

📍 Avenue V. Maistriau 8a
B-7000 Mons

☎ +32 (0)65 33 81 54

📧 scitech-mons@heh.be

WWW.HEH.BE

Gestion de projets

Cours aide python.

Bachelier en informatique et systèmes – Finalité
Télécommunications et réseaux – 2^{ème} Année.

Table des matières

1.	Variables et types de données simples.....	5
	Entiers	5
	Flottants.....	6
	Chaînes de caractères	6
	Modification d'une chaîne de caractères	7
	Propriété itérable des chaînes de caractères	8
	Longueur d'une chaîne de caractères.....	8
	Concaténation de chaînes de caractères.....	8
	Conversion en chaîne de caractères	9
	Booléens	9
2.	Types de données complexes	11
	Listes	11
	Définition	11
	Création d'une liste	11
	Accès aux données.....	12
	Fonctions de manipulation d'une liste	12
	Tuples.....	11
	Dictionnaires.....	11
3.	Opérateurs.....	12
	Opérateur d'affectation.....	12
	Opérateurs arithmétiques	12
	Addition	12
	Soustraction.....	13
	Multiplication	13
	Division	13
	Division entière	13
	Opération composée	14
	Opérateurs de comparaison.....	14
	Opérateurs d'appartenance	14
	Opérateurs logiques.....	15
4.	Structures conditionnelles	17
5.	Boucles.....	19
	Boucle while.....	19
	Boucle for	20
6.	Fonctions	22
	Cas classique	22
	Pas de valeurs retournées.....	23
	Plusieurs valeurs retournées	23

Paramètres par défaut.....	24
Rendre le code modulaire	26
7. Programmation orientée objet	27
Programmation objet et différences principales avec d'autres langages	28
Constructeur et attributs	29
Méthodes membres	32
Python et l'encapsulation	36
Méthodes spéciales	40
Héritage	43
Créer ses propres modules et packages.....	48
Modules.....	48
Packages	49
Travaux pratiques.....	50
Le problème	50
Solution.....	50
8. Définition et intérêt du GPIO	54
9. Broches d'entrée/sortie numériques	55
10. Port I2C.....	57
11. Acquérir des données analogiques.....	59
Étalonner le capteur.....	59
En pratique.....	60
12. Utiliser le Bluetooth	62
Lecture et écriture	65
Fonctionnement par notifications	67
13. Lire et écrire dans des fichiers	70
Ouvrir et fermer un fichier	70
Écriture	71
Lecture.....	73
Cas des fichiers binaires	75
Écriture	76
Lecture.....	77
14. Utiliser des CSV	79
15. Programmation multithreading et calcul parallèle	82
Utiliser les threads.....	82
Calcul multiprocessing.....	84
Joblib	86
16. Documenter le code	88
Tester le code	91
Unittest	92
Pytest	94

17.	Profiler le code.....	96
-----	-----------------------	----

1. Variables et types de données simples

Python n'est pas un langage fortement typé, contrairement à d'autres langages, comme C++ ou Java.

Cela signifie que lors d'une allocation d'une variable, vous n'êtes pas, comme en C, obligé de préciser le type de variable.

Par exemple, en C, l'allocation d'un entier est faite par l'instruction suivante :

```
int i = 0;
```

En Python, c'est l'interpréteur qui choisit le type de la variable au moment de la première affectation. Par exemple :

```
i = 500
```

Les types de base du langage sont les entiers (**int**), les flottants (**float**) et les complexes (**complex**), pour ce qui est des types numériques.

À ces derniers s'ajoutent les chaînes de caractères (**str**).

Pour rappel, un nombre entier n'a pas de valeur décimale. Un flottant, au contraire, est un chiffre possédant des décimales.

Entiers

Sur une architecture de type 32 bits, une variable de type **int** peut coder des valeurs comprises entre -2^{31} et $2^{31}-1$, soit une valeur comprise entre -2 147 483 648 et 2 147 483 647.

En Python 3, tous les entiers sont de type **int**. En effet, la recommandation PEP 237 a supprimé le type **long**, qui codait des nombres d'une taille supérieure aux entiers.

Nous avons vu que c'est lors de l'affectation que l'interpréteur Python détermine le type d'une variable. Pour les entiers, il existe plusieurs façons de réaliser l'affectation. Vous pouvez préciser la valeur représentée sous différents types de base. En Python, les représentations sont au nombre de trois :

- Binaire : les éléments unitaires prennent leur valeur entre 0 et 1. Pour utiliser cette représentation, il faut préciser **0b** dans la valeur en représentation binaire.
- Décimale : les éléments unitaires prennent leur valeur entre 0 et 9. C'est la représentation par défaut, il n'est pas nécessaire d'ajouter un symbole quelconque pour utiliser cette représentation.
- Hexadécimale : les éléments unitaires prennent leur valeur entre 0 et F (les valeurs sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Pour utiliser cette représentation, il est nécessaire d'ajouter **0x** devant la chaîne à représenter.

Voici quelques exemples que vous pouvez tester dans votre interpréteur : Représentation

binaire

```
In [19]: i = 0b10100
```

```
In [20]: print(i)
```

```
20
```

Représentation décimale

```
In [12]: i = 6789
```

```
In [13]: print(i)
```

```
6789
```

Représentation hexadécimale

```
In [21]: i = 0x1F
```

```
In [22]: print(i)
```

```
31
```

Flottants

Le type flottant (`float`) permet de représenter toutes les valeurs numériques décimales (les chiffres à virgule).

Comme pour les entiers, c'est l'interpréteur qui réalise le choix du type lors de la première affectation. Par exemple :

```
i = 3.98
```

Si la valeur de la première affectation est un entier, et si vous savez que votre variable va contenir dans le futur des valeurs décimales, il est de bonne pratique de forcer une valeur décimale lors de la première affectation afin de rendre le code plus lisible et de lever tous les risques d'ambiguïté.

Par exemple, il faut préférer la syntaxe suivante :

```
i = 3.0
```

à

```
i=3
```

Il est également possible de convertir un entier en flottant à n'importe quel moment dans le code grâce à la fonction de conversion `float()`.

```
In [37]: i = 3
```

```
In [38]: type(i) Out[38]: int
```

```
In [39]: i = float(i)
```

```
In [40]: type(i) Out[40]: float
```

Nous avons présenté ici la méthode de conversion d'un entier en flottant. La conversion d'un flottant en entier est également possible. Par exemple `i = int(3.2)`. Si cette conversion est possible, il est nécessaire de savoir qu'elle est dangereuse puisque la partie flottante (après la virgule) sera tronquée. L'exemple ci-après l'illustre :

```
In [23]: i = int(3.2)
```

```
In [24]: type(i)
```

```
Out[24]: int
```

```
In [25]: print(i)
```

```
3
```

Chaînes de caractères

Pour vulgariser, on peut dire qu'une chaîne de caractères (`Str`) peut contenir toutes sortes d'éléments pouvant être représentés sous forme de texte.

Cependant, la représentation du texte est quelque chose de complexe et un problème récurrent. En effet, la prise en charge des caractères de différents alphabets ainsi que de symboles spéciaux a toujours été un

« serpent de mer » pour la communauté informatique. Les langages ont suivi les normes internationales de représentation qui, au fil du temps, gèrent de plus en plus d'alphabets et de caractères.

Pour ce qui est de Python, la version 3.5 adopte l'encodage UTF-8. L'avantage de l'UTF-8 est qu'il couvre un grand nombre de symboles et donc, pour les Français, il n'est plus nécessaire de réaliser d'opération supplémentaire pour manipuler des chaînes avec des caractères accentués. C'est une évolution confortable pour le développeur par rapport aux versions antérieures.

Comme pour les entiers et les flottants, c'est l'interpréteur qui réalise le choix du type de variable pour stocker des chaînes de caractères. La contrainte étant de mettre la chaîne de caractères entre simples ou doubles quotes, soit respectivement " ou " ".

```
In [43]: s = "Chaîne de caractères"
```

```
In [44]: type(s) Out[44]: str
```

Une chaîne de caractères est ni plus ni moins qu'un tableau de valeurs contenant le code de représentation de chaque caractère.

Par conséquent, il est possible d'accéder aux valeurs de la chaîne à partir de ce tableau :

```
In [43]: s = "Chaîne de caractères" In [44]: type(s)  
Out[44]: str
```

```
In [45]: s[0]  
Out[45]: 'C'
```

```
In [46]: s[3]  
Out[46]: 'i'
```

Modification d'une chaîne de caractères

Une chaîne de caractères n'est pas directement modifiable après son affectation à une variable. Il est impossible de réaliser directement la modification suivante :

```
In [47]: s[3] = 'k'
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-47-5780f0e03450> in <module>()  
----> 1 s[3] = 'k'
```

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

Pour modifier une chaîne de caractères, il est nécessaire d'utiliser les fonctions liées au type `str`, par exemple `replace()`, `upper()`.

La première remplace un caractère par un autre :

```
In [48]: s.replace('a', 'e') Out[48]: 'Cheine de  
cerectères'
```

La seconde met la chaîne en majuscules :

```
In [49]: s.upper()  
Out[49]: 'CHAINE DE CARACTÈRES'
```

Il existe de nombreuses fonctions liées au type `Str`. Nous ne les traiterons pas dans cet ouvrage, car elles ne seront pas nécessaires au développeur Python pour la cible Raspberry Pi. Toutefois, elles sont toutes documentées dans l'aide de référence du langage Python :
<https://docs.python.org/fr/3.5/library/stdtypes.html#str>

Propriété itérable des chaînes de caractères

Une autre propriété du type `str` est qu'il est itérable. Par conséquent, le parcours via une boucle `for` est très simple :

```
In [51]: for c in s:
...:     print(c)
...:
C
h
a
i
n
e

d
e

c
a
r
a
c
t
è
r
e
s
```

Nous verrons un peu plus loin dans ce chapitre l'usage de la boucle `for`.

Longueur d'une chaîne de caractères

Il est très souvent nécessaire de connaître la longueur d'une chaîne de caractères. Pour cela, il faut utiliser la fonction `len()`.

```
In [52]: len(s)
Out[52]: 20
```

La valeur retournée par `len` est un entier et nous pouvons le stocker dans une variable comme ceci :

```
In [54]: i = len(s)

In [55]: print(i)
20
```

Concaténation de chaînes de caractères

La concaténation consiste à mettre bout à bout deux chaînes de caractères pour en former une nouvelle. En Python, cette opération est réalisée à l'aide de l'opérateur `+`.

```
In [1]: a = "Bonjour" In [2]: b
= "Monsieur" In [3]: c = a + b
```

```
In [4]: print(c)
```


BonjourMonsieur

Conversion en chaîne de caractères

Il est possible de convertir une valeur entière ou flottante en chaîne de caractères. Pour cela, il suffit d'utiliser la fonction `str()`. L'exemple suivant illustre l'insertion d'une valeur entière dans une chaîne de caractères.

```
In [12]: a = str(2)
```

```
In [13]: b = "la valeur de a est : " + a
In [14]: print(a)
2
```

Une des limites de cette méthode est qu'il est impossible, dans le cas de nombres décimaux, de préciser le nombre de décimales après la virgule. Pour aller au-delà de cette limite, il est nécessaire d'utiliser les opérateurs de formatage de chaînes.

Par exemple :

```
In [15]: import math
In [17]: pi = math.pi
In [18]: print(str(pi))
3.141592653589793
```

```
In [19]: s = '%1.4f' % (pi)
In [20]: print(s)
3.1416
```

Dans cet exemple, la valeur de `pi` est chargée dans la variable `pi`. Si nous procédons à un affichage brut, il y a 14 décimales par défaut. Pour avoir moins de décimales, afin de ne pas surcharger l'affichage par exemple, il est nécessaire de passer par l'opérateur de formatage `%f`.

Examinons la ligne suivante :

```
In [19]: s = '%1.4f' % (pi)
```

Ici, nous créons une chaîne de caractères nommée `s` qui est composée d'un flottant, car nous utilisons l'opérateur de formatage `%f`. De plus, `1.4f` signifie que le flottant inséré dans la chaîne `s` est composé d'une valeur pour la partie entière et de quatre valeurs pour la partie décimale.

`%f` n'est pas le seul opérateur de formatage, `%d` peut être utilisé pour insérer des entiers, et `%s` des chaînes de caractères. Les opérateurs peuvent donc être employés pour construire des chaînes complètes sans passer par l'opérateur de concaténation. L'exemple ci-après illustre l'usage de ces opérateurs :

```
In [24]: s = '%s %1.4f %s %d' % ('la partie décimale de pi : ', pi, ' est : ', 3)

In [25]: print(s)
la partie décimale de pi : 3.1416 est : 3
```

Il est également possible de convertir un texte en valeur numérique à l'aide des fonctions `float()` et `int()`. Dans ce cas, le développeur doit avoir la pleine connaissance de ce qu'il réalise. Ces fonctions n'opèrent que des conversions et retournent systématiquement une valeur en sortie. Il est du devoir du développeur de s'assurer que les entrées à convertir sont correctes.

Booléens

Comme la plupart des langages, Python possède un type booléen pour coder les deux états logiques qui sont : vrai et faux. Ces deux valeurs sont respectivement codées par les mots-clés `True` et `False`.

Une illustration de l'affectation d'un booléen est donnée ci-après :

```
In [31]: v = True In [32]:
```

```
f = False
```

```
In [33]: print(v, f) True False
```

2. Types de données complexes

Nous entendons par données complexes, non des données complexes au sens mathématique du terme, mais au sens de la structuration informatique. C'est-à-dire des structures permettant de stocker des ensembles de données hétérogènes ou non. Ces structures ont été ajoutées à la librairie de base du langage dans le but de rendre plus simple leur manipulation et d'éviter aux débutants de devoir les réimplémenter (tâche difficile...).

En Python, ces structures de données sont au nombre de trois :

- les list (listes)
- les dict (dictionnaires)
- Les tuples (tuples)

Avant de présenter ces éléments, il est nécessaire d'introduire succinctement la notion d'objet (voir le chapitre Modularité), car elle est nécessaire pour définir les structures de données.

Un objet est un type de données (à l'image des entiers `int`) qui est composé d'un ensemble de données, qu'on appelle attributs membres, et de fonctionnalités, appelées méthodes membres. Les objets informatiques permettent de conceptualiser un problème particulier. Nous n'irons pas ici plus dans la définition, car la notion d'objet et de classe fera l'objet d'un chapitre particulier.

Maintenant cette notion d'objet introduite, les structures complexes peuvent être présentées.

Listes

Définition

Une liste (`list`), en référence aux listes chaînées, n'est ni plus ni moins qu'un tableau d'objets. Comme tout tableau, la liste est indicée. L'indice de début de la liste est 0.

À la différence des tableaux en C, les listes en Python n'ont pas une taille fixe. Il est donc possible à tout moment d'ajouter ou de retirer des éléments dans la liste.

Création d'une liste

La création d'une liste est très simple, il suffit d'affecter à une variable les valeurs de la liste qui sont précisées entre crochets. Par exemple:

```
In [34]: maliste = [12, 15, 19, 0]
```

```
In [35]: print(maliste)
```

```
[12, 15, 19, 0]
```

Il est très souvent nécessaire dans un algorithme d'avoir à définir une liste vide au début d'une fonction. Ceci s'effectue en Python en ne précisant aucune valeur entre l'opérateur `[]`:

```
In [36]: maliste = [] In [37]:
```

```
print(maliste) []
```

Une particularité de Python est qu'il permet de stocker des objets de type hétérogène dans une liste. Il est possible de mêler chaîne de caractères, entier, flottant ou tout type d'objet défini par l'utilisateur. Il est même possible de stocker des listes dans une liste.

```
In [38]: liste_hétérogène = [1, 1.3, "Bonjour", [1, 2, 3]]
```

```
In [39]: print(liste_hétérogène)
```

```
[1, 1.3, 'Bonjour', [1, 2, 3]]
```

Accès aux données

Comme nous l'avons précisé en définition, une liste est indicée. Cela signifie que nous pouvons accéder au contenu en précisant l'indice de l'élément auquel nous voulons avoir accès.

L'exemple suivant montre comment afficher le contenu du premier élément (indice 0) de la liste, ainsi que le troisième (indice 2).

```
In [40]: print(liste_hétérogène[0], liste_hétérogène[2])  
1 Bonjour
```

Il est possible d'accéder aux données de la liste par plage en précisant l'indice de début et l'indice de fin à l'intérieur de l'opérateur d'accès [] sous la forme `liste[debut:fin]`.

Si l'élément de début ou l'élément de fin n'est pas précisé, le début ou la fin de la liste est pris par défaut. Ce premier exemple permet d'obtenir toute la liste à partir de l'indice 1 :

```
In [42]: print(liste_hétérogène[1:3])  
[1.3, 'Bonjour']
```

Ce second exemple permet de récupérer toutes les valeurs jusqu'au troisième élément.

```
In [44]: print(liste_hétérogène[:3])  
[1, 1.3, 'Bonjour']
```

L'indice peut être également utilisé pour supprimer un élément de la liste à l'aide de la fonction `del`.

```
In [46]: print(liste_hétérogène[:3])  
[1, 1.3, 'Bonjour']
```

```
In [47]: del liste_hétérogène[2] In [48]:
```

```
print(liste_hétérogène)  
[1, 1.3, [1, 2, 3]]
```

Comme pour l'accès classique, il est possible de préciser une plage pour supprimer un ensemble de données de la liste.

```
In [48]: print(liste_hétérogène)  
[1, 1.3, [1, 2, 3]]
```

```
In [49]: del liste_hétérogène[1:] In [50]:
```

```
print(liste_hétérogène)  
[1]
```

Fonctions de manipulation d'une liste

Les listes sont des objets, elles ont donc des fonctions associées. Les plus courantes sont :

- `remove()`
- `pop()`
- `append()`
- `insert()`

`extend()`

`sort()`

La fonction `remove()` supprime un élément dont la valeur est égale à celle précisée en paramètre.

```
In [51]: liste_hétérogène = [1, 1.3, "Bonjour", [1, 2, 3]]
```

```
In [52]: liste_hétérogène.remove("Bonjour")
```

```
In [53]: print(liste_hétérogène)
```

```
[1, 1.3, [1, 2, 3]]
```

La fonction `pop()` permet également la suppression d'éléments de la liste. Cette fonction n'a pas d'argument, car elle permet de supprimer systématiquement le dernier élément de la liste. Elle retourne par ailleurs l'élément supprimé.

```
In [54]: liste_hétérogène.pop()
```

```
Out[54]: [1, 2, 3]
```

```
In [55]: print(liste_hétérogène)
```

```
[1, 1.3]
```

La fonction `append()` ajoute à la fin de la liste un élément précisé en argument.

```
In [56]: liste_hétérogène.append("Merci")
```

```
In [57]: print(liste_hétérogène)
```

```
[1, 1.3, 'Merci']
```

La fonction `insert()` insère une valeur à un indice particulier de la liste.

```
In [58]: liste_hétérogène.insert(2, 3.14)
```

```
In [59]: print(liste_hétérogène)
```

```
[1, 1.3, 3.14, 'Merci']
```

La fonction `extend()` fusionne une liste (passée en argument) à la fin de la liste sur laquelle la fonction a été appelée.

```
In [60]: liste_hétérogène.extend(["Décembre", 3, 4])
```

```
In [61]: print(liste_hétérogène)
```

```
[1, 1.3, 3.14, 'Merci', 'Décembre', 3, 4]
```

Enfin, la fonction `sort()` trie la liste. Il est nécessaire d'avoir une liste composée d'objets homogènes.

```
In [66]: l_str = ["Décembdre", "Bonjour", "Wastringue", "Lundi"] In [67]: l_str.sort()
```

```
In [68]: print(l_str)
```

```
['Bonjour', 'Décembdre', 'Lundi', 'Wastringue']
```

Tuples

Les tuples ont un comportement comparable aux listes, à la différence près qu'il n'est pas possible de modifier le contenu d'un tuple une fois créé. Les fonctions de modification de contenu que nous avons exposées ne sont pas utilisables. En cas d'appel d'une de ces fonctions, une exception est levée et apportée à la connaissance du développeur.

L'opérateur de création d'un tuple est `()`.

```
In [66]: l_str = ["Décembre", "Bonjour", "Wastringue", "Lundi"] In [67]: l_str.sort()
```

```
In [68]: print(l_str)
['Bonjour', 'Décembre', 'Lundi', 'Wastringue']
```

Comme les listes, les tuples sont indicés et l'accès au contenu par un indice est fait à l'aide de l'opérateur `[]`.

```
In [71]: print(t_str[2]) Janvier
```

```
In [72]: print(t_str[1])
2
```

L'intérêt des tuples est de permettre explicitement au développeur d'avoir une structure de données dont il est sûr qu'elle n'est pas modifiable. En pratique, les tuples peuvent être utilisés pour modifier des données passées en argument d'une fonction (voir le chapitre Modularité).

Dictionnaires

Un dictionnaire est une structure de données qui stocke n'importe quel type de données et qui, contrairement aux listes, n'est pas indicée à l'aide d'indices, mais à l'aide de clés. Une clé d'accès peut être un entier ou une chaîne de caractères. Pour créer un dictionnaire, il faut utiliser l'opérateur `{}`.

```
In [73]: d={'Nièvre': 58, 'Saone et Loire': 71, 'Cote d or': 21, 'Yonne': 89} In [74]: d
Out[74]: {'Cote d or': 21, 'Nièvre': 58, 'Saone et Loire': 71, 'Yonne': 89}
```

L'accès aux données du dictionnaire se réalise à l'aide de l'opérateur `[]`, mais contrairement à une liste, il ne faut pas préciser l'indice, mais la clé.

```
In [76]: print(d['Cote d or']) 21
```

Il est possible de mettre à jour un élément du dictionnaire à l'aide de sa clé.

```
In [77]: d['Cote d or'] = 12
In [78]: print(d['Cote d or']) 12
```

Le dictionnaire peut donc être modifié après sa création.

Les fonctions membres `values()` et `keys()` listent respectivement les valeurs et les clés contenues dans le dictionnaire.

```
In [80]: d.keys()
Out[80]: dict_keys(['Saone et Loire', 'Yonne', 'Cote d or', 'Nièvre']) In [81]: d.values()
Out[81]: dict_values([71, 89, 12, 58])
```

3. Opérateurs

Comme tous les langages contemporains, Python possède différents types d'opérateurs suivant les données à manipuler. Un opérateur est une fonction spéciale qui réalise une opération entre des opérandes d'un type défini suivant l'opérateur pour fournir au final un résultat. Un opérateur est représenté par un symbole ou mot-clé du langage, on parle aussi d'instruction réservée du langage.

Les différents types d'opérateurs sont détaillés dans la suite de cette section. Dans un but de clarté, les opérateurs sont regroupés par types de données qu'ils permettent de manipuler.

Opérateur d'affectation

L'opérateur d'affectation attribue une valeur à une variable. Cet opérateur est représenté par le symbole `=`.

Dans le cas de Python, ce symbole permet de réaliser, lors de la première utilisation d'une variable, l'allocation de celle-ci. C'est en fonction du type de la donnée à droite de l'opérateur que le type de variable est choisi par l'interpréteur.

D'un point de vue de la syntaxe, il est possible de réaliser plusieurs affectations sur une même ligne de la manière suivante :

```
In [82]: i, j = 12, 500
In [83]: print(i,j)
12 500
```

Même si le langage offre cette possibilité, il est souvent déconseillé de l'utiliser pour le manque de lisibilité qu'elle procure.

Python permet également de réaliser une affectation multiple. La syntaxe est la suivante :

```
In [84]: c = t = s = r = 1000
In [85]: print(c, t, s, r) 1000 1000
1000 1000
```

Comme pour l'affectation de plusieurs variables sur une même ligne, cette opération est déconseillée, car elle rend le code assez illisible. Notez que ceci n'est qu'une recommandation. Un développeur peut toujours passer outre.

Opérateurs arithmétiques

Tous les opérateurs arithmétiques s'appliquent sur des valeurs entières ou flottantes.

Addition

Le symbole `+` est l'opérateur d'addition. Il retourne le résultat de l'addition des deux valeurs placées de chaque côté de l'opérateur.

```
In [88]: a = 1
In [89]: b = 4
In [90]: a + b
Out[90]: 5
```

Pour utiliser le résultat, il est nécessaire de réaliser une affectation à une troisième variable :

```
In [91]: c = a + b
In [92]: c
Out[92]: 5
```

Si l'une des variables de l'opération est un flottant, alors le résultat est un flottant :

```
In [93]: d = 3.14
In [94]: c = a + d In [95]: c
Out[95]: 4.140000000000001
In [96]: type(c) Out[96]:
float
```

Soustraction

La soustraction - fonctionne exactement sur le même principe que l'addition. De la même manière, si un des opérandes est un flottant, le résultat est automatiquement un flottant :

```
In [102]: a = 5.18
In [103]: b = 5
In [104]: c = a - b
In [105]: c
Out[105]: 0.17999999999999972

In [106]: type(c)
Out[106]: float
```

Multiplication

L'opération de multiplication * réalise la multiplication entre deux opérandes, qu'ils soient entiers ou flottants. Si un opérande est un flottant, alors le résultat est un flottant.

Lorsque l'opérateur de multiplication est doublé (**), l'opération réalisée est la mise à la puissance de l'opérande de gauche par le facteur de puissance précisé par l'opérande de droite.

```
In [125]: a=2
In [126]: b=5
In [127]: c = a**b In [128]:
c
Out[128]: 32
```

Lorsque la multiplication est appliquée sur une liste, c'est-à-dire lorsqu'une liste est placée à gauche de l'opérateur et un entier à droite, une nouvelle liste est générée. Cette liste contient n fois la liste initiale, n étant l'entier placé en opérande à droite.

```
In [110]: c = a * 2 In [111]:
c
Out[111]: [1, 2, 3, 1, 2, 3]
```

Division

L'opération de division / réalise la division entre deux opérandes, qu'ils soient entiers ou flottants. Quel que soit le résultat de l'opération, c'est-à-dire qu'il soit entier ou décimal, le type de la variable de sortie est un flottant.

```
In [112]: a = 35
In [113]: b = 5
In [114]: c=a/b
In [115]: type(c)
Out[115]: float
```

Division entière

La division entière est représentée par l'opérateur //. Cette opération retourne seulement la partie entière du résultat de la division euclidienne, quelle que soit la nature des opérandes : entier ou flottant.

```
In [122]: a = 35.8//5.8 In [123]: a
Out[123]: 6.0
```


Opération composée

Comme en C++, il existe des opérateurs arithmétiques composites. Ces opérateurs sont +=, -=, *=, /=.

Le principe de ces opérations est toujours le même. Prenons l'addition affectation : <op1> += <op2> réalise l'addition de <op1> et de <op2> et affecte le résultat à <op1>. Par exemple :

```
In [130]: a = 5
In [131]: b = 7 In [132]:
a += b In [133]: a
Out[133]: 12
```

Le principe est exactement le même pour la soustraction affectation (-=), la multiplication affectation (*=) et la division affectation (/=).

Opérateurs de comparaison

Les opérateurs de comparaison permettent de savoir si les opérandes placés autour de l'opérateur vérifient ou non la condition dudit opérateur.

L'opération de comparaison quelle qu'elle soit retourne **True** (vrai) si la condition est vérifiée, et **False** dans le cas contraire.

Les différents opérateurs de comparaison sont les suivants :

- < (strictement inférieur) teste si l'opérande de gauche est strictement inférieur à l'opérande de droite.
- > (strictement supérieur) teste si l'opérande de gauche est strictement supérieur à l'opérande de droite.
- <= (inférieur ou égal) teste si l'opérande de gauche est inférieur ou égal à l'opérande de droite.
- >= (supérieur ou égal) teste si l'opérande de gauche est supérieur ou égal à l'opérande de droite.
- == (égal) teste si l'opérande de gauche est égal à l'opérande de droite.
- != (différent) teste si l'opérande de gauche est différent de l'opérande de droite.

Opérateurs d'appartenance

L'opérateur d'appartenance **in** permet de savoir si une valeur est incluse dans une liste, un tuple ou un dictionnaire.

Pour les listes et les tuples, le fonctionnement est le même. C'est-à-dire que **True** est retourné si la valeur est incluse dans la liste ou le tuple, sinon **False** est retourné.

```
In [1]: l = [1, 2, 3]
In [2]: t = (4, 5, 6)
In [3]: 1 in l Out[3]:
True
In [4]: 7 in t Out[4]:
False
```

Sur un dictionnaire, le fonctionnement est différent puisque seules les clés sont testées.

```
In [5]: d = {'11': 'Novembre', '12': 'Décembre'} In [6]: '12' in d
Out[6]: True
In [7]: 'Novembre' in d Out[7]:
False
```

Dans le cas des dictionnaires, l'opérateur d'appartenance ne peut donc être utilisé pour tester le contenu de la liste.

L'opérateur d'appartenance peut être utilisé sur des chaînes de caractères pour savoir si un caractère ou une chaîne est inclus au sein de la chaîne testée :

```
In [8]: s = 'bépoèdljzw' In [9]:
'bépo' in s Out[9]: True
```

Opérateurs logiques

Python propose également les opérateurs de logique booléenne. Avant de présenter leur fonctionnement, il est nécessaire de présenter celui des opérateurs de logique de manière indépendante du langage.

Pour ce faire, on utilise généralement des tables de vérité. Une table de vérité présente le résultat en sortie de l'opérateur en fonction de l'état des opérandes placés en entrée.

Pour la négation, la sortie consiste à inverser l'état de l'entrée :

A (entrée)	S (sortie)
1	0
0	1

L'opérateur « ET logique » propose en sortie la valeur 1 (Vrai) quand les opérandes d'entrée valent également 1 (Vrai). Dans tous les autres cas, la sortie prend la valeur 0 (Faux) :

A (entrée)	B (Entrée)	S (sortie)
0	0	0
0	1	0
1	0	0
1	1	1

À l'inverse, l'opérateur « OU logique » prend la valeur 1 (Vrai) quand au moins un des opérandes a la valeur 1 (Vrai). Dans tous les autres cas, la sortie prend la valeur 0 (Faux).

A (entrée)	B (Entrée)	S (sortie)
0	0	0
0	1	1
1	0	1
1	1	1

Le dernier opérateur logique est le « OU exclusif » qui prend en sortie la valeur 1 (Vrai) lorsque l'une ou l'autre des entrées vaut 1 (Vrai). Si les entrées valent simultanément 0 (Faux) ou 1 (Vrai), alors la sortie a la valeur 0 (Faux).

Notez que cet opérateur est également nommé **XOR**.

A (entrée)	B (Entrée)	S (sortie)
0	0	0
0	1	1
1	0	1
1	1	0

Après avoir passé en revue le fonctionnement théorique des opérateurs de logique booléenne, nous allons voir leur fonctionnement en Python.

La négation avec le mot-clé réservé `not` fonctionne uniquement sur des valeurs de type booléen.

```
In [19]: a = True
In [20]: not a
Out[20]: False
```

Le « ET logique » utilise le mot-clé réservé `and`. Cet opérateur fonctionne sur des données de type booléen ou flottant. Si le type est booléen, alors la table de vérité présentée précédemment est directement appliquée :

```
In [23]: a = True In [24]: b
= False In [25]: c = True In
[26]: a and b Out[26]:
False
```

```
In [27]: a and c Out[27]:
True
```

Si le type est entier, alors l'opération booléenne « ET logique » est appliquée bit à bit pour chaque bit de la donnée :

```
In [33]: a = 0b0101 In [34]:
b = 0b0011

In [35]: bin(a and b) Out[35]:
'0b11'
```

➔ La fonction **bin()** affiche la valeur passée en argument au format binaire.

Le « OU logique » dont l'instruction Python est **Or** a exactement le même fonctionnement que le « ET logique », mais c'est la table de vérité du « OU logique » qui s'applique. L'exemple ci-après présente le fonctionnement sur des données de type booléen.

```
In [36]: a = 0b0101 In [37]:
b = 0b0011

In [38]: bin(a or b)
Out[38]: '0b101'
```

Sur des entiers, l'opération est réalisée bit à bit :

```
In [36]: a = 0b0101 In [37]:
b = 0b0011

In [38]: bin(a or b)
Out[38]: '0b101'
```

Le « OU exclusif » qui correspond à l'opérateur **^** fonctionne exactement comme les deux derniers opérateurs présentés. Les exemples ci-après illustrent le fonctionnement sur des booléens et sur des entiers.

```
In [61]: a = True
```

```
In [62]: b = True In [63]:
a ^ b Out[63]: False
```

```
In [64]: a = 0b0101 In [65]: b
= 0b0011

In [66]: bin(a ^ b)
Out[66]: '0b110'
```

4. Structures conditionnelles

Les conditions sont un concept très utilisé en informatique. L'idée est de n'exécuter certaines instructions de code que lorsque certaines conditions sont remplies.

En Python, il n'existe qu'une seule structure conditionnelle : la structure `if`(Si). Dans sa forme la plus simple, la syntaxe est la suivante :

```
if <condition évaluée> :  
    instruction 1  
    instruction 2
```

Si la condition évaluée est réalisée, alors les instructions 1 et 2 sont exécutées. Le plus souvent, les conditions évaluées le sont à l'aide d'opérateurs conditionnels.

Le code ci-après illustre le fonctionnement du `if` dans sa version la plus simple.

```
a = True  
if(a is True):  
    print("a is True")
```

In [1]: %run if_simple.py a is True

Une version plus évoluée est le `if`, ..., `else`. La syntaxe est la suivante :

```
if <condition évaluée> :  
    instruction 1  
    instruction 2  
else :  
    instruction3
```

Dans cette forme, si la condition évaluée dans le `if` n'est pas vérifiée, alors les instructions après le mot-clé `Else` sont exécutées. L'exemple suivant illustre le procédé :

```
a = False  
  
if(a is True):  
    print("a is True")  
else:  
    print("a is False")
```

Ce qui produit le résultat :

In [4]: %run if_else.py a is False

Enfin, le mot-clé `elif` permet d'évaluer plusieurs instructions complémentaires pour une structure conditionnelle de type `if`.

```
if <condition évaluée> :  
    instruction 1  
elif <condition 2 évaluée> : instruction  
    2  
...  
elif <condition n évaluée> : instruction  
    n  
else :  
    instruction par défaut
```

L'exemple ci-après illustre la structure conditionnelle avec tous les éléments possibles :

```
a = 2

if(a == 1):
    print("a = 1")
elif(a == 2):
    print("a = 2")
else:
    print("a ne vaut ni 1 ni 2")
```

Le résultat de l'exécution de ce script est :

```
In [5]: %run if_elif_else.py a = 2
```

Seule l'instruction if est obligatoire, les instructions else et elif ne sont que facultatives.

5. Boucles

Les boucles sont un concept informatique qui permet de répéter un certain nombre de fois une ou des instructions.

Le nombre d'itérations de la boucle (ou nombre de répétitions) peut être soumis à une condition. Cette condition peut être un nombre de répétitions défini, par exemple : répéter huit fois ; ou ce peut être quand un état logique particulier est atteint, par exemple : tant qu'une valeur booléenne vaut `True`.

L'avantage est de réduire l'effort de syntaxe : on évite ainsi d'écrire `n` lignes de code, il suffit d'itérer la boucle `n` fois.

Il existe en Python deux boucles : la boucle `while` et la boucle `for`.

Boucle `while`

Cette boucle permet de répéter des instructions tant qu'une condition est vraie (`True`). La syntaxe est la suivante :

```
while <condition> :  
    <instruction 1>  
    <instruction 2>  
    <instruction 3>  
    ...  
    <instruction n>
```

Cette boucle est très simple à comprendre, puisque tant que la condition `<condition>` vaut `True`, alors les instructions `<instruction 1>` à `<instruction n>` sont exécutées.

Notez que l'indentation doit être absolument respectée pour que le comportement obtenu soit celui escompté. Le script suivant affiche la valeur d'un compteur tant que la valeur du compteur est strictement inférieure à 10.

```
cpt = 0  
  
while cpt < 10:  
    print("Valeur du compteur : ", cpt)  
    cpt += 1
```

```
In [1]: %run while.py  
Valeur du compteur : 0  
Valeur du compteur : 1  
Valeur du compteur : 2  
Valeur du compteur : 3  
Valeur du compteur : 4  
Valeur du compteur : 5  
Valeur du compteur : 6  
Valeur du compteur : 7  
Valeur du compteur : 8  
Valeur du compteur : 9  
.
```

Une erreur classique de programmation est d'avoir une « boucle infinie ». Cette expression signifie que vous avez placé une condition dans le `while` qui est toujours valide. Par conséquent, votre boucle et par le fait vos instructions seront répétées indéfiniment. Dans ce cas de figure, vous pouvez stopper l'exécution de votre script à l'aide de la combinaison de touches `[Ctrl]+c`.

Boucle for

La boucle `for` est la seconde boucle disponible en langage Python. Cette instruction en C++ et Java diffère fortement, au niveau de son comportement, de l'instruction `for` de Python (ce qui peut être surprenant ou rebutant pour les développeurs C++ ou Java).

En effet, en C, la syntaxe est : `for (début;fin;incrément)`. Par exemple :

```
for(int i=0;i<10;i++){  
    cout<<i ;  
}
```

Cela a pour effet de répéter les instructions de la boucle 10 fois. En Python, l'idée est tout autre et rien que la syntaxe le reflète :

```
for <élément> in séquence> :  
    <instruction 1>  
    ...  
    <instruction n>
```

La boucle fonctionne de la manière suivante : les instructions sont répétées autant de fois qu'il y a d'éléments dans la séquence `<séquence>`. Pour chaque itération, l'élément `<élément>` prend la valeur de l'élément courant correspondant à la position de la valeur de l'itération dans la séquence.

Notez que l'élément `<élément>` est utilisable dans toute la portée de la boucle.

`<Séquence>` peut être tout type d'objet itérable : liste, tuple, dictionnaire, chaîne de caractères ou tout autre objet possédant un itérateur.

Le script suivant affiche toutes les valeurs d'une liste et un message particulier si un élément de cette liste vaut 12 :

```
l = [1, 2, 3, 4, 5, 12]  
  
for elt in l:  
    print(elt)  
  
    if elt==12:  
        print("Valeur 12 détectée dans la séquence")
```

L'exécution de ce script produit la sortie suivante :

```
In [3]: %run for_liste.py 1  
2  
3  
4  
5  
12  
Valeur 12 détectée dans la séquence
```

Ce fonctionnement possède des avantages, mais les utilisateurs de C par exemple seront très vite bloqués par le fait que la boucle `for` de Python ne manipule pas directement un indice, qui est généralement très pratique pour accéder à des valeurs d'un tableau.

Il existe plusieurs solutions pour pallier cette limite. La première est d'utiliser le mot-clé `enumerate` et de compléter la syntaxe de la boucle `for` de la manière suivante :

```
for <indice>, <élément> in enumerate(sequence) :
```

Avec cette syntaxe, `indice` prend la valeur correspondant à l'indice de l'élément accédé par la boucle à chaque itération. À chaque itération, l'élément de la liste est également retourné.

Le script ci-après illustre ce procédé :

```
l = [1, 2, 3, 4, 5]

for ind, elt in enumerate(l):
    print("Valeur indice : ", ind, " Valeur élément : ", elt)
```

Il produit le résultat suivant :

```
In [5]: %run for_enumerate.py
Valeur indice :      0    Valeur élément :      1
Valeur indice :      1    Valeur élément :      2
Valeur indice :      2    Valeur élément :      3
Valeur indice :      3    Valeur élément :      4
Valeur indice :      4    Valeur élément :      5
```

Ce procédé est intéressant si vous avez besoin de l'indice et de la valeur. Il existe des cas où seul l'indice courant est intéressant. Dans ce cas, il est nécessaire d'utiliser la fonction `range()`. La fonction `range()` attend comme paramètre un entier qui permet de générer une séquence de la taille précisée en paramètre. La séquence générée comporte des valeurs entre 0 et taille-1. Par exemple, `range(3)` génère la séquence : 0, 1, 2.

Au final, la syntaxe pour la boucle `for` devient :

```
for <indice> in range(len(sequence)) :
```

L'exemple ci-après illustre cette troisième syntaxe :

```
l = [1, 2, 3, 4, 5]

for ind in range(len(l)):
    print("Valeur indice : ", ind)
```

Ce qui produit le résultat :

```
In [10]: %run for_range.py
Valeur indice :      0
Valeur indice :      1
Valeur indice :      2
Valeur indice :      3
Valeur indice :      4
```

Les différentes versions montrées ici devraient permettre aux développeurs C d'appréhender tous les cas de figure.

Pour clore cette section, les instructions **break** et **continue** doivent être présentées. L'instruction **break** permet de sortir de la boucle en cours.

À titre d'exemple :

```
l = [1, 2, 3, 4, 5]

for elt in l:
    if elt == 3:
        break
    print("Valeur élément liste : ", elt)
```

Dans ce code, si la valeur contenue dans la liste est égale à 3, nous sortons de la boucle. Le résultat attendu est donc un affichage des valeurs 1 et 2 de la liste, puis une sortie de la boucle. C'est ce qui peut être vérifié dans la sortie générée par le script :

```
In [11]: %run for_break.py
Valeur
élément liste : 1 Valeur élément
liste : 2
```


6. Fonctions

En informatique, le développeur débutant se retrouve parfois à écrire un code proche d'un code qu'il a déjà écrit précédemment. Dans ce cas de figure, plutôt que d'écrire une seconde fois une portion de code identique, on doit alors écrire une fonction pour encapsuler cette portion de code afin de le capitaliser.

Cette encapsulation est en fait une stratégie de modularité de code. En effet, en créant une fonction, le développeur crée un module qu'il pourra utiliser facilement et rapidement dans son application en cours de développement, mais également potentiellement dans de futurs projets.

Une fonction permet de regrouper un ensemble d'instructions afin de les exécuter au moment de l'appel de la fonction à l'endroit voulu dans le code source.

Une fonction est généralement créée pour réaliser une fonctionnalité particulière, par exemple :

- tri d'une liste
- recherche d'un maximum
- extraction d'une sous-chaîne au sein d'une chaîne de caractères

La réalisation de la fonctionnalité contenue au sein d'une fonction peut nécessiter d'avoir des informations a priori pour réaliser la tâche. Ces informations sont données à l'aide des paramètres de la fonction.

Une fonction peut calculer un résultat à la suite de son exécution et le développeur peut vouloir récupérer ce résultat. Ceci se réalise à l'aide des valeurs retournées par la fonction.

Cas classique

En Python, la syntaxe d'une fonction est la suivante :

```
def ma_fonction(<param1>, <param2>, ..., <paramn>) :  
    <instruction 1>  
    <instruction 2>  
    ...  
    <instruction n>  
return <val1>, <val2>, ... , <val n>
```

Il apparaît, à partir de cette définition générique, que nous pouvons passer autant de paramètres que nécessaire et que nous pouvons retourner autant de valeurs que nécessaire. Notez qu'une fonction doit être définie avant d'être appelée. Nous devons donc placer le code de définition de la fonction avant l'appel.

La fonction suivante retourne le résultat de la somme de deux valeurs passées en paramètre.

```
# Définition de la fonction  
def somme(a, b):  
    c = a + b  
    return c  
  
val_a = 12  
val_b = 18  
# appel de la fonction  
val_somme = somme(val_a, val_b)  
print("Résultat de la somme : ", val_somme)
```

La valeur retournée par la fonction `somme()` est bien 30 :

```
In [5]: %run fonction_simple.py  
Résultat de la somme : 30
```

Pas de valeurs retournées

Une fonction ne retourne pas nécessairement un résultat. Par exemple, si la fonction à réaliser doit seulement afficher la chaîne « Bonjour », elle n'a rien à retourner. Dans ce cas, il suffit de supprimer la ligne `return`.

```
# Définition de la fonction aff
def aff():
    print("Bonjour")

#appel de la fonction aff
aff()
```

Résultat de l'exécution :

```
In [6]: %run fonction_return_none.py
Bonjour
```

→ Lorsqu'une fonction ne retourne pas de valeur, on parle également de procédure.

Plusieurs valeurs retournées

Nous avons vu dans la définition générique d'une fonction en Python qu'il existe la possibilité de retourner plusieurs valeurs. Nous allons illustrer le mécanisme en créant une fonction qui retourne la somme et le produit de deux valeurs passées en paramètre.

```
# Déclaration de la fonction
def somme_produit(a, b):
    p = a * b
    s = a + b

    return p, s

val_a = 2
val_b = 5
produit, somme = somme_produit(val_a, val_b)
print("Valeur produit : ", produit, " Valeur somme : ", somme)
```

Pour retourner ces deux valeurs, il faut placer deux valeurs au niveau de l'instruction `return`. Pour récupérer ces valeurs, il suffit de préciser deux valeurs séparées par une virgule lors de l'appel.

L'exemple précédent produit le résultat qui suit :

```
In [7]: %run fonction_return_multiple.py
Valeur produit : 10 Valeur somme : 7
```

Les valeurs sont récupérées dans le même ordre qu'elles ont été écrites après l'instruction `return`. C'est donc à l'utilisateur de s'assurer qu'il récupère les valeurs dans le bon ordre.

Il se peut que la fonction soit définie avec plusieurs valeurs retournées, mais que pour l'un de ses usages, seulement une partie des résultats retournés nous intéresse. Pour ce cas de figure, Python offre la possibilité, lors de l'appel, de placer le caractère `_` à la place d'une valeur retournée par la fonction. Modifions l'exemple pour ne récupérer que la valeur de la somme et l'afficher.

```
# Déclaration de la fonction
def somme_produit(a, b):
    p = a * b
    s = a + b

    return p, s

val_a = 2
val_b = 5
_, somme = somme_produit(val_a, val_b)
print(" Valeur somme : ", somme)
```

Ce qui produit le résultat :

```
In [8]: %run fonction_return_partiel.py
Valeur somme : 7
```

Le gros intérêt de ce fonctionnement est de ne pas écrire une fonction supplémentaire qui ne retournerait que la somme.

Ce mécanisme illustré pour deux valeurs est généralisable pour n valeurs.

Paramètres par défaut

Un cas de figure très courant en développement est d'avoir besoin de tous les arguments passés en paramètre ou d'avoir besoin de paramètre supplémentaire par rapport à la version implémentée de la fonction.

Ces situations se gèrent dans certains langages grâce à la surcharge de fonction. La surcharge consiste à avoir plusieurs versions de la fonction, c'est-à-dire des fonctions portant le même nom, mais avec des types de paramètres et des nombres de paramètres différents.

Malheureusement en Python, la surcharge n'existe pas. Pour pallier cela, on utilise les arguments par défaut. Ce qui permet, dans certains cas, d'éviter de passer la valeur de ces paramètres en argument.

Pour réaliser cela, il suffit de préciser au moment de la déclaration, pour chaque argument, la valeur par défaut précédée de l'opérateur `=`, comme ceci :

```
def ma_fonction(<arg1 = «valeur_par_defaut_1>, <arg2 =
«valeur_par_defaut_2>, ... , <argn =
«valeur_par_defaut_n>)

    <instruction1>
    <instruction2>
    ...
    <instructionn>

    return <valeur1>, <valeur2>, ..., <valeurn>
```

Illustrons ce comportement en modifiant l'exemple de la section précédente. Une valeur supplémentaire `c` est ajoutée en argument. La valeur de cette variable est utilisée lors du calcul de la somme $a+b+c$ et du produit $a*b*c$. Enfin, des valeurs par défaut sont précisées pour `b` (4) et `c` (10). Lors de l'appel, seul `a` est précisé, ce qui signifie que les valeurs par défaut de `b` et `c` sont utilisées.

```
def somme(a, b=4, c=10):
    produit = a * b * c
    somme = a + b + c

    return produit, somme

val_a = 7
p, s = somme(val_a)
print("Valeur produit : ", p, " Valeur somme : ", s)
```

Le résultat est le suivant :

```
%run fonction_val_defaut.py
Valeur produit : 280 Valeur somme : 21
```

Si nous voulons utiliser le paramètre par défaut pour `b` et préciser une valeur pour `c` (par exemple 3), nous ne pouvons pas écrire `p, s = somme(val_a, 3)` lors de l'appel de la fonction. En effet, dans ce cas, l'interpréteur considère que la valeur 3 est celle de `b` et que `c` utilise la valeur par défaut. Pour réaliser le comportement voulu, il faut explicitement nommer le paramètre suivi de l'affectation de sa valeur : `p, s = somme(val_a, c=3)`.

```
def somme(a, b=4, c=10):
    produit = a * b * c
    somme = a + b + c

    return produit, somme

val_a = 7
p, s = somme(val_a)
print("Valeur produit : ", p, " Valeur somme : ", s)

p, s = somme(val_a, c=3)
print("Valeur produit : ", p, " Valeur somme : ", s)
```

De cette manière, la valeur de `c` est bien prise en compte :

```
In [14]: %run fonction_val_defaut_2.py
Valeur produit : 280 Valeur somme : 21
Valeur produit : 84 Valeur somme : 14
```

Rendre le code modulaire

La manière dont nous avons procédé pour illustrer le mécanisme des fonctions intègre la fonction au sein même du code qui l'appelle. Cette manière de faire n'est pas la meilleure. En effet, les fonctions définies ainsi ne peuvent être utilisées que par le programme au sein du même fichier, elles ne peuvent donc pas être capitalisées en les intégrant dans d'autres projets.

Une solution est de stocker ces fonctions dans un fichier .py indépendant et d'importer les fonctions dont nous avons besoin dans le script qui appelle lesdites fonctions. L'importation se fait à l'aide de l'instruction suivante :

```
from <nom_fichier> import <nom_fonction>
```

Notez que <nom_fichier> ne comporte pas l'extension .py. L'exemple qui suit illustre ce mécanisme de modularité.

Dans un premier temps, un fichier mes_fonctions.py est créé :

```
def somme(a, b=4, c=10):  
    produit = a * b * c  
    somme = a + b + c  
  
    return produit, somme
```

Dans un second temps, il faut créer le programme principal qui appelle la fonction après avoir importé cette dernière. Ce fichier se nomme main.py :

```
from mes_fonctions import somme  
  
val_a = 7  
p, s = somme(val_a)  
print("Valeur produit : ", p, " Valeur somme : ", s)  
  
p, s = somme(val_a, c=3)  
print("Valeur produit : ", p, " Valeur somme : ", s)
```

Et cela produit toujours le même résultat :

```
In [16]: %run main.py  
Valeur produit : 280 Valeur somme : 21  
Valeur produit : 84 Valeur somme : 14
```

7. Programmation orientée objet

La programmation orientée objet (POO) est un concept informatique. Les puristes emploient le terme de paradigme, qui consiste à représenter les informations d'un problème sous la forme d'objets.

Un objet peut être tout élément du problème qui peut être réduit à un ensemble de données (on parle d'attributs) et de fonctionnalités (qu'on nomme méthodes).

Vus sous cet angle, tous les éléments d'un problème quelconque peuvent être vus comme des objets. Afin d'illustrer ce que peut être un objet, citons quelques exemples :

- un point (au sens mathématique)
- un fichier
- une voiture
- un flux de données
- un service web

On voit donc qu'un objet peut servir à représenter un objet physique, un élément informatique ou même un concept.

Un des intérêts des objets est que, de par leur modularité, il est possible de les réutiliser d'un projet informatique à l'autre.

Comme nous l'avons dit précédemment, un objet contient des données appelées attributs. Ces attributs peuvent être de type simple comme des entiers, des flottants ou des chaînes de caractères.

Un objet est créé à partir d'un moule qu'on nomme classe. Ce moule, cette classe contient un ensemble de fonctionnalités qui permettent de traduire le comportement de l'objet, ce sont les méthodes. Une classe possède également un ensemble de données qu'on nomme attribut. La création d'un objet à partir de son moule (la classe) est appelé l'instanciation. Un objet issu de la procédure d'instanciation sera nommé instance.

D'autres concepts sont généralement couverts par la programmation orientée objet : le typage, le polymorphisme, la surcharge d'opérateurs et de fonctions.

Ces considérations théoriques sont la base de travail des concepteurs des différents langages qui font le choix d'une implémentation particulière du concept POO en fonction de l'usage particulier du langage. En effet, l'usage de PHP n'est pas le même que celui de C++, ce qui impacte les choix des responsables de ces langages en termes de stratégie et de priorité d'implémentation du concept POO.

Notez que certains choix d'implémentation du concept POO sont moins rationnels et tiennent pour certains de la conviction de l'équipe en charge de l'évolution du langage.

Python possède sa propre implémentation de la POO et la suite de ce chapitre est consacrée à la présentation des possibilités de Python en termes de POO.

- **Le but de cet mini-guide n'étant pas de produire un traité d'informatique, nous limiterons notre présentation aux éléments les plus utiles dans le cadre de l'utilisation de Python pour le Raspberry Pi.**

Programmation objet et différences principales avec d'autres langages

Python est un langage tout objet. Vous avez déjà manipulé sans le savoir, dans les chapitres précédents, des objets !

En effet, les éléments les plus basiques comme les `float` et les `int` sont des objets qui possèdent des attributs et des méthodes. C'est également le cas des `list`, dont vous avez manipulé quelques méthodes, par exemple `append()`.

Python possède sa propre implémentation du paradigme de POO. Elle est, en comparaison à celle de C++, beaucoup plus accessible au débutant. En termes de différences :

- C++ permet la surcharge de méthodes, alors la surcharge n'existe pas en Python pour les fonctions. C'est également le cas pour les méthodes des classes.
- C++ permet de réduire plus ou moins l'accès aux attributs et aux méthodes à l'aide de mots-clés : `public`, `private` et `protected`. Python n'offre pas explicitement cette possibilité.

Outre ces différences principales, Python permet l'usage des propriétés, décorateurs, qui n'existent pas en C++. Enfin, pour conclure cette section, donnons la définition générique en langage Python :

```
class <nom_de_la_classe> (object):
    # définition du constructeur
    def __init__(self, arg1, arg2, ..., argn) :
        <attribut 1> = val1
        <attribut 2> = val2
        ...
        <attribut n>

    def <nom_methode1> (self, arg1, arg2, ..., argn):
    def <nom_methode2> (self, arg1, arg2, ..., argn):
    ...
    def <nom_methode3> :
```

Nous reviendrons sur tous les éléments de cette définition par la suite, mais notez que le mot-clé `class` permet de savoir qu'il s'agit de la définition d'une classe.

- **Selon la recommandation PEP 8 le nom de la classe doit être écrit sans espace entre les mots et avec une majuscule au début des mots constituant le nom de la classe, par exemple `CafetiereExpresso` ou `Point2D`...**

Constructeur et attributs

Le constructeur est la première fonction qui est appelée lors de l'instanciation d'une classe. L'instanciation consiste à la création d'un objet à partir d'un modèle, ici une classe.

Dans la définition générique de la classe, le constructeur correspond à la méthode spéciale : `__init__` (`self, ...`) .

En Python, cette méthode contient toutes les définitions et les initialisations d'attributs. Pour illustrer le propos, nous allons modéliser un téléphone. Commençons par ses attributs :

- marque
- modele
- couleur
- annee

L'exemple qui suit correspond à la déclaration des attributs en question. `marque`, `modele` et `couleur` sont des chaînes de caractères initialisées à la valeur vide " ". Quant à `annee`, c'est un entier initialisé à 0.

```
class Telephone(object):  
  
    def __init__(self):  
        self.marque = ""  
        self.modele = ""  
        self.couleur = ""  
        self.annee = 0
```

Il est important que le premier argument du constructeur soit toujours `self`, ce qui correspond à l'instance en cours de la classe.

Par ailleurs, les attributs de la classe sont toujours précédés de `self`. Ce qui signifie que nous manipulons la classe instanciée.

- **Dans la plupart des langages, l'instance de la classe en cours n'utilise pas le mot-clé réservé `self`, mais `this`. De plus, dans de nombreux langages, on ne manipule pas le mot-clé `this` pour manipuler les attributs de la classe. C'est par exemple le cas en C++. De ce point de vue, la syntaxe de Python est un peu verbeuse et est assez critiquée par les développeurs, car elle alourdit le code. C'est pour cette raison que certains développeurs utilisent `s` en lieu et place de `self`. Nous vous encourageons à conserver la convention `self`, qui fait plutôt consensus dans la communauté des développeurs de packages Python open source.**

Nous avons défini la classe, nous pouvons donc maintenant l'utiliser en créant une instance de cette classe. Nous allons également modifier l'attribut `marque` et afficher ensuite toutes les valeurs des attributs de la classe.

```

class Telephone(object):

    def __init__(self):
        self.marque = ""
        self.modele = ""
        self.couleur = ""
        self.annee = 0

# Instanciation de la classe
t = Telephone()
# modification de l'attribut marque
t.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("Valeur pour l'attribut marque : ", t.marque)
print("Valeur pour l'attribut modele : ", t.modele)
print("Valeur pour l'attribut couleur : ", t.couleur)
print("Valeur pour l'attribut annee : ", t.annee)

```

Le résultat produit est bien celui attendu, c'est-à-dire que seul l'attribut `marque` est modifié, les autres valeurs étant celles par défaut issues de la construction de l'objet :

```

%run constructeur1.py
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele :
Valeur pour l'attribut couleur :
Valeur pour l'attribut annee : 0

```

Cette manière de faire est intéressante, mais l'utilisateur peut avoir besoin de ne passer les valeurs d'initialisation qu'au moment de l'instanciation de l'objet. Pour cela, il suffit d'ajouter des paramètres au constructeur, puis de passer des valeurs, comme on le ferait avec une fonction classique.

Modifions les attributs `modele` et `couleur` lors de l'instanciation de l'objet :

```

class Telephone(object):

    def __init__(self, val_modele, val_couleur):
        self.marque = ""
        self.modele = val_modele
        self.couleur = val_couleur
        self.annee = 0

# Instanciation de la classe
t = Telephone("3310", "jaune")
# modification de l'attribut marque
t.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("Valeur pour l'attribut marque : ", t.marque)
print("Valeur pour l'attribut modele : ", t.modele)
print("Valeur pour l'attribut couleur : ", t.couleur)
print("Valeur pour l'attribut annee : ", t.annee)

```

Maintenant, les attributs `modele` et `couleur` de la classe sont bien initialisés par les valeurs passées en paramètre :

```
%run constructeur1.py
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 3310
Valeur pour l'attribut couleur : jaune
Valeur pour l'attribut annee : 0
```

Dans certains cas, l'utilisateur peut avoir besoin de préciser une valeur lors de l'instanciation, et dans d'autres cas, non. Dans cette situation, il est nécessaire d'utiliser les paramètres par défaut, comme nous l'avons fait pour les fonctions.

L'exemple ci-après montre la modification de la classe pour utiliser une valeur par défaut pour l'attribut `annee` quand aucune valeur n'est précisée par l'utilisateur.

Il montre également les deux façons d'instancier une classe en précisant la valeur, en utilisant la valeur par défaut.

```
class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):
        self.marque = ""
        self.modele = val_modele
        self.couleur = val_couleur
        self.annee = val_annee

print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 1997)
# modification de l'attribut marque
t1.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("Valeur pour l'attribut marque : ", t1.marque)
print("Valeur pour l'attribut modele : ", t1.modele)
print("Valeur pour l'attribut couleur : ", t1.couleur)
print("Valeur pour l'attribut annee : ", t1.annee)

print("\nInstance t2 avec annee par défaut")
# Instanciation de la classe en utilisant la valeur par défaut pour annee
t2 = Telephone("5510", "bleu")
# modification de l'attribut marque
t2.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("Valeur pour l'attribut marque : ", t2.marque)
print("Valeur pour l'attribut modele : ", t2.modele)
print("Valeur pour l'attribut couleur : ", t2.couleur)
print("Valeur pour l'attribut annee : ", t2.annee)
```

Le résultat obtenu est le suivant :

```
%run constructeur1.py
Instance t1 avec tous les arguments
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 3310
Valeur pour l'attribut couleur : jaune
Valeur pour l'attribut annee : 1997
```

```
Instance t2 avec annee par défaut
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 5510
Valeur pour l'attribut couleur : bleu
Valeur pour l'attribut annee : 2002
```

Il apparaît que la seconde instance a bien utilisé la valeur par défaut pour l'attribut `annee` et que la première utilise bien la valeur passée en paramètre, soit 2002.

Méthodes membres

Une classe est composée d'attributs, mais également de méthodes. Le terme méthode désigne les fonctions manipulant les données de la classe et pouvant être accédées depuis une instance de l'objet.

La définition d'une méthode est semblable à la définition d'une fonction classique, à la différence près qu'elle est faite au sein de la classe et qu'une méthode comporte toujours comme premier paramètre `self`.

Afin d'illustrer ceci, ajoutons à la classe `Telephone` une méthode pour afficher les valeurs des quatre attributs. Cette méthode se nomme `aff_attribut`.

```
class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):
        self.marque = ""
        self.modele = val_modele
        self.couleur = val_couleur
        self.annee = val_annee

    def aff_attribut(self):
        print("Valeur pour l'attribut marque : ", self.marque)
        print("Valeur pour l'attribut modele : ", self.modele)
        print("Valeur pour l'attribut couleur : ", self.couleur)
        print("Valeur pour l'attribut annee : ", self.annee)
```

Cette méthode réalise l'affichage successif des quatre attributs de la classe. Comme pour le constructeur, pour accéder aux attributs, il faut passer par `self.<nom_attribut>`. Si ce n'est pas le cas, l'interpréteur Python indique que la variable à laquelle nous voulons accéder n'est pas membre de la classe, comme dans le message suivant : « `NameError: name 'marque' is not defined` » .

Grâce à cette fonction, nous pouvons réduire la taille du code qui manipule les instances de la classe `Telephone`.

```

class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):
        self.marque = ""
        self.modele = val_modele
        self.couleur = val_couleur
        self.annee = val_annee

    def aff_attribut(self):
        print("Valeur pour l'attribut marque : ", self.marque)
        print("Valeur pour l'attribut modele : ", self.modele)
        print("Valeur pour l'attribut couleur : ", self.couleur)
        print("Valeur pour l'attribut annee : ", self.annee)

print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 1997)
# modification de l'attribut marque
t1.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
t1.aff_attribut()

print("\nInstance t2 avec annee par défaut")
# Instanciation de la classe en utilisant la valeur par défaut pour annee
t2 = Telephone("5510", "bleu")
# modification de l'attribut marque
t2.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
t2.aff_attribut()

```

L'intérêt de ce genre de méthode est de réduire la taille du code quand nous utilisons les objets. Ici, les huit lignes nécessaires à l'affichage sont remplacées par deux occurrences de la ligne :

```
t2.aff_attribut()
```

Le résultat produit est, lui, toujours le même :

```

In [2]: %run methode1.py
Instance t1 avec tous les arguments
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 3310
Valeur pour l'attribut couleur : jaune
Valeur pour l'attribut annee : 1997

Instance t2 avec annee par défaut
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 5510
Valeur pour l'attribut couleur : bleu
Valeur pour l'attribut annee : 2002

```

Les méthodes peuvent se comporter exactement comme les fonctions. C'est-à-dire que des valeurs ou des objets peuvent être passés en paramètre. Ces paramètres peuvent avoir des valeurs par défaut.

De la même manière, une méthode peut retourner une ou plusieurs valeurs. Comme pour les fonctions, il est possible de ne récupérer qu'une partie des paramètres.

En guise d'exemple, ajoutons une méthode qui calcule la cote d'un téléphone. Pour cela, il est nécessaire de passer la valeur de l'année en cours et un booléen qui vaut Vrai (`True`) si le téléphone fait partie d'une série spéciale. Dans le cas contraire, ce booléen vaut Faux (`False`). Ce booléen prend la valeur `False` par défaut. Un dernier argument est nécessaire, c'est la valeur à l'achat du téléphone. La cote est retournée par la méthode.

```
def calc_cote(self, annee_en_cours, valeur_achat, serie_spec = False):

    bonus = 0
    nb_annee = annee_en_cours - self.annee
    if nb_annee < 10:
        decote = 0.1 * nb_annee
    else:
        decote = 0.9

    if serie_spec == True:
        bonus = 0.1 * valeur_achat

    valeur = valeur_achat - (valeur_achat * decote) + bonus

    return valeur
```

L'appel de cette méthode est effectué de la manière suivante :

```
# calcul des cotes pour t1 et t2
val_t1 = t1.calc_cote(2017, 35)
val_t2 = t2.calc_cote(2017, 35, True)

# affichage de la cote calculée
print("\n")
print("Valeur code t1 : ", val_t1)
print("Valeur code t2 : ", val_t2)
```

D'une manière générique, nous pouvons écrire l'appel d'une méthode de l'instance d'une classe de la manière suivante :

```
<valeur_rtn1>, <valeur_rtn2>, ... <valeur_rtnn> =
<nom_objet>.methode(<arg1>, <arg2>, ... , <argn>)
```

Ce qui permet de transformer le code de la manière suivante :

```
class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):
        self.marque = ""
        self.modele = val_modele
        self.couleur = val_couleur
        self.annee = val_annee

    def aff_attribut(self):
        print("Valeur pour l'attribut marque : ", self.marque)
        print("Valeur pour l'attribut modele : ", self.modele)
        print("Valeur pour l'attribut couleur : ", self.couleur)
        print("Valeur pour l'attribut annee : ", self.annee)

    def calc_cote(self, annee_en_cours, valeur_achat, serie_spec = False):

        bonus = 0
        nb_annee = annee_en_cours - self.annee
        if nb_annee < 10:
            decote = 0.1 * nb_annee
        else:
            decote = 0.9

        if serie_spec == True:
            bonus = 0.1 * valeur_achat

        valeur = valeur_achat - (valeur_achat * decote) + bonus

        return valeur

print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 1997)
```

```

# modification de l'attribut marque
t1.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
t1.aff_attribut()

print("\nInstance t2 avec annee par défaut")
# Instanciation de la classe en utilisant la valeur par défaut pour annee
t2 = Telephone("5510", "bleu")
# modification de l'attribut marque
t2.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
t2.aff_attribut()

# calcul des cotes pour t1 et t2
val_t1 = t1.calc_cote(2017, 35)
val_t2 = t2.calc_cote(2017, 35, True)

# affichage de la cote calculée
print("\n")
print("Valeur code t1 : ", val_t1)
print("Valeur code t2 : ", val_t2)

```

L'exécution de ce script produit le résultat suivant :

```

%run methode1.py
Instance t1 avec tous les arguments
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 3310
Valeur pour l'attribut couleur : jaune
Valeur pour l'attribut annee : 1997

Instance t2 avec annee par défaut
Valeur pour l'attribut marque : Nokia
Valeur pour l'attribut modele : 5510
Valeur pour l'attribut couleur : bleu
Valeur pour l'attribut annee : 2002

Valeur code t1 : 3.5
Valeur code t2 : 7.0

```

Python et l'encapsulation

En informatique, l'encapsulation est un principe qui consiste à protéger ou à cacher des données de certains objets. En d'autres termes, cela consiste à ne pas rendre possible l'accès à certains attributs depuis une instance d'une classe.

Dans beaucoup de langages, cela s'effectue à l'aide du mot-clé `private`, qui permet de protéger l'accès aux données définies sous la portée de ce mot-clé.

Le langage Python ne propose pas de mot-clé `private` pour réaliser l'encapsulation. Les notions d'attribut et de méthode privés n'existent pas directement en Python.

Les développeurs pallient cette limite en appliquant un ensemble de règles de bon usage (voir par exemple : <https://google.github.io/styleguide/pyguide.html>).

En pratique, la syntaxe admise pour représenter ce qui est privé est très simple : tous les attributs ou méthodes qui possèdent un caractère `_` au début de leur nom sont définis comme privés.

Il est donc de la responsabilité du développeur de ne pas utiliser directement un attribut privé lorsqu'il manipule une instance d'un objet.

Cette syntaxe (qui caractérise Python) doit être connue et respectée, notamment dans le cadre d'un projet collaboratif ou de l'utilisation de sources disponibles en open source.

Transformons maintenant la classe `Telephone` pour rendre les variables privées. Les seules transformations sont à réaliser dans le constructeur, ainsi que là où les attributs sont utilisés :

```
class Telephone(objec):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):
        self._marque = ""
        self._modele = val_modele
        self._couleur = val_couleur
        self._annee = val_annee

    def aff_attribut(self):
        print("Valeur pour l'attribut marque : ", self._marque)
        print("Valeur pour l'attribut modele : ", self._modele)
        print("Valeur pour l'attribut couleur : ", self._couleur)
        print("Valeur pour l'attribut annee : ", self._annee)

    def calc_cote(self, annee_en_cours, valeur_achat, serie_spec = False):

        bonus = 0
        nb_annee = annee_en_cours - self._annee
        if nb_annee < 10:
            decote = 0.1 * nb_annee
        else:
            decote = 0.9

        if serie_spec == True:
            bonus = 0.1 * valeur_achat

        valeur = valeur_achat - (valeur_achat * decote) + bonus

        return valeur
```

Maintenant que nous avons adopté cette syntaxe permettant de protéger les données, il est nécessaire d'homogénéiser l'usage de la classe instanciée. En analysant le code, il apparaît que nous rencontrons un problème. En effet, lors de l'instanciation de la classe, nous modifions directement un attribut des deux objets instanciés :

```
t1.marque = "Nokia"
```

Si nous respectons le principe de l'encapsulation, nous ne pouvons plus utiliser l'attribut `_marque`, car nous le considérons comme privé.

Pour accéder à cet attribut, nous devons utiliser des méthodes spéciales nommées accesseurs et mutateurs, qui permettent respectivement d'accéder et de modifier un attribut.

Dans de nombreux langages, il s'agit des fonctions `get()` et `set()`. Elles sont publiques et permettent d'accéder indirectement aux attributs privés.

En Python, le mécanisme est quelque peu différent. Il est possible d'utiliser des accesseurs et des mutateurs, mais sans les appeler explicitement. Ce mécanisme est rendu possible grâce au décorateur. En guise d'exemple, définissons les mutateur et accesseur pour l'attribut `marque`.

```
@property
def marque(self):
    return self._marque

@marque.setter
def marque(self, value):
    self._marque = value
```

La première fonction après le mot-clé `@property` est l'accesseur (`get`) et la seconde après l'expression `@marque.setter` est le mutateur (`set`).

Jusqu'ici, à la syntaxe près, il y a peu de différence avec d'autres langages. Et pourtant, il en existe une majeure !

En effet, c'est l'usage qui est très différent. Python permet de manipuler ce qui semble être un attribut, mais qui est en fait une illusion, car on manipule les accesseurs et mutateurs. L'exemple qui suit permet de manipuler l'attribut `_marque`, considéré comme privé.

```
# modification de l'attribut marque
t1.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("La marque de t1 est : ", t1.marque)
```

La première instruction fait appel au mutateur pour modifier l'attribut `_marque` en lui attribuant la valeur `Nokia`. La seconde instruction fait appel à l'accesseur pour afficher la valeur de l'attribut `_marque` à l'aide de la fonction `print()`.

Cette façon de faire peut être appliquée à tous les attributs, dans la mesure où il est nécessaire d'accéder et de modifier leurs valeurs.

Le code qui suit contient toutes les modifications pour tous les attributs de la classe `Telephone`.

```
class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_annee = 2002):

        self._marque = ""
        self._modele = val_modele
        self._couleur = val_couleur
        self._annee = val_annee

    def aff_attribut(self):
        print("Valeur pour l'attribut marque : ", self._marque)
        print("Valeur pour l'attribut modele : ", self._modele)
        print("Valeur pour l'attribut couleur : ", self._couleur)
        print("Valeur pour l'attribut annee : ", self._annee)

    def calc_cote(self, annee_en_cours, valeur_achat, serie_spec = False):

        bonus = 0
        nb_annee = annee_en_cours - self._annee
        if nb_annee < 10:
            decote = 0.1 * nb_annee
        else:
            decote = 0.9

        if serie_spec == True:
            bonus = 0.1 * valeur_achat

        valeur = valeur_achat - (valeur_achat * decote) + bonus

        return valeur

    @property
    def marque(self):
        return self._marque

    @marque.setter
    def marque(self, value):
        self._marque = value

    @property
    def modele(self):
        return self._modele

    @modele.setter
    def modele(self, value):
        self._modele = value

    @property
    def couleur(self):
        return self._couleur

    @couleur.setter
    def couleur(self, value):
        self._couleur = value
```

```

@property
def annee(self):
    return self._annee

@annee.setter
def annee(self, value):
    self._annee = value

print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 1997)
# modification de l'attribut marque
t1.marque = "Nokia"
# Affichage de toutes les valeurs des attributs
print("La marque de t1 est : ", t1.marque)

```

Ce qui produit bien le résultat escompté, c'est-à-dire la modification de la valeur de l'attribut `_marque` (Nokia) :

```

In [23]: %run encapsulation.py
Instance t1 avec tous les arguments
La marque de t1 est : Nokia

```

- **Si vous voulez permettre l'accès à un attribut uniquement en lecture seule, il suffit de ne définir que l'accessor, en d'autres termes, ne pas définir de mutateur.**

Méthodes spéciales

Python ne permet pas la surcharge de fonction et la stratégie pour pallier ce manque avec les paramètres par défaut a été présentée. Il existe encore un cas de figure où la surcharge est utile. Il s'agit de la surcharge d'opérateur.

La surcharge d'opérateur consiste à réécrire le comportement d'un opérateur (arithmétique par exemple) pour l'adapter à la structure d'une classe définie par l'utilisateur. À titre d'exemple, il y a peu de chance que l'opérateur `+` puisse être utilisé directement entre deux instances d'une classe définie par l'utilisateur. Il est donc nécessaire de définir le comportement de l'opérateur d'addition lorsqu'il est appelé sur de tels objets.

En Python, pour réaliser cette opération, il existe un ensemble de méthodes dites spéciales. Pour être tout à fait exact, les méthodes spéciales ne servent pas uniquement à surcharger des opérateurs, elles peuvent être utilisées pour redéfinir le comportement des fonctions.

Par exemple, lorsque vous tapez dans une console IPython (après l'exécution de votre script) :

```
In [2]: t1
```

vous obtenez en sortie :

```
Out[2]: <_main_.Telephone at 0x745fc630>
```

Cette manipulation est généralement faite à des fins de débogage. Or, les informations obtenues en sortie sont peu utiles pour détecter un bogue.

La fonction spéciale `repr` permet de redéfinir le comportement de la fonction mise en œuvre ici et qui permet de contrôler comment est représenté un objet pour obtenir des informations plus intéressantes. Cette redéfinition se fait en ajoutant directement une méthode supplémentaire au sein de la classe `Telephone` :

```
def_repr__(self):
    return "Année : " + str(self.annee) + " Marque : " +
self.marque + \
    " Modele : " + self.modele + " Couleur : " +
self.couleur
```

De cette manière, nous obtenons bien l’affichage de tous les attributs quand nous tapons `t1` dans l’environnement IPython.

```
In [8]: t1
Out[8]: Année : 1997 Marque : Nokia Modele : 3310 Couleur : jaune
```

La seconde fonction spéciale qui est fort utile est `__str__`. Elle permet de surcharger la fonction `print()`. De cette manière, il est possible d’exécuter directement l’instruction `print(instance_objet)` afin d’obtenir un affichage des informations sur mesure définies par l’utilisateur.

Redéfinissons la fonction `print()` de manière à afficher tous les attributs :

```
def_str__(self):
    return "Valeur pour l’attribut marque : " + self._marque + \
        "\nValeur pour l’attribut modele : " + self._modele + \
        "\nValeur pour l’attribut couleur : " + self._couleur + \
        "\nValeur pour l’attribut annee : " + str(self._annee)
```

Nous pouvons maintenant appeler directement la fonction `print()` sur une instance de la classe :

```
In [17]: print(t1)
Valeur pour l’attribut marque : Nokia
Valeur pour l’attribut modele : 3310
Valeur pour l’attribut couleur : jaune
Valeur pour l’attribut annee : 1997
```

Cet exemple d’utilisation de `print()` est réalisé dans l’environnement IPython, mais il est possible d’utiliser `print()` dans un script et sur n’importe quelle instance de l’objet `Telephone`.

Les fonctions spéciales vues jusqu’ici redéfinissent le comportement de fonctions. Mais il en existe d’autres qui permettent de redéfinir le comportement d’opérateurs.

Prenons l’opérateur arithmétique `+`. L’idée est d’additionner deux instances de la classe `Telephone`, `t1 + t2`, pour obtenir la somme des valeurs de `t1` et `t2`. Pour réaliser cela, il faut redéfinir l’opération d’addition à l’aide de la fonction spéciale `__add__`.

Avant de redéfinir cet opérateur, il est nécessaire de créer un attribut permettant de coder la valeur nommé `_valeur` qui sera passé en paramètre lors de la construction :

```
class Telephone(object):

    def_init__(self, val_modele, val_couleur, val_valeur = 750,
        val_annee = 2002):
        self._marque = ""
        self._modele = val_modele
        self._couleur = val_couleur
        self._annee = val_annee
        self._valeur = val_valeur
```

Il est également nécessaire d'ajouter l'accesseur et le mutateur pour manipuler cet attribut.

```
@property
def valeur(self):
    return self._valeur

@valeur.setter
def valeur(self, value):
    self._valeur = value
```

Maintenant, il est possible de redéfinir le comportement de l'opérateur d'addition à l'aide du code suivant :

```
def _add_(self, objet_add):
    val_tot = self._valeur + objet_add.valeur
    return val_tot
```

Cette fonction comporte, outre le paramètre `self`, un second paramètre qui correspond à l'opérande à droite du symbole de l'addition. Dans ce cas, cet objet est supposé être un objet de type `Telephone`.

Par conséquent, la valeur retournée par la fonction `_add_` est la somme de l'attribut `_valeur` de la classe et de l'attribut `_valeur` de l'objet passé en paramètre accessible grâce à l'accesseur de la classe (d'où la syntaxe `objet_add.valeur`).

Il est donc maintenant possible d'écrire le code pour appliquer cette addition sur deux instances de la classe :

```
print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 35, 1997)

print("Instance t2 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t2 = Telephone("3310", "carbone", 45, 2000)

# utilisation de l'opérateur + redéfini pour la classe Telephone
val_cumul = t1 + t2
print("Valeur cumulée des deux téléphones : ", val_cumul)
```

L'exécution du script modifié dans IPython produit :

```
In [20]: %run fonctions_speciales.py
Instance t1 avec tous les arguments
Instance t2 avec tous les arguments
Valeur cumulée des deux téléphones : 80
```

La valeur obtenue, 80, est bien la somme des deux valeurs cumulées, ce qui valide le fonctionnement de la redéfinition de l'opérateur d'addition.

Ce qui a été illustré avec l'opérateur d'addition est valable pour un grand nombre d'opérateurs pour lesquels nous allons dresser une liste, mais sans illustrer le fonctionnement par un exemple, ce qui n'aurait guère d'intérêt :

- `__sub__` pour redéfinir l'opérateur `-`
- `__mul__` pour redéfinir l'opérateur `*`
- `__truediv__` pour redéfinir l'opérateur `/`
- `__floordiv__` pour redéfinir l'opérateur `//` (division entière)
- `__mod__` pour redéfinir l'opérateur `%` (modulo)
- `__eq__` pour redéfinir l'opérateur `==`
- `__ne__` pour redéfinir l'opérateur `!=`
- `__gt__` pour redéfinir l'opérateur `>`
- `__ge__` pour redéfinir l'opérateur `≥`
- `__lt__` pour redéfinir l'opérateur `<`
- `__le__` pour redéfinir l'opérateur `≤`

Héritage

L'héritage est une autre manière de réutiliser du code existant, c'est-à-dire de le capitaliser. Le principe est le suivant : une classe dite « fille » hérite des méthodes et attributs de la classe « mère » dont elle dérive.

Généralement, la classe fille est plus spécifique que la classe mère : elle récupère tous les attributs et méthodes de la classe mère et elle est enrichie de nouveaux attributs et de nouvelles méthodes.

Afin d'illustrer notre propos, créons une classe `Smartphone` qui dérive de la classe `Telephone`, mais qui a en plus un attribut `_os`, ainsi que les accesseurs et mutateurs associés.

```
class Smartphone(Telephone):

    def __init__(self, val_modele, val_couleur, val_os = "Symbian", val_valeur=750,
                 val_annee=2002):
        super(Smartphone, self).__init__(val_modele, val_couleur, val_valeur,
                                         val_annee)

        self._os = val_os

    @property
    def os(self):
        return self._os

    @os.setter
    def os(self, value):
        self._os = value
```

Dans la définition de la classe `Smartphone`, la classe `Telephone` apparaît entre parenthèses. Cela signifie que la classe `Smartphone` dérive de la classe `Telephone`. Par conséquent, elle hérite de tous les attributs et méthodes de la classe `Telephone`.

Le constructeur de la classe `Smartphone` a en paramètre une valeur permettant d'initialiser le nouvel attribut de la classe ainsi que les valeurs permettant d'initialiser les attributs hérités de la classe `Telephone`.

Cette initialisation des attributs de la classe mère se fait à l'aide de la ligne suivante :

```
super(Smartphone, self)._init_(val_modele, val_couleur, val_valeur,
val_annee)
```

La fonction `super()` prend comme premier paramètre le nom de la classe fille et comme deuxième paramètre une instance d'objet, ici `self`, l'instance courante.

À l'aide de `super()`, nous allons explicitement appeler le constructeur de la classe mère, ce qui infine permet d'initialiser les attributs de la classe mère et par conséquent d'en faire usage dans la classe fille.

Il est donc possible maintenant de réaliser la modification du script pour utiliser la classe `Smartphone`.

```
print("Instance sp de la classe Smartphone dérivée de la classe Telephone") #
Instanciation de la classe en précisant tous les paramètres
sp = Smartphone("6820", "Gris", "Symbian", 300, 2005)

print("Modification de l'attribut marque") sp.marque =
"Nokia"
print("Marque de l'instance : ", sp.marque) print("Os de
l'instance : ", sp.os)
```

Dans cet exemple, un objet `sp` est instancié, puis l'attribut `_marque` (qui provient de la classe mère) est modifié en utilisant le mutateur. Enfin, l'attribut `_marque` dérivé de la classe mère est affiché, ainsi que l'attribut `_os`. Pour les deux, l'accès aux données se fait à l'aide de l'accesseur. Ceci produit à l'exécution le résultat suivant :

```
In [25]: %run heritage.py
```

```
Instance sp de la classe Smartphone dérivée de la classe Telephone
```

```
Modification de l'attribut marque
```

```
Marque de l'instance : Nokia Os
```

```
de l'instance : Symbian
```

Voici le code complet de l'exemple qui nous a servi à illustrer les concepts de la POO :

```
class Telephone(object):

    def __init__(self, val_modele, val_couleur, val_valeur = 750, val_annee =
        2002):
        self._marque = "" self._modele
        = val_modele self._couleur =
        val_couleur self._annee =
        val_annee self._valeur =
        val_valeur

    def __repr__(self):
        return "Année : " + str(self.annee) + " Marque : " + self.marque +\
            " Modele : " + self.modele + " Couleur : " + self.couleur +\
            " Valeur : "
            + str(self.valeur)

    def __str__(self):
        return "Valeur pour l'attribut marque : " + self._marque + \
            "\nValeur pour l'attribut modele : " + self._modele +\
            "\nValeur pour l'attribut couleur : " + self._couleur +\
            "\nValeur pour l'attribut valeur : " + self._valeur +\
```

```
"\nValeur pour l'attribut annee : " + str(self._annee)
```

```
def_add(self, objet_add):
```

```
    val_tot = self._valeur + objet_add.valeur
```

```
    return val_tot
```

```
def aff_attribut(self):
```

```
    print("Valeur pour l'attribut marque : ", self._marque) print("Valeur pour
```

```
    l'attribut modele : ", self._modele) print("Valeur pour l'attribut couleur :
```

```
    ", self._couleur) print("Valeur pour l'attribut valeur à gauche : ",
```

```
    self._valeur) print("Valeur pour l'attribut annee : ", self._annee)
```

```
def calc_cote(self, annee_en_cours, valeur_achat, serie_spec = False):
```

```
    bonus = 0
```

```
    nb_annee = annee_en_cours - self._annee
```

```
    if nb_annee < 10:
```

```
        decote = 0.1 * nb_annee
```

```
    else:
```

```
        decote = 0.9
```

```
    if serie_spec == True:
```

```
        bonus = 0.1 * valeur_achat
```

```
    valeur = valeur_achat - (valeur_achat * decote) + bonus return
```

```
    valeur
```

```
@property
```

```
def marque(self): return
```

```
    self._marque
```

```
@marque.setter
```

```
def marque(self, value):
```

```
    self._marque = value
```

```
@property
```

```
def modele(self): return
```

```
    self._modele
```

```
@modele.setter
```

```
def modele(self, value):
```

```
    self._modele = value
```

```
@property
```

```
def couleur(self): return
```

```
    self._couleur
```

```
@couleur.setter
```

```
def couleur(self, value):
```

```
    self._couleur = value
```



```
@property
def annee(self): return
    self._annee
```

```
@annee.setter
def annee(self, value):
    self._annee = value
```

```
@property
def valeur(self): return
    self._valeur
```

```
@valeur.setter
def valeur(self, value):
    self._valeur = value
```

```
class Smartphone(Telephone):
```

```
    def __init__(self, val_modele, val_couleur, val_os = "Symbian",
                  val_valeur=750, val_annee=2002):
        super(Smartphone, self).__init__(val_modele, val_couleur, val_valeur,
                                          val_annee)
        self._os = val_os
```

```
    @property
    def os(self):
        return self._os
```

```
    @os.setter
    def os(self, value): self._os
        = value
```

```
print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres t1 =
Telephone("3310", "jaune", 35, 1997)

print("Instance t2 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres t2 =
Telephone("3310", "carbone", 45, 200)

# utilisation de l'opérateur + redéfini pour la classe Telephone val_cumul
= t1 + t2
print("Valeur cumulée des deux téléphones : ", val_cumul)
print("Instance sp de la classe Smartphone dérivée de la classe Telephone")

# Instanciation de la classe en précisant tous les paramètres sp =
Smartphone("6820", "Gris", "Symbian", 300, 2005)

print("Modification de l'attribut marque") sp.marque =
"Nokia"

print("Marque de l'instance : ", sp.marque) print("Os de l'instance : ", sp.os)
```


Créer ses propres modules et packages

Modules

La création de modules a déjà été abordée implicitement dans la section sur les fonctions. En fait, un module est simplement un regroupement de fonctions ou de classes dans un fichier.

L'utilisation des éléments formant le module à l'intérieur d'une autre source se fait à l'aide des instructions `from` et `import`.

Si nous consignons la définition des classes `Telephone` et `Smartphone` dans le fichier `telephonie.py`, il est nécessaire d'importer les deux classes en début de fichier, de la manière suivante :

```
from telephonie import Telephone
from telephonie import Smartphone
```

```
from telephonie import Telephone
from telephonie import Smartphone

print("Instance t1 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t1 = Telephone("3310", "jaune", 35, 1997)

print("Instance t2 avec tous les arguments")
# Instanciation de la classe en précisant tous les paramètres
t2 = Telephone("3310", "carbone", 45, 200)

# utilisation de l'opérateur + redéfini pour la classe Telephone
val_cumul = t1 + t2
print("Valeur cumulée des deux téléphones : ", val_cumul)
print("Instance sp de la classe Smartphone dérivée de la classe Telephone")

# Instanciation de la classe en précisant tous les paramètres
sp = Smartphone("6820", "Gris", "Symbian", 300, 2005)

print("Modification de l'attribut marque")
sp.marque = "Nokia"

print("Marque de l'instance : ", sp.marque)
print("Os de l'instance : ", sp.os)
```

Ce faisant, nous obtenons l'exemple suivant :

Ce qui produit bien toujours le bon résultat :

```
In [28]: %run test_module.py
Instance t1 avec tous les arguments
Instance t2 avec tous les arguments
Valeur cumulée des deux téléphones : 80
Instance sp de la classe Smartphone dérivée de la classe Telephone
Modification de l'attribut marque
Marque de l'instance : Nokia
Os de l'instance : Symbian
```

Cette manière de faire est une possibilité, mais ce n'est pas la seule pour rendre le code modulable et réutilisable. Une solution existe et est beaucoup plus utilisée par la communauté Python : les packages.

Packages

Il est facile de créer un package en Python. En effet, un package est simplement un regroupement de modules dans un dossier. Ici, le fichier `telephonie.py` est déplacé dans le package nommé `package_tel` dont le dossier éponyme doit être créé au préalable.

L'importation des classes se fait maintenant de la manière suivante :

```
from package_tel.telephonie import Telephone
from package_tel.telephonie import Smartphone
```

La seule différence avec les modules est qu'il faut mettre le nom du package avant le nom du fichier.

- **Au fil de vos lectures et recherches de packages existants, vous pourrez trouver des packages qui possèdent un fichier `__init__.py` qui peut être vide ou contenir un code d'initialisation. En Python 3, ce fichier est optionnel alors qu'il est obligatoire en Python 2. La présence de ce fichier est motivée pour des raisons de compatibilité.**

Travaux pratiques

Le problème

Nous proposons ici de coder deux méthodes de cryptage qui sont :

- le cryptage de Vigenère
- le cryptage par décalage

Avant toute chose, définissons comment fonctionne chaque méthode (source Wikipédia) :

- Le cryptage de Vigenère : « Ce chiffrement introduit la notion de clé. Une clé se présente généralement sous la forme d'un mot ou d'une phrase. Pour pouvoir chiffrer notre texte, à chaque caractère nous utilisons une lettre de la clé pour effectuer la substitution. Évidemment, plus la clé sera longue et variée et mieux le texte sera chiffré. Il faut savoir qu'il y a eu une période où des passages entiers d'œuvres littéraires étaient utilisés pour chiffrer les plus grands secrets. Les deux correspondants n'avaient plus qu'à avoir en leurs mains un exemplaire du même livre pour s'assurer de la bonne compréhension des messages. ».

Nous nous limiterons à utiliser une clé de la même taille que le texte à coder afin de réduire la difficulté, mais en conservant le principe.

- Le cryptage par décalage : « En cryptographie, le chiffrement par décalage, aussi connu comme le chiffre de César ou le code de César [...], est une méthode de chiffrement très simple utilisée par Jules César dans ses correspondances secrètes (ce qui explique le nom "chiffre de César").

Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Par exemple, avec un décalage de 3 vers la droite, A est remplacé par D, B devient E, et ainsi jusqu'à W qui devient Z, puis X devient A, etc. Il s'agit d'une permutation circulaire de l'alphabet. La longueur du décalage, 3 dans l'exemple évoqué, constitue la clé du chiffrement qu'il suffit de transmettre au destinataire – s'il sait déjà qu'il s'agit d'un chiffrement de César – pour que celui-ci puisse déchiffrer le message. Dans le cas de l'alphabet latin, le chiffre de César n'a que 26 clés possibles (y compris la clé nulle, qui ne modifie pas le texte). »

Solution

Dans un premier temps, créons la classe mère qui possède les méthodes publiques qui permettent d'appeler les méthodes des classes filles lorsqu'un objet est instancié :

```
class BaseChiffrement(object):

    def _check_input(self, texte, cle):
        if not texte.isalpha():
            raise ValueError(
                '"texte" ne doit contenir que des lettres alphabétiques.')
        if cle is not None:
            if len(texte) != len(cle):
                raise ValueError('La clé doit avoir le même nombre de '
                                   'caractères que le texte à encoder.')
            if not cle.isalpha():
                raise ValueError(
                    '"cle" ne doit contenir que des lettres alphabétiques.')

    def encoder(self, texte_clair, cle=None):
        self._check_input(texte_clair, cle)
        if cle is None:
            return self._encoder(texte_clair)
        else:
```

```

        return self._encoder(texte_clair, cle)

def decoder(self, texte_chiffre, cle=None):
    self._check_input(texte_chiffre, cle)
    if cle is None:
        return self._decoder(texte_chiffre)
    else:
        return self._decoder(texte_chiffre, cle)

```

Définissons maintenant la classe permettant le cryptage par décalage :

```

class ChiffrementDecalage(BaseChiffrement):

    def _init_(self, decalage=None):
        if decalage is None:
            self.decalage = randint(0, 26)
        else:
            self.decalage = decalage
        self.encode_dict = {chr(ord('a') + i):
                            chr(ord('a') + (i + self.decalage) % 26)
                            for i in range(26)}
        self.decode_dict = {chr(ord('a') + i):
                             chr(ord('a') + (i - self.decalage) % 26)
                             for i in range(26)}

    def _encoder(self, texte_clair):
        resultat = [self.encode_dict[caractere] for caractere in texte_clair]
        return ''.join(resultat)

    def _decoder(self, texte_chiffre):
        resultat = [self.decode_dict[caractere] for caractere in texte_chiffre]
        return ''.join(resultat)

```

Cette classe qui dérive de la classe `BaseChiffrement` comporte le code pour l'encodage et le décodage d'une chaîne.

Le fonctionnement est strictement le même pour la classe qui gère le chiffrement de Vigenère :

```

class ChiffrementVigenere(BaseChiffrement):

    def _encoder(self, texte_clair, cle=None):
        if cle is None:
            caracteres_int = [randint(0, 26) for i in range(len(texte_clair))]
            cle = ''.join([chr(ord('a') + caractere)
                           for caractere in caracteres_int])

        return (''.join([chr(ord('a') +
                             ((ord(t) - ord('a')) + (ord(c) - ord('a')) % 26)
                             for t, c in zip(texte_clair, cle)]),
                        cle)

    def _decoder(self, texte_chiffre, cle):
        return ''.join([chr(ord('a') +

```

```
((ord(t) - ord('a')) - (ord(c) - ord('a'))) % 26)
for t, c in zip(texte_chiffre, cle)]]
```

Nous pouvons maintenant tester les classes :

```
from chiffrement import ChiffrementVigenere
from chiffrement import ChiffrementDecalage

vignere = ChiffrementVigenere()
decalage = ChiffrementDecalage(12)

chaine = "bonjour"

chaine_encode_v = vignere.encoder(chaine, "abbpoau")
print("Chaine encodée avec Vigenere : ", chaine_encode_v[0])
chaine_decode_v = vignere.decoder(chaine_encode_v[0], "abbpoau")
print("Chaine décodée avec Vigenere : ", chaine_decode_v)

chaine_encode_d = decalage.encoder(chaine)
print("Chaine encodée avec décalage : ", chaine_encode_d)
chaine_decode_d = decalage.decoder(chaine_encode_d)
print("Chaine décodée avec décalage : ", chaine_decode_d)
```

Ce qui produit le résultat suivant :

```
In [1]: %run test_chiffrement.py
Chaine encodée avec Vigenere : bpoycul
Chaine décodée avec Vigenere : bonjour
Chaine encodée avec décalage : nazvagd
Chaine décodée avec décalage : bonjour
```

Afin d'avoir une vision globale de la solution, voici l'ensemble du code :

```
from random import randint

class BaseChiffrement(object):

    def _check_input(self, texte, cle):
        if not texte.isalpha():
            raise ValueError(
                '"texte" ne doit contenir que des lettres alphabétiques.')
        if cle is not None:
            if len(texte) != len(cle):
                raise ValueError('La cle doit avoir le même nombre de '
                                   'caractères que le texte a encode.')
            if not cle.isalpha():
                raise ValueError(
                    '"cle" doit contenir que des lettres alphabétiques.')

    def encoder(self, texte_clair, cle=None):
        self._check_input(texte_clair, cle)
        if cle is None:
```

```

        return self._encoder(texte_clair)
    else:
        return self._encoder(texte_clair, cle)

def decoder(self, texte_chiffre, cle=None):
    self._check_input(texte_chiffre, cle) if cle
    is None:
        return self._decoder(texte_chiffre) else:
        return self._decoder(texte_chiffre, cle)

```

```
class ChiffrementDecalage(BaseChiffrement): def
```

```

    init_(self, decalage=None):
        if decalage is None:
            self.decalage = randint(0, 26) else:
            self.decalage = decalage
        self.encode_dict = {chr(ord('a') + i):
                            chr(ord('a') + (i + self.decalage) % 26) for i
                            in range(26)}
        self.decode_dict = {chr(ord('a') + i):
                            chr(ord('a') + (i - self.decalage) % 26) for i in
                            range(26)}

    def _encoder(self, texte_clair):
        resultat = [self.encode_dict[caractere] for caractere in texte_clair] return
        "".join(resultat)

    def _decoder(self, texte_chiffre):
        resultat = [self.decode_dict[caractere] for caractere in texte_chiffre] return
        "".join(resultat)

```

```
class ChiffrementVigenere(BaseChiffrement): def
```

```

    _encoder(self, texte_clair, cle=None):
        if cle is None:
            caracteres_int = [randint(0, 26) for i in range(len(texte_clair))] cle =
            "".join([chr(ord('a') + caractere)
                    for caractere in caracteres_int])

        return ("".join([chr(ord('a') +
                            ((ord(t) - ord('a')) + (ord(c) - ord('a'))) % 26) for t, c
                            in zip(texte_clair, cle)]),
                cle)

    def _decoder(self, texte_chiffre, cle): return
        "".join([chr(ord('a') +
                            ((ord(t) - ord('a')) - (ord(c) - ord('a'))) % 26) for t, c in
                            zip(texte_chiffre, cle)])

```


8. Définition et intérêt du GPIO

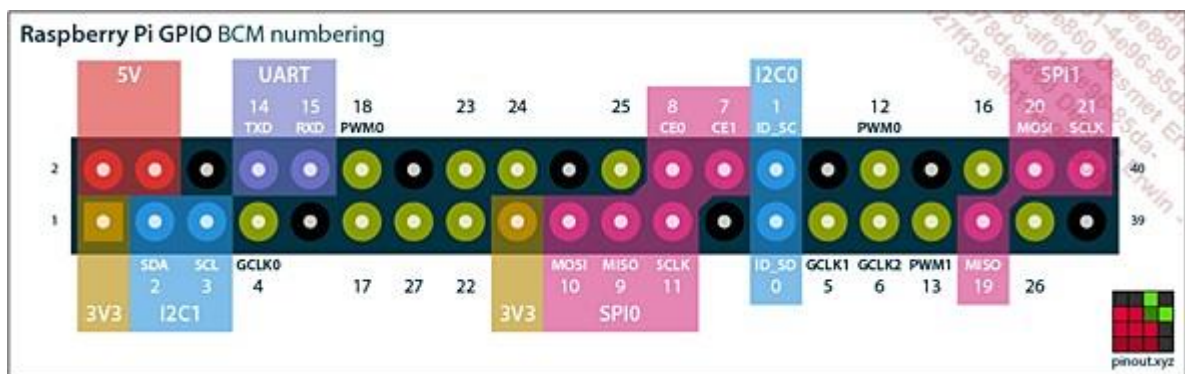
Une des caractéristiques pouvant expliquer le succès du Raspberry Pi est le fait qu'il possède une interface matérielle d'entrée/sortie permettant de communiquer avec différents instruments, dont des capteurs. On peut donc voir ici le Raspberry Pi comme un microcontrôleur qui offre l'avantage d'accueillir un système d'exploitation complet et autorise nativement l'accès à des services qu'on pourrait difficilement retrouver sur des microcontrôleurs classiques.

Cette particularité permet de répondre aux besoins de prototypage d'informaticiens n'ayant pas les compétences bas niveau pour manipuler le matériel. En effet, la couche logicielle permet une abstraction qui simplifie l'accès au matériel à l'aide de bibliothèques ; dans ce cas, les packages Python.

Les électroniciens dans une tâche de prototypage d'un capteur bénéficient d'un gain de temps important en utilisant les services du système d'exploitation : Bluetooth, réseau... sans avoir recours à une implantation fastidieuse et chronophage sur une plateforme matérielle classique, par exemple l'ARM Cortex M4.

Le port GPIO (*General Purpose Input Out*) comporte un connecteur possédant deux séries de vingt broches tant pour le modèle 3 B que pour les modèles Zero 1.3 et Zero W. La seule différence étant que les modèles Zero ne possèdent pas de connecteur soudé en usine, vous devrez donc le souder vous-même.

Voici un schéma des différentes broches du port GPIO (crédit : <https://pinout.xyz>) :



Notez que la broche numéro 1 est symbolisée par une pastille en cuivre carrée sur le circuit imprimé.

Maintenant que nous avons une référence pour nous repérer sur le port, nous pouvons énumérer les différents éléments constituant ce port :

- Les broches d'alimentation positive (VCC) : 1 et 17 (3,3 V-500 mA), 2 et 4 (5 V- jusqu'à 1,5 A suivant votre alimentation).
- Les broches de masse (GND) : 6, 9, 14, 20, 25, 30, 34 et 39 (toutes connectées en interne).
- Le port DPI (*Display Parallel Interface*) permettant l'affichage : toutes les broches hors VCC et GND.
- Le port GPCLK (*General Purpose Clock*) permettant d'avoir en sortie différentes horloges sans procéder à des divisions d'horloges internes : 7, 29 et 31.
- Le port JTAG (*Joint Test Action Group*) permettant de connecter une sonde de débogage pour analyser votre Raspberry Pi (registres, mémoires...) : 7, 13, 15, 16, 18, 22, 29, 31, 32, 33 et 37.
- Le port PCM (*Pulse-Code Modulation*) permettant de générer des signaux à fournir à un convertisseur numérique analogique : 12, 35, 38 et 40.
- SDIO (*SD Card Interface*) permettant de connecter un lecteur de carte SD ou eMMC : 13, 15, 16, 22 et 37.
- I2C (*Inter Integrated Circuit*) permettant la manipulation de composants connectés sur le port I2C : 2, 5, 27, 28.
- SPI (*Serial Peripheral Interface*) permettant la manipulation de composants connectés via le standard SPI : 11, 12, 19, 21, 23, 24, 26, 35, 36, 38 et 40.
- Les broches d'entrée/sortie numériques simples (changement d'état logique : 1 ou 0) : 7, 8, 10, 11, 12, 13, 15, 16, 18, 19, 21, 22, 23, 24, 26, 27, 28, 29, 31, 32, 33, 35, 36, 37, 38 et 40.

Comme vous pouvez le voir, un certain nombre de broches sont communes à différents ports. Par conséquent, vous

ne pouvez pas utiliser simultanément deux ports qui ont des broches communes. Il est donc nécessaire de consulter le schéma d'occupation des broches et de bien définir vos besoins.

Avant d'utiliser un ou plusieurs éléments du port GPIO, lisez en détail la documentation technique du composant avec lequel vous voulez réaliser l'interfaçage du Raspberry Pi. La lecture de ce document doit a minima répondre à deux questions :

- Est-ce que les caractéristiques électriques de mon composant sont compatibles avec le port ?
- Comment dois-je câbler le composant sur le port ?

Sans les réponses à ces deux questions, l'intégrité de votre port GPIO n'est pas garantie. En effet, les éléments du port sont sensibles aux surcharges et courts-circuits.

Dans la suite de ce chapitre, nous nous focaliserons sur deux types de ports largement utilisés dans la communauté Raspberry Pi, et plus généralement dans le secteur de l'électronique numérique grand public, qui sont les broches d'entrée/sortie numériques (tout ou rien) et le port I2C.

9. Broches d'entrée/sortie numériques

Les broches d'entrée/sortie numériques permettent d'afficher en sortie, lorsqu'elles sont configurées en écriture, la valeur logique 1 ou 0. Elles permettent de lire l'état logique imposé par un capteur lorsque la broche est configurée en lecture.

Nous allons réaliser ce processus de lecture et d'écriture à l'aide d'une bibliothèque officielle Python développée pour le système Raspbian. Cette librairie se nomme RPi.GPIO. Il est probable qu'elle soit installée, mais les lignes ci-après permettent de réaliser soit l'installation si la librairie est absente, soit sa mise à jour.

```
sudo apt-get update
sudo apt-get python-dev python-rpi.gpio
```

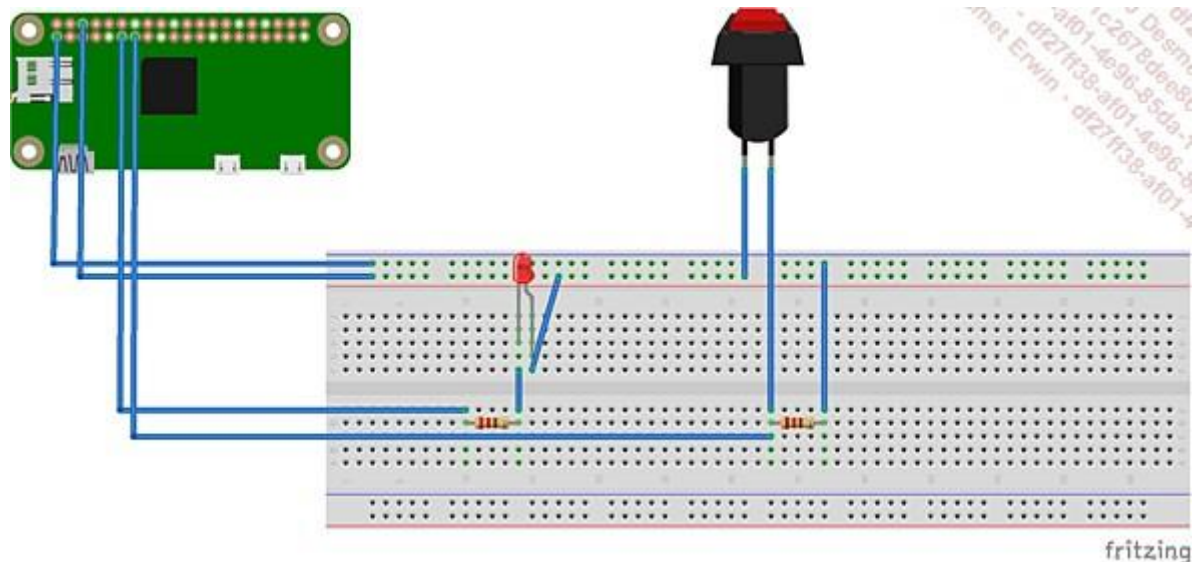
Une fois cette librairie installée, nous pouvons concevoir un montage électronique sommaire pour simuler la présence d'entrée et de sortie sur certaines des broches du montage.

Pour cela, nous avons besoin de :

- deux résistances
- une LED
- un switch
- et quelques câbles de connexion

La LED est câblée sur la broche BCM17 qui est définie comme une sortie et le bouton poussoir est câblé juste avant la résistance sur la broche BCM27 qui est définie comme une entrée.

Ce montage est illustré à la figure suivante :



Commençons par importer la librairie permettant de manipuler les entrées et sorties logiques :

```
import RPi.GPIO as GPIO
```

Lors de cette importation, nous créons directement un objet nommé GPIO.

Passons en mode de numérotation BCM :

```
GPIO.setmode(GPIO.BCM)
```

Pour manipuler la LED comme une sortie, définissons la broche BCM17 comme une sortie :

```
GPIO.setup(17, GPIO.OUT)
```

Puis, pour lire l'état de la broche connectée au bouton poussoir, définissons la broche BCM27 comme une entrée :

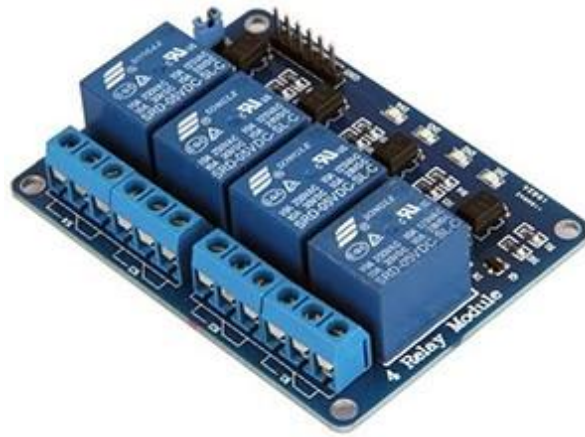
```
GPIO.setup(27, GPIO.IN)
```

Passons maintenant au programme très simple qui consiste à allumer la LED lorsque le bouton poussoir est fermé. En d'autres termes, lorsqu'un état haut est appliqué sur la broche BCM27, alors un état haut est appliqué sur la broche BCM17.

```
while True:
    if(GPIO.input(27) is True):
        GPIO.output(17) is True
    else :
        GPIO.output(17) is False
```

Voilà, dix lignes de code suffisent pour manipuler des composants extérieurs à votre Raspberry. De cette manière, vous pouvez lire tous les capteurs fournissant des informations tout ou rien et produire des états hauts ou des états bas sur les sorties du port GPIO.

Vous pouvez par exemple écrire sur des écrans LCD ou mettre en route des appareils de puissance par l'intermédiaire de la commande d'un relais, c'est-à-dire un appareil ayant un besoin d'alimentation supérieur aux capacités d'alimentation du Raspberry Pi et par le fait nécessitant une alimentation externe telle une pompe pour arroser votre jardin !



10. Port I2C

I2C est proche du port série RS232, mais possède le grand avantage de connecter plusieurs éléments sur un même port, grâce à un système d'adressage des périphériques.

Pour faire simple, nous allons instancier un objet qui permet de manipuler un composant du port. À l'instanciation, nous allons passer comme paramètre l'adresse du composant sur le port. Les lectures et écritures en direction de ce composant seront donc effectuées grâce à son adresse. Afin de manipuler différents éléments connectés sur le port, il est possible d'instancier plusieurs objets. La seule limite étant le nombre maximum de composants connectables ainsi que le débit maximum du port (100 ou 400 kHz selon la configuration du port et les spécifications du composant).

Avant d'écrire le code relatif à la manipulation du port GPIO, il est nécessaire de réaliser l'activation du port I2C.

Pour cela, il faut passer par l'outil de configuration du Raspberry à l'aide de l'instruction :

```
sudo raspi-config
```



Puis, il faut sélectionner l'option **5Interfacing Options** et ensuite **P5 I2C**. Pour finalement choisir d'activer le port I2C à l'aide de l'option **Yes**.

Une fois l' I2C activé, nous pouvons scanner son port en utilisant l'instruction :

```
sudo i2cdetect -y 1
```

Dans ce cas, deux capteurs sont présents à l'adresse 0x48 et 0x76.

```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  48  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  76  --  --  --  --  --  --  --  --  --
pi@raspberrypi:~ $
```

Manipuler nativement le port I2C est un peu fastidieux. Il existe une librairie dont le développement a été initié par Adafruit et qui encapsule la manipulation du port I2C pour un usage plus simple.

Pour installer cette librairie, il faut se rendre sur le dépôt GitHub https://github.com/adafruit/Adafruit_Python_GPIO, et exécuter les instructions données par le propriétaire :

```
sudo apt-get update
sudo apt-get install build-essential python-pip python-dev python-smbus git
git clone https://github.com/adafruit/Adafruit_Python_GPIO.git
cd Adafruit_Python_GPIO
sudo python setup.py install
```

Pour utiliser ce package qui permet de manipuler cette librairie, c'est très simple. Il faut dans un premier temps importer la librairie :

```
import Adafruit_GPIO.I2C as I2C
```

Puis, nous pouvons instancier un objet en utilisant le constructeur qui attend comme paramètre l'adresse de l'esclave à adresser, ici 0x48.

```
device = i2c.get_i2c_device(0x48)
```

Une fois cette instanciation réalisée, nous pouvons réaliser toutes sortes de lectures et d'écritures.

À l'aide de cette librairie, il est possible de réaliser des écritures de mots de 8 bits :

```
registre_cible = 0xF0
val = 24
device.write8(registre_cible, val)
```

Pour écrire un mot de 16 bits :

```
registre_cible = 0xF0
val = 256
device.write8(registre_cible, val)
```

Il existe de la même manière des fonctions de lecture de registre en 16 et 8 bits. Ces fonctions sont dédoublées : `readU8()` et `readU16()` pour la lecture de mots non signés (unsigned), et `readS8()` et `readS16()` pour la lecture de mots signés. L'usage de l'une ou l'autre de ces versions dépend du type de données attendues dans le registre qui a été défini par le constructeur du composant.

Voyons à présent comment réaliser l'interfaçage d'un composant disponible sur le port I2C du Raspberry.

11. Acquérir des données analogiques

Le Raspberry Pi, quel que soit le modèle, ne possède pas de broche d'entrée/sortie analogique. À l'inverse de son plus sérieux concurrent l'Arduino, il n'est pas possible de connecter directement sur le GPIO des capteurs analogiques. Nous appelons ici capteur analogique un capteur qui mesure un phénomène et dont l'intensité ou la tension en sortie est directement (linéairement ou non) liée au phénomène mesuré.

Pour récupérer l'information provenant du capteur, il est nécessaire d'effectuer une conversion analogique/numérique. Cette conversion est facilement réalisable sans pour autant disposer de connaissances importantes en traitement du signal et en électronique numérique. En effet, il existe un nombre important de capteurs réalisant ce genre de tâches. Nous avons retenu le composant produit par Texas Instrument dont la référence est : ADS 1115.

Ce capteur présente les particularités suivantes :

- ♦ sortie de numérisation sur 16 bits
- ♦ quatre entrées analogiques
- ♦ gain programmable
- ♦ sortie en I2C
- ♦ adresse esclave I2C programmable entre 0x48 et 0x4B à l'aide d'un système de cavaliers

Pour résumer, l'ADS 1115 est un capteur qui permet de récupérer les données d'un capteur (plus généralement d'un signal) analogique à l'aide du port I2C. Ce capteur permet de numériser deux sources à la fois. Comme il est possible de mettre jusqu'à quatre composants ADS 1115 sur le port I2C, nous avons donc la possibilité de connecter 16 (4 X 4) sources de signal numérique sur le Raspberry Pi !

Passons maintenant à la pratique : l'acquisition des données d'un capteur d'humidité. Ce capteur est très simple. Il possède trois broches :

- ♦ VCC : entre 3,3 V et 5 V
- ♦ GND : 0 V
- ♦ VOUT : entre 0 et 3,7 V (si l'alimentation est en 5 V)

L'alimentation du capteur se fait à l'aide de VCC dont la valeur nominale est située entre 3,3 V et 5 V. Ici, nous choisissons d'avoir 5 V en alimentation. Cette source peut être prise directement sur le GPIO du Raspberry ; c'est l'alimentation qui permet de « tirer le plus de courant ».

Dans cette configuration, la broche VOUT propose en sortie une tension de 0 V en l'absence d'humidité et une tension de 3,7 V en saturation d'humidité (100 %).

Étalonner le capteur

Nous allons ici étalonner de façon succincte le capteur. En effet, nous allons supposer que le capteur a un comportement linéaire entre 0 % et 100 % d'humidité. En d'autres termes, cela signifie qu'une droite représente le comportement du pourcentage d'humidité en fonction de la tension fournie à la sortie du capteur (VOUT). La relation est donc de type $y = ax$ où x est la valeur de VOUT et y le taux d'humidité du sol. Il est simplement nécessaire de déterminer a . Pour cela, il est conseillé de réaliser une mesure fournissant la vérité de terrain. C'est-à-dire une valeur pour laquelle le taux d'humidité est connu sans erreur possible ou très faible. Le plus simple étant de tremper la partie capacitive du capteur dans un verre d'eau et de mesurer VOUT avec un voltmètre. Pour plus de précision, il est conseillé de réaliser plusieurs mesures successives et d'en faire la moyenne. De cette manière, il est possible de déterminer le paramètre de la relation linéaire précédemment

exposé.

Dans notre cas, nous avons $V_{OUT} 100\% = 3,43\text{ V}$, ce qui entraîne $a = 100/3,43 = 29,1545$.

Pour toute valeur numérique de la tension obtenue par une lecture du composant ADS 1115, il est nécessaire d'utiliser la relation suivante $T_{hum} = 29,1545 * V_{OUT}$ pour obtenir le taux d'humidité mesuré par le capteur d'humidité.

En pratique

Nous avons vu le fonctionnement général du capteur analogique de mesure de taux d'humidité d'un sol et la manière dont une valeur lue sur le port I2C peut être transformée en valeur numérique représentant le taux d'humidité. Nous allons maintenant voir comment récupérer des valeurs sur le composant ADS 1115.

Contrairement au cas du BME 280 précédemment exposé, nous allons utiliser ici une librairie existante qui possède les fonctionnalités dont nous avons besoin et bien d'autres. Cette librairie open source dont le développement a été initié par Adafruit est disponible sous licence MIT dans le dépôt GitHub suivant : https://github.com/adafruit/Adafruit_Python_ADS1X15

Pour installer cette librairie, il faut exécuter les lignes suivantes :

```
sudo apt-get install git build-essential python-dev
cd ~
git clone https://github.com/adafruit/Adafruit_Python_ADS1x15.git
cd Adafruit_Python_ADS1x15
sudo python setup.py install
```

Mais avant d'utiliser directement cette librairie, il est nécessaire de prendre connaissance du code pour voir quelles sont les fonctionnalités disponibles.

Nous allons donc éditer le fichier `./Adafruit_ADS1x15/ADS_1x15.py`.

La première information notable se trouve à la ligne 25 :

```
ADS1x15_DEFAULT_ADDRESS = 0x48
```

Cette ligne définit l'adresse par défaut du convertisseur sur le bus I2C : 0x48. Si, dans votre cas, l'adresse est différente, vous pouvez changer la valeur ici, mais comme vous pouvez le voir ci-après, le constructeur nous indique qu'il est possible de passer l'adresse du composant à lire lors de l'instanciation de la classe.

```
def __init__(self, address=ADS1x15_DEFAULT_ADDRESS, i2c=None,
**kwargs):
```

La dernière fonction dont il est impératif de prendre connaissance est la fonction de lecture simple disponible à la ligne 105.

```
def _read(self, mux, gain, data_rate, mode):
```

Cette fonction peut paraître un peu abstraite : elle permet de prendre en charge différents modes de fonctionnement de l'ADS 1115. Nous utiliserons le mode le plus simple, c'est-à-dire lire la voie 0 sur laquelle est connecté notre capteur avec un gain de 1. Dans ce cas, le code pour lire le capteur d'humidité devient :

```
import Adafruit_ADS1x15

tx_hum_adc = Adafruit_ADS1x15.ADS1115(address=0x4B)
vout_num = tx_hum_adc.read_adc(0, gain=1)
val_tx_hum = vout_num * 29.1545
```


Dans cet exemple, nousinstancions un objet de type ADS 1115 pour manipuler un composant connecté sur le port I2C dont l'adresse est 0x4B. Puis, nous procédons à la lecture de la voie 0 avec un gain de 1.

Pour finir, procédons à la conversion de la valeur VOUT en taux d'humidité. En effet, la lecture du convertisseur nous fournit une valeur numérique de VOUT, signal de sortie du capteur d'humidité. Il faut maintenant appliquer la relation linéaire qui lie VOUT et le taux d'humidité.

Pour afficher toutes les secondes le taux d'humidité, il suffit d'ajouter les lignes suivantes au script :

```
import Adafruit_ADS1x15
import time

tx_hum_adc = Adafruit_ADS1x15.ADS1115(address=0x4B)
while (True) :

    vout_num = tx_hum_adc.read_adc(0, gain=1)
    val_tx_hum = vout_num * 29.1545
    print('Valeur du taux d'humidité de votre sol : ' + val_tx_hum)
    time.sleep(1)
```

Vous avez maintenant toutes les clés en main pour réaliser l'acquisition de signaux analogiques à l'aide d'un convertisseur numérique connecté à votre Raspberry Pi. Ceci permet donc d'aller au-delà d'une limitation technologique du Raspberry Pi, c'est-à-dire l'absence de port d'entrée/sortie analogique au sein du GPIO.

12. Utiliser le Bluetooth

Le Raspberry Pi 3 B+ possède l'avantage de proposer un module Bluetooth 4.2. Cette version du format Bluetooth est optimisée pour l'Internet des objets (IoT, *Internet of Things*). Elle est particulièrement adaptée pour la connexion à des objets autonomes et pour l'échange de quantités de données importantes. En effet, la norme Bluetooth 4.2 permet d'avoir un débit de 25 Mbit/s au maximum, ainsi qu'une portée de 60 m.

Le Bluetooth reste un protocole difficile à exploiter même s'il existe des bibliothèques pour réduire la complexité de la tâche. Pour comprendre comment une communication Bluetooth entre un périphérique et un central fonctionne, il est nécessaire d'expliquer le fonctionnement du protocole.

D'une manière générale, un périphérique, qui est identifié de manière unique par une adresse MAC, propose un ou des services Bluetooth. Ces services sont normalisés par le consortium Bluetooth. À titre d'illustration, quelques services sont listés ci-après :

- CGMS (*Continuous Glucose Monitoring Service*) : suivi en continu de la glycémie
- CPS (*Cycling Power Service*) : service de gestion des capteurs de puissance pour le cyclisme
- ESS (*Environmental Sensing Service*) : service de gestion des paramètres environnementaux : température, pression...
- BAS (*Battery Service*) : service de gestion de la batterie : récupération d'informations indiquant le niveau, si la batterie est en charge...

Les services possèdent un niveau de description supplémentaire qu'on nomme caractéristiques. Ces caractéristiques sont les attributs du service, et permettent de converser en entrée ou en sortie avec le capteur.

Il existe un nombre important de services décrit par le consortium Bluetooth : <https://www.bluetooth.com/specifications/gatt>

Notez que plusieurs services peuvent cohabiter sur le même composant, par exemple un service pour la gestion de la batterie et un service pour la surveillance du glucose.

En outre, il est possible de créer des services de toutes pièces. C'est ce que fait l'entreprise Nordic Semiconductor, laquelle réalise des composants Bluetooth qui intègrent des modules paramétrables et gérant nativement le Bluetooth et des cœurs de microcontrôleur ARM.

Ces composants de la série nRF52 sont utilisés par de nombreux acteurs des objets connectés. Le développement pour ces composants est facilité grâce à un SDK proposé par Nordic Semiconductor.

Dans ce cadre, une fonctionnalité est particulièrement intéressante : la mise à disposition par Nordic Semiconductor d'un service d'émulation UART au travers du Bluetooth. Cette solution permet de transmettre des données via le Bluetooth comme au travers d'un port série. Ce qui a été réalisé précédemment avec le port USB.

Nous pouvons donc facilement transmettre et récupérer des données avec des capteurs connectés à un composant comme le nRF52.

Pour illustrer le fonctionnement des communications Bluetooth, nous allons utiliser un composant nRF52832 sur lequel est connectée une cellule inertielle (accéléromètre, gyroscope, magnétomètre) via un port USB.

Ce composant est configuré pour avoir un port série émulé au travers de la transmission Bluetooth. Ceci permet de communiquer avec le firmware du composant.

Ce firmware est très simple. Il lit les informations provenant du Bluetooth. S'il reçoit une valeur correspondant à un `Start`, les données de la cellule inertielle sont envoyées par le Bluetooth. Une commande `Stop` permet, quant à elle, d'arrêter la transmission des données.

Maintenant passons au script permettant de se connecter, de lire et d'écrire sur le port Bluetooth.

Au préalable, il est nécessaire d'installer la librairie `glib2.0`, puis la librairie `bluepy` en utilisant `pip` :

```
sudo apt-get install libglib2.0-dev
sudo pip3 install bluepy
```

Notez que `bluepy` doit être installé avec les droits du superutilisateur, d'où l'emploi du mot-clé `sudo`.

Pour se connecter à un service, il faut connaître l'adresse du périphérique.

En ce sens, nous scanons les composants disponibles dans le voisinage du Raspberry Pi à l'aide de l'application BLE Scan qui a été installée par le package `bluepy` :

```
pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap4/script$ sudo blescan
Scanning for devices...
Device (new): e1:4b:e8:9f:77:fc (random), -49 dBm
  Complete Local Name: 'nRF52832'
  Flags: <05>
Device (new): 00:e0:4c:f3:54:9f (public), -67 dBm
  Flags: <0a>
  Complete 16b Services: <6666>
  Manufacturer: <ee039f54f34ce000>
  Complete Local Name: 'POLAP200'
  Appearance: <d007>
```

Dans notre cas, les lignes importantes sont :

```
Device (new): e1:4b:e8:9f:77:fc (random), -49 dBm
  Complete Local Name: 'nRF52832'
  Flags: <05>
```

L'adresse MAC du composant est : `e1:4b:e8:9f:77:fc`.

Nous pouvons donc maintenant tester la connexion au composant grâce à la librairie `bluepy` :

```
from bluepy import btle

print("Connexion en cours...")
dev = btle.Peripheral("e1:4b:e8:9f:77:fc", "random")
print("Connexion NRF52832 Ok!")
```

Si la connexion s'est bien passée, le second message s'affiche, car la méthode de connexion est en l'état bloquante. Nous devons donc obtenir le résultat suivant :

```
In [1]: %run ble_read_write.py
Connexion en cours...
Connexion NRF52832 Ok!
```

Nous pouvons également lister les services disponibles grâce au code suivant :

```
from bluepy import btle

print("Connexion en cours...")
dev = btle.Peripheral("e1:4b:e8:9f:77:fc", "random") print("Connexion
NRF52832 Ok!")

print("Services disponibles : ") for
ser in dev.services:
    print(str(ser))
```

Ce qui produit le résultat suivant :

```
In [1]: %run ble_read_write.py
Connexion en cours...
Connexion NRF52832 Ok!
Services disponibles :
Service <uuid=Generic Access handleStart=1 handleEnd=7>
Service <uuid=Generic Attribute handleStart=8 handleEnd=8>
Service <uuid=6e400001-b5a3-f393-e0a9-e50e24dcca9e handleStart=9 handleEnd=65535>
```

C'est cette dernière ligne qui est utile pour notre application, et notamment la chaîne `uuid=6e400001-b5a3-f393-e0a9-e50e24dcca9e`. Cela correspond au service d'émulation RS232 que nous allons utiliser pour envoyer et réceptionner des données.

Cependant, pour manipuler des données, il est nécessaire de passer par les caractéristiques. Le code suivant liste les caractéristiques, du service UART :

```

from bluepy import btle

print("Connexion en cours...")
dev = btle.Peripheral("e1:4b:e8:9f:77:fc", "random")
print("Connexion NRF52832 Ok!")

print("Services disponibles : ")
for ser in dev.services:
    print(str(ser))

print("Caractéristiques disponibles sur service UART : ")
nRF52_uart_uuid = btle.UUID("6e400001-b5a3-f393-e0a9-e50e24dcca9e")

uart_service = dev.getServiceByUUID(nRF52_uart_uuid)

for carac in uart_service.getCharacteristics():
    print(carac)

dev.disconnect()

```

- **Les caractéristiques sont des variables qui permettent d'échanger des informations avec un service bluetooth autant en lecture qu'en écriture.**

Ce qui produit la sortie console suivante :

```

In [5]: %run ble_read_write.py
Connexion en cours...
Connexion NRF52832 Ok!
Services disponibles :
Service <uuid=Generic Access handleStart=1 handleEnd=7>
Service <uuid=Generic Attribute handleStart=8 handleEnd=8>
Service <uuid=6e400001-b5a3-f393-e0a9-e50e24dcca9e handleStart=9 handleEnd=65535>
Caractéristiques disponibles sur service UART :
Characteristic <6e400003-b5a3-f393-e0a9-e50e24dcca9e>
Characteristic <6e400002-b5a3-f393-e0a9-e50e24dcca9e>

```

Les deux dernières lignes correspondent respectivement aux caractéristiques à employer pour la transmission de données (6e400003-b5a3-f393-e0a9-e50e24dcca9e) et la réception de données (6e400002-b5a3-f393-e0a9-e50e24dcca9e).

Il est donc maintenant possible d'envoyer et de lire les données disponibles à l'aide de ces caractéristiques.

Lecture et écriture

Pour l'écriture :

```

uuidTx = btle.UUID("6e400003-b5a3-f393-e0a9-e50e24dcca9e")
Tx_Data = uart_service.getCharacteristics(uuidTx)[0]
Tx_Data.write(bytes(0xFF))

```

Pour la lecture :

```
uuidRx = btle.UUID("6e400002-b5a2-f393-e0a9-e50e24dcca9e")
Rx_Data = uart_service.getCharacteristics(uuidRx)[0]
val = Rx_Data.read()
```

Ce qui nous permet d'envoyer un octet de start 0xFF, puis de lire les données dans le buffer toutes les secondes pendant 5 secondes :

```
import time
import binascii
from bluepy import btle

print("Connexion en cours...")
dev = btle.Peripheral("e1:4b:e8:9f:77:fc", "random")
print("Connexion NRF52832 Ok!")

print("Services disponibles : ")
for ser in dev.services:
    print(str(ser))

print("Caractéristiques disponibles sur service UART : ")
nRF52_uart_uuid = btle.UUID("6e400001-b5a3-f393-e0a9-e50e24dcca9e")

uart_service = dev.getServiceByUUID(nRF52_uart_uuid)

for carac in uart_service.getCharacteristics():
    print(carac)

uuidTx = btle.UUID("6e400003-b5a3-f393-e0a9-e50e24dcca9e")
Tx_Data = uart_service.getCharacteristics(uuidTx)[0]
Tx_Data.write(bytes(0xFF))

time.sleep(2.0)

uuidRx = btle.UUID("6e400002-b5a2-f393-e0a9-e50e24dcca9e")
Rx_Data = uart_service.getCharacteristics(uuidRx)[0]

count = 0
while count < 5:
    val = Rx_Data.read()
    print("Donnée Lues : ", binascii.b2a_hex(val))
    count += 1
    time.sleep(1)

dev.disconnect()
```

Avec le firmware dans le composant NRF52, nous obtenons :

```
In [8]: %run ble_read_write.py
Connexion en cours...
Connexion NRF52832 Ok!
Services disponibles :
Service <uuid=Generic Access handleStart=1 handleEnd=7>
Service <uuid=Generic Attribute handleStart=8 handleEnd=8>
Service <uuid=6e400001-b5a3-f393-e0a9-e50e24dcca9e handleStart=9 handleEnd=65535>
Caractéristiques disponibles sur service UART :
Characteristic <6e400003-b5a3-f393-e0a9-e50e24dcca9e>
Characteristic <6e400002-b5a3-f393-e0a9-e50e24dcca9e>
Donnée Lues : b'bf'
Donnée Lues : b'bf'
Donnée Lues : b'bf'
Donnée Lues : b'bf'
Donnée Lues : b'bf'
```

Fonctionnement par notifications

La lecture proposée plus haut n'est pas la meilleure façon pour récupérer des informations provenant d'un service Bluetooth.

En effet, dans la majorité des cas, l'utilisateur n'a pas à lire les informations sur la caractéristique, c'est le service qui va transmettre directement les données (au travers de la caractéristique) à l'application tierce. On parle de service par notification. L'application tierce sera notifiée d'une donnée disponible et n'aura plus qu'à la lire.

En pratique, pour fonctionner sous cette forme, il est nécessaire de s'inscrire aux notifications en écrivant une chaîne de caractère spéciale «\0x01\0x00» sur la caractéristique qui permet de recevoir des données. Dans notre cas, pour recevoir des données provenant du service uart, il est nécessaire d'écrire cette ligne :

```
dev.writeCharacteristic(Rx_Data.getHandle()+1, b'\x01\x00')
```

En plus de cette configuration, il est nécessaire de créer un service délégué. Ceci se fait en trois temps :

- ♦ D'une part, la mise en place d'un service délégué pour lire les notifications provenant du service Bluetooth. Sans rentrer dans le détail, on approximera qu'un service délégué a le même comportement qu'un callback, à la différence près que sa mise en place s'effectue en dérivant une classe fille d'une classe contenue dans le module btle. En pratique la mise en place de ce service s'effectue en plaçant le code suivant juste après la connexion aux composants Bluetooth :

```
dev.setDelegate(MyDelegate())
```

Dans un second temps, il faut créer la classe dérivée. Cette classe possède un constructeur qui appelle explicitement le constructeur de la classe mère et une méthode `handleNotification()` qui sera appelée avec réception de notification. C'est donc dans cette méthode qu'il faut mettre le code d'un éventuel traitement. Ici seul un affichage est réalisé :

```
class MyDelegate(btle.DefaultDelegate) :
```

```
    def __init__(self):
        btle.DefaultDelegate.__init__(self)

    def handleNotification(self, cHandle, data):
        print('données lues : ', data)
        print('longueur chaine : ', len(data))
        print('type data : ', type(data))
```

Enfin dans la boucle principale. Il est nécessaire d'appeler dans une boucle infinie, par exemple, la méthode (`waitForNotifications()`) qui met l'objet `btle` en attente de notification. Cette méthode en paramètre la valeur de timeout, c'est-à-dire le temps pendant lequel on va attendre et les notifications et sortir de la méthode qui est bloquante :

```
dev.waitForNotifications(0.1)
```

Le code complet de l'application est disponible ci-dessous :

```
import time
import signal
import sys
from bluepy import btle

class MyDelegate(btle.DefaultDelegate):
    def __init__(self):
        btle.DefaultDelegate.__init__(self)

    def handleNotification(self, cHandle, data):
        print('données lues : ', data)
        print('longueur chaine : ', len(data))
        print('type data : ', type(data))

if __name__ == '__main__':

    signal.signal(signal.SIGINT, reaction_intercept)

    print("Connection en cours...")
    dev = btle.Peripheral("C9:9A:84:D5:6D:72", "random")
    print("Connection NRF52832 Ok!")
    dev.setDelegate(MyDelegate())

    print("Services disponibles : ")
    for ser in dev.services:
        print(str(ser))
```

```

print("Caratérisitiques disponibles sur service UART : ")
nRF52_uart_uuid = btle.UUID("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")
uart_service = dev.getServiceByUUID(nRF52_uart_uuid)
for carac in uart_service.getCharacteristics():
    print(carac)

uuidTx = btle.UUID("6e400002-b5a3-f393-e0a9-e50e24dcca9e")
Tx_Data = uart_service.getCharacteristics(uuidTx)[0]

uuidRx = btle.UUID("6e400003-b5a3-f393-e0a9-e50e24dcca9e")
Rx_Data = uart_service.getCharacteristics(uuidRx)[0]
# inscription pour fonctionnement avec notifications
dev.writeCharacteristic(Rx_Data.getHandle()+1, b'\x01\x00')

buf = bytearray(5)
buf[0] = 0x01
buf[1] = 0x02
buf[2] = 0x03
buf[3] = 0x04
buf[4] = 0xa5

Tx_Data.write(buf)
#
#
print("Buf envoyé : ", buf)
#
status = True
# count = 0
while status:
    if dev.waitForNotifications(0.1):
        # appelle de la fonction déléguée
        continue

```


13. Lire et écrire dans des fichiers

Écrire dans un fichier peut être utile pour plusieurs raisons. Par exemple, pour y placer les résultats d'une tâche particulière : un score, des données issues d'un calcul... L'écriture dans un fichier est très utile lorsque l'on traite une grande quantité de données à déboguer. Une fois les données exportées dans un fichier, il est possible d'en faire l'analyse a posteriori. De cette manière, on peut affiner les conditions de réalisation d'un bogue et finalement créer un test qui permet de reproduire lesdites conditions.

Mais avant d'effectuer ces manipulations, il est nécessaire d'ouvrir un fichier.

Ouvrir et fermer un fichier

Ouvrir un fichier ou toute manipulation de fichier en langage Python est assez aisé. Il n'est pas nécessaire d'importer un quelconque package ou module pour réaliser cette tâche, il suffit d'écrire l'instruction suivante :

```
<mon_fichier> = open(<nom_fichier_a_ouvrir>,<mode>)
```

La méthode `open()` retourne un objet (`<mon_fichier>`) pour manipuler le fichier nommé `<nom_fichier_a_ouvrir>` selon le mode de fonctionnement précisé à l'aide de `<mode>`.

`<nom_fichier_a_ouvrir>` est une chaîne de caractères qui comporte l'adresse du fichier sur le disque.

Les modes précisés par `<mode>` peuvent être :

- `'w'` ouverture en mode écriture
- `'r'` ouverture en mode lecture
- `'a'` ajout des données en fin de fichier quand le fichier est ouvert en mode écriture
- `'b'` pour préciser que le fichier manipulé est un fichier binaire

Il existe d'autres modes, mais les modes précédemment cités sont les plus usités. Notez que les modes sont cumulables. Par exemple, le mode `wab` signifie que le fichier est ouvert en mode écriture, que les données sont placées en fin de fichier et que le fichier manipulé est un fichier binaire.

La tâche complémentaire à l'ouverture est la fermeture. La fermeture se réalise grâce à l'appel de la méthode `close()` de l'objet créé lors de l'ouverture :

```
<mon_fichier>.close()
```

Avant de présenter un exemple, quelques recommandations :

- Il est nécessaire de s'assurer que le chemin du fichier est exact et qu'il est précisé sans adresse relative. C'est-à-dire qu'il faut préciser le chemin `/home/documents/pi/.../mon_fichier.txt`, et non `~/.../mon_fichier.txt`. Pour que le fichier soit créé dans le même dossier que le script, il suffit de préciser un nom de fichier ; aucune adresse n'est nécessaire, car Python place le fichier dans le répertoire courant.
- Il est nécessaire de tester si l'ouverture s'est bien passée avant de faire des manipulations qui crasheraient le programme. Pour cela, il faut associer les instructions `with... as...` à la méthode `open()` de la façon suivante :

```
with open(<nom_fichier_a_ouvrir>,<mode>) as <mon_fichier> :  
    <instruction 1>  
    <instruction 2>  
    ...  
    <instruction n>
```

De cette manière, les instructions du bloc indenté ne sont effectuées que si le fichier a pu être ouvert.

Par conséquent, le code suivant :

```
with open('test_ouverture_fermeture.txt', 'w') as mon_fichier:
    print('ouverture et création fichier OK!')
    mon_fichier.close()
```

donne le résultat suivant :

```
In [3]: %run ouverture_fermeture.py
ouverture et création fichier OK!
```

Un fichier nommé test_ouverture_fermeture.txt est créé sur le disque au même emplacement que le script de test.

Écriture

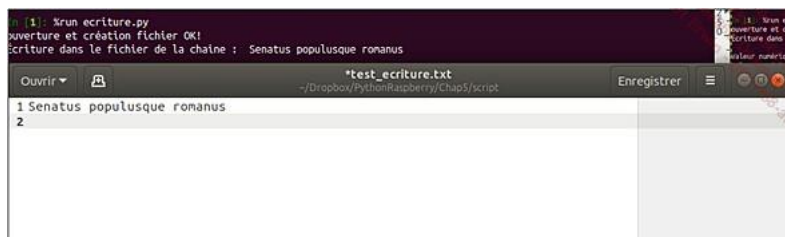
L'écriture est extrêmement simple et passe par l'appel de la méthode `write()` par l'objet créé à l'ouverture du fichier. L'exemple qui suit permet d'écrire la chaîne « Senatus populusque romanus » :

```
with open('test_ecriture.txt', 'w') as mon_fichier:
    print('ouverture et création fichier OK!')
    chaine_ecrire = "Senatus populusque romanus\n"
    mon_fichier.write(chaine_ecrire)
    print("Écriture dans le fichier de la chaîne : ", chaine_ecrire)
    mon_fichier.close()
```

L'exécution de ce script donne le résultat suivant :

```
In [5]: %run ecriture.py
ouverture et création fichier OK!
Écriture dans le fichier de la chaîne : Senatus populusque romanus
```

La capture d'écran suivante montre le contenu dans le fichier créé test_ecriture.txt.



Le fichier étant ouvert en mode texte, par opposition au mode binaire qui sera développé ci-après, il n'est possible d'écrire que des chaînes de caractères. En effet, si des valeurs numériques sont passées en paramètre de la manière suivante :

```
mon_fichier.write(2)
```

alors, une erreur est générée par l'interpréteur :

```

-----
TypeError                                Traceback (most recent call last)
~/Documents/Redaction/PythonRaspberry/Chap5/script/ecriture.py in
<module>()
      4     mon_fichier.write(chaine_ecrire)
      5     print("Écriture dans le fichier de la chaîne : ", chaine_ecrire)
----> 6     mon_fichier.write(2)
      7     mon_fichier.close()

```

TypeError: write() argument must be str, not int

Pour réaliser ce genre d'écriture, il est nécessaire de convertir la valeur à l'aide de la fonction `str()` :

```
mon_fichier.write(str(2))
```

À présent, voyons comment ajouter au fichier le code nécessaire à l'écriture d'un tableau de 10 valeurs numériques générées aléatoirement. Ces valeurs seront suivies d'un renvoi à la ligne, il ne doit y avoir qu'une valeur par ligne.

Les valeurs aléatoires sont générées à l'aide de la fonction `randint()`, contenue dans la librairie `random` qui doit être importée.

```

from random import randint
with open('test_ecriture.txt', 'w') as mon_fichier:
    print('ouverture et création fichier OK!')
    chaine_ecrire = "Senatus populusque romanus\n"
    mon_fichier.write(chaine_ecrire)
    print("Écriture dans le fichier de la chaîne : ", chaine_ecrire)

    for i in range(10):
        val = randint(0, 100)
        mon_fichier.write(str(val)+'\n')
        print("Valeur numérique ajoutée au fichier : " + str(val))
    mon_fichier.close()

```

Ce qui produit la sortie console suivante :

```

In [12]: %run ecriture.py
ouverture et création fichier OK!
Écriture dans le fichier de la chaîne : Senatus populusque romanus

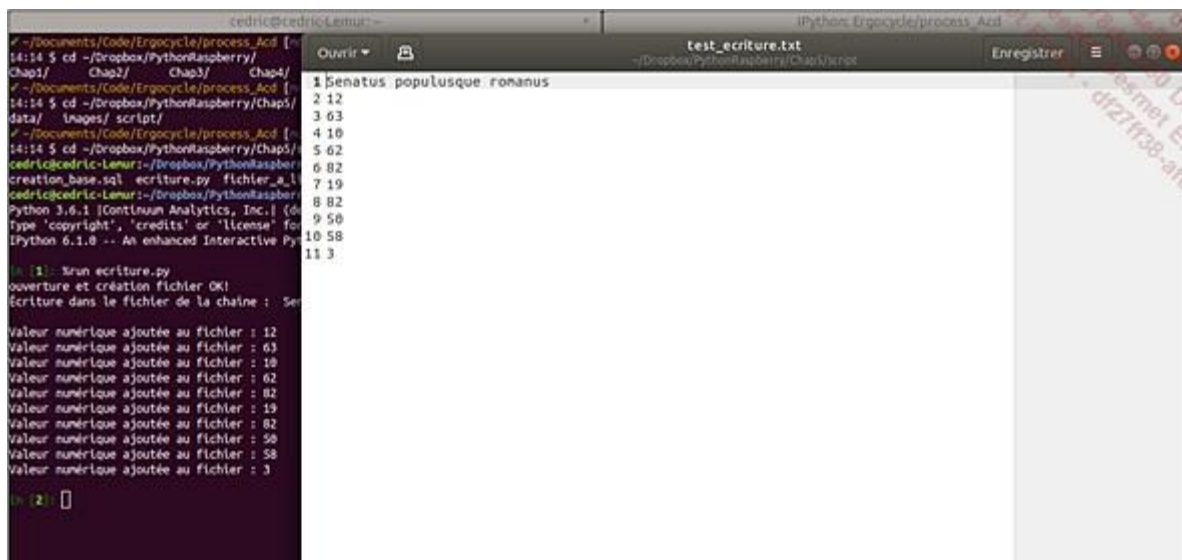
```

```

Valeur numérique ajoutée au fichier : 60
Valeur numérique ajoutée au fichier : 84
Valeur numérique ajoutée au fichier : 75
Valeur numérique ajoutée au fichier : 39
Valeur numérique ajoutée au fichier : 99
Valeur numérique ajoutée au fichier : 54
Valeur numérique ajoutée au fichier : 19
Valeur numérique ajoutée au fichier : 89
Valeur numérique ajoutée au fichier : 27
Valeur numérique ajoutée au fichier : 45

```

La capture d'écran suivante montre le contenu du fichier après écriture :



Lecture

La lecture fonctionne exactement sur le même principe. Il y a une condition nécessaire à l'exécution correcte d'un script de lecture : le fichier que l'on veut lire doit exister. Cela peut paraître bête, mais c'est une faute de débutant qui peut vous faire perdre pas mal de temps !

