Chapitre 3

Introduction à l'informatique avec Python

Résumé

PYTHON *est un langage de programmation que nous allons utiliser en mathématiques pour différentes utilisations. Ce chapitre contient la totalité des structures de base qu'il faut connaître.*

Le reste des connaissances Python sera vu au fur et à mesure des chapitres de mathématiques.

Plan du cours

> *« J'ai une théorie. C'est en fait une grave erreur que notre cervelet n'ait pas été correctement connecté à notre cerveau. Il s'agit là sans doute du plus grand bug survenu dans notre programmation. Quelqu'un nous a mal conçus. C'est pourquoi on aurait dû nous remplacer par un autre modèle. Si notre cervelet avait été connecté au cerveau, nous aurions joui de la pleine connaissance de notre anatomie, des processus survenant à l'intérieur de notre corps. »*
>
> Olga Tokarczuk (1962 –). *Sur les ossements des morts*

Objectifs

La liste ci-dessous représente les éléments à maitriser absolument. Pour cela, il faut savoir refaire les exemples et exercices du cours, ainsi que ceux de la feuille de TD.

① les ensembles classiques ..... □

I. Expressions et types simples

1. Expression

Une expression est une suite de caractères définissant une valeur. Pour calculer cette valeur, la machine doit évaluer l'expression. Voici des exemples d'expressions : `42`, `1+4`, `1.2/3.0`, `x+3`.

Le résultat du calcul peut dépendre de l'environnement au moment où le calcul est effectué. Ainsi une expression est-elle plus complexe à évaluer qu'une valeur constante. On parlera des valeurs possibles d'une expression.

En Python, pour évaluer une expression, il suffit de la saisir dans un interpréteur, qui calcule et affiche alors la valeur qu'il a calculée :

Console Python
```
>>> 42
42
>>> 1+4
5
```

Les valeurs en Python sont typés selon l'objet qu'elles représentent. Une valeur peut ainsi être de type entier, de type flottant, de type chaîne de caractères ... Des types similaires existent dans la plupart des langages de programmation. Leur représentation en mémoire varie beaucoup d'un langage à l'autre, mais ce sont souvent les mêmes objets que l'on cherche à traduire.

Une expression en Python n'a pas de type a priori car le type de sa valeur dépend de l'environnement, plus précisément des types de ses sous-expressions. Pour simplifier, on considèrera dans un premier temps des expressions dont les valeurs sont toutes d'un même type. L'expression `42` est de type entier alors que `1.2 / 3.0` est de type flottant.

Pour afficher le type d'une expression après l'avoir évaluée, on utilise type :

Console Python
```
>>> type(42)
<class 'int'>
>>> type(1.2/3.0)
<class 'float'>
```

Le mot retourner par la fonction type indique le type de la valeur, entier (int pour integer en anglais) pour la première expression et flottant (float en anglais) pour la seconde.

Dans la plupart des langages de programmation, une expression est :

- soit une constante, comme `42` ;
- soit un nom de variable (valeurs littérales comme `x` ou `compteur`) ;
- soit une expression entre parenthèses, comme en mathématiques `(2-3)` ;
- soit composée de plusieurs expressions réunies à l'aide d'un opérateur, comme `1 + (3 * 4)` où les expressions `1` et `3 * 4` sont réunies par l'opérateur `+` ;
- soit composée d'une fonction appliquée à d'autres expressions, comme `fact(4)`.

Dans les prochaines sections, on présente les constantes et les opérateurs sur les types simples usuels, puis on verra la notion de variable. On présentera enfin des types plus complexes. La notion de fonction sera l'objet de la partie suivante.

2. Entiers

a. Constantes

Les constantes entières sont écrites en base 10 avec des nombres ayant autant de chiffres qu'on le souhaite.

On dit que Python utilise des entiers longs ; par abus de langage, on parle également de précision arbitraire pour exprimer que tous les chiffres significatifs sont mémorisés.

Pour écrire un nombre négatif, on utilise – devant le nombre.
Par exemple :

Console Python
```
>>> 42
42
>>> -12
-12
>>> 123456789123456789123456789
123456789123456789123456789
```

b. Opérateurs sur les entiers

Les opérateurs sur les entiers sont :

- l'addition utilisant le symbole `+` ;
- la soustraction utilisant le symbole `-` ;
- la multiplication utilisant le symbole `*` ;
- le quotient et le reste dans la division euclidienne utilisant respectivement les symboles `//` et `%`.

  Console Python
  ```
  >>> 17//5
  3
  >>> 17%5
  2
  ```

- l'exponentiation utilisant le symbole `**` (`n**m` calcule n^m). En combinant cela avec la précision arbitraire, on peut déjà faire des calculs élaborés :

  Console Python
  ```
  >>> 10**50
  100000000000000000000000000000000000000000000000000
  ```

- l'opposé utilisant également le symbole - mais devant une expression, on parle alors de position préfixe.

  Console Python
  ```
  >>> -(2+3)
  -5
  ```

> **Remarque**
> Dans le contexte de la programmation on parle de division entière et de modulo plutôt que de quotient et de reste. Ainsi l'expression `2 % 5` se lit 2 modulo 5 ;

c. Précédence des opérateurs sur les entiers

Si on considère l'expression `2+3*4`, il y a deux manières de l'évaluer : `(2+3)*4` ou `2+(3*4)`. Pour ne pas avoir à demander à l'utilisateur laquelle des deux expressions calculer, les langages de programmation appliquent automatiquement des règles de placement des parenthèses. Celles-ci généralisent les règles de priorité des opérations usuelles ; on parle de précédence.
Dans une expression, en l'absence de parenthèses, on calcule en priorité :

- les exponentiations (opérateur `**`) ;
- puis les multiplications, divisions entières et modulos (opérateurs `*`, `//` et `%`) ;
- et enfin les additions et soustractions (opérateurs `+` et `-`).

Remarque

Ces règles ne résolvent pas toutes les ambigüités. Il reste toujours le cas des expressions telles que `1-3-2`, qu'on peut comprendre comme `(1-3)-2` (de valeur -4) ou comme `1-(3-2)` (de valeur 0). Pour la plupart des opérateurs, on choisit toujours la première possibilité (associativité à gauche). Une exception notable est l'exponentiation, qui dans une expression comme `2**1**3` est calculée à droite d'abord. Dans le doute mettre des parenthèses !

3. Flottants

a. Constantes

Les nombres à virgules sont appelés dans le cadre de l'informatique des nombres à virgules flottantes. Pour écrire un nombre flottant on utilise le symbole . à la place de la virgule.

Console Python
```
>>> 3.2
3.2
>>> type(3.2)
<class 'float'>
>>> type(2.)
<class 'float'>
>>> type(2)
<class 'int'>
```

Remarque

Attention, l'exemple ci-dessus montre que `2.` et `2` ne sont pas le même objet, ils ne seront pas codés de la même façon en mémoire. Et notamment alors que en python les entiers ont une précision arbitraire (tant que l'ordinateur peut fournir la mémoire nécessaire) les flottants sont codés sur un nombre de bits fixes (64 généralement). Ils n'auront donc pas une précision arbitraire.

On peut vouloir écrire un flottant sous la forme $m \times 10^n$, où m la mantisse et n l'exposant sont des nombres relatifs. Pour traduire cela en Python, on utilise la lettre `e` pour séparer la mantisse et l'exposant :

Console Python
```
>>> 12e-4
0.0012
>>> -123e1
-1230.0
```

On remarque que Python ajoute `.0` même quand il s'agit d'un entier au sens mathématique du terme. Cela permet de rappeler d'un coup d'oeil qu'il est du type flottant.

b. Opérateurs sur les flottants

Les opérateurs +, -, *, ** sont également définis sur les flottants, avec les mêmes règles de précédence.

On peut remarquer ici que les flottants offrent une précision moins bonne que les entiers pour les calculs sur des valeurs entières :

Console Python
```
>>> 2**50
1125899906842624
>>> 2.0**100
1.2676506002282294e+30
```

⚠ **Attention**

Attention, lors d'une opération entre un flottant et un entier, l'entier considéré est converti à la volée en flottant.

Console Python
```
>>> 1.2*2
2.4
```

Les flottants disposent également d'un opérateur de division flottante /.

Console Python
```
>>> 1.0/3.0
0.3333333333333333
```

Remarque

Il est important de faire attention aux types des expressions sur lesquelles on travaille. Notamment pour savoir quels sont les opérateurs qu'on peut leur appliquer, mais également comme on l'a vu ici car cela a une incidence sur la précision du résultat mais également sur l'efficacité du calcul. Un processeur est généralement beaucoup plus rapide dans les opérations sur les entiers que sur les flottants.

Il faudra par exemple faire attention pour utiliser le nombre 42, on écrira 42 si l'on veut un entier et 42.0 ou simplement 42. si l'on veut un flottant.

On peut, en cas de besoin, convertir un entier en flottant en lui appliquant la fonction `float` et un flottant en un entier par la fonction `int`.

Attention toutefois : `int(x)` ne calcule pas la partie entière de x mais le tronque, c'est-à-dire calcule la partie entière de |x| puis affecte le résultat du signe de x.

4. Booléens

a. Constantes

Les booléens constituent un type spécial, le type bool, dont les constantes sont particulièrement simples : il n'y en a que deux, `True` et `False`. Ils servent à représenter le résultat de l'évaluation d'expressions logiques qui peuvent prendre soit la valeur vraie, représentée par `True`, soit la valeur fausse, représentée par `False`.

Console Python
```
>>> type(True)
<class 'bool'>
```

b. Opérateurs sur les booléens

Les opérateurs sur les booléens correspondent aux connecteurs logiques que l'on manipule en mathématiques :

- La négation ou non logique, dont le symbole Python est **not**. L'expression `not b` a la valeur `True` si b s'évalue à `False`, et la valeur `False` si b s'évalue à `True`.
- La conjonction ou et logique, dont le symbole Python est **and** . L'expression `b1 and b2` a la valeur `True` si b1 et b2 s'évaluent à `True`. Si une des deux expressions s'évalue à `False` alors `b1 and b2` a la valeur `False`.
- La disjonction ou ou logique, dont le symbole Python est **or**. L'expression `b1 or b2` a la valeur `False` si b1 et b2 s'évaluent à `False`. Si une des deux expressions s'évalue à `True` alors `b1 or b2` a la valeur `True`.

