Le langage **Python**

I. Traits généraux

I.1. Typage dynamique

L'interpréteur détermine le type à la volée lors de l'éxécution du code.
Sur un ordinateur, toutes les données manipulées sont représentées par une suite de bits (une quantité qui vaut 0 ou 1) regroupés en octets (= 8 bits). Cette suite de bits est appelée la *valeur* de la donnée. Mais connaître la valeur de la donnée n'est pas suffisant pour déterminer de quel objet il s'agit, car des objet de natures différentes peuvent posséder la même valeur.

Exemple
L'entier `88` et le caractère `'X'` possèdent tous deux la valeur `01011000` dans un codage usuel sur un octet.

C'est pourquoi une donnée est représentée par sa valeur *et par un type* qui décrit la façon dont doit être interprétée cette suite de bits.

```
In [1] : type(88)
Out[1] : list

In [2] : type('X')
Out[2] : str
```

Lorsqu'on définit une donnée par l'intermédiaire du clavier, une analyse syntaxique est réalisée lors de l'exécution du code pour déterminer le type de l'objet souhaité, puis sa valeur est calculée.

$$2 \xrightarrow{\text{analyse syntaxique}} \texttt{int} \xrightarrow{\text{représentation}} (\texttt{int}, \texttt{00000010})$$

$$\texttt{'2'} \xrightarrow{\text{analyse syntaxique}} \texttt{str} \xrightarrow{\text{représentation}} (\texttt{str}, \texttt{00110010})$$

$$2. \xrightarrow{\text{analyse syntaxique}} \texttt{float} \xrightarrow{\text{représentation}} (\texttt{float}, \texttt{00010000})$$

À l'inverse, lorsqu'un objet doit être affiché sur l'écran, son type permet de déterminer quelle forme lui donner.

$$(\texttt{int}, \texttt{01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{88}$$

$$(\texttt{str}, \texttt{01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{'X'}$$

$$(\texttt{float}, \texttt{01011000}) \xrightarrow{\text{affichage à l'écran}} \texttt{1024}$$

Remarque
Cette détermination « à la volée » permet de ne pas avoir à déclarer le type des paramètres utilisés par une fonction. Néanmoins, pour des raisons aussi bien pédagogiques que de sûreté du code, il est possible de préciser le type des arguments et du résultat des fonctions (c'est ce que faisait l'école Centrale dans ses sujets d'informatiques). Ainsi :

```
def maFonction(n : int, X : [float], c : str, u) -> (int, numpy.ndarray) :
```

signifie que la fonction `maFonction` prend quatre arguments :

- le premier `n` est un entier,
- le deuxième `X` est une liste de flottants,
- le troisième `c` est une chaîne de caractère,
- le type du dernier, `u`, n'est pas précisé.

Cette fonction renvoie un couple dont :

- le premier élément est un entier,
- le second est un tableau `numpy`.

Néanmoins il ne vous est pas demandé d'utiliser cette syntaxe et vous pouvez utiliser l'écriture plus classique :

```
def maFonction(n, X, c, u) :
```

I.2. Principe d'indentation

Les impératifs de la programmation structurée nécessitent la définition de blocs d'instructions au sein des structures de contrôles (`def`, `for`, `while`, `if`...). Certains langages utilisent des délimiteurs pour encadrer ces blocs d'instructions (des parenthèses en C, des mots-clés en Fortran ou Scilab, etc), mais le langage **Python** se distingue en utilisant l'*indentation*, qui favorise la lisibilité du code.
Le début d'un bloc d'instructions est défini par un double-point (`:`), la première ligne pouvant être considérée comme un en-tête. Le corps du bloc est alors indenté d'un nombre d'espaces fixes (quatre par défaut), et le retour à l'indentation de l'en-tête marque la fin du bloc.

```
1  en-tête :
2      bloc ..................................
3      ......................................
4      d'instructions ........................
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres.

```
1  en-tête 1 :
2      ......................................
3      ......................................
4      en-tête 2 :
5          bloc ..........................
6          ..............................
7          d'instructions ................
8      ......................................
9      ......................................
```

Cette structuration sert entre autre à définir de nouvelles fonctions, à réaliser des tests ou à effectuer des instructions répétitives.

I.3. Portée lexicale

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

```
In  [3] :   L = [1, 2, 3]
In  [4] :   def f() :
....            L[0] = 0
....            return L
In  [5] :   f(), L
Out [5] :   [0, 2, 3], [0, 2, 3]
```

```
In  [6] :   L = [1, 2, 3]
In  [7] :   def f() :
....            L = [1, 2, 3]
....            L[0] = 0
....            return L
In  [8] :   f(), L
Out [8] :   [0, 2, 3], [1, 2, 3]
```

```
In  [9] :   def f() :
....            L[0] = 0
....            return L
In  [10] :  f()
NameError : name 'L' is not defined
```

Considérons les trois exemples précédents.

- Dans le premier cas, la liste `L` n'est pas définie au sein de la fonction `f`, donc c'est la liste définie dans l'espace global en ligne 1 qui est modifiée.
- Dans le deuxième cas, une liste `L` est définie au sein de la fonction `f` donc c'est elle qui est modifiée par la fonction.
- Dans le dernier cas, la liste `L` n'est définie ni dans l'espace de la fonction, ni dans l'espace global donc l'interpréteur renvoie une erreur.

I.4. Appel de fonction par valeur

L'exécution de l'appel `f(x)` *évalue d'abord* `x` *puis exécute* `f` *avec la valeur calculée.*
Observons le code suivant.

