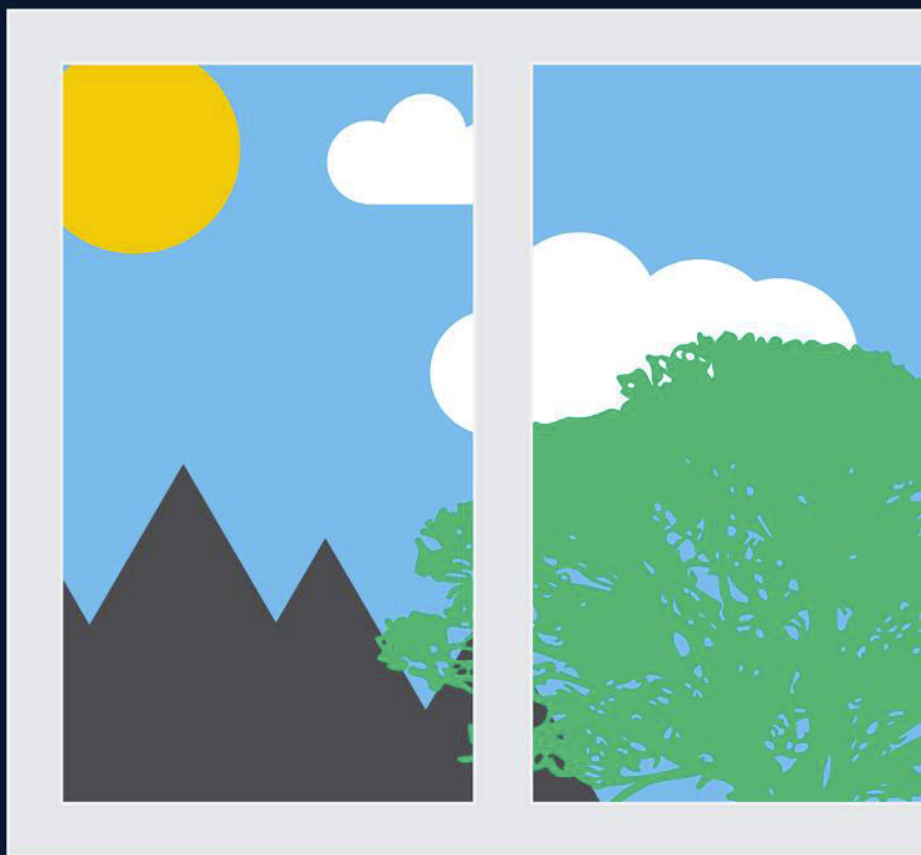


python

Par la pratique

Première édition



Par Honoré Hounwanou

Premiers pas avec Python

Le livre que j'aurais aimé lire lorsque je débute en programmation...

Honoré Hounwanou

Ce livre est en vente à <http://leanpub.com/premiers-pas-avec-python>

Version publiée le 2017-05-31



Ce livre est publié par [Leanpub](#). Leanpub permet aux auteurs et aux éditeurs de bénéficier du Lean Publishing. [Lean Publishing](#) consiste à publier à l'aide d'outils très simples de nombreuses itérations d'un livre électronique en cours de rédaction, d'obtenir des retours et commentaires des lecteurs afin d'améliorer le livre.

© 2015 - 2016 Honore Hounwanou

Dédicaces

- *À ma famille qui a toujours cru en moi.*
- *À Moustapha Loum pour cette magnifique couverture.*
- *À toute l'équipe d'Upperskies pour ces congés qui m'ont permis d'écrire paisiblement chaque ligne de cet ouvrage.*
- *À toute l'équipe des TEACHERS DU NET et à nos précieux abonnés.*

Table des matières

Dédicaces	iii
L’auteur	i
1. Introduction	1
1.1 Faire un don	1
1.2 Prérequis	1
1.3 Corrections et suggestions	2
1.4 Résumé	2
2. Jargon Informatique	3
2.1 Qu’est-ce que la programmation ?	3
2.2 Qu’est-ce qu’un langage de programmation ?	5
2.3 Le langage Python	9
2.4 Résumé	10
3. Installation de Python	12
3.1 Installer Python sous Windows	12
3.2 Installer Python sous Linux	13
3.3 Installer Python sous Mac OS X	14
3.4 La console d’interprétation de Python	14
3.5 Résumé	16
4. Notre premier jeu	17
4.1 Le principe du jeu	17
4.2 Les chaines de caractères	19
4.3 Les variables	21
4.4 Les commentaires	23
4.5 Passons au code source !	24
4.6 Résumé	28
5. Les fonctions	30
5.1 Objectifs de la modularité	30
5.2 Les fonctions	30
5.3 Résumé	45

TABLE DES MATIÈRES

6. les conditions et les boucles	47
6.1 Les conditions	47
6.2 Les itérations ou boucles	54
6.3 Résumé	60
7. Les structures de données (Partie 1/2)	61
7.1 Les tuples	61
7.2 Les listes	62
7.3 Les dictionnaires	64
7.4 Résumé	66
8. Les structures de données (Partie 2/2)	67
8.1 Les types non scalaires	67
8.2 Les chaînes de caractères	67
8.3 Les tuples	72
8.4 Les listes	73
8.4.2 Le full slicing	75
8.5 Les méthodes split et join	75
8.6 Résumé	76
9. Modules & Packages	77
9.1 Les modules	77
9.2 Les packages	85
9.3 Résumé	85
10. Jeu de capitales	86
10.1 Le principe du jeu	86
10.2 Le module random	87
10.3 Challenge	93
10.4 Exemple de Solution	93
10.5 Faites vous confiance !	95
10.6 Petit exercice	96
10.7 Résumé	97
11. Les fichiers et les exceptions	98
11.1 Les données sont très souvent externes	98
11.2 La fonction open	99
11.3 Contenu du fichier capitales.txt	101
11.4 Un code plus sûr	104
11.5 Jeu de capitales - Version 2	119
11.6 Petit exercice	121
11.7 Résumé	122

TABLE DES MATIÈRES

Conclusion	123
-----------------------------	------------

L'auteur

Honoré Hounwanou est un développeur, formateur et entrepreneur résidant au Canada.

Titulaire d'un Master en Télécommunications, il effectue actuellement un Doctorat en Sécurité Informatique à l'Université LAVAL.

Il a fondé il y a de cela 4 ans, la plateforme d'E-learning [LES TEACHERS DU NET](http://teachersdunet.com/)¹ offrant des formations et tutoriels de catégories diverses expliqués de manière simple et chirurgicale.

Il est Co-fondateur et Directeur Technique du Département Développement & Recherche de la startup [PAYDUNYA](https://paydunya.com)² (Solution de paiement en ligne via Téléphone Mobile).

Il parle couramment le Python, Java, PHP, Ruby, HTML/CSS/JavaScript, C/C++, C#, SQL, Scratch.

1. <http://teachersdunet.com/>

2. <https://paydunya.com>

1. Introduction

Très souvent lorsque nous nous lançons dans l'apprentissage d'un nouveau langage de programmation, nous commençons par une phase théorique dans laquelle nous essayons un tant soit peu de comprendre le principe de fonctionnement dudit langage et tout le jargon informatique associé. Ce n'est qu'après cela, que nous pensons généralement à **réellement pratiquer**.

Mais vu d'un autre angle et étant donné que l'objectif final consiste habituellement à réaliser plus tard quelques projets avec ce fameux langage, je me suis dit qu'il pourrait être également intéressant, de faire pourquoi pas, d'une pierre deux coups. **C'est-à-dire apprendre, mais cette fois-ci en pratiquant!**

Ainsi, grâce à ce livre, nous apprendrons à programmer en Python en nous amusant à créer une série d'applications ludiques. Python est un langage très simple d'apprentissage et vous pouvez l'utiliser pour créer par exemple des applications Web, des jeux vidéos, des applications mobiles, voir même un moteur de recherche !

Histoire de vous donner une idée des différents types d'applications que vous pourrez réaliser avec le langage Python, je vous invite à visiter la page [Histoires à succès](#)³ du site officiel de Python — Désolé aux allergiques à la langue de Shakespeare :).

Nous commencerons en douceur, en abordant les notions les plus simples afin de ne perdre personne en chemin. Ce sera ensuite le moment de nous intéresser à des notions beaucoup plus avancées comme celles des structures de données (les listes, les tuples, les dictionnaires...), des modules, des packages, des fichiers, du concept de programmation orientée objet...

Hmm sacré programme n'est-ce pas ? Alors ready ? Let's go !

1.1 Faire un don

J'ai mis des mois à travailler sur ce livre et je suis sûr que vous allez l'adorer. Il est entièrement **GRATUIT** et il le restera ! Toutefois, vous pouvez m'aider dans cette mission de partage gratuit du savoir en faisant un [don](#)⁴ symbolique. Sachez que j'apprécierai énormément. Voici encore le [lien](#)⁵. Merci :).

1.2 Prérequis

Il vous faut tout simplement avoir un ordinateur avec n'importe lequel des principaux systèmes d'exploitation — Windows, Linux, Mac et je crois que ce sera tout :).

3. <https://www.python.org/about/success/>

4. https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=YNQAM2MKNDC8L

5. https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=YNQAM2MKNDC8L

1.3 Corrections et suggestions

Nul n'est parfait, surtout lorsqu'il s'agit d'écrire un document technique. Si vous trouvez des fautes d'orthographe ou de grammaire, des erreurs de programmation etc, je vous prie de bien vouloir me le notifier par mail. Si l'erreur s'avère vérifiée, une modification sera apportée au document afin de corriger ladite erreur. Mon adresse e-mail personnelle : mercuryseries@gmail.com.

Merci.

1.4 Résumé

Dans ce premier chapitre, nous avons appris que :

- Python est actuellement l'un des meilleurs langages de programmation. Il est à la fois simple, élégant et puissant.
- Python est un langage de programmation fascinant et idéal pour des personnes qui souhaitent découvrir le monde de la programmation.
- Python est un langage polyvalent. On peut l'utiliser pour concevoir des applications variées : sites Web, jeux vidéos, applications mobiles...

2. Jargon Informatique

Ce chapitre aurait dû avoir pour titre “De la pure théorie !” car c’est ce à quoi vous aurez droit tout au long de ce chapitre. L’idée est de pouvoir découvrir un tant soit peu le jargon informatique affilié au monde de la programmation et ce via des exemples concrets.

Si vous êtes débutant, les quelques petits termes techniques dont regorge ce chapitre pourront vous donner l’impression que les choses ont l’air un tantinet compliquées. Soyez sans crainte, ce n’est pas du tout le cas. *Programmer a toujours été fun !*

D’autre part, si vous n’êtes pas à votre premier langage de programmation, vous risquez d’avoir l’air de vous ennuyer en lisant ce chapitre. Mais comme on le dit, la répétition est pédagogique et je peux vous garantir que vous apprendrez forcément quelque chose de nouveau.

Vous êtes toutefois libres de passer directement à la section consacrée à [l’installation de Python](#).

2.1 Qu’est-ce que la programmation ?

Comme vous le savez sûrement, il existe une pléthore de langages de programmation à savoir le C, C++, Java, Perl, PHP, Python, Ruby pour ne citer que ceux-là. Jugez-en par vous même, [la liste est longue](#)⁶ :). Mon choix s’est plutôt porté sur le dernier PYTHON pour son dynamisme, sa portabilité et sa syntaxe assez élégante.

Alors qu’est-ce que cela signifie : langage de programmation, portabilité, dynamisme, syntaxe ?

Si ces termes vous semblent abstraits et représentent des boites noires à vos yeux, ne vous inquiétez surtout pas, nous apporterons lumière à tout ceci dans les sections suivantes. C’est en effet l’objectif de ce chapitre.

2.1.1 Connaissances déclarative et impérative

Dans la vie réelle, il existe deux types de connaissances. Nous avons la connaissance dite *déclarative* et celle dite *impérative*.

La **connaissance déclarative** est composée d’énonciations à priori vraies. Par exemple :

- Il faut bien manger pour être en bonne santé.
- Quand le feu est rouge, les voitures doivent s’arrêter et laisser les piétons passer.
- Ou encore y est une racine carrée de x si et seulement si $y^2 = x$.

6. https://fr.wikipedia.org/wiki/Liste_des_langages_de_programmation

Le dernier exemple semble plutôt intéressant, mais peut déjà faire rougir les allergiques aux mathématiques.

Détaillons un peu, histoire d'être sur la même longueur d'onde. Même si je sais très bien que cela n'en vaut pas la peine, je me dis que ça peut arriver de temps en temps quelques petites fuites de cerveau :).

Alors voilà :

- 4 est une racine carrée de 16 car $4 \times 4 = 16$
- 3 est une racine carrée de 9 car $3 \times 3 = 9$ — aussi simple que cela.

La connaissance impérative quant à elle fait référence à un procédé permettant d'accomplir une tâche. C'est un peu comme une recette.

Je vous donne un petit exemple pour éclaircir les choses.

Pour calculer le double du tiers d'un nombre, il faut :

1. Avoir le nombre lui-même (Et c'est à ce moment là que vous vous dites : c'est évident !).
2. Le diviser par 3 pour avoir son tiers.
3. Multiplier ensuite par 2 le résultat obtenu en 2., pour avoir le double du tiers dudit nombre.

A priori, en suivant cette petite recette, on peut avoir le double du tiers de n'importe quel nombre.

C'est cela en quelque sorte la connaissance impérative.

Si je vous ai défini ce que sont la connaissance déclarative et la connaissance impérative, c'est parce qu'un bon programmeur doit avoir ces deux types de connaissances, du moins en fonction de ce qu'il souhaite réaliser comme programme.

Je m'explique. Si vous voulez écrire un programme qui calcule l'intensité de la force électrostatique entre deux charges électriques q_1 et q_2 séparées par une distance r .

Il faut avoir premièrement la connaissance déclarative, c'est-à-dire qu'est-ce que l'intensité de la force électrostatique ?



La force électrostatique

L'intensité de la force électrostatique entre deux charges électriques est proportionnelle au produit des deux charges et est inversement proportionnelle au carré de la distance entre les deux charges. - Cf Wikipédia

Et deuxièmement la connaissance impérative, c'est-à-dire comment calculer la force électrostatique créée par 2 charges ?

1. Avoir les valeurs numériques de q_1 et q_2 .
2. Multiplier q_1 et q_2 .
3. Multiplier ensuite le résultat obtenu en 2. par la constante de coulomb K
4. Diviser le résultat obtenu en 3. par le carré de la distance entre les deux charges (r^2).

Bah ne vous inquiétez surtout pas si vous ne savez toujours pas ce qu'est la force électrostatique :). Ceci n'est pas l'objectif recherché dans cet ouvrage. Comprenez plutôt la logique qui est derrière cet exemple. Nous avons eu besoin des connaissances déclarative et impérative pour résoudre notre problème précédent. C'est tout ce qu'il y a à comprendre !

2.1.2 Algorithme, programme et instruction

Voici trois définitions fondamentales que vous devrez absolument retenir. N'hésitez pas à les relire plusieurs fois s'il le faut :).

- Un algorithme est tout simplement un procédé permettant de résoudre un problème donné.
- Un programme quant à lui est un ensemble d'instructions exécutées par l'ordinateur. (C'est en effet la traduction de votre algorithme dans un langage de programmation).
- Une instruction est une tâche que doit exécuter l'ordinateur.

Certains programmeurs parlent plutôt de **script** en lieu et place de programme. On peut dire que programme = script. Ce ne sont en grosso-modo que des synonymes.

2.2 Qu'est-ce qu'un langage de programmation ?

Rappelons qu'un programme est juste une séquence d'instructions disant à l'ordinateur ce qu'il doit faire. Évidemment, nous avons besoin de fournir ces instructions dans un langage que l'ordinateur peut comprendre. Il aurait été très intéressant si nous pouvions juste dire à un ordinateur ce qu'il doit faire dans notre langue maternelle, comme dans les films de science-fiction. Malheureusement, en dépit de plusieurs efforts des informaticiens (y compris moi :)), créer des ordinateurs capables de comprendre le langage humain est un problème encore non résolu.

Même si les ordinateurs pouvaient nous comprendre, le langage humain n'est vraiment pas commode pour décrire des algorithmes assez complexes car il est rempli d'ambiguïté et d'imperfection.

Prenons un petit exemple (j'aime bien les exemples :)). Si je disais "J'ai vu un homme dans le parc avec des jumelles". Est-ce que cela veut dire que j'avais des jumelles dans les mains ? Ou plutôt c'était l'homme en question que j'ai vu dans le parc qui avait des jumelles ? Et de plus qui est-ce qui était dans le parc ?

J'ose espérer que vous comprenez à présent ce que je voulais dire. Les informaticiens ont donc essayé de résoudre ce problème en créant des notations spéciales pour exprimer nos intentions de manière

précise et non ambiguë. L'ensemble de ces notations spéciales forme ce qu'on appelle un **langage de programmation**.

Chaque structure dans un langage de programmation a une forme précise (syntaxe) et une signification correcte.

Python est un exemple de langage de programmation et c'est ce dernier que nous allons découvrir dans ce livre. Vous avez peut être déjà entendu parler d'autres langages de programmation comme le C++, C#, Ruby, Java...

Bien que ces langages diffèrent de beaucoup de détails, ils partagent la propriété d'avoir une syntaxe bien définie et non ambiguë et une sémantique.

Tous les langages cités plus haut sont des exemples de langages de haut niveau. En fait, plus le langage de programmation se rapproche du langage humain, plus on dit qu'il est de haut niveau. (mais évidemment la traduction en binaire prendra un peu plus de temps :))

Binaire ? L'ordinateur ne comprend en réalité qu'un seul langage qu'on appelle communément **langage machine ou langage binaire**. Ce langage n'est composé que de deux chiffres 0 et 1 appelés aussi **bits (élément binaire)**. En d'autres termes, l'ordinateur ne comprend que des 0 et 1. Écrire un programme avec des 0 et 1 est un vrai parcours de combattant.

Ainsi grâce au langage de programmation qui se rapproche un peu plus du langage naturel (anglais, français ...), nous pourrons écrire nos programmes (on dit aussi coder) plus aisément. Ce code sera ensuite traduit en langage machine afin que l'ordinateur puisse le comprendre.