En voici quelques exemples :

Console Python
```
>>> True or False
True
>>> not True
False
>>> not (not True)
True
```

Remarque

Les opérateurs and et or sont dits paresseux : ils ne calculent que ce qui est nécessaire pour évaluer une expression. Par exemple :

Console Python
```
>>> 0 != 0 and 1/0 == 2
False
```

Le booléen situé à gauche de **and** valant **False**, celui de droite n'est pas évalué. Écrite dans l'ordre inverse, l'expression aurait produit une erreur de division par zéro. De même, l'opérateur **or** n'évalue pas son membre droit si son membre gauche vaut **True**.

c. Précédence des opérateurs sur les booléens

L'opérateur **not** a précédence sur **or** et **and**.

Console Python
```
>>> not True or True
True
>>> not (True or True)
False
```

L'opérateur **and** a précédence sur **or**.

Console Python
```
>>> True or False and False
True
>>> (True or False) and False
False
```

d. Opérateurs de comparaison

L'apparition la plus fréquente de booléens se fait lors de comparaisons d'autres types.

La comparaison la plus élémentaire est le test d'égalité. L'expression `e1 == e2` s'évalue au booléen **True** si e1 et e2 s'évaluent à des valeurs égales, sinon elle s'évalue à **False**.

Console Python
```
>>> 1 == 3-2
True
>>> 1 == 0
False
```

Python dispose d'un raccourci pour l'expression **not** `(e1 == e2)`, à savoir `e1 != e2`.

Les types que l'on va considérer dans ce cours sont pour la plupart également comparables pour une relation d'ordre fixée. On peut alors utiliser cet ordre pour comparer deux expressions.

L'ordre strict :

Console Python
```
>>> 1<3
True
```

ou dans l'autre sens :

Console Python
```
>>> 1>3
False
```

Pour effectuer des comparaisons au sens large, on ajoute = après le symbole de comparaison :

Console Python
```
>>> 1 <= 3
True
>>> 1 <= 1
True
```

Enfin, il est possible d'effectuer des comparaisons avec plus de deux éléments. On écrit alors cela comme en mathématiques :

Console Python
```
>>> 1<2<3
True
```

On peut également combiner les opérateurs :

Console Python
```
>>> 0 <= 1 < 2 == 4-2 < 5
True
```

II. Variables

1. Notion de variable

a. État, valeur d'une variable et valeur d'une expression

Pour mémoriser une valeur, on va la stocker en mémoire. On peut représenter l'adresse mémoire par un identificateur. Le couple (identificateur, valeur) est appelé une variable. En fait, il faut imaginer que l'identificateur est une version simplifiée de l'adresse mémoire à laquelle sera stockée la valeur de la variable.

Le nom de la variable est une chaîne de caractères, arbitrairement longue. Seuls quelques expressions (fonctions primitives de Python) ne peuvent être utilisés. Python distingue les majuscules, des minuscules. Ainsi, les chaînes `Aa` et `aa` sont deux identificateurs différents.

L'ensemble des variables définies à un instant donné de l'exécution d'un programme est appelé l'état. La valeur de toute expression contenant un nom de variable dépend donc de l'état courant. Lors de l'évaluation de l'expression, le compilateur remplace tous les identificateurs par leur valeur courante.

Au cours de l'évaluation d'une expression, l'état ne peut pas changer. Ainsi, dans l'expression `x*x`, aux deux occurrences du nom x sera substituée la même valeur.

Console Python
```
>>> nomunpeulong = 1234
>>> Nomunpeulong = 5678
>>> Nomunpeulong - nomunpeulong
4444
```

Si, lors de l'évaluation d'une expression, un nom de variable est utilisé alors qu'il n'apparaît pas dans l'état courant, il devient impossible d'attribuer une valeur à cette expression. Python renvoie alors le message suivant :

```
>>> x
Traceback (most recent call last):
```

```
NameError: name 'x' is not defined
```

b. Mots réservés

Comme nom de variable, on peut utiliser toute chaîne de caractères alphanumériques (qui ne commence pas par un chiffre) à l'exception de quelques mots réservés dont voici la liste des plus courants :

```
and assert break class continue def del elif else except exec finally for from global if import
in is lambda not or pass print raise return try while yield True False
```

c. Déclaration, initialisation et instruction

Déclarer une variable consiste à l'ajouter à l'état. En Python, cette déclaration est toujours accompagnée d'une initialisation. Elles se font en évaluant une instruction de la forme :
`nom de variable = expression`.
Jusqu'ici, on a uniquement évalué des expressions. Une instruction est une autre forme d'interaction qui demande d'effectuer une modification de l'état. En général, une instruction n'a pas de valeur après l'avoir exécutée, Python n'affiche rien. Par exemple :

Console Python
```
>>> x=2
```

redonne directement l'invite de l'interpréteur, non sans avoir modifié l'état. Il suffit d'évaluer une expression à la suite pour le constater :

Console Python
```
>>> x+1
3
```

d. Affectation

Pour changer la valeur d'une variable, on utilise la même instruction que pour la déclaration. Si on est dans l'état où x vaut 1, l'instruction :

Console Python
```
>>> x=2
```

fait passer dans l'état dans lequel x vaut 2.

> **Remarque**
> On note au passage que le symbole d'affectation est le signe =, même si l'affectation d'une valeur à une variable n'a rien à voir avec l'égalité en mathématique !

Pour ajouter 1 à la variable x, on écrit :

Console Python
```
>>> x = x+1
```

Dans cette instruction, `x` joue deux rôles bien différents. À gauche, il s'agit du nom de la variable sur laquelle s'effectue l'affectation. À droite, il apparaît comme élément de l'expression qui va être évaluée. Dans une affectation, on commence tout d'abord par évaluer l'expression de droite, puis on affecte cette valeur à la variable de gauche.

> **Remarque**
> Python apporte un raccourci à l'instruction `x=x+1`, cette dernière peut s'écrire de façon

équivalente `x+=1`, qu'on peut lire « ajouter 1 à `x` ». Les autres raccourcis sont présentés dans le paragraphe suivant.

e. Opérateurs avec assignation

Il est très fréquent d'avoir à incrémenter une variable, c'est-à-dire à augmenter sa valeur de 1. On a vu que Python permet de raccourcir l'instruction `x=x+1` en `x+=1`. On dit que `+=` est un **opérateur avec assignation**. Plus généralement

- `x += y` équivaut à `x = x + y`
- `x -= y` équivaut à `x = x - y`
- `x *= y` équivaut à `x = x * y`
- `x **=y` équivaut à `x = x ** y`
- `x /= y` équivaut à `x = x / y`
- `x //= y` équivaut à `x = x // y`
- `x %= y` équivaut à `x = x % y`

2. Types de variables

Une variable peut contenir des données de n'importe lequel des types que nous avons déjà étudié. En Python, il n'est pas nécessaire de préciser le type de valeur que l'on va placer dans une variable. En fait, un même identificateur peut être successivement associé à des types de données totalement différentes. En fait le type est déterminé seulement au moment de l'évaluation, c'est ce que l'on appelle un typage dynamique.

Console Python

```
>>> n = 1234 # ici n est un entier
>>> type(n)
<class 'int'>
>>> n + n
2468
>>> n= '1234' # ici n est une chaîne de caractères
>>> type(n)
<class 'str'>
>>> n + n
'12341234'
>>> n=[1,2,3,4]
>>> type(n)
<class 'list'>
>>> n+n
[1, 2, 3, 4, 1, 2, 3, 4]
```

On précisera un peu les types `str` (pour *string* ou chaine en français) et `list` dans la suite.

Exercice 3.1

On suppose que l'on a deux variable `x` et `y` dont on souhaite échanger le contenu. Écrire la suite d'instruction afin que `x` contienne le contenu de `y`, et `y` celui de `x`.

Solution

Une première idée serait de faire naïvement ceci :

Console Python
```
>>> x = 1
>>> y = 2
>>> x = y
>>> y = x
>>> x,y
(2, 2)
```

Le problème vient évidemment du fait que lorsqu'on modifie la valeur de y, celle de x a déjà été modifiée. Pour contourner cette difficulté, on peut utiliser une variable auxiliaire :

Console Python
```
>>> x = 1
>>> y = 2
>>> t = x
>>> x = y
>>> y = t
>>> x,y
(2, 1)
```

Python permet de réaliser plus simplement encore, de façon simultanée :

Console Python
```
>>> x = 1
>>> y = 2
>>> x,y = y,x # ou (x,y)=(y,x), ou [x,y]=[y,x]
>>> x,y
(2, 1)
```

III. Types composés

Sont de type composé les valeurs formées de plusieurs valeurs de types plus simples. Par exemple, les couples d'entiers sont de type composé.
De nombreuses constructions sont définies sur tous les types composés.

1. Les listes

Une liste en python est une collection (on peut dire aussi une séquence ou un tableau) ordonnée d'éléments quelconques (de n'importe quel type de python, ça peut par exemple être une autre liste). La liste est dite ordonnée car à chaque élément (on parlera également de composante d'une liste) est associé un numéro, son indice, donnant son rang dans la liste.
On se concentrera ici sur les éléments essentiels liés aux listes.

a. Construction

Pour construire une liste il suffit de placer ses éléments entre crochets séparés par des virgules.
Voici une liste composé d'un entier et d'un flottant :

Console Python
```
>>> [1,2.2]
[1, 2.2]
>>> type([1,2.2])
<class 'list'>
```

Il est bien sûr possible d'affecter une liste à une variable pour l'ajouter à l'état :

Console Python
```
>>> L = [1,2.2]
>>> L
[1, 2.2]
```

Remarque

Il est possible de construire une liste vide, c'est-à-dire une liste ne contenant pas d'éléments, avec l'instruction `[]`.

Par exemple, avec l'instruction :

Console Python
```
>>> V = []
```

on défini une liste `V`. Cela peut paraitre inutile au premier abord, mais l'on verra que c'est une façon d'initialiser une liste pour ensuite la faire grandir avec des éléments voulus. Le point important est que cela permet de dire à Python que `V` est de type `list`, on pourra donc l'impliquer dans des opérations spécifiques aux listes telle que la concaténation.

b. Accès aux composantes

Pour accéder aux composantes d'une liste, c'est-à-dire aux sous-valeurs qu'elle contient, on utilise l'expression `L[i]` où `i` est le rang de la composante, on parle de son **indice**.

En python, comme dans la plupart des langages de programmation, on commence à numéroter à partir de 0 et non de 1.

En ayant définie la liste L comme précédemment, pour obtenir sa première composante on pourra évaluer :

Console Python
```
>>> L[0]
1
```

et pour la seconde :

Console Python
```
>>> L[1]
2.2
```

c. Déconstruction

Il est également possible de déconstruire une liste en affectant simultanément ses composantes à différentes variables.