```
In [11] :   a = 2
In [12] :   def f(x) :
    ....        x = 3
    ....        return x
In [13] :   f(a), a
Out[13] :   3, 2
```

Dans un langage de programmation qui réaliserait un appel de fonction par *variable* plutôt que par *valeur*, le résultat retourné en sortie 3 serait `3, 3` (la variable globale `a` aurait été modifiée).
Il convient néanmoins de noter que si l'argument de la fonction est un objet *mutable* (typiquement une liste) alors l'objet en question est effectivement modifié, car la valeur d'un objet mutable est l'adresse où cet objet est stocké en mémoire (nous y reviendrons dans le cours sur les listes).

```
In [14] :   L = [1, 2, 3]
In [15] :   def f(X) :
    ....        X[0] = 0
    ....        return X
In [16] :   f(L), L
Out[16] :   [0, 2, 3], [0, 2, 3]
```

II. Types de base

II.1. Opérations sur les entiers (`int`)

Les opérations suivantes sont à connaître.

- `+` : l'addition de deux entiers,
- `-` : la soustraction de deux entiers,
- `*` : le produit de deux entiers,
- `**` : l'élévation à la puissance d'un entier (ne pas confondre avec `^` qui est une opération hors programme sur les entiers),
- `//` : le quotient de la division euclidienne d'un entier par un autre entier,
- `%` (*avec des opérandes positifs*) : le reste de la division euclidienne d'un entier par un autre entier.

Ces opérations renvoient toujours un résultat de type `int`.

II.2. Opérations sur les flottants (`float`)

Les opérations suivantes sont à connaître.

- `+` : l'addition de deux flottants,
- `-` : la soustraction de deux flottants,
- `*` : le produit de deux flottants,
- `/` : le quotient de deux flottants,
- `**` : l'élévation à la puissance d'un flottant.

Ces opérations renvoient toujours un résultat de type `float`.

II.3. Opérations sur les booléens (`bool`)

`not`, `or`, `and` *et leur caractère paresseux.*
Il convient d'expliquer la signification du caractère paresseux des opérateurs `or` et `and` : lors de l'évaluation d'une expression logique de type `(A and B)`, l'expression `A` est d'abord évaluée, *mais l'expression* `B` *n'est évaluée que si le résultat de l'évaluation de* `A` *est égal à* `True`. En effet, si `A` a été évaluée à `False`, l'expression `(A and B)` sera elle aussi évaluée à `False`, quelle que soit l'évaluation de `B`.
Pour des raisons analogues, lors de l'évaluation de l'expresion `(A or B)`, l'expression `A` est d'abord évaluée, puis l'expression `B` *mais uniquement dans le cas où* `A` *aura été évaluée à* `False`.

Pour comprendre l'intérêt de cette évaluation paresseuse, considérons une fonction qui recherche un élément dans une liste et renvoie l'indice correspondant à sa première apparition. Le code ci-dessous renvoie un résultat correct lorsque l'élément est présent dans la liste, mais déclenche une exception dans le cas contraire.

```
def recherche(x, L) :
    i = 0
    while L[i] != x :
        i = i + 1
    return i
```

```
In [17] : recherche(3, [1, 2, 3, 4])
Out[17] : 2
In [18] : recherche(5, [1, 2, 3, 4])
IndexError : list index out of range
```

Pour ne pas déclencher cette exception, il est nécessaire de veiller à ce que la variable `i` n'excède pas la longueur de la liste. Cependant, le code qui suit ne fonctionne pas, car le test ajouté ne figure qu'*après* l'instruction qui déclenche l'exception.

```
def recherche(x, L) :
    i = 0
    while L[i] != x and i < len(L) :
        i = i + 1
    return i
```

```
In [19] : recherche(5, [1, 2, 3, 4])
IndexError : list index out of range
```

En inversant les deux comparaisons, le caractère paresseux du `and` permet d'empêcher le déclenchement non désiré de l'exception.

```
def recherche(x, L) :
    i = 0
    while i < len(L) and L[i] != x :
        i = i + 1
    return i
```

```
In [20] : recherche(5, [1, 2, 3, 4])
Out[20] : 4
```

Avec cette dernière version, lorsque l'élément n'est pas présent dans la liste, cette fonction renvoie la longueur de cette dernière, mais il est facile de la modifier pour qu'elle ne renvoie rien dans ce cas.