Alors comment est-ce que notre code est traduit en langage machine ??

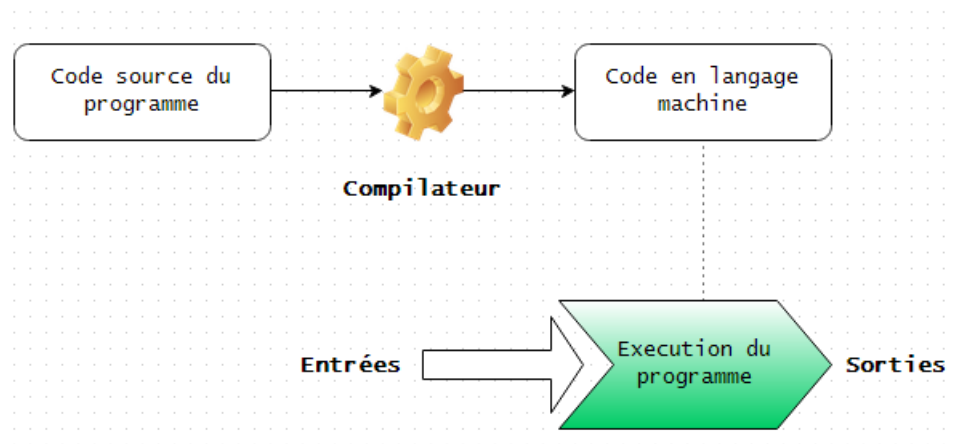
2.2.1 Compilateur Vs Interpréteur

Il existe de manière générale deux moyens principaux pour traduire un code écrit dans un langage de programmation en langage machine. Un langage de programmation peut être soit compilé ou soit interprété.

Compilateur

Un compilateur est un programme complexe qui prend un programme écrit dans un langage de programmation (en C par exemple) et le traduit en un programme équivalent en langage machine.

Le programme écrit dans un langage de programmation est appelé **code source** et le résultat obtenu après traduction en code machine est un programme que l'ordinateur peut directement exécuter.

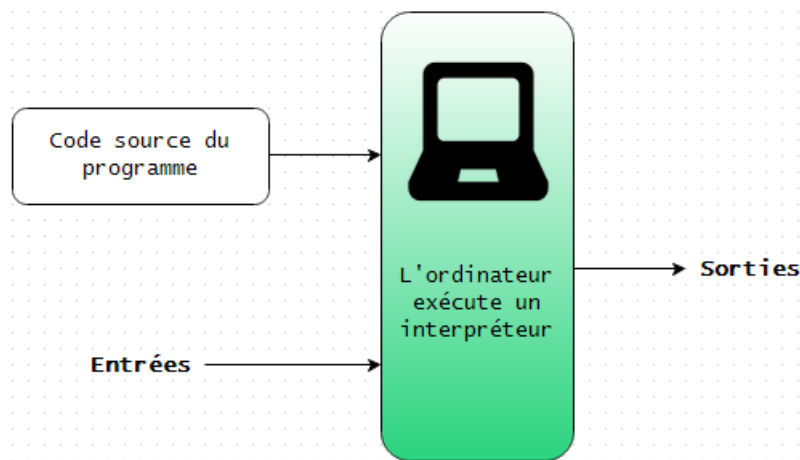


Processus de compilation d'un programme

Interpréteur

Un interpréteur est un programme qui simule un ordinateur qui comprend un langage de haut-niveau. Plutôt que traduire le code source du programme en langage machine, l'interpréteur analyse et exécute le code source instruction par instruction.

Python est un exemple de langage interprété. Ce qui revient à dire que nous aurons besoin d'un interpréteur Python afin de pouvoir exécuter nos futurs programmes. Nous verrons comment l'installer dans le prochain chapitre.



Processus d'interprétation d'un programme

La différence entre l'interprétation et la compilation est que la compilation est un peu plus courte lors de la traduction en langage machine. Une fois le programme compilé et que tout fonctionne, il peut être exécuté plusieurs fois sans avoir besoin de compiler à nouveau le code source. Alors que dans le cas de l'interpréteur, il a besoin d'être lancé à chaque fois qu'on exécute le programme vu qu'il devra de nouveau analyser et exécuter le code source instruction par instruction. Les programmes

compilés ont donc tendance à être plus rapide à l'exécution, vu que la traduction est faite une fois pour toute, mais les langages interprétés permettent une plus grande flexibilité pour l'environnement de programmation car les programmes peuvent être développées et exécutés de manière interactive.

Le processus d'interprétation fait ressortir un autre avantage, c'est que les programmes écrits pourront être généralement exécutés sur plusieurs plateformes (plusieurs systèmes d'exploitation différents) ; on parle de **portabilité**.

Pour résumer, sachez qu'un programme écrit dans un langage de programmation peut être exécuté sur plusieurs sortes d'ordinateurs tant qu'il y a un compilateur ou interpréteur approprié.

2.2.2 Syntaxe, sémantique statique, sémantique

Syntaxe

La **syntaxe** permet de définir les séquences de caractères et de symboles constituant une chaîne bien formée.

Exemple:

- $3 + 5$ (Bonne syntaxe)
- $3\ 5$ (Mauvaise syntaxe)

En effet, une expression est composée d'une opérande suivi d'un opérateur suivi à son tour d'une autre opérande. Pour faire simple, $3 + 5$ est une expression car nous avons bel et bien une opérande (3), un opérateur (+) et une autre opérande (5). Ce qui n'est pas le cas dans la deuxième expression (elle n'en a pas une) où il n'y a pas d'opérateur.

Sémantique statique

La **sémantique statique** quant à elle vérifie si les chaînes bien formées ont une signification.

Exemple:

- 'Premier' + 'cours' (Bonne syntaxe et bonne sémantique statique) car on peut additionner deux textes. On parle en jargon informatique de **concaténation**. Mais soyez sans crainte, nous en reparlerons.
- $3 + \text{'abc'}$ (Bonne syntaxe mais mauvaise sémantique statique) cela n'a aucun sens ! On ne peut additionner banane et pomme :)

Sémantique

La **sémantique**, une fois l'expression évaluée, vérifie si l'opération est possible ou non.

Exemple: $3 / 0$. Ici la syntaxe et la sémantique statique sont correctes mais par contre la sémantique est mauvaise car on ne peut diviser un nombre par 0.

Tous ces termes ne sont que techniques. Donnez vous le temps de les assimiler ou de les oublier :).

2.3 Le langage Python

Dans cette section, nous parlerons du langage Python. Nous verrons de façon sommaire son histoire et ses forces.

2.3.1 Naissance de Python

Comme vous le savez maintenant, Python est un langage de programmation. Il a été créé en 1991 par le développeur néerlandais **Guido van Rossum**.



Guido van Rossum

Guido a décidé de baptiser son projet **Python** en référence à la série télévisée des [Monty Python](https://fr.wikipedia.org/wiki/Monty_Python)⁷ dont il en est un grand fan.

La syntaxe de Python ressemble un tant soit peu à celle de Perl, Ruby ou encore Smalltalk.

7. https://fr.wikipedia.org/wiki/Monty_Python

Ne vous inquiétez surtout pas si vous ne connaissez aucun de ces trois langages de programmation. Je vous expliquerai tout ce que vous devez savoir sur Python. Mais si par contre vous connaissez l'un de ces langages, il va s'en dire que votre apprentissage de Python en sera facilité.

2.3.2 Pourquoi choisir Python ?

Vous avez probablement choisi d'apprendre à programmer en Python parce que vous avez entendu du bien de ce langage ou tout simplement parce que vous souhaitez le découvrir par pure curiosité. Toutefois, il peut vous sembler encore difficile à ce moment précis de répondre à la question suivante : *“Pourquoi Python et pas l'un des autres langages ?”*.

Laissez moi vous donner quelques raisons de choisir le langage Python :

- Python est facile d'apprentissage.
- Python est un langage orienté objet. Si le concept de programmation orientée objet ne vous est pas familier, ne vous inquiétez surtout pas, un chapitre tout entier y sera dédié. Après l'avoir lu, vous comprendrez pourquoi c'est une bonne chose que Python soit orienté objet. Mais si vous avez déjà eu à faire de la programmation orientée objet dans un langage autre que Python, vous verrez que Python possède certaines particularités.
- Python a une élégante syntaxe, ce qui facilitera grandement la lisibilité de tous vos scripts Python.
- Python dépend fortement des espaces, ce qui vous contraint en quelque sorte à écrire des programmes plus lisibles.
- Pas d'obligation à utiliser des points virgules comme en langage C ou encore PHP. Python fera le travail pour vous.
- Avec Python vous n'avez qu'à dire ce que vous souhaitez faire, et le travail sera fait. Par exemple si je veux trier un ensemble de valeurs, je n'aurai qu'à dire *“sort”* (trier en anglais). Si je veux rechercher une valeur, je n'aurai qu'à dire *“find”* (rechercher en anglais)...

2.4 Résumé

Dans ce second chapitre, nous avons appris que :

- Un bon programmeur doit s'assurer de toujours avoir les connaissances déclarative et impérative affiliées au programme qu'il souhaite réaliser.
- L'ordinateur ne comprend que le langage binaire. C'est-à-dire que des 0 et des 1.
- Un langage de programmation est un ensemble de notations spéciales nous permettant de dialoguer avec notre ordinateur de manière non ambiguë.
- Python est un langage de programmation.
- Python n'est pas le seul langage de programmation. Il existe d'autres langages de programmation comme le C, C++, Java, C#, Ruby...

- Il existe de manière générale, deux moyens principaux pour traduire un code écrit dans un langage de programmation en langage machine : la compilation et l'interprétation.
- Python est un langage interprété à la différence des langages comme le C, C++, Visual Basic... qui quant à eux sont des langages dits compilés.
- Python a été créé en 1991 par le programmeur néerlandais Guido van Rossum.
- Python est un langage de programmation orienté objet.
- Python possède une élégante syntaxe et encourage grandement la lisibilité de votre code source.

3. Installation de Python

Enfin, les choses sérieuses peuvent démarrer...

Comme nous allons le voir dans les lignes qui suivent, installer Python est la chose la plus facile qui puisse exister au monde ! Je ne le dis pas uniquement pour ceux qui sont sous Windows, mais également pour les linuxiens et les fanatiques Mac OS.

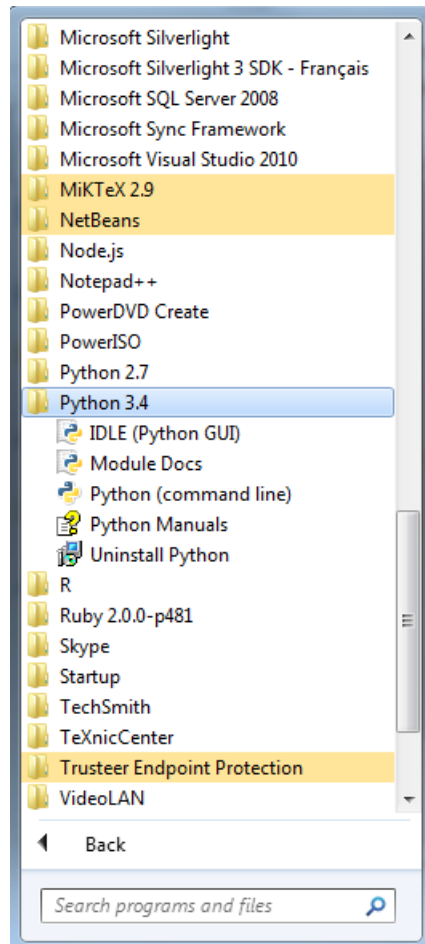
Quelque soit votre système d'exploitation, la première étape consiste à vous rendre sur le site officiel de Python : <http://python.org>⁸

3.1 Installer Python sous Windows

1. Cliquez sur le lien *Downloads* au niveau du menu principal de la page d'accueil
2. Sélectionnez la version de Python que vous souhaitez installer. (Je vous conseille d'opter pour la dernière version en date.)
3. Sans grande surprise, le téléchargement du fichier d'installation de Python va alors débuter.
4. Une fois le téléchargement terminé, exécutez ensuite le fichier d'installation et suivez les différentes étapes. Pensez à cocher la case "Add Python 3.x to PATH" lors de l'installation afin que Python soit accessible en ligne de commandes. Cela s'avéra utile plus tard si vous souhaitez créer des sites web en un temps record avec [Django](http://djangoproject.com/)⁹.
5. A présent, votre installation est normalement terminée. Vous pouvez, histoire d'avoir confirmation, vous rendre dans le menu *Démarrer > Tous les programmes*, un dossier Python devrait normalement être présent au niveau de la liste.

8. <http://python.org>

9. <http://djangoproject.com/>



Dossier Python dans le menu Démarrer



Quel lien choisir ?

Si vous avez à faire un choix entre plusieurs liens d'installation. Sélectionner celui qui correspondra à votre type de processeur. Au cas où, vous ne connaissez pas le type de votre système (32 bits ou 64 bits), choisissez tout simplement la version << x86 >>. Si vous aimez également les vidéos, [celle-ci](https://www.youtube.com/watch?v=exwWzZ5VhQI)¹⁰ fera l'affaire !

3.2 Installer Python sous Linux

Sur la plupart des distributions Linux, Python est généralement pré-installé. Cependant, il est possible que vous n'ayez pas la dernière version en date.

10. <https://www.youtube.com/watch?v=exwWzZ5VhQI>



Version de Python installée

Afin de connaître la version de Python installée, tapez dans un terminal la commande `python -V`

Il est très probable que ce soit une version de la branche 2.x, comme 2.6 ou 2.7, pour des raisons de compatibilité. Dans tous les cas, je vous conseille d'installer la dernière version en date de la branche 3.x.

Cliquez sur *Downloads* et téléchargez la dernière version de Python. Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive :

1. Décompressez l'archive en tapant : `tar -xvfz Python-3.4.1.tar.bz2` (cette commande est bien entendu à adapter suivant la version et le type de compression).
2. Une fois la décompression terminée, vous devez vous rendre dans le dossier qui vient d'être créé dans le répertoire courant (Python-3.4.1 dans mon cas).
3. Il faudra lancer le script de configuration en tapant `./configure` dans la console. Une fois que la configuration terminée, il n'y a plus qu'à compiler en tapant `make` puis `make install` en tant que root (super-utilisateur) pour installer Python à proprement dit.



Quelqu'un peut-il m'aider ?

Si vous rencontrez des difficultés, [cette vidéo](#)¹¹ pourra vous guider dans votre installation de Python.

3.3 Installer Python sous Mac OS X

Sur le site officiel de Python, vous trouverez des paquetages pour Mac OS similaires à ceux proposés sous Windows.

Téléchargez la dernière version en date. Ouvrez le fichier `.dmg` et faites un double-clic sur le paquet d'installation `Python.mpkg`.

Suivez les différentes étapes de l'assistant d'installation et vous aurez Python installé sur votre précieux Mac :).

3.4 La console d'interprétation de Python

Est-ce vrai ce qu'on dit ? Python est installé sur votre machine ? C'est l'information que je viens de recevoir en regardant le JT de 20h :). Qu'est-ce qu'on attend donc pour le lancer ?

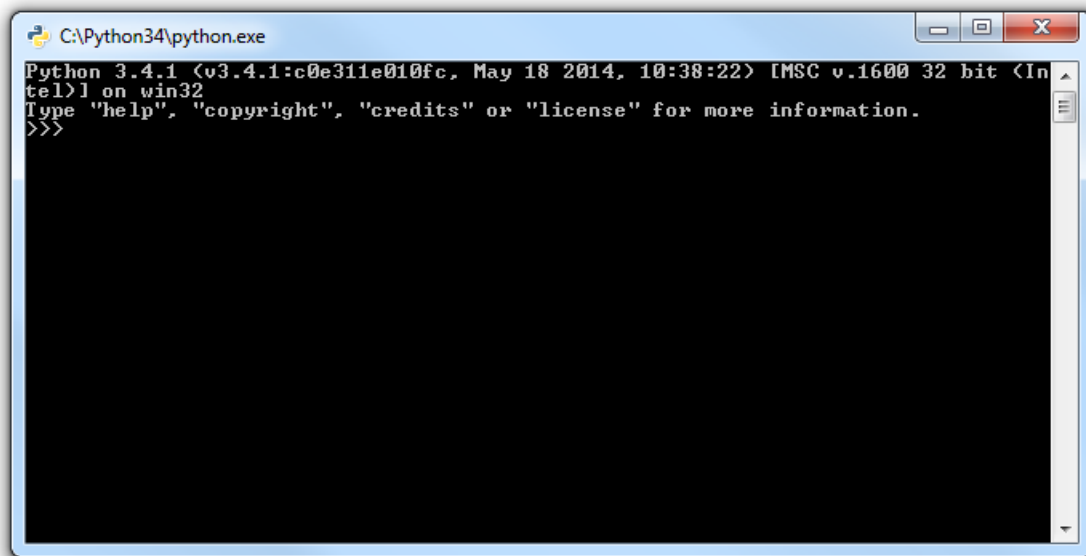
11. <https://www.youtube.com/watch?v=exwWzZ5VhQI>

3.4.1 Sous Windows

Vous disposez de plusieurs moyens pour accéder à la ligne de commande Python.

Méthode 1

Cette méthode est la plus simple ! Il vous faut passer par les menus Démarrer > Tous les programmes > Python 3.4 > Python (command line). Vous devriez voir cette sublime console :



Ligne de commande Python

Méthode 2

Il faudra premièrement utiliser le raccourci clavier Windows + R ou si vous préférez passer par les menus Démarrer > Tous les programmes > Accessoires > Exécuter.

Dans la fenêtre qui s'affiche, tapez tout simplement python afin de lancer la ligne de commande Python.

Choisissez la méthode qui vous plaira et comme nous venons de le voir tout chemin mène à Rome :).