La liste `L` étant la liste précédemment définie, l'instruction :

Console Python
```
>>> x,y = L
>>> x
1
>>> y
2.2
```

permet de déclarer et d'initialiser les variables `x` et `y` avec respectivement les valeurs `L[0]` et `L[1]`.

d. Modification d'une composante

La composante d'une liste se comporte comme une variable, on peut la voir comme un emplacement mémoire.

De la même façon qu'avec une variable on peut affecter une valeur à une composante d'une liste déjà formée :

Console Python
```
>>> L
[1, 2.2]
>>> L[1] = 3
>>> L
[1, 3]
```

Tous les types composés proposés par python (tels que le n-uplets ou les chaines de caractères) ne permettent pas d'affecter de nouvelles valeurs aux composantes (voir le paragraphe sur les chaines de caractères). On dit de ces types qu'ils sont immuables. Les listes sont plus souples que ces types mais le prix à payer est qu'elles sont moins efficaces.

Remarque

Il n'est bien sûr pas possible d'affecter une valeur à une composante qui n'existerait pas. Python renvoie dans ce cas un message d'erreur. Par exemple :

```
L
L[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

e. Concaténation

Il est possible de coller une liste à n éléments à une liste à p éléments pour en obtenir une à $(n+p)$ éléments. On parle de **concaténation**. L'opérateur correspondant en Python est l'opérateur +.
Exemples :

Console Python
```
>>> [1,2] + [3,4,5]
[1, 2, 3, 4, 5]
>>> L1 = [1,2]
>>> L2 = [3,4,5]
>>> L1+L2
[1, 2, 3, 4, 5]
```

Cette opération est particulièrement utile pour faire grandir une liste à partir d'une liste initiale. Si on ajouter un élément à une liste, il faut bien faire attention à lui concaténer une liste à un 1 élément.
En effet, l'expression suivante produit une erreur :

```
>>> [1,2]+3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

Par contre la suite d'instructions suivantes permet de construire une liste à 1 élément :

Console Python
```
>>> L=[]
>>> L = L+[1]
>>> L
[1]
```

f. Ajout et suppression d'un élément : les méthodes append et pop

On présente ces méthodes bien qu'elles ne soient pas explicitement au programme car elles apparaissent la plupart du temps en annexe des sujets. Il faut donc savoir les utiliser.

Une méthode est une fonction associée à un type d'objet en particulier. On les appelle avec l'opérateur . à la suite d'un objet avec des expressions de la forme : `objet.methode(arguments)`.

La méthode **append** associée aux listes permet d'ajouter un élément en paramètre de la fonction en bout de liste.

Exemple :

Console Python
```
>>> L = [1,2]
>>> L.append(3)
>>> L
[1, 2, 3]
```

La méthode pop permet de supprimer le dernier élément d'une liste si on ne lui donne pas d'arguments et l'élément au rang donné en paramètre sinon.

Exemple :

Console Python
```
>>> L = [1,2,3,4]
>>> L.pop()
4
>>> L
[1, 2, 3]
>>> L.pop(0)
1
>>> L
[2, 3]
```

g. Longueur d'une liste

On obtient la longueur, c'est-à-dire le nombre d'éléments, d'une liste à l'aide de la fonction `len` :

Console Python
```
>>> len([2,3,5])
3
>>> len([])
0
```

h. Sous-liste : slicing

Le terme anglais de *slice* est associé à l'idée de découpage (une part de gâteau ou de pizza). En Python en particulier, un *slice* permet le découpage de structures de types composés comme les listes.

Les *slices* sont des expressions du langage Python qui vous permettent en une ligne de code d'extraire des sous-listes d'une liste.

Par exemple l'expression `L[i:j]` représente la sous-liste de `L` allant de l'indice `i` à l'indice `j` **exclu**.

Console Python
```
>>> L = [1,2,3,4]
>>> L[1:3]
[2, 3]
```

Pour extraire les éléments depuis le rang 0 jusqu'au rang i exclu on pourra utiliser le raccourci suivant `L[:i]`.

Et extraire les éléments depuis le rang i jusqu'au dernier on pourra utiliser le raccourci suivant `L[i:]`.

i. Test d'appartenance

Il est possible de tester si une valeur appartient à une liste à l'aide de l'opérateur **`in`** :

Console Python
```
>>> 3 in [1,2,3]
True
>>> 4 in [1,2,3]
False
```

2. Chaînes de caractères : `strings`

a. Construction

Le type des chaînes de caractères, *string* en anglais et dans Python, est celui permettant de représenter des textes. On considère dans un premier temps des textes élémentaires, ceux composés d'une unique lettre ; on les appelle les caractères.
En Python, les caractères peuvent être n'importe quelle lettre de l'alphabet, mais aussi des symboles, comme les signes de ponctuation :

Console Python
```
>>> 'a'
'a'
>>> type('a')
<class 'str'>
```

Une chaîne de caractères est une suite finie de caractères consécutifs, qu'on note entre apostrophes ou guillemets :

Console Python
```
>>> 'Ceci est une chaîne'
'Ceci est une chaîne'
>>> type("Voici une chaîne")
<class 'str'>
```

La chaine vide se note `''` ou `""`.

b. Accès à un caractère

Comme pour les listes, on peut stocker une chaîne dans une variable et accéder à chacun des caractères à l'aide de la construction **`s[i]`** :

Console Python
```
>>> s = "Ceci est une chaîne"
>>> s[1]
'e'
```

Par contre contrairement aux listes, les chaines sont **immuables**, on ne peut pas modifier leurs caractères un à un :

```
>>> s = "Ceci est une chaîne"
>>> s[1] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

c. Concaténation

Comme pour les listes, on concatène deux chaines à l'aide de l'opérateur + :

Console Python
```
>>> 'Bonjour '+'George !'
'Bonjour George !'
```

On remarquera au passage que l'espace au bout de la chaine 'Bonjour ' est un caractère comme un autre.

d. Longueur

Comme pour les listes, on utilise `len` pour obtenir la longueur d'une chaîne :

Console Python
```
>>> len("Bonjour")
7
```

e. Sous-chaines

Comme pour les listes on peut réaliser du slicing de chaines pour obtenir des sous-chaines avec la même syntaxe que pour les listes :

Console Python
```
>>> s='Hello world !'
>>> s[6:11]
'world'
```

f. Test d'appartenance

Comme pour les lites il est possible de tester si un caractère appartient à une chaine à l'aide de l'opérateur **in** :

Console Python
```
>>> 'o' in 'bonjour'
True
```

Toutefois cet opérateur n'étant pas au programme on cherchera à s'en passer dans les problèmes sauf si il est rappelé en annexe.

3. n-uplets

Le n-uplet (ou *tuple* en anglais) est un type de python très similaire aux listes. C'est aussi une collection d'éléments ordonnés quelconques.

Ce n'est pas un type au programme c'est pourquoi on ne s'y attardera pas.

On dira juste qu'on les construit en écrivant leurs éléments entre parenthèses séparés par des virgules. L'accès aux éléments se fait de façon analogue aux listes par contre ce sont des objets non mutables. Ils peuvent ainsi paraitres moins intéressants que les listes, leur intérêt est qu'ils sont beaucoup plus efficace pour certaines opérations.

Console Python
```
>>> n = (1,2,3)
>>> type(n)
<class 'tuple'>
>>> n[1]
2
```

IV. Instructions

1. Notion de programme

On peut utiliser Python en mode interactif, à la manière d'une calculatrice : les instructions (ou commandes) sont ainsi, l'une après l'autre, entrées au clavier, interprétées et suivies d'un résultat souvent réutilisé par la suite.

Si elle en vaut la peine, une séquence d'instructions peut être sauvegardée dans un fichier texte, un programme (on parle aussi de script Python). On peut dès lors ouvrir ce script et l'exécuter, de façon automatisée, comme si les instructions qu'il contient étaient à nouveau entrées au clavier (chronologiquement de la première à la dernière ligne). Dérouler les mêmes instructions dans le même ordre doit bien sûr posséder un minimum d'intérêt. Pour apporter un peu de profondeur et/ou de fantaisie à tout ça, on peut, à l'intérieur du script lui-même :

- répéter un certain nombre de fois un bloc d'instructions ;
- n'exécuter certaines instructions que si (ou que tant qu') une condition est vraie ;
- appeler un autre script ;
- orienter le déroulement du script suivant certaines informations fournies par l'utilisateur (informations qui pourraient par exemple être passées au démarrage du script : on parle alors des paramètres d'appel).

Évidemment, tout cela est possible en Python. On se contentera ici d'une première approche modeste.

2. Cinq types d'instructions

De manière générale, on appelle instruction un ordre de modification de l'état courant. Ce sont les éléments constitutifs des programmes. Leur assemblage, dans un ordre précis conduit au résultat attendu.

On peut distinguer deux types d'instructions : les instructions simples qui modifient directement l'état et les instructions composées qui assemblent d'autres instructions qui elles vont modifier l'état.

- la déclaration de variable qui rajoute cette variable à l'état ;
- l'affectation de variable qui modifie la valeur de la variable ;
- la séquence d'instructions qui exécute deux instructions à la suite l'une de l'autre ;
- l'instruction conditionnelle qui sert à n'exécuter une instruction que dans certains états ;
- la boucle conditionnelle ou inconditionnelle qui exécute plusieurs fois la même instruction.

Il est remarquable que ces 5 types d'instructions permettent, à elles seules, d'exprimer tous les algorithmes.

3. Entrées sorties

a. Affectation par l'utilisateur

Il existe une expression particulière, `input()`, qui attend que l'utilisateur saisisse une expression au clavier.

Pour être utile cette expression est alors affectée à une variable :

```
>>> n=input('entre ton age : n= ')
entre ton age : n= 23
>>> n
'23'
```

Le chiffre 23 est le nombre entré par l'utilisateur.

Remarque

Attention, on remarque sur l'exemple précédent que bien que l'utilisateur est entré un entier sans guillemets la variable n est une chaine de caractères. Retenir donc que la fonction input interprète systématiquement la donnée entrée par l'utilisateur comme une chaine de caractère. Si le but est de récupérer un entier pour ensuite pouvoir effectuer des calculs dessus il faut convertir la chaine de caractères en entier. Exemple :

```
>>> n=input('entre ton age : n= ')
entre ton age : n= 23

>>> n+2
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str

>>> n=int(input('entre ton age : n= ') )
entre ton age : n= 23

>>> n
23

>>> n+2

25
```

b. Affichage d'une valeur

Inversement, l'instruction `print` affiche à l'écran les instructions qui lui sont données en argument. Elle ne modifie pas l'état, mais seulement l'aspect de l'écran. On s'en sert souvent pour afficher des chaînes de caractères, mais elle peut afficher des expressions quelconques.

4. Séquences d'instructions

En programmation, pour combiner (ou imbriquer) des séquences d'instructions, on est amené à grouper des instructions successives, et à considérer ce groupe comme une seule instruction. Mais encore faut-il trouver un moyen de marquer (par des éléments du langage et/ou par des dispositions visuelles) les limites d'un tel groupe d'instructions.

Notons par exemple groupe une suite d'instructions `instruction 1`, `instruction 2`, ..., `instruction p`.

Imaginons également une condition `test`, vraie ou fausse à un moment donné du déroulement du script.

On cherche à écrire une séquence d'instructions du genre :

```
Commencer ici ...
Si la condition test est vraie , alors évaluer les instructions de groupe
De toutes façons , continuer là...
```

Voici deux formulations possibles. La première solution est pour le moins ambigüe (est-ce qu'on évalue uniquement `instruction 1` si la condition test est vraie ?).