```
def recherche(x, L) :
    i = 0
    while i < len(L) and L[i] != x :
        i = i + 1
    if i < len(L) :
        return i
```

II.4. Comparaisons

Sont à connaître les opérations de comparaison suivantes.

- `==` : l'égalité de deux expressions,
- `!=` : la différence de deux expressions,
- `<` : strictement inférieur,
- `>` : strictement supérieur,
- `<=` : inférieur ou égal,
- `>=` : supérieur ou égal.

Ces opérations de comparaison renvoient une valeur booléenne.

III. Types structurés

III.1. Structures indicées immuables (chaînes, tuples)

Définition

- Une structure de donnée est *indicée* lorsqu'on peut accéder à ses éléments individuels par l'intermédiaire de leur indice.
- Une structure de donnée est *immuable* lorsque ses éléments individuels ne peuvent être modifiés.

Deux structures de ce type sont à connaître.

- les chaînes de caractères (le type `str`) qui sont des suites de caractères alphanumériques délimités par des guillemets simples ou doubles. Par exemple `'La horde du contrevent'` est une chaîne de caractères.
- les tuples (le type `tuple`) ou n-uplets qui sont des suites finies de valeurs séparées par des virgules (si besoin enclos par des parenthèses). Par exemple, `(2, 3, 5, 7, 11)` est un tuple.

Ces structures partagent les opérations suivantes :

× la fonction `len` (de l'anglais *length*) permet de calculer leur longueur,

```
In [21] : len('La horde du contrevent')
Out[21] : 22
In [22] : len((2, 3, 5, 7, 11))
Out[22] : 5
```

× on accède aux éléments individuels avec la syntaxe `[k]` où `k` est un indice positif valide,

```
In [23] : 'La horde du contrevent'[4]
Out[23] : 'o'
In [24] : (2, 3, 5, 7, 11)[3]
Out[24] : 7
```

× on peut en calculer une tranche avec la syntaxe `[i:j]` qui extrait tous les éléments dont les indices sont compris entre `i` et `j` exclus,

```
In [25] : 'La horde du contrevent'[3:8]
Out[25] : 'noeud'
```

× ces structures peuvent être concaténées avec l'opérateur `+` et répétées avec l'opérateur `*`.

```
In [26] : 'Alain' + ' Damasio'
Out[26] : 'Alain Damasio'
In [27] : (1,) * 10
Out[27] : (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

(Notez au passage comment se définit un tuple de longueur `1`.)

III.2. Listes

Un chapitre complet sera consacré aux listes (et aux dictionnaires). On résume dans cette section et la suivante les essentiels à connaître sur ces deux types de variables.

III.2.a) Modes de création

Trois modes de création sont à connaître.

- par *compréhension* en suivant la syntaxe `[expr for x in s]`,
- par *duplication* en suivant la syntaxe `[expr] * n`,
- par `.append` successifs.

La création par duplication présente deux inconvénients :

× elle ne permet que de créer des listes de valeurs égales,

× *elle se révèle incorrecte lorsque cette valeur est de type mutable.*

Elle est donc plutôt à déconseiller.
Pour créer la liste des dix premiers carrés, on écrira donc l'une ou l'autre des versions suivantes :

```
1  carres = [x**2 for x in range(1, 11)]
```

```
1  carres = []
2  for x in range(1, 11) :
3      carres.append(x**2)
```

III.2.b) Opérations sur les listes

Tout comme pour les tuples et les chaînes de caractères :

× la fonction `len` renvoie la longueur d'une liste

× on accède aux éléments par indice positif valide,

× on peut en calculer une tranche et réaliser une concaténation de deux listes par l'opérateur `*`.

Cependant, il faut prendre garde au fait que les deux syntaxes `L.append(x)` et `L = L + [x]`, même si elles conduisent au même résultat apparent, *ne sont pas équivalentes*. En effet, la seconde version *recrée* une nouvelle liste augmentée d'un élément, là où la première se contente d'ajouter un élément à une liste existante. On s'en doute, la deuxième version peut se révéler beaucoup plus coûteuse pour une liste de grande taille. Pour cette raison, on utilisera avec la plus grande parcimonie l'opérateur de concaténation `+` pour les listes.

La principale différence avec les structures de données précédentes est que les listes sont des structures de données *mutables*, dans le sens où il est possible de modifier un élément individuel d'une liste sans avoir à recréer la liste dans son entièreté. Par ailleurs :

- La copie d'une liste `L` se réalise par la méthode `L.copy()` ou par le calcul d'une tranche comprenant l'entièreté de la liste `L[:]`.
- la méthode `L.pop()` supprime de la liste `L` son dernier élément et le renvoie en valeur de retour. Cette méthode modifie la liste en temps constant, contrairement à une syntaxe de type `L = L[0:len(L)-1]` qui recalculerait l'entièreté de la liste moins son dernier élément (et serait donc bien moins efficiente).

III.3. Dictionnaires

Un dictionnaire présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments individuels par le biais d'un indice, on y accède par le biais d'une *clé*.
La création d'un dictionnaire se réalise en suivant la syntaxe :

```
{c1 : v1, ..., cn : vn}
```

où :

× `c1`, ..., `cn` sont des clés (nécessairement deux-à-deux distinctes),

× `v1`, ..., `vn` sont les valeurs qui leur sont associées.

Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clé, alors :

× l'expression `c in D` renvoie un booléen indiquant si la clé est présente ou non dans le dictionnaire,

× `D[c]` renvoie la valeur associée à la clé si celle-ci est présente dans le dictionnaire (et provoque une erreur sinon),

× `D[c] = v` crée une nouvelle association si la clé n'est pas présente dans le dictionnaire et modifie l'association précédente sinon.

