.

```
for x in [6, 8, 9, 5, -12]:
    print(x)
```

```
6
8
9
5
-12
```

Ceci est particulièrement intéressant lorsqu'il s'agit de faire un traitement sur les éléments d'un tableau. Par exemple, si l'on souhaite calculer la somme des nombres contenus dans un tableau, on pourra s'y prendre ainsi.

1. « for » signifie « pour » en anglais.

```
tab = [6, 8, 9, 5, -12]
somme = 0

for x in tab:
    somme = somme + x

print("La somme des éléments de", tab, "vaut", somme, ".")
```

```
La somme des éléments de [6, 8, 9, 5, -12] vaut 16 .
```

Attention

Encore une fois, l'indentation aura une importance capitale dans une boucle `for`. Reprenons le programme précédent et supposons que l'on souhaite afficher `somme` à chaque tour de boucle. On peut alors procéder ainsi

```
for x in tab:
    somme = somme + x
    print(somme)
```

Alors que si l'on écrit ceci,

```
for x in tab:
    somme = somme + x
print(somme)
```

`somme` ne sera affichée qu'une fois la boucle terminée.

Astuce

Il est possible d'utiliser une structure conditionnelle dans une boucle `for`. On dit alors qu'on *imbrique* un `if` dans un `for`. L'indentation est encore primordiale dans ce cas. Voici un exemple.

```
for c in "zoubidou":
    if c == 'o':
        print("J'ai trouvé un 'o' !")
```

```
J'ai trouvé un 'o' !
J'ai trouvé un 'o' !
```

Exercice

Exercice 25

Écrire un programme Python qui demande une phrase à l'utilisateur et retourne le nombre de caractères `'e'` qu'elle contient.

I.2 Itération sur les entiers

Une autre manière d'utiliser la boucle `for` est de parcourir tous les entiers d'un intervalle. Pour cela, on utilise la fonction `range(a, b)`, comme ceci.

```
for i in range(4, 13):
    print(i)
```

```
4
5
6
7
8
```

```
9
10
11
12
```

`range` signifie « intervalle » en anglais. `range(a, b)` peut être vu comme un tableau contenant tous les entiers de l'intervalle $[a\,;b[$.

Attention

Dans `range(a, b)`, la valeur `b` est exclue ! Si l'on veut l'inclure, il faut alors écrire `range(a, b + 1)`.

Astuce

Si l'on souhaite commencer à `0`, au lieu d'écrire `range(0, n)`, on peut écrire plus simplement `range(n)`.

Astuce

Il est possible de parcourir les entiers de deux en deux ou de trois en trois voir même de -1 en -1, c'est à dire à l'envers ! Pour cela, on peut passer la valeur en troisième paramètre de la fonction `range`, comme ceci.

```
for k in range(1, 10, 2):
    print(k)
```

```
1
3
5
7
9
```

```
for k in range(5, 0, -1):
    print(k)
```

```
5
4
3
2
1
```

Info

Dans une instruction comme `for i in range(...)`, on dit que la variable `i` est *l'indice de la boucle*.

Exercice

Exercice 26

Écrire un programme qui demande à l'utilisateur un entier `n` supérieur à 1 et qui affiche la somme des entiers de 1 à `n`.

Exercice

Exercice 27

Modifier le programme précédent pour qu'il affiche non plus la somme mais le produit.

Info

Il est tout à fait possible de ne pas utiliser l'indice de la boucle, mais il convient tout de même de lui donner un nom.

```
for k in range(6):
    print("C'est moi ou je me répète ?")
```

```
C'est moi ou je me répète ?
C'est moi ou je me répète ?
C'est moi ou je me répète ?
C'est moi ou je me répète ?
C'est moi ou je me répète ?
C'est moi ou je me répète ?
```

II La boucle `while`

La boucle `while`[2], aussi appelée *boucle non bornée* est utilisée lorsque le nombre d'itérations n'est pas connu à l'avance. On l'utilise souvent lorsque le critère d'arrêt de la boucle n'est pas connu à l'avance mais dépend justement de l'exécution de la boucle.

II.1 S'arrêter, oui, mais à une condition !

La boucle `while` utilise une expression booléenne comme critère d'arrêt. Ceci signifie qu'à chaque tour de boucle, on va tester cette condition, si elle est vraie, on repart pour au moins un tour de boucle, si elle est fausse, on sort de la boucle.