```
# Formulation incorrecte
Commencer ici ...
Si la condition test est vraie, alors
instruction 1
instruction 2 ...
instruction p
De toutes façons , continuer là . . .
```

La seconde solution est correcte car elle indique clairement les limites du groupe d'instructions

à évaluer si la condition test est vraie.

```
# Formulation correcte
Commencer ici ...
Si la condition test est vraie, alors
Début de groupe
instruction 1
instruction 2
...
instruction p
Fin de groupe
De toutes façons , continuer là ...
```

En algorithmique, on délimite les groupes d'instructions par les mots clés Début et Fin.
En Python, deux instructions de même profondeur logique doivent avoir la même **indentation** (décalage par rapport à la marge de gauche). Du coup, avec une telle convention, il est inutile de délimiter les blocs d'instruction par un `begin group`, `end group`, par contre, il faut scrupuleusement respecter l'indentation.

```
en-tête1 : # entête du bloc 1
    instruction
    en-tête2 : # entête du bloc 2
        instruction
        ...
        instruction
    instruction # instruction en dehors du bloc 2
instruction # instruction en dehors du bloc 1 et du bloc 2
```

> **Remarque**
> On remarquera dans l'exemple générique précédent que chaque est introduit par un entête terminé par le symbole `:`, ça sera systématiquement le cas quelque soit la structure. Donc ne pas oublier ces `:` et l'indentation qui va derrière puisqu'en python un bloc doit contenir au minimum une instruction.

V. Instructions conditionnelles

1. Test simple

Une instruction conditionnelle n'est exécutée que si une condition est vérifiée par l'état courant. Pour la programmer, on utilise l'instruction

</> Code Python

```
if condition:
      instruction
      instruction
      ...
      instruction
```

Le bloc d'instruction n'est exécuté que si la condition est vérifiée. Dans l'exemple qui suit, on multiplie x par 3 et on lui rajoute 1 si x est impair.

</> Code Python

```
if x%2 != 0:
   x = 3*x+1
```

2. Test avec alternative

Suivant la valeur (vraie ou fausse) d'une condition, on peut dévier le flot d'instructions

</> Code Python

```
if condition:
    instruction
    instruction
    ...
    instruction
else:
    instruction
    instruction
    ...
    instruction
```

Dans le bloc suivant, on multiplie x par 3 et on lui rajoute 1 si x est impair, on le divise par 2 s'il est pair :

</> Code Python

```
if x%2 != 0:
    x = 3*x+1
else:
    x = x/2
```

a. Test imbriqués

Plutôt que d'emboîter des clauses `if`, on peut scinder le flot d'instructions suivant plusieurs cas :

</> Code Python

```
if condition1:
        instruction
        instruction
        ...
        instruction
elif condition2:
        instruction
        instruction
        ...
        instruction
elif condition3:
        instruction
        instruction
        ...
        instruction
else: # si toutes les conditions sont fausses on exécute ce bloc
        instruction
        instruction
        ...
        instruction
```

En pratique : pour écrire une instruction conditionnelle, avec alternatives, éventuellement imbriquées, vous devez

1. identifier les cas qui nécessitent un traitement différent
2. déterminer des expressions booléennes qui traduisent chaque cas
3. remplir les différents blocs d'instruction correspondant à chacun des cas en n'oubliant pas de respecter l'indentation !

3. Expressions conditionnelles

Python offre la possibilité de former des expressions dont l'évaluation est soumise à une condition. La syntaxe est la suivante
`expression1 if condition else expression2`.

Console Python
```
>>> x= -1 ; print('x positif' if x>0 else 'x negatif ou nul')
x negatif ou nul
>>> x = 0; print('x positif' if x>0 else 'x negatif' if x<0 else 'x nul')
x nul
```

VI. Boucles inconditionnelles

1. Boucles `for`

Pour répéter un certain nombre de fois un bloc d'instructions, on utilisera la construction suivante :

Code Python
```
for variable in objet:  # pour chaque élément de objet
    bloc d instructions # on parcourt ce bloc
```

En fait, objet est ici toute construction susceptible d'être parcourue : on pense bien sûr aux intervalles (qui utiliseront la notion de `range`), mais aussi aux chaînes (parcourues caractère par caractère), aux listes, aux n-uplets...

Code Python
```
for k in range(0,10):   # pour k = 0, puis k = 1, etc. jusqu'à k = 9
     bloc instructions  # on parcourt ce bloc
for x in 'abcdef':      # pour x = 'a', puis x = 'b', etc. jusqu'à x = 'f'
     bloc instructions  # on parcourt ce bloc
```

Le bloc qui fait suite à l'instruction for peut contenir deux instructions particulières, souvent attachées à un test `if` :

- `break` provoque la sortie immédiate de la clause `for`.
- `continue` passe directement à l'étape suivante de la boucle (ce qui reste du bloc après continue est donc ignoré).

Exercice 3.2

Écrire un programme en Python déterminant si un entier n, entré par l'utilisateur, est premier ou non.

Solution

Voici une proposition :

Code Python

```python
n=int(input('Entrez un entier, n= '))
premier = True
for k in range (2,n):
    if n % k == 0:      # S'il est divisible par k
        premier = False # alors n n'est pas premier
        break
if premier:
    print(n,'est premier')
else:
    print(n,"n'est pas premier")
```

2. Itérateurs

a. L'itérable range

L'intervalle `range(a,b,h)` doit être vu comme une succession de valeurs, en partant de a, et en progressant vers b (sans jamais l'atteindre !), dans le sens croissant ou décroissant selon que le pas h est positif ou négatif.

Ainsi :

- l'expression `range(7)` représente la succession des valeurs 0, 1, 2, 3, 4, 5, 6
- l'expression `range(1,7)` représente la succession des valeurs 1, 2, 3, 4, 5, 6
- l'expression `range(1,7,2)` représente la succession des valeurs 1, 3, 5
- l'expression `range(7,2)` est vide (ici le pas a sa valeur par défaut, c'est-à-dire 1)
- l'expression `range(7,2,-1)` représente la succession des valeurs 7, 6, 5, 4, 3
- si $a \leqslant b$, l'intervalle `range(a,b)` est formé de $b-a$ valeurs (il est notamment vide si $a = b$)

On fera bien attention que l'expression `range(a,b)` génère la liste des entiers de a inclus, à b exclu !

Pour tester l'appartenance d'une valeur à un intervalle, on utilise le mot réservé **`in`** (résultat **`True`** ou **`False`**).

⚠ **Attention**

Les intervalles dont il est question ici sont des échantillons de valeurs entières, ce ne sont donc pas des intervalles au sens mathématique du terme.

Voici quelques exemples :

Console Python

```
>>> r=range(100,1000,2) # intervalle des entiers pairs, de 100 inclus à 1000 exclu.
>>> 100 in r # 100 fait-il partie de l'intervalle, réponse oui.
True
>>> 1000 in r # 1000 fait-il partie de l'intervalle, réponse non.
False
```

b. Autres types itérables

Il y a un autre cas de figure qui nécessite de pouvoir faire une boucle, lorsqu'on veut manipuler une valeur de type composé : comme un n-uplet, une chaîne de caractères ou encore une liste. On aimerait pouvoir effectuer un traitement sur chacun des éléments de cette valeur, ce qui est raisonnable car

- on sait identifier le premier élément que l'on va traiter
- si on vient de traiter un élément, on sait quel est le suivant

Toute valeur de type composé pour laquelle ces deux opérations sont possibles est appelé un **itérable**. C'est donc le cas pour les chaînes de caractères et les listes.

Par exemple, faisons la somme des éléments d'un n-uplet :

Console Python
```
>>> t = (1, 5, -2.4, 4.9)
>>> somme = 0
>>> for e in t:
...     somme = somme + e
...
>>> somme
8.5
```

Exercice 3.3

Écrire un programme qui compte le nombre d'occurrences de la lettre 'e' dans un texte saisi par l'utilisateur.

Solution

Par exemple :

Code Python
```
texte = input('Saisissez votre texte ')
occurence = 0
for lettre in texte:   # on parcourt toutes les lettres
    if lettre == 'e':  # si la lettre est 'e'
       occurrence +=1  # on ajoute 1 au compteur
print('La lettre e apparaît ',occurrence,' fois dans votre texte')
```

VII. Boucles conditionnelles

Lorsque vous devez répéter un bloc d'instructions un certain nombre de fois bien déterminé, ou plus généralement lorsqu'on souhaite parcourir un itérable, on effectuera une boucle **`for`**. Mais lorsque le nombre d'itérations n'est pas connu à l'avance et que la décision d'arrêter la boucle ne peut s'exprimer que par un test, on choisira la boucle **`while`**.

1. Boucles `while`

Pour répéter un bloc d'instructions tant qu'une condition est réalisée, Python nous propose la clause while :

Code Python
```
while condition: # tant que la condition est vraie
    bloc si condition est vraie # alors on parcourt ce bloc
```

En pratique, pour écrire une boucle conditionnelle :

1. vous identifiez la condition de la boucle. Il est souvent plus facile de déterminer la condition de sortie de boucle : il faut alors prendre sa négation.
2. vous complétez le bloc d'instructions qui doit être répété (corps de la boucle) en s'assurant que la valeur de la condition pourra être modifié à certaines itérations !

Remarque

Quelques remarques classiques sur ce genre de construction :

- si la condition est fausse dès le départ, le bloc qui suit n'est jamais parcouru.
- dans la plupart des cas, le bloc qui suit l'instruction d'en-tête **`while`** agit sur la condition, de sorte que celle-ci, vraie au départ, devient fausse et provoque la sortie de la

clause.
- on peut écrire une clause **while** avec une condition toujours vraie (par exemple **while** 1: ou **while** True:) à condition (pour éviter une boucle infinie) de sortir par un autre moyen (notamment par **break** ou **return**).

Exemple 3.4 (Suites de Syracuse)

On appelle **suite de Syracuse** une suite définie par une valeur initiale x_0 et la règle suivante :

$$x_{n+1} = \begin{cases} \dfrac{x_n}{2} & \text{si} \quad x_n \text{ est pair} \\ 3x_n + 1 & \text{sinon} \end{cases}$$

La conjecture de Syracuse dit qu'à un certain rang n, x_n vaut 1 (et la suite boucle alors sur les valeurs $1, 4, 2, 1$). On étudie ici le comportement très intéressant obtenu pour la valeur initiale $x_0 = 27$:

Console Python

```
>>> x=27 # on part de la valeur 27
>>> while x != 1: # tant que est différent de 1
...     if x%2 != 0:
...         x = 3*x+1 # si x est impair, on le remplace par 3x + 1
...     else:
...         x = x // 2 # sinon, on le divise par 2
...     print(x, end=' ')
...
82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103 310 \
↪ 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251 754 \
↪ 377 1132 566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 \
↪ 3644 1822 911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 \
↪ 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

On constate que la suite atteint bien 1 au bout d'un (certain long) moment.

Le bloc qui fait suite à l'instruction **while** peut contenir deux instructions particulières, souvent attachées à un test **if** :

- **break** provoque la sortie immédiate de la clause **while**.
- **continue** ramène à l'évaluation de la condition (ce qui restait du bloc après continue est donc ignoré).

Exercice 3.5

Déterminez le rang du dernier terme strictement positif de la suite récurrente définie par $u_0 = 890$ et pour tout n, $u_{n+1} = \frac{1}{2}u_n - 3n$.

Solution

Par exemple :

Console Python

```
>>> u = 890
>>> n = 0
>>> while u > 0:          # tant que la suite est positive
...     u = u/2 - 3 * n   # on calcule le rang suivant
...     n = n+1           # et augmente le rang
...
>>> n-1                   # attention : le rang a été augmenté de 1 dans la boucle
5
```

VIII. Fonctions

Dans un même programme, une opération, ou une séquence d'opérations peut intervenir à plusieurs reprises.

En ce cas, il est intéressant de définir une **fonction** qui exécute cette instruction (ou ce bloc d'instructions). Elle pourra être appelée plusieurs fois dans un script ou même dans plusieurs scripts.

Ainsi, une fonction en Python, est comme un sous-programme à l'intérieur du programme principal. Il se constituera comme le programme principal d'un entête et du corps de la fonction. Il pourra aussi posséder ses propres variables, que l'on qualifiera de variables locales.

Organiser un programme à l'aide de fonctions présente de multiples avantages :

- cela permet d'éviter les redondances ;
- cela améliore grandement la lisibilité des programmes ;
- cela permet aussi de faciliter le développement du programme
  - en fractionnant le travail dans le temps : on peut s'occuper d'abord du programme principal puis des fonctions ;
  - ou de partager le travail entre plusieurs intervenants.

1. Syntaxe

Une fonction est définie par la donnée de

- son nom
- ses arguments (qui seront les valeurs auxquelles seront appliquées les instructions)
- éventuellement une valeur de retour, communiquée au programme principal à la fin d'exécution.

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

</> Code Python

```
def nom_de_fonction(liste_paramètres_séparées_par_virgule):
    instruction
    instruction
    instruction
    return valeur_de_retour
```

Le mot-clé `def` annonce la définition d'une fonction. Il doit être suivi par le nom de la fonction et une liste entre parenthèses de paramètres formels suivie de deux-points.

Une fonction peut ne pas avoir de paramètres, pour autant il faudra faire suivre le nom de la fonction de parenthèses sans rien dedans, par exemple `def mafonc():`. On trouve ensuite le corps de la fonction.

Exemple 3.6

Nous pouvons créer une fonction qui écrit la suite de Syracuse de premier terme a jusqu'à un rang n quelconque :

Code Python

```python
def vol(a,n) :
    """
    Retourne une liste contenant la trajectoire du vol d'altitude a jusqu'à l'étape n
    """
    u = a
    resultat=[a]
    for k in range (n):
        if u%2 == 0:
            u = u//2
        else:
            u=3*u+1
        resultat.append(u)
    return resultat

```

Ce qui est entre trois guillemets est ce qu'on appelle la **documentation** de la fonction, qui indique ce qu'elle fait. C'est bien sûr facultatif, surtout à l'écrit sur papier.

Dans cet exemple,

- `n` est le paramètre formel,
- `k, u, resultat` sont des variables locales qui sont initialisées et utilisées dans la fonction,
- la valeur finale de `resultat` sera retournée.

Le corps de la fonction, commencent sur la ligne suivant la déclaration, constitue un bloc d'instructions, il doit être indentée par des espaces pour le différencier du reste du programme.

Remarque

Lorsqu'on conçoit une fonction, il est préférable de lui donner un nom explicite. En effet la fonction est destinée à apparaître à plusieurs reprises dans le programme.

En revanche, les arguments formels peuvent être courts car leur portée, et donc leur signification est limitée au corps de la fonction.

2. Le retour de valeur

Certaines fonctions (comme la fonction `vol` définie plus haut) sont capables d'effectuer un calcul et non pas simplement d'afficher le résultat d'un calcul. Cela se fait avec l'instruction **`return`**.

Par exemple, on peut écrire une fonction `hypotenuse` qui à deux paramètres formels `x` et `y` associe le réel x^2+y^2.

Code Python

```python
def hypotenuse (x,y):
    resultat = (x**2+ y**2)**(1/2)
    return(resultat)
```

Attention

On ne doit pas confondre **`return`** et `print`. Il arrive de confondre ces deux instructions à cause de l'interpréteur en mode interactif qui affiche automatiquement la valeur resultat renvoyée par `f` lorsqu'on saisit `f(arguments)` exactement comme après une instruction `print(resultat)`.

Malgré les apparences leurs rôles sont en réalité totalement différents :

- l'instruction `print` n'a pas de valeur, elle a pour seul effet d'afficher un texte à l'écran.

- l'instruction **return** au contraire n'affiche rien mais décide de ce que renvoie la fonction, et donc de la valeur de l'appel `f()`.

Ainsi, si on commet l'erreur d'utiliser `print` à la place de **return** dans une fonction, celle-ci affichera la valeur calculée mais elle ne la renverra pas (elle retournera la valeur **None**). De manière générale, dans un programme ou un algorithme, il est important de distinguer ce qui doit être calculé des sorties affichées à l'écran.

Il est très important de faire la distinction entre un résultat calculé et un résultat simplement affiché. Dans une expression où il apparait une fonction on peut considérer que cette dernière est équivalente à ce qu'elle retourne. Il ne sera pas possible de réutiliser le calcul d'une fonction qui ne retourne rien.

Prenons un exemple explicite. Définissons la fonction `puissance` ainsi :

Code Python

```
def puissance (x,n):
    """
       Calcule puis affiche x puissance n,
       en supposant que x>0, reel ou entier, et n >= 0 un entier naturel
       Ne retourne rien !
    """
    r=1
    for i in range (n):
        r = r * x
    print(r)
```

La fonction puissance définie ainsi affiche le résultat mais ne le retourne pas. Si on tente d'inclure cette fonction dans une opération arithmétique python renverra un message d'erreur :

```
>>> puissance(2,3)
8
>>> 2*puissance(2,3)
8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

Dans la deuxième expression, python commence par exécuter `puissance(2,3)`, ce qui affiche 8 puis cherche à calculer 2 fois la valeur retournée par puissance. Cette dernière n'existant pas on comprend le message d'erreur.

Par contre on peut réaliser une telle opération avec la fonction `hypotenuse` définie précédemment car elle retourne une valeur :

Console Python

```
>>> hypotenuse(2,3)
3.605551275463989
>>> 2*hypotenuse(2,3)
7.211102550927978
```

Remarque

Comme break dans une boucle, l'instruction **return** a pour effet d'interrompre le déroulement de la fonction. Ainsi

Console Python
```
>>> def puissance (x,n):
...     if x==0:
...        return(0)
...     r=1
...     for i in range (n):
...        r = r * x
...     return r
...
>>> puissance(0,5)
0
```

Dans ce cas c'est le premier return qui est exécuté et seulement lui.

3. Portées des variables

Il existe deux types de variables, les variables locales et les variables globales.

Une variable globale est une variable déclarée dans l'espace de noms global. Une variable définie à l'intérieur d'une fonction est toujours locale.

Voici un exemple de création d'une variable globale `x` et d'une variable locale `y` :

Console Python
```
>>> x=7
>>> def f():
...     y=12*x # y est ici locale a la fonction
...     return y
...
>>> y = 1      # y est ici une variable globale ce n'est pas la meme que dans la fonction
>>> f()
84
>>> y
1
```

Du fait que la variable `y` est déclarée à l'intérieure de la fonction python l'inscrira dans les variables locales lors de l'exécution de la fonction. Même si une autre variable y est définie en dehors de la fonction (et donc globale) celle-ci ne pourra pas y avoir accès. Ainsi le script suivant renvoie un message d'erreur :

```
>>> def f():
...     print(y)
...     y=12*x
...     return y

>>> x=7
>>> y=1
>>> print(f())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: cannot access local variable 'y' where it is not associated with a value
```

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les constantes du problème.

Comme pour les fonctions, il est préférable de donner aux variables globales des noms longs et explicites, ce qui les distinguera de fait des variables locales qui portent habituellement des noms courts (comme les paramètres formels).

IX. Modules et paquetages

1. Les instructions `import`, `from`

Un module est associé à un fichier avec l'extension .py. Le module porte le nom du fichier et peut contenir des définitions de fonctions, de variables, aussi bien que des instructions exécutables. Ces instructions sont destinées à initialiser le module et ne sont exécutées qu'une seule fois à la première importation du module.

Pour importer un module on utilise le mot clef `import` et pour utiliser une fonction du module on fait précéder le nom de la fonction du nom du module suivi d'un point `module.fonction`.

Par exemple :

Console Python
```
>>> import random, math
>>> random.randint(0,10)
7
>>> math.pi
3.141592653589793
```

On peut aussi importer des noms particuliers d'un module. Dans ce cas, ces éléments sont directement référencés par leur nom : `randint, choice` sans qu'il soit nécessaire de les préfixer par le nom du module comme ci-dessus :

Console Python
```
>>> from random import randint, choice
>>> randint(0,10)
9
>>> choice(range(100))
79
```

Enfin, on peut donner un alias au nom du module pour raccourcir le nom :

Console Python
```
>>> import random as rd
>>> rd.randint(0,10)
3
```

Il y a une variante pour importer tous les noms d'un module, exceptés ceux qui commencent par un underscore (_) :

Console Python
```
>>> from random import *
>>> choice(range(10))
8
```

Dans la plupart des cas, on utilisera l'instruction `import ...` ou `import ... as ...`.

2. Modules standard

Python est livré avec une bibliothèque de modules standard, décrite dans un document séparé, Python Library Reference.

Nous utiliserons dans le cours de mathématiques les modules suivants, les seuls au programme :

- `math` : fonctions et constantes mathématiques de base $(\sin, \cos, \exp, \pi, ...)$;
- `numpy` : package Python dont le but principal est le traitement des tableaux multidimensionnels comme par exemple des vecteurs, matrices, images, ou encore des feuilles de calcul mais il contient aussi la plupart des fonctions définies dans le module `math` ;
- `random` : génération de nombres aléatoires ;
- `matplotlib` : un module utilisé pour la représentation graphique.

Ils seront explicités et détaillés dans les cours de mathématiques lors de leur utilisation.

En deuxième année, on ajoutera `Axes3D` afin de tracer des surfaces dans l'espace.

Exercices
3

Exercices

Généralités

●○○ Exercice 1 Découverte (10 min.)

Pour chacune des instructions suivantes, déterminer le résultat, puis vérifier dans une console :

Code Python
```
5+3
2-9
7+3*4
(7+3)*4
3**3
3**0.5
5/2
5.0/2
5/2.0
float(5)/2
4*2.5/3
```

Les commandes suivantes définissent et utilisent des variables (regarder ce qui se passe dans l'explorateur de variable au fur et à mesure) :

Code Python
```
x=10
x=x+1
largeur=20
hauteur=5*9.3
v=largeur*hauteur
print(v)
largeur=10
print(v)
```

Remarque

`x=x+1` peut aussi s'écrire `x+=1`.

Pour finir, tester les commandes :

Code Python
```
type(largeur)
type(hauteur)
type(v)
```

●○○ Exercice 2 Un premier `import` (5 min.)

Soit le programme suivant :

Code Python

```python
from math import pi
r = float(input("Entrez le rayon du disque : "))
s = pi*r**2
print("L'aire du disque est", s)
```

Que représentent les variables `r` et `s` ? Que font les commandes `input` et `print` ? En supprimant la ligne `from math import pi`, que se passe-t-il ?

●○○ Exercice 3 Une premier fonction (15 min.)

On considère un cylindre de hauteur h et de rayon r. On note S sa surface (c'est la somme de la surface latérale et de la surface des deux disques).

1. Créer un nouveau programme qui demande `h` et `r` et qui calcule, puis affiche, la surface du cylindre de rayon r et de hauteur h.
2. On suppose désormais qu'on impose le volume du cylindre à 0.33 L. Modifier le programme précédent pour qu'il demande `h` mais plus `r` : `r` est calculé en fonction de `h` de sorte que le volume du cylindre soit 0.33.
3. Plutôt que de demander la valeur avec un `input`, on préférera systématiquement introduire une **fonction**, qui prend un ou plusieurs arguments. Soit par exemple la fonction suivante :

Code Python

```python
from math import pi, sqrt # Permet d'importer pi et la racine carrée
def surf_canette(h):
  # h : hauteur de la canette (cm)
  # Résultat : surface de la canette de hauteur h, rayon r
  # r choisi de sorte que le volume soit 330 cm^3=33cl
  v=330
  r=sqrt(v/pi/h)
  return 2*pi*r**2+2*pi*r*h
```

En tapant ensuite dans la console :

Code Python

```python
surf_canette(5)
surf_canette(6)
surf_canette(7)
surf_canette(8)
surf_canette(9)
```

qu'obtient-on ?

4. En utilisant la fonction précédente, déterminer la valeur du rayon pour laquelle la surface est minimale.
5. Justifier mathématiquement la valeur obtenue à la question précédente.

●○○ Exercice 4 Le module `math` (10 min.)

On utilisera souvent le module `math` de PYTHON qui permet d'accéder aux fonctions du module en question. Pour cela, en début de programme, on écrira :

Code Python

```python
from math import *
```

qui signifie qu'on importe toutes les fonctions du module `math`.

1. Que renvoient les instructions suivantes, après l'import des fonctions du module `math` ?

Code Python
```
print(pi, sin(pi),cos(pi))
print(e, exp(1), log(e))
```

2. *Datation au carbone 14.* À la mort d'un être vivant, le carbone 14 présent dans son organisme se désintègre au fil des années de sorte que, si p est la proportion de C_{14} restant au bout de N années, alors $N = -8310\ln(p)$.
   a) Écrire une fonction `datation_C14(p)` qui calcule `N` en fonction de `p`.
   b) La momie Ötzi retrouvée dans un glacier en 1991 contenait 52.8% du C_{14} initial à 1% près. Donner un encadrement de son âge.
3. Au lieu d'importer toutes les fonctions du module `math`, on peut utiliser le module explicitement. Ainsi, en important ainsi :

Code Python
```
import math
```

on peut utiliser la fonction racine carrée en tapant `math.sqrt`.
Réécrire le programme précédent en utilisant cette manière d'importer le module `math`.

Boucles, instructions conditionnelles

●○○ Exercice 5 Des instructions conditionnelles (10 min.)

Les conditions mises dans le `if` sont des comparaisons. En PYTHON, les comparaisons s'écrivent naturellement :

- `a > b` : la valeur de `a` est-elle supérieure stricte à la valeur de `b` ;
- `a >= b` : la valeur de `a` est-elle supérieure ou égale à la valeur de `b` ;
- on de même `a < b` et `a <= b` ;
- `a == b` : la valeur de `a` est-elle égale à celle de `b` et `a != b` : la valeur de `a` est-elle différente de celle de `b`

⚠ **Attention**
On fera attention à ne pas écrire `if a = b:` mais bien `if a==b:`.

On dispose du programme suivant :

Code Python
```
def mystere(n):
    if n >= 10:
        return 1
    elif n<=5:
        return -1
    else:
        return 0
```

1. Que fait cette fonction ?
2. Écrire une fonction `def resultat_bac(n):` qui prend un flottant en argument, et qui renvoie 0 si la moyenne entrée ne lui donne pas le bac, 1 si le bac est obtenu sans mention, 2 s'il l'est avec la mention assez bien, 3 s'il l'est avec la mention bien, et 4 s'il l'est avec la mention très bien.

●○○ Exercice 6 Des divisibilités (10 min.)

1. Voici le résultat de quelques calculs :

Console Python

```
>>> 10//3
3
>>> 10%3
1
>>> 41//5
8
>>> 41%5
1
```

Que renvoie `a//b` ? Et `a%b` ?

2. Écrire une fonction `def pair(n):` qui renvoie 1 si le nombre est pair, 0 sinon.
3. Écrire une fonctoin `def divise(n, p):` qui renvoie 1 si n est divisble par p, 0 sinon.

●○○ Exercice 7 `range` et boucle (15 min.)

1. Déterminer les instructions `range` permettant d'obtenir les séquences suivantes :
   a) $2, 4, 6, 8, 10$.
   b) $0, 5, 10, 15, 20, 25, 30$.
   c) $7, 8, 9, 10, 11$.
2. Écrire une fonction `def somme_entier(n):` qui prend un entier `n` et renvoie la somme des entiers de 1 à n.
3. Écrire une fonction `def somme_carre(n):` qui prend un entier `n` et renvoie la somme des carrés des entiers de 1 à n.
4. Écrire une fonction `def somme_impaire(n):` qui prend un entier `n` et renvoie la somme des entiers impairs compris entre 1 et $2n+1$. Conjecturer, à l'aide des premières valeurs, la valeur de cette somme.
5. Démontrer la conjecture précédente par récurrence.

●○○ Exercice 8 La boucle `while` (15 min.)

On considère les fonctions suivantes :

Code Python

```
1  import random
2
3  def mystere1(n): # n est un entier entre 0 et 10
4    i = 0
5    alea = random.randint(0,10)
6    while (alea != n):
7      i = i+1
8      alea = random.randint(0,10)
9    return i
```

Code Python

```
1  def mystere2(n):
2    res=0
3    i=0
4    while (i<n):
5      res = res + i*i
6      ....
7    return res
```

1. En cherchant dans l'aide de Python, déterminer à quoi sert l'instruction ligne 5 de la fonction `mystere1`.
2. Tester la fonction `mystere1` sur plusieurs valeurs de `n`. Que fait-elle ?
3. Créer une fonction `def mystere1bis(n,p):` pour qu'elle prenne deux arguments : `n`, le nombre à chercher, et `p` la borne supérieure de l'aléatoire et qui se comporte ensuite comme la fonction `mystere1`. Ainsi, la fonction `mystere1` codée plus haut sera obtenue en faisant

`mystere1bis(n, 10)`.

4. Compléter la ligne 6 de la fonction `mystere2` pour qu'elle termine. Dans ce cas, que calcule-t-elle ?

Fonctions et listes

●○○ Exercice 9 Des fonctions (15 min.)

1. Écrire une fonction `def mult(n,p):` qui prend deux entiers `n` et `p` et qui renvoie le produit $n \times p$.
2. Écrire une fonction `def energie(m):` qui prend un flottant `m` en entrée et renvoie la quantité d'energie E avec $E = mc^2$ (on rappelle à toutes fins utiles que $c = 300000000\ m.s^{-1}$.)
3. Écrire une fonction `def majeur(n):`, qui prend un entier `n` comme argument, et qui renvoie `True` si $n \geqslant 18$, et `False` sinon.

On souhaite déterminer si une année est bissextile ou non.

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
  - Si c'est le cas, on regarde si elle est multiple de 400.
    - Si c'est le cas, l'année est bissextile.
    - Sinon, elle n'est pas bissextile.
  - Sinon, elle est bissextile.

4. Écrire une fonction `def bissextile(n):` qui prend comme argument un entier `n` et qui renvoie `True` si elle est bissextile, et `False` sinon.

●○○ Exercice 10 Bases sur les listes (20 min.)

1. On donne `L=[2,3,5,7,11,13,17,19,23,29,31]`. Créer une variable `cinq` contenant le cinquième élément de la liste `L`.

Pour parcourir tous les éléments d'une liste, il est souvent utile de disposer d'un compteur qui va de 0 jusqu'à $n-1$ où n est la taille totale de la liste. Pour accéder à cette dernière information, vous disposez de la fonction `len`, ce qui permet d'écrire un programme du type du suivant qui construit une liste `L2` constituée des carrés des éléments d'une liste `L1` donnée.

</> Code Python

```
n = len(L1) # On récupère la taille de la liste L1
L2 = [0]*n # On initialise la liste L2 avec des 0
for i in range(n): # Pour chaque position de la liste L1
    L2[i] = L1[i]**2 # on remplace le 0 de L2 par l'élément de L1 au carré
```

2. Stocker dans la variable `liste_des_cubes` la liste qui contient les cubes des éléments de la liste `L` de la question précédente.
3. Écrire une fonction `def cout_total(prix_au_kg, masse_voulue)` qui prend en argument deux listes (`prix_au_kg` et `masse_voulue`) qui représentent respectivement les prix au kilogramme de chaque ingrédient et la masse (en kg) nécessaire à la fabrication de l'onguent, et qui renvoie le coût total.

> **Remarque**
>
> Quand une liste `L` contient des autres listes, alors les éléments `L[i]` (pour `i` dans `range(len(L))`) sont eux-aussi des listes, donc peuvent accepter la syntaxe des crochets de telle sorte que `L[i][j]` (pour `j` dans `range(len(L[i]))`) est le j-ième élément de la i-ième liste.

4. On donne la liste `L=[[0,1],[2,3,4],[5,6]]`. Que donne `L[0][1]` ? `len(L)` ? `len(L[1])` ? Comment obtenir le nombre 4 ? Le nombre 5 ?
5. On donne la fonction suivante :

Code Python

```
def somme(L):
    resultat=0
    for i in range(len(L)):
        for j in range(len(L[i])):
            resultat=resultat+L[i][j]
    return resultat
```

Que fait ce programme ?

6. Modifier ce programme pour calculer la somme des carrés des nombres présents dans le tableau.

Pour aller plus loin

●●● Exercice 11 Carré magique (20 min.)

Un carré magique est une grille carrée (que l'on représentera en PYTHON avec une liste de listes) dans laquelle des nombres sont placés de telle sorte que la somme des nombres de chaque colonne, et chaque ligne soit la même. De plus, le carré doit contenir une fois chaque nombre, de 1 au nombre de cases de la grille.

1. On donne la fonction suivante :

Code Python

```
1  def ligne_magique(L):
2    n=len(L)
3    somme=0
4    for i in range(n):
5      somme=somme+L[0][i]
6    for i in range(1,n):
7        ligne=0
8        for j in range(n):
9          ligne = ligne + L[i][j]
10       if (ligne!=somme):
11         return False
12   return True
```

   a) Tester ce programme avec les deux listes `L=[[1,4],[2,3]]` et `M=[[1,2,3],[4,5,6],[7,8,9]]`.
   b) A quoi servent les lignes 4 à 5 ?
   c) Que fait-on sur les lignes 6 à 11 ?
   d) Que fait ce programme ?
2. Ecrire une fonction `def colonne_magique(L)` qui prend comme argument une liste de liste `L`, qui détermine si les colonnes vérifient la propriété ou non. Elle doit renvoyer `True` ou `False`.
3. Ecrire une fonction `def est_carre_magique(L)` qui prend comme argument une liste de liste `L` et qui détermine si le carré est magique ou non.

●●● Exercice 12 Appariement (20 min.)

En parallèle du grand marché de la ville, auquel vous accompagnez vos amis marchands, un ensemble de jeux sont organisés pour les habitants, en particulier la fameuse « course à 3 jambes » : cette course se déroule par équipes de deux personnes dont deux des jambes sont attachées par une corde.

Afin de constituer les équipes au hasard, une sorte de tirage au sort est organisé mais cela prend beaucoup de temps à faire manuellement, vous décidez d'aider les organisateurs en écrivant une fonction `appariement` qui prend en argument une liste d'entiers différents que chaque participant

a librement choisi.

Les équipes sont constituées ainsi : la personne ayant choisi le plus petit entier est avec celle ayant choisi le plus grand, celle ayant choisi le deuxième plus petit est avec celle ayant choisi le deuxième plus grand, et ainsi de suite.

Vous devrez renvoyer la liste des compositions de chacune des équipes (chaque personne est identifiée de manière unique par la position de son vote dans la liste donnée en argument), dans l'ordre : d'abord celle dont le plus petit numéro fait partie, puis celle dont le second plus petit numéro fait partie, et ainsi de suite. Au sein de chaque équipe on affichera d'abord le plus petit numéro puis le plus grand. On vous garantit que tous les numéros sont différents.

Voici un exemple :

Console Python

```
>>> tirage = [10, 32, 29, 45, 72, 2]
>>> appariement(tirage)
[[5, 4], [0, 3], [2, 1]]
```

On pourra utiliser la fonction `remove` qui enlève un élément d'une liste lorsque celui-ci existe :

Console Python

```
>>> M = [10,2,4,8,5]
>>> M.remove(8)
>>> M
[10, 2, 4, 5]
```

Corrigés

Corrigés des exercices

Exercice 1

Ce sont des instructions de base à connaître et maitriser.

Console Python

```
>>> 5+3
8
>>> 2-9
-7
>>> 7+3*4
19
>>> (7+3)*4
40
>>> 3**3
27
>>> 3**0.5
1.7320508075688772
>>> 5/2
2.5
>>> 5.0/2
2.5
>>> 5/2.0
2.5
>>> float(5)/2
2.5
>>> 4*2.5/3
3.3333333333333335
>>> x=10
>>> x=x+1
>>> largeur=20
>>> hauteur=5*9.3
>>> v=largeur*hauteur
>>> print(v)
930.0
>>> largeur=10
>>> print(v)
930.0
>>> type(largeur)
<class 'int'>
>>> type(hauteur)
<class 'float'>
>>> type(v)
<class 'float'>
```

Exercice 2

r demande à l'utilisateur un nombre, qui représente le rayon du disque. s désigne alors πr^2, c'est-à-dire la surface du disque.

Sans la ligne `from math import pi`, le programme ne peut fonctionner, car `pi` n'est pas connu.

Remarque

Ainsi, l'instruction `from math import pi` permet de récupérer la constante `pi` du module `math` afin de pouvoir l'utiliser ensuite.

Exercice 3

1. On rappelle que la surface d'un cylindre est composée de 3 parties : deux disques, de rayon r, et la face latérale, donc l'aire est donnée par la formule $2\pi rh$. On obtient le programem suivant :

Code Python

```
from math import pi
h = float(input("Hauteur ?"))
r = float(input("Rayon ?"))
S = 2*pi*r**2 + 2*pi*r*h
print(S)
```

2. Dans ce cas, on prend h et on en déduit r. Le volume d'un cylindre vaut $\pi r^2 \times h$. Puisque ce volume est fixé, on en déduit r. Attention, 0.33 L donne $330\,cm^3$.

Code Python

```
from math import pi, sqrt # Pour pi et racine
h = float(input("Hauteur ?"))
r = sqrt(330/(pi*h))
S = 2*pi*r**2 + 2*pi*r*h
print(S)
```

3. On obtient la même chose que précédemment, à savoir la surface de la canette en cm^2, mais qu'on peut appliquer autant de fois que l'on veut.
4. On peut faire un balayage, avec un certain pas, pour obtenir la surface la plus petite. Par exemple :

Code Python

```
def mini(pas):
  a = surf_canette(2) # Valeur de départ
  h = 2
  hmin = 2
  while h < 10:
    if surf_canette(h) < a:
      a = surf_canette(h)
      hmin = h
    h += pas
  return sqrt(330/pi/hmin)
```

ce qui donne :

Console Python

```
>>> mini(0.1)
3.742410318509558
```

5. La surface, dépendant de h, s'écrit

$$s(h) = 2\pi * \frac{330}{\pi * h} + 2\pi\sqrt{\frac{330}{\pi * h}} \times h = \frac{660}{h} + 2\sqrt{330\pi}\sqrt{h}$$

Cette fonction est dérivable sur $\mathbb{R}_+^*$ et on a, pour tout $h > 0$:

$$\begin{aligned} s'(h) &= -\frac{660}{h^2} + 2\sqrt{330\pi}\frac{1}{2\sqrt{h}} \\ &= \frac{1}{\sqrt{h}}\left(-\frac{660}{h^{3/2}} + \sqrt{330\pi}\right). \end{aligned}$$

Remarquons que, par croissance de la fonction $x \longmapsto x^{3/2}$ sur $\mathbb{R}_+^*$:

$$\begin{aligned} -\frac{660}{h^{3/2}} + \sqrt{330\pi} \geqslant 0 &\iff h^{3/2} \geqslant \frac{660}{\sqrt{330\pi}} \\ &\iff h \geqslant \left(\frac{660}{\sqrt{330\pi}}\right)^{2/3}. \end{aligned}$$

La fonction est alors décroissante avant cette valeur, et croissante ensuite : elle possède bien un minimum.
On obtient alors un rayon avec une surface minimal :

$$r = \sqrt{\frac{330}{\pi * \left(\frac{660}{\sqrt{330\pi}}\right)^{2/3}}}$$

de valeur approchée

$$r \approx 3.74493.$$

Exercice 4

1. Les instructions sont assez explicites. Cela donne :

Console Python

```
>>> print(pi, sin(pi),cos(pi))
3.141592653589793 1.2246467991473532e-16 -1.0
>>> print(e, exp(1), log(e))
2.718281828459045 2.718281828459045 1.0
```

On remarquera que cela n'affiche pas 0 mais une valeur approchée. On se souviendra que les résultats informatiques sont toujours des valeurs approchées.
2. On remarquera que le logarithme népérien, noté ln en mathématiques, est implémenté sous le nom de `log`. On obtient alors :

Code Python

```
1 from math import *
2 def datation_C14(p):
3   return -8310*log(p)
```

En appliquant, on obtient comme valeur approchée :

Console Python

```
>>> datation_C14(0.528)
5307.2562507425255
```

En réalité, on a une précision à $0,1\%$ près, donc, par croissance de la fonction ln :

$$0,527 \leqslant p \leqslant 0,529 \implies -8310\ln(0,529) \leqslant N \leqslant -8310\ln(0,527)$$

ce qui donne

$$5291,5 \leqslant N \leqslant 5323.$$

3. On remplace les fonctions par leurs versions pointées :

Code Python

```python
import math
def datation_C14(p):
  return -8310*math.log(p)
```

Exercice 5

1. La fonction renvoie 1 si l'entier est supérieur ou égal à 10, -1 si l'entier est inférieur ou égal à 5 et 0 sinon, c'est-à-dire si le nombre est strictement entre 5 et 10.
2. On enchaîne les `elif` :

Code Python