IV. Structures de contrôle

En informatique, on appelle **structure de contrôle** une commande qui contrôle l'ordre dans lequel les différentes instructions d'un programme sont exécutées. En **Python**, on peut distinguer plusieurs types de structures de contrôle :

a) les **structures séquentielles** : les différentes commandes sont exécutées les unes à la suite des autres *i.e.* dans un ordre séquentiel.

b) les **structures conditionnelles** : qui permettent d'introduire des branchements conditionnels dans le programme. Un programme peut comporter plusieurs branches. Chacune d'elle est associée à une condition. La branche exécutée est la première dont la condition est réalisée.

c) les **structures itératives** : qui permettent d'effectuer la répétition (*i.e.* l'itération) de commandes. Ces répétitions peuvent s'effectuer un nombre de fois fixé explicitement par le programmeur ou peuvent avoir lieu tant qu'une condition n'est pas réalisée.

IV.1. Instruction d'affectation

L'affectation se réalise avec `=`. Notons qu'il est possible de *dépaqueter* des tuples, autrement dit d'utiliser la syntaxe suivante pour affecter simultanément les valeurs `a`, `b`, `c` aux variables `x`, `y` et `z`.

```
1  x, y, z = a, b, c
```

Ceci peut s'avérer utile pour permuter le contenu de deux variables `x` et `y`. On écrirait alors :

```
1  x, y = y, x
```

IV.2. Instruction conditionnelle : le `if`

IV.2.a) Structure conditionnelle à deux branches

Syntaxe de la commande :
En programmation, il est parfois utile de réaliser un branchement : **si** une **condition** est réalisée **alors** le programme exécute les instructions de la première branche ; **sinon** le programme exécute les instructions de la deuxième branche. La syntaxe utilisée dans les langages de programmation, basée sur les mots clés `if` et `else`, est présentée ci-dessous.

```
if condition :
    instruction_1
else :
    instruction_2
```

Détaillons les éléments de cette syntaxe.

- `condition` : représente une expression **Python** dont l'évaluation est soit `True` soit `False`. Autrement dit, un objet de type `boolean`.
- `instruction_i` : représente une séquence de commandes **Python**, qu'on appelle aussi un bloc d'instructions.

Ordre d'exécution
L'exécution d'une structure conditionnelle à deux branches s'effectue de la manière suivante :

1) **si** `condition` est réalisée, *i.e.* si cette condition est évaluée à `True` **alors** le bloc `instruction_1` est exécuté.

2) **sinon**, autrement dit si `condition` n'est pas réalisée (*i.e.* évaluée à `False`), le bloc `instruction_2` est exécuté.

IV.2.b) Structures conditionnelles imbriquées

Syntaxe
Il faut bien comprendre que les blocs `instruction_1` et `instruction_2` sont des séquences d'instructions. Ces blocs peuvent donc comporter des structures conditionnelles. Par exemple, la deuxième branche d'une conditionnelle peut déboucher sur une nouvelle structure à deux branches, créant ainsi une structure conditionnelle à trois branches. La syntaxe d'une telle construction est la suivante.

```
if condition_1 :
    instruction_1
else :
    if condition_2 :
        instruction_2
    else :
        instruction_3
```

On peut itérer ce procédé : le bloc `instruction_3` peut lui aussi être un conditionnelle et ainsi de suite. Cela permet de créer des structures conditionnelles comportant un nombre (fini) quelconque de branches.

Ordre d'exécution
L'exécution de la structure imbriquée précédente est la suivante.

1) **si** `condition_1` est réalisée, **alors** le bloc `instruction_1` est exécuté.

2) **sinon**, autrement dit si `condition_1` n'est pas réalisée :

a) soit `condition_2` est réalisée, auquel cas le bloc `instruction_2` est exécuté.
(cas où `condition_1` *non réalisée et* `condition_2` *réalisée)*

b) soit `condition_2` n'est pas réalisée, auquel cas le bloc `instruction_3` est exécuté.
(cas où `condition_1` *et* `condition_2` *non réalisées)*

IV.2.c) Structure conditionnelles à n branches

Syntaxe

La syntaxe précédente décrivant une conditionnelle à trois branches est un peu lourde. L'écriture de conditionnelles à n branches est possible via la syntaxe allégée suivante.

```
if condition_1 :
    instruction_1
elif condition_2 :
    instruction_2
...
elif condition_n-1 :
    instruction_n-1
else :
    instruction_n
```

Ordre d'exécution

L'exécution d'une telle structure à n branches s'effectue comme suit.

1) **si** `condition_1` est réalisée, **alors** le bloc `instruction_1` est exécuté.