Par exemple, pour trouver la première puissance de 2 supérieur ou égale à 1000, on a écrit le programme suivant.

```
puissance = 1
while puissance < 1000:
    puissance = puissance * 2
    print("Valeur intermédiaire de puissance :", puissance)
print("Valeur finale de puissance :", puissance)
```

```
Valeur intermédiaire de puissance : 2
Valeur intermédiaire de puissance : 4
Valeur intermédiaire de puissance : 8
Valeur intermédiaire de puissance : 16
Valeur intermédiaire de puissance : 32
Valeur intermédiaire de puissance : 64
Valeur intermédiaire de puissance : 128
Valeur intermédiaire de puissance : 256
Valeur intermédiaire de puissance : 512
Valeur intermédiaire de puissance : 1024
Valeur finale de puissance : 1024
```

Dans cet exemple, on remarque que, comme pour la boucle `for`, l'indentation est primordiale ! Les deux dernière lignes n'ont pas la même indentation : la première est décalée, elle est exécutée à chaque tour de boucle alors que la seconde n'est exécutée qu'une fois la boucle terminée.

Ici, le critère d'arrêt de la boucle est l'expression « `puissance < 1000` ». À chaque fin de boucle, cette expression est évaluée : tant qu'elle est vraie, on ré-exécute le corps de la boucle.

Exercice 28

Modifier le programme précédent pour trouver le premier entier naturel n tel que la somme des entiers de 0 à n est supérieure à 5000.

2. « while » signifie « tant que » en anglais.

Savoir-faire

Choisir entre une boucle `for` et une boucle `while`

Si on connaît à l'avance le nombre de répétitions à effectuer, ou que l'on veut parcourir une structure dont le nombre d'éléments est fixe (un tableau ou une chaîne de caractères par exemple), c'est la boucle **`for`** qu'il faut choisir.
En revanche, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle **`while`** qui convient.

Exercice

Exercice 29

Quelle boucle est adaptée à l'écriture de programmes traitant les problèmes suivants :

- le calcul du total à payer à une caisse enregistreuse,
- la recherche du jour le plus pluvieux d'une année,
- le calcul du périmètre d'un polygone,
- le calcul de la durée d'une émission de radio, connaissant ses horaires de début et de fin ?

Info

Un langage de programmation qui ne permettrait que d'écrire :

- des expressions,
- des affectations de variables,
- des séquences d'instructions,
- des structures conditionnelles,
- des boucle non bornées (**`while`**)

est dit *complet*. Ceci signifie que tous les programmes que l'on peut imaginer peuvent être écrits dans ce langage. Autrement dit, tout ce qui n'est pas dans ce langage (comme la boucle **`for`** ou les fonctions que nous étudierons au chapitre suivant) n'est qu'une syntaxes pour faciliter l'écriture et la lecture des programmes [3].
De même que tous les objets qui nous entourent sont formés de trois types de particules : les protons, les neutrons et les électrons, que tous les textes que nous lisons sont formés de vingt-six lettres et de quelques signes de ponctuation, que toutes les musiques que nous entendons peuvent être exprimées à l'aide de douze notes, tous les programmes que nous utilisons peuvent ultimement être exprimés avec ces quatre instructions : l'affectation, la séquence, le test et la boucle.

Astuce

Parfois, il n'est pas possible d'exprimer la condition pour rester dans la boucle de manière simple et il est plus pratique (ou plus lisible ou plus efficace) d'utiliser une autre condition dans le corps de la boucle. Pour cela, on utilise l'instruction **`break`**, souvent dans un **`if`**.
Par exemple, dans l'exemple suivant, on demande à l'utilisateur le prix des articles, tant que leur somme n'a pas atteint les 100 euros. On souhaite aussi laisser la possibilité à l'utilisateur d'intérrompre à tout moment avec la touche `'q'`.

```
somme = 0

while somme < 100:
    reponse = input("Entrer le prix suivant ou 'q' pour quitter :")
    if reponse == 'q':
        break
    else:
        somme = somme + float(reponse)

print("La somme atteinte est ", somme)
```

3. On parle de « sucres syntaxiques ».

```
Entrer le prix de l'article suivant ou 'q' pour quitter : 10
Entrer le prix de l'article suivant ou 'q' pour quitter : 11
Entrer le prix de l'article suivant ou 'q' pour quitter : 23.90
Entrer le prix de l'article suivant ou 'q' pour quitter : q
La somme atteinte est  44.9
```

Astuce dans l'astuce : l'instruction **break** peut aussi être utilisée dans une boucle **for**.

II.2 La boucle `for`, cas particulier de la boucle `while`

Il est toujours possible de remplacer une boucle **for** par une boucle **while**. En effet, par exemple, la séquence

```
for i in range(n):
    {une séquence d'instructions}
```

peut être remplacée par [4]

```
i = 0
while i < n:
    {une séquence d'instructions}
    i = i + 1
```

Astuce

Avec les boucles **while**, on a souvent besoin d'ajouter des valeurs à des variables. Ceci peut se faire de manière compacte avec le symbole « `+=` ». En effet, l'instruction « `x = x + 1` » s'écrit aussi « `x += 1` ». On dispose également des instructions « `-=` », « `*=` » et « `/=` ».

Exercice

Exercice 30

Écrire un programme qui affiche tous les nombres impairs de 0 à 100, l'un avec une boucle **for**, l'autre avec une boucle **while**.