```python
def resultat_bac(n):
  if n < 10:
    return 0
  elif n < 12:    # nécessairement, n ⩾ 10
    return 1
  elif n < 14:    # nécessairement, n ⩾ 12
    return 2
  elif n < 16:    # nécessairement, n ⩾ 14
    return 3
  else:           # nécessairement, n ⩾ 16
    return 4
```

Exercice 6

1. `a//b` renvoie la division entière de `a` par `b`, c'est-à-dire le quotient de la division euclidienne de a par b.
`a%b` renvoie le reste de la division euclidienne de a par b.
2. On utilise ce qui précède : n est pair si et seulement si le reste de la division par 2 est nul. Ainsi :

Code Python

```python
def pair(n):
  if n%2 == 0:    # si le reste est nul, donc si n est pair
    return 1
  else:           # sinon
    return 0
```

3. De la même manière :

Code Python

```python
def divise(n,p):
  if n%p == 0:    # si le reste est nul, donc n est divisible par p
    return 1
  else:           # sinon
    return 0
```

Exercice 7

On utilise l'instruction `range(deb,fin,pas)`, qui va de `deb` (inclus) à `fin` (exclus) par pas de `pas`.

1. Rapidement :

</> Code Python

```
range(2,12,2)
range(0,35,5)
range(7,12)   # Le pas vaut 1 par défaut
```

2. On fait une simple boucle, en initialisant la somme à 0 :

</> Code Python

```
def somme_entier(n):
  somme = 0
  for i in range(n+1):     # Attention on veut la somme jusqu'à n
    somme = somme + i
  return somme
```

3. De même :

</> Code Python

```
def somme_carre(n):
  somme = 0
  for i in range(n+1):     # Attention on veut la somme jusqu'à n
    somme = somme + i**2
  return somme
```

Par exemple :

Console Python

```
>>> somme_carre(10)
385
```

4. On veut la somme des impairs : on fait par pas de 2 :

</> Code Python

```
def somme_impaire(n):
  somme = 0
  for i in range(1, 2*n+2, 2):     # Attention on veut la somme jusqu'à 2n+1
    somme = somme + i
  return somme
```

Remarquons les premières valeurs :

Console Python

```
>>> [somme_impaire(i) for i in range(1, 10) ]
[4, 9, 16, 25, 36, 49, 64, 81, 100]
```

On remarque alors qu'il semblerait que

$$\forall\, n \geqslant 1, \quad 1+3+5+\ldots+2n+1 = \sum_{k=0}^{n} 2k+1 = (n+1)^2.$$

5. On démontre alors par récurrence le résultat. Pour l'hérédité, on constatera par hypothèse de récurrence que

$$\begin{aligned}\sum_{k=0}^{n+1} 2k+1 &= \left(\sum_{k=0}^{n} 2k+1\right) + 2n+3 \\ &= (n+1)^2 + 2n + 3 = n^2 + 4n + 4 = (n+2)^2\end{aligned}$$

Exercice 8

1. L'instruction `random.randint(a,b)` permet de renvoyer un nombre entier aléatoire (ou plutôt, *pseudo-aléatoire*) entre `a` et `b`, tous les deux inclus. Dans l'année, on verra la même instruction, mais venant de la bibliothèque `numpy`. Dans ce cas-là, ce sera de `a` inclus à `b` exclus.

2. La fonction `mystere1` compte le nombre de lancer nécessaire avant d'obtenir le chiffre n.
3. On modifie simplement les bornes de l'instruction `random.randint` :

Code Python

```
import random

def mysterelbis(n,p): # n est un entier entre 0 et p
  i = 0
  alea = random.randint(0,p)
  while (alea != n):
    i = i+1
    alea = random.randint(0,p)
  return i
```

4. La fonction `mystere2` effectue une boucle qui s'arrête à la n-ième itération, lorsque i vaut n. Elle met dans `res` à chaque fois le terme i^2. Ainsi, elle calcule la somme des carrées de 0 à $n-1$. Pour que cela fonction, et que la boucle puisse s'arrêter, il faut augmenter i de 1 à chaque étape. Il faut donc rajouter à la ligne 6 :

```
i = i+1
```

Exercice 9

Ce sont des fonctions simples, pour se mettre en jambe sur les fonctions. Bien sûr, cela utilise éventuellement d'autres fonctions usuelles (instructions conditionnelles, par exemple)

Code Python

```
1  def mult(n,p):
2    return n*p
3
4  def energie(m):
5    return m*(300000000**2)
6
7  def majeur(n):
8    if n >= 18:
9      return True
10   else:
11     return False
```

On peut réécrire plus simplement la dernière fonction, en utilisant les booléens :

Code Python

```
1  def majeur(n):
2    return n>=18    # Renvoie True si c'est vrai, False sinon
```

Pour l'année bissextile, on utilise les divisibilités vues précédemment.

Code Python

```
1  def bissextile(n):
2    if n%4 == 0:
3      if n%100 == 0:
4         if n%400 == 0:
5           return True   # Multiple de 400, c'est oui
6         else:
7           return False  # Multiple de 100 uniquement, c'est non
8      else:
9        return True       # Multiple de 4 uniquement : c'est oui
10   else:
11     return False        # Pas multiple de 4 : c'est non
```

En utilisant les opérations sur les booléens (et, ou, ...), on peut être très efficace, à défaut d'être lisible :

Code Python

```python
def bissextile2(n):
    return (n%4==0 and n%100!=0) or (n%4==0 and n%100==0 and n%400==0)
```

Console Python

```
>>> bissextile2(2020)
True
>>> bissextile2(2021)
False
>>> bissextile2(2024)
True
```

Exercice 10

C'est un exercice d'utilisation des règles sur les listes.

1.

Code Python

```python
L=[2,3,5,7,11,13,17,19,23,29,31]
cinq = L[4] # Attention : on commence à 0 !
```

2.

Code Python

```python
n = len(L) # La taille de la liste L
liste_des_cubes=[0]*n # Liste de n éléments nuls
for i in range(n):
    liste_des_cubes[i] = L[i]**3 # On met le terme de L au cube
```

3.

Code Python

```python
def cout_total(prix_au_kg, masse_voulue):
  n = len(prix_au_kg)
  cout = 0 # La variable qui contiendra le coût
  for i in rang(n): # Pour chaque article
    cout = cout + prix_au_kg[i]*masse_voulue[i] # On ajoute le prix de l'article
  return cout
```

4. `L[0]` renvoie le premier élément de `L`, c'est-à-dire `[0,1]`. Ainsi, `L[0][1]` renvoie le 2e élément de cette liste, c'est-à-dire 1.
`len(L)` renvoie 3, car il y a trois éléments (eux même étant des listes). `len(L[1])` renvoie 3, car `L[1]` vaut `[2,3,4]` composé de trois éléments. On obtient 4 en faisant `L[1][2]` et 5 en faisant `L[2][0]`. On vérifie :

Console Python

```
>>> L=[[0,1],[2,3,4],[5,6]]
>>> L[0][1]
1
>>> len(L)
3
>>> len(L[1])
3
>>> L[1][2]
4
>>> L[2][0]
5
```

5. On parcourt toutes les listes de listes et on ajoute les éléments à la variable `resultat`. Cela calcule donc la somme de tous les termes présents dans la liste.
6. Il suffit de modifier l'ajout :

Code Python

```
def somme(L):
    resultat=0
    for i in range(len(L)):
       for j in range(len(L[i])):
          resultat=resultat+L[i][j]**2
    return resultat
```

Corrigés des exercices approfondis

Exercice 11

1. a) On a :

Console Python

```
>>> ligne_magique([[1,4],[2,3]])
True
>>> ligne_magique([[1,2,3],[4,5,6],[7,8,9]])
False
```

b) Elles servent à créer une variable `somme` contenant la somme des termes de la ligne 0 (i.e. de la première ligne). Dans le cas de `L`, cela vaut $1+4=5$ et dans le deuxième cas cela vaut $1+2+3=6$.
c) On vérifie, pour toutes les lignes, si la somme des termes valent `somme`. Si ce n'est pas le cas, on renvoie `False`.
d) A la fin, si on ne s'est pas arrêté avant, on renvoie `True`. Ce programme permet donc de vérifier si la somme des termes de chaque ligne donne la même valeur.
2. On s'inspire du précédent, mais en faisant sur les colonnes :

Code Python

```
def colonne_magique(L):
  n=len(L)
  somme=0
  for i in range(n):
    somme=somme+L[i][0] # Somme de la première colonne
  for i in range(1,n):
      colonne=0
      for j in range(n):
        colonne = colonne + L[j][i] # On ajoute les termes de la colonne i
      if (colonne!=somme):
        return False
  return True
```

3. On vérifie trois choses : que la somme des lignes soit la même (fonction `ligne_magique`, que la somme des colonnes soit la même (fonction `colonne_magique` mais aussi que cette somme soit la même sur les lignes et les colonnes :

Code Python

```
def est_carre_magique(L):
  if ligne_magique(L) and colonne_magique(L): # ça commence bien
     ligne = 0
     colonne = 0
     for i in range(len(L)):
```

```
        ligne = ligne + L[0][i]       # Somme sur la première ligne
        colonne = colonne + L[i][0]   # Somme sur la première colonne
      return ligne==colonne           # est-ce le même nombre ?
  else:
      return False                    # sinon, pas carré magique
```

Exercice 12

On va simplement parcourir la liste à chaque fois, en prenant le plus grand (via `max`) et le plus petit (via `min`). On les enlève via l'instruction `remove`.

Code Python

```
def appariement(L):
  if len(L)<=1: return L # S'il y a un seul élement, on ne peut rien faire
  M=L[:]                 # On créé une copie
  tirage=[]              # Notre tirage, pour l'instant vide
  while len(M)>=2:       # Tant qu'on peut faire des couples
    petit=min(M)         # On prend le plus grand
    grand=max(M)         # On prend le plus petit
    tirage.append([L.index(petit), L.index(grand)]) # On cherche l'index (i.e.
    M.remove(petit)      # la position du plus petit et grand, puis on les
    M.remove(grand)      # enlève de la liste.
  return tirage

tirage = [10, 32, 29, 45, 72, 2]

print(appariement(tirage))
```

Remarque

Pour ne pas détruire la liste de départ, on créé une copie de la liste. Pour cela, l'instruction `M = L` ne convient pas, car cela en réalité désigne la même liste. Il faut donc écrire `M = L[:]` pour forcer une copie.