2) **sinon**, **si** `condition_2` est réalisée, **alors** le bloc `instruction_2` est exécuté.
(cas où `condition_1` *non réalisée et* `condition_2` *réalisée)*

...

n-1) **sinon**, **si** `condition_n-1` est réalisée, **alors** le bloc `instruction_n-1` est exécuté.
(cas où `condition_1`*, ...,* `condition_n-2` *non réalisées et* `condition_n-1` *réalisée)*

n) **sinon**, le bloc `instruction_n` est exécuté.
(cas où `condition_1`*, ...,* `condition_n-1` *non réalisées)*

IV.2.d) Le mot clé `else`

La dernière branche d'une structure conditionnelle est portée par le mot clé `else` mais n'est pas suivie d'une condition. En conséquence :

- soit au moins l'une des conditions est vérifiée. Le bloc `instruction_i` correspondant à la première condition réalisée sera alors exécuté.
- soit aucune condition n'est réalisée et le bloc `instruction_n` est exécuté.

Ainsi, quoiqu'il arrive, l'un des blocs `instruction_i` sera exécuté. On dit, dans ce cas, que le filtrage est **exhaustif**.

Formellement, il n'y a pas d'obligation, au niveau langage, à n'utiliser que des filtrages exhaustifs. L'utilisation du mot clé `else` peut donc être considéré comme une nouvelle règle de bonne conduite.

IV.2.e) Écrire des tests à l'aide d'opérateurs de comparaison

Les opérateurs permettant d'écrire des conditions sont présentés lors de la description du type `boolean` dans la fiche « Types de variables en **Python** ». Essentiellement, on dispose :

- des opérateurs de comparaison `>`, `<`, `>=`, `<=`, `==`, `!=` qui combinent des objets de type `float` ou `integer` pour former des expressions de type `boolean`.
- des opérateurs `or`, `and` qui permettent de combiner des expressions de type `boolean` pour en former de nouvelles.

Exemple
Pour illustrer notre propos, nous considérons de nouveau un programme consistant à calculer les racines réelles d'un polynôme du second degré.
Il faut un mécanisme de test permettant de s'assurer du caractère positif du discriminant avant d'en calculer la racine carrée. On propose donc le programme suivant contenant une structure conditionnelle.

```
1  # Le script suivant demande la valeur d'un polynôme
2  # et affiche ses deux racines réelles si elles existent
3  # ou un message "Attention..." dans le cas contraire
4  import math
5  a = float(input('Entrez la valeur du coefficient a : '))
6  b = float(input('Entrez la valeur du coefficient b : '))
7  c = float(input('Entrez la valeur du coefficient c : '))
8
9  delta = b**2 - 4*a*c
10 P = str(a) + 'X^2 + ' + str(b) + 'X + ' + str(c)
11
12 if delta > 0 :
13     xPlus = (-b + math.sqrt(delta)) / (2*a)
14     xMoins = (-b - math.sqrt(delta)) / (2*a)
15     print('Voici les deux racines du polynôme ' + P)
16     print('La plus grande racine vaut ' + str(xPlus))
17     print('La plus petite racine vaut ' + str(xMoins))
18 elif delta == 0 :
19     x0 = -b / (2*a)
20     print('Le polynôme ' + P + ' admet une unique racine')
21     print('Cette racine est ' + str(x0))
22 else :
23     print('Attention ' + P + ' n a pas de racine réelle')
```

Ce programme permet par exemple d'obtenir, en l'exécutant, l'affichage suivant.

```
In [28] : runfile('/Info/Exemple_cours/exemple_If.py',
          wdir='/Info/Exemple_cours')
          Entrez la valeur du coefficient a : 3
          Entrez la valeur du coefficient b : 1
          Entrez la valeur du coefficient c : 4
          Attention 3.0X^2 + 1.0X + 4.0 n a pas de racine
          réelles
```

IV.3. Structures itératives

IV.3.a) Les boucles bornées : boucles `for`

Syntaxe de la commande

- Syntaxe générale
  Une boucle `for` permet de réaliser la répétition d'un bloc d'instructions. La syntaxe de cette commande est la suivante.

```
for elt in liste :
    instruction
```

  Détaillons les éléments de cette syntaxe.
  - × `liste` : est une liste de valeurs.
  - × `elt` : est un élément qui prend successivement chacune des valeurs contenues dans `liste`.
  - × `instruction` : désigne un bloc d'instructions qui va être exécuté, de manière successive, pour toutes les valeurs de `elt`.
- Syntaxe du cas classique d'utilisation
  Généralement, la liste `liste` sera présentée sous la forme `range(n,m,pas)`. Ainsi, l'élément `elt` prendra successivement les valeurs de `n` à `m` -1 avec une distance de `pas`. Si la quantité `pas` n'est pas précisée, elle prendra par défaut la valeur 1. Si la quantité `n` n'est pas précisée, elle prendre par défaut la valeur 0.
  La syntaxe classique d'utilisation est la suivante.

```
for elt in range(n, m, p) :
    instruction
```

Ordre d'exécution

Afin de détailler la séquence d'exécution on considère l'exemple consistant à calculer les `m` premiers éléments d'une suite $(u_n)_{n\in\mathbb{N}}$ où `m` est une quantité choisie par l'utilisateur. Considérons ici la suite de terme général $u_n = n^3$.

```
# Calcul des m premiers éléments de la suite u_n = n^3
m = int(input('Entrez la valeur de m : '))
u = []
for i in range(m):
    u.append(i**3)
```

La boucle `for` s'exécute comme suit :

0) initialement, la variable `i` est affectée à la valeur `0`.
↪ l'instruction `u.append(0**3)` est alors exécutée.

1) la variable `i` est ensuite affectée à la valeur `1` (`0` + le pas de `1`).
↪ l'instruction `u.append(1**3)` est alors exécutée.

2) la variable `i` est ensuite affectée à la valeur `2` (`1 + 1`).
↪ l'instruction `u.append(2**3)` est alors exécutée.

...

m-2) la variable `i` est ensuite affectée à la valeur `m-2`.
↪ l'instruction `u.append((m-2)**3)` est alors exécutée.

m-1) la variable `i` est ensuite affectée à la valeur `m-1`.
↪ l'instruction `u.append((m-2)**3)` est alors exécutée.

En sortie de boucle, la liste `u` contient les `m` premières valeurs de la suite de terme général $u_n = n^3$. La variable `i`, quant à elle, contient alors la valeur `m-1`.

Remarque Dans cet exemple, on peut aussi obtenir une telle liste `u` par compréhension de liste :

```
u = [i**3 for i in range(m)]
```

IV.3.b) Les boucles non bornées : boucles while

Syntaxe de la commande

Une boucle `while` permet de réaliser la répétition d'un bloc d'instructions sous réserve qu'une condition est vérifiée. La syntaxe de cette commande est la suivante.

```
while condition :
    instruction
```

Détaillons les éléments de cette syntaxe.

- `condition` : représente une expression **Python** dont l'évaluation est soit `True` soit `False`. Autrement dit, un objet de type `boolean`.
- `instruction` : désigne un bloc d'instructions qui va être exécuté, de manière successive, **tant que** `condition` est vraie. Il est à noter que cette condition peut dépendre de variables qui sont modifiées lors de l'exécution de `instruction`.

Terminaison : dans le cas d'une boucle `while`, l'exécution se termine dès que *condition* n'est plus vérifiée. Si *condition* est toujours vérifiée (comme `0==0` ou `True`) le programme admet une exécution infinie. On parle de **boucle infinie**.

Ordre d'exécution

Afin de bien comprendre l'ordre d'exécution d'une boucle `while`, on va l'illustrer à l'aide d'un exemple consistant à trouver, dans une liste, le premier élément vérifiant une condition donnée. Commençons par générer une liste de valeurs aléatoires en important la fonction `rd.random` de la librairie `random`.

```
In [29] : import random as rd
In [30] : r = [rd.random() for i in range(8)]
Out[30] : [0.49,  0.18,  0.75,  0.94,  0.21,  0.58,
           0.26]
```

On demande alors à l'utilisateur d'entrer un réel `t` entre `0` et `1` dans le but de déterminer le premier élément de la liste `r` plus grand que `t`.

```
# Le 1er élément de r plus grand que le réel t demandé
t = float(input('Choisissez une valeur pour t : '))
i = 0
while r[i] < t:
    i = i + 1
print('Dans r, le 1er élément >= à t est ', r[i])
```

Si l'utilisateur choisit la valeur `0.5`, on obtient le retour console suivant.

```
In [31] : runfile('/Info/Exemple_cours/exempleWhile.py',
          wdir='/Info/Exemple_cours')
          Choisissez une valeur pour t : 0.5
          Dans r, le 1er élément >= à t est 0.75
```

Dans cet exemple, la variable `i` est initialement affecté à la valeur `0`. La boucle `while` s'exécute alors comme suit :

0) on teste si `r[0] < 0.5`.
↪ c'est le cas (`r[0] = 0.49`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `1` (`0 + 1`).

1) on teste si `r[1] < 0.5`.
↪ c'est le cas (`r[1] = 0.18`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `2` (`1 + 1`).

2) on teste si `r[2] < 0.5`.
↪ c'est le cas (`r[2] = 0.75`). Les instructions de la boucle sont donc exécutées. La variable `i` prend alors la valeur `3` (`2 + 1`).

3) on teste si `r[3] < 0.5`.
↪ ce n'est pas le cas (`r[3] = 0.94`). Les instructions de la boucle ne sont pas exécutées et il y a sortie de la boucle.

4) L'instruction d'affichage `print` est alors exécutée.

Si l'utilisateur choisit la valeur `0.95`, on obtient le retour console suivant.

```
In [32] : runfile('/Info/Exemple_cours/exempleWhile.py',
          wdir='/Info/Exemple_cours')
          Choisissez une valeur pour t : 0.95
IndexError : list index out of range
```

Pour comprendre ce message d'erreur, étudions l'exécution de la boucle `while` correspondante. La variable `i` est initialement affectée à la valeur `0`. Puis, on teste si les éléments successifs de la liste `r` sont strictement plus petits que `0.95`. Revenons sur le dernier test.

6) on teste si `r[6] < 0.95`.
↪ c'est le cas (`r[6] = 0.26`). Les instructions de la boucle sont exécutées. La variable `i` prend alors la valeur `7` (`7 + 1`).

7) on teste si `r[7] < 0.95`.
↪ ce test provoque l'affichage d'un message d'erreur. En effet, on tente un accès hors index puisqu'il n'y a que 7 éléments dans la liste `r`.