II.3 Attention aux boucles infinies !

Avec une boucle **while**, il est possible que le programme ne s'arrête jamais, on parle de *non-terminaison*. Par exemple, la boucle

```
while i != 0:
    i = i - 1
```

termine seulement si la variable `i` est positive ou nul avant d'entrer dans la boucle. Il est parfois utile de vouloir créer une boucle infinie. Pour cela, on utilise l'instruction « **while True:** ». On sort alors de la boucle avec l'instruction **break**.

Info

On peut aussi créer une boucle infinie qui ne fait rien avec l'instruction suivante.

```
while True:
    pass
```

Exercice

Exercice 31

La boucle ci-dessous se termine-t-elle ?

4. Attention tout de même à la différence de valeur de `i` à la sortie des deux boucles. `i` vaut `n - 1` après la boucle **for** et `n` après le **while**.

```
valeur = 0
while valeur != 17:
    valeur += 2
```

Exercices de fin de chapitre

Exercice

Exercice 32 (tiré du sujet 0 de NSI)

Quelle est la valeur affichée à l'exécution du programme Python suivant ?

```
x = 1
for i in range(10):
    x = x * 2
print(x)
```

(a) 2
(b) 1024
(c) 20000000000
(d) 2048

Exercice

Exercice 33 (tiré du sujet 0 de NSI)

Que peut-on dire du programme Python suivant de calcul sur les nombres flottants ?

```
x = 1.0
while x != 0.0:
    x = x - 0.1
```

(a) L'exécution peut ne pas s'arrêter, si la variable `x` n'est jamais égale à `0.0`.
(b) À la fin de l'exécution, la variable `x` vaut `-0.000001`.
(c) À la fin de l'exécution, la variable `x` vaut `0.000001`.
(d) L'exécution s'arrête sur une erreur `FloatingPointError`.

Exercice

Exercice 34

Reprendre le programme de la bataille navale puis le modifier pour qu'il permettre à l'utilisateur de jouer tant qu'il n'a pas gagné.

Exercice

Exercice 35 Turtle 1

Après avoir lu la section II du chapitre 6 qui concerne l'introduction au module graphique Turtle, écrire un programme qui demande à l'utilisateur un nombre de côté et une longueur et affiche le polygone régulier correspondant.
Avec quels paramètres a-t-on l'impression d'avoir tracé un cercle ?

Exercice

Exercice 36 Méthode de Héron

En mathématiques, la méthode de Héron permet d'approcher la valeur d'une racine carrée. À chaque étape, on calcule une nouvelle valeur, plus proche que la précédente du résultat souhaité. Par exemple, dans le cas de l'approximation de $\sqrt{2}$, si l'on a déjà une valeur approchée, x, la valeur suivante, x', se calcule comme ceci :

$$x' = \frac{1}{2}\left(x + \frac{2}{x}\right).$$

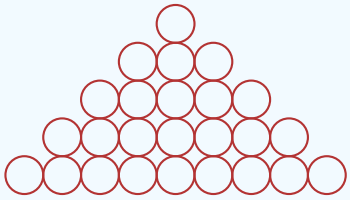
En partant d'une valeur initiale égale à 2, programmer la méthode de Héron pour le calcul de la valeur approchée $\sqrt{2}$. Le programme s'arrêtera lorsque le carré de la valeur trouvée est proche

de 2 à 10^{-9} près.

Exercice

Exercice 37 Turtle 2

Après avoir lu la section II du chapitre 6 qui concerne l'introduction au module graphique Turtle, reproduire la forme ci-dessous.



Chapitre 5

Les fonctions

I Structurer son code avec des fonctions : diviser pour régner

Les *fonctions* sont des fragments de code réutilisables réalisant une tâche donnée pouvant dépendre d'un certain nombre de paramètres. Un fonction est définie par un *nom*, a généralement des *paramètres d'entrée* et retourne *une ou plusieurs valeurs*. Elles deviennent très utiles lorsqu'une action est réalisé à plusieurs endroit d'un programme.

I.1 Définir une fonction

Une fonction associe une séquence d'instruction à un nom. Voici l'exemple d'une fonction qui permet de dessiner un angle droit de longueur 100 à l'aide de l'outil turtle (pour plus de précisions sur l'outil turtle, se référer à la description du module plus bas dans le document).

```
def angle_droit():
        forward(100)
        left(90)
        forward(100)
```

La définition de la fonction commence par le mot-clé *def*, suivi par le nom de la fonction *angle_droit*, puis une paire de parenthèse et un deux-points. Tous les éléments indentés (avec une tabluation) appartiennent à la fonction, c'est le *corps de la fonction*. Par la suite, cette fonction sera appelé dans le code principal par la syntaxe **angle_droit()**. L'appel de cette fonction produit le même effet que si l'on avait exécuté les trois instructions présentes dans la fonction **angle_droit()**.