À titre d'exemple, nous allons réaliser la lecture du fichier précédemment conçu et qui sera renommé fichier a lire.text.

Afin d'effectuer la lecture, nous utilisons l'instruction `readline()` qui retourne le contenu d'une ligne. Pour lire la première ligne du fichier `fichier_a_lire.txt`, il est nécessaire d'adapter le code précédent de la manière suivante :

```
with open('fichier_a_lire.txt', 'r') as mon_fichier:
    print('ouverture fichier OK!')
    chaine_lue = mon_fichier.readline()
    print("Chaîne lue : ", chaine_lue)
    mon_fichier.close()
```

La sortie console permet de vérifier que nous avons bien la première ligne du fichier qui contient : « Senatus populusque romanus ».

```
In [15]: %run lecture.py
ouverture fichier OK!
Chaîne lue : Senatus populusque romanus
```

- **Nous utilisons, ici, la méthode `readline()` qui permet de lire ligne par ligne, et non la fonction `read()` qui permet de lire tout le fichier en une seule fois. L'utilisation de `read()` est possible, mais dans ce cas, sous-optimale, car elle nécessite un très gros traitement de la chaîne de caractères qui contient toutes les valeurs du fichier.**

Afin de relire les valeurs numériques, il est nécessaire de faire une boucle pour parcourir chaque valeur puis la convertir en entier avec `int()`.

Pour réaliser cette boucle, il y a deux solutions. Si nous connaissons la quantité de données à lire, nous utilisons une boucle `for` :

```
with open('fichier_a_lire.txt', 'r') as mon_fichier:
    print('ouverture fichier OK!')
    chaine_lue = mon_fichier.readline()
    print("Chaîne lue : ", chaine_lue)

    for i in range(10):
        val_str = mon_fichier.readline()
        val_int = int(val_str)

        print("Entier lu : ", str(val_int))
    mon_fichier.close()
```

Sinon, nous recourons à une boucle `while` :

```
with open('fichier_a_lire.txt', 'r') as mon_fichier:
    print('ouverture fichier OK!')
    chaine_lue = mon_fichier.readline()
    print("Chaîne lue : ", chaine_lue)

    while True:
        val_str = mon_fichier.readline()
        if not val_str:
            break
        val_int = int(val_str)

        print("Entier lu : ", str(val_int))
    mon_fichier.close()
```

Le principe est un peu plus complexe. Nous créons une boucle infinie (ce qui est un peu dangereux) avec `while True`, mais à chaque itération une ligne est lue et mise dans une instance de l'objet `string` nommée `val_str`. S'il n'y a rien à lire, alors l'objet n'est pas créé. Le test conditionnel qui suit permet, s'il n'y a plus de ligne, de casser la boucle `while` à l'aide de l'instruction `break`.

Dans les deux cas, le résultat obtenu est le suivant :

```
In [1]: %run lecture.py
ouverture fichier OK!
Chaine lue : Senatus populusque romanus
```

```
Entier lu : 60
Entier lu : 84
Entier lu : 75
Entier lu : 39
Entier lu : 99
Entier lu : 54
Entier lu : 19
Entier lu : 89
Entier lu : 27
Entier lu : 45
```

- **La lecture d'un fichier nécessite d'avoir le mode d'emploi pour le lire. Il est donc important que le développeur qui a créé le fichier vous donne la méthode pour lire à minima sa structure. En effet, aucune méthode magique de lecture n'existe, il faut l'adapter au fichier que vous avez sous les yeux.**

Cas des fichiers binaires

Les manipulations effectuées jusqu'ici sont relatives à un fichier directement lisible par le système Raspbian (c'est également vrai pour d'autres systèmes d'exploitation).

Or, cette façon de faire nécessite à chaque fois une conversion (qui est transparente pour vous) entre les valeurs à écrire et les valeurs dans le fichier. De plus, des valeurs sont ajoutées, comme les fins de ligne et les retours chariot. Des valeurs d'en-tête peuvent également être ajoutées.

Dans le cas où vous n'avez pas besoin de lire un fichier en clair avec un éditeur de texte et que vous voulez réduire au maximum la taille des données contenues dans le fichier ainsi que le temps de manipulation de ces données (lecture et écriture), alors le format binaire est le plus adapté.

Pour utiliser ce format, il faut indiquer lors de l'ouverture la valeur 'b' à l'argument permettant de préciser le mode.

```
with open('test_ecriture.txt', 'wb') as mon_fichier:
```

Pour lire et écrire, nous disposons des méthodes `read()` (et non `readline()`) et `write()`. Notez qu'il est nécessaire, lors de l'utilisation de `read()`, de préciser le nombre d'octets à lire.

Encore une fois, il est très important d'avoir des informations sur les données à lire : en l'occurrence, le nombre de données, et à la différence du format texte, le type de données afin de connaître le nombre d'octets à lire.

Dans un premier temps, réalisons l'écriture dans un fichier binaire de 10 valeurs de type `int` aléatoires :

```
from struct import pack
from struct import unpack
from random import randint

with open('test_ecriture.bin', 'wb') as mon_fichier:
    print('ouverture et création fichier en mode binaire : OK!')

    for i in range(10):
        val = randint(0, 100)
        mon_fichier.write(pack(">I", val))
        print("Valeur numérique ajoutée au fichier : " + str(val))
```

Examinons la ligne suivante :

Cette ligne signifie que nous allons écrire des données. La fonction `pack()` transforme `val` en une valeur entière (précisée grâce à l'argument `'>I'`) en un objet de type `bytes` qui peut être écrit par la fonction `write()`.

```
mon_fichier.write(pack(">I", val))
```

Ce script crée la sortie console suivante :

```
In [59]: %run lecture_ecriture_binaire.py
ouverture et création fichier en mode binaire : OK!
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 35
Valeur numérique ajoutée au fichier : 30
Valeur numérique ajoutée au fichier : 68
Valeur numérique ajoutée au fichier : 42
Valeur numérique ajoutée au fichier : 86
Valeur numérique ajoutée au fichier : 48
Valeur numérique ajoutée au fichier : 46
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 94
```

Nous pouvons vérifier que le fichier est bien créé en listant le répertoire du script à l'aide de la commande `ls -la`.

Ce qui produit :

```
pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap5/script$ ls -la
total 36
drwxr-xr-x 3 pi pi 4096 mars  4 18:56 .
drwxr-xr-x 4 pi pi 4096 mars  4 18:48 ..
-rw-r--r-- 1 pi pi  475 mars  4 15:57     ecriture.py
-rw-r--r-- 1 pi pi   57 mars  4 16:12     fichier_a_lire.txt
-rw-r--r-- 1 pi pi  642 mars  4 18:56 lecture_ecriture_binaire.py
-rw-r--r-- 1 pi pi  367 mars  4 17:34     lecture.py
-rw-r--r-- 1 pi pi  135 mars  4 15:14 ouverture_fermeture.py
drwxr-xr-x 2 pi pi 4096 mars  4 15:07     .ropeproject
-rw-r--r-- 1 pi pi   40 mars  4 18:44     test_ecriture.bin
-rw-r--r-- 1 pi pi    0 mars  4 18:23     test_ecriture.txt
```

```
-rw-r--r-- 1 pi pi    0 mars  4 15:10 test_ouverture_fermeture.txt
```

Il apparaît que le fichier `test_ecriture.bin` créé par le script fait 40 octets, ce qui est conforme à nos attentes : 10 `int` soit 10 x 4 octets, d'où les 40 octets disponibles sur le disque.

Nous pouvons vérifier le contenu du fichier à l'aide de la commande : `hexdump test_ecriture.bin`.

Ce qui produit :

```
pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap5/script$
hexdump test_ecriture.bin
00000000 0000 5d00 0000 2300 0000 1e00 0000 4400
00000010 0000 2a00 0000 5600 0000 3000 0000 2e00
00000020 0000 5d00 0000 5e00
000000
```

Ces valeurs sont codées en hexadécimal. La première valeur est `0000 5d00`. Cette valeur est notée selon le formalisme big endian. Cette convention signifie que chaque lot de 2 octets est codé avec l'octet précisant les valeurs de la partie basse en premier. Pour la compréhension humaine, il faut transformer `0000 5d00` en `0000 005d`. C'est cette dernière valeur qu'il faut convertir en décimale, (à l'aide de la calculatrice de Raspbian), ce qui donne 93, qui est la valeur que nous devons écrire en premier dans le fichier.

Lecture

La lecture s'effectue exactement selon le même principe.

```
with open('test_ecriture.bin', 'rb') as mon_fichier:
    print('ouverture fichier en mode binaire : OK!')
    for i in range(10):
        val = mon_fichier.read(4)
        val_int = unpack(">i", val)
        print("Valeur numérique ajoutée au fichier : " + str(val_int[0]))
```

Nous lisons 4 octets à l'aide de la fonction `read()` :

```
val = mon_fichier.read(4)
```

Nous dépaquetons pour les reconvertir en valeurs entières, c'est à dire que nous transformons une séquence d'octet en éléments éléments individuels dont nous préciserons le codage :

```
val_int = unpack(">i", val)
```

Pour l'affichage, nous utilisons l'instruction `str(val_int[0])`, c'est-à-dire la conversion du premier élément de `val_int`, car la sortie de `unpack()` est de type `tuple`.

Le code qui réalise successivement l'écriture et la lecture du fichier `test_ecriture.bin` est :


```

from struct import pack
from struct import unpack
from random import randint

with open('test_ecriture.bin', 'wb') as mon_fichier:
    print('ouverture et création fichier en mode binaire : OK!')
    chaîne_ecrire = "Senatus populusque romanus\n"
    for i in range(10):
        val = randint(0, 100)
        mon_fichier.write(pack(">I", val))
        print("Valeur numérique ajoutée au fichier : " + str(val))

with open('test_ecriture.bin', 'rb') as mon_fichier:
    print('ouverture fichier en mode binaire : OK!')
    for i in range(10):
        val = mon_fichier.read(4)
        val_int = unpack(">i", val)
        print("Valeur numérique ajoutée au fichier : " + str(val_int[0]))

```

Il produit le résultat suivant :

```

In [59]: %run lecture_ecriture_binaire.py
ouverture et création fichier en mode binaire : OK!
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 35
Valeur numérique ajoutée au fichier : 30

Valeur numérique ajoutée au fichier : 68
Valeur numérique ajoutée au fichier : 42
Valeur numérique ajoutée au fichier : 86
Valeur numérique ajoutée au fichier : 48
Valeur numérique ajoutée au fichier : 46
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 94
ouverture fichier en mode binaire : OK!
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 35
Valeur numérique ajoutée au fichier : 30
Valeur numérique ajoutée au fichier : 68
Valeur numérique ajoutée au fichier : 42
Valeur numérique ajoutée au fichier : 86
Valeur numérique ajoutée au fichier : 48
Valeur numérique ajoutée au fichier : 46
Valeur numérique ajoutée au fichier : 93
Valeur numérique ajoutée au fichier : 94

```

14. Utiliser des CSV

Outre le format texte et le format binaire, qui permettent d'enregistrer et plus généralement de manipuler des données, un format s'est imposé pour stocker des données : le CSV.

Le formalisme du CSV est très simple, ce qui permet à un développeur de créer un fichier à partir de quasiment n'importe quel langage, sans librairie particulière.

Le CSV est surtout intéressant pour stocker plusieurs échantillons d'un phénomène (ces phénomènes peuvent être par exemple physique : plusieurs mesures de températures ou des données par exemple age...). C'est typiquement le genre d'informations qui sont stockés dans un tableur. Le format CSV est d'ailleurs facilement importable et exportable dans un tableur, tel LibreOffice Calc disponible sur un système Raspbian.

Un CSV contient donc des données d'un tableau en deux dimensions représentées sous forme de lignes.

Prenons les données concernant les passagers du Titanic, disponibles sur un dépôt GitHub à l'adresse suivante :

<https://github.com/mwaskom/seaborn-data/blob/master/titanic.csv>

La première ligne du fichier contient le nom des champs décrivant les données. Ces éléments sont séparés par des virgules :

```
survived,pclass,sex,age,sibsp,parch,fare,embarked,class,  
who,adult_male,deck,embark_town,alive,alone
```

Les données à proprement parler occupent les lignes suivantes dans le fichier. Chaque ligne représente un échantillon dont les caractéristiques répondent à la description de la première ligne. Les caractéristiques sont données dans le même ordre et séparées par des virgules.

Ainsi, le premier élément, c'est-à-dire le premier passager, du fichier est :

```
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False
```

Le deuxième est :

```
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes,False...
```

Voyons maintenant les méthodes nécessaires à la manipulation des données.

La première étape consiste à ouvrir le fichier :

```
with open('../data/titanic.csv', 'r') as fichier_csv:
```

La deuxième étape consiste à lire le contenu du fichier à l'aide de l'objet CSV que nous avons importé au préalable :

```
data_titanic = csv.reader(fichier_csv, delimiter=',')
```

Le premier argument est l'objet créé lors de l'ouverture du fichier, le second est le type de caractère qui sert à délimiter les données ; ici, des virgules.

L'objet retourné par cette lecture est une liste qui correspond à chaque ligne du fichier.

Chaque élément d'une liste est également une liste des valeurs.

Il est possible d'accéder aux valeurs en imbriquant deux boucles :

```
for ligne in data_titanic:
    val_aff = ''

    for val in ligne:
        val_aff += (val + ',')
    print(val_aff)
```

Le code final pour afficher toutes les données est donc :

```
import csv
with open('../data/titanic.csv', 'r') as fichier_csv:
    data_titanic = csv.reader(fichier_csv, delimiter=',')
    for ligne in data_titanic:
        val_aff = ''
        for val in ligne:
            val_aff += (val + ',')
        print(val_aff)
```

Il produit le résultat suivant :

```
%run manip_csv.py
survived,pclass,sex,age,sibsp,parch,fare,embarked,class,who,adult_male,deck
embark_town,alive,alone,
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False,
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes,False,
1,3,female,26.0,0,0,7.925,S,Third,woman,False,,Southampton,yes,True,
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,False,
0,3,male,35.0,0,0,8.05,S,Third,man,True,,Southampton,no,True,
...
```

➤ **Seules les six premières lignes ont été affichées ici. Le reste a été tronqué pour plus de lisibilité.**

Notez que la première ligne fournit le nom des champs. Lors d'un éventuel traitement, il sera nécessaire d'exclure ces valeurs.

L'écriture dans le fichier est assez comparable à la lecture.

Pour cela, ouvrons en mode écriture ('w') un nouveau fichier que nous nommons titanic_edit.csv :

```
with open('../data/titanic_edit.csv', 'w') as fichier_csv:
```

Puis, créons un objet permettant d'écrire dans un fichier CSV :

```
data_edit = csv.writer(fichier_csv, delimiter=',')
```

Pour écrire une ligne en entier, nous utilisons la méthode :

```
data_edit.writerow(nouvelle_ligne)
```

Adaptons le code précédent pour récupérer la première ligne du fichier titanic.csv qui contient le nom des champs. Cette ligne sera placée dans un nouveau fichier titanic_edit.csv. Une nouvelle ligne sera également ajoutée à ce

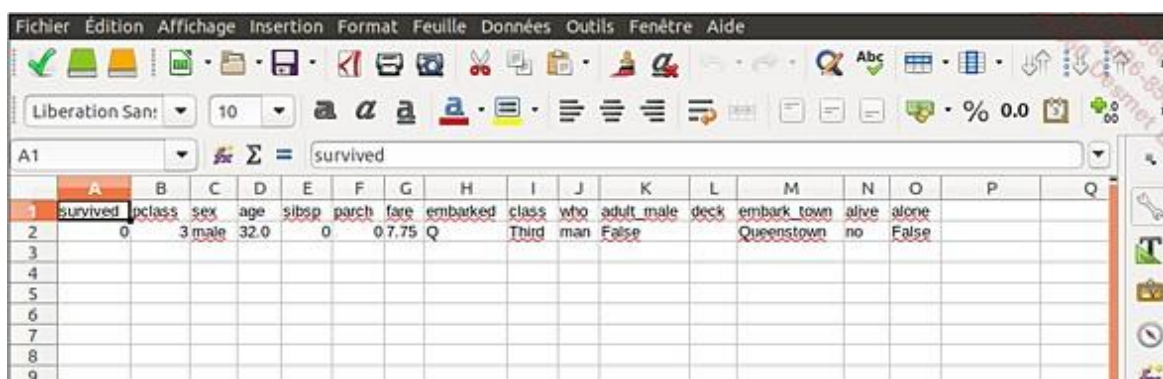
fichier.

```
import csv
with open('../data/titanic.csv', 'r') as fichier_csv:
    data_titanic = csv.reader(fichier_csv, delimiter=',')
    for i, ligne in enumerate(data_titanic):
        if i == 0:
            premiere_ligne = ligne
            val_aff = ''
        for val in ligne:
            val_aff += (val + ',')
        print(val_aff)

with open('../data/titanic_edit.csv', 'w') as fichier_csv:
    data_edit = csv.writer(fichier_csv, delimiter=',')
    nouvelle_ligne = [0, 3, "male", 32.0, 0, 0, 7.75,
                      "Q", "Third", "man", "False", "",
                      "Queenstown", "no", "False"]

    data_edit.writerow(premiere_ligne)
    data_edit.writerow(nouvelle_ligne)
```

L'impression d'écran qui suit montre les données ouvertes au sein du tableur LibreOffice Calc.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult	male	deck	embark town	alive	alone	
2	0	3	male	32.0	0	0	7.75	Q	Third	man	False		Queenstown	no	False		
3																	
4																	
5																	
6																	
7																	
8																	
9																	

15. Programmation multithreading et calcul parallèle

La grande majorité des processeurs sont multithreading, Ils possèdent même dans la plupart des cas plusieurs cœurs, on parle alors de processeur multicœur. C'est le cas du Raspberry Pi 3 B+ qui possède quatre cœurs.

Cette évolution des processeurs aboutissant à des capacités de calculs multitâches importantes est le fruit de l'évolution du matériel informatique. En effet, dans le milieu des années 2000, la montée en fréquence des processeurs s'est ralentie, voire stoppée pour des raisons purement technologiques. Il était devenu très difficile de concevoir des processeurs toujours plus rapides avec des technologies permettant de proposer des prix intéressants pour le grand public.

La solution trouvée par les acteurs du secteur a été de paralléliser le calcul en ajoutant la possibilité d'exécuter toujours plus de tâches en parallèle, soit en utilisant une taille de pipeline plus importante, soit, plus récemment, en utilisant plusieurs cœurs au sein d'un processeur. Ces cœurs pouvant communiquer les uns avec les autres, bien évidemment.

En Python, il existe deux façons d'exploiter l'architecture multitâche du processeur. La première, en exécutant des tâches en parallèle grâce aux threads. La seconde en utilisant une librairie qui envoie sur chaque processeur ou cœur (selon l'architecture) une partie des calculs.

Utiliser les threads

La création d'un thread (une tâche) en Python est assez concise. Il y a minima trois étapes.

- l'importation du module nécessaire :

```
from threading import Thread
```

- la création d'un thread :

```
t = Thread(target=worker, args=(i,))
```

- le démarrage d'un thread :

```
t.start()
```

Voici un exemple complet :

```
import random
import time
from threading import Thread

def worker(i):
    temps = random.randrange(5)
    print("Le thread {} dormira {} sec".format(i, temps))
    time.sleep(temps)
    print("Le thread {} a fini de dormir".format(i))

for i in range(10):
    t = Thread(target=worker, args=(i,))
    t.start()
```

Ici, nous démarrons 10 threads séquentiellement. Au sein de chaque thread, nous générons un nombre aléatoire compris entre 0 et 5 :

```
temps = random.randrange(5)
```

Cette valeur permet de réaliser une pause d'une durée égale à la valeur générée :

```
time.sleep(temps)
```

Le reste du code affiche simplement des messages avant et après la pause.

Le résultat obtenu par l'exécution du script est :

```
In [9]: %run threading.py
Le thread 0 dormira 1 sec
Le thread 1 dormira 4 sec
Le thread 2 dormira 2 sec
Le thread 3 dormira 1 sec
Le thread 4 dormira 4 sec
Le thread 5 dormira 2 sec
Le thread 6 dormira 2 sec
Le thread 7 dormira 1 sec
Le thread 8 dormira 2 sec
Le thread 9 dormira 3 sec
Le thread 0 a fini d dormir
Le thread 3 a fini d dormir
Le thread 7 a fini d dormir
Le thread 2 a fini d dormir
Le thread 5 a fini d dormir
Le thread 6 a fini d dormir
Le thread 8 a fini d dormir
Le thread 9 a fini d dormir
Le thread 1 a fini d dormir
Le thread 4 a fini d dormir
```

Chaque thread est créé indépendamment et démarré séquentiellement. Le résultat obtenu permet de voir que chaque élément n'est pas bloquant, car chaque thread répond de manière indépendante des autres.

Nous pouvons donc, de cette manière simple, lancer des tâches concurrentes sur le système d'exploitation.

- **Il est très probable que vous ayez besoin d'accéder aux mêmes données avec deux threads différents. Ce cas de figure doit être géré, car vous devez accéder aux données au bon moment, celles-ci pouvant être modifiées par un thread. On parle alors de synchronisation. Pour réaliser cette synchronisation, il est nécessaire de bloquer l'accès à certaines données à l'aide d'un objet `Rlock` ().**

Calcul multiprocessing

Dans certains cas, l'accélération matérielle consiste à exécuter le même programme sur des données différentes. On parle généralement de parallélisme de données.

Dans ce cas de figure, il existe un module particulièrement intéressant pour réaliser l'exécution de tâches similaires en parallèle : multiprocessing. Son importation se réalise généralement de la manière suivante :

```
import multiprocessing as mp
```

Créons une liste de processus à exécuter :

```
processes = [mp.Process(target=chaine_aleatoire, args=(5, sortie))  
              for x in range(100)]
```

Cette liste fonctionne sur le principe des callbacks. Nous donnons le nom de la fonction à exécuter : `chaine_aleatoire`. Nous précisons avec le tuple `args=(5, sortie)` les arguments de la fonction `chaine_aleatoire()`.

Il ne reste plus qu'à démarrer le traitement des données pour l'ensemble des processus :

```
for p in processes:  
    p.start()
```

Le code qui suit illustre l'utilisation de la librairie multiprocessing pour une pile de 100 processus :

```
import multiprocessing as mp
import random
import string

random.seed(123)

sortie = mp.Queue()

def chaine_aleatoire(taille, sortie):
    s_aleatoire = ''.join(random.choice(
        string.ascii_lowercase +
        string.ascii_uppercase +
        string.digits)
        for i in range(taille))
    sortie.put(s_aleatoire)

processes = [mp.Process(target=chaine_aleatoire, args=(5, sortie))
              for x in range(100)]

for p in processes:
    p.start()

for p in processes:
    p.join()

res = [sortie.get() for p in processes]

print(res)
```

Ce qui produit en sortie le résultat suivant :

```
In [10]: %run parallel_multiprocessing.py
['LYkHl', '3czTn', 'YsfR0', 'tgNSG', 'dkV92', 'dSuAK',
'Fs5lA', 'y25rg', 'F2nDl', 'MduRd', 'NeDMp', 'f24tU', '3Rymz',
'IFARd', 'pbT8w', 'secQz', '2oSnN', 'HoYoa', 'a8W6j', 'lFwxW',
'Id3BA', 'UBfBK', 'vzw2G', 'qmSYx', 'tFFyn', 'r0NaZ', 'KRHFS',
'zI3n9', '0t4uE', 'HshlM', 'DXDY0', 'RP0Z7', 'lZD4I', 'nChIL',
'6xyGI', 'AIsE0', 'e4YxF', 'o3WRd', 'YNOVE', 'DlJvr', 'e8zar',
'0eDHz', 'T2Vj2', 'Pk2wL', 'lSOXE', '88rMU', 'ruOuC', 'Ggazf',
'ZeZlI', 'OHxRI', '8NRVj', 'aiUGz', '56lDR', '0WPB7', 'FwCSR',
'9qGsm', 'Pd95v', 'Vn0MB', 'tTSIE', 'FPi37', 'f8mRS', 'ht9cx',
'GZLaB', '4JRIQ', 'Qx6tT', 'FQCI7', 'voUyK', 'HkmYC', 'UJ2uT',
'gtgeo', 'gSmvA', 'J2lJJ', 'F0p8U', 'K2Tij', 'MZAYB', '2BZeq',
'4dtHc', 'e5XqF', 'NlAMb', 'jKdTK', 'qqKmX', 'qSehM', 'SN22Y',
'oz046', '8ta8f', 'Aa5Vt', 'l8Dhv', 'CzYTq', 'mgJRI', 'TBR83',
'N5AU6', 'shwmM', 'oaaRJ', 'qBevm', 'KVtWn', 'bKynV', 'Higco',
'PEKy6', 'NUn4u', 'svsNL']
```

Nous avons bien ici 100 résultats qui sont le fruit de 100 processus, dont certains ont été réalisés en parallèle sur les différents cœurs du Raspberry Pi. En effet, quand toutes les ressources de calcul sont occupées à une tâche, les tâches sont en attente de la libération des ressources. Le calcul est donc parallèle, mais dans la limite physique des ressources du système.

La librairie joblib est dédiée au calcul parallèle. Elle est le fruit de travaux collaboratifs de la communauté du libre.

Pour installer joblib, il suffit d'utiliser le gestionnaire de paquets :

```
pip3 install joblib
```

Vérifions le fonctionnement dans un environnement IPython en saisissant :

```
Import joblib
```

Sans erreur retournée lors de l'importation, nous pouvons considérer que la librairie est correctement importée.

Faire du calcul parallèle, c'est exécuter la même fonction, le même code sur des unités de calcul différentes. Dans notre cas, sur les quatre cœurs du Raspberry Pi. Mais cela pourrait être les n cœurs d'un cluster de calcul, par exemple un cluster de Raspberry Pi (on voit ce genre de clusters fleurir sur Internet) qui offre une puissance de calcul intéressante.

Pour illustrer le fonctionnement de joblib, nous allons réemployer la fonction de génération de chaînes de caractères aléatoires :

```
def chaine_aleatoire(taille):  
    rand_str = ''.join(random.choice(  
        string.ascii_lowercase +  
        string.ascii_uppercase +  
        string.digits)  
        for i in range(taille))  
    return rand_str
```

La seule modification est que nous retournons cette fois-ci directement une chaîne de caractères.

La mise en place de la distribution du calcul par Joblib est vraiment très simple puisqu'elle se fait en une seule ligne :

```
res = Parallel(n_jobs=4)(delayed(chaine_aleatoire)(5)  
    for _ in range(100))
```

Il s'agit ici de gérer une liste de 100 tâches à exécuter en parallèle sur les quatre cœurs du Raspberry Pi 3 B+. Cette liste est réalisée grâce à la fonction `delayed()` qui prend, pour chaque élément de la liste, le nom de la fonction à exécuter ainsi que les paramètres des fonctions.

La liste des tâches est exécutée en parallèle grâce à la fonction `Parallel()`.

Le code complet est donc :

```
import random
import string

from joblib import Parallel, delayed

random.seed(123)

def chaine_aleatoire(taille):
    rand_str = ''.join(random.choice(
        string.ascii_lowercase +
        string.ascii_uppercase +
        string.digits)
        for i in range(taille))
    return rand_str

res = Parallel(n_jobs=4)(delayed(chaine_aleatoire)(5)
                        for _ in range(100))

print(res)
```

Le résultat est le suivant :

```
In [1]: %run joblib.py ['KgPY1', '5s2PW', 'C6lr1', 'l0b5C', '4EK0w', 'tzwjU',
'AN7x3', '7Fjro', 'DsmX0', 'RgjpZ', '56iWk', '3T3AD', 'fnG0V',
'7gt7n', 'sTA96', 'fjbfs', '7L00a', 'UgUxS', 'ldr3E', 'pUo3E',
'Bpget', 'Vysxs', 'zya3Y', 'bKKnJ', '0G6D1', 'MsrxK', 'OJVjR',
'7YC5o', 'oSS9f', 'eoWoM', '3hdtr', 'j0xbM', 'Mm23f', '25TdM',
'FXwSP', 'kJcWe', 'MX9ID', 'YTSx8', 'reIeQ', '0j8FC', 'tl02s',
'Pjl0S', '8SxWv', 'GwRZ9', 'n1V5V', '5RNRI', 'lLMYU', 'Vrh5c',
'gotXh', 'ePdrZ', '50y50', 'BarJE', 'ujCnc', 'efJQX', 'ZfNlu',
'cV2U0', 'id2ge', 'gZDq0', 'xwKUj', 'TCYGF', '0bgb9', '9k9pf',
'arheH', 'tznZW', 'xOwpY', '8GwkJ', '0VJwA', 'fiDNj', 'FGmBq',
'7oFXs', 'jUCnS', 'qrPp7', 'THBX0', 'Sc0V5', 'WG7Jt', 'IzCuz',
'5p5fY', 'KskVD', 'PHxqp', 'uCopk', 'QitDA', 'fE7DL', 'B5k6Q',
'26ACW', 'EpgvL', 'GuErr', 'eQUmt', 'rSZid', 'xgyER', 'My5dg',
'EuwQR', 'hA14s', 'Vocur', '8TOU0', '4H9G6', 'vKLyg', 'YL2Wg',
'hOLpw', 'F3mdJ', 'FZMlo']
```

 **Le choix d'une des méthodes présentées ici dépend de votre problème et du contexte dans lequel vous voulez utiliser l'accélération des calculs proposée par la structure multicœur de votre Raspberry Pi.**

16. Documenter le code

Il existe un ensemble de recommandations pour Python qui sont formulées dans les documents PEP. La recommandation PEP 257 donne un ensemble de conseils pour documenter et commenter le code.

D'une manière générale, le code doit être commenté pour chaque fonction. PEP recommande également de ne pas commenter de façon excessive. En effet, les commentaires du type :

```
# ceci est un commentaire
```

ne doivent être utilisés que pour signaler à un relecteur du code ou rappeler à son créateur une particularité qui ne serait pas explicite à la lecture du code en question (le commentaire est à placer, à proximité du passage commenté).

Le code doit comporter en particulier des lignes de documentation pour toutes les fonctions. Ces blocs de documentation sont délimités à l'aide de guillemets :

```
"""Ceci est un bloc de commentaire"""
```

Il est possible d'insérer des commentaires sur une ou plusieurs lignes.

D'une manière générale, un commentaire comporte toujours en première ligne la description du rôle de la fonction. Le PEP 257 précise explicitement que cette ligne ne doit pas être un copier-coller du prototype de la fonction. Les lignes ci-après illustrent la documentation d'une méthode qui ne retourne pas de paramètre et ne prend pas de paramètre. Il y a donc une seule ligne :

```
def augmente_carburant(self):  
    """augmente le volume de carburant d'un litre"""  
    self.vol_carburant += 1
```

Dans un cas où il y a des paramètres, le commentaire qui décrit sur chaque ligne leur type et leur rôle. La valeur par défaut, quand il y en a une, doit être également indiquée. Il en va de même pour la valeur retournée. À titre d'illustration :

```
def maj_val(self, nouv_val=10000):  
    """Mise à jour de la valeur de la voiture en Euros  
  
    paramètre :  
        nouv_val -- valeur pour la mise à jour (10 000 par défaut)  
  
    return : entier -- valeur de l'attribut valeur après mise à jour  
    """  
    # -*- coding: utf-8 -*-  
    self.valeur = nouv_val  
    return self.valeur
```

commentons la classe `Voiture` suivante :

```

class Voiture(object):

    def __init__(self, modele, marque, annee):
        """constructeur de la classe voiture

        paramètres :
            modele -- modèle de la voiture

            marque -- marque du véhicule
            annee -- année de fabrication
        """
        self.modele = modele
        self.marque = marque
        self.annee = annee
        self.vol_carburant = 0
        self.valeur = 0

    def augmente_carburant(self):
        """augmente le volume de carburant d'un litre"""
        self.vol_carburant += 1

    def reservoir_plien(self):
        """vérifie si le niveau de carburant est à 50L

        return : booléen -- True si la valeur est 50 False sinon
        """
        if self.augmente_carburant == 50:
            return True
        else:
            return False

    def maj_val(self, nouv_val=10000):
        """Mise à jour de la valeur de la voiture en Euros

        paramètre :
            nouv_val -- valeur pour la mise à jour (10 000 par défaut)

        return : entier -- valeur de l'attribut valeur après mise à jour
        """
        # -*- coding: utf-8 -*-
        self.valeur = nouv_val
        return self.valeur

if __name__ == '__main__':
    ma_voiture = Voiture(modele='406', marque='peugeot', annee=2014)
    ma_voiture.augmente_carburant()

```

L'intérêt des blocs de commentaire, qu'on nomme par ailleurs docstrings, n'est pas seulement d'illustrer le code. Ils permettent aux outils d'autocomplétion de proposer le contenu d'une docstring lorsqu'on utilise une fonction liée à cette docstring. La figure ci-après illustre l'affichage (en haut dans la fenêtre de droite) d'une docstring du code précédent lors de l'usage d'un outil d'autocomplétion (plug-in YouCompleteMe) dans l'éditeur Vim.

```
1 augmente le volume de carburant d'un litre
2
3
4 """constructeur de la classe voiture
5
6 paramètres :
7     modele -- modèle de la voiture
8     marque -- marque du véhicule
9     annee -- année de fabrication
10
11 self.modele = modele
12 self.marque = marque
13 self.annee = annee
14 self.vol_carburant = 0
15 self.valeur = 0
16
17 def augmente_carburant(self, vol_carburant):
18     """augmente le volume de carburant d'un litre
19     """
20     self.vol_carburant += vol_carburant
21
22 def reservoir_plein(self):
23     """vérifie si le réservoir est plein
24     """
25     return self.vol_carburant == self.valeur
26
27 def verifie(self, vol_carburant):
28     """vérifie si le réservoir est plein
29     """
30     return self.vol_carburant == self.valeur
31
32 def maj_val(self, vol_carburant):
33     """Mise à jour du volume de carburant
34     """
35     self.vol_carburant = vol_carburant
36
37 if __name__ == '__main__':
38     ma_voiture = voiture('Renault', 'Clio', 2014)
39     ma_voiture.augmente_carburant(1)
40
41 classe_voiture.py[+]
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594

```

Tester le code

Le test de scripts est très utilisé dans les grands projets informatiques. La réalisation de tests automatiques permet de garantir une qualité de produit. Cette pratique fait partie de la liste des points cruciaux du test de Joël (<https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/>). Ce test évalue la capacité d'une équipe de développement à produire des applications de qualité.

Les tests sont cruciaux en développement informatique. C'est d'ailleurs l'un des postes de dépenses les plus importants pour Apple.

Il existe plusieurs types de tests :

- tests unitaires
- tests d'intégration

Les tests d'intégration sont réalisés lors de la phase d'assemblage des différentes parties de code du programme final. Ils sont généralement précédés par les tests unitaires qui sont l'objet de cette section.

Les tests unitaires permettent de tester des portions particulières du code afin de vérifier si les spécifications sont suivies. Les tests sont importants même pour le développeur qui travaille seul et dont le travail ne sera pas directement utilisé par un tiers.

En effet, les tests permettent d'éviter la « régression » d'un code ou d'un projet. Imaginons que vous développez une librairie de fonctions et que chaque fonction est couverte par des tests. Quelques mois après la finalisation du développement de cette librairie, un événement vous amène à changer les spécifications de votre librairie et donc à modifier votre code. Les tests vous permettront en partie d'assurer la compatibilité entre les anciennes utilisations de votre librairie et les utilisations postérieures à vos modifications.

Il existe différentes stratégies pour créer les tests unitaires. Certaines pratiques encouragent à créer les tests lors de la conception du projet ou de l'objet à développer, c'est-à-dire avant d'écrire le code. Ceci peut paraître étrange, mais présente de nombreux avantages, principalement l'absence de biais dans la conception des tests.

En effet, si l'auteur d'un code est également l'auteur des tests (ce qui ne pose pas de problème intrinsèque), et si ce même auteur écrit les tests après avoir écrit le code, alors il peut, de façon volontaire ou complètement involontaire, concevoir des tests qui ne couvrent que les éléments qu'il juge intéressants après le développement du code. Le risque est donc de tester uniquement des cas que le développeur sait qu'ils fonctionneront avec le code implémenté.

Pour réduire ce risque et donc enlever le biais potentiel, il est préférable d'écrire les tests avant le développement du code ou de les faire écrire par un tiers. Cette méthodologie de création des tests postérieure au développement du code est nommée *Test Driven Development* (en français, développement piloté par les tests).

Après ces considérations théoriques sur l'intérêt de tester le code, il est nécessaire de se poser la question sur les éléments à tester : lesquels ? tous ?...

Sur ce point, il n'existe pas de réponse parfaite. Les pratiques sont propres aux équipes de développement : c'est donc le contexte du projet et l'expérience de l'équipe qui va guider les choix.

Nous pouvons tout de même donner quelques éléments généraux de bonnes pratiques :

- Tester toutes les propriétés disponibles dans la spécification.
- Chercher à couvrir 100 % du code, c'est-à-dire que chaque ligne du code est couverte au moins par un test.
- Couvrir uniquement les éléments publics du code. En effet, les éléments privés sont appelés par les éléments publics. Par conséquent, quand on couvre un élément public par un test unitaire, les éléments privés qu'il utilise sont par voie de conséquence couverts eux-mêmes par le test.
- Retirer aucun test qui était déjà présent dans la base de tests existante.

Les sections qui suivent présentent deux technologies permettant de réaliser le test d'un élément de code. Afin de les illustrer, considérons la classe suivante :

```
class Voiture(object):
    def __init__(self, modele, marque, annee):
        self.modele = modele
        self.marque = marque
        self.annee = annee
        self.vol_carburant = 0

    def augmente_carburant(self):
        self.vol_carburant += 1
```

Unittest

Unittest est un module qui permet d'automatiser les tests. Il est disponible dans la librairie standard de Python. Il suffit donc d'importer le module pour l'utiliser :

```
import unittest
```

La méthodologie de test est assez simple. On crée une classe qui dérive de la `unittest.TestCase`. On nomme cette classe en respectant généralement le formalisme suivant : "nom de la classe à tester" accolé à la chaîne de caractères `Test`. Dans le cas présent : `VoitureTest`.

```
class VoitureTest (unittest.TestCase):
```

Il est nécessaire d'ajouter des méthodes à cette classe. Chaque méthode va tester, en général, une méthode particulière de la classe à tester. Il faut donc être explicite dans le nommage des méthodes.

À l'intérieur de ces méthodes, on peut instancier un objet et faire appel aux méthodes de cet objet.

Enfin, on utilisera des fonctions d'assertion de la librairie `unittest` pour vérifier si les valeurs retournées par une méthode ou si les attributs modifiés par la méthode ont bien les valeurs attendues. L'exemple qui suit illustre le procédé :

```
def test_cons(self):

    ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
    self.assertEqual(ma_voiture.modele, 'Mustang')
```

Ici, nousinstancions un objet `Voiture` et nous testons avec la méthode de la classe `unittest` `self.assertEqual()` que l'attribut `ma_voiture.modele` vaut bien `'Mustang'`.

Les éléments de test étant contenus dans une classe, il n'est pas possible d'exécuter directement le fichier Python. Il serait nécessaire d'instancier la classe. Cependant, le plus simple, comme l'illustre la documentation du module, est d'ajouter la fonction `main` suivante en fin de fichier :

```
if __name__ == '__main__':
    unittest.main()
```

Examinons à présent le code de test complet suivant :

```
import unittest
from classe_voiture import Voiture
```

```

class VoitureTest (unittest.TestCase):

    def test_cons(self):

        ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
        self.assertEqual(ma_voiture.modele, 'Mustang')

    def test_augmente_carburant(self):

        ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
        ma_voiture.augmente_carburant()
        self.assertEqual(ma_voiture.vol_carburant, 1)

if __name__ == '__main__':
    unittest.main()

```

Pour exécuter le test, il faut exécuter la ligne suivante dans une console :

```
ipython classe_voiture_test.py
```

Ce qui produit le résultat suivant :

```

pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$
ipython classe_voiture_test.py
..

```

```
-----
Ran 2 tests in 0.001s
```

```
OK
```

Cette sortie de console indique que deux tests ont été exécutés en 0,001 s et qu'ils se sont soldés par un succès.

Voici un exemple de sortie quand un des tests échoue :

```

pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$ ipython
classe_voiture_test.py

```

```
F.
```

```
=====
FAIL: test_augmente_carburant (_main__.VoitureTest)
-----
```

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

```
  File "/home/pi/Documents/Redaction/PythonRaspberry/Chap9/script/
  classe_voiture_test.py",
```

```
line 16, in test_augmente_carburant
```

```
    self.assertEqual(ma_voiture.vol_carburant, 2)
```

```
AssertionError: 1 != 2
```

```
-----
Ran 2 tests in 0.003s
```

```
FAILED (failures=1)
```


Pour le test `self.assertEqual()`, la valeur obtenue est 1 alors que la valeur que nous souhaitons obtenir est 2. Ces informations sont disponibles aux lignes :

```
self.assertEqual(ma_voiture.vol_carburant, 2)
AssertionError: 1 != 2
```

Pytest

Unitest permet de réaliser des tests de manière intéressante, mais il existe un module beaucoup plus pertinent : `pytest`. Le gros avantage de celui-ci est de nécessiter une écriture des tests moins longue. Cet avantage n'est pas négligeable, car le temps d'écriture des tests est dans certains cas au moins aussi long, voire plus long que le temps de rédaction du code.

Avec `pytest` il n'est pas nécessaire de surcharger une classe. Il suffit d'écrire des fonctions. Et d'utiliser la fonction `assert` pour tester les valeurs retournées par des méthodes ou les valeurs des attributs.

Avant toute chose, il faut installer le module `pytest` à l'aide de `pip`.

```
pip install pytest
```

Le code qui suit teste la classe `Voiture` avec `pytest` :

```
from classe_voiture import Voiture
def test_cons():

    ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
    assert ma_voiture.modele == 'Mustang'
    assert type(ma_voiture.modele) is str
    assert type(ma_voiture.marque) is str
    assert type(ma_voiture.marque) is int or float

def test_augmente_carburant():

    ma_voiture = Voiture(modele='Mustang', marque='ford', annee='1968')
    ma_voiture.augmente_carburant()
    assert ma_voiture.vol_carburant == 1
```

Pour exécuter `pytest`, il suffit de saisir la ligne suivante dans un terminal :

```
pytest classe_voiture_test_pytest.py
```

Le résultat obtenu est le suivant :

```
pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$ pytest
classe_voiture_test_pytest.py
=====
===== test session starts
=====
platform linux2 -- Python 2.7.13, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /home/pi/Documents/Redaction/PythonRaspberry/Chap9/script, inifile:
collected 2 items

classe_voiture_test_pytest.py .
```

```

.
=====
===== 2 passed in 0.12 seconds
=====
=====

```

Il est également possible de rendre pytest plus verbeux à l'aide de l'option `-v`. Ce qui donne :

```

pi@raspberrypi:~/Documents/Redaction/PythonRaspberry/Chap9/script$ pytest
-v classe_voiture_test_pytest.py
=====
===== test session starts
=====
=====
platform linux2 -- Python 2.7.13, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
-- /usr/bin/python
cachedir: .cache
rootdir: /home/pi/Documents/Redaction/PythonRaspberry/Chap9/script, inifile:
collected 2 items

classe_voiture_test_pytest.py::test_cons PASSED
classe_voiture_test_pytest.py::test_augmente_carburant PASSED

=====
===== 2 passed in 0.04 seconds
=====
=====

```

17. Profiler le code

Profiler le code consiste à mesurer les ressources nécessaires pour l'exécuter, tant les ressources mémoire que les ressources de calcul.

La stratégie consiste généralement, lors d'une phase d'optimisation, à mesurer le temps d'exécution des éléments constituant le programme et à chercher à réduire le temps d'exécution des éléments les plus chronophages. Il est alors possible de comparer différentes versions d'une fonctionnalité.

Python propose un module nommé `cProfile` qui réalise ces mesures.

Pour l'utiliser, on crée généralement la structure suivante :

```
pr = cProfile.Profile()
pr.enable()
# Placer ici des éléments à tester
pr.disable()
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

Les éléments à tester sont placés entre l'appel de la méthode `enable()` et l'appel de la méthode `disable()`.

Reprenons la classe `Voiture` :

```

class Voiture(object):

    def __init__(self, modele, marque, annee):
        """constructeur de la classe voiture

        paramètres :
            modele -- modèle de la voiture
            marque -- marque du véhicule
            annee -- année de fabrication
        """
        self.modele = modele
        self.marque = marque
        self.annee = annee
        self.vol_carburant = 0
        self.valeur = 0

    def augmente_carburant(self):
        """augmente le volume de carburant d'un litre"""
        self.vol_carburant += 1

    def reservoir_plien(self):
        """vérifie si le niveau de carburant est à 50L

        return : booléen -- True si la valeur est 50 False sinon
        """
        if self.augmente_carburant == 50:
            return True
        else:
            return False

    def maj_val(self, nouv_val=10000):
        """Mise à jour de la valeur de la voiture en Euros

        paramètre :
            nouv_val -- valeur pour la mise à jour (10 000 par défaut)

        return : entier -- valeur de l'attribut valeur après mise à jour """
        # -*- coding: utf-8 -*-
        self.valeur = nouv_val
        return self.valeur

if __name__ == '__main__':
    ma_voiture = Voiture(modele='406', marque='peugeot', annee=2014)
    ma_voiture.augmente_carburant()

```

Nous pouvons produire le script de test suivant :

```
import cProfile
import pstats
import io

from classe_voiture import Voiture

if_name == '__main__':

    pr = cProfile.Profile()
    pr.enable()
    for i in range(1000):
        ma_voiture = Voiture(marque='VW', modele='polo', annee=2018)
    pr.disable()
    s = io.StringIO()
    sortby = 'cumulative'
    ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
    ps.print_stats()
    print(s.getvalue())
```

Notez que la classe `voiture` est instanciée 1000 fois pour obtenir des temps de calcul plus long :

```
for i in range(1000):
    ma_voiture = Voiture(marque='VW', modele='polo', annee=2018)
```

Le but est d'avoir des valeurs statistiquement plus vraies. Mais c'est surtout parce que l'instanciation de l'objet est tellement rapide qu'avec une seule instance le module ne peut mesurer les temps nécessaires.

Le résultat obtenu est le suivant :

```
In [2]: %run profilage.py
        1001 function calls in 0.006 seconds

Ordered by: cumulative time
ncalls tottime percall cumtime percall filename:lineno(function)
1000    0.006    0.000    0.006    0.000
/home/pi/Documents/Redaction/PythonRaspberry/Chap9/script/classe_voiture.py:3(__init__)
1      0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Ceci indique que 1 001 fonctions ont été appelées : la fonction d'instanciation 1 000 fois et la fonction `disable()` 1 fois. Par ailleurs les 1 000 instanciations prennent 0,006 seconde.

Pour clore ce chapitre, réalisons la comparaison de deux codes : le premier dit non pythonic, le second, pythonic.

Un code est dit pythonic lorsqu'il utilise des voies de programmation qui optimisent la lisibilité et améliorent les performances.

Par exemple, le code qui suit est non pythonic :

```
for i in range(1000):
    l = []
    for i in range(100):
        l.append(i)
```

Sa version pythonic est :

```
for i in range(1000):
    l = [i for i in range(100)]
```

Comparons les performances :

Pour la version non pythonic :

```
In [9]: %run profilage_np.py
        100001 function calls in 0.053 seconds
```

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
100000	0.053	0.000	0.053	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Pour la version pythonic :

```
In [10]: %run profilage_p.py
        1001 function calls in 0.039 seconds
```

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1000	0.039	0.000	0.039	0.000	
/home/pi/Documents/Redaction/PythonRaspberry/Chap9/script/profilage_p.py:11(<listcomp>)					
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Nous avons donc une performance de 30 % supérieure avec la syntaxe dite pythonic.