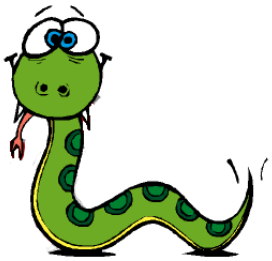
GROUPE INP La Prépa Bordeaux

Informatique - Semestre 1

Algorithmique et Programmation



python™

Année universitaire 2022-2023

Table des matières

0 – Généralités avant de commencer

- Ce fascicule contient l'ensemble du « cours » et des exercices pour le module d'informatique du premier semestre « Algorithmique et Programmation », vous devez le prendre avec vous pour chaque séance.
- Le langage utilisé dans le cadre de ce module est le Python. Ce langage est portable (vous pourrez l'installer sur votre machine quelque soit son système d'exploitation), gratuit et sa syntaxe est très simple (ce qui le rend particulièrement bien adapté à l'apprentissage de la programmation). (C'est aussi le langage imposé par le programme.)
- Une spécificité du langage Python est qu'il est interprété. Sans entrer dans les détails, et pour simplifier, toutes les instructions sont traduites en langage machine (i.e. interprétées) et exécutées **au fur et à mesure** qu'elles sont lues. D'autres langages sont compilés (*C*, *C++*, *Java*, ...) : les instructions du programme sont d'abord toutes traduites en langage machine, puis le programme ainsi traduit dans son ensemble est exécuté.
- La version que vous allez utiliser est Python 3.
- Ce module (environ 25h au total) se décompose en 17 séances (de 1h20) et 2 heures d'évaluation.
- La progression **prévisionnelle** des TP est la suivante :
  - Séances 1 à 3 : Chapitre 1 : Bases de Python
  - Séances 4 et 5 : Chapitre 2 : Fonctions
  - Séances 6 à 9 : Chapitre 3 : Structures de contrôle
  - Séances 10 à 13 : Chapitre 4 : Types composés
  - Séances 14 et 15 : Chapitre 5 : Représentation des nombres
  - Séances 16 et 17 : Chapitre 6 : Dichotomie

  Les étudiants qui seraient en avance sur ce planning peuvent entamer les chapitres et exercices des TP suivants (attention néanmoins à bien comprendre tout ce qui est fait). S'il vous reste quelques exercices non traités à la fin d'un de ces TP, vous les finirez « à la maison ». Le rythme prévu doit vous permettre de traiter tous les exercices en TP, sous réserve d'être actif.

Site de référence

Le site de référence pour Python est à l'adresse `https://docs.python.org/fr/3/`.
Il faut prendre l'habitude de l'utiliser pour toute question que vous vous posez ... et tenter de répondre à celle–ci via une recherche appropriée avant de faire appel à votre professeur !

Programmer en Python en ligne

Si vous souhaitez écrire et tester vos programmes Python sans faire d'installation sur votre machine, vous pouvez utiliser un interpréteur en ligne, par exemple :

`https://repl.it/`

(Il faudra vous créer un compte.)

Installer Python sur sa machine (pour ceux qui le souhaitent)

- Pour les utilisateurs de linux : installer depuis votre gestionnaire les paquets `python3` (s'il n'est pas déjà installé) et `spyder3`.
- Pour les utilisateurs de windows ou Mac, une solution simple est d'installer une distribution comme Anaconda incluant un grand nombre de paquets (dont les paquets scientifiques que nous utiliserons). Vous pouvez télécharger la distribution Anaconda à l'adresse suivante :

  `https://www.anaconda.com/download/`

Très important

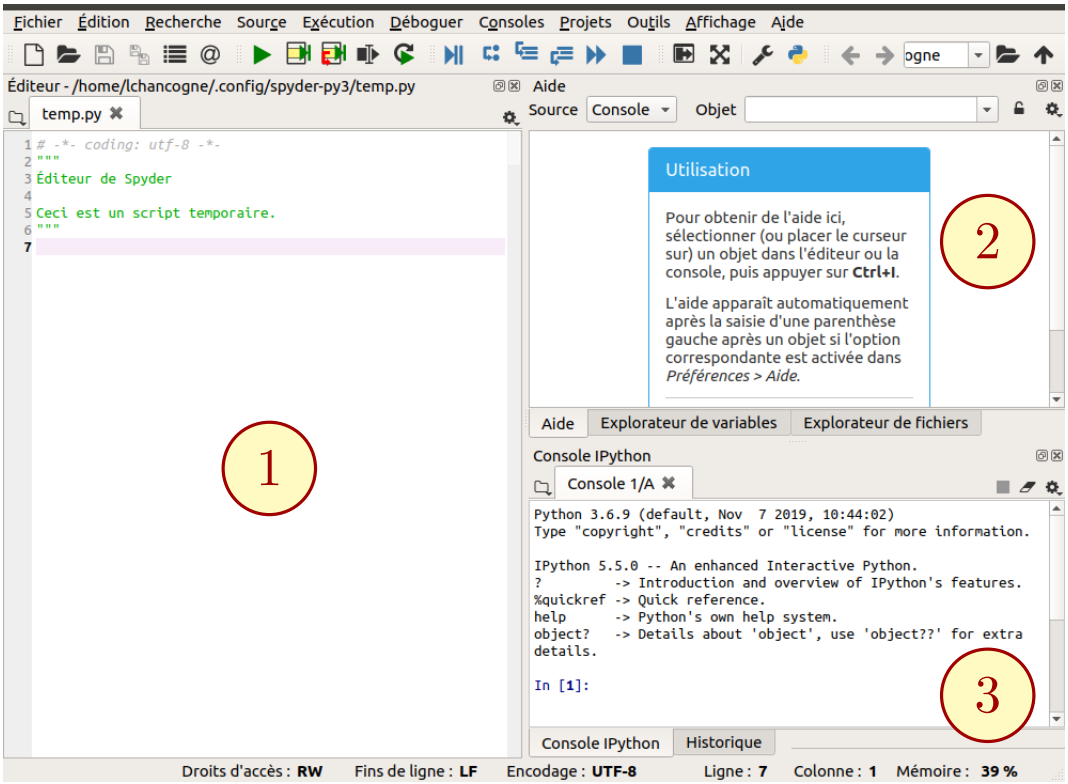
- Ce fascicule contient beaucoup d'exemples signalés avec le symbole . Il faut écrire tous ces exemples (ou pour certains les récupérer sur moodle) et prendre le temps de **comprendre** le résultat des diverses instructions écrites. La compréhension des retours est impérative sous peine d'être complètement perdu par la suite.
- Les exercices signalé par sont **incontournables** : ce sont des exercices d'application directe de la partie cours, vous devez pouvoir les faire tout seul. (Et vous devez les avoir faits s'ils sont à chercher pour une séance.)
- Les exercices signalé par sont **facultatifs** : il permettront aux plus à l'aise d'aborder des exercices un peu plus compliqués, mais ils ne sont pas nécessaires ni pour la compréhension du chapitre, ni pour la suite. Vous pouvez les aborder quand tous les autres exercices du chapitre sont finis.
- Les exercices signalés par ★ sont un peu plus difficiles que les autres : ils vous demanderont peut-être un peu de persévérance pour en arriver au bout.

Les séances de TP **ne sont pas** une course de vitesse pour savoir qui finira le fascicule en premier !!!
Il est vivement conseillé de prendre des notes au fur et à mesure des TP.
Aucune aide ne sera donnée si la réponse fait partie d'un TP précédent.

1 – Bases de Python

1.1 – Environnement de programmation

Vous allez programmer en Python via un EDI (environnement développement intégré, IDE en anglais) qui est **Spyder**. Au démarrage de Spyder, on obtient une fenêtre semblable à l'image ci-dessous :



Le logiciel Spyder est présenté lors des premières séances d'informatique. L'image ci-dessus représente le logiciel tel qu'il s'affichera lorsque vous l'utiliserez pour la première fois. La fenêtre est divisée en 3 zones :

- Zone 1 : L'éditeur de programme (ou script). C'est la zone dans laquelle vous écrirez votre programme.
- Zone 2 : Une aide à la « programmation ». On se servira en particulier de l'explorateur de variables lors de la mise au point des programmes.
- Zone 3 : La console. C'est ici que seront affichés vos résultats. De plus, la console est en « mode interactif » : on peut écrire directement des instructions (au niveau de ce que l'on appelle l'invite de commande, `In [n]:`) qui sont interprétées à chaque retour à la ligne.

Notes

- À ce stade, on peut déjà utiliser Python en mode interactif comme une « calculatrice ».
- Exemple : Dans la zone 3, taper « 1+2 » puis « Entrée ».

```
In [1]: 1+2
Out[1]: 3
```

- Pour éviter de copier plusieurs fois la même ligne, on peut bien sûr utiliser les raccourcis claviers standards Ctrl + C et Ctrl + V (pour « Copier » et « Coller »), mais on peut aussi utiliser une fonctionnalité de Spyder : depuis la console interactive on peut naviguer dans l'historique des commandes tapées avec les flèches « haut » ↑ et « bas » ↓. (Essayez !)

1.2 – Objets, expressions et types simples

Un programme en Python manipule des « objets ». Chaque objet possède un type qui définit l'ensemble des valeurs possibles pour l'objet et les propriétés qui caractérisent tous les objets du même type. Les types peuvent être des scalaires (types simples) ou non. Les types composés (ou non scalaires) seront abordés plus tard.

Les principaux types de « scalaires » en Python sont :

- Le type `int` (pour « integer »), pour représenter les entiers. En principe, la taille maximale des entiers de type `int` dépend de la machine et donc du codage en 32 bits ou 64 bits. En 32 bits, les entiers vont de -2^{31} à $2^{31}-1$ (soit de $-2\,147\,483\,648$ à $2\,147\,483\,647$). En 64 bits, les entiers vont de -2^{63} à $2^{63}-1$ (soit de $-9\,223\,372\,036\,854\,775\,808$ à $9\,223\,372\,036\,854\,775\,807$).

  Python3 permet en fait de manipuler des entiers aussi longs que l'on veut, la seule limite est la mémoire disponible. En Python3 la représentation des entiers relatifs en mémoire n'est donc pas tout à fait la représentation standard (qui est celle du complément à 2 qui sera vue au chapitre 5). Cette représentation est plus coûteuse pour les opérations mais enlève la limite maximale imposée par la plupart des langages pour manipuler des entiers.
- Le type `float`, pour représenter les « nombres réels ». Le séparateur décimal est le point. Les nombres sont stockés en mémoire suivant un principe de « virgule flottante » et sont communément appelés des **flottants**. Pour rappel, les capacités mémoire étant finies, il est impossible de représenter l'ensemble des nombres réels ; ainsi, un flottant n'est qu'une **approximation** d'un réel donné. Par exemple `1.0` n'est pas le réel 1 mais une approximation de celui–ci.
- Le type `bool` pour représenter les valeurs booléennes « Vrai » et « Faux » (`True` et `False` en Python).
- Le type `NoneType`, qui n'a qu'une seule valeur : `None` (« Rien » ou « Vide »).

Une combinaison des divers objets à l'aide d'opérateurs s'appelle une **expression**.

Les principaux opérateurs

Opérateurs	Symbole	Type des opérandes	Type du résultat	Remarque
Opposé	`-`	`int`	`int`	
		`float`	`float`	
Addition	`+`	`int`	`int`	
		`float`	`float`	
Soustraction	`-`	`int`	`int`	
		`float`	`float`	
Produit	`*`	`int`	`int`	
		`float`	`float`	
Division	`/`	`int`	`float`	Retourne toujours un `float`.
		`float`	`float`	
Puissance	**	`int`	`int`	`a**b` pour a^b
		`float`	`float`	
Division entière	`//`	`int`	`int`	Quotient de la division entière[1]
Modulo	`%`	`int`	`int`	Reste de la division entière[1]
Comparaison	`<`	`int` ou `float`	`bool`	Ne pas confondre[2] `==` et `=`
	`<=`			
	`>`			
	`>=`			
	`==`			
	`!=`			« différent de »
ET	`and`	`bool`	`bool`	
OU	`or`	`bool`	`bool`	
NON	`not`	`bool`	`bool`	

(1) Attention : Le quotient et le reste de la division entière de « Python » correspondent bien aux quotient et reste de la division euclidienne pour la division par un **positif** ... Mais la définition Python est différente de celle des mathématiques pour la division par des négatifs. (Essayez !)

(2) La comparaison pour déterminer si les valeurs de deux objets sont égales se fait avec un « double égal », le « simple égal » ayant un autre rôle (voir 1.4 - Variables et affectation).

Remarques

- La priorité des opérateurs est la même qu'en mathématiques.
- Si l'on fait un calcul qui mélange entier et flottant, Python fait ce qu'on appelle une promotion de type. Les entiers sont convertis à la volée en flottants.
- Pour connaître le type d'un objet ou d'une expression, on utilise la fonction `type`.

  `type(expression)` : par exemple, `type(2*5.6-3.14)`.
- Pour déterminer si une expression est d'un type donné, on utilise la fonction `isinstance`.

  `isinstance(expression,type)` : par exemple, `isinstance(2*5.5-3,int)`.

  La valeur retournée est un booléen.

Valeur absolue

Fonction	Notation Python	Type en entrée	Type en sortie	Remarque
Valeur absolue	`abs`	`int`	`int`	`abs(x)` pour $\|x\|$
		`float`	`float`	

Les nombres complexes

Remarque

Le type « `complex` » est abordé ici pour faciliter d'éventuelles recherches sur les types simples (en groupant les informations au même endroit). Pour les étudiants n'ayant pas vu les nombres complexes en terminale, cette partie peut être passée.
Un rappel sera fait lors des exercices utilisant les nombres complexes et vous pourrez revenir sur cette partie plus tard.

- Le nombre complexe $a+ib$ avec a et b réels s'obtient avec la fonction `complex` : `complex(a,b)`.

  Le type d'une expression complexe est `complex`. Sa partie réelle et sa partie imaginaire sont de type `float`.
- Le nombre complexe i s'écrit `1j` en Python. De manière plus générale une valeur numérique suivie de `j` représente un nombre complexe de partie entière 0 et de partie imaginaire cette valeur numérique. Par exemple, `5j` représente $5i$. Attention à ne pas mettre d'espace entre la valeur numérique et `j`.

  On peut donc obtenir le nombre complexe $a+ib$ d'au moins 2 façons : `complex(a,b)` ou `a+1j*b`.
- Quelques « fonctions » sur les complexes :
  - ◇ `abs` permet d'obtenir la norme d'un nombre complexe. Exemple : `abs(complex(1,1))`.
  - ◇ `real` et `imag` permettent d'obtenir respectivement la partie réelle et la partie imaginaire d'un nombre complexe (attention, ce ne sont pas des fonctions).

    Exemple : `complex(1,2).real`, `complex(1,2).imag`.
  - ◇ `conjugate()` permet d'obtenir le conjugué d'un nombre complexe.

    Exemple : `complex(1,2).conjugate()`.

1.3 – Utiliser les fonctions d'une bibliothèque

Si l'on essaie des calculs autres que ceux de base, par exemple `exp(1)` ou `pi`, on obtient une erreur. Python serait très limité si nous n'avions accès qu'aux opérations « de base » ... Python peut faire ces calculs mais il faut d'abord mettre en mémoire les fonctions à utiliser avec ce qu'on appelle un import.

L'import d'une bibliothèque ou de l'une de ses fonctions peut se faire de plusieurs façons différentes.

Par exemple pour la bibliothèque `math` :

- `import math` : Après cette instruction et pour toute la suite du programme, on peut utiliser les fonctions du module en faisant précéder leur nom de celui du module, par exemple `math.sqrt(2)`.
- `import math as m` : Le résultat est le même que pour `import math` mais avec un alias pour le nom du module. Par exemple, pour utiliser la racine carrée on écrira `m.sqrt(2)`.
- `from math import sqrt` : On importe uniquement la fonction dont on a besoin (ici `sqrt`). Par la suite, on peut faire appel à cette fonction directement sans spécifier le module, par exemple : `sqrt(2)`.
  ⚠ Attention, comme le module n'est plus spécifié, il pourrait y avoir « télescopage » avec une fonction `sqrt` définie ailleurs.
- `from math import *` : On importe toutes les fonctions du module. Par la suite on peut faire appel à toutes ces fonctions directement sans spécifier le module.
  ⚠ Même si cela peut paraître plus pratique, c'est en fait assez dangereux. Le problème est le même que ci–dessus, mais cette fois pour toutes les fonctions du module et dans ce cas sans avoir la visibilité de l'ensemble des noms des fonctions. À titre d'exemple, la fonction cosinus est définie dans le module `math` mais aussi dans un module `numpy` qui sert au calcul numérique. Ces fonctions sont construites différemment suivant le module et offrent donc des fonctionnalités différentes. En utilisant un « `import *` » pour les deux modules, cela posera donc un problème à l'utilisation.
  Cette dernière forme d'importation est donc à utiliser avec **parcimonie** ...

Une fois l'import réalisé, on peut travailler avec toutes les fonctions correspondantes.

1.3.1 – Quelques fonctions et constantes du module math

- `e` et `pi` : les constantes $e = \exp(1)$ et π.
- `log(x)` et `exp(x)` : pour obtenir $\ln(x)$ et e^x.
- `log(x,b)` : pour obtenir le logarithme de x en base b, $\log_b(x) = \dfrac{\ln(x)}{\ln(b)}$.
- `cos(x)`, `sin(x)` et `tan(x)` : les fonctions trigonométriques (x en radians).
- `acos(x)`, `asin(x)` et `atan(x)` : les fonctions trigonométriques réciproques.
- `floor(x)` et `ceil(x)` : pour obtenir $\lfloor x \rfloor$ et $\lceil x \rceil$.
- `sqrt(x)` : pour obtenir $\sqrt{x}$.
- `factorial(x)` pour obtenir $x!$.

Exemple 1.1.

Les instructions suivantes sont à taper dans la console interactive puis à valider avec Enter .

```
In [1]: from math import pi, log, sin, acos, sqrt, factorial
In [2]: print(pi)
In [3]: print(sin(pi/2))
In [4]: print(log(1))
In [5]: print(log(10))
In [6]: print(log(10**3,10))
In [7]: print(acos(1))
In [8]: print(sqrt(9))
In [9]: print(factorial(3))
In [10]: print(factorial(0))
In [11]: print(factorial(10))
In [12]: print(factorial(20))
```

Note

- Pour plus de précisions sur le module `math`, voir :

  `https://docs.python.org/fr/3/library/math.html`

1.3.2 – Un peu d'aléatoire

Un autre module intéressant, pour faire des tests par exemple, est le module `random`. Ce module contient en particulier deux fonctions (bien penser à réaliser un import de `random` **avant** d'utiliser ces fonctions) :

- `random()` : génère un nombre aléatoire dans l'intervalle $[0;1[$.
- `randint(a,b)` : génère un entier aléatoire dans $[\![a,b]\!]$.
- `uniform(a, b)` : génère un flottant aléatoire dans $[a,b]$.

Exemple 1.2.

Les instructions suivantes sont à taper dans la console interactive puis à valider avec Enter .

```
In [13]: from random import randint, random, uniform
In [14]: random()
In [15]: randint(1,100)
In [16]: uniform(pi,2*pi)
```

Note

- Pour plus de précisions sur le module `random`, voir :

  `https://docs.python.org/fr/3/library/random.html`

1.4 – Variables et affectation

Vous trouverez sur moodle un lien vers une vidéo expliquant la notion de variable.

Dans un programme, une **variable** sert à désigner une zone de la mémoire de l'ordinateur qui peut être vue comme une sorte de « boîte ». On peut y stocker une valeur, la changer et y accéder. Pour faire référence à cette zone de la mémoire de l'ordinateur on utilise un **nom de variable**. Une variable porte donc un nom et est associée à un « objet » en mémoire.

Si on écrit `var=2`, on lie le nom `var` à un objet de type `int`. On dit que l'on a **affecté** (ou **assigné**) 2 à `var`. Le symbole `=` sert donc à l'affectation des variables (et il faudra donc bien penser lors de la comparaison de deux variables à utiliser `==` sous peine que le résultat de l'instruction soit quelque peu différent de celui attendu). On peut alors par la suite utiliser cette variable et même la ré–affecter.

Afin de faciliter la relecture d'un programme, il est important de bien choisir les noms utilisés : pas trop longs, mais suffisamment explicites.

En Python, contrairement à d'autres langages, on n'a pas à définir le type d'une variable à l'avance. La variable prend le type de l'objet qui lui est affecté (au moment de l'affectation), et le type d'une variable peut donc changer au fur et à mesure des instructions. On dit que le typage des variables sous Python est un typage **dynamique**, par opposition au typage **statique** (par exemple en *C++* ou en *Java*).

Sous Python, les noms de variables doivent respecter quelques règles simples :

- Un nom de variable est une séquence de lettres (a à z, A à Z) et de chiffres (0 à 9), qui doit toujours commencer par une lettre.
- Les espaces et les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère « _ » (souligné). Les lettres accentuées et les cédilles peuvent être utilisées, mais c'est **fortement** déconseillé.
  Dans le cadre « de La Prépa », **seules les lettres ordinaires** (et « _ ») **seront autorisées.**
- La casse est significative (les caractères majuscules et minuscules sont distingués).
   Attention : Variable, variable, VARIABLE sont donc **des variables différentes** !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en améliorer éventuellement la lecture.

En plus de ces règles, on ne peut pas utiliser comme nom de variable les 32 mots réservés ci–dessous (ils sont utilisés par le langage lui–même) :

`and`	`continue`	`except`	`if`	`not`	`try`	`True`
`as`	`def`	`finally`	`import`	`or`	`while`	`False`
`assert`	`del`	`for`	`in`	`pass`	`with`	`None`
`break`	`elif`	`from`	`is`	`raise`	`yield`	
`class`	`else`	`global`	`lambda`	`return`		

De plus, **vous ne devez pas utiliser** `exec` et `print` comme nom de variable (même si c'est possible).

Exemple 1.3. (Rappels)

Les instructions (signalées par) sont à taper dans la console interactive et puis valider avec Enter .

- Affecter la valeur 3 à la variable `var` : `var=3`.

  Attention, l'affectation se fait toujours à gauche : `10=var` est incorrect.
- Afficher le contenu de la variable `var` : `print(var)`.
- Connaître le contenu de la variable `var` : on peut aussi tout simplement utiliser `var`. Le résultat est alors précédé de « `Out` ». Attention, ce n'est valable que dans la console interactive.
- Affecter simultanément des valeurs à plusieurs variables (affectation multiple) : `x,y,z=1,2,3`.

  (Vous pouvez ensuite visualiser le contenu des variables : `print(x,y,z)`.)
- Supprimer/Effacer la variable `var` : `del(var)`. (Essayer ensuite l'instruction `print(var)`.)
- Taper plusieurs instructions sur une seule ligne : il suffit de séparer les instructions par des points-virgules. Les instructions seront réalisées de gauche à droite : `A=5;B=A-3;A=3`.

  (Vous pouvez ensuite visualiser le contenu des variables : `print(A,B)`.)

1.5 – Chaînes de caractères

1.5.1 – Introduction

Python n'a pas de type spécifique pour les caractères. Le type `str` (abréviation de string) est utilisé pour représenter les chaînes de caractères. Un caractère est alors une chaîne de caractères particulière (sa longueur est 1). On parle de **séquence** de caractères pour une chaîne de caractères. En Python, les chaînes de caractères sont délimitées par des **guillemets** ou des **apostrophes** (au choix). Par exemple, on pourra écrire `'Bonjour'` ou `"Bonjour"`.

Les chaînes de caractères peuvent contenir des caractères « spéciaux ». Pour indiquer à Python qu'il s'agit de caractères spéciaux, on les fait précéder d'un « backslash » (i.e. \). Par exemple : une tabulation (`\t`), un retour à la ligne (`\n`), une apostrophe dans une chaîne délimitée par des apostrophes (`\'`) ou un guillemet dans une chaîne délimitée par des guillemets (`\"`).

Certaines opérations définies pour les types numériques peuvent être utilisées pour les chaînes de caractères, mais elles ont sur ces dernières un comportement différent : on dit que ces opérateurs sont **surchargés**.

Quelques commandes sur les chaînes de caractères (nous reviendrons plus en détail sur le type `str` dans le chapitre 4 - Types composés) :

- L'opération « + » avec plusieurs chaînes de caractères réalise un concaténation de ces chaînes (i.e. une juxtaposition de celles-ci).
- Le produit d'une entier n avec une chaîne de caractères réalise la concaténation de n fois cette chaîne de caractères.
- La longueur chaîne de caractères c (i.e. le nombre de caractères de la chaîne) s'obtient avec `len(c)`.

Exemple 1.4.

Les instructions suivantes sont à taper dans la console interactive puis à valider avec Enter .

```
In [1]: mot1='Bonjour !'
In [2]: mot2="Au revoir."
In [3]: print(mot1, mot2)
In [4]: print("\"guillemets\"")
In [5]: print('"guillemets"')
In [6]: motEt='et'
In [7]: mot=mot1+motEt+mot2
In [8]: print("Concaténation : ",mot)
In [9]: print(mot1+"\n et \n"+mot2)
In [10]: print("Répétition : ", 4*mot1)
In [11]: print(len(mot1))
In [12]: variable=5
In [13]: print(3*'variable')
In [14]: print(3*variable)
```

1.5.2 – Premières fonctionnalités

On considère une variable `chaine` contenant une chaîne de caractères de longueur n.

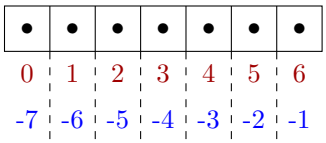
- On obtient le nombre de caractère de `chaine` avec la fonction `len` : `len(chaine)`.
- On peut obtenir un caractère de `chaine` à l'aide de son « index » : `chaine[index]`.

  Attention ! Le premier caractère de la chaîne `chaine` est indexé **par 0** ...

  Les caractères de `chaine` sont indexés de 0 à $n-1$.
- On peut utiliser des index négatifs. Le dernier élément de la chaîne `chaine` peut s'obtenir avec l'index -1, l'avant dernier avec -2, etc., et le premier élément avec $-n$.

  Il est ainsi équivalent d'écrire :

  → `chaine[-1]` ou `chaine[n-1]`,
  → `chaine[-2]` ou `chaine[n-2]`,
  → ...,
  → `chaine[-n]` ou `chaine[0]`.



- « Sous-chaîne » : On accède une sous-chaîne en la découpant par l'intermédiaire des « index » de début et de fin. On utilise la syntaxe `chaine[debut:fin]`.

  Attention ! Le caractère indexé par `fin` **est exclu**.

  – Sous-chaîne des caractères de la position a à b : `chaine[a:b]`.
  Sous-chaîne de a inclus à b **exclu**.
  – Sous-chaîne des caractères de la position a à la fin : `chaine[a:]`.
  – Sous-chaîne des caractères du début à la position b (**exclue**) : `chaine[:b]`.

Exemple 1.5.

Les instructions suivantes sont à taper dans la console interactive puis à valider avec Enter .

```
In [1]: mot='Bonjour'
In [2]: n=len(mot) ; print(n) # Nombre de lettre (7)
In [3]: print(mot[0]) # Première lettre (B)
In [4]: print(mot[-1]) # Dernière lettre (r)
In [5]: print(mot[-n]) # Première lettre (B)
In [6]: print(mot[n-1]) # Dernière lettre (r)
In [7]: print(mot[2:4]) # 3eme et 4eme lettre (nj)
In [8]: print(mot[3:]) # De la 4eme lettre à la fin (jour)
In [9]: print(mot[:3]) # Du début à la 3eme lettre (Bon)
```

1.5.3 – Formater l'affichage

La fonction `print` a été utilisée avec plusieurs paramètres pour afficher un « mélange » de texte et de variables. Il existe une méthode plus souple qui utilise le formatage d'une chaîne de caractères à l'aide de ce que l'on appelle une **méthode** (pour simplifier, une méthode est une sorte de fonction spécifique à un type d'objet). La méthode pour formater une chaîne de caractères est « `format` ».

Les exemples ci-dessous sont directement tirés du tutoriel Python :

`https://docs.python.org/fr/3/tutorial/inputoutput.html`

- L'utilisation de base de la méthode `str.format()` ressemble à cela :

```
In [1]: print('We are the {} who say "{}!"'.format('knights', 'Ni'))
Out[1]: We are the knights who say "Ni!"
```

Les accolades et les caractères qu'ils contiennent (appelés les champs de formatage) sont remplacés en fonction de leur ordre d'apparition par les objets passés en paramètres à la méthode `str.format()`.

- Pour spécifier un formatage de l'objet, on utilise le caractère « `:` » suivi du format souhaité.
Ceci offre un niveau de contrôle plus fin sur la façon dont les valeurs sont formatées.
L'exemple suivant arrondit π à trois décimales :

```
In [6]: import math
In [7]: print("The value of PI is approximately {:.3f}".format(math.pi))
Out[7]: The value of PI is approximately 3.142
In [8]: print("The value of PI is approximately {:10.3f}".format(math.pi))
Out[8]: The value of PI is approximately      3.142
```

- La donnée d'un entier après le caractère « `:` » indique la largeur minimale de ce champ en nombre de caractères. C'est utile pour faire de « jolis » tableaux.
Exemple :

```
In [9]: ligne1 = "{0:10} ==> {1:10d}".format("Laurent", 4127)
In [10]: ligne2 = "{0:10} ==> {1:10d}".format("Cedric", 98)
In [11]: ligne3 = "{0:10} ==> {1:10d}".format("Toto", 127678)
In [12]: tableau = ligne1 + "\n" + ligne2 + "\n" + ligne3
In [13]: print(tableau)
```

Remarques

- Le « `f` » dans les accolades signifie « `float` » et le « `d` » signifie « `int` ».
- On verra une manière plus efficace d'afficher un tableau avec les listes et les boucles.

- On peut aussi utiliser ce que l'on appelle les « f-strings » ou chaînes de caractères formatées qui permettent d'inclure la valeur d'expressions de manière directe en écrivant les expressions dans les accolades. Pour cela, il suffit de préfixer les chaînes de caractères d'un `f`. L'expression (dont la valeur est à inclure) est placée entre accolade et peut être suivie d'un formatage spécifique, comme pour la méthode format.
Exemple :

```
In [14]: x=math.pi/4
In [15]: print(f"The value of PI is approximately {4*x:.3f}")
```

- **En résumé**

« `format` » permet de séparer le « fond » de la « forme ». La chaîne de caractère (la forme) comprend des accolades qui sont remplacés en fonction de leur ordre d'apparition par les paramètres (le fond) donnés à la méthode `str.format()`. Exemple : `"{}+{}={}".format(2,3,2+3)` → `"2+3=5"`.

On peut préciser dans les accolades le nombre d'espaces réservés pour l'affichage, le type de l'objet affiché, pour les flottants le nombre de chiffres après la virgule.

- `"{}+{:3}={:2}".format(2,3,5)` → `"2+  3= 5"` (`"2+␣␣3=␣5"`)
- `"{0}~{0:5.2f}".format(1.123456)` → `"1.123456~ 1.12"` (`"1.123456~␣1.12"`)

  `0` : position du paramètre / `5` : nombre d'espaces réservés pour l'affichage / `2` : nombre de chiffres après la virgule (pour les flottants) / `f` : type de l'objet affiché (`f` : flottants ; `d` : entiers)

1.6 – Scripts Python

Les instructions que l'on a utilisées dans la console en mode interactif peuvent être écrites dans un fichier texte (avec l'extension `.py`) afin d'être exécutées plus tard sans avoir à tout ré–écrire. On peut créer un nouveau fichier avec « Nouveau fichier ... » du menu « Fichier » ou son « bouton » dans la barre d'outils, ou encore avec le raccourci Ctrl + N . Pour ouvrir un fichier existant, il faut utiliser « Ouvrir ... » du menu Fichier ou son « bouton » dans la barre d'outils (ou encore Ctrl + O). Ces fichiers sont appelés « scripts » ou « programmes ».

Dans un script (ou programme), il n'y a pas d'invite de commande (`In [n]:`) et on écrit en général une instruction par ligne (mais il est possible comme en mode interactif d'en écrire plusieurs avec le séparateur « ; »).

Exécuter un script, c'est demander à Python d'interpréter et d'exécuter ligne par ligne toutes les instructions du fichier. On peut exécuter un script avec « Exécution » du menu « Exécution » ou son « bouton » dans la barre d'outils (la flèche verte pointant sur la droite). On peut aussi utiliser le raccourci F5 pour exécuter un programme ...

1.6.1 – Commentaires

Pour pouvoir écrire des commentaires dans un script (ou programme) et que Python les ignore lors de l'exécution du script, on peut s'y prendre de deux façons :

- Faire précéder le commentaire du caractère `#`.
- Pour écrire un commentaire sur plusieurs lignes, il suffit de l'encadrer par des « triples–quote » (`'''` ou `"""`).

1.6.2 – Exemple

Exemple 1.6.

Écrire, sauvegarder puis exécuter le script suivant :

```
# -*- coding: utf-8 -*-

print("Hello World !")
print('Premier ',end="")
print('script')

# definition d'une variable et affectation d'une valeur
var=1

# Les commentaires n'ont aucun effet sur les instructions
var=var+1 # var est augmentee de 1

var+100
'''
Python calcule effectivement var+100 mais rien n'apparait
dans l'affichage car on ne demande pas a Python d'afficher
le resultat
'''

print(var+10)
print("*******")

a=2021
b=72
quotient=a//b
reste=a%b

# La fonction print accepte plusieurs arguments
print(f'Le quotient de la division de {a} par {b} est {quotient}')
print(f"Le reste de la division de {a} par {b} est {reste}")
```

Les résultats sont affichés dans la console. L'exécution du script est équivalente à l'exécution de chacune des lignes dans la console en mode interactif, en particulier, on peut continuer à faire des calculs avec les variables utilisées dans le script (essayer par exemple `a==(b*quotient+reste)`).

Pour réinitialiser la console (afin d'effacer toute trace du programme), il faut utiliser la commande « Redémarrer le noyau » du menu « Consoles » ou utiliser le raccourci Ctrl + . .

Le paramètre « `end=""` » dans le 2ème « `print` » du script ci-dessus permet d'indiquer qu'après l'affichage de « Premier » Python ne doit pas revenir à la ligne comme le comportement par défaut de `print`.

« `end=` ... » indique la chaîne de caractère à afficher en fin de ligne (à la place du comportement par défaut), on peut évidemment choisir n'importe quelle chaîne de caractères (par exemple `end="@"`, `end="--"`, ou si l'on ne veut rien afficher de plus après le « `print` » : `end=""`).

« `end="\n"` » correspond au comportement par défaut.

Notes

- Pour pouvoir utiliser des caractères non « standard » (c'est à dire des accents ici), il faut spécifier à Python la norme d'encodage à utiliser (ici `utf-8`). C'est le rôle de l'instruction qui se trouve en première ligne « `# -*- coding: utf-8 -*-` ». (Cette instruction doit se trouver en première ou deuxième ligne du script.) Elle peut être omise si l'on n'utilise **que des caractères standards** mais il est fortement recommandé de spécifier dans tous vos programmes le type d'encodage souhaité (en général `utf-8` ou `latin-1`).
  Ce problème n'est pas lié à Python, c'est un problème « global » en informatique.
- Comme rappelé ci-dessus, on peut écrire plusieurs instructions sur une seule ligne : il suffit de séparer les instructions par des points-virgules. Les instructions seront réalisées de gauche à droite.
- Il est aussi possible d'écrire une instruction sur plusieurs lignes (pour améliorer la lisibilité par exemple), il suffit de finir la ligne par un « anti–slash » (\) qui indique à Python que l'instruction n'est pas finie.

1.6.3 – « input »

Pour rendre nos programmes (ou scripts) plus « modulaires », on va établir un « dialogue » avec l'utilisateur lui permettant de spécifier certains paramètres. On parle de flux d'entrées/sorties pour les données saisies par l'utilisateur/retournées par le programme. Pour cela, on utilise la fonction `input`. Cette fonction permet de récupérer ce qui est saisi par l'utilisateur sous forme d'une **chaîne de caractères**.

Exemple 1.7.

Écrire, sauvegarder puis exécuter le programme suivant :

```
# -*- coding: utf-8 -*-
prenom = input('Quel est votre prénom ? ')
print("Bonjour {}".format(prenom))
```

Exemple 1.8.

Écrire, sauvegarder puis exécuter le programme suivant :

```
# -*- coding: utf-8 -*-
x = input('Donnez un nombre entier : ')
print(f"Le triple du nombre {x} est : {3*x}")
```

Le deuxième programme ne semble pas fonctionner correctement ...

En effet `input` retourne une chaîne de caractères (`str`). La variable `x` étant donc du type `str`, l'expression `3*x` correspond donc à la concaténation de 3 fois la chaîne de caractères « `x` ». Python ne calcule donc pas le triple du nombre `x` comme on le souhaiterait.

Conversion de type

On peut être amené à devoir changer le type d'une variable dans un programme. Dans l'exemple 1.8, afin de pouvoir multiplier le nombre par trois il faudrait d'abord le convertir en entier.

On peut procéder de la façon suivante :

```
# -*- coding: utf-8 -*-
x = input('Donnez un nombre entier : ')
y = int(x)
print(f"Le triple du nombre {y} est : {3*y}")
```

Ou de manière plus compacte :

```
# -*- coding: utf-8 -*-
x = int(input('Donnez un nombre entier : '))
print(f"Le triple du nombre {x} est : {3*x}")
```

Les fonctions `int`, `float` et `str` permettent de convertir une expression donnée en, respectivement, un entier, un flottant, une chaîne de caractères. Attention, il faut que la conversion soit possible ! Par exemple `int("5.36")` est incorrect car 5.36 correspond à un flottant.

Exemple 1.9.

Les instructions suivantes sont à taper dans la console interactive puis à valider avec Enter .

```
In [1]: a=7
In [2]: float(a)
In [3]: int(float(a))
In [4]: int(1.54)
In [5]: str(1.54)
In [6]: str(4591)
In [7]: float("2")
In [8]: float("5.36")
In [9]: int("1234")
In [10]: int("5.36")
In [11]: int(float("5.36"))
```

1.7 – Exercices

Exercice 1.1 *(Incontournable)*

Quel sera le résultat des instructions suivantes ?
(Répondez sans utiliser Python, puis vérifier votre réponse en tapant l'instruction dans la console interactive.)

1. `(3**2)**(1//2)`
2. `(3**2)**1//2`
3. `(3**2)**0.5`
4. `5//4`
5. `5/4`
6. `floor(5/4)`
7. `ceil(pi)`
8. `5%4`

Exercice 1.2 *(Incontournable)*

Parmi les noms de variables suivants, quels sont ceux qui sont valides ?
(Répondez sans utiliser Python, puis vérifier votre réponse en essayant d'affecter une valeur à la variable.)

1. `total$`
2. `from`
3. `2lambda`
4. `i_8`
5. `lambda2`
6. `_var`
7. `var#12`
8. `lambda`

Exercice 1.3 *(Incontournable)*

Quel sera le résultat des instructions suivantes ?
(Répondez sans utiliser Python, puis vérifier votre réponse en tapant l'instruction dans la console interactive.)

1. `var="Echo" ; print(5*var)`
2. `var="Echo" ; print(5*"var")`
3. `var=2 ; print(type(var+var)) ; print(var+var)`
4. `var=2. ; print(type(var+2)) ; print(var+2)`
5. `var="2" ; print(type(var+var)) ; print(var+var)`
6. `var=False ; print(type(var or var)) ; print(var or var)`
7. `i=7 ; j=5 ; print("L'entier "+i+" est plus petit que l'entier "+j)`
8. `i=7 ; j=5 ; print("L'entier "+str(i)+" est plus grand que l'entier "+str(j))`
9. `i=7 ; j=5 ; print("L'entier {} est plus grand que l'entier {}".format(i,j))`
10. `i=7 ; j=5 ; print(f"L'entier {i} est plus grand que l'entier {j}")`

Exercice 1.4 *(Incontournable)*

Les variables `varEcho`, `varEspace` et `varLigne` sont initialisées par `varEcho="Echo"`, `varEspace=" "` et `varLigne="\n"`. Pour cet exercice, vous utiliserez uniquement les opérations `*`, `+`, les parenthèses, les nombres entiers, les trois variables ci-dessus, et **une seule** fonction `print`.

Donner les instructions permettant d'afficher :

1. `Echo Echo Echo Echo Echo`
2. `Echo`
   `␣Echo`
   `␣␣Echo`

(Les symboles « ␣ » ne sont pas à afficher. Ils ne sont là que pour signaler les espaces que vous devez obtenir.)

Exercice 1.5 *(Incontournable)*

1. Écrire puis exécuter le programme suivant :

```
# -*- coding: utf-8 -*-
A=3;B=12
chaine1 = str(A)+" fois 4 = "+str(B)
chaine2 = str(B+1)+" = "+str((B+1)//A)+" x "+str(A)+" + "+str((B+1)%A)
print("Chaine 1 :")
print(chaine1)
print("Chaine 2 :")
print(chaine2)
```

2. Définir les variables `ch1` et `ch2` à l'aide de la méthode `format` puis d'une « f-string » afin que les instructions `print(ch1)` et `print(ch2)` produisent le même affichage que les instructions `print(chaine1)` et `print(chaine2)` ci-dessus.

   Par exemple si `P=3.14` et `chaine0="pi="+str(P)`, on définirait `ch0` à l'aide :

   - de la méthode `format` par `ch0="pi={}".format(P)`,
   - et à l'aide d'une « f-string » par `ch0=f"pi={P}"`.

Exercice 1.6

Écrire un programme qui calcule (et affiche) la moyenne de 3 notes, déterminées aléatoirement entre 0 et 20, qui seront stockées dans trois variables `var1`, `var2` et `var3`. Puis, faites plusieurs essais de votre programme.

Exercice 1.7

Écrire un programme qui à partir du rayon donné par l'utilisateur affiche le périmètre et la surface du disque correspondant. Puis, faites plusieurs essais de votre programme.

Exercice 1.8

On veut écrire un programme qui demande à l'utilisateur un nombre de secondes et affiche la durée correspondante au « format » heures/minutes/secondes.

Par exemple, si l'utilisateur saisit `9447`, le programme affichera alors `2h 37min 27s`.

1. Vérifier le résultat donné en exemple ci-dessus et donner ce que le programme affichera pour `18769`.
2. Écrire l'algorithme permettant de trouver le résultat (c'est-à-dire la suite des « opérations » à effectuer).
3. Écrire le programme en Python.
4. Proposez des tests pour vérifier votre programme (et faites les ...).

Exercice 1.9

On donne le programme suivant (incomplet) :

```
# -*- coding:utf-8 -*-
x = 1
y = 2
print("Initialement :")
print(f'x "contient" : {x}')
print(f'y "contient" : {y}')
'''
Ecrire ici les instructions qui permettent
d'echanger le contenu des variables x et y
'''
print("Apres echange :")
print(f'x "contient" : {x}')
print(f'y "contient" : {y}')
```

Compléter ce programme :

1. À l'aide d'une variable intermédiaire (que l'on pourra nommer `z`).
2. Sans variable intermédiaire, à l'aide de l'affectation multiple.

Remarque : Les exercices qui suivent doivent être réalisés ***sans utiliser*** les instructions `while` et `for` (qui seront vues ultérieurement).

Exercice 1.10

Écrire un programme qui demande à l'utilisateur un entier L et qui affiche une ligne horizontale d'étoiles de longueur L. Exemple de fonctionnement du programme :

```
Longueur de la ligne horizontale ? 7
*******
```

Exercice 1.11

Écrire un programme qui demande à l'utilisateur un entier C et qui affiche une ligne verticale d'étoiles de longueur C. Exemple de fonctionnement du programme :

```
Longueur de la ligne verticale ? 2
Colonne :
*
*
```

Exercice 1.12

Écrire un programme qui demande à l'utilisateur deux entiers L et C, et qui affiche un rectangle d'étoiles de L lignes et C colonnes. Exemple de fonctionnement du programme :

```
Combien de lignes ? 3
Combien de colonnes ? 5
Rectangle :
*****
*****
*****
```

Exercice 1.13

Écrire un programme qui demande à l'utilisateur deux entiers L et C, et qui affiche un cadre d'étoiles de L lignes et C colonnes. Exemple de fonctionnement du programme :

```
Combien de lignes ? 4
Combien de colonnes ? 10
Cadre :
**********
*        *
*        *
**********
```

2 – Fonctions

2.1 – Définition et utilisation d'une fonction

Une fonction est une suite d'instructions qui dépend d'une liste de paramètres. Cette fonction pourra être « appelée » par la suite à chaque fois que l'on veut exécuter le bloc d'instruction correspondant.

Pour définir une fonction en Python, on utilise le mot clé `def` et la syntaxe suivante :

```
def <nom de la fonction>(<liste de paramètres>):
    <bloc d'instructions>
```

Pour appeler par la suite la fonction, on écrira : `<nom de la fonction>(<liste de paramètres>)`.

Exemple 2.1.

Écrire le programme suivant puis l'exécuter.

Norme d'encodage des caractères

Imports nécessaires

Définition des fonctions

Le programme proprement dit

```
# -*- coding: utf-8 -*-
#########################
from math import log
#########################
def f1(x):
    log(x**2+1)

def f2(x):
    print(log(x**2+1))

def f3(x):
    return log(x**2+1)

def hello(prenom):
    print(f"Bonjour {prenom} !")
    return None
#########################
var=input("Quel est votre prenom ? ")
hello(var)
```

Notes

- Les deux points à la fin de la première ligne des définitions des fonctions (lignes avec le mot clé `def`). Ces deux points sont **obligatoires**.
- Noter l'indentation des blocs d'instructions dans les fonctions : il y a exactement 4 espaces avant ; cette indentation est **obligatoire**. En Python, la « séparation » des blocs d'instructions se fait par l'indentation. Par exemple, ci–dessus Python sait que la définition de la fonction `hello` est finie dès que l'on écrit l'instruction « `var=`... » en début de ligne sans indentation (la ligne de « `#` » est une ligne de commentaires et n'est donc pas prise en compte).

  Lors du retour à la ligne, l'indentation se fait automatiquement dans l'éditeur `spyder`. (Si après votre première ligne de définition d'une fonction, vous oubliez les deux points, l'éditeur va normalement le détecter et les ajouter tout seul ...)
- Par défaut dans l'éditeur la touche  correspond à 4 espaces.

Organisation de vos programmes

Noter la structure du programme ci–dessus. Ce programme est « découpé » en plusieurs morceaux :

1. Une ligne pour l'encodage des caractères.
2. Les imports nécessaires au programme.
3. Les définitions des fonctions.
4. Le programme proprement dit.

Vous devrez utiliser le même « découpage » pour tous vos futurs programmes.

2.2 – L'instruction « return »

L'instruction `return` permet à la fonction de retourner un résultat (ou plusieurs). Si une fonction ne comporte pas l'instruction `return`, par défaut elle retournera `None`.

Il est conseillé d'écrire tout de même l'instruction `return None` dans le cas où on construit une fonction ne renvoyant rien (on sait comme cela que c'est intentionnel par exemple). Quand on écrit des fonctions qui ne retournent pas de valeurs, on parle souvent de **procédures** au lieu de fonctions.

L'instruction `return` provoque une sortie de la fonction (et la « fermeture de son espace local »). Ainsi tout le code qui suit le `return` dans la fonction ne sera pas exécuté. Il est donc possible d'avoir plusieurs `return` dans une fonction (par exemple pour faire un retour différent suivant la valeur d'une condition comme la fonction « `exam` » de l'exemple 3.6 page 29 au chapitre suivant).

Dans l'exemple 2.1 page 21, les définitions des fonctions `f1`, `f2` et `f3` peuvent sembler similaires au premier abord. En fait il n'en est rien, elles ont des « comportements » totalement différents.

Comme après avoir exécuté un programme il est possible de continuer à se servir des fonctions définies dans la console interactive, le comportement des différentes fonctions peut être testé comme dans l'exemple ci-dessous.

Exemple 2.2.

Après avoir exécuté le programme de l'exemple 2.1, saisir (et exécuter) les instructions suivantes dans la console interactive :

```
In [1]: a=3
In [2]: A1=f1(a)
In [3]: A2=f2(a)
In [4]: A3=f3(a)
In [5]: print(A1)
In [6]: print(A2)
In [7]: print(A3)
In [8]: A1
In [9]: A2
In [10]: A3
```

On peut constater que les fonctions `f1` et `f2` ne retournent aucun résultat (les variables `A1` et `A2` contiennent `None`). Ces fonctions ne servent donc, à priori, « à rien ». On ne pourra pas récupérer le résultat du calcul effectué et l'utiliser dans une autre expression. Ce comportement est normal puisque ces fonctions ne contiennent aucun `return`, c'est comme si l'on avait ajouté en dernière ligne du bloc de ces fonctions `return None`.

Par contre, on remarque que l'instruction numéro 3 provoque l'affichage d'un résultat (comme demandé dans la fonction à l'aide de `print`). Même si l'on visualise ce résultat, on ne pourra malheureusement pas l'exploiter (voir ci-dessus).

Enfin, la fonction `f3` a le comportement que l'on souhaite à priori. Elle se comporte comme une fonction mathématique : pour une valeur donnée en entrée, elle nous retourne un résultat. On pourra donc utiliser cette fois le résultat de la fonction dans un autre calcul. Le retour de la valeur se fait avec l'instruction `return`.

En résumé :

Attention à ne pas confondre `return` et `print` (erreur courante au début).
La confusion peut provenir du mode interactif de la console qui, quand on appelle une fonction, affiche le retour de celle–ci. Néanmoins, même dans la console en mode interactif on peut distinguer les deux « affichages ». L'affichage provoqué par le retour d'une fonction est précédé de « `Out [n]:` », ce qui n'est pas le cas lorsqu'on utilise la fonction `print`.

- La fonction `print` ne renvoie pas de valeur (elle ne retourne rien), elle permet seulement d'afficher quelque chose à l'écran.
- L'instruction `return` n'affiche rien, mais définit ce qui est renvoyé par la fonction.

Si on utilise la fonction `print` au lieu de l'instruction `return` dans une fonction, celle-ci affichera le résultat mais renverra `None` et le résultat affiché ne pourra donc pas être récupéré pour faire d'autres calculs.

En général, on n'utilisera pas de `print` dans une fonction, sauf par exemple pour une mise au point afin de trouver une erreur, ou pour des fontions spécifiques d'affichage ou nécessitant un dialogue avec l'utilisateur.

Attention, une fonction informatique n'est pas tout à fait une fonction mathématique dans le sens où elle peut ne pas avoir de variable, ou **retourner plusieurs valeurs** (il faut pour cela les séparer par des virgules après l'instruction `return`, voir exemple 2.3 ci-dessous) !

Exemple 2.3.

Exemple de fonction retournant plusieurs valeurs.

```
def separe(valeur):
    partie_entiere=int(valeur)
    partie_decimale=valeur-partie_entiere
    return partie_entiere,partie_decimale

x=float(input("Donnez une valeur numerique : "))

E,D=separe(x)

print("La partie entiere de {} est {}".format(x,E))
print("La partie decimale de {} est {}".format(x,D))
```

Notes

- Les noms de fonctions suivent les mêmes règles que les noms de variables.
- La définition d'une fonction porte bien son nom ... Elle ne fait que définir la fonction, elle ne l'exécute pas.
- La liste des paramètres donnés à la fonction s'appelle les paramètres d'entrée (ou parfois simplement entrées) et la (ou les) valeur(s) retournée(s) par la fonction s'appelle la sortie.

2.3 – Fonctions lambda

Pour une fonction qui retourne un simple calcul comme la fonction `f3`, il existe une façon plus rapide de l'écrire en Python : il s'agit des fonctions `lambda`. Le mot clé `lambda` permet de définir une fonction comme on le ferait en mathématiques en donnant pour un élément quelconque son image.

« $f : x \mapsto$ (expression de x) » s'écrira en Python : « `f = lambda x: (expression de x)` ».

On aurait pu définir la fonction `f3` de manière équivalente par `f3 = lambda x: log(x**2+1)`.

2.4 – Espace local et portée des variables

Quand Python exécute une fonction, il crée en mémoire un **espace local** spécifique et dédié à cette fonction.

Quand on définit une variable dans une fonction, celle–ci sera définie dans cet espace réservé. À la fermeture de la fonction (i.e. juste après l'instruction `return ...`, ou la dernière instruction de la fonction), cet espace est « détruit » avec toutes les variables qu'il contient.

Dans la fonction, le mécanisme utilisé par Python est le suivant :

- si on assigne une valeur à une variable non déjà définie dans l'espace local, cette variable est alors créée dans l'espace local ;
- si on fait référence à une variable, Python vérifie si la variable existe dans l'espace local. Sinon, Python va regarder dans l'espace dans lequel la fonction a été appelée, ...

Les paramètres d'entrée sont considérés comme des variables locales à la fonction. Comme les variables locales sont détruites à la fermeture de la fonction, il n'est donc « pas possible », dans une fonction, de modifier par affectation la valeur d'une variable extérieure à son espace local ...

Notes

- Une fonction peut prendre tout type d'objet en paramètre, y compris une autre fonction.
- On peut aussi définir une fonction dans une fonction ... (La fonction définie localement sera, comme les autres variables, détruite à la fermeture de l'espace de la fonction dans laquelle elle est définie.)
- La notion de variable locale est similaire à celle de variable muette en mathématiques.

Exemple 2.4.

1. Récupérer le programme `fonctions.py` sur moodle et l'exécuter.
   (Le contenu de ce programme est donné page 25.)
2. Taper ensuite dans la console les commandes suivantes (et bien comprendre tout ce qui se passe) :

```
In [1]: x=3
In [2]: reinitialise1()
In [3]: print(x)
In [4]: reinitialise2(x)
In [5]: print(x)
In [6]: f(x)
In [7]: print("Après la fonction, x={}".format(x))
In [8]: fct2(7)
In [9]: fct3(257)
In [10]: f=lambda t: 2*t+3
In [11]: u(f,x) # On peut passer une fonction en paramètre
```

Contenu du programme `fonctions.py` :

```
from math import exp

def reinitialise1():
    x=0
    return None

def reinitialise2(x):
    x=0
    return None

def f(x):
    print(f"Dans la fonction f, avant modification, x={x}")
    x=x*2
    print(f"Dans la fonction f, apres modification, x={x}")
    return None

def fct1(x):
    return 100*x

def fct2(x): # fct2 fait appel a une autre fonction fct1 definie dans ce programme
    return x+fct1(x)

def fct3(x):
    def z(y): # Fonction locale
        return y/1000
    return z(x)+x

def u(v,x): # On peut passer une fonction en parametre
    return exp(v(x))
```

2.5 – Exercices

Exercice 2.1 *(Incontournable)*

On considère les deux fonctions Python définies ci-dessous :

```
g1 = lambda x: (x**2, x**3)
g2 = lambda x,n: round(x**0.5,n)
```

Écrire ces deux fonctions à l'aide de la définition standard des fonctions (`def g1...`, `def g2...`).

Exercice 2.2 *(Incontournable)*

On considère les deux fonctions Python définies ci-dessous :

```
def f1(entier):
    carac=str(entier)
    nombre=len(carac)-1
    return nombre
```

```
def f2(chaine):
    m=len(chaine)//2
    centre=chaine[m:m+2]
    return centre
```

Écrire ces deux fonctions à l'aide des fonctions `lambda`.

Exercice 2.3

1. Écrire une fonction `demi` qui prend en argument une chaîne de caractères et qui renvoie la première moitié de celle–ci (on arrondira à l'entier inférieur).
2. Écrire une fonction `censure` qui prend en argument une chaîne de caractères et qui renvoie une autre chaîne où tous les caractères sauf le premier et le dernier ont été remplacés par « # ».
3. Écrire une fonction `bidon` qui prend en argument une chaîne de caractères et renvoie le premier et le dernier caractère de la chaîne, ainsi que sa longueur.
4. Faire quelques tests pour vérifier vos fonctions.

Exercice 2.4

Écrire une fonction `f` qui prend 2 réels x et y en entrée et qui retourne la valeur : $f(x,y) = x^2 + y^2 - \lfloor x+y \rfloor$

Exercice 2.5

1. Écrire une fonction `derive(f,x)` qui retourne une valeur approchée de la dérivée de la fonction f en x. On utilisera la formule suivante : $f'(x) \approx \dfrac{f(x+10^{-6}) - f(x)}{10^{-6}}$.
   (Pour rappel $f'(x) = \lim\limits_{h\to 0} \dfrac{f(x+h)-f(x)}{h}$.)
2. Définir la fonction `carre(x)` qui retourne la valeur x^2 pour un réel x donné en entrée, puis tester votre fonction `derive` avec la fonction `carre` et des valeurs de `x` de votre choix.

Exercice 2.6 ★

1. Écrire une fonction `logfx(u,x)` qui retourne la valeur $\ln(u(x))$ pour une fonction u et un réel x donnés en entrée.
2. Définir la fonction `carre(x)` qui retourne la valeur x^2 pour un réel x donné en entrée, puis tester votre fonction `logfx` avec la fonction `carre` et des valeurs de `x` de votre choix.
3. Écrire une fonction `logf(u)` qui retourne la **fonction** $\ln(u)$ pour une fonction u donnée en entrée.
4. Tester votre fonction `logf` avec la fonction `carre` en calculant les images des valeurs de votre choix. (Par exemple : `logf(carre)(1)`, `logf(carre)(3)`.)

Exercice 2.7 *(Facultatif)*

1. On donne ci-dessous la fonction `somme(f,g)` qui retourne la fonction $f+g$ pour 2 fonctions f et g données en entrée.

```
def somme(f,g):
    s=lambda x : f(x)+g(x)
    return s
```

   (a) Écrire la fonction Python « `i` » correspondant à la fonction identité ($i : x \mapsto x$).
   (b) Quelle est la fonction « `d` » définie par `d=somme(i,i)` ?
   Valider votre réponse à l'aide de quelques tests. (Par exemple, `d(0)`, `d(1)`, `d(5)`, `d(1.3)`, ...)
2. Écrire les fonctions Python « `f1` » et « `f2` » correspondant respectivement aux fonctions $x \mapsto \frac{x}{2}$ et $x \mapsto 2x$.
3. (a) Écrire une fonction `produit(f,g)` qui retourne la fonction $f \times g$ pour 2 fonctions f et g données en entrée de deux façons différentes (avec `lambda` et avec `def`).
   (b) Tester la fonction `produit` à l'aide des fonctions `f1` et `f2`. Vous pourrez pour cela soit définir une nouvelle fonction, soit utiliser directement la fonction `produit(f,g)`.
   Exemples :
   - `h=produit(f1,f2)` ; puis `h(0)`, `h(2)`, `h(5)`, ...
   - ou directement : `produit(f1,f2)(0)`, `produit(f1,f2)(2)`, `produit(f1,f2)(5)`, ...
4. Écrire une fonction `composee(f,g)` qui retourne la fonction $f \circ g$ pour 2 fonctions f et g données en entrée.
   Une fois la fonction définie, la tester avec les fonctions `f1` et `f2`, puis avec les fonctions `log` et `exp`.

3 – Structures de contrôle

3.1 – Instructions conditionnelles

3.1.1 – Test simple

Une instruction conditionnelle n'est exécutée que si une condition donnée est vérifiée par l'état courant.

```
1 si Condition;
2 alors
3 |  Bloc d'instructions;
```

Le « bloc d'instructions » ne sera exécuté que si « Condition » est vraie. (Condition doit donc être une expression booléenne.) En Python, cela s'écrit :

```
if condition :
    Instructions
```

Notes

Comme pour les fonctions, on peut noter 2 points particulièrement importants :

- Les deux points à la fin de la ligne `if`.
- L'indentation de la ligne avec « Instructions » : il y a exactement 4 espaces avant. Comme pour les fonctions, lors du retour à la ligne l'indentation se fait automatiquement dans l'éditeur `spyder`. (Si vous oubliez les deux points à la fin de la ligne, l'éditeur va normalement le détecter et les ajouter tout seul ...)

Exemple 3.1.

- Dans la console, il faudra appuyer une deuxième fois sur entrée après avoir validé la dernière instruction pour signifier à Python la fin du bloc.

```
In [1]: x=1234
In [2]: if x%2==0 :
   ...:     print(f"{x} est pair")
   ...:
1234 est pair
```

- Dans l'éditeur (enregistrer votre programme sous le nom `estPair.py` par exemple) :

```
x = int(input('Donnez un nombre entier : '))
if x%2==0 :
    print(f"{x} est pair")
```

3.1.2 – Bloc d'instructions et indentation

Rappel et complément de la deuxième note page 21 :
Un bloc d'instructions, ou tout simplement bloc, est une suite d'instructions qui s'exécutent les unes à la suite des autres. En Python, pour identifier les instructions d'un bloc donné, il est nécessaire (et suffisant) de les indenter au **même niveau**.

Exemple 3.2.

Écrire, sauvegarder puis exécuter le script suivant en le testant avec une valeur positive, zéro et une valeur négative :

```
# -*- coding: utf-8 -*-
x = int(input('Donnez un nombre entier : '))
if x>=0 :
    #debut du bloc1
    print("***")
    print("C'est un entier positif ",end="")
    if x==0 :
        #debut du bloc2
        print("ou nul",end="")
        #fin du bloc2
    print("\n***")
    #fin du bloc1
print("Au revoir.")
```

3.1.3 – Test avec alternative

Pour enrichir l'instruction de sélection `if`, on peut spécifier une alternative lors d'un test :

```
si Condition;
alors
    Instructions si Condition est vraie;
sinon
    Instructions si Condition est fausse;
```

En Python, cela s'écrit :

```
if Condition :
    Instructions si vraie
else :
    Instructions si fausse
```

Exemple 3.3.

```
if N>=10 :
    print("Vous êtes reçu")
else :
    print("Vous êtes collé")
```

Note

Bien noter que l'utilisation de l'alternative (le « `else` ») est **facultative** (voir le test simple ci-dessus).

3.1.4 – Tests imbriqués

Il est possible de faire un test à l'intérieur d'un autre test (attention à l'indentation) :

Exemple 3.4.

```
# -*- coding: utf-8 -*-
x = int(input('Donnez un nombre : '))
if x==0 :
    print("Le nombre est nul")
else :
    if x>0 :
        print("Le nombre est strictement positif")
    else :
        print("Le nombre est strictement negatif")
```

Il est possible d'écrire les différentes instructions de test au même niveau pour améliorer la lisibilité à l'aide du mot « `elif` » (contraction de `else` et `if`, « sinon si ») :

Exemple 3.5.

```
# -*- coding: utf-8 -*-
x = int(input('Donnez un nombre : '))
if x==0 :
    print("Le nombre est nul")
elif x>0 :
    print("Le nombre est strictement positif")
else :
    print("Le nombre est strictement negatif")
```

Un exemple complet avec une fonction (disponible sous moodle : `examen.py`) :

Exemple 3.6.

```
# -*- coding: utf-8 -*-
#########################
def exam(n):
    if n>=10:
        return 'recu'
    elif n>=8:
        return 'au rattrapage'
    else:
        return 'colle'
#########################
note=float(input("Donnez votre note a l'examen : "))
resultat=exam(note)
print(f'Vous etes {resultat}.')
```

Noter la présence de multiples « `return` » comme évoqué dans la section 2.2 du chapitre précédent.
Pour rappel :

L'instruction `return` provoque une sortie de la fonction (et la « fermeture de son espace local »). Ainsi tout le code qui suit le `return` dans la fonction ne sera pas exécuté. Il est donc possible d'avoir plusieurs `return` dans une fonction (par exemple pour faire un retour différent suivant la valeur d'une condition).

Tester ce programme et vérifier la sortie de la fonction au premier « `return` » rencontré.
(On pourra tester les valeurs 5, 9 et 12.)

3.2 – Boucles et itérations

On considère la suite numérique réelle définie par $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \cos(u_n)$. On souhaite écrire un programme Python permettant de calculer u_n pour un n donné. On va donc devoir exécuter ici une instruction du type « `u=cos(u)` » *un certain nombre de fois* (n) (ou bien, *tant que* une condition est vraie). On parle alors de **boucle**. On appelle **itération** chaque exécution du bloc d'instruction, ou passage dans la boucle.

3.2.1 – Boucles inconditionnelles

Si l'on connaît à l'avance le nombre d'itérations qui doivent être effectuées on utilise une **boucle inconditionnelle** ou **boucle déterminée**. Il faut tout simplement indiquer combien de fois le bloc d'instruction est à exécuter. Pour le répéter n fois, on pourra écrire :

```
1 pour i de 1 à n;
2  faire
3  | Bloc d'instructions;
```

Ainsi, pour déterminer u_n pour un n donné, on écrira en langage naturel :

```
1 n ←?;
2 u ← 1;
3 pour i de 1 à n;
4  faire
5  | u ← cos(u);
6 Afficher u
```

Exemple 3.7.

En Python le *pour* s'écrit « `for` ». L'algorithme ci–dessus s'écrira :

```
from math import cos
n=int(input('Donner le rang du terme a calculer : '))
u=1
for i in range(n):
    u=cos(u)
print(f"Le terme de rang {n} vaut environ {u}")
```

Note

L'instruction « `for i in range(n)` » fait prendre à la variable `i` les valeurs de 0 à $n-1$ (et non de 1 à n). Cela ne change en rien notre problème ici, la boucle est toujours parcourue n fois. Mais cela peut avoir une incidence si l'on utilise la variable `i` dans le bloc d'instructions.
Si l'on souhaite faire varier `i` de a à b **inclus** (a et b entiers), on écrira « `for i in range(a,b+1)` ». Si la valeur a n'est pas précisé, Python commence par défaut à 0. Ainsi « `for i in range(n)` » est équivalent à « `for i in range(0,n)` ».

3.2.2 – Boucles conditionnelles

On considère maintenant la suite numérique réelle définie par $v_0 = 1.1$ et $\forall n \in \mathbb{N}$, $v_{n+1} = v_n^2$. On souhaite écrire un programme Python permettant de calculer le rang n à partir duquel $v_n > 10$. (Problème de valeur seuil abordé en terminale.)

On ne connaît pas à l'avance le nombre de boucles à effectuer (ou, autrement dit, l'arrêt de la boucle dépend d'une condition), on parle alors de **boucle conditionnelle** ou **boucle indéterminée**. On écrira :

```
1 tant que Condition;
2  faire
3  | Bloc d'instructions;
```

Ainsi, pour déterminer le rang n à partir duquel $v_n > 10$, on écrira en langage naturel :

```
1 v ← 1.1;
2 n ← 0;
3 tant que v ⩽ 10;
4  faire
5  |  v ← v²;
6  |  n ← n + 1;
7 Afficher n
```

Exemple 3.8.

En Python le *tant que* s'écrit « `while` ». L'algorithme ci–dessus s'écrira :

```
v=1.1
n=0
while v<=10:
    v=v**2
    n=n+1
print(f"La suite dépasse 10 à partir du rang {n}")
```

Note

Toute boucle inconditionnelle peut s'écrire sous la forme d'une boucle conditionnelle en introduisant une variable qui servira de compteur (et qui sera incrémentée à chaque itération).

Exemple 3.9.

Le programme de l'exemple 3.7 peut s'écrire à l'aide d'une boucle « `while` » :

```
from math import cos
n=int(input('Donner le rang du terme a calculer : '))
u=1
compteur=0
while compteur<n:
    u=cos(u)
    compteur=compteur+1
print(f"Le terme de rang {n} vaut environ {u}")
```

Terminaison de la boucle

Il faut bien vérifier que la boucle se finit. C'est à dire que le corps de la boucle modifie la condition à certaines itérations et que ces modifications finiront par rendre la condition fausse.

Ici, c'est bien le cas : la variable `compteur` commence à 0 et augmente de 1 à chaque passage dans la boucle. Elle finira donc nécessairement par être supérieure ou égale à n.

Si malgré vos vérifications votre programme semble parti dans une boucle infinie, vous pouvez l'interrompre de force avec Ctrl + C .

3.3 – Exercices

Exercice 3.1 *(Incontournable)*

1. Que permet de calculer le programme suivant ?

```
if a>=b:
    if b>=c: m=b
    elif a>=c: m=c
    else: m=a
else :
    if a>=c: m=a
    elif b>=c: m=c
    else: m=b
```

2. Réaliser plusieurs tests pour vérifier votre réponse.

Exercice 3.2 *(Incontournable)*

Écrire deux programmes permettant d'afficher les carrés des entiers de 2 à 20 : un avec une boucle `for` et un avec une boucle `while`.

Exercice 3.3 *(Incontournable)*

Écrire un programme permettant d'afficher la table de multiplication de n. On affichera les résultats jusqu'au produit par m. n et m seront initialisées en début de programme.

Pour $n = 2$ et $m = 5$, le résultat sera le suivant :

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

Exercice 3.4 *(Incontournable)*

Écrire un programme permettant d'afficher les tables de multiplication de 2 à n. Pour chaque table on affichera les résultats jusqu'au produit par m. n et m seront initialisées en début de programme. Vous écrirez deux versions de votre programme pour obtenir les affichages suivants (obtenus avec $n = 3$ et $m = 4$) :

(1)

```
Table de 2
----------
2 x 1 = 2
----------
2 x 2 = 4
----------
2 x 3 = 6
----------
2 x 4 = 8

Table de 3
----------
3 x 1 = 3
----------
3 x 2 = 6
----------
3 x 3 = 9
----------
3 x 4 = 12
```

(2) Pour obtenir ce deuxième affichage, vous ne ferez qu'une seule modification à votre programme précédent : vous ne ferez que déplacer votre instruction `print("----------")`.

```
Table de 2
----------
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8

Table de 3
----------
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
```

Exercice 3.5

L'objectif de l'exercice est d'écrire deux fonctions qui pour un entier naturel n donné en paramètre déterminent respectivement : (a) $n!$ (b) $\sum_{k=0}^{n} k$

1. Écrire (en langage « naturel ») les deux algorithmes permettant d'obtenir les résultats demandés.
2. Écrire les deux fonctions demandées.
3. Proposez des tests pour vérifier vos fonctions (et faites les ...).

Exercice 3.6

Pour un examen donné :

- On est reçu si la moyenne est supérieure ou égale à 10.
- On obtient une mention « Assez Bien » entre 12 inclus et 14 exclu, « Bien » entre 14 inclus et 16 exclu, « Très Bien » à partir de 16.
- En plus de la mention « Très Bien », on reçoit les félicitations du jury si la moyenne est supérieure ou égale à 18.

1. Écrire une fonction « `mention(note)` » qui retourne la mention correspondant à la note. (C'est à dire : `""`, `"Assez Bien"`, `"Bien"`, `"Très Bien"` ou `"Très Bien avec félicitation du jury"`.)

   Vous n'utiliserez qu'une seule comparaison pour chaque test « `if` » ou « `elif` » effectué. Par exemple, pour cet exercice, « `12<=m<14` » ou « `12<=m and m<14` » **ne sont pas autorisés**.
2. Écrire un programme qui demande à l'utilisateur sa moyenne et qui affiche s'il est reçu (ou non) et dans le cas où il est reçu sa mention éventuelle. (Utiliser la fonction « `mention` ».)
3. Proposez des tests pour vérifier votre programme (et faites les ...).

Exercice 3.7

1. Écrire une fonction « `ppcm(a,b)` » qui calcule et retourne le plus petit commun multiple de deux entiers naturels `a` et `b`.

   Pour cette fonction **vous utiliserez l'algorithme « naïf » suivant** : On teste **tous les entiers** à partir de 1 et **jusqu'à ce qu**'on trouve un multiple de `a` et de `b`. Remarque : l'algorithme proposé s'arrêtera nécessairement, dans le pire des cas, pour l'entier `a`×`b`.
2. Écrire un programme demandant deux entiers naturels `a` et `b` à l'utilisateur et affichant leur plus petit commun multiple (en utilisant votre fonction `ppcm`).
3. Proposez des tests pour vérifier votre programme (et faites les ...).

Exercice 3.8

Écrire une fonction `pgcd` qui prend en argument deux entiers et retourne leur PGCD à l'aide de l'algorithme d'Euclide.

Exercice 3.9

1. Écrire un programme donnant le nombre d'occurrences de la lettre `a` dans une chaîne de caractères que vous initialiserez en début de programme. Par exemple pour la chaîne de caractères `Happy new year`, le programme devrait donner 2.
2. Écrire un programme donnant le nombre d'occurrences du chiffre 1 (dans l'écriture avec le système décimal) dans un entier N que vous initialiserez en début de programme. (Vous pourrez convertir N en un autre type de variable ...) Par exemple pour l'entier 21561141, le programme devrait donner 4.
3. Écrire un programme donnant le nombre total d'occurrences du chiffre 1 dans les nombres de 1 à N. Quel est le résultat obtenu pour $N = 999$? pour $N = 9999$?

Exercice 3.10

La suite de Fibonacci est définie par : $u_0 = 0$, $u_1 = 1$ et pour tout $n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.

Pour un entier naturel n donné en entrée, écrire une fonction retournant le $n^{\text{ième}}$ terme de la suite de Fibonacci à l'aide d'une boucle.

Exercice 3.11

Écrire des fonctions qui affichent les triangles de caractères « `c` » de taille n suivant les modèles donnés dans l'exemple ci–dessous. n et `c` seront donnés en entrée des fonctions (et celles-ci ne **retourneront** rien).

Exemples :

- Modèle 1 :

```
In [1]: triangle1("#",4)
####
###
##
#
```

- Modèle 3 :

```
In [1]: triangle3("*",4)
*
**
***
****
***
**
*
```

- Modèle 2 :

```
In [1]: triangle2("*",6)
     *
    **
   ***
  ****
 *****
******
```

- Modèle 4 :

```
In [1]: triangle4("+",5)
    +
   +++
  +++++
 +++++++
+++++++++
```

Compléments/Rappels sur les nombres complexes

- Le nombre complexe $a + ib$ avec a et b réels s'obtient avec la fonction `complex` : `complex(a,b)`.
  Le type d'une expression complexe est `complex`. Sa partie réelle et sa partie imaginaire sont de type `float`.
- Le nombre complexe i s'écrit `1j` en Python. De manière plus générale une valeur numérique suivie de `j` représente un nombre complexe de partie réelle 0 et de partie imaginaire cette valeur numérique. Par exemple, `5j` représente $5i$. Attention à ne pas mettre d'espace entre la valeur numérique et `j`.
  On peut donc obtenir le nombre complexe $a + ib$ d'au moins 2 façons : `complex(a,b)` ou `a+1j*b`.
- Quelques « fonctions » sur les complexes :
  - ◇ `abs` permet d'obtenir la norme d'un nombre complexe. Exemple : `abs(complex(1,1))`.
  - ◇ `phase` permet d'obtenir l'argument en radian d'un nombre complexe (dans l'intervalle $[-\pi;\pi]$).
    Exemple : `phase(complex(1,1))`.
    Note : Pour pouvoir utiliser la fonction `phase`, il faut d'abord l'importer de la bibliothèque `cmath` (avec `from cmath import phase`).
  - ◇ `real` et `imag` permettent d'obtenir respectivement la partie réelle et la partie imaginaire d'un nombre complexe (attention, ce ne sont pas des fonctions).
    Exemple : `complex(1,2).real`, `complex(1,2).imag`.
  - ◇ `conjugate()` permet d'obtenir le conjugué d'un nombre complexe.
    Exemple : `complex(1,2).conjugate()`.

Exercice 3.12 *(Incontournable)*

1. Écrire une fonction « `degre2(a,b,c)` » qui calcule et retourne la ou les racine(s) du polynôme de degré 2 de $\mathbb{R}_2[X]$: $aX^2 + bX + c$ (pour $a \neq 0$, b et c réels).
2. Écrire un programme demandant à l'utilisateur les coefficients a, b, c et affichant les racines du polynôme correspondant en utilisant votre fonction.
3. Réalisez des tests pour vérifier votre programme.
4. Modifier votre fonction pour qu'elle prenne aussi en compte les polynômes de degré 1 ou constants.

Exercice 3.13 *(Facultatif)*

Écrire une fonction « `degre2C(a,b,c)` » qui calcule et retourne la ou les racine(s) du polynôme de degré 2 de $\mathbb{C}_2[X]$: $aX^2 + bX + c$ (pour $(a, b, c) \in \mathbb{C}^3$).

Exercice 3.14 *(Facultatif)*

1. Écrire une fonction qui donne l'ordre de grandeur d'un nombre $x \neq 0$ donné en entrée, c'est–à–dire l'entier $n \in \mathbb{Z}$ tel que $10^n \leqslant |x| < 10^{n+1}$. (Sans utiliser les fonctions logarithmiques du module `math`.)
2. Proposez des jeux de tests pour vérifier votre fonction (et faites les tests ...). Vous pouvez, pour vos tests, utiliser les fonctions logarithmiques du module `math`.

Exercice 3.15 ★

1. Écrire une fonction `somme_f` qui prend 2 paramètres n (entier) et f (fonction) en entrée et qui renvoie la valeur $\displaystyle\sum_{k=1}^{n} f(k)$.
2. Écrire une fonction `indice_somme` qui qui prend 2 paramètres M (entier) et f (fonction) en entrée et qui renvoie le plus petit des entiers non nuls n tel que $\displaystyle\sum_{k=1}^{n} f(k) \geqslant M$.
3. Tester le fonctionnement de vos deux fonctions `somme_f` et `indice_somme` avec la fonction $x \to x^2$. Pour n vous pourrez essayer les valeurs 1, 2, 3, 4, ... et pour M les valeurs 0, 1, 5, 6, 10, 14, 15, 200, ...
   Pour vérifier les résultats et pour rappel : $1^2 + 2^2 + \ldots + n^2 = \displaystyle\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$.
4. Écrire un programme qui affiche les plus petits des entiers non nuls n tel que $\displaystyle\sum_{k=1}^{n} f(k) \geqslant M$ dans les cas suivants :
   (a) f définie par $f(x) = \dfrac{1}{x}$ et pour chaque valeur de M dans $\{2,\ 3,\ 4,\ 5\}$
   (b) f définie par $f(x) = \dfrac{5}{x^2 + x}$ et pour chaque valeur de M dans $\{2,\ 3,\ 4,\ 5\}$
      i. Dans ce cas, que constate-t-on pour $M = 5$?
      ii. Vérifier que $\dfrac{5}{x} - \dfrac{5}{x+1} = \dfrac{5}{x^2 + x}$, puis en déduire une expression simplifiée de $\displaystyle\sum_{k=1}^{n} f(k)$.
      iii. Expliquer alors le comportement du programme pour $M = 5$.

4 – Types composés (séquences)

4.1 – Les séquences

Une séquence en Python est un objet (une structure de données) composé d'autres objets de type simple ou non (par exemple des entiers, flottants, ... ou aussi d'autres séquences). D'autre part les objets contenus dans la séquence sont ordonnés. Une séquence est donc similaire à un ensemble fini et ordonné d'éléments ; si la séquence est composée de n éléments, ceux-ci seront indicés de 0 à $n-1$. Parmi ces séquences on trouve les chaînes de caractères, les n-uplets et les listes. Ces trois types d'objets étant des séquences, leur structure est similaire et ils partagent donc un certain nombre de fonctionnalités.

Les instructions communes à ces séquences sont données dans cette section, puis les spécificités de chaque type sont abordées dans les sections suivantes. Pour l'accès aux éléments on va notamment retrouver **la même syntaxe** que celle qui a été vue pour les chaînes de caractères dans la section 1.5.2 page 13.

Dans tous les exemples suivants `variable_seq` représente une séquence (de type chaîne de caractères, n-uplet ou liste). En attendant de voir plus en détail les n-uplets ou les listes, vous pouvez essayer les instructions proposées avec une chaîne de caractère (par exemple `variable_seq="Bonjour"`), puis vous pourrez revenir plus tard dessus en essayant cette fois des n-uplets ou les listes.

D'autre part on considérera que la séquence `variable_seq` est de longueur n.

4.1.1 – Accès aux éléments

- On peut accéder à un élément d'une séquence à l'aide de son « index » : `variable_seq[index]`.

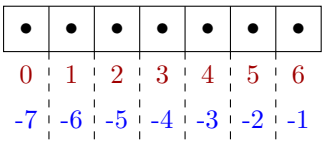
  ⚠ **Attention !** Le premier élément de la séquence `variable_seq` est indexé **par 0** ...

  Les éléments de la séquence `variable_seq` sont indexés de 0 à $n-1$.

- On peut utiliser des index négatifs. Le dernier élément de la séquence `variable_seq` peut s'obtenir avec l'index -1, l'avant dernier avec -2, etc., et le premier élément avec $-n$.

  Il est ainsi équivalent d'écrire :

  → `variable_seq[-1]` ou `variable_seq[n-1]`,
  → `variable_seq[-2]` ou `variable_seq[n-2]`,
  → ...,
  → `variable_seq[-n]` ou `variable_seq[0]`.



- « Sous–séquence » : On accède une sous–séquence (« tranche » ou « slice ») en découpant la séquence par l'intermédiaire des « index » de début et de fin. On utilise la syntaxe `variable_seq[debut:fin]` (ou `variable_seq[debut:fin:pas]` pour spécifier en plus un pas d'avancement, voir ci-dessous).

  ⚠ **Attention !** L'élément indexé par `fin` **est exclu** de la séquence.

  – Sous-séquence des éléments de la position a à b : `variable_seq[a:b]`.
  ⚠ Sous-séquence de a inclus à b **exclu**.
  – Sous-séquence des éléments de la position a à la fin : `variable_seq[a:]`.
  – Sous-séquence des éléments du début à la position b (**exclue**) : `variable_seq[:b]`.
  – Sous-séquence des éléments de la position a à b (b **exclu**) avec un pas de c : `variable_seq[a:b:c]`. C'est-à-dire les éléments en position « $p = a + k \times c$ » tels que $k \in \mathbb{N}$ et $p < b$. (La formule n'est pas correcte si l'on mélange index positif et négatif.)

4.1.2 – Quelques instructions & fonctions

- Le nombre d'éléments (la longueur) d'une séquence s'obtient avec `len` : `len(variable_seq)`
- L'opérateur `+` réalise une concaténation des séquences : On obtient une nouvelle séquence constituée des éléments de la première séquence suivis dans l'ordre par les éléments de la deuxième séquence.
- Somme des éléments de la séquence (si les éléments sont d'un type numérique) : `sum(variable_seq)`
- Minimum des éléments de la séquence (si les éléments sont d'un type numérique) : `min(variable_seq)`
- Maximum des éléments de la séquence (si les éléments sont d'un type numérique) : `max(variable_seq)`
- Appartenance à une séquence : `element in variable_seq`, retourne `True` si `element` est dans la séquence et `False` sinon.
- Non appartenance à une séquence : `element not in variable_seq`, retourne `True` si `element` n'est pas dans la séquence et `False` sinon.
- Nombre d'occurrences d'un élément dans une séquence : `variable_seq.count(element)`, retourne le nombre d'apparitions de `element` dans `variable_seq`.
- Position de la première occurrence d'un élément dans une séquence : `variable_seq.index(element)`.

Remarque

On appelle **méthode** une fonction associée à un objet. La syntaxe pour utiliser une méthode est généralement du type : `<nom de la variable>.<nom de la méthode>(<paramètres>)`.

4.2 – Les chaînes de caractères

Exemple 4.1. Instructions et fonctions « génériques »

```
In [1]: mot='Bonjour'
In [2]: print(mot[0])
In [3]: print(mot[3])
In [4]: print(mot[len(mot)-1])
In [5]: print(mot[-1])
In [6]: print(mot[-len(mot)])
In [7]: print(mot[0:1])
In [8]: print(mot[0:3])
In [9]: print(mot[0:len(mot)])
In [10]: print(mot[-2:len(mot)])
In [11]: print(mot[2:])
In [12]: print(mot[:4])
In [13]: "a" in mot
In [14]: "a" not in mot
In [15]: mot.count("o")
In [16]: mot.count("a")
In [17]: mot.index("o")
```

Les chaînes de caractères ne sont pas modifiables, on dit qu'elles **ne sont pas mutables** ou qu'elles sont **immuables**. Si l'on souhaite modifier une chaîne de caractères, on est obligé d'affecter le résultat de la modification à une variable.

Exemple 4.2.

```
In [18]: mot[0]="b" # on ne peut pas modifier une chaîne de caractères
In [19]: MOT2="b"+mot[1:] # il faut affecter le résultat à une variable
In [20]: print(MOT2)
In [21]: mot="b"+mot[1:] # mais on peut le réaffecter à la même variable
In [22]: print(mot)
```

4.2.1 – Quelques méthodes spécifiques

Les chaînes de caractères étant immuables, toutes les méthodes ci–dessous **ne modifient pas la chaîne de caractères d'origine**.

- On peut remplacer une sous–chaîne par une autre avec `replace`.

Exemple 4.3.

```
In [23]: "abcdefabcdefabcdef".replace("abc", "zoo")
In [24]: "abcdefabcdefabcdef".replace("abc", "zoo", 2)
In [25]: "les x qui disent y".replace("x", "chevaliers").replace("y", "Ni")
```

- On peut changer la casse (minuscule ou majuscule) des caractères.

Exemple 4.4.

```
In [26]: "monty PYTHON".upper()
In [27]: "monty PYTHON".lower()
In [28]: "monty PYTHON".swapcase()
In [29]: "monty PYTHON".capitalize()
In [30]: "monty PYTHON".title()
```

- Et enfin, on peut « nettoyer » une chaîne des caractères « inutiles » qui l'entourent avec `strip` (uniquement à gauche : `lstrip`, uniquement à droite `rstrip`).

Exemple 4.5.

```
In [31]: chaine = " une chaine avec des trucs qui depassent \n"
In [32]: chaine
In [33]: chaineEpuree = chaine.strip()
In [34]: chaineEpuree
In [35]: "***-Un texte-***".rstrip("*-")
In [36]: "***-Un texte-***".lstrip("*-")
In [37]: "xyzaxyz Un texte xyz finx".strip("xyz")
```

4.2.2 – Retour à la ligne et tabulation

Pour obtenir un retour à la ligne dans une chaîne de caractères, on utilise « `\n` ».
Pour une tabulation, on utilise « `\t` ».

Exemple 4.6.

```
In [38]: print("Ligne 1\nLigne2")
In [39]: print("Mot 1\tMot2 après une tabulation")
```

4.3 – tuples

Les n–uplets ou tuples en anglais peuvent être vus comme la traduction informatique d'un produit cartésien d'ensembles. Comme en mathématique, les tuples s'écriront comme une suite d'expressions entre **parenthèses** séparées par des virgules. Ces séquences d'objets peuvent être hétérogènes.

De même que pour les chaînes de caractères les tuples sont **immuables**, c'est–à–dire, on ne peut pas affecter de nouvelle valeur à une de ses composantes.

Il est possible de « déconstruire » un tuple en affectant simultanément ses composantes à différentes variables (comme cela a déjà été utilisé pour réaliser des affectations multiples).

Exemple 4.7. Instructions et fonctions « génériques »

```
In [1]: A=(1, 8, 3, 6, 0, 8, 2, 5, 4, 7)
In [2]: type(A)
In [3]: B=(31,"a",0.7,(8,9)) # Les composantes peuvent être de types différents
In [4]: print(A[0])
In [5]: print(B[3][1])
In [6]: print(A[-1])
In [7]: print(A[1:4])
In [8]: print(A[:3])
In [9]: print(A[3:])
In [10]: print(A[1:5:2])
In [11]: print(A[::-1])
In [12]: C=();print(C);print(type(C)) # tuple vide
In [13]: D=(1,);print(D);print(type(D)) # Singleton. /!\ La virgule est obligatoire
In [14]: E=(1);print(E);print(type(E)) # Sans virgule ...   ce n'est pas un tuple
In [15]: x,y,z=A[1:4] # Déconstruction
In [16]: print(x, y, z)
In [17]: print(len(A))
In [18]: print(sum(A))
In [19]: print(min(A))
In [20]: print(max(A))
In [21]: print(A+B)
In [22]: 1 in A
In [23]: 9 in A
In [24]: A.count(8)
In [25]: A.index(8)
```

Exercice 4.1  *(Incontournable)*

1. Écrire une fonction `moyenne` prenant en paramètre d'entrée un n–uplet et retournant la moyenne des valeurs de ce n–uplet.
2. Écrire une fonction `ecart` prenant en paramètre d'entrée un n–uplet et retournant l'écart entre la plus petite et la plus grande des valeurs.
3. Écrire une fonction `moyennegauche` prenant en paramètre d'entrée un n–uplet et retournant la moyenne des 3 premières valeurs (ou 0 si le n–uplet est composé de moins de 3 valeurs).
4. Écrire une fonction `moyennedroite` prenant en paramètre d'entrée un n–uplet et un entier `k` et retournant la moyenne des k dernières valeurs (ou 0 si le n–uplet est composé de moins de k valeurs).
5. Proposez des tests pour vérifier vos fonctions (et faites les ...).

4.4 – Les listes

De même que les n–uplets, les listes peuvent être vues comme la traduction informatique d'un « produit cartésien d'ensembles », peuvent être hétérogènes, et il est possible de les déconstruire de la même façon. Elles s'écriront comme une suite d'expressions entre **crochets** séparées par des virgules.

Contrairement aux chaînes de caractères et aux n–uplets, les listes sont **modifiables**, on dit qu'elles sont **mutables** (c'est la principale différence avec les n–uplets). C'est–à–dire qu'on peut affecter une nouvelle valeur à une de ses composantes. (On dit qu'on peut les modifier **en place**.)

Exemple 4.8. Instructions et fonctions « génériques »

Même exemple pour l'essentiel que l'exemple 4.7 pour les n–uplets.

```
In [1]: A=[1, 8, 3, 6, 0, 8, 2, 5, 4, 7]
In [2]: type(A)
In [3]: B=[31,"a",0.7,(8,9)] # Les composantes peuvent être de types différents
In [4]: print(A[0])
In [5]: print(B[3][1])
In [6]: print(A[-1])
In [7]: print(A[1:4])
In [8]: print(A[:3])
In [9]: print(A[3:])
In [10]: print(A[1:5:2])
In [11]: print(A[::-1])
In [12]: C=[] ; print(C) # liste vide
In [13]: D=[1,] ; print(D) # Singleton. La virgule n'est pas obligatoire
In [14]: E=[1] ; print(E) # Singleton, sans la virgule
In [15]: x,y,z=A[1:4] # Déconstruction
In [16]: print(x, y, z)
In [17]: print(len(A))
In [18]: print(sum(A))
In [19]: print(min(A))
In [20]: print(max(A))
In [21]: print(A+B)
In [22]: 1 in A
In [23]: 9 in A
In [24]: A.count(8)
In [25]: A.index(8)
```

Exemple 4.9. Modification d'un élément d'une liste

Illustration de la modification « en place » d'une liste (spécificité du type `list` (listes) par rapport aux types `str` (chaînes de caractères) et `tuple` (n–uplets)).

```
In [26]: A=[0,1,2,3,4,5]
In [27]: A[1]=9
In [28]: print(A)
In [29]: A[-1]=10
In [30]: print(A)
In [31]: A[6]=20 # Attention ! Erreur en cas de dépassement
```

Exercice 4.2 *(Incontournable)*

1. Écrire une fonction `vabsolue` prenant en paramètre d'entrée une liste de nombres et qui remplaçera chaque nombre par sa valeur absolue. (La fonction ne retournera rien (`None`).)
2. Écrire une fonction `permutation` prenant en paramètre d'entrée une liste et qui la modifie en plaçant le dernier élément en première position. (La fonction ne retournera rien (`None`).)
3. Proposez des tests pour vérifier vos fonctions (et faites les ...).

Quelques fonctionnalités supplémentaires

Attention!!! Les méthodes (c'est-à-dire les fonctions écrites après la variable suivie d'un point) de cette section appliquées à une liste modifient celle-ci. Ceci est possible car les listes sont **modifiables** ou **mutables**.

- On peut ajouter un élément à la fin d'une liste avec `append`.

Exemple 4.10.

```
In [1]: A=[5,7,4,2,6]
In [2]: A.append(3)
In [3]: print(A)
```

- On peut ajouter les éléments d'une liste à la fin d'une autre liste avec `extend`.

Exemple 4.11.

```
In [4]: B=[1,9]
In [5]: A.extend(B)
In [6]: print(A)
```

- On peut insérer un élément dans une liste à une position donnée avec `insert` ou avec un « slice ».

Exemple 4.12.

```
In [7]: A.insert(3,33) # première solution
In [8]: print(A)
In [9]: A[1:1]=[11] # deuxième solution
In [10]: print(A)
```

- On peut supprimer un élément d'une liste par sa position ou sa valeur.

Exemple 4.13.

```
In [11]: A.remove(3) # Attention ! ne supprime que la première occurrence
In [12]: print(A)
In [13]: var=A.pop(1) # supprime l'élément donné par son indice et le retourne
In [14]: print("Liste A : {}. Élément supprimé : {}".format(A,var))
In [15]: var=A.pop() # Sans indice, c'est le dernier élément qui est visé
In [16]: print("Liste A : {}. Élément supprimé : {}".format(A,var))
```

- On peut mélanger les éléments d'une liste de manière aléatoire avec `shuffle`.

Exemple 4.14.

```
In [17]: from random import shuffle
In [18]: shuffle(A)
In [19]: print(A)
```

- Enfin, on peut trier ou « inverser » les éléments d'une liste.

Exemple 4.15.

```
In [20]: B=sorted(A) # trier une liste solution 1
In [21]: print(A,B)
In [23]: A.sort() ; A # inverser une liste solution 2
In [24]: print(A)
In [25]: B=A[::-1] # inverser une liste solution 1
In [26]: print(A,B)
In [31]: A.reverse() ; A # inverser une liste solution 2
In [32]: print(A)
```

Exercice 4.3

1. Pour une liste `L`, quelle est la différence entre `L.reverse()` et `L[::-1]` ?
2. Pour une liste `L`, quelle est la différence entre `L.sort()` et `sorted(L)` ?

Remarque

Les fonctions `sorted` et `reversed`, ainsi que l'instruction `L[::-1]` peuvent être utilisées avec les n–uplets, mais les fonctions retournent des listes (et pas des n–uplets).
`L.reverse()` et `L.sort()` ne fonctionnent évidemment pas avec les n–uplets (essayez !).

4.5 – Conversion de types

Exemple 4.16.

```
In [1]: A=[1,2,3,4,5]
In [2]: B=tuple(A) # conversion list -> tuple
In [3]: type(B)
In [4]: print(B)
In [5]: C=list(B) # conversion tuple -> list
In [6]: type(C)
In [7]: print(C)
In [8]: mot="Bonjour"
In [9]: L=list(mot) # conversion str -> list
In [10]: type(L)
In [11]: print(L)
In [12]: T=tuple(mot) # conversion str -> tuple
In [13]: type(T)
In [14]: print(T)
```

On peut découper une chaîne selon un séparateur pour obtenir une liste, et à l'inverse, reconstruire une chaîne à partir d'une liste avec, respectivement, `split` et `join`.

Exemple 4.17.

```
In [15]: "abc=:=def=:=ghi=:=jkl".split("=:=")
In [16]: "-".join(["abc", "def", "ghi"])
In [17]: "abc;def;ghi;jkl;".split(";")
In [18]: "*".join(["H","e","l","l","o"])
In [19]: mots="Au revoir"
In [20]: L=["a","e","i","o","u"]
In [21]: mots.split(" ")
In [22]: ":".join(L)
In [23]: "".join(L)
```

4.6 – Références partagées

Lorsqu'on affecte un objet à une variable, on affecte en fait la référence de cet objet (i.e. un « identifiant » de cet objet). Ce type d'affectation peut avoir des effets de bord pour les objets mutables tels que les listes.

Note : En informatique, une instruction est dite à effet de bord, si elle modifie un état autre que celui sur lequel elle « travaille ».

Exemple 4.18.

```
In [1]: L1=[1,2,3,4,5]
In [2]: L2=L1
```

```
In [3]: print(L1)
In [4]: print(L2)
In [5]: L2[0]='new'
```

```
In [6]: print(L2)
Mais aussi ...
In [7]: print(L1)
```

On dit que `L1` et `L2` ont une **référence partagée**.

Pour illustrer ce qui se passe, on va utiliser la fonction `id` qui donne l'identifiant d'un objet :

Vous trouverez sur moodle un lien vers une vidéo expliquant la notion de référence partagée.

Objet immuable

```
In [1]: v1=5
In [2]: id(v1)
Out [2]: 10914624
```

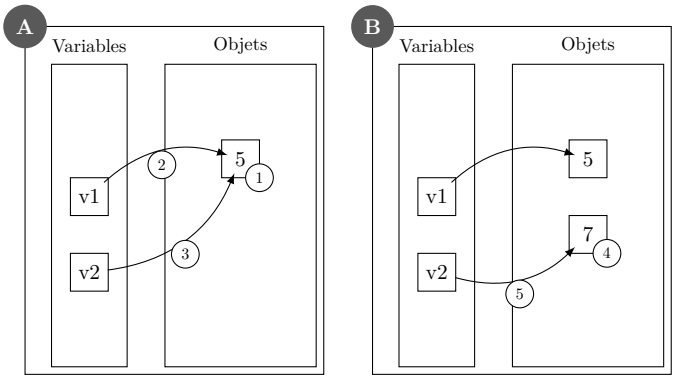
L'objet « 5 » est créé en mémoire ① puis est assigné à `v1` ②.

```
In [3]: v2=v1
In [4]: id(v2)
Out[4]: 10914624
```

`v1` est assigné à `v2` ③. Les deux variables ont la même référence/identifiant.

```
In [5]: v2=7
In [6]: id(v2)
Out [6]: 109146 88
```

L'objet 5 est immuable, donc l'objet 7 est créé en mémoire ④ puis est assigné à `v2` ⑤. Il n'y a donc pas de problème dans ce cas.



Objet mutable

```
In [1]: L1=[3,5] # (1) et (2)
In [2]: id(L1)
Out[2]: 140619280926344
```

Les objets 3 et 5 sont créés en mémoire, puis la liste est créée ①. Le premier élément de la liste « pointe » vers 3 et le deuxième vers 5. Enfin, la liste est assignée à `L1` ②.

```
In [3]: L2=L1 # (3)
In [4]: id(L2)
Out[4]: 140619280926344
```

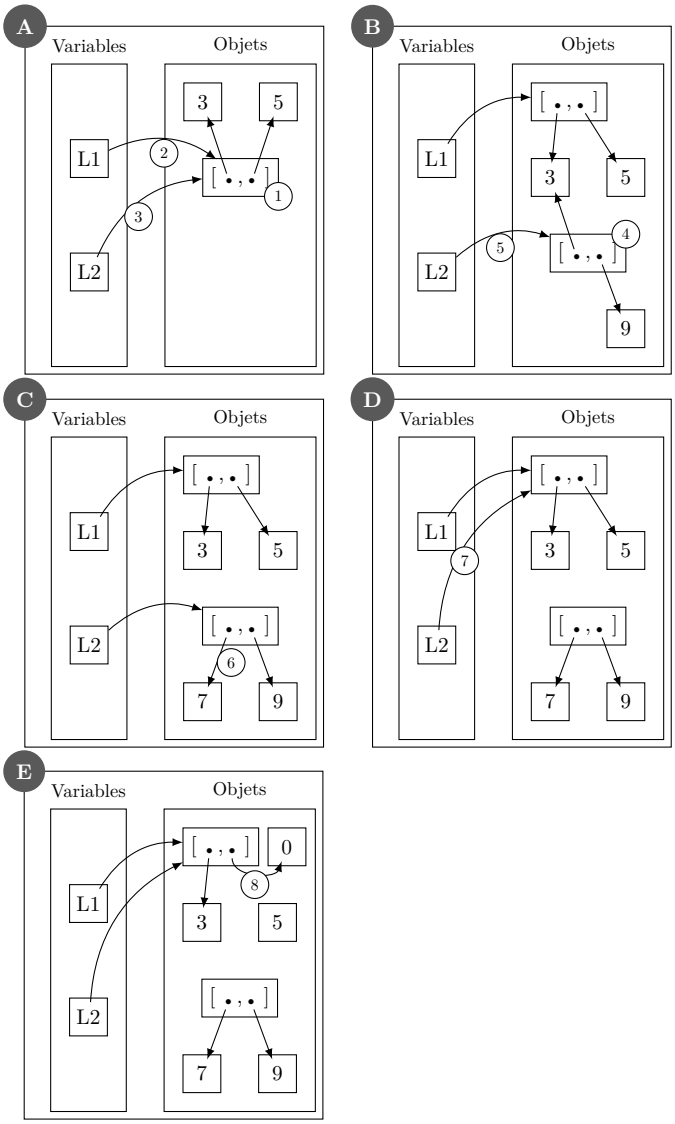
`L1` est assigné à `L2` ③. Les deux variables ont la même référence/identifiant.

```
In [5]: L2=[3,9] # (4) et (5)
In [6]: id(L2)
Out[6]: 140619281026696
In [7]: L2[0]=7 # (6)
In [8]: print(L2) ; print(L1)
[7, 9]
[3, 5]
```

Une nouvelle liste est créée en mémoire ④ et assignée à `L2` ⑤. La première valeur de la liste `L2` est changée « en place » car une liste est mutable ⑥. Une nouvelle référence est créée pour 7 car 5 est immuable.
Là encore pas de problème particulier.

```
In [9]: L2=L1 # (7)
In [10]: id(L2)==id(L1)
Out[10]: True
In [11]: L2[1]=0 # (8)
In [12]: id(L2)==id(L1)
Out[12]: True
In [13]: L1[1]
Out[13]: 0
```

On assigne à nouveau `L1` à `L2` ⑦. Puis la première valeur de la liste `L2` est modifiée ⑧. Cette modification est faite « en place » car une liste est un objet mutable. La conséquence est que `L1` qui partage la référence avec `L2` est aussi modifiée.



Copie

La question se pose donc de savoir comment on fait une copie d'une liste (de façon à ne pas modifier la liste originale). Cette copie peut se faire en Python à l'aide des « sous–listes ».

Vous trouverez sur moodle un lien vers une vidéo expliquant la copie de listes.

Exemple 4.19.

```
In [1]: L1=[1,2,3]
In [2]: L2=L1[:] # On réalise une copie de la liste
In [3]: id(L2)==id(L1) # Le retour est bien False
In [4]: L2[0]='new'; L2
In [5]: L1 # L1 n'est pas modifiée
```

Mais même une copie peut poser problème. La copie réalisée par `[:]` est une copie de « surface ».

Exemple 4.20.

```
In [1]: L1=[3,[5,7]] # Noter que le 2ème élément de la liste est lui-même une liste
In [2]: L2=L1[:] # On réalise une copie de la liste
In [3]: id(L2)==id(L1) # Le retour est bien False
In [4]: L2[1][0]=9
In [5]: L2
In [6]: L1 # Surprise!
```

Faire des schémas similaires aux précédents pour montrer ce qu'il se passe.

deepcopy

On peut réaliser une copie en profondeur d'une liste de façon à avoir une autre liste totalement indépendante. Il faut faire appel au module `copy`.

Exemple 4.21.

```
In [1]: import copy
In [2]: L1=[3,[5,7]]
In [3]: L2=copy.deepcopy(L1) # Copie en profondeur
In [4]: L2[1][0]=9
In [5]: L2
In [6]: L1
```

4.7 – Objets itérables

4.7.1 – Définition

On dit qu'un objet de type composé est **itérable** si l'on sait identifier le premier élément et si pour tout élément on sait quel est le suivant. C'est en particulier le cas des séquences vues dans ce chapitre (`str`, `tuple` et `list`).

Les propriétés d'un itérable permettent donc de parcourir celui–ci du premier au dernier élément et on peut donc les utiliser dans une boucle `for`.

Exemple 4.22.

Écrire puis exécuter le programme suivant :

```
for lettre in "Bonjour!":
    print(lettre)
for p in (2,3,5,7,11):
    print(p)
for t in [3,6,9,12]:
    print(t)
```

On peut aussi parcourir ces objets en récupérant la position **et** la valeur des éléments avec la fonction `enumerate`.

Exemple 4.23.

Écrire puis exécuter le programme suivant :

```
mot="Bonjour!"
print(mot)
for (position, valeur) in enumerate(mot):
    print(f"Position : {position}. Valeur : {valeur}.")
T=(2,3,5,7,11)
print(T)
for (position, valeur) in enumerate(T):
    print(f"Position : {position}. Valeur : {valeur}.")
L=[3,6,9,12]
print(L)
for (position, valeur) in enumerate(L):
    print(f"Position : {position}. Valeur : {valeur}.")
```

4.7.2 – Construction d'une liste en compréhension

De même qu'en mathématiques, on peut définir des **listes en compréhension** (à l'aide des objets itérables).

Exemple 4.24.

```
In [1]: [k for k in range(10)]
In [2]: [k for k in range(9,0,-1)]
In [3]: [k for k in range(3,25,2)]
In [4]: [x**2 for x in range(10)]
In [5]: [x**2 for x in range(10) if x%2==0]
In [6]: [x**2 for x in [2*y+1 for y in range(5)]]
In [7]: [n + p for n in [5, 7] for p in [10, 20, 30]]
In [8]: [[n + p for n in [5, 7]] for p in [10, 20, 30]]
```

Exercice 4.4 *(Incontournable)*

1. Quel est le rôle des paramètres `a`, `b` et `c` dans l'instruction `range(a,b,c)` ?
2. Donner une instruction permettant d'obtenir une liste des entiers de l'ensemble $\left\{(2n)^3 \mid n \in [\![3;42]\!]\right\}$.
3. Donner une instruction permettant d'obtenir une liste de tous les multiples de 5 de 15 à 95.

4.8 – Exercices

Exercice 4.5 *(Incontournable)*

On se donne la chaîne de caractères `alphabet='abcdefghijklmnopqrstuvwxyz'`.

Pour chacune des sous–chaînes ci-dessous, écrire une expression utilisant les sous–listes (i.e. une expression du type `alphabet[<debut>:<fin>:<pas>]`) qui renvoie celle–ci.

L'expression `alphabet[<votre code>]==<la chaine>` devrait retourner `True`.

Par exemple pour obtenir `'a'`, on peut écrire `alphabet[0]`.

Liste des chaînes à obtenir :

1. `'z'`
2. `'zyxwvutsrqponmlkjihgfedcba'`
3. `'efgh'`
4. `'uvwx'`
5. `'vwxyz'` (avec une seule constante)
6. `'fhjlnprtvxz'` (avec deux constantes)
7. `'yvspmjgd'`

Exercice 4.6 *(Incontournable)*

Quel sera le résultat des instructions suivantes ?

(Répondez sans utiliser Python, puis vérifier votre réponse en tapant l'instruction dans la console interactive.)

1. `type("aaa")`
2. `type(["a","a","a"])`
3. `type(("a","a","a"))`
4. `type(3*"5")`
5. `3*"5"`
6. `type(3*5)`
7. `type(3*["5"])`
8. `3*["5"]`
9. `type(3*[5])`
10. `3*[5]`
11. `type(3*("5"))`
12. `3*("5")`
13. `type(3*(5))`
14. `3*(5)`
15. `type(3*(5.))`
16. `3*(5.)`
17. `type(3*("5",))`
18. `3*("5",)`
19. `type(3*(5,))`
20. `3*(5,)`
21. `"3"*["5"]`

Exercice 4.7 *(Incontournable)*

1. Écrire une fonction qui prend en paramètres d'entrée une liste `L` et un élément `e` et qui retourne le booléen Vrai si `e` est dans `L` et Faux sinon. Pour cela, vous comparerez `e` avec les éléments successifs de `L` jusqu'à le trouver ou à arriver à la fin de la liste. (Sans utiliser « `e in L` ».)
2. Vous testerez votre fonction en comparant vos résultats avec le retour de la commande « `e in L` » .

Exercice 4.8 *(Incontournable)*

1. Écrire une fonction qui prend en paramètres d'entrée une liste `L` et un élément `e` et qui retourne le nombre d'occurrences de `e` dans la liste `L` (sans utiliser `count`).
2. Vous testerez votre fonction en comparant vos résultats avec le retour de la commande « `L.count(e)` » .

Exercice 4.9 *(Incontournable)*

Écrire une fonction qui prend en paramètre d'entrée une liste de numériques `L` et retourne une liste de coordonnées de type `[x,cos(x)]` où `x` prend toutes les valeurs de `L`.

Exercice 4.10

Écrire un programme qui trouve le plus petit des entiers non nuls tel que les chiffres qui le composent s'inversent quand on multiplie ce nombre par 9.

Exercice 4.11

1. Écrire la fonction `fmoy` (resp. `fsom`, `fmax`, `fmin`) prenant en paramètre d'entrée une liste de numériques `L` et retourne la moyenne (resp. la somme, le maximum, le minimum) des éléments. (Sans utiliser les fonctions `sum`, `max`, `min`, `sort` et `sorted`.)
2. Proposez des tests pour vérifier vos fonctions (et faites les tests ...).

Exercice 4.12

On considère les fonctions f, g, h et les variables L et M définies ci-dessous.

```
def f(liste1,liste2):
    liste1.extend(liste2)
    liste1[-1][0]=9
    return liste1
#------------------------
def g(liste1,liste2):
    liste1=liste1+liste2
    liste1[-1][0]=9
    return liste1
#------------------------
def h(liste1,liste2):
    liste2=liste2+[[9,9]]
    liste1=liste1+liste2
    liste1[-1][0]=0
    return liste1
#------------------------
L=[[1,2],[3,4]]
M=[[5,6],[7,8]]
```

1. Ces fonctions ont-elles des effets de bord ? Autrement dit, si à la suite du programme ci-dessus on écrit `print(f(L,M))` (respectivement `print(g(L,M))`, `print(h(L,M))`), les variables L et M sont-elles modifiées ?
2. Expliquez pourquoi (pour chacune des fonctions).

Exercice 4.13

On considère les fonctions repete1 et repete2 définies ci-dessous.

```
def repete1(L):
    for i in range(len(L)):
        L.append(L[i])
    return None
#------------------------
def repete2(L):
    i=0
    while i<len(L):
        L.append(L[i])
        i=i+1
    return None
```

Ces deux fonctions doivent modifier la liste donnée en entrée en lui ajoutant les mêmes éléments (ou autrement dit concaténer la liste avec elle-même). Par exemple si une liste est au départ `L=[1,2,3]`, après exécution de la fonction on doit avoir `L=[1,2,3,1,2,3]`.

1. A l'utilisation, une des deux fonctions ne se termine pas et semble faire une boucle infinie. Laquelle ? Pourquoi ?
2. Pourquoi l'autre fonction, elle, produit bien le résultat voulu ? (Et ne provoque pas de boucle infinie.)
3. Proposer une correction, en gardant le même type de boucle, de la fonction provoquant une boucle infinie pour que celle-ci fonctionne correctement.

Exercice 4.14 ★

1. Écrire un programme qui affiche la $n^{\text{ième}}$ ligne du triangle de Pascal (i.e. les coefficients $\binom{n}{k}$ pour k entre 0 et n). Vous utiliserez les listes et la relation $\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}$.

   Par exemple, la $3^{\text{ème}}$ ligne du triangle, dont les coefficients sont 1, 3, 3 et 1, sera stockée en mémoire comme la liste `[1,3,3,1]`. La liste correspondant à la $4^{\text{ème}}$ ligne sera calculée avec la relation ci-dessus de la manière suivante : `[1,1+3,3+3,3+1,1]` (et donnera `[1,4,6,4,1]`).

2. Modifier votre programme pour qu'il affiche le triangle de Pascal complet (de 1 à n).

Exercice 4.15

Écrire un programme qui permet à l'utilisateur de saisir une série de valeurs jusqu'à ce qu'il tape `fin`. les valeurs seront stockées dans une liste. Puis, votre programme vérifiera si les valeurs sont distinctes deux à deux, et affichera un message donnant le résultat de la vérification.

Exercice 4.16 *(Facultatif)*

On considère l'expérience du lancer de deux dés équilibrés et on définit X la variable aléatoire égale à la somme des résultats des deux dés.

Écrire un programme qui permet d'établir expérimentalement (en utilisant la fonction `random` et un nombre assez « grand » de tirages aléatoires) la loi de X et son espérance.

Comparer avec les résultats théoriques exacts.

Exercice 4.17

Tri sélection-permutation

Pour le **tri sélection-permutation** on commence par regarder l'ensemble des éléments. Parmi ceux-ci on cherche le plus petit et on échange sa position avec le premier élément. Puis on regarde les éléments à partir du deuxième, et on échange le plus petit avec le deuxième élément, etc. Ainsi à chaque étape, on va regarder la fin de la liste et placer son plus petit élément en dernière position de la première partie de la liste qui est déjà triée.

Principe formalisé : Supposons que les i premières valeurs de la liste `L` soient triées et correspondent aux i plus petits éléments. On cherche alors l'indice j du plus petit élément parmi les valeurs suivantes (de la $(i+1)^{\text{ième}}$ à la fin) puis on échange les éléments des positions $(i+1)$ et j. On recommence l'opération jusqu'au bout de la liste.

Exemple : Application du tri insertion à la liste [5, 2, 1, 4, 8].
À chaque étape, le plus petit élément parmi les valeurs restantes à trier est en gras.
Les éléments entourés d'un cercle sont les valeurs placées définitivement.

Étape 1 : [5, 2, **1**, 4, 8] → [①, 2, 5, 4, 8]
(Aucune valeur n'étant placée pour l'étape 1, on recherche le minimum de toute la liste.)

Étape 2 : [①, **2**, 5, 4, 8] → [①, ②, 5, 4, 8]

Étape 3 : [①, ②, 5, **4**, 8] → [①, ②, ④, 5, 8]

Étape 4 : [①, ②, ④, **5**, 8] → [①, ②, ④, ⑤, 8]

Étape 5 : [①, ②, ④, ⑤, **8**] → [①, ②, ④, ⑤, ⑧]

---

1. Écrire une fonction `indice_min` avec comme paramètres d'entrée une liste `L` et un entier `debut`, et qui retourne l'indice du plus petit élément de la liste parmi les éléments de `debut` à la fin (c'est à dire parmi la sous-liste `L[debut:]`).
2. Écrire une fonction `echange` avec comme paramètre d'entrée une liste `L`, un entier `i` et un entier `j` et qui échange les éléments numéros `i` et `j` dans `L`. (La fonction retournera `None`.)
3. Écrire une fonction `tri_selection` avec comme paramètre d'entrée une liste `L` et qui trie celle–ci en utilisant le principe ci-dessus (la fonction retournera `None`).

Exercice 4.18

Tri Insertion

Le **tri par insertion** est le tri du « joueur de cartes ». On regarde les éléments de la liste un par un dans leur ordre d'arrivée et on place chaque nouvel élément parmi les précédents de façon à garder une liste triée. Il n'y a donc rien à faire pour traiter le premier élément qui est une liste triée à lui tout seul. On range ensuite le second élément pour obtenir une liste triée de longueur 2, puis troisième élément pour construire une liste triée de longueur 3, etc.

Principe formalisé : Supposons que les i premières valeurs de la liste `L` soient triées, on insère alors la $(i+1)^{\text{ième}}$ valeur à sa place parmi les i premières (pour cela on la fait « descendre » jusqu'à sa place).

Exemple : Application du tri insertion à la liste [5, 2, 1, 4, 8].
À chaque étape, l'élément à placer est en gras. (Il doit être placé parmi les premières valeurs déjà traitées.)
Les éléments entourés d'un cercle sont les « i premières valeurs triées ».

Étape 1 : [⑤, 2, 1, 4, 8] (Pour l'instant aucune valeur n'a été placée, la $1^{\text{ère}}$ valeur est donc triée)

Étape 2 : [⑤, **2**, 1, 4, 8] → [②, ⑤, 1, 4, 8]

Étape 3 : [②, ⑤, **1**, 4, 8] → [①, ②, ⑤, 4, 8]

Étape 4 : [①, ②, ⑤, **4**, 8] → [①, ②, ④, ⑤, 8]

Étape 5 : [①, ②, ④, ⑤, **8**] → [①, ②, ④, ⑤, ⑧]

- Écrire une fonction `tri_insertion` avec comme paramètre d'entrée une liste `L` et qui trie celle–ci en utilisant le principe ci-dessus (la fonction retournera `None`).
  (Vous pouvez utiliser la fonction `echange` écrite pour l'exercice précédent.)

Exercice 4.19 *(Facultatif)*

Tri à bulles

Le **tri à bulles** est un algorithme de tri qui consiste à comparer répétitivement les éléments consécutifs d'une liste, et à les permuter lorsqu'ils sont mal triés. (Les plus grands éléments remontent en fin de liste comme des bulles d'air remonteraient à la surface d'un liquide.)

Principe :

- L'algorithme parcourt la liste des éléments et compare les éléments adjacents. Lorsque les éléments ne sont pas dans l'ordre, ils sont échangés.
- Après un premier parcours complet de la liste, le plus grand élément est nécessairement en fin de liste, à sa position définitive. En effet, dès que le plus grand élément est rencontré, il est mal positionné par rapport à tous les éléments suivants, donc échangé à chaque fois jusqu'à la fin de la liste. Le plus grand élément est alors à sa position définitive et il n'a plus à être traité.
- Par contre, le reste de la liste est encore en désordre. Il faut donc la parcourir à nouveau, en s'arrêtant à l'avant-dernier élément. Après ce deuxième parcours, les deux plus grands éléments sont à leur position définitive. On continue ainsi jusqu'à ce que les deux plus petits éléments soient placés à leur position définitive.

Exemple : Application du tri à bulles à la liste [5, 2, 4, 1, 8].
Pour chaque étape, les éléments comparés sont en gras.
Quand un élément est placé définitivement, il est entouré d'un cercle.

Étape 1 : [**5**, **2**, 4, 1, 8] → [**2**, **5**, 4, 1, 8] (Échange car $5 > 2$.)
[2, **5**, **4**, 1, 8] → [2, **4**, **5**, 1, 8] (Échange, car $5 > 4$.)
[2, 4, **5**, **1**, 8] → [2, 4, **1**, **5**, 8] (Échange, car $5 > 1$.)
[2, 4, 1, **5**, **8**] → [2, 4, 1, **5**, ⑧] (Pas d'échange, car $5 < 8$.)

Étape 2 : [**2**, **4**, 1, 5, ⑧] → [**2**, **4**, 1, 5, ⑧] (Pas d'échange.)
[2, **4**, **1**, 5, ⑧] → [2, **1**, **4**, 5, ⑧] (Échange.)
[2, 1, **4**, **5**, ⑧] → [2, 1, **4**, ⑤, ⑧] (Pas d'échange.)

Étape 3 : [**2**, **1**, 4, ⑤, ⑧] → [**1**, **2**, 4, ⑤, ⑧] (Échange.)
[1, **2**, **4**, ⑤, ⑧] → [1, **2**, ④, ⑤, ⑧] (Pas d'échange.)
Note : À ce stade, la liste est déjà triée mais l'algorithme le « sait pas » encore.

Étape 4 : [**1**, **2**, ④, ⑤, ⑧] → [**1**, ②, ④, ⑤, ⑧] (Pas d'échange.)

---

1. Écrire une fonction `tri_bulles` avec comme paramètre d'entrée une liste `L` et qui trie celle–ci en utilisant le principe ci-dessus (la fonction retournera `None`).
2. Donner un exemple pour lequel le nombre d'échanges est maximal (le « pire des cas »).

Exercice 4.20 *(Facultatif)*

Codage de César

1. Créer une liste « `alphabet` » de taille 26 avec les lettres de l'alphabet dans l'ordre. (Nul besoin pour cela de les écrire dans l'ordre ....)
2. Écrire une fonction `valeur` prenant en argument un caractère de l'alphabet et donnant sa position dans la liste.
3. Écrire une fonction `lettre` prenant en argument un entier entre 0 et 25 et donnant le caractère correspondant.
4. Écrire une fonction `normalise` prenant en argument une chaîne de caractères et renvoyant la même chaîne en minuscule débarrassée de tous les caractères qui ne sont pas dans l'alphabet (comme « espace » par exemple). On dira que la chaîne est normalisée.
5. Écrire une fonction `code` prenant en argument une chaîne de caractères et renvoyant la liste des valeurs des caractères de la chaîne normalisée correspondante.
6. Écrire une fonction `decode` prenant en argument une liste de valeurs (entre 0 et 25) et renvoyant la chaîne de caractètres correspondante.

Le système de codage « de César » consiste à décaler toutes les lettres de l'alphabet de K positions oàu K, qui s'appelle la clé, n'est connue que des utilisateurs du code.
Par exemple si $K = 3$, a sera changée en d, b en e, ..., w en z, x en a, y en b et z en c.
Le codage consiste ainsi à une addition de la valeur correspondant à un caractère et de K modulo 26.

7. Écrire une fonction `codage_cesar` prenant en argument une chaîne de caractères et un entier K (entre 0 et 25), et renvoyant la chaîne normalisée codée par le code de César.
8. Écrire une fonction `decodage_cesar` permettant l'opération inverse.

Attaque du système : ce codage n'est pas du tout sûr, en effet il conserve la fréquence d'apparition de chaque caractère. Ainsi, en français, le e étant le plus fréquent, il correspondra certainement à la lettre la plus fréquente du message (et sinon à la suivante ...). On peut donc ainsi trouver la clé utilisée.

9. Écrire une fonction `cle_code` permettant d'obtenir, à priori, la clé utilisée pour coder une chaîne de caractères donnée en entrée.
10. Écrire enfin une fonction `casse_cesar` qui à partir d'une chaîne codée avec le code de César retourne le message en clair. Appliquer votre fonction au message codé donné sur moodle.

5 – Représentation des entiers

5.1 – Entiers en base b

Notre système de numération appelé système décimal est un système de numération positionnelle utilisant 10 chiffres. Dans cette écriture, un chiffre donne le nombre d'unités de la puissance de 10 correspondant à sa position. Par exemple : $7294 = 4 \times 10^0 + 9 \times 10^1 + 2 \times 10^2 + 7 \times 10^3$

On dit qu'on utilise un système en base 10.

Définition 5.1. Écriture en base b

Dans un système de numération positionnelle en base b, on note un nombre N par $(a_n a_{n-1} \dots a_1 a_0)_b$.
Avec a_i un « chiffre » strictement inférieur à b pour $i \in [\![0; n]\!]$.

Le nombre N ainsi représenté est : $N = \sum_{i=0}^{n} a_i \times b^i = a_0 \times b^0 + a_1 \times b^1 + \dots + a_n \times b^n$.

Remarque

Comme nous ne disposons que de 10 chiffres, se pose le problème de l'écriture avec des bases supérieures à 10 … En général, après le chiffre 9, on utilise les lettres de l'alphabet. Par exemple en base 16 (appelée base hexadécimale), les « chiffres » seront : 0, …, 9, A, B, C, D, E, F.
On aura donc $(2\text{FA})_{16} = 10 \times 16^0 + 15 \times 16^1 + 2 \times 16^2 = 762$

« Il y a 10 *sortes de personnes : celles qui comprennent la notation binaire, et les autres … »*

Propriété 5.1. Conversion de la base b à la base 10

Pour convertir un nombre écrit en base b vers la base 10, il suffit d'appliquer la définition de l'écriture en base b.

Propriété 5.2. Conversion de la base 10 à la base b

Pour convertir un nombre écrit en base 10 vers la base b, on effectue des divisions euclidiennes successives de ce nombre (des quotients obtenus) par b jusqu'à obtenir un quotient nul. L'écriture du nombre en base b s'obtient alors, écrivant tous les restes obtenus du dernier au premier (on « remonte » les restes).

Exemple 5.1.

Pour écrire 171 en base 8 : $171 = 21 \times 8 + 3$; $21 = 2 \times 8 + 5$; $2 = 0 \times 8 + 2$. Ainsi $171 = (253)_8$.

5.2 – Entiers en mémoire

Le bit est l'unité de mesure de base en informatique, c'est la quantité minimale d'information transmise dans un message. Le bit ne peut prendre que deux valeurs souvent représentées par les chiffres 0 et 1. Un octet est un groupement séquentiel de 8 bits, il permet par exemple de coder les entiers naturels de 0 à 255 à l'aide de leur écriture en base 2.

D'une manière générale un codage sur n bits permet de représenter les entiers naturels de 0 à $(2^n - 1)$ en utilisant l'écriture de ces entiers en base 2.

Remarque

Ce codage a ses limites et on peut avoir un dépassement de capacité (« overflow ») qui produit donc un résultat erroné. Par exemple avec des nombres entiers (positifs) codés sur 8 bits, la somme des deux entiers $a = 128$ et $b = 192$ conduit à un résultat erroné de 64. (Dans le cas d'un codage sur 8 bits les additions des entiers naturels se font modulo 256.)

5.2.1 – Complément à 1

Une approche « naturelle » pour coder les entiers relatifs est de fixer le premier bit et de l'utiliser pour donner le signe, le reste des bits correspond à la valeur absolue de l'entier écrit en base 2.

Cette approche présente cependant plusieurs inconvénients :

- Il y a deux zéros, un positif et un négatif ...
- On perd une valeur représentée : par exemple sur 8 bits on représentera les entiers de -127 à 127 soit 255 valeurs et non 256.
- Avec ce codage, il faudra programmer séparément l'addition et la soustraction, car $a - b \neq a + (-b)$.

5.2.2 – Complément à 2

La notation en complément à 2 est la représentation usuelle pour représenter les entiers relatifs.

Les entiers relatifs sont codés sur n bits, ce codage permettant alors de représenter les entiers appartenant à $[\![-2^{n-1}; 2^{n-1} - 1]\!]$.

Pour un entier $a \in [\![-2^{n-1}; 2^{n-1} - 1]\!]$, le principe est le suivant :

- Si $a \geqslant 0$: a est simplement représenté par son écriture en base 2.

  Le bit de poids fort (le premier bit) sera 0.
- Si $a < 0$: a est représenté par $2^n + a$.

  Le bit de poids fort (le premier bit) sera 1.

Avec cette approche, on règle les inconvénients de la notation en complément à 1 :

- Il y a un unique zéro.
- On présente le maximum de nombres pouvant être codés sur n bits.
- L'addition est la même pour les entiers positifs et négatifs.

À titre d'exemples les intervalles de codage sont :

- sur 8 bits (1 octet) : $[\![-128; 127]\!]$;
- sur 16 bits (2 octet) : $[\![-32768; 32767]\!]$;
- sur 32 bits (4 octet) : $[\![-2147483648; 2147483647]\!]$.

Remarque Opposé d'un nombre

Pour obtenir l'opposé d'un entier relatif dans la représentation en complément à deux, il suffit d'inverser les bits ($0 \longleftrightarrow 1$), puis d'ajouter 1.
(Attention, ce n'est pas valable pour le nombre -2^{n-1} évidemment.)

5.3 – Exercices

Exercice 5.1 *(Incontournable)*

1. Lesquelles des suites de chiffres : 10101100, 10102011, 10108141, 2A0GF00 peut être la représentation d'un nombre en base 2, 8 ou 16 ?
2. Convertir en base 10 les nombres suivants :
   - $(0)_2$, $(10010)_2$, $(100101011)_2$, $(11100100)_2$
   - $(BAFF)_{16}$, $(A4B)_{16}$, $(5AC)_{16}$, $(EF1)_{16}$, $(59D)_{16}$
3. Convertir en base 2 puis 16 les nombres suivants exprimés en base de 10 : 1, 5, 25, 210, 2021, 2231.
4. Convertir $(BAC)_{16}$ en base 2.
5. Convertir en base hexadécimale (base 16) les nombres : $(11101010)_2$ et $(1100110010)_2$.
6. Comment est représenté b^k en base b ?

Exercice 5.2

1. Déterminer la valeur de b pour que l'égalité $(256)_{10} = (100)_b$ soit vraie.
2. Déterminer la valeur de b pour que l'égalité $(554)_{10} = (1052)_b$ soit vraie.

Exercice 5.3

1. Coder sur 16 bits (en complément à 2) les nombres suivants (écrits en base 10) : 0, 1, 2, -1, -2, -184 et 21845.
2. Donner les valeurs en base de 10 du nombre binaire 10010010 en considérant dans un premier temps un codage en base 2 (sur 8 bits), puis un codage en complément à 2 (sur 8 bits).
3. Effectuer sur 4 bits (en codage en complément à 2) les opérations suivantes, puis vérifier les opérations en les réécrivant en notation décimale :

   (a) $1010 + 0101$ (b) $1010 + 0110$ (c) $1010 + 1010$

Exercice 5.4

Python possède quelques fonctions de conversion pour passer de l'écriture d'une base à une autre (en particulier pour les bases 2 et 16).

Exemple 5.2.

(Vous exécuterez les instructions proposées dans la console interactive.)

- Conversion d'un nombre écrit en base 10
  - Exemples vers la base 2 (binaire) : `bin(11)` ; `bin(11)[2:]` ; `bin(101)` ; `bin(101)[2:]`.
  - Exemples vers la base 16 (hexadécimale) : `hex(11)` ; `hex(11)[2:]` ; `hex(26)` ; `hex(26)[2:]`.
- Conversion vers la base 10
  - Exemples en base binaire : `int('101',2)` ; `int('0b110',2)` ; `int('0b1001',0)` ; `int('1001',0)` (Attention ! ...).
  - Exemples en base hexadécimale : `int('A3',16)` ; `int('a3',16)` ; `int('0xA3',16)` ; `int('0xA3',0)` ; `int('A3',0)` (génère une erreur).

1. Que fait la fonction `int` lorsque son deuxième paramètre est à 0 ?
2. Écrire une fonction « `bin2int(mot_bin)` » qui retourne la valeur d'un entier à partir de son écriture binaire sous forme de chaîne de caractères donnée en entrée (**sans utiliser** les fonctions de conversion de Python). Par exemple `bin2int('101')` retournera `5`.
3. Écrire une fonction « `int2bin(entier)` » qui retourne la chaîne de caractères de l'écriture binaire d'un entier naturel donné en entrée (sans utiliser les fonctions de conversion de Python).
   Par exemple `int2bin(5)` retournera `'101'`.
4. Tester vos fonctions sur des exemples en vérifiant que les résultats obtenus sont les mêmes que les fonctions de conversion de Python.

Exercice 5.5

Dans cet exercice les nombres binaires seront écrits dans des tuples où chaque éléments sera 0 ou 1. Par exemple $(10010101)_2$ sera représenté par le tuple `(1,0,0,1,0,1,0,1)`. On suppose de plus que l'on utilise la **représentation en complément à 2 sur 8 bits** des entiers.

1. Écrire une fonction qui pour la représentation binaire d'un entier (sous forme de tuple) donnée en entrée retourne sa valeur en notation décimale.
2. Écrire une fonction qui pour un entier donné en entrée (en base 10) retourne le tuple correspondant à sa représentation binaire.
3. Écrire une fonction qui pour la représentation binaire d'un entier (sous forme de tuple) donnée en entrée retourne le tuple correspondant à son opposé.
4. Écrire une fonction qui pour deux tuples (représentants 2 entiers en binaire) donnés en entrée retourne le tuple correspondant à leur somme.
5. Écrire une fonction qui pour deux tuples (représentants 2 entiers en binaire) donnés en entrée retourne le tuple correspondant à la différence entre le premier et le deuxième.

Exercice 5.6 *(Facultatif)*

1. Reprendre les fonctions de l'exercice 5.4 pour les généraliser à une base quelconque entre 3 et 36.

   Vos fonctions prendront en entrée un deuxième paramètre b pour la base choisie. (Vous pourrez aussi renommer les fonctions, par exemple : `base2int(mot_bin,b)` et `int2base(entier,b)`.)
2. Pourquoi se limiter à 36 pour la base ?
3. Écrire un programme permettant de vérifier vos fonctions sur plusieurs valeurs pour la base hexadécimale.

6 – Dichotomie

Problématique :
De nombreux problèmes en modélisation se ramènent à la résolution d'équations à une inconnue réelle de type $f(x) = 0$. Une résolution exacte n'est en général pas envisageable et même si c'était le cas la valeur numérique de la solution est parfois donnée à l'aide d'un calcul approché ...

On est donc amené à calculer des solutions approchées. Une des méthodes possibles est la méthode de la dichotomie. Étymologiquement, « dichotomie » signifie « couper en deux ». C'est souvent la stratégie utilisée, par exemple, pour deviner un nombre entier compris entre 0 et 100.

6.1 – Introduction

Exercice 6.1

1. Écrire un programme où l'utilisateur doit déterminer un entier aléatoire entre 0 et 100 par essais successifs. Le programme demandera à l'utilisateur un entier et affichera « + » si le nombre à deviner est plus grand, « - » s'il est plus petit, jusqu'à ce que le nombre soit trouvé.
   Par exemple le déroulement d'un jeu pourrait être le suivant :

```
Vous devez deviner un entier entre 0 et 100
Votre proposition : 23
+
Votre proposition : 71
-
Votre proposition : 37
Bravo, vous avez trouvé en 3 questions.
```

2. Modifier votre programme pour que l'utilisateur détermine un réel aléatoire entre 0 et 1 à 10^{-2} près.
   Par exemple le déroulement d'un jeu pourrait être le suivant :

```
Vous devez deviner un réel entre 0 et 1 à 10**-2 près
Votre proposition : 0.3
+
Votre proposition : 0.67
-
Votre proposition : 0.43
Bravo, vous avez trouvé en 3 questions.
```

Exercice 6.2

1. Écrire un programme où l'ordinateur doit déterminer par essais successifs un entier (entre 0 et 100) choisi par l'utilisateur. Le programme proposera à l'utilisateur des entiers jusqu'à ce que le nombre soit trouvé. L'utilisateur saisira « + » si le nombre à deviner est plus grand que celui proposé, « - » s'il est plus petit et « = » si le nombre a été trouvé.
   Par exemple le déroulement d'un jeu pourrait être le suivant :

```
Je vais deviner l'entier entre 0 et 100 auquel vous pensez
Est-ce 50 ? +
Est-ce 75 ? -
Est-ce 62 ? =
J'ai trouvé en 3 questions.
```

2. Modifier votre programme pour que l'ordinateur détermine un réel aléatoire entre 0 et 1 (à 10^{-2} près) choisi par l'utilisateur. Par exemple le déroulement d'un jeu pourrait être le suivant :

```
Je vais deviner un réel entre 0 et 1 auquel vous pensez (à 10**-2 près)
Est-ce 0.5 ? +
Est-ce 0.75 ? -
Est-ce 0.62 ? =
J'ai trouvé en 3 questions.
```

6.2 – Principe de résolution de $f(x) = 0$

Cadre de travail :
- f fonction réelle définie et continue sur un intervalle $[a, b]$.
- f s'annule en un unique point de $[a, b]$ que l'on notera α et $f(a)f(b) < 0$ (ou autrement dit, $f(a)$ et $f(b)$ sont de signes contraires).

Dans le cadre de la résolution de $f(x) = 0$ sur $[a, b]$, il s'agit de localiser la racine par rapport à $\frac{a+b}{2}$.
On réalise une comparaison des signes de $f(a)$ et de $f(\frac{a+b}{2})$:

- s'ils sont différents, alors $\alpha \in [a, \frac{a+b}{2}]$
- s'ils sont identiques, alors $\alpha \in [\frac{a+b}{2}, b]$

On recommence alors cette procédure dans l'intervalle où se situe la racine ...

Remarque
C'est aussi le principe de la preuve (constructive) du théorème des valeurs intermédiaires.

6.3 – Algorithme

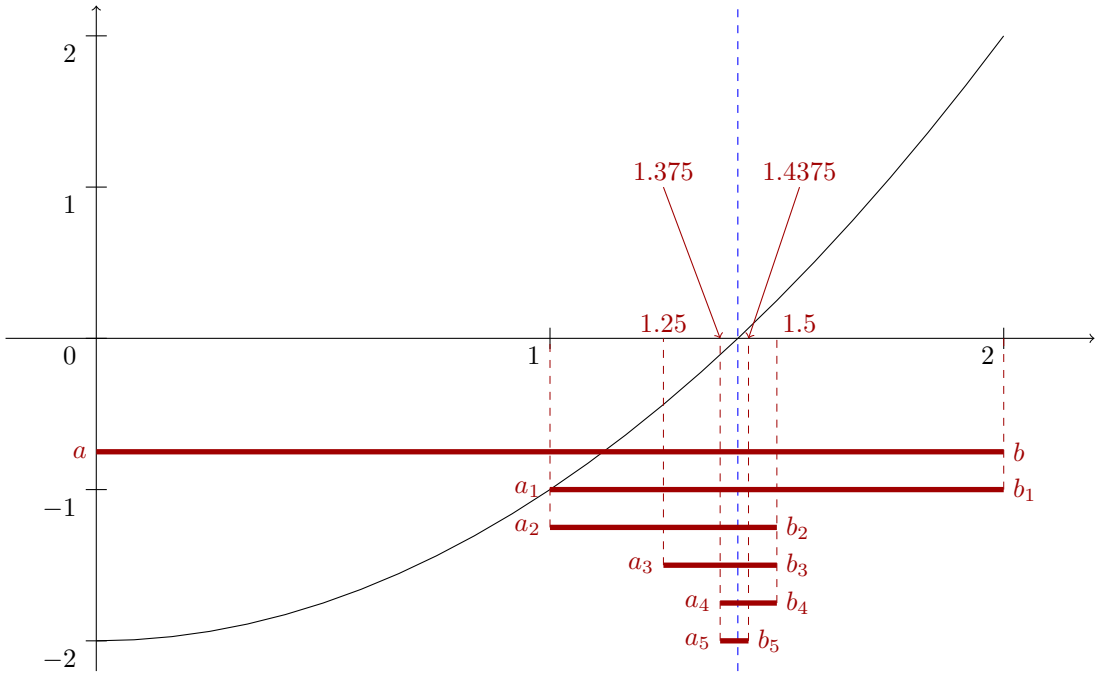
On arrête la boucle dès que la longueur de l'intervalle est inférieure à 2ε et on retourne le milieu du dernier intervalle trouvé.

- **Algorithme :**

```
   Entrées : f,a,b,ε;                          /* précondition : a < b */
1  x ⟵ a;
2  y ⟵ b;
   /* Les variables x et y ne sont pas nécessaires mais facilitent les
      notations par la suite                                          */
3  tant que y − x > 2ε faire
4  |  m ⟵ (x+y)/2;
5  |  si f(x) × f(m) ⩽ 0 alors
6  |  |  y ⟵ m;
7  |  sinon
8  |  |  x ⟵ m;
9  retourner (x+y)/2
```

6.4 – Exemple

Quelques itérations pour la résolution de l'équation $x^2 - 2 = 0$ sur $[0, 2]$.



- $f(0) = -2 < 0 < 2 = f(2)$ ou $f(0)f(2) \leqslant 0$ donc $\alpha \in [0, 2]$. On détermine le signe de $f(1)$.
- $f(1) = -1 < 0 < f(2)$ (ou $f(0)f(1) > 0$) donc $\alpha \in [1, 2]$. On détermine le signe de $f(1.5)$.
- $f(1.5) = 0.25 > 0 > f(1)$ (ou $f(1)f(1.5) \leqslant 0$) donc $\alpha \in [1, 1.5]$. On détermine le signe de $f(1.25)$.
- $f(1.25) = -0.4375 < 0 < f(1.5)$ (ou ...) donc $\alpha \in [1.25, 1.5]$. On détermine le signe de $f(1.375)$.
- etc.

6.5 – Exercices

Exercice 6.3

Pour a, b, p donné et $\varepsilon = 10^{-p}$, déterminer le nombre n de boucles exécutées par l'algorithme proposé.

Exercice 6.4

1. Écrire la fonction `dichotomie` (en traduisant l'algorithme proposé en `python`).
2. Dans l'algorithme de la dichotomie, on propose de retourner comme valeur non plus le milieu du dernier segment, mais l'extrémité dont l'image par f est la plus faible en valeur absolue. Écrire la fonction `dichotomie_extr` correspondante.
3. Le nombre d'itération étant prévisible, modifier la fonction initiale pour qu'elle soit écrite avec une boucle inconditionnelle à la place de la boucle conditionnelle. Vous nommerez cette fonction `dichotomie_cond`. *(Indication : Utiliser l'exercice 6.3)*

Exercice 6.5

La mise en œuvre des fonctions sera réalisée pour résoudre l'équation $x^2 - 2 = 0$ sur $\mathbb{R}_+$.

Remarque : Les graphiques seront abordés de manières complète au semestre 2. Pour réaliser les graphiques de cet exercice vous récupérerez le programme `graphedichotomie.py` sur moodle, dont le contenu est ci-dessous. Il ne vous restera plus qu'à compléter le code pour que la variable `N` **contienne la liste des valeurs de** n (nombre d'itérations) testées et les variables `precisions_th` et `precisions_eff` **contiennent les listes des précisions obtenues** (dans l'ordre) pour les différentes valeurs de n. On pourra faire varier n de 1 à 50.

1. Écrire la fonction `dichotomie_n` qui retourne le résultat après n itérations de la méthode de la dichotomie (n étant donné en paramètre à la place de la précision voulue).
2. Réaliser un graphique donnant la précision théorique obtenue (c'est-à-dire la longueur du dernier intervalle divisée par 2) en fonction du nombre d'itération, ainsi que la précision effective obtenue (c'est-à-dire l'écart entre la valeur obtenue et celle donnée par python `sqrt(2)`) pour la fonction `dichotomie_n`. Que constatez-vous ? Comment l'expliquer ?
3. Écrire la fonction `dichotomie_extr_n` qui retourne le résultat après n itérations de la méthode de la dichotomie où la valeur retournée est l'extrémité dont l'image par f est la plus faible en valeur absolue (n étant donné en paramètre à la place de la précision voulue).
4. Réaliser un graphique donnant la précision théorique obtenue (c'est-à-dire la longueur du dernier intervalle) en fonction du nombre d'itération, ainsi que la précision effective obtenue pour la fonction `dichotomie_extr_n`. Que constatez-vous ? Que peut-on en dire par rapport à la méthode de la dichotomie initiale ?

• Contenu du programme `graphedichotomie.py`.

```
import matplotlib.pyplot as plt
import numpy as np

'''
Ecrire ici le code pour obtenir les variables N, precisions_th et precisions_eff
N contient la liste des nombres d'iterations
precisions_th contient la liste des precisions theoriques correspondant a N
precisions_eff contient la liste des precisions effectives correspondant a N
'''

plt.plot(N, precisions_th, "r", label="Precision theorique")
plt.plot(N, precisions_eff, "b", label="Precision effective")
plt.grid()
plt.yscale('log')
plt.legend()
plt.show()
```

Exercice 6.6

Résolution sur $\mathbb{R}^+$ de l'équation (E) : $e^{-x} = 3 - 2x$.

Après une étude rapide de la fonction adéquate pour monter l'unicité de la solution de l'équation (E) et la localiser, déterminer une solution approchée à 10^{-15} près à l'aide de la méthode de la dichotomie.

Remarque :

Vous pourrez comparer les résultats obtenus avec la fonction `bisect` de la bibliothèque Scipy (pour bissection, autre dénomination de la dichotomie).

Dichotomie pour $x^2 - 2 = 0$:

```
In  [1]: import scipy.optimize as sc
In  [2]: sc.bisect(lambda t : t**2-2, 0, 2, xtol=1e-16)
Out [2]: 1.414213562373095
```

Exercice 6.7 ★

Écrire une fonction `racine` qui pour une valeur positive x donnée en entrée calcule une valeur approchée de $\sqrt{x}$ à 10^{-12} près à l'aide de la méthode de la dichotomie. Vous comparerez les résultats obtenus avec ceux donnés par la fonction `sqrt` de Python.

Remarque : Nous verrons au semestre 2 une méthode beaucoup plus efficace que la dichotomie pour effectuer ce calcul.

7 – Bibliographie

[1] B. WACK, *Informatique pour tous*, Éditions Eyrolles, 2013

[2] E. LE NAGARD, *Informatique, Initiation à l'algorithmique en Scilab et Python*, Éditions Pearson, 2013

[3] A. CASAMAYOU-BOUCAU, P. CHAUVIN, G. CONAN, *Programmation en Python pour les mathématiques*, Éditions Dunod, 2012

[4] G. SWINNEN, *Apprendre à Programmer avec Python*, Éditions O'Reilly, 2004

[5] S. CHAZALLET, *Python, Les fondamentaux du langage*, Éditions ENI, 2012