Il faut donc modifier le script du programme de sorte à ne pas réaliser d'accès hors index. Pour ce faire, il suffit de vérifier que `i` ne dépasse pas la taille de la liste avant de réaliser le test `r[i] < t`.

```
1  # Le 1er élément de r plus grand que le réel t demandé
2  t = float(input('Choisissez une valeur pour t : '))
3  n = len(r)
4  i = 0
5  while (i <= n-1) and (r[i] < t):
6      i = i + 1
7  if (i > n-1):
8      print('La liste r ne possède pas d éléments >= à t')
9  else:
10     print('Dans r, le 1er élément >= à t est ', r[i])
```

On teste ce nouveau script. Si l'utilisateur choisit la valeur `0.95`, on obtient maintenant le retour console suivant.

```
In [33] : runfile('/Info/Exemple_cours/exempleWhile.py',
          wdir='/Info/Exemple_cours')
          Choisissez une valeur pour t : 0.95
          La liste r ne possède pas d éléments >= à t
```

La modification a bien empêché l'accès hors index. Ceci est notamment dû au mécanisme d'évaluation de l'opérateur `and`. À chaque tour de boucle, la condition `(i <= n-1) and (r[i] < t)` est testée. On rappelle que l'évaluation de ce test se déroule comme suit.
Tout d'abord, l'interpréteur teste si `(i <= n-1)`.

(i) Si c'est le cas, `(r[i] < t)` est alors testé.

- ***a)*** Si c'est le cas, la condition est vérifiée et les instructions de la boucle sont exécutées.
- ***b)*** Sinon, la condition n'est pas réalisée et on sort de la boucle.

(ii) Sinon, la condition n'est pas réalisée et on sort de la boucle.

IV.3.c) Les différences entre les boucles for et while

Simuler une boucle for à l'aide d'un while

On peut simuler un `for` à l'aide d'un `while` en utilisant un compteur que l'on incrémente à chaque tour de boucle.
Sur l'exemple de la Partie **II**, ceci se fait de la manière suivante.

```
# Calcul des m premiers éléments de la suite u_n=1/n
m = int(input('Entrez la valeur de m : '))
u = []
i = 0
while i <= m-1:
    u.append(i**3)
    i = i + 1
```

Dans cet exemple, la variable `i` est initialement affectée à la valeur `0`. La boucle `while` s'exécute alors comme suit :

0) on teste si `0 <= m-1`.
↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[0]` prend la valeur `0` (`0**3`) et `i` prend la valeur `1` (`0 + 1`).

1) on teste si `1 <= m-1`.
↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[1]` prend la valeur `1` (`1**3`) et `i` prend la valeur `2` (`1 + 1`).

...

m-1) on teste si `m-1 <= m-1`.
↪ c'est le cas. Les instructions de la boucle sont donc exécutées. Ainsi `u[m-1]` prend la valeur `(m-1)**3` et `i` prend la valeur `m`.

m) on teste si `m <= m-1`.
↪ ce n'est pas le cas. Les instructions de la boucle ne sont pas exécutées et il y a sortie de la boucle.

Finalement, on obtient bien le même résultat que le précédent : la liste `u` contient à la fin du script les `m` premiers éléments de la suite de teme général $u_n = n^3$.

Boucle for ou boucle while : que choisir ?

Le choix de la structure itérative est fonction du problème que l'on a à résoudre. Plus précisément :

1) si l'on connaît a priori le nombre d'itérations à effectuer, on se tournera plutôt vers une boucle `for`.

- ***a)*** Modifier tous les coefficients d'une liste.
- ***b)*** Calculer les n premiers éléments d'une suite.
- ***c)*** Calculer une somme $\sum_{k=0}^{n}$ ou un produit $\prod_{k=0}^{n}$.

2) si l'on ne connaît pas a priori le nombre d'itérations à effectuer, on se tournera plutôt vers une boucle while.

- ***a)*** Trouver l'indice du premier coefficient non nul d'une liste.
- ***b)*** Pour une suite donnée, trouver l'indice du premier élément vérifiant une condition donnée.
- ***c)*** Connaissant le chiffre d'affaire \$ d'une entreprise et une estimation de son pourcentage de progression pour les 10 années à venir, calculer en combien de temps \$ sera doublé.

Exercice 1

1) Écrire une fonction `facto(n)` permettant le calcul de $n!$ à l'aide d'une structure itérative.

2) Écrire une fonction `binom(n, p)` permettant le calcul de $\binom{n}{p}$. On fera appel à la fonction précédente.

Exercice 2

Écrire une fonction `sommeInv(n,r)` calculant $\sum_{k=1}^{n} \frac{1}{k^r}$.

IV.4. Définition d'une fonction

La définition d'une fonction suit la syntaxe suivante.

```
def f(x1, ..., xn) :
    instruction
    return ...
```

En l'absence de `return`, la valeur renvoyée est égale à `None`.
Un `return` peut se trouver au sein du bloc d'instructions, auquel cas son exécution interrompt immédiatement l'exécution du code au sein de la fonction.

V. Divers

V.1. Commentaires

Les commentaires sont précédés du caractère `#`.