I.2 Fonction avec paramètres

Dans notre fonction *angle_droit*, la longueur est fixée à 100. Pour dessiner des angles droits de longueur quelconque, on peut ajouter un *paramètre* à notre fonction.

```
def angle_droit(x):
        forward(x)
        left(90)
        forward(x)
```

Le paramètre est désigné par un nom, ici **x**, ajouté entre les parenthèses. Il est ensuite utilisé dans le corps de la fonction, comme une variable. Le paramètre désigne une valeur qui pourra être différente à chaque appel à la fonction *angle_droit*. Il est possible de renseigner plusieurs paramètres à la définition de notre fonction. Dans ce cas, chaque paramètre est séparé d'une *virgule*, et toujours renseigné à l'intérieur des parenthèses. Par exemple, précisons l'angle que réalisera la fonction.

```
def angle_droit(x, angle):
        forward(x)
        left(angle)
        forward(x)
```

Lors de l'appel de la fonction, on renseignera autant de paramètre que la fonction en nécessite, dans le cas actuel, pour une longueur de 20 et un angle de 60 :

```
angle_droit(20, 60)
```

Toutes les instructions que nous avons utilisés jusqu'à présent, comme **print()** ou encore **range()** sont des fonctions natives et prédéfinies par le langage.

Exercice 38

A l'aide de la fonction angle_droit(), dessiner un escalier de 10 marches. Modifier ensuite votre programme pour que l'utilisateur puisse renseigner le nombre de marche et leur longueur.

Exercice 39

Écrire une fonction qui permet de dessiner un rectangle dont les longueurs seront renseigner par l'utilisateur. Modifier ensuite votre programme afin de demander à l'utilisateur si il souhaite que le rectangle soit colorié à l'intérieur et l'exécuter. Le remplissage de la forme géométrique sera réalisé à l'appel de la fonction *begin_fill()*, et ce jusqu'à l'appel de la fonction *end_fill()*.

I.3 Renvoyer un résultat

Dans la partie précédente, notre fonction angle_droit() permettait d'obtenir un résultat sous forme d'affichage graphique via la bibliothèque turtle. Néanmoins, les fonction Python peuvent aussi représenter des fonctions mathématiques qui calculent des valeurs. Par exemple, le code :

```
def f(x):
        return 2*x + 4
```

définit la fonction mathématique qui associé au nombre x, associe le nombre $2x+4$. Le mot-clé **return** spécifie la valeur *renvoyée* comme le résultat de la fonction.

```
print("Lorsque 'x' vaut 5, f(x) = ", f(5))
```

```
Lorsque "x" vaut 5, f(x) = 14
```

Il ne faut pas hésiter à donner à la fonction un nom bien explicite, même si il est un peu long. Le jours de la définition de la fonction, un nom simple peu sembler suffisant, mais c'est rarement le cas plusieurs jours ou semaines plus tard. Ainsi, une fonction qui définit les caractéristiques principales d'un personnage secondaire devrait s'appeler *definition_principale_personnage_secondaire* plutôt que *def_car_sec*. On peut aussi utiliser la syntaxe **camelCase** comme ceci : *definitionPrincipalePersonnageSecondaire*.

Exercice 40

Écrire une fonction qui permet d'appliquer une TVA à 19,5% à un montant renseigné par l'utilisateur et l'afficher.

Exercice 41

Écrire une fonction qui divise n fois par x un nombre renseigné par l'utilisateur et l'afficher.

II Variable globale, variable locale, portée des variables

D'après l'emplacement de la définition, une variable peu être *locale* ou *globale*. Une variable locale ne sera utilisé que dans la fonction où elle est définie, et supprimée lorsque l'on sort de la fonction. Une variable globale est définie dans le programme principale et est accessible à tout moment par l'ensemble du programme (et des fonctions donc).

```
Def addition_de_deux_entiers(x, y):
        return x + y

A = 10
B = 15
print (addition_de_deux_entiers(A, B))
```

```
25
```

Dans l'exemple ci-dessus, A et B sont définies comme des variables globales, puisqu'elles sont situées dans le programme principal. A contrario, les variables x et y sont des variables locales puisqu'elles sont sont définies dans la fonction addition_de_deux_entiers(). Lorsque la fonction addition_de_deux_entiers(A, B) est appelée, la fonction associe A à x et B à y. les deux variables locales x et y sont utilisées pour effectuer l'addition, puis sont détruites.

Au supermarché. On cherche à simuler la distribution de tickets dans une fil d'attente, comme au supermarché.

```
numero_ticket = 0

def prend_ticket():
    numero_ticket += 1
    return numero_ticket
```

```
for i in range(5):
    print(prend_ticket())
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-3-57e7268bb6a5> in <module>()
      1 for i in range(5):
----> 2     print(prend_ticket())

<ipython-input-2-edc590c61d7d> in prend_ticket()
      2
      3 def prend_ticket():
----> 4     numero_ticket += 1
      5     return numero_ticket

UnboundLocalError: local variable 'numero_ticket' referenced before assignment
```

Le problème c'est que Python interprète la variable `numero_ticket` comme une *variable locale* alors qu'on aimerai qu'elle soit *globale*. Pour cela, il faut lui indiquer avec le mot clef `global`.

```
numero_ticket = 0

def prend_ticket():
    global numero_ticket
    numero_ticket += 1
    return numero_ticket
```

```
for i in range(5):
    print(prend_ticket())
```

```
1
2
3
4
5
```

Exercice 42

Écrire une fonction qui tire au hasard un nombre entre 1 et 100 tout en comptant le nombre de fois qu'elle est appelée. On utilisera pour cela la fonction `randint` du module `random` dont la documentation se trouve ici : `https://docs.python.org/3/library/random.html#random.randint`.

III Passage par valeur ou par référence

Intéressons nous à la fonction suivante.

```
def swap(a, b):
    tmp = a
    a = b
    b = tmp
```

Qu'affiche alors le code ci-dessous ?

```
a = 7
b = 12
swap(a, b)
print(a, b)
```

Étonnant non ? En effet, ce ne sont pas les variables `a` et `b` qui sont passées à la fonction `swap` mais simplement leur *valeur*. On parle alors de passage par valeur. Il est impossible de modifier le contenu des variables globales `a` et `b` en les passant à la fonction `swap`[1]. On peut en revanche procéder comme ceci.

```
def swap(a, b):
    return (b, a)
```

```
a = 7
b = 12
a, b = swap(a, b)
print(a, b)
```

```
12 7
```

1. Il faudrait utiliser le mot clef `global`.

Intéressons nous maintenant à la fonction suivante.

```
def modifie_tableau(tab):
    tab[0] = 13
```

Qu'affiche alors le code ci-dessous ?

```
mon_tableau = [1, 2, 3, 4, 5]
modifie_tableau(mon_tableau)
print(mon_tableau)
```

Mais alors, on peut en fait modifier une variable passée en paramètre d'une fonction ?! Si la variable est un tableau, elle n'est en fait pas passée par valeur mais par *référence* à la fonction. C'est à dire que ce n'est pas le tableau qui est passé à la fonction mais, en quelque sorte, son emplacement dans la mémoire. On peut alors accéder au tableau et en modifier les valeurs !

Exercice

Exercice 43

Écrire une fonction `swap` qui prend en paramètre un tableau à deux éléments et les échange. On ne demande pas de retourner le résultat.

Exercice

Exercice 44

Comment peut-on modifier la fonction de l'exercice précédent pour qu'elle vérifie que le tableau passé en paramètre est bien de taille 2.

IV Conditions sur les arguments et la valeur de retour, assertions

Il arrive souvent que les fonctions ne soient valables que pour certaines valeurs des paramètres. Voici par exemple une fonction qui calcule le PGCD de deux entiers.

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Cette fonction n'est valable que pour `a` et `b` des entiers positifs, on parle alors de *préconditions* sur les arguments de la fonction. On souhaiterai que l'utilisateur soit averti de cette limitation. On peut alors ajouter un commentaire au code de la fonction.

```
# Retourne le PGCD de a et b
# Conditions : a et b sont des entiers positifs
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

C'est déjà mieux mais l'utilisateur n'a pas toujours le code d'une fonction sous les yeux. C'est pour cela qu'au lieu d'utiliser des commentaires, on utilise des *docstring* : ce sont des chaînes de caractères délimitées par trois guillemets « """ » et placées juste après les deux points qui définissent la fonction.

```
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être positifs."
    while b != 0:
        a, b = b, a % b
    return a
```

L'utilisateur qui n'a pas le code de la fonction sous les yeux peut alors consulter la documentation de la fonction avec `help(pgcd)`.

```
Help on function pgcd in module __main__:

pgcd(a, b)
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
```

C'est déjà très clair mais on peut faire encore plus robuste ! On voudrait que l'utilisateur reçoive un message d'erreur lorsqu'il utilise mal la fonction. On peut alors utiliser le mot clef **assert**, comme ceci.

```
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être strictement positifs."
    while b != 0:
        a, b = b, a % b
    return a
```

```
pgcd(96, -20)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-e79328fdcd81> in <module>()
----> 1 pgcd(96, -20)

<ipython-input-11-8ed2b06e45dc> in pgcd(a, b)
      2 # Conditions : a et b sont des entiers positifs
      3 def pgcd(a, b):
----> 4     assert a > 0 and b > 0, "a et b doivent être strictement positifs."
      5     while b != 0:
      6         a, b = b, a % b

AssertionError: a et b doivent être strictement positifs.
```

Astuce

On peut aussi utiliser **assert** pour faire des tests. Soit des tests en dehors de la fonction comme ici,

```
assert pgcd(96, 20) == 4, "mauvaise valeur retournée par la fonction pgcd."
```

on peut aussi s'assurer qu'une condition est vraie avant de retourner le résultat de la fonction.

```python
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être strictement positifs."
    while b != 0:
        a, b = b, a % b
    assert b <= a, "erreur interne dans la fonction pgcd."
    return a
```

Exercice

Exercice 45

Modifier la fonction `pgcd` pour qu'elle vérifie que les arguments sont de type `int`.

Exercices de fin de chapitre

Exercice

Exercice 46 tiré du sujet 0 de NSI

Avec la définition de la fonction `f` suivante en Python, quelle est la valeur retournée par l'appel `f(42, 21)` ?

```python
def f(x, y):
    if x > y:
        return y, x
    else:
        return x, y
```

(a) `(21, 21)` (b) `(21, 42)` (c) `(42, 42)` (d) `(42, 21)`

Exercice

Exercice 47 tiré du sujet 0 de NSI

Dans la définition suivante de la fonction `somme` en Python, quelle est l'instruction à ajouter pour que la valeur retournée par l'appel `somme([10, 11, 12, 13, 14])` soit `60` ?

```python
def somme(tab):
    s = 0
    for i in range(len(tab)):
        ...
    return s
```

(a) `s = s + i`
(b) `tab[i] = tab[i] + s`
(c) `s = tab[i]`
(d) `s = s + tab[i]`

Exercice

Exercice 48 tiré du sujet 0 de NSI

Le programme Python suivant ne calcul pas toujours correctement le résultat de x^y. Parmi les tests suivant, lequel va permettre de détecter l'erreur ?

```python
def puissance(x, y):
    p = x
    for i in range(y - 1):
        p = p * x
    return p
```

(a) `puissance(2, 0)` (b) `puissance(2, 1)` (c) `puissance(2, 2)` (d) `puissance(2, 10)`

Exercice

Exercice 49 tiré du sujet 0 de NSI

La fonction suivante calcule la racine carré du double d'un nombre flottant.

```
from math import sqrt

def racine_du_double(x):
    return sqrt(2 * x)
```

Quelle est la précondition sur les arguments de cette fonction ?

(a) `2 * x > 0` (b) `x < 0` (c) `x >= 0` (d) `sqrt(x) >= 0`

Exercice

Exercice 50

Écrire une fonction `maximum` qui retourne la plus grande des deux valeurs qui lui sont passées en paramètres.
Mettre en place un jeu de tests avec des `assert`.

Exercice

Exercice 51

Modifier la fonction de l'exercice précédent pour quelle travaille non plus avec deux valeurs mais avec un tableau.
Mettre aussi en place un jeu de tests avec des `assert`.

Exercice

Exercice 52

Définir une fonction *testPythagore* qui prend 3 arguments entiers a, b, c et retourne un booléen si $a^2 + b^2 = c^2$.

Exercice

Exercice 53

Le cryptage des messages a été utilisé en premier par Jules César pour cacher les messages destinés, entre autre, à ses armées. Il utilisait une technique de décalage de l'alphabet, appelée *clé de cryptage*. Par exemple, avec un clé de 3 et la lettre C, on obtient la lettre F. Écrire une fonction qui permet de réaliser ce cryptage. *Rappel : on passe d'un caractère à un entier avec la fonction ord(), et l'inverse avec la fonction str(chr()).*

Exercice

Exercice 54

Réaliser un programme qui permet d'affecter la clé de cryptage à chaque caractère d'une chaîne de caractère. Pour cela, il sera nécessaire de modifier la fonction de l'exercice précédent afin d'appliquer la clé de cryptage à chaque élément du tableau.

Astuce

La séparation de la chaîne de caractère sera réalisé avec la fonction ci-dessous. Cette fonction associe un caractère à une case de notre tableau.

```
def separation(mot):
        return list(mot)
```

et le ré-assemblage (appelé *concaténation*) sera réalisé avec la fonction suivante. L'élément "" indique de n'associer aucun caractère à la concaténation.

```
def concatenation(mot):
        return "".join(mot)
```

Chapitre 6

Les modules de la librairie standard

Certaines fonctions comme `print`, `input`, `type`, etc. font partie du langage Python. Cependant, il existe beaucoup de fonctions qui ne font pas directement partie du langage mais qui sont disponibles dans des *modules*.

Par exemple, la plupart des fonctions mathématiques sont présentes dans le module `math`, on peut manipuler des dates avec `datetime`, tirer des nombre au hasard avec `random` ou encore dessiner avec le module `turtle`.

I Importer un module

Pour charger les fonctions d'un module, on utilise l'instruction `import`. Par exemple, l'instruction `import math` importe toutes les fonctions du module `math`. Pour y accéder, on utilise le nom du module suivi d'un point, comme ceci : `math.cos(12)`.

Il est possible de charger seulement une fonction d'un module avec l'instruction `from math import cos`. On peut alors utiliser directement la fonction `cos`, sans faire `math.cos`.

Enfin, il est possible d'importer toutes les fonctions d'un module comme ceci : `from math import *` mais cette pratique n'est pas recommandée.


Attention

L'utilisation de l'instruction `from mon_module import *` est dangereuse car si `mon_module` possède une fonction du même nom qu'une que j'ai déjà définie, alors celle-ci sera écrasée par l'import ! C'est aussi le cas lorsqu'on importe deux modules de cette manière : si certaines fonctions partagent le même nom, les fonctions du dernier import écraseront celles du premier.

Plus tard, vous écrirez vos propres modules que vous importerez.

II Le module `turtle`

Le module Turtle permet de dessiner à l'écran. Sa documentation se trouve ici : `https://docs.python.org/fr/3/library/turtle.html`. Pour comprendre le fonctionnement de Turtle, il faut imaginer un robot sous forme de tortue partant de l'origine d'un repère orthonormé et se déplaçant sur l'écran, en dessinant derrière lui.

II.1 Premiers dessins avec Turtle


Exercice

Exercice 55

Saisir le code ci-dessous et observer le résultat. Modifier le programme pour comprendre le fonctionnement de Turtle.

```
from turtle import *
```

```
forward(100)  # on avance de 100 pixels
left(90)      # on tourne à gauche de 90°
forward(50)   # on avance de 50 pixels
right(90)     # on tourne à droite de 90°
forward(50)   # on avance de 50 pixels
done()        # on indique à Turtle qu'on a fini de dessiner
```

On peut faire en sorte que la tortue soit représentée par une vraie tortue, et pas une simple flèche, avec la commande `shape("turtle")`.

On peut aussi modifier la couleur du trait avec des commandes comme `color("red")` ou `color("blue")`.

Exercice

Exercice 56

Écrire un programme permettant de tracer un carré bleu de côté 150 pixels.

Exercice

Exercice 57

Écrire un programme dessinant un triangle équilatéral vert de côté 200 pixels.

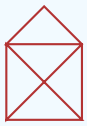
Il est possible d'aller directement en un point en donnant ses coordonnées avec la commande `goto(x, y)`. Heureusement, la tortue peut se déplacer sans écrire, en « levant le crayon » grâce à l'instruction `penup()`. Pour écrire de nouveau, il faut utiliser l'instruction `pendown()`. Par exemple, le code ci-dessous place la tortue au point de coordonnées $(-100\,;50)$ puis trace un segment de 200 pixels de long.

```
penup()          # on lève le crayon
goto(-100, 50)   # on se place au point de coordonnées (-100, 50)
pendown()        # on pose le crayon
forward(200)     # on trace un segment de 200 pixels
done()           # on indique à Turtle qu'on a fini de dessiner
```

Exercice

Exercice 58

Dessiner la forme suivante avec Turtle :



II.2 Des boucles pour des formes plus complexes

Exercice

Exercice 59

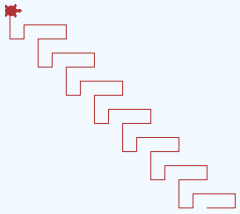
Compléter le programme suivant pour qu'il trace un escalier.

```
for i in range(10):
    forward(20)
    left(90)
    ...
done()
```

Exercice

Exercice 60

Modifier le programme précédent pour qu'il trace la forme ci-contre.



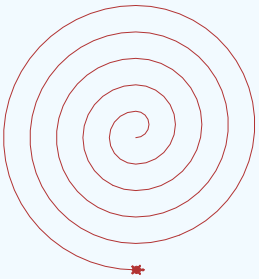
Info

On peut tracer des arcs de cercles avec l'instruction `circle(rayon, angle)`.

Exercice

Exercice 61

Écrire un programme qui trace la spirale ci-contre.



Exercice

Exercice 62

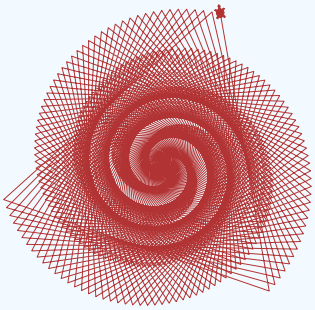
Écrire le code suivant et comprendre ce qu'il trace.

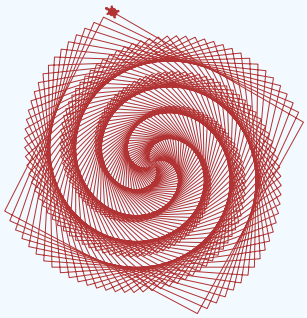
```
speed("fastest")
for i in range(300):
    left(119)
    forward(i)
done()
```

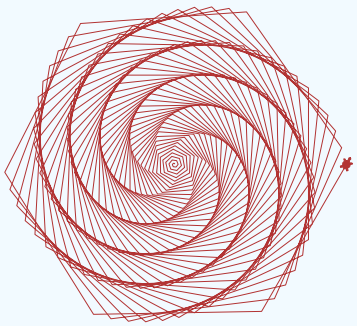
Exercice

Exercice 63

Modifier le code de l'exercice précédent pour obtenir les figures ci-dessous.







II.3 Tracer une fonction avec Turtle

Dans cette partie, on cherche à tracer la représentation graphique d'une fonction avec Turtle. On utilisera la fonction $f : x \mapsto \frac{1}{10}x^2 - 3$, et la fenêtre de visualisation sera réglée avec les paramètres suivants : `xmin = -10`, `xmax = 10`, `ymin = -5` et `ymax = 6`.

La fonction f peut être définie comme ceci en Python :