3.4.2 Sous Linux

Une fois installé, un lien vers l'interpréteur Python a été normalement créé. Ce dernier aura comme libellé **python3.X** (où X représente le numéro de la version installée). Si, par exemple, vous avez installé Python 3.4, vous pourrez y accéder grâce à la commande `python3.4` comme présenté ci-dessous :

```
1 $ python3.4
2 Python 3.4.1 (v3.4.1:c0e311e010fc, Lun 11 2014, 21:34:56)
3 [GCC 4.3.3] on linux2
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>>
```

Pour fermer la ligne de commande, il vous suffira d'utiliser la combinaison clavier CTRL + D.

3.4.3 Sous Mac OS X

Dans le dossier Applications, recherchez un répertoire nommé Python. Une fois que vous l'avez trouvé, il suffira d'ouvrir l'application IDLE présente dans ce répertoire afin de lancer Python.

3.5 Résumé

Dans ce chapitre, nous avons pu voir que quelque soit le système d'exploitation utilisé, installer Python est chose facile !

4. Notre premier jeu

4.1 Le principe du jeu

Le jeu que nous allons concevoir dans ce chapitre est extrêmement simple. Comme je vous l'avais dit, nous allons commencer en douceur. Eh bien, c'est de cette douceur dont il était question :).

Ce que nous allons faire, c'est demander à l'utilisateur d'entrer son nom, puis afficher "Mama Miya" suivi de son nom.

Un exemple d'exécution de notre programme serait le suivant :

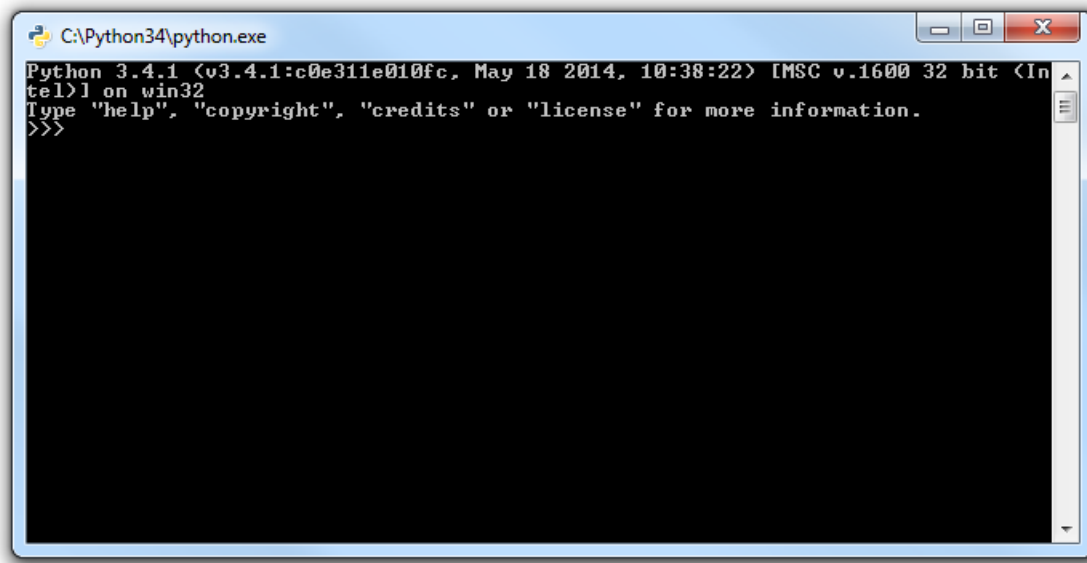
```
1 >>>
2 Entrer votre nom:
3 Honore Hounwanou
4 Mama Miya Honore Hounwanou
5 >>>
```

Alors bien avant de se lancer à tête baissée dans l'écriture de notre script, il va nous falloir répondre dans un premier temps à un bon nombre de questions.

Comment afficher quelque chose à l'écran ? Comment recueillir une valeur provenant de l'utilisateur ? Comment conserver une valeur en mémoire ?

Pour ce faire, ouvrez rapidement votre console d'interprétation de Python si ce n'est pas déjà fait !

Vous l'avez oubliée ? Je veux parler de cette fameuse console que nous avons vu dans le chapitre précédent :



Ligne de commandes Python

Amusez vous à présent à entrer des valeurs numériques puis valider :

```
1 >>> 2
2
3 >>> 9
4 9
5 >>> 14
6 14
7 >>>
```

Qu'est-ce que vous remarquez ? La valeur entrée vous est tout simplement retournée ! En effet, quel que soit ce que vous tapez la console d'interprétation essaie de vous fournir un résultat. Dans notre cas, elle nous retourne le nombre saisi vu qu'il n'y a quasiment rien à faire.

Ce que vous pouvez à présent faire, c'est utiliser l'interpréteur comme une simple calculatrice :

```
1 >>> 12 + 6
2 18
3 >>> 11 - 2
4 9
5 >>> 39 * 2
6 78
7 >>> 5 / 2
8 2.5
9 >>> 5 // 2
```

```

10 2
11 >>> 9 % 2
12 1
13 >>>

```

Vous connaissez sans doute les quatre opérations arithmétiques de base +, -, *, /. Mais c'est quoi ces mushibishis (choses) // et %.

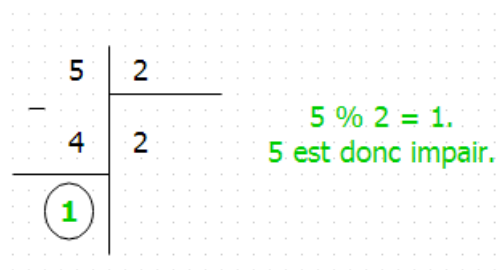
Le // vous retourne le résultat de la division entière entre deux nombres. Il n'y aura donc pas de valeur comportant une virgule vu que cette partie est supprimée. En gros que des entiers comme résultat.

```

1 >>> 12 // 5
2 2
3 >>> 15 // 2
4 7
5 >>> 9 // 2
6 4
7 >>>

```

Le % quant à lui est ce qu'on appelle l'opérateur **modulo**. Il vous retourne le reste de la division entière entre deux nombres. Il peut vous servir par exemple à déterminer la parité d'un nombre. En effet, un nombre pair s'il est divisible par 2. Ainsi si $5 \% 2 = 0$ alors 5 est pair, dans le cas contraire 5 est impair. Aussi simple que cela !



5	2
4	2
1	

5 % 2 = 1.
5 est donc impair.

L'opérateur modulo

4.2 Les chaines de caractères

Si vous aimez bidouiller, c'est sûr et certain que vous avez tenté d'entrer du texte. Et à votre grande surprise, vous avez eu une erreur du genre :

```

1 >>> toto
2 Traceback (most recent call last):
3   File "<pyshell#0>", line 1, in <module>
4     toto
5 NameError: name 'toto' is not defined
6 >>>

```

C'est tout à fait normal, car vous vous êtes un tout petit peu mal exprimé avec notre interpréteur. Et lorsque l'interpréteur ne comprend pas ce que vous lui demandez de faire, il vous le dit sans tourner autour du pot (pas comme certains ami(e)s :)). Il vous fournit en outre quelques détails lorsqu'il est de bonne humeur (Il est toujours de bonne humeur :)). Ici, il nous indique que le nom "toto" n'a pas été défini.



Mais c'est en Anglais ?

Ouh c'est vrai ! Je ne l'avais même pas remarqué ! Je crois que nous devons commencer à nous y habituer car à ce qu'il paraît ce sera comme ça tout le temps :).

Si vous voulez afficher du texte, il faudra le dire clairement à l'interpréteur en délimitant votre texte par des simples, doubles ou triples quotes comme ceci :

```

1 >>> 'toto'
2 'toto'
3 >>> "toto"
4 'toto'
5 >>> """je peux
6 utiliser plusieurs
7 lignes :)"""
8 'je peux\nutiliser plusieurs\nlignes :)'
9 >>>

```



C'est quoi le \n ?

Il symbolise tout simplement un retour à la ligne.

Les quotes permettent à l'interpréteur de savoir que vous voulez afficher du texte et non une variable. En effet, si vous ne mettez pas les quotes, il se dira qu'il doit rechercher un élément (variable, fonction, classe...) ayant comme nom ce que vous avez entré, et vu que cet élément n'existe pas pour l'instant, c'est la raison pour laquelle on avait cette erreur. Ainsi dans notre cas précédent, il a recherché par exemple une variable qui avait pour nom **toto** (qui n'existe bien sûr pas pour le moment).

Alors comme vous pouvez le voir, programmer en Python c'est simple. Il faudra apprendre la langue que comprend notre interpréteur afin de ne jamais avoir à nous bagarrer et le tour sera joué !

4.3 Les variables

C'est quoi une variable ? C'est vrai que j'ai employé ce terme dans la section précédente, sans pour autant l'expliquer.

Les variables vont nous permettre de pouvoir stocker des valeurs de manière **temporaire** au niveau de la mémoire vive aussi appelée mémoire système ou mémoire RAM(Random Access Memory). La mémoire RAM est une mémoire **volatile**, c'est-à-dire que dès que votre ordinateur cesse d'être alimenté en électricité, le contenu de cette mémoire est automatiquement supprimé, ce qui implique donc une perte totale des données mémoire. Ne comptez donc pas sur les variables pour stocker des valeurs (le nom de vos utilisateurs, leur mot de passe...) de manière permanente. Pour cela, il vous faudra utiliser un autre type de mémoire dit **non volatile** (Ex : le disque dur). Vous pourrez donc stocker des informations censées être permanentes au niveau d'un fichier, d'une base de données... Mais vous savez quoi ? Nous n'en sommes pas encore là pour l'instant.

Par rapport à ce qui a été dit plus haut, vous pouvez vous demander quel est alors l'intérêt d'utiliser une variable si après avoir éteint votre ordinateur elle ne sera plus disponible au prochain démarrage ? La réponse à cette question est très simple. Les variables vont nous permettre de stocker des valeurs de manière **temporaire** et même les changer tout au long de l'exécution de notre programme.

Prenons un exemple pour que tout soit clair dans votre esprit. Pour notre jeu que nous souhaitons réaliser a-t-on réellement besoin de stocker le nom de l'utilisateur dans un fichier ou dans une base de données ? Pas du tout, pourquoi se compliquer la vie vu qu'après avoir affiché "Mama Miya" suivi du nom de l'utilisateur on ne fera plus rien. Le programme s'arrête donc là !

Si par contre, on souhaitait demander une seule fois à l'utilisateur d'entrer son nom, puis afficher ce dernier précédé de "Mama Miya" à chaque fois qu'il exécutera notre programme sans avoir à le lui redemander même s'il redémarre son ordinateur, dans ce cas là, on pourrait penser à stocker son nom au niveau d'un fichier par exemple. En effet, vu que le contenu de la mémoire RAM sera supprimé lorsque l'ordinateur ne sera plus alimenté en électricité, nous n'aurons aucun autre moyen de récupérer son nom que de le lui redemander. J'espère que tout est à présent clair dans votre esprit.

Pour ceux qui sont fans de théorie, sachez également qu'une mémoire ne peut pas être à la fois rapide d'accès et grande en capacité. En d'autres termes, si elle est rapide d'accès, cela signifie qu'elle sera d'une petite taille. De même si elle est de grande capacité, elle sera un tout petit peu lente en accès.

La mémoire du disque dur est d'une grande capacité, on a même des Teras = 1000Go de nos jours pour des ordinateurs personnels. Ainsi, il faut donc par la même occasion savoir que pour y accéder au travers d'un programme cela va demander un peu plus de temps que si on voulait accéder à une donnée présente au niveau de la mémoire RAM car cette dernière n'a pas une très grande capacité, ce qui signifiait donc qu'elle est ... d'accès. (Je vous laisse compléter les pointillés si vous avez compris).

Mais la rapidité à laquelle je fais allusion n'est pas vraiment remarquable pour un être humain. Par contre, pour des programmes de grande envergure, il faudrait que vous sachez clairement où et comment stocker vos données afin d'avoir un programme qui s'exécute le plus rapidement possible.

Sachez pour terminer que cette caractéristique de volatilité de la mémoire RAM a tendance à disparaître avec les dernières évolutions technologiques conduisant à des types de mémoire RAM non-volatile, comme les [MRAM](#)¹².

Je vois qu'on se perd un tout petit peu. Revenons à nos moutons :).

4.3.1 Déclarer une variable

Je me suis lancé dans l'écriture d'un mini roman sans le savoir dans la section précédente :). Et avec tout ça, savons-nous au moins comment déclarer une variable ?

Rectifions donc le tir ! Une variable se déclare comme ceci en Python :

```
1 >>> age_toto = 21
2 >>> toto_birthday = "16/05/1993"
3 >>> 1tata = 3
4 SyntaxError: invalid syntax
5 >>>
```

Il faudra premièrement mettre le **nom de votre variable**, suivi de l'égalité "=" puis d'une **valeur** d'initialisation. Dans notre figure précédente, nous avons donc créer deux variables : `age_toto` et `toto_birthday`. En jargon informatique, on parle de déclaration de variables. Nous avons donc déclarer deux variables `age_toto` et `toto_birthday` auxquelles nous avons respectivement affecté comme valeur l'entier 21 et la chaîne de caractères "16/05/1993".



CamelCase vs snake_case

Vous avez peut-être remarqué que pour nos noms de variable, j'ai eu à séparer chaque nouveau mot par un underscore ou blanc souligné (_). C'est ce qu'on appelle la notion **snake_case** et c'est cette notation qui est majoritairement adoptée par la plupart des développeurs Python. Ex : `nom_de_mon_pere`. Vous retrouverez également dans l'industrie (par exemple en Java), la notation **CamelCase** où chaque nouveau mot après le premier mot commencera par une lettre majuscule. Ex : `nomDeMonPere`. Pour en apprendre davantage, vous pouvez visiter cette page [PEP 0008 - Style Guide for Python Code](#)¹³.

Vous l'avez peut être remarqué, notre interpréteur n'a pas du tout aimé la déclaration de la troisième variable, résultat, il nous a affiché une belle erreur !

Ce que je ne vous ai pas dit, c'est que le nom d'une variable ne doit pas commencer par un chiffre, il doit forcément commencer par une lettre et être suivi par n'importe quel chiffre, lettre, underscore mais pas d'espaces. Je vous recommande fortement d'éviter d'utiliser les caractères accentués comme é,à,è... dans vos noms de variables même si cela ne génère pas forcément des erreurs...

12. https://fr.wikipedia.org/wiki/Magnetic_Random_Access_Memory

13. <https://www.python.org/dev/peps/pep-0008/>

```

1 >>> nom_de_mon_père = "Léonard" #A éviter
2 >>> nom_de_mon_pere = "Léonard" #Beaucoup mieux
3 >>> nom_de_ma_mere = "Reine" #Nom de variable valide
4 >>>

```

Comme vous pouvez le voir, je peux utiliser mes caractères accentués au niveau de ma chaîne de caractères, aucun souci à ce niveau là vu que ce n'est que du texte. Pour exemple, j'ai écrit ici "Léonard".

Au niveau des noms de variables, cela fonctionne également mais comme dit précédemment, je vous conseille de les éviter lors de l'écriture de vos noms de variables afin de ne pas avoir de mauvaises surprises.



Les constantes

Une constante est une variable dont la valeur ne sera pas amenée à changer tout au long de l'exécution d'un programme. Dans d'autres langages de programmation, vous avez la possibilité de déclarer explicitement une constante mais ce n'est pas le cas en python. Ainsi comme ruse, les programmeurs python choisissent d'écrire le nom de la variable en majuscules histoire de savoir plus tard qu'il s'agit d'une constante. Voici donc quelques déclarations de constantes valides : `NOMBRE_DE_VIES = 5`, `LARGEUR_FENETRE = 300`... Je le répète : ce n'est pas une obligation, mais je vous conseille de toujours respecter cette convention.

Autre chose, j'ai eu à mettre `#A éviter` et `#Beaucoup mieux` qu'est-ce c'est ?

4.4 Les commentaires

Les commentaires vont nous permettre d'apporter un tout petit peu de documentation à notre code source. En grosso modo, on se sert des commentaires pour expliquer nos différents blocs de code.

Tout ce qui sera précédé d'un `#` sera considéré par l'interpréteur Python comme un commentaire.

Profitez-en c'est gratuit, mais n'en abusez surtout pas comme suit :

```

1 >>> nom_de_mon_pere = "Léonard" #Je déclare une variable nom_de_mon_pere qui a p\
2 our contenu "Léonard"
3 >>> nom_de_ma_mere = "Reine" #Je déclare une variable nom_de_ma_mere qui a pour \
4 contenu "Reine"
5 >>>

```



Python est sensible à la casse

Sachez que Python fait une distinction entre les lettres majuscules et minuscules. On dit dans ce cas, qu'il est sensible à la casse. Ainsi les variables `toto`, `ToTo`, `TOTTo` et `TOTO` sont toutes différentes les unes des autres.

4.5 Passons au code source !

Nous avons à présent toutes les cartes en main pour écrire notre jeu. Nous allons pour ce faire créer un fichier **jeu1.py**. Tous les fichiers python doivent avoir pour extension **py** ou **pyw**. Contentons nous de l'extension **py** pour l'instant.

Pour écrire votre code source, vous pouvez utiliser n'importe lequel des éditeurs de texte. Le Bloc Notes de Windows fera aussi l'affaire.

4.5.1 Choisir un environnement de développement intégré

Dans ce livre, nous utiliserons IDLE (Integrated DeveLopment Environment) qui est un environnement de développement intégré Python. Vous n'aurez rien à installer, vu qu'il vient par défaut lorsque vous installez Python.