V.2. Utilisation simple de `print`

La fonction `print` renvoie un chaîne de caractère en direction de la sortie standard (par défaut la console dans laquelle s'exécute le code). À ne surtout pas confondre avec le résultat d'une fonction renvoyé par `return`.

V.3. Importation de modules

On importe l'entièreté d'un module par la syntaxe `import module` ou `import module as alias`. Par exemple :

```
import numpy
```

```
import numpy as np
```

Dans le premier cas, chaque fonction du module `numpy` devra être précédée du préfixe `numpy`, dans le second cas du préfixe `np` :

```
In  [34] : numpy.cos(numpy.pi)
Out [34] : -1.0
```

```
In  [35] : np.cos(np.pi)
Out [35] : -1.0
```

Il est aussi possible de n'importer qu'une seule fonction d'un module donné à l'aide de la syntaxe :

```
from module import fonction
```

V.4. Manipulation de fichiers texte

La documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est étudié.
Pour illustrer les différentes manipulations permises sur un fichier texte, nous allons prendre l'exemple d'un fichier CSV (pour *Comma-Separated Values*). Il s'agit d'un fichier texte représentant des données tabulaires sous formes de valeurs séparées par des virgules. Pour notre exemple, nous allons imaginer un fichier intitulé `naissances.csv` dont le contenu est le suivant :

Sexe	Prenom	Annee de naissance
M	Alphonse	1932
F	Beatrice	1964
F	Charlotte	1988

Avant de pouvoir être lu, un fichier texte doit être ouvert à l'aide de l'instruction `open`.

```
fichier = open('naissances.csv', 'r')
```

- Le paramètre `'r'` indique que l'accès au fichier se fera en mode lecture (*read*).
- La méthode `.read` lit tout le contenu d'un fichier et renvoie une chaîne de caractère unique.

```
In [36] : fichier.read()
Out[36] : Sexe, Prenom, Annee de naissance\nM, Alphonse,
          1932\nF, Beatrice, 1964\nF, Charlotte, 1988
```

Le caractère `\n` qui apparaît dans la réponse correspond à l'encodage d'un passage à la ligne.

- La méthode `read.line()` lit une ligne du fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de la méthode, la ligne suivante est renvoyée.

```
In [37] : fichier.readline()
Out[37] : 'Sexe, Prenom, Annee de naissance\n'
```

À la fin du fichier, cette léthode renvoie la chaîne de caractères vide `''`, ce qui permet de réaliser une itération conditionnelle d'un fichier. Cependant, il est plus simple dans ce cas d'utiliser une énumération du fichier :

```
1 for ligne in fichier :
2     print(ligne)
```

```
Out[38] : M, Alphonse, 1932
          F, Beatrice, 1964
          F, Charlotte, 1988
```

Notez que la première ligne n'apparaît pas car elle a déjà été lue par l'instruction 37, et que le caractère `\n` a bien été interprété comme un passage à la ligne par l'instruction `print`.

- La méthode `.readlines()` lit l'entièreté du fichier et renvoie une liste dont les éléments sont les lignes de ce fichier.

```
In [39] : L = fichier.readlines()
In [40] : L
Out[40] : ['Sexe, Prenom, Annee de naissance\n',
          'M, Alphonse, 1932\n', 'F, Beatrice, 1964\n',
          'F, Charlotte, 1988\n']
```

- Si on veut modifier le contenu d'un fichier, il faut l'ouvrir avec l'instruction `open` mais avec en option `'w'` (pour *write*) pour créer un nouveau fichier (dans ce cas, un fichier existant du même nom serait détruit) ou `'a'` (pour *append*) pour ajouter du texte supplémentaire à un fichier existant, puis utiliser la méthode `.write`. Par exemple, pour ajouter une ligne au format CSV donné en exemple, il faudrait écrire :

```
1 fichier = open('naissances.csv', 'a')
2 fichier.write('M, Dereck, 2002\n')
```

- Dans tous les cas de figure (lecture ou écriture), il est de bon ton de fermer un fichier ouvert une fois le travail terminé.

```
1 fichier.close()
```

- Pour utiliser les données d'un fichier CSV une fois lues, il faut être capable de séparer ces données textuelles qui, pour l'instant, sont fournies sous la forme d'une chaîne de caractères. La méthode `.split()` permet de séparer les différents éléments d'une chaîne de caractère. Son argument sert à préciser le séparateur utilisé.

```
In [41] : L[3]
Out[41] : 'F, Charlotte, 1988\n'
In [42] : L[3].split(', ')
Out[42] : ['F', 'Charlotte', '1988\n']
```

V.5. Assertion

Un `assert` est une aide au débogage qui, à l'entrée d'une fonction, vérifie si certaines conditions sont vérifiées. Pour qu'une fonction ne puisse s'appliquer qu'à un entier positif (par exemple pour définir la fonction factorielle), on écrira :

```
1  def facto(n) :
2      assert isinstance(n, int)
3      assert n >= 0
4      f = 1
5      for k in range(2, n+1) :
6          f = f * k
7      return f
```

```
In [43] :       fact(5)
Out [43] :      120
In [44] :       fact(-5)
AssertionError :
In [45] :       fact(5.)
AssertionError :
```