```
def f(x):
    return x ** 2 / 10 - 3
```


Exercice

Exercice 64

Compléter le code ci-dessous pour qu'il trace la courbe de la fonction f.

```
xmin = -10
xmax = 10
ymin = -5
ymax = 6

# on ajuste les coordonnées de la fenêtre
setworldcoordinates(xmin, ymin, xmax, ymax)

# la tortue va vite !
speed("fastest")

# on va au début de la courbe, sans tracer
penup()
goto(...)
pendown()

# quel pas ?
pas = 0.1

x = xmin
while x < xmax:
    x = x + ...
    goto(...)

done()
```


Exercice

Exercice 65

Modifier le programme précédent pour ajouter des axes et des graduations.

Chapitre 7

Utiliser Python à l'extérieur du lycée

I Installer Python et l'environnement Jupyter

N'exploiter cette partie que si vous souhaitez installer une distribution Python sur votre machine personnelle. Dans tous les cas, l'utilisation de CAPYTALE sera suffisante pour l'intégralité de l'année.

I.1 Sur une distribution GNU/Linux

Si vous utilisez une distribution GNU/Linux, il est fort probable que Python soit déjà installé. Pour installer Jupyter, deux choix s'offrent à vous. Soit vous disposez des droits administrateurs et vous pouvez réaliser l'installation côté système, soit vous faite l'installation uniquement pour votre utilisateur.

I.1.1 Installation sur une distribution basée sur Debian

Si vous utilisez une distribution basée sur Debian, comme Ubuntu par exemple, ouvrez un terminal et lancez la commande suivante :

```
$ sudo apt-get update && sudo apt-get install python3 pip3 python3-jupyter-client
```

Ceci a pour effet d'installer l'environnement Jupyter. Cependant, JupyterLab n'est pas encore packagé pour ces distribution, il convient donc de l'installer "à la main" :

```
$ sudo pip3 install jupyterlab
```

JupyterLab peut maintenant être lancé avec la commande

```
$ jupyter lab
```

I.1.2 Installation sans les droits administrateurs

Vous devez disposer de `pip` en version 3, souvent appelé `pip3`, qui est le gestionnaire de paquets de Python. Ouvrez un terminal et lancez la commande suivante :

```
$ pip3 install jupyter jupyterlab
```

JupyterLab peut maintenant être lancé avec la commande [1]

```
$ jupyter lab
```

I.2 Sur un système Mac OS ou Windows

Nous conseillons d'installer Anaconda en suivant le lien `https://www.anaconda.com/distribution/` et en prenant soin de choisir la version Python 3 (et pas 2) après avoir sélectionné votre système d'exploitation.

1. Pour peu que le chemin `~/.local/bin` se trouve dans votre variable d'environnement `PATH`.

Bibliographie

Thibaut Balabonski, Sylvain Conchon et Jean-Christophe Filliâtre : *Spécialité Numérique et sciences informatiques 1re : 30 leçons et 300 exercices corrigés*. Ellipses, 2019. ISBN 978-2340033641.

Gilles Dowek, Jean-Pierre Archambault, Emmanuel Baccelli, Claudio Cimelli, Albert Cohen, Christine Eisenbeis, Thierry Viéville et Benjamin Wack : *Informatique et sciences du numérique : Spécialité ISN en terminale S avec des exercices corrigés et idées de projets*. Eyrolles, 2013. ISBN 9782212136760.

Benjamin Wack, Sylvain Conchon, Judicaël Courant, Marc de Falco, Gilles Dowek, Jean-Christophe Filliâtre et Stéphane Gonnord : *Informatique pour tous en classes préparatoires aux grandes écoles*. Eyrolles, 2013. ISBN 9782212137002.

Note : En l'état actuel et pour des raisons de calendrier serré, certains passages de ce document peuvent avoir été copiés de certains ouvrages ci-dessus.

Index