Pour ceux qui utilisent une distribution Linux, vous pourrez écrire votre code source avec votre éditeur favori puis l'exécuter en ligne de commande comme suit :

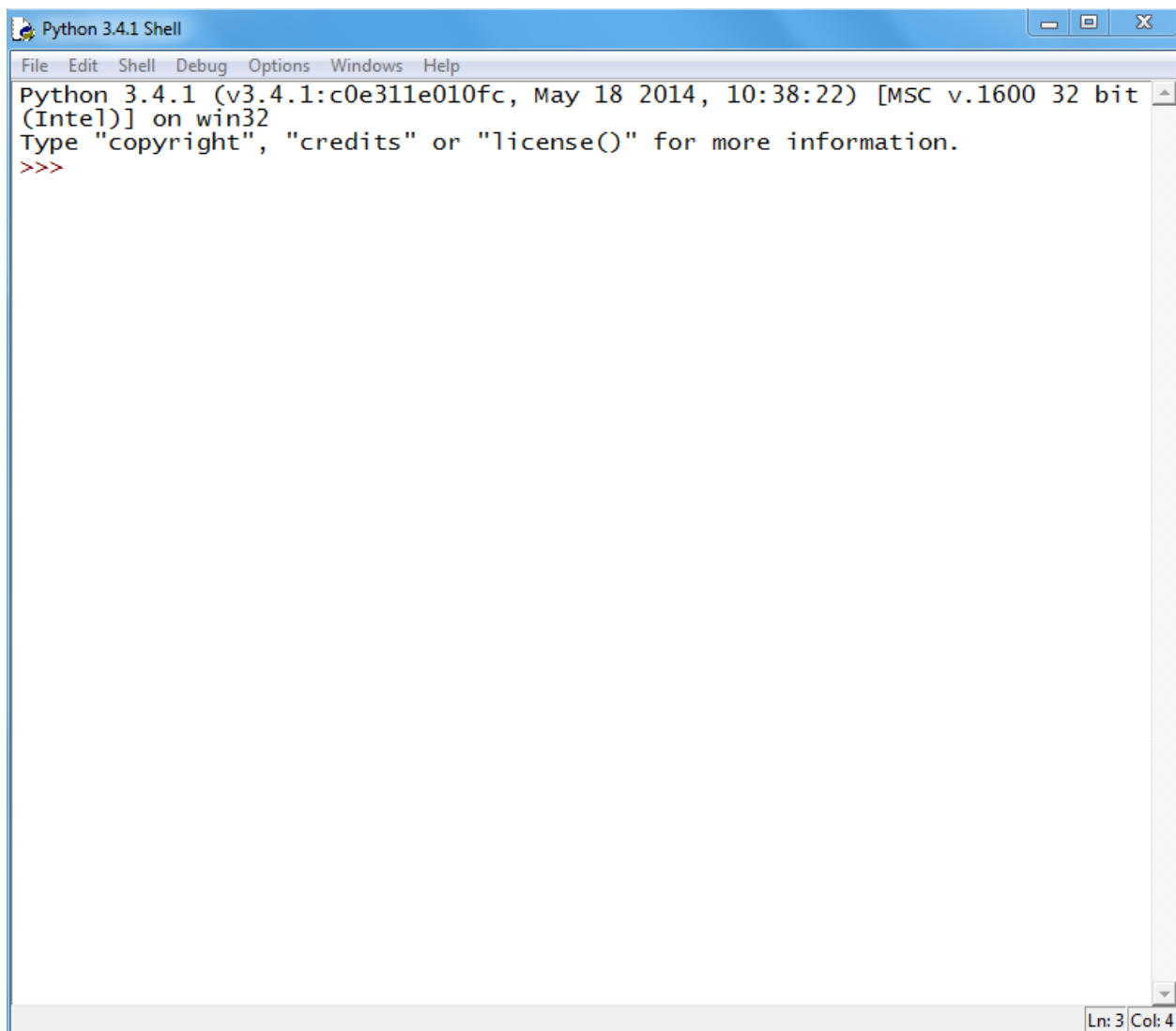
```
1 $ python3.4 nom_du_fichier.py
```

Pensez à être dans le dossier contenant votre fichier Python ! Vous pouvez également installer IDLE si cela vous intéresse. Une petite recherche sur Google et vous aurez le nom du paquet à installer.

4.5.2 Création du fichier jeu1.py

Il nous faudra premièrement ouvrir IDLE. Pour ce faire, on peut passer par les menus Démarrer > Tous les programmes > Python 3.4 > IDLE (Python GUI)

Vous devez normalement voir apparaître un truc de ce genre :



IDLE (Integrated DeveLopment Environment)

Pour créer ensuite notre script, il faudra passer par la barre de menus `File > New File` ou utiliser le raccourci clavier `CTRL + N`.

Mettez ensuite le code ci-dessous comme contenu de notre fichier puis enregistrer le par exemple sous le nom "jeu1.py". Veillez à ne pas inclure les numéros de ligne.


```
1  # Python par la pratique
2  # -----
3  # Notre premier jeu
4
5  print("Entrer votre nom : ")
6  nom = input()
7  print("Mama Miya " + nom)
```

Exécutez ensuite notre programme en cliquant sur Run > Run Module ou utilisez le raccourci clavier F5.

Notre programme doit normalement fonctionner sans problème.

```
1  >>>
2  Entrer votre nom:
3  Honore Hounwanou
4  Mama Miya Honore Hounwanou
5  >>>
```

Félicitations !! Vous avez écrit et exécuté votre premier programme Python. Wouhou :) !

4.5.3 Explication du code source

Je sais très bien qu'il y a des lignes dans notre code que vous ne comprenez pas, mais ne vous inquiétez surtout pas car je vais expliquer chacune de ces lignes. A partir de maintenant, ce ne sera que de la pratique. Je vous présenterai un challenge, le code qui ira avec et on essaiera de l'expliquer instruction après instruction.

```
1  # Python par la pratique
2  # -----
3  # Notre premier jeu
4
5  print("Entrer votre nom : ")
6  nom = input()
7  print("Mama Miya " + nom)
```

- Les lignes 1. à 3. ne sont rien d'autres que des commentaires. Elles nous permettent donc de documenter un temps soit peu notre code.
- La ligne 4. comme vous pouvez le voir est une ligne vide. Elle nous sert tout simplement à espacer notre code source afin de le rendre plus lisible.
- Au niveau de la ligne 5., nous utilisons la fonction **print**. Cette fonction permet d'afficher quelque chose au niveau de la console. Si je veux afficher **J'aime python.**, alors j'écrirai :

```
1 print("J'aime Python.")
```

Remarquez que là j'utilise les doubles quotes en lieu et place des simples quotes sinon j'aurai eu droit à une belle erreur. En effet si j'avais écrit : `python print('J'aime Python.')` L'interpréteur se dira que ma chaîne de caractère est 'J' et tout le reste sera incompréhensible pour lui. Si vous voulez tout de même utiliser les simples alors dans ce cas il va vous falloir échapper votre apostrophe en utilisant l'antislash encore appelé backslash afin que l'interpréteur ne la considère pas comme celle indiquant la fin de notre chaîne de caractères.

```
1 print('J\'aime Python.')
2 print("J'aime \"Python\".")
```

- Au niveau de la quatrième ligne, nous utilisons la fonction **input**, cette fonction permet de récupérer une valeur saisie par l'utilisateur. Nous récupérerons donc ce que l'utilisateur aura saisi et nous stockons cette valeur au niveau de la variable **nom**.
- Il ne reste plus qu'à afficher à présent, "Mama Miya" suivi du nom de l'utilisateur. Et c'est justement ce que nous faisons au niveau de cette ligne. Notez ici que j'ai eu à utiliser l'opérateur **+**. Lorsque vous faites appel à l'opérateur **+** avec comme opérandes des chaînes de caractères, ce qu'il fera c'est coller les deux chaînes de caractères. On parle de **concaténation**. Il va donc concaténer nos deux chaînes de caractères. Et voilà c'est tout !

Notez que je ne pouvais pas écrire :

```
1 print("Mama Miya nom")
```

Si je le faisais, il considérerait **nom** comme faisant partie de chaîne de caractères vu qu'il était dans la délimitation avec les doubles quotes. Soyez donc prudents !

Alors c'est tout pour ce chapitre, mais avant de terminer je vais vous montrer une autre manière d'écrire notre code. Notez que pour l'instant la fonction **input** n'a rien entre les parenthèses. Ce qu'on peut faire, c'est spécifier un message qui sera affiché avant de recueillir la valeur de l'utilisateur. On parle de *prompt*. Ainsi nous n'aurons plus besoin d'utiliser ici la fonction **print**.

```
1 # Python par la pratique
2 # -----
3 # Notre premier jeu
4
5 nom = input("Entrer votre nom : ")
6 print("Mama Miya " + nom)
```

Pour terminer, voici encore une autre manière d'écrire le même script. Je vous laisse comme des grands me dire ce qui se passe ici. Rien de vraiment compliqué !

```

1  # Python par la pratique
2  # -----
3  # Notre premier jeu
4
5  print("Mama Miya " + input("Entrer votre nom : "))

```

Allez, je vous l'explique. Vous l'avez mérité :) !

Notre programme ne comprend qu'une seule ligne de code et cette ligne demande à afficher "Mama Miya " + input("Entrer votre nom : ").

L'interpréteur Python verra qu'il peut afficher "Mama Miya" mais par contre pour afficher l'autre partie il faudra demander à l'utilisateur de saisir une valeur au travers de la fonction **input**. La fonction *input* entre donc en action, elle recueille le nom de l'utilisateur tout en s'assurant d'afficher bien avant le prompt "Entrer votre nom :".

Le nom étant maintenant récupéré, on pourra ensuite le concaténer à "Mama Miya". Et voilà !

Bien vrai que ces trois codes produisent le même résultat à l'exécution, j'ai tendance ici à préférer le second qui me semble plus clair et lisible ce qui n'est pas forcément le cas du troisième script.

```

1  # Python par la pratique
2  # -----
3  # Notre premier jeu
4
5  nom = input("Entrer votre nom : ")
6  print("Mama Miya " + nom)

```

4.6 Résumé

Dans ce chapitre, nous avons appris que :

- L'interpréteur Python est très convivial et peut être utilisé comme une mini-calculatrice.
- Le modulo est un opérateur permettant d'obtenir le reste de la division entière entre deux nombres.
- En programmation, en lieu et place de dire "texte", on parle très souvent de chaînes de caractères.
- Le caractère `\n` représente un retour à la ligne.
- Les variables vont nous permettre de pouvoir stocker des valeurs de manière temporaire au niveau de la mémoire RAM.
- La mémoire RAM est une mémoire volatile, c'est-à-dire que dès que notre ordinateur cesse d'être alimenté en électricité, le contenu de cette mémoire est automatiquement supprimée.

- Pour déclarer une variable en Python, il vous faut tout d'abord mettre le nom de votre variable, ensuite l'égalité "=" et pour terminer une valeur d'initialisation.
- Une constante est une variable dont la valeur ne sera pas amenée à changer tout au long de l'exécution d'un programme.
- Par convention, les noms des constantes sont écrits en majuscules.
- Les commentaires nous servent à documenter notre code source.
- Python est un langage sensible à la casse. C'est-à-dire qu'il fait une distinction entre les lettres majuscules et minuscules.
- IDLE est un environnement de développement intégré Python.
- La fonction print permet d'afficher quelque chose au niveau de la console.
- La fonction input permet de récupérer une valeur saisie par l'utilisateur.
- La concaténation est l'opération consistant à coller bout à bout deux chaînes de caractères.
- Il existe toujours plusieurs solutions à un problème donné.

5. Les fonctions

Un bon programmeur, ce n'est pas celui qui écrit plusieurs milliers de lignes de code, mais en réalité c'est celui qui en écrit le moins et dont le résultat parle de lui-même. Car en général ce n'est pas la longueur du programme qui importe mais plutôt les fonctionnalités de ce dernier. Alors la question à se poser c'est comment faire pour écrire moins tout en produisant plus ? La réponse à cette interrogation est le concept de **modularité**.

Dans ce chapitre, nous parlerons des **fonctions**. Cela nous permettra d'introduire en douceur ce nouveau concept. Nous reparlerons un peu plus en profondeur de la **modularité** au niveau du [chapitre 9](#).

5.1 Objectifs de la modularité

Les deux objectifs principaux de la modularité sont :

- **La décomposition** qui consiste à créer une bonne structuration de notre code source (par exemple en créant des modules qui peuvent être des fonctions, des classes...). Les modules ont l'avantage d'avoir leur propre contenu et d'être réutilisable dans un autre programme.
- **L'abstraction** qui permet de supprimer les détails. On ne s'occupe pas forcément de comment le code fonctionne mais on l'utilise tout simplement. Par exemple, lorsque nous avons utilisé l'opération de concaténation, nous ne nous sommes pas souciés de savoir comment est-ce que Python réalisait cette opération, nous l'avons tout simplement utilisée.

5.2 Les fonctions

Les fonctions représentent la forme de modularité la plus simple en Python. Elles vont nous permettre de regrouper un ensemble d'instructions jouant en quelque sorte le rôle de petits programmes autonomes, effectuant une tâche spécifique et que nous pourrons par la suite intégrer dans notre programme principal. Après avoir créé une fonction, nous pourrons l'utiliser à tout moment et à n'importe quel emplacement dans notre programme. Cela nous fera gagner énormément de temps et aussi de l'énergie vu que nous n'aurons pas à réexpliquer à l'ordinateur une vingtaine de fois ce qu'il est censé faire.

Prenons un exemple simple. Supposons que nous sommes en 2030 et vous êtes devenu le développeur Python par excellence de la planète Mars en réussissant à faire en sorte que l'ordinateur puisse produire du café et de surcroît du bon café :).

Afin de ne pas très tôt dévoiler votre algorithme miraculeux (et aussi parce que je ne sais pas faire du café), nous allons nous contenter de décrire les différentes étapes de la conception de votre café par étape1, étape2, étape3 et ainsi de suite.

Ainsi si je veux faire du café, je n'aurai qu'à dire :

- Etape 1
- Etape 2
- Etape 3
- ...

Mais avouez que cela va devenir très vite énervant si je dois à chaque fois dire à l'ordinateur :

- Etape 1
- Etape 2
- Etape 3
- ...

Et c'est là que les fonctions font leur entrée en fanfare !

Grâce aux fonctions, vous aurez la possibilité de regrouper cet ensemble d'instructions (on parle très souvent de bloc d'instructions) et l'appeler autant de fois que vous le souhaitez.

Je pourrai donc créer une fonction `faire_du_cafe` qui se présentera comme suit :

```
1 def faire_du_cafe():  
2     Etape 1  
3     Etape 2  
4     Etape 3  
5     ...
```

et le jour où j'aurai besoin d'avoir du café, je n'aurai qu'à dire au niveau de mon programme :

```
1 faire_du_cafe()
```

et j'aurais comme par magie ceci :



Café produit en 2030 à partir de votre merveilleux algorithme

Si vous avez compris tout ce qui a été dit plus haut, alors c'est que vous avez compris l'utilité des fonctions. Voyons à présent plus sérieusement comment créer une fonction en Python et comment l'utiliser.

5.2.1 Création d'une fonction

```
1 def nom_de_la_fonction(parametre1, parametre2, ... , parametren):
2     """Docstring de la fonction"""
3
4     Instruction
5     Instruction
6     ...
```

5.2.2 Explication

On a dans l'ordre :

- Le mot-clé **def**, qui est l'abréviation de **define** (définir en anglais) et qui constitue le prélude à toute construction de fonction. En d'autres termes sans ce fameux **def** l'on ne saura pas qu'il s'agit d'une fonction.
- Ensuite vient le nom de la fonction. Les règles à suivre pour le choix du nom d'une variable sont les mêmes que celles d'une fonction.
- Entre parenthèses sont listés les paramètres de la fonction. Nous y reviendrons, mais tout ce qu'il y a à savoir pour l'instant c'est que la liste des paramètres est délimitée par des parenthèses ouvrante et fermante et que les paramètres sont séparés les uns des autres par des virgules.
- Viennent ensuite **les deux points** : qui terminent la ligne. Ils sont **très très très** importants ces deux points. Ils servent à délimiter notre bloc d'instructions (le bloc qui va permettre de dire ce que fera notre fonction).

Dans d'autres langages comme le langage C, les blocs d'instructions sont délimités par des accolades comme suit :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(int argc, char **argv){
5
6      char nom[30];
7      printf("Entrer votre nom : ");
8      scanf("%s", nom);
9      printf("Mama Miya %s", nom);
10
11     return 0;
12 }
```

Ce programme correspond à notre jeu du chapitre précédent traduit en langage C. Avouez que cela fait déjà un peu peur :).



Le langage C est nul alors ?

Oulala, ne pensez pas que le langage C est NUL. Loin de là ! Il est ce qu'on appelle un langage de bas niveau, ce qui fait qu'il se rapproche un peu plus de la machine à la différence de Python qui comme nous l'avons dit est un langage de haut niveau dans la mesure où ce dernier se rapproche beaucoup plus du langage humain. Les programmes en Python seront généralement plus courts et plus faciles à comprendre que ceux écrits en C et cela est tout à fait logique. Nous avons énormément de choses en Python qui nous sont facilitées (les listes, les tuples...). En langage C nous aurions eu à les coder par nous-mêmes. Pour fermer cette petite parenthèse, sachez que bon nombre de langages de programmation comme Python ou encore Java ont été écrits en langage C. Ainsi, sans le langage C, Python n'aurait probablement jamais vu le jour, de même que ce livre :).

Voici par exemple la déclaration d'une fonction en langage C.

```
1  int main(int argc, char **argv){
2
3      char nom[30];
4      printf("Entrer votre nom : ");
5      scanf("%s", nom);
6      printf("Mama Miya %s", nom);
7
8      return 0;
9  }
```

Comme vous le voyez, la fonction **main** a son bloc d'instructions qui a été délimité par des accolades ouvrante et fermante et notez que chaque instruction se termine par un point virgule alors qu'en Python pour symboliser la fin d'une instruction il suffit tout simplement d'aller à la ligne suivante.

Guido van Rossum, le créateur du langage Python a trouvé plus judicieux d'utiliser les deux points et l'indentation pour délimiter les blocs d'instructions.

En effet ce code en C aurait aussi fonctionné :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  int main(int argc, char **argv){
4      char nom[30]; printf("Entrer votre nom : ");scanf("%s", nom);printf("Mama Miy\
5  a %s", nom);return EXIT_SUCCESS;}
```

Même si ce dernier produira le même résultat que le précédent, ce code est tout du moins illisible. Guido van Rossum s'est donc demandé comment forcer l'utilisateur à écrire un code propre et non un charabia comme celui présenté plus haut ? La réponse l'**indentation**.

5.2.3 L'indentation

Si j'écris ce code en Python,

```
1  def faire_du_cafe():
2      print("Etape 1") print("Etape 2") print("Etape 3")
```

cela ne va tout simplement pas fonctionner. La raison est toute simple. Python utilise le retour à la ligne pour symboliser la fin d'une instruction. Ainsi il ne saura jamais où est-ce que se termine la première instruction, pour ensuite se préoccuper de détecter le début de la seconde instruction !

Cela était possible en C parce que le point virgule symbolisait catégoriquement la fin d'une instruction.

Dans la même optique, vu que les blocs d'instructions en C sont délimités par des accolades, vous pouvez présenter votre bloc d'instructions comme vous le souhaitez. Le compilateur C saura que votre bloc d'instructions sera composé de tout ce qui se trouve à l'intérieur des accolades.

En Python par contre, ce sera différent et croyez-moi, c'est pour notre bien. Voilà comment nous devons procéder :

```
1  def dire_bonjour():
2      print("Bonjour Maman")
3      print("Bonjour Papa")
4      print("Bonjour Gloradie") #Ma grande sœur adorée :)
5
6  print("Je ne fais pas partie du bloc d'instructions.")
```

- Il faudra mettre premièrement mettre les deux points pour signifier que nous allons maintenant commencer à définir notre bloc d'instructions.

- et ensuite chaque instruction de notre bloc doit avoir le même nombre d'espaces à gauche, on parle **d'indentation**.

Exemple :

```
1 def dire_bonjour():
2     print("Bonjour Maman")
3     print("Bonjour Papa")
4     print("Bonjour Gloradie")
5
6 print("Je ne fais pas partie du bloc d'instructions")
```

Vous pouvez remarquer que notre bloc d'instructions au niveau de la fonction `dire_bonjour` est composé de trois instructions étant donné que ces dernières ont la même indentation. Vous pouvez laisser autant d'espaces que vous le souhaitez mais généralement 4 espaces par niveau d'indentation sont choisis par convention.

La dernière instruction quant à elle ne fait pas partie de notre bloc d'instructions vu qu'il n'y a pas d'indentation. Autrement dit, elle n'a pas la même indentation que les autres instructions.

Si nous souhaitons plus tard l'ajouter au contenu de notre fonction (on dit plutôt corps de la fonction), il va falloir alors revoir le niveau d'indentation comme ceci :

```
1 def dire_bonjour():
2     print("Bonjour Maman")
3     print("Bonjour Papa")
4     print("Bonjour Gloradie")
5
6     print("Je fais maintenant partie du bloc d'instructions :)")
```

Gardez en tête ce principe d'indentation pour la gestion des blocs d'instructions car il sera également de la partie lorsque nous aborderons très bientôt les notions de conditions, boucles...

5.2.4 Définition d'une fonction avec des paramètres

Jusque là notre fonction `dire_bonjour()` nous permettait de ne saluer que les membres de notre famille. Il serait vraiment poli de pouvoir également saluer les personnes que nous rencontrerons dans la rue. N'ai-je pas raison ?

Pour cela, notre fonction aura besoin de ce qu'on appelle des paramètres. Un paramètre c'est tout simplement un ingrédient que vous pourrez donner à votre fonction afin que cette dernière puisse vous concocter quelque chose de beaucoup plus délicieux. Vous pourrez donc ajouter autant de paramètres que vous le souhaitez. En gros, c'est un peu comme une sauce, on peut y ajouter du piment, du sel, du sucre :)

Voici donc notre nouvelle fonction avec cette fois-ci un paramètre nom:

```
1 def dire_bonjour_a(nom):  
2     """ Permet de saluer la personne ayant le nom donné en paramètre """  
3     print("Salut " + nom)
```

Remarquez que premièrement, j'ai eu à changer le nom de notre fonction en ajoutant un petit a à la fin afin d'avoir quelque chose de beaucoup plus commode. Rappelez vous que le nom de votre fonction doit refléter au mieux ce que fait cette dernière.

Une fois que notre fonction récupère le paramètre nom, elle n'aura qu'à afficher "Salut " concaténé au nom qui sera passé en argument. Notez ce petit espace après le t de Salut afin de ne pas avoir le nom accolé au mot Salut .

Maintenant que notre fonction a été à présent créée, voyons voir maintenant comment l'utiliser, car il serait vraiment triste de créer une fonction qui ne sera jamais utilisée.

En lieu et place de dire utiliser une fonction, les programmeurs vous diront appeler une fonction. Tachez donc de vous en souvenir histoire d'avoir une place dans les discussions de tordus.

Pour faire simple, nous avons deux grandes étapes en ce qui concerne les fonctions :

- La définition de la fonction (Création de la fonction)
- L'appel de la fonction (Utilisation de la fonction)

Nous avons donc défini notre fonction `dire_bonjour_a()`, voyons à présent comment l'utiliser !

```
1 >>> dire_bonjour_a("Honore")  
2 Salut Honore  
3 >>> dire_bonjour_a("Armand")  
4 Salut Armand  
5 >>> dire_bonjour_a("Cynthia")  
6 Salut Cynthia  
7 >>> dire_bonjour_a(2)  
8 Traceback (most recent call last):  
9   File "<pyshell#11>", line 1, in <module>  
10     dire_bonjour_a(2)  
11   File "<pyshell#7>", line 2, in dire_bonjour_a  
12     print("Salut " + nom)  
13 TypeError: cannot concatenate 'str' and 'int' objects  
14 >>>
```

Pour appeler une fonction, comme vous pouvez le remarquer, c'est vraiment très simple. Il suffit de mettre le nom de la fonction, ici `dire_bonjour_a`, et ensuite lui fournir les différents arguments.



C'est quoi encore un argument ?

Lorsqu'on définit une fonction, les ingrédients ajoutés entre parenthèses sont appelés **paramètres**. Par contre, lors de l'appel de la fonction, on parle plutôt d'**arguments**. C'est aussi simple que cela.

Les exemples précédents `dire_bonjour_a("Honore")`, `dire_bonjour_a("Cynthia")` ... fonctionnent à la perfection mais ceci n'est pas le cas lorsque je passe cette fois-ci un 2 en argument. Pourquoi ?

Ne pensez surtout pas que Python est assez intelligent pour deviner que 2 n'est pas un nom. J'avoue que cela aurait été super cool, mais malheureusement ce n'est pas le cas.

Là nous avons tout simplement une erreur de sémantique statique. Pour ceux qui ont eu à lire le [chapitre 2](#), nous avons dit que la sémantique statique visait à vérifier que les expressions ayant une bonne syntaxe ont une **signification**. Vu que nous avons passé 2 en argument, notre expression n'a donc aucune signification ! De quoi je parle ? Voyons cela pas à pas.

Lorsque nous appelons notre fonction en mettant `dire_bonjour_a(2)`, qu'est-ce qui se passe ? Notre fonction `dire_bonjour_a` est appelée. Notre paramètre **nom** aura donc comme valeur 2 vu que c'est ce fameux deux (2) qui a été passé en argument.

Ainsi on a donc notre fonction qui devra exécuter cette instruction :

```
1 print("Salut " + 2)
```

Pour vous `"Salut " + 2`, qu'est-ce que cela signifie ? Saluer avec les deux mains peut être :). Mais pour notre interpréteur Python cela ne signifie rien. Il est donc un peu confus. Et comme je vous l'ai dit, lorsque l'interpréteur ne comprend pas quelque chose il vous le dit clairement. Résultat on a ceci :

```
1 Traceback (most recent call last):
2   File "<pyshell#11>", line 1, in <module>
3     dire_bonjour_a(2)
4   File "<pyshell#7>", line 2, in dire_bonjour_a
5     print("Salut " + nom)
6   TypeError: cannot concatenate 'str' and 'int' objects
```

Dans tout ce charabia, ce qui nous intéresse le plus, c'est la dernière ligne.

```
1   TypeError: cannot concatenate 'str' and 'int' objects
```

L'interpréteur nous dit ici qu'il ne peut pas concaténer une chaîne de caractères et un entier. `str` est le diminutif de `string` qui signifie chaîne de caractères et `int` est le diminutif de `integer` qui signifie entier.

Ainsi pour l'instant, comme vous pouvez le voir, si ce n'est pas une chaîne de caractères qui est passée en argument, nous aura droit à coup sûr à une belle erreur.

Petite confidence 2 est le nom d'un ami Japonais **Yung 2** que j'ai eu à rencontrer lors d'un séminaire au Japon. Il a été vraiment surpris que je ne l'ai pas salué lorsqu'on s'est revus hier au restaurant. Ce qu'il ne sait pas, c'est que ce n'était pas de ma faute, mais celle de notre fonction.



Qui l'a écrite cette fonction ?

Vous me laissez tomber ? Ce n'est vraiment pas gentil de votre part :)

Je compte revoir **Yung 2** demain. Il faut donc que notre fonction puisse être en mesure de le saluer afin d'apaiser la tension qui règne désormais entre nous.

Première solution et plutôt simple, serait de transformer 2 en "2". Le premier 2 est un entier (un **int**), le second quant à lui est une chaîne de caractères (**string** ou en plus court **str**).

La preuve :

```
1 >>> type(2)
2 <type 'int'>
3 >>> type("2")
4 <type 'str'>
5 >>>
```

La fonction **type** vous retourne le type de l'objet passé en argument.

On aurait pu aussi avoir le type en utilisant l'attribut `__class__`:

```
1 >>> (2).__class__
2 <type 'int'>
3 >>> "2).__class__
4 <type 'str'>
5 >>>
```

Regardez le code suivant et essayez de deviner ce que cet attribut `__doc__` permet de faire :)

```

1 def dire_bonjour_a(nom):
2     """Permet de saluer la personne ayant le nom donné en paramètre"""
3     print("Salut " + nom)
4 >>> dire_bonjour_a.__doc__
5 'Permet de saluer la personne ayant le nom donn\x3\xa9 en param\x3\xa8tre'
6 >>>

```

J'ai tendance à appeler Python le langage des underscores. J'espère que vous comprendrez maintenant pourquoi.

Trêve de bavardage ! Ce que je veux dire c'est qu'on aurait pu appeler notre fonction comme suit :

```

1 >>> dire_bonjour_a("2")
2 Salut 2

```

Cela fonctionne, je suis d'accord. Mais le fait est que je veuille garder l'entier 2 car Yung m'a dit que cela fait partie de leur culture et il faudrait être vraiment fou pour porter atteinte à la culture d'un japonais n'est-ce pas ?

Vous allez voir, ce sera très simple ! Nous allons utiliser ce qu'on appelle **le cast encore appelé casting ou encore transtypage ou même conversion de types**. Tous ces mots pour signifier la même chose.

Le *casting* consiste à convertir un objet d'un type donné en un autre. Ainsi grâce au casting, vous aurez la possibilité de convertir un entier en une chaîne de caractères, un nombre réel en un entier.... Passons sans plus tarder à une petite phase pratique pour ainsi sortir de cette atmosphère d'abstraction.

```

1 >>> int("3") #Convertir la chaîne de caractères "3" en entier
2 3
3 >>> int(3.0) #Convertir le réel 3.0 en entier
4 3
5 >>> int(3.5) #Convertir le réel 3.5 en entier
6 3
7 >>> str(3) #Convertir l'entier 3 en chaîne de caractères
8 '3'
9 >>> int('123') #Convertir la chaîne de caractères '123' en entier
10 123
11 >>> int('maman')
12
13 Traceback (most recent call last):
14   File "<pyshell#6>", line 1, in <module>
15     int('maman')

```

```

16 ValueError: invalid literal for int() with base 10: 'maman'
17 >>> float(3)
18 3.0
19 >>> int('123.09')
20
21 Traceback (most recent call last):
22   File "<pyshell#11>", line 1, in <module>
23     int('123.09')
24 ValueError: invalid literal for int() with base 10: '123.09'

```

Grâce à ces exemples vous avez pu remarquer qu'appliquer le principe du casting est vraiment très simple en Python. La syntaxe est la suivante : `python nouveau_type(valeur)` La conversion d'un nombre réel en un entier consiste à opérer tout simplement une troncature. 23.45 deviendra donc 23 en entier. Toute la partie après la virgule est supprimée. Nous verrons après comment procéder à des arrondis lorsque nous verrons un peu plus en détails les modules built-in Python.

Vous devez normalement avoir des questions par rapport aux deux erreurs que nous avons eu. La première erreur est tout à fait normale, convertir "maman" en entier, je me demande bien ce que j'aurais donné comme réponse ?! Peut-être bien 5 vu que ma mère a 5 enfants :) (Je vous dis trop de choses sur moi dans ce livre hmmm). Cela nous permet de noter qu'il faudrait avoir un minimum de bon sens dans cette histoire de casting car on ne peut pas convertir n'importe quoi et n'importe comment.

La deuxième erreur par contre est un peu étrange je peux l'avouer. Nous disons là que nous souhaitons convertir la chaîne '123.09' en entier. Ça semble logique et ça l'est. Par contre comme vous pouvez le voir l'interpréteur lui ne comprend pas ce qu'on veut dire. Il va donc falloir le guider.

Pour ce faire, tentons de remplacer `int('123.09')` par `float('123.09')` et voyons voir ce que cela donne :

```

1 >>> float('123.09')
2 123.09
3 >>>

```

Parfait ! L'interpréteur comprend parfaitement ce que l'on veut si il s'agit d'une conversion vers un float (réel). Vous voyez maintenant comment on pourra résoudre notre problème ? Si si regardez on peut ...?

```

1 >>> int( float('123.09') )
2 123
3 >>>

```

J'avoue que c'est vraiment bizarre que cela ne soit pas disponible directement en Python sans avoir à passer par un casting intermédiaire. Mais bref, nous devons nous contenter des moyens du bord. Ainsi si un jour, il vous arrive de vouloir convertir un réel qui est représenté sous forme de chaîne de caractères en entier, vous savez dorénavant comment le faire.

5.2.5 Dire bonjour à Yung 2

Notre vrai problème était que nous puissions appeler la fonction `dire_bonjour_a` et lui passer en paramètre l'entier 2. Avec cette petite escapade sur le fonctionnement du casting que vous avez eu à faire, je peux vous assurer que vous êtes en mesure de pouvoir maintenant régler ce problème et sauver par la même occasion ma vie. Please guys, c'est un japonais :).

La solution que je vous propose est la suivante :

```
1 def dire_bonjour_a(nom):  
2     print("Salut " + str(nom) )
```

Et si on exécute notre fonctionne, tout fonctionne à la perfection.

```
1 >>> dire_bonjour_a(2)  
2 Salut 2  
3 >>> dire_bonjour_a("Honore")  
4 Salut Honore  
5 >>>
```

Paramètres avec valeurs par défaut

Dans la vie réelle, j'ai pour habitude d'appeler toutes les personnes dont je ne connais pas le nom **Boss**. Notre fonction devra donc réagir de la sorte :

- Si je lui donne un nom, alors elle devra affiche **Salut suivi du nom**
- Dans le cas contraire, elle devra affiche tout simplement **Salut Boss**

On peut réaliser cela très facilement en Python en donnant ce qu'on appelle une valeur par défaut à notre paramètre. Cela donne donc en code :

```
1 def dire_bonjour_a(nom = "Boss"):  
2     print("Salut " + str(nom) )
```

Notre fonction `dire_bonjour_a` a donc maintenant une valeur par défaut **"Boss"**. C'est en gros une simple initialisation de notre paramètre. Mais grâce à cette soit disant petite initialisation, je peux à présent appeler ma fonction `dire_bonjour_a` sans passer obligatoirement d'argument.


```

1 >>> dire_bonjour_a(2)
2 Salut 2
3 >>> dire_bonjour_a()
4 Salut Boss
5 >>>

```

Vous pouvez donner donc une valeur par défaut à un paramètre afin de le rendre optionnel.

5.2.6 Signature ou prototype d'une fonction

On entend par *signature de fonction*, les éléments qui permettent au langage d'identifier ladite fonction. En Python, comme vous avez pu le voir, on ne précise pas les types des paramètres. Ainsi en Python, la signature d'une fonction est tout simplement **le nom de la fonction**. Cela signifie que vous ne pouvez pas définir deux fonctions ayant le même nom. Si vous le faites, l'ancienne définition est écrasée par la nouvelle.

```

1 def carre():
2     """ Première fonction chargée d'afficher une figure """
3
4     print """
5         *****
6         *   *
7         *****"""
8
9     carre() #Affichera notre joli petit carré
10
11 def carre(nombre):
12     """ Seconde fonction écrasant la première et qui affiche le carré
13     d'un nombre passé en argument"""
14
15     resultat = nombre * nombre
16     print("Le carré de " + str(nombre) + " est " + str(resultat))
17
18 carre(2) #Affichera 4
19 carre(4) #Affichera 16
20 carre() #Affichera une erreur

```

Si on exécute notre programme, on obtient le résultat suivant :

```

1      *****
2      *      *
3      *****
4  Le carré de 2 est 4
5  Le carré de 4 est 16
6
7  Traceback (most recent call last):
8    File "C :/Users/honore.h/Desktop/toto.py", line 18, in <module>
9      carre()
10  TypeError: carre() takes exactly 1 argument (0 given)

```

Nous avons une erreur lorsqu'on appelle la fonction **carre()** sans paramètre vu que la seconde fonction a écrasé la première ainsi celle-ci n'est plus disponible. Ce qui fait que pour appeler la nouvelle fonction, il va falloir forcément passer un argument.



Afficher une figure

Vous avez pu remarquer qu'on pouvait afficher une figure en utilisant une chaîne de caractères délimitée par les triples guillemets :). Ce que vous mettrez entre les triples guillemets sera affiché tel quel.

5.2.7 L'instruction return

Très souvent une fonction ne sert pas uniquement à afficher quelque chose comme on l'a fait jusque-là. Une fonction peut être en effet utilisée pour retourner une valeur.

Imaginez que vous vous rendez dans un McDonald et que vous commandez un Big Mac. Et en lieu et place de vous remettre votre Big Mac, vous voyez à votre grande surprise un message s'afficher à l'écran : **Vous venez de manger un Big Mac. Merci et à très bientôt :)** (avec même de petits jeux de lumière autour de l'écran).

Serez-vous heureux ? Bien sûr que non :).

C'était juste une parenthèse. Ce que je veux dire, c'est que très souvent une fonction fera un travail et vous retournera une valeur. Pour ce faire, il faudra utiliser l'instruction **return**.

Un petit exemple pour boucler la boucle :

```

1 def est_pair(nombre):
2     """Renvoie True si nombre est pair
3     et False dans le cas contraire"""
4
5     if(nombre % 2 == 0):
6         return True
7     else:
8         return False

```

Le plus important ici pour moi, ce n'est pas que vous puissiez comprendre ce que fait cette fonction `est_pair` mais tout simplement que vous puissiez remarquer que l'on peut utiliser l'instruction `return` afin de retourner une valeur. Nous en reparlerons dans le chapitre suivant lorsque nous parlerons des conditions.

5.2.8 Arguments nommés

Supposons que je souhaite créer une fonction `maximum` qui prend 2 paramètres `nbre1` et `nbre2` et retourne le maximum de ces deux nombres. Pour ce faire, on va utiliser la fonction native de python `max` qui retourne la valeur maximale d'une séquence de valeurs qui lui est passée en argument. Il existe également la fonction `min` qui fait tout simplement le contraire de ce que fait la fonction `max`. Elle retourne donc la valeur minimale d'une séquence de valeurs qui lui est passée en argument.

Pour pimenter un peu les choses, on va également donner des valeurs par défaut à `nbre1` et `nbre2`, respectivement 0 et 100.

Notre fonction se présente donc comme suit :

```

1 def maximum(nb1 = 0, nb2 = 100):
2     return max(nb1, nb2)

```

On pourra donc appeler notre fonction comme suit :

```

1 maximum() #retournera 100
2 maximum(40, 300) #Retournera 300

```

Et si maintenant je veux uniquement préciser la valeur de `nbre2` ? Comment est-ce que je fais ?

En effet si je fais ceci

```

1 maximum(500)

```

Notre fonction va considérer `500` comme la valeur affectée à `nbre1` et `nbre2` aura comme valeur `100` (la valeur par défaut). Pour résoudre ce problème, il faudra utiliser ce qu'on appelle **un argument nommé**. Ce sera très simple, vous allez le voir. Qu'est-ce que je veux ? Préciser uniquement la valeur de `nbre2`. Eh bien je n'ai qu'à le dire à ma fonction comme suit :

```
1 maximum(nbre2 = 500)
```

Pour vous le prouver, modifions un tout petit peu notre fonction **maximum**:

```
1 def maximum(nbre1 = 0, nbre2 = 100):  
2     print("Nombre 1 :", nbre1)  
3     print("Nombre 2 :", nbre2)  
4  
5     return max(nbre1, nbre2)
```

Lorsque vous utilisez la fonction **print** en séparant les éléments par une virgule, elle se chargera d'afficher vos éléments séparés par un espace. De plus aucun casting n'est requis.

```
1 print("Salut", 2) #Aucune erreur ! Affichera Salut 2
```

Eh oui, vous pouvez me gronder pour vous avoir fatigué précédemment avec notre ami Yung 2 ;).

Appelons à nouveau notre fonction avec un argument nommé :

```
1 >>> maximum(500)  
2 Nombre 1: 500  
3 Nombre 2: 100  
4 500  
5 >>> maximum(nbre2 = 500)  
6 Nombre 1: 0  
7 Nombre 2: 500  
8 500  
9 >>>
```

5.3 Résumé

- Une fonction vous permet de regrouper un bloc d'instructions que vous pourrez appeler comme vous le souhaitez et autant de fois que vous le souhaitez.
- Les deux points ":" permettent de délimiter un bloc d'instructions en Python.
- Les espaces sont très très importants en Python.
- Nous avons deux grandes étapes en ce qui concerne les fonctions, la définition de la fonction et l'appel de la fonction.
- Lorsqu'on définit une fonction, les ingrédients ajoutés entre parenthèses sont appelés paramètres. Par contre lors de l'appel de la fonction, on parle plutôt d'arguments.
- Une fonction peut être en effet utilisée pour retourner une valeur et en général c'est ce qui est fait.

- En Python, une fonction peut avoir des paramètres avec des valeurs par défaut.
- Le casting ou transtypage consiste à convertir un objet d'un type donné en un autre.
- La signature d'une fonction en Python est tout simplement son nom.
- En Python, on peut fournir à l'appel d'une fonction ce qu'on appelle des arguments nommés.

6. les conditions et les boucles

6.1 Les conditions

Jusque-là nos programmes étaient tout du moins séquentiels. Aucun suspense ! Juste une série de - Fais ceci, - Ensuite fais cela - Après fais ceci...

Mais dans la vie réelle, les choses ne fonctionnent pas de cette manière. On souhaite faire quelque chose de différent si une situation particulière se présente.

Vous souhaitez par exemple faire une action précise s'il fait jour, une autre action s'il fait nuit....

Comment faire alors pour résoudre ce problème ? Et bien, il nous faudra utiliser les conditions encore appelées structures conditionnelles.

6.1.1 Les opérateurs de comparaison

Je vous propose de découvrir dans un premier temps, les différents opérateurs de comparaison disponibles en Python.

Opérateurs	Signification
==	Égal à
>	Strictement supérieur à
>=	Supérieur ou égal à
<	Strictement inférieur à
<=	Inférieur ou égal à
!=	Différent de



Attention !

Il ne faut surtout pas confondre = et ==. = représente l'opérateur d'affectation. == quant à lui, représente l'opérateur d'égalité. Soyez donc vigilants car cela est parfois une source d'erreur pour les débutants.

Il existe trois (3) formes de structures conditionnelles en Python :

- La forme minimale **if**
- La forme moyenne en **if - else**
- La forme complète en **if - elif - else**

Mais bien avant de se lancer dans la description de chacune d'elles, je vous propose dans un premier temps de découvrir ce que c'est que le type booléen étant donné que les conditions ne sont rien d'autres que des **booléens**.

6.1.2 Le type booléen

Un booléen en programmation est un type de variable à deux états. Les variables de ce type sont ainsi soit à l'état **vrai** ou soit à l'état **faux** (en anglais on dira **True** ou **False**).

Quelques exemples :

- La lampe a deux états. Elle est soit allumée ou soit éteinte.
- La porte a deux états. Elle est soit fermée ou soit ouverte.

En Python, les deux valeurs que peut prendre un booléen sont **True** et **False**.



Attention !

Il faudrait remarquer que le T de **True** et le F de **False** sont en MAJUSCULE. Si vous l'oubliez, attendez vous à avoir droit à une belle erreur.

Voilà c'est tout ce que vous devez savoir sur les booléens. Assez rapide n'est-ce pas ?

6.1.3 La forme minimale if

La forme la plus simple de structures conditionnelles est la forme minimale **if**.

La syntaxe est la suivante :

```
1 if(condition):
2     instruction
3     ...
```

Exemple :

```
1 nombre_de_mains = 2
2
3 if(nombre_de_mains == 2):
4     print("Vous êtes un humain")
```

Les parenthèses entourant la condition sont facultatives mais je vous conseille de toujours les mettre afin d'avoir un code lisible.

L'exemple présenté plus haut est assez simple à comprendre.

- Nous affectons la valeur 2 à la variable `nombre_de_mains`.
- Ensuite, nous faisons un petit test : si la valeur contenue dans la variable `nombre_de_mains` est égale à 2 alors nous affichons **Vous êtes un humain**. Dans le cas contraire nous ne faisons rien.

Etant donné que la variable `nombre_de_mains` est bel et bien égale à 2, alors on verra afficher **Vous êtes un humain**. Ce que vous devez comprendre, c'est que la condition est premièrement évaluée et une valeur booléenne est obtenue. Soit donc `True` ou `False`. Si nous avons `True` avec le bloc d'instructions contenu dans le `if` sera exécuté. Dans le cas contraire le programme continuera comme si de rien n'était.

L'exemple suivant affichera **Coucou les amis**. Remarquez que l'instruction `print("Coucou les amis")` ne fait pas partie du bloc d'instructions (Pas d'indentation).

```
1 nombre_de_mains = 3
2
3 if(nombre_de_mains == 2):
4     print("Vous êtes un humain")
5
6 print("Coucou les amis")
```

6.1.4 La forme moyenne if - else

Avec la forme minimale, comme vous l'avez sûrement remarqué, il n'était pas possible de faire une autre action dans le cas où la condition était évaluée à `False`. C'est tout simplement, ce que vient corriger la forme `if - else`.

La syntaxe est la suivante :

```
1 if(condition):
2     instruction
3     ...
4 else:
5     instruction
6     ...
```

Exemple :


```
1 nombre_de_mains = 3
2
3 if(nombre_de_mains == 2):
4     print("Vous êtes un humain")
5 else:
6     print("Peut être bien un animal :)")
```

Dans le cas où la condition est évaluée à True, le bloc du if sera exécuté. Dans le cas contraire ce sera celui du else qui sera exécuté.

On verra donc afficher **Peut être bien un animal** :).

6.1.5 La forme complète if - elif - else

Il peut arriver que nous ayons plus d'une condition à évaluer. Dans ce cas, nous pouvons utiliser la forme complète if - elif - else.

La syntaxe est la suivante :

```
1 if(condition):
2     instruction
3     ...
4 elif(condition):
5     instruction
6     ...
7 elif(condition):
8     instruction
9     ...
10 else:
11     instruction
12     ...
```

Exemple :

```
1 note = 12
2
3 if(note > 18):
4     print("Vous êtes excellent.")
5 elif(note > 15):
6     print("Très bien.")
7 elif(note > 10):
8     print("Pas mal.")
9 else:
10    print("Quel nullard :).")
```

elif est la contraction de **else if** qui signifie *sinon si*.

Ainsi si l'on devait traduire notre programme en français on aurait eu ceci :

```
1 note = 12
2
3 si(note > 18):
4     print("Vous êtes excellent.")
5 sinon si(note > 15):
6     print("Très bien.")
7 sinon si(note > 10):
8     print("Pas mal.")
9 sinon:
10    print("Quel nullard :).")
```

Vu que la note est égale à 12, alors on verra afficher **Pas mal**.

6.1.6 L'indentation

Je sais qu'on en a déjà parlé, mais je préfère en reparler pour être sûr que vous l'avez bien compris.

- L'indentation est indispensable en Python, sinon vous aurez droit à une erreur.
- Cela n'est pas le cas par exemple pour d'autres langages de programmation comme le C ou le JAVA, qui utilisent un marqueur de fin d'instruction ";" et des délimiteurs de bloc d'instructions {}.

Tout compte fait, indenter son code reste une bonne habitude car cela permet de rendre son code lisible et compréhensible sans trop de difficultés.

6.1.7 Parité d'un nombre

Vous êtes maintenant en mesure de comprendre le programme qu'on avait utilisé dans le chapitre précédent portant sur les fonctions.

```
1 def est_pair(nombre):
2     """Renvoie True si nombre est pair
3     et False dans le cas contraire"""
4
5     if(nombre % 2 == 0):
6         return True
7     else:
8         return False
```

Ce programme utilise donc la forme moyenne if-else. En grosso modo, on demande à retourner le booléen **True** si le nombre passé en argument est divisible par 2 (ce qui signifiera qu'il est pair) et dans le cas contraire **False** pour signifier qu'il est impair.

Une chose que vous devez également savoir c'est que dès que l'interpréteur exécute l'instruction **return**, tout ce qui vient après ne sera pas exécuté.

Ainsi dans l'exemple qui suit, **Je suis cool** ne sera donc pas affiché, seule la chaîne **Je serai affichée** sera affichée. Tachez de vous en souvenir.

```
1 def petit_test():
2     print("Je serai affichée")
3     return True
4     print("Je suis cool")
```

6.1.8 Les opérateurs logiques

Très souvent l'on ait amené à tester plus d'une condition. Dans ce cas, on se sert des opérateurs logiques. Ces opérateurs logiques sont les suivants :

- Le OU logique ou **or** en anglais
- Le ET logique ou **and** en anglais
- Le NON logique ou **not** en anglais

Le OU logique

L'opérateur *or* ou *OU logique* vous renvoie True si l'une des conditions est vraie. Prenons un exemple simple. Pour être le major de sa promotion, il faut soit faire assez d'exercices **ou** bien suivre les explications du professeur en classe.

A cause de l'utilisation du OU logique, notre étudiant sera major de sa promotion si l'une des conditions est vraie. Soit il fait donc beaucoup d'exercices, soit il suit bien en classe.

```
1 >>> a, b = 4, 5 #a recoit 4 et b recoit 5. Assez cool non :)?
2 >>> a == 4 or b == 9 #Vu que l'une des conditions est vraie on a donc True
3 True
4 >>>
```

Le ET logique

L'opérateur *and* ou *ET logique* vous renvoie True si toutes les conditions sont vraies. Prenons encore le même exemple que précédemment. Pour être le major de sa promotion, il faut faire assez d'exercices **et** bien suivre les explications du professeur en classe.

A cause de l'utilisation du ET logique, cette fois-ci notre étudiant sera major de sa promotion s'il fait donc non seulement beaucoup d'exercices mais également s'il suit bien en classe.

Il faut donc que nos deux conditions soient vérifiées pour qu'on ait True.

```
1 >>> a, b, c = 4, 5, 9 #a recoit 4 et b recoit 5, c recoit 9.
2 >>> a == 4 and b == 5 and c == 6 #Vu que la dernière condition est fausse on a d\
3 onc False
4 False
5 >>>
```

Le NON logique

C'est l'opérateur logique le plus simple à mon avis. En effet, il vous retourne True si vous lui donnez False et False si vous lui donnez True.

Voyons cela en pratique :

```
1 est_malade = True
2
3 if(not est_malade):
4     print("Vous n'êtes pas malade.")
5 else:
6     print("Vous êtes malade!")
```

La variable `est_malade` vaut au départ True. Le fameux `not est_malade` retournera le contraire de True, ce qui revient à dire False. La condition sera donc évaluée à False. Le bloc `else` sera ainsi exécuté et **Vous êtes malade** sera affiché.

Cela peut sembler ambigu j'avoue :).

6.2 Les itérations ou boucles

Les boucles permettent de répéter un certain nombre de fois des instructions de votre programme. Pour être sincère avec vous, on ne peut se passer des boucles lors de l'écriture de vrais programmes. Prenons un exemple tout simple. Supposons que vous souhaitez afficher tous les nombres de 0 à 100. Une solution possible serait la suivante :

```
1 print(0)
2 print(1)
3 print(2)
4 print(3)
5 ...
6 #Bonne chance
```

Comme vous pouvez le voir, il serait vraiment fastidieux de procéder de la sorte. Heureusement les boucles sont là pour nous sauver la vie ! En Python il existe uniquement 2 structures itératives ou boucles :

- la boucle **while**
- la boucle **for**

6.2.1 La boucle while

La boucle **while** permet de répéter un bloc d'instructions tant qu'une condition est vraie (**while** signifie **tant que** en anglais).

Sa syntaxe est la suivante :

```
1 while(condition):
2     instruction
3     ...
```

Ainsi pour afficher tous les nombres de 0 à 100 on écrira :

```
1 i = 0 #On commence a zéro
2
3 #Tant qu'on a pas encore atteint 100, on continue
4 while(i <= 100):
5     print(i)
6     i = i + 1 #On ajoute à chaque fois 1 à la valeur de i
```

La dernière instruction de notre programme est ce qu'on appelle **l'incrémentation**. En d'autres termes incrémenter une valeur, c'est tout simplement ajouter 1 à cette valeur.

```
1 i = i + 1
```

Etant donné que cette opération est très courante, il existe un petit raccourci en python.

```
1 i += 1
```

De même, nous avons la décrémentation qui elle consiste à retirer 1 à la valeur concernée.

```
1 i = 10
2 i = i - 1 # i vaut 10 - 1 = 9
3 i -= 1 #i vaut 9 - 1 = 8
4 print(i) #Affiche 8
```

Le principe reste le même pour la multiplication, la division et le modulo.

```
1 i = 10
2 i *= 3 #i vaut 10 * 3 = 30
3 i /= 15 #i vaut 30 / 15 = 2
4 i %= 2 #i vaut 2 % 2 = 0
5 print(i) #Affiche 0
```

Ainsi notre programme précédent pouvait également être écrit comme suit :

```
1 i = 0
2
3 while(i <= 100):
4     print(i)
5     i += 1
```

Notons que sans l'incrémentation de la valeur de la variable *i* à chaque tour de boucle, nous aurons eu droit tout simplement à une boucle infinie. Une boucle infinie est une boucle qui ne se termine jamais. On a rarement besoin d'une boucle qui ne s'arrête jamais, à moins que vous souhaitez faire planter votre ordinateur. Ainsi lorsque vous écrivez vos boucles, assurez vous que ces dernières prendront fin à un moment donné. Be careful !!

6.2.2 La boucle for

La boucle **for** travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données (**for** signifie **pour** en anglais).

Sa syntaxe est la suivante :

```
1  for element in sequence:
2      instruction
3      ...
```

Vu qu'une image vaut mieux que mille mots, voyons quelques exemples :

```
1  nom = "Honore"
2  for lettre in nom:
3      print(lettre)
```

Affichera :

```
1  >>>
2  H
3  o
4  n
5  o
6  r
7  e
8  >>>
```

Une chaîne de caractères est composée d'une séquence de caractères. Ainsi, dans notre cas on demande à afficher chaque élément de cette séquence, ce qui revient à afficher l'ensemble des caractères les uns après les autres.

Par défaut, la fonction **print** après avoir affiché votre message met le curseur automatiquement à la ligne suivante. Si vous ne souhaitez pas avoir cet effet, vous pouvez tout simplement écrire :

```
1  print('Toto', end="")
```

Ainsi après avoir affiché **Toto**, le curseur restera sur la même ligne.

```
1 print("Mes langages préférés sont ", end="")
2 print("le langage C", end=", ")
3 print("le langage Java", end=" ")
4 print("et le sublime Python :)")
```

On aura comme résultat :

```
1 Mes langages préférés sont le langage C, le langage Java et le sublime Python :)
```

Comme vous pouvez le voir avec le paramètre **end** on peut préciser n'importe lequel des séparateurs que nous souhaitons. En gros le paramètre **end** permet de dire après avoir affiché ma ligne mets ceci (le séparateur que vous aurez choisi) à la suite.

Voyons un deuxième exemple avant de clôturer ce chapitre.

```
1 #Affiche tous les nombres de 0 a 100
2 for i in range(0, 101):
3     print(i)
```

La fonction range

Synopsis : `range([début,] fin[, pas])`

- La fonction `range()` permet de retourner une séquence virtuelle de nombres de **début** à **fin** par pas de **pas**.
- Le début et le pas par défaut sont respectivement 0 et 1. Les éléments entre crochets sont dits optionnels.
- Le premier paramètre est inclus, le second quant à lui est exclus.

Exemple :

- `range(15)` retourne une séquence virtuelle de nombres de 0 à 15 (15 étant exclus) : 0,1,2,3,...,14
- `range(1,15)` retourne une séquence virtuelle de nombres de 1 à 15 (15 étant exclus) : 1,2,3,4,...,14.
- `range(0,15,2)` retourne une séquence virtuelle de nombres de 0 à 15 (15 étant exclus) par pas de 2 \Rightarrow c'est-à-dire les nombres pairs compris entre 0 et 15 : 0,2,4,6,8,10,12,14.

Amusez-vous à faire quelques tests afin de vérifier que vous avez bien compris le principe de fonctionnement de cette fonction **range**. Croyez-moi, vous allez l'utiliser un bon nombre de fois.


```

1 >>> list(range(15))
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
3 >>> list(range(1,15))
4 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
5 >>> list(range(0,15,2))
6 [0, 2, 4, 6, 8, 10, 12, 14]
7 >>>

```

Ne vous intéressez pas à l'utilité de la fonction `list` pour le moment, mais ici vous avez besoin de la mettre sinon vous ne verrez rien de concret. C'est pour cette raison qu'on parle de séquence virtuelle.

```

1 >>> range(15)
2 range(0, 15)
3 >>>

```

Le mot-clé `in`

Le mot-clé `in` peut être utilisé ailleurs que dans une boucle `for`.

En voici une illustration :

```

1 chaine= "Hello World"
2 for letter in chaine:
3     if letter in "aeiouyAEIOUY":
4         print(letter)

```

Ce programme n'affichera que les voyelles contenues dans la variable `chaine`.

6.2.3 Les mots clés `break` et `continue`

Le mot-clé `break` permet d'arrêter la boucle appelante.

```

1 for i in range(0, 15):
2     if(i == 2):
3         break
4     print(i)

```

Ce programme affichera seulement 0, 1 étant donné que lorsque `i` vaudra 2 la boucle s'arrêtera automatiquement à cause du fameux `break`.

```
1  for i in range(0, 15):
2      for j in range(10, 20):
3          print(i, j)
4          break
```

Ce programme affichera :

```
1  >>>
2  0 10
3  1 10
4  2 10
5  3 10
6  4 10
7  5 10
8  6 10
9  7 10
10 8 10
11 9 10
12 10 10
13 11 10
14 12 10
15 13 10
16 14 10
17 >>>
```

Rappelez-vous, j'ai dit que le **break** arrêterait la boucle appelante et non toutes les boucles. Ainsi notre fameux **break** arrêtera simplement la seconde boucle.

Voyons en détails ce qui passe pour que tout soit clair :

- Lors du premier tour de boucle **i** vaudra 0. On rentre ensuite dans la seconde boucle **j** vaut 10. Ainsi on affiche **0 10**. Ensuite on rencontre le **break**, ce qui fait que le **j** ne passera pas à 11. On sort donc de la seconde boucle et on retourne à la première boucle.
- A présent **i** vaut 1, on rentre dans la seconde boucle **j** vaut 10. Ainsi on affiche **1 10**. Ensuite on rencontre le **break**, ce qui fait que le **j** ne passera pas également à 11. On sort donc de la seconde boucle et retourne à la première boucle.
- Le **i** vaut maintenant 2 et ainsi de suite...

Le mot-clé **continue** permet de passer simplement au prochain tour de boucle.

```
1  #Affiche les nombres pairs de 0 à 15
2  for i in range(0, 15):
3      if(i % 2 != 0):
4          continue
5      print(i)
```

Le mot-clé continue comme on vient de le voir n'arrête pas la boucle mais permet de passer simplement au prochain tour de boucle.

6.3 Résumé

Dans ce chapitre, nous avons appris que :

- Les structures conditionnelles nous permettent d'exécuter un bloc d'instructions en fonction de certaines conditions.
- Il ne faut surtout pas confondre = et ==. = représente l'opérateur d'affectation, == quant à lui, représente l'opérateur d'égalité.
- Il existe trois (3) formes de structures conditionnelles en Python : la forme minimale if, la forme moyenne if-else et la forme complète if-elif-else.
- Un booléen a comme valeur True ou False.
- Indenter son code reste une bonne habitude car cela permet de rendre son code lisible et compréhensible sans trop de difficultés.
- Comme opérateurs logiques nous avons le OU logique, le ET logique et le NON logique.
- Les boucles permettent de répéter un certain nombre de fois un bloc d'instructions.
- En Python il existe uniquement deux structures itératives à savoir les boucles while et for.
- Le mot-clé in peut être utilisé autre part que dans une boucle for.
- Le mot-clé break permet d'arrêter la boucle appelante.
- Le mot-clé continue permet de passer au prochain tour de boucle.

7. Les structures de données (Partie 1/2)

Nous avons vu que les variables représentaient un excellent moyen de stocker une valeur en mémoire. De plus, si on le souhaite, cette valeur peut être amenée à changer au fur et à mesure de l'exécution de notre programme.

Mais que faire si nous avons besoin de stocker une longue liste d'informations et de plus qui ne change pas avec le temps ? Disons par exemple, les noms des mois de l'année. Ou peut-être une longue liste d'informations qui, elle, change au fil du temps ? Par exemple, les noms de tous vos chats. Vous pouvez obtenir de nouveaux chats, certains peuvent mourir, d'autres peuvent par contre se transformer en diner (nous devrions alors négocier recettes :)). Qu'en est-il d'un annuaire téléphonique ? Vous aurez une liste de noms, et vous devrez attacher à chacun de ces noms, un numéro de téléphone. Comment feriez-vous cela ?

La réponse à toutes ces interrogations : les tuples, les listes, les dictionnaires.

7.1 Les tuples

Les tuples vous permettent de stocker une liste de valeurs. Le seul bémol c'est que vous ne pouvez pas changer les valeurs une fois votre tuple initialisé. En d'autres termes, les valeurs que vous lui donnez lors de son initialisation, seront les valeurs que vous serez coincés d'utiliser dans tout le reste du programme.

Afin de pouvoir récupérer plus tard chacune de ces valeurs de manière aisée, nous allons associer à chacune d'elles un indice ou index. La première valeur aura pour indice 0, la seconde pour indice 1 et ainsi de suite. Remarquez bien que le premier indice est 0 et non 1.

Par exemple les noms des mois de l'année pouvaient être stockés à l'aide un tuple à moins que dans deux jours Février deviennent MamaMiya :) ?

Les tuples sont assez faciles à créer. Vous donnez un nom à votre tuple, puis après, la liste des valeurs qu'il comportera. Par exemple, les mois de l'année :

```
1 mois = ("Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août", \
2 "Septembre", "Octobre", "Novembre", "Décembre")
```

- "Janvier" est l'élément à l'indice 0 (On commence toujours à zéro),
- "Février" est l'élément à l'indice 1,

- “Mars” est l’élément à l’indice 2,
- ... Je vous laisse terminer. Ce sera un bon exercice.
- Notez que j’ai eu à mettre un antislash tout juste après “Août”. C’est en effet un moyen de pouvoir décomposer notre code sur plusieurs lignes afin de le rendre beaucoup plus lisible.
- Les parenthèses utilisées pour délimiter le début et la fin de notre tuple sont optionnelles. Vous n’êtes donc pas obligé de les utiliser. Mais je vous conseille de toujours les mettre afin de ne pas vous compliquer la vie mais celle également de notre interpréteur. Il a déjà assez souffert dans son enfance :).

Voyons à présent comment afficher un élément de notre tuple. Pourquoi pas **Janvier**, **Mars** et **Juillet**. Rappelons que Mars est le 3ème élément de notre tuple mais il a pour indice 2.

```

1  # Je suis également un tuple croyez-moi :(
2  mois = "Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août", \
3  "Septembre", "Octobre", "Novembre", "Décembre"

1  #Declaration de notre tuple
2  mois = ("Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août", \
3  "Septembre", "Octobre", "Novembre", "Décembre")
4
5  print(mois[0]) #Affiche Janvier
6  print(mois[2]) #Affiche Mars
7  print(mois[6]) #Affiche Juillet

```

Comme vous pouvez le voir, c’est vraiment très simple ! Il suffit de mettre le nom de notre tuple et ensuite préciser entre crochets l’indice de l’élément auquel on souhaite accéder.

7.2 Les listes

Les listes sont ce qu’elles semblent être - une liste de valeurs. A la différence des tuples, vous pouvez ajouter, modifier et supprimer des valeurs de la liste comme bon vous semble. Encore une fois, chaque valeur aura un indice et on commencera à compter à partir de zéro et non 1. Un exemple de liste sera les noms de vos nombreux chats.

```

1  nom_de_mes_chats = ["Panpidou", "Milou", "Mamole"]

```

Comme vous le voyez, le code est exactement le même que celui de la déclaration d’un tuple, SAUF que toutes les valeurs sont mises entre crochets, pas de parenthèses. Les crochets ici ne sont pas du tout optionnels.

Quel est l’élément à l’indice 2 ? Hmm je crois bien que c’est “Mamole”. Est-ce exact ? Quel est l’élément à l’indice 5 ? Hmm je crois bien que c’est “...” ?

La plupart du temps, nous utilisons des listes en lieu et place des tuples parce que généralement ce que nous voulons c’est changer facilement les valeurs de nos éléments si le besoin se fait ressentir.

```

1  #Déclaration de notre liste
2  nom_de_mes_chats = ["Panpidou", "Milou", "Mamole"]
3
4  print(nom_de_mes_chats[0]) #Affiche Panpidou
5  print(nom_de_mes_chats[2]) #Affiche Mamole
6  print(nom_de_mes_chats[5]) #Affiche une belle erreur

```

Comme vous pouvez le voir, notre liste comporte 3 éléments. Les indices varient de 0 à 2. Ainsi lorsqu'on tente d'accéder à l'élément à l'indice 5, notre interpréteur est confus et il nous le fait savoir en affichant une erreur. En réalité on dira qu'une exception de type `IndexError` est levée.

```

1  Traceback (most recent call last):
2    File "<pyshell#71>", line 1, in <module>
3      print(nom_de_mes_chats[5])
4  IndexError: list index out of range

```

Je vous avais dit qu'à la différence des tuples, on pouvait ajouter, modifier ou supprimer une valeur au niveau d'une liste. Voyons donc comment le faire.

```

1  >>> nom_de_mes_chats = ["Panpidou", "Milou", "Mamole"] #Déclaration de notre liste
2  te
3  >>> nom_de_mes_chats.append("Bobby") #Ajout d'un élément à la liste
4  >>> nom_de_mes_chats
5  ['Panpidou', 'Milou', 'Mamole', 'Bobby']
6  >>> nom_de_mes_chats.remove("Milou") #Suppression d'un élément grâce à sa valeur
7  >>> nom_de_mes_chats
8  ['Panpidou', 'Mamole', 'Bobby']
9  >>> nom_de_mes_chats[1] = "Toto" #Modification d'une valeur
10 >>> nom_de_mes_chats
11 ['Panpidou', 'Toto', 'Bobby']
12 >>> del nom_de_mes_chats[1] #Suppression d'un élément via son indice
13 >>> nom_de_mes_chats
14 ['Panpidou', 'Bobby']
15 >>>

```

Je crois que les commentaires parlent d'eux-mêmes. Mais pour résumer on peut dire que :

- Pour ajouter un élément à une liste, on met le nom de la liste, suivi d'un joli petit point, suivi de la méthode `append`, et pour terminer entre parenthèses nous précisons la valeur à rajouter. Il faut préciser que les éléments n'ont pas forcément besoin d'être du même type. J'aurai pu rajouter l'entier 5 à ma liste de chats et personne ne m'aurait blâmer, pas même l'interpréteur.

- Pour modifier un élément d'une liste, on utilise l'affectation classique. On accède premièrement à l'élément qu'on souhaite modifier via son indice et on lui affecte une nouvelle valeur.
- Pour supprimer un élément d'une liste, on peut le faire de plusieurs façons : via son indice ou via sa valeur. J'ai tendance à utiliser la commande `del` pour la suppression. J'utilise donc la méthode de suppression via un indice. Mais libre à vous de vous faire votre propre opinion de la question.

7.3 Les dictionnaires

Les dictionnaires sont similaires à ce que leur nom indique : un dictionnaire. Dans un dictionnaire, vous avez un « index » de mots, et pour chacun d'eux une définition. En Python, le mot est appelé une « clé », et la définition une « valeur ». Les valeurs d'un dictionnaire ne sont pas numérotées comme c'est le cas pour les listes et les tuples. Notez également que les éléments ne sont pas dans un ordre particulier. Nous y reviendrons.

L'annuaire téléphonique sera un exemple parfait de dictionnaire.

```
1 contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000", "Dad": "32283\
2 904484"}
```

On aurait pu utiliser un entier pour les numéros de téléphone mais j'ai préféré utiliser ici une chaîne de caractères parce que c'est ce qu'il y a de plus adapté à mon avis. De plus, si le numéro est un peu long, un entier ce n'est vraiment pas ce qu'il y a de conventionnel :).

Alors un peu de pratique sur les dictionnaires, ça vous dit ?

```
1 >>> contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000", "Dad": "3\
2 2283904484"}
3 >>> contacts["Marie"] = "4579303332" #Ajout d'un nouveau contact
4 >>> contacts
5 {'Ma princesse': '000000000', 'Dad': '32283904484', 'Marie': '4579303332', 'Mama\
6 n': '77562024848'}
7 >>> contacts["Maman"] = 30393974 #Modifier un contact
8 >>> contacts
9 {'Ma princesse': '000000000', 'Dad': '32283904484', 'Marie': '4579303332', 'Mama\
10 n': 30393974}
11 >>> contacts.keys() #Affiche toutes les clés
12 ['Ma princesse', 'Dad', 'Marie', 'Maman']
13 >>> contacts.values() #Affiche toutes les valeurs
14 ['000000000', '32283904484', '4579303332', '77562024848']
15 >>> del contacts["Ma princesse"] #Suppression d'un contact
16 >>> contacts
```

```

17 {'Dad': '32283904484', 'Marie': '4579303332', 'Maman': 30393974}
18 >>> mes_cles = contacts.keys() #On stocke les clés dans la variable mes_cles
19 >>> mes_cles.sort() #On les met en ordre
20 >>> mes_cles
21 ['Dad', 'Maman', 'Marie']
22 >>>

```

Comme je vous l'avais dit, les dictionnaires ne sont rien d'autres qu'un ensemble de clés et de valeurs. La syntaxe est la suivante :

```
1 dictionnaire = { cle1 : valeur1, cle2 : valeur2, cle3 : valeur3 }
```

Ainsi très facilement nous déclarons notre dictionnaire contacts.

```

1 contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000", "Dad": "32283\
2 904484"}

```

Pour ajouter un nouvel élément à notre dictionnaire, pas de méthode append cette fois-ci comme c'était le cas pour les listes. Il vous suffit d'ajouter une nouvelle clé et une nouvelle valeur à votre dictionnaire. Si cette clé existe déjà, alors il procédera à une modification, dans le cas contraire, ce sera un ajout.

- Vous avez pu remarqué, comme je vous l'avais dit, qu'avec les dictionnaires les éléments ne sont pas stockés dans un ordre particulier. Ce sera donc à vous de le faire si vous souhaitez par exemple stocker vos contacts par ordre alphabétique par exemple. Pour l'instant j'ai eu à vous montrer comment ordonner séparément les clés. Mais nous y reviendrons.
- Vous avez la possibilité de récupérer les différentes clés et valeurs d'un dictionnaire en utilisant respectivement les méthodes **keys()** et **values()**.

Il existe un grand nombre de fonctions que nous pouvons appliquer aux listes et aux dictionnaires. Mais on ne pourra pas toutes les découvrir. Cependant, je m'efforcerai au fur et à mesure que nous avançons d'en présenter des nouvelles. Vous devez sans aucun doute utiliser [la documentation officielle](https://www.python.org/doc/)¹⁴ si vous voulez avoir une liste exhaustive des ces fonctions. Je crois bien que c'est le moment de prendre une petite pause café afin de digérer tout ce qu'on a vu jusque là parce que dans la section suivante on aborde un autre gros morceau.

14. <https://www.python.org/doc/>

7.4 Résumé

Dans ce chapitre, nous avons appris que :

- Python nous donne également la possibilité de pouvoir stocker un ensemble de valeurs au travers des tuples, des listes et des dictionnaires.
- Une fois un tuple initialisé, ses valeurs ne peuvent être modifiées.
- A la différence des tuples, nous pouvons ajouter, modifier et supprimer des valeurs de la liste comme bon nous semble.
- Les dictionnaires nous permettent de stocker des éléments de type clé - valeur.
- Les valeurs d'un dictionnaire ne sont pas numérotées comme c'est le cas pour les listes et les tuples.
- Dans un dictionnaire, les éléments ne sont pas ordonnés.

8. Les structures de données (Partie 2/2)

Dans le chapitre précédent, nous avons introduit ce qu'étaient les structures de données. Dans cette seconde partie, je vous invite à entrer un peu plus en profondeur.



Je répéterai volontairement certaines notions que nous avons vu dans le chapitre précédent pour question de pédagogie. Croyez-moi plus on reprend quelque chose, mieux on la comprend.

8.1 Les types non scalaires

Comme nous l'avons déjà vu, Il existe trois types de données en Python nous permettant de rassembler des données. Il s'agit des tuples, des listes et des dictionnaires.

Ce sont des types **non scalaires** car ils peuvent contenir plusieurs éléments à la fois, contrairement aux types scalaires (**int**, **float**...) qui eux ne peuvent contenir qu'un seul élément à la fois.

Ce que vous devez savoir c'est que les chaînes de caractères font partie des types **non scalaires** puisqu'elles peuvent contenir plusieurs éléments à la fois. En effet, une chaîne de caractères n'est rien d'autre qu'une séquence de caractères.

8.2 Les chaines de caractères

Vu qu'une chaîne de caractères n'est rien d'autre qu'une séquence de caractères, je peux demander à récupérer le premier caractère, le second, le dernier...

```
1 >>> chaine = "Python is amazing"
2 >>> print(chaine[0])
3 P
4 >>> print(chaine[-1])
5 g
6 >>>
```

Comme vous pouvez le voir, on peut très facilement récupérer le dernier élément d'une séquence de données en utilisant un indice négatif. Dans notre cas **-1**. Si je voulais avoir l'avant dernier caractère, il m'aurait fallu écrire **chaine[-2]** et ainsi de suite. Cela marche également sur les tuples et les listes.

```

1 >>> ma_liste = [1,2,3,4]
2 >>> ma_liste[-1]
3 4
4 >>> mon_tuple = (1,2,3,4)
5 >>> mon_tuple[-2]
6 3
7 >>>

```

Les chaînes de caractères sont dites **immuables** étant donné qu'une fois définies, on ne peut pas changer la valeur d'un élément spécifique.

```

1 >>> chaine = "Python is amazing"
2 >>> chaine[0] = "M"
3 Traceback (most recent call last):
4   File "<pyshell#6>", line 1, in <module>
5     chaine[0] = "M"
6   TypeError: 'str' object does not support item assignment

```

Au passage, rappelez-vous qu'on avait dit également que les tuples étaient immuables. Pour preuve :

```

1 >>> mon_tuple = (1,2,3,4,5)
2 >>> mon_tuple[1] = 3
3 Traceback (most recent call last):
4   File "<pyshell#8>", line 1, in <module>
5     mon_tuple[1] = 3
6   TypeError: 'tuple' object does not support item assignment
7 >>>

```

8.2.1 Les slices

Les slices nous permettent de sélectionner une partie ou toute une séquence de données.

La syntaxe est la suivante : **sequence[indice_debut : indice_fin]**. Et on dira qu'on souhaite récupérer tous les éléments de notre séquence ayant leurs indices variant de **indice_debut** à **indice_fin**, (indice_fin étant exclu)

Ainsi l'intervalle des indices représenté mathématiquement sera : **[indice_debut, indice_fin[**, ce qui signifie que l'indice de fin ne fait pas partie des indices à inclure.

```
1 >>> chaine = "Python is amazing"
2 >>> chaine[1:2]
3 'y'
4 >>> chaine[3:4]
5 'h'
6 >>>
```

Autre chose que vous devez savoir, c'est ce que `indice_debut` et `indice_fin` ont comme valeur par défaut.

- Si vous ne précisez donc pas la borne inférieure du slice, `indice_debut` vaudra par défaut 0.
- Si vous ne précisez donc pas la borne supérieure du slice, `indice_fin` sera égal par défaut à la taille de votre séquence.

Ainsi :

```
1 >>> chaine = "Python is amazing"
2 >>> chaine[:]
3 'Python is amazing'
4 >>>
```

comme vous pouvez le voir récupèrera la séquence entière. On utilise très souvent cette technique pour créer une copie d'une liste. On y reviendra dans les détails.

Je vous propose pour clore cette section, deux autres exemples afin de vous permettre de vous familiariser avec cette notion de slicing.

```
1 >>> chaine = "Python is amazing"
2 >>> chaine[4:]
3 'on is amazing'
4 >>> chaine[: -1]
5 'Python is amazin'
```

8.2.2 Modification d'une chaîne de caractères

A voir le titre de la section, vous vous êtes sûrement dit : Mais de quoi parle ce mec ? Il nous a dit il y a quelques minutes de cela qu'on ne pouvait pas modifier les éléments d'une chaîne de caractères une fois définie !

Vous avez tout à fait raison. Et je maintiens ma position ! Le fait est qu'avec l'utilisation du slicing on pourra faire semblant de résoudre ce petit problème en trichant un tout petit peu.

```

1 >>> chaine = 'baba'
2 >>> chaine = 'd' + chaine[1:]
3 >>> chaine
4 'daba'
5 >>> chaine = 'P' + chaine[1] + 'p' + chaine[3:]
6 >>> chaine
7 'Papa'
8 >>>

```



Les chaines de caractères sont immuables !

Attention, cela ne veut pas dire que les chaines de caractères sont muables, ce serait alors se contredire, mais c'est juste une combinaison de concaténation de chaîne de caractères et d'affection.

8.2.3 Parcourir une chaine de caractères

On peut parcourir tous les éléments d'une chaîne de caractères avec les deux types de boucles que nous avons vus à savoir la boucle `for` et la boucle `while`. Notons néanmoins pour rappel, que la boucle `for` est la mieux adaptée pour parcourir une séquence de données.

Avec la boucle `for`

```

1 chaine = "coucou"
2
3 for caractere in chaine:
4     print(caractere)

```

Avec la boucle `while`

```

1 chaine = "coucou"
2 i = 0
3
4 while( i < len(chaine) ):
5     print(chaine[i])
6     i += 1

```



La fonction `len`

La fonction `len` vous renvoie la taille d'une séquence de données. En d'autres termes, le nombre d'éléments qui la compose.

8.2.4 Méthodes courantes applicables aux chaînes de caractères

Il existe plusieurs méthodes (fonctions) s'appliquant aux chaînes de caractères. Un simple `help(str)` dans l'interpréteur, vous donnera une liste exhaustive de ces méthodes.

Découvrons en quelques unes si vous le permettez :

```

1  >>> chaine = "Qui est-ce qui n'aime pas PYTHON?"
2  >>> chaine.lower() #Conversion des caractères en minuscules
3  "qui est-ce qui n'aime pas python?"
4  >>> chaine.upper() #Conversion des caractères en majuscules
5  "QUI EST-CE QUI N'AIME PAS PYTHON?"
6  >>> chaine.swapcase()
7  "qUI EST-CE QUI N'AIME PAS python?" # Devinez :)
8  >>> chaine.capitalize() #Conversion du premier caractère en majuscule
9  "Qui est-ce qui n'aime pas python?"
10 >>> chaine.title() #Conversion du premier caractère de chaque mot en majuscule.
11 "Qui Est-Ce Qui N'Aime Pas Python?"
12 >>> chaine = " j'ai des espaces "
13 >>> chaine.strip() #Suppression des espaces à gauche et à droite
14 "j'ai des espaces"
15 >>> chaine.find('e') #Recherche un sous-chaîne dans une chaîne de caractères. Si \
16 la sous-chaîne a été trouvée, l'indice de sa première occurrence est retourné. D\
17 ans le cas contraire la valeur -1 est retournée.
18 7
19 >>> chaine.find("z")
20 -1
21 >>> chaine.strip().capitalize().replace('espaces', 'femmes') #Ce exemple parle d\
22 e lui-même :)
23 "J'ai des femmes "
24 >>> chaine = "BARACK OBAMA"
25 >>> chaine.replace("A", "*")
26 'B*R*CK OB*M*'
27 >>>

```



Comme vous l'avez remarqué avec `chaine.strip().capitalize().replace('espaces', 'femmes')`, nous avons la possibilité de chaîner les différentes méthodes. Ici on supprime premièrement les espaces à gauche et à droite grâce à la méthode `strip()`. Ensuite on met le premier caractère en majuscule via la méthode `capitalize()` et pour terminer pour le bon plaisir on remplace `espaces` par `femmes` en utilisant la méthode `replace()`. Une petite remarque toutefois, ne tombez surtout pas dans l'abus. Privilégiez toujours la lisibilité à la paresse :)

Pour plus de méthodes, je vous invite à consulter la documentation.

Attention ! les méthodes appliquées à la chaîne de caractères ne la modifie en aucun cas. Elles renvoient tout simplement un nouvel objet (une nouvelle chaîne de caractères) modifié.

```
1 >>> chaine = "TEST"
2 >>> chaine.lower()
3 'test'
4 >>> chaine
5 'TEST'
6 >>> chaine = chaine.lower()
7 >>> chaine
8 'test'
9 >>>
```

8.3 Les tuples

Au risque de me répéter, les tuples tout comme les chaînes de caractères sont immuables, ce qui veut dire qu'une fois créés on ne peut pas les modifier.

Un tuple peut contenir d'autres tuples et n'importe quel autre type de données.

```
1 >>> mon_tuple = ([1,2,3], True, 3.4)
2 >>> mon_tuple[0]
3 [1, 2, 3]
4 >>> mon_tuple[0][0]
5 1
6 >>>
```

Nous avons toutefois la possibilité de concaténer plusieurs tuples.

```
1 >>> tuple1 = (1,2,3)
2 >>> tuple2 = (3,4,5)
3 >>> tuple1 + tuple2
4 (1, 2, 3, 3, 4, 5)
5 >>>
```

Pour ajouter ainsi un seul élément à un tuple déjà créé, nous pouvons créer un nouveau tuple composé de ce seul élément, le concaténer avec le premier tuple et ensuite l'affecter à ce dernier.

```

1  >>> tuple_test = (1,2,3,4)
2  >>> tuple_new = (5,) #Sans la virgule, ce ne serait plus un tuple mais l'entier \
3  5.
4  >>> tuple_test = tuple_test + tuple_new
5  >>> tuple_test
6  (1, 2, 3, 4, 5)
7  >>> tuple_test = tuple_test[0:2] + (2.5,) + tuple_test[3:]
8  >>> tuple_test
9  (1, 2, 2.5, 4, 5)
10 >>> len(tuple_test)
11 5
12 >>>

```

8.3.1 Parcourir un tuple

Avec la boucle while

```

1 tuple_test = (1, 'abc', 3.45)
2 i = 0
3
4 while( i < len(tuple_test) ):
5     print(tuple_test [i])
6     i += 1

```

Avec la boucle for

```

1 tuple_test = (1, 'abc', 3.45)
2
3 for element in tuple_test:
4     print(element)

```



```

1 tuple_test = (1, 'abc', 3.45)
2
3 for i in range( len(tuple_test) ):
4     print(tuple_test[i])

```

8.4 Les listes

Les listes sont muables, ce qui veut dire qu'on peut les modifier une fois créées. Cela représente un avantage considérable par rapport aux tuples.

Une liste composée de 10 fois le chiffre 3, ca vous tente ?


```

1 >>> liste = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
2 >>> len(liste)
3 10
4 >>>

```

Avouez que ça craint ! Je vous propose une meilleure solution :

```

1 >>> liste = [3]*10
2 >>> liste
3 [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
4 >>> len(liste)
5 10
6 >>>

```

On peut utiliser à peu près le même principe avec les chaînes de caractères. Admirez :

```

1 >>> "cou" * 2
2 'coucou'
3 >>> "cou" * 4
4 'coucoucoucou'
5 >>>

```

Le principe du slicing reste également valide :

```

1 >>> liste = list( range(10) )#Nous sommes paresseux :)
2 >>> liste
3 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4 >>> liste[2:]
5 [2, 3, 4, 5, 6, 7, 8, 9]
6 >>> liste[3:6]
7 [3, 4, 5]
8 >>>

```

8.4.1 Parcourir une liste

Avec la boucle for

Je vous laisse ceci comme exercice.

Avec la boucle while

Je vous laisse cela également comme exercice :).

8.4.2 Le full slicing

Je vous avais dit qu'on y reviendra sur cette écriture `liste[:]` encore appelée *full slicing*. Le moment est enfin arrivé! Le **full slicing** nous permet de copier une liste dans une autre sans pour autant créer un alias. Ce qui permettra ainsi de modifier l'une des deux listes indépendamment de l'autre puisqu'elles ne pointeront pas sur le même objet.

Un exemple :

```
1 >>> liste = ['Paris', 'Rabbat', 'Abidjan']
2 >>> liste1 = liste #liste et liste1 sont maintenant identiques. Modifier liste1 \
3 impliquera également la modification de liste
4 >>> liste2 = liste[:] #On copie juste les valeurs de liste dans liste2. Ce sont \
5 donc deux listes distinctes
6 >>> liste1[1] = 'Dakar' #liste sera aussi modifiée
7 >>> liste
8 ['Paris', 'Dakar', 'Abidjan']
9 >>> liste1
10 ['Paris', 'Dakar', 'Abidjan']
11 >>> liste2 #liste2 n'est pas du tout modifiée
12 ['Paris', 'Rabbat', 'Abidjan']
13 >>>
```

Les alias permettent de donner de multiples noms à un objet. Dans l'exemple ci-dessus `liste2` est un alias.

8.5 Les méthodes split et join

- La méthode `split()` permet de convertir une chaîne de caractères en liste.
- La méthode `join()` permet de faire le contraire, c'est-à-dire convertir une liste en chaîne de caractères.

```
1 >>> chaine = "Salut les amis"
2 >>> chaine.split()
3 ['Salut', 'les', 'amis']
4 >>> liste = ['Salut', 'Monsieur']
5 >>> ' '.join(liste)
6 'Salut Monsieur'
7 >>>
```

Vous avez peut-être remarqué que j'employais le terme **méthodes** en lieu et place de fonctions et de même les méthodes applicables aux chaînes de caractères que nous avons vues dans ce chapitre étaient appelées un peu différemment par rapport aux fonctions que nous avons vues jusque-là (Ex : `chaine.upper()` en lieu et place de `upper(chaine)`). En effet, cela est en rapport avec la notion de programmation orientée objet en Python. Pour l'instant, faites-moi confiance et essayez de juste accepter que vous devez les appeler ainsi. Nous verrons dans un chapitre dédié le pourquoi.

8.6 Résumé

Dans ce chapitre, nous avons appris que :

- Les types non scalaires peuvent contenir plusieurs éléments à la fois, contrairement aux types scalaires qui eux ne peuvent contenir qu'un seul élément à la fois.
- Les chaînes de caractères et les tuples sont dits immuables étant donné qu'une fois définis, on ne peut pas changer la valeur d'un élément spécifique.
- Les slices nous permettent de sélectionner une partie ou toute une séquence de données.
- En tapant `help(str)` au niveau de l'interpréteur, nous obtenons une liste exhaustive de méthodes applicables aux chaînes de caractères.
- Il est possible de concaténer plusieurs tuples.
- Pour déclarer un tuple contenant un seul élément, il nous faut rajouter une **virgule** à la fin.
Ex : ("toto",)
- Les listes sont mutables, ce qui veut dire qu'on peut les modifier une fois créées. Cela représente un avantage considérable par rapport aux tuples.
- Le full slicing nous permet de copier une liste dans une autre sans pour autant créer un alias. Ce qui permettra ainsi de modifier l'une des deux listes indépendamment de l'autre puisqu'elles ne pointeront pas sur le même objet.
- La méthode `split()` permet de convertir une chaîne de caractères en liste.
- La méthode `join()` permet de faire le contraire, c'est-à-dire convertir une liste en chaîne de caractères.

9. Modules & Packages

Dans ce chapitre, nous continuerons notre exploration du concept du modularité en décrouvrant les notions de modules et de packages.

9.1 Les modules

9.1.1 Définition

Un module n'est rien d'autre qu'un **fichier**. Dans un module l'on peut regrouper plusieurs fonctions, variables et classes ayant un **rapport** entre elles. Par exemple dans un module `calcul`, on peut avoir une fonction qui permettra de faire l'addition de deux nombres passés comme arguments, une autre qui se chargera de faire la multiplication et ainsi de suite... Après lorsqu'on voudra travailler avec les fonctionnalités offertes par le module `calcul`, il suffira de **l'importer**.

9.1.2 La méthode import

Python met à notre disposition un grand nombre de modules sans qu'il ne soit nécessaire d'installer des bibliothèques supplémentaires. Ces modules font partie de ce que l'on appelle la **Python Standard Library** ou PSL (Librairie Standard Python pour les allergiques à la langue de Shakespeare). C'est le cas par exemple du module `math` qui comporte comme son nom l'indique des fonctions mathématiques, du module `random` qui contient un tas de fonctions ayant attrait à tout ce qui est aléatoire (le monde du hasard si vous le souhaitez).

Pour utiliser des fonctions contenues dans un module, il va falloir en premier lieu importer ledit module au risque d'avoir une erreur.

Pour ce faire il suffit de taper :

```
1 import nom_du_module
```

Dans notre cas, si nous voulons utiliser le module `math` il nous faudra donc saisir :

```
1 import math
```

Une fois cette instruction exécutée, Python va importer le module `math`, c'est-à-dire que toutes les fonctions, variables, classes contenues dans ce module seront dorénavant accessibles.

Au moment d'importer votre module, Python va lire (ou créer s'il n'existe pas) un fichier `.pyc`. Depuis la version 3.2 de Python, ce fichier se trouve dans un dossier `__pycache__`. Ce fichier est généré par Python et contient le code compilé (ou presque) de votre module. Il ne s'agit pas réellement de langage machine mais d'un format que Python décode un peu plus vite que le code que vous pouvez écrire. Tout ce que je viens de dire est purement technique, pas besoin de vous embrouiller avec tout ceci. Vous pourrez revenir lire cette partie si le moment où vous devez épater vos confrères se fait ressentir :).

9.1.3 Appel d'une fonction d'un module

Pour appeler une fonction d'un module, il suffit de taper le nom du module suivi d'un point ".", suivi du nom de la fonction que l'on souhaite appeler.

Exemple :

```
1 import math
2
3 print(math.ceil(1.5)) # Affichera 2
4 print(math.floor(1.5)) # Affichera 1
5 print(math.sqrt(4)) # Affichera 2
```

- La fonction `ceil` du module `math` renvoie la valeur entière immédiatement supérieure au nombre passé en argument.
- La fonction `floor` quant à elle renvoie la valeur entière immédiatement inférieure au nombre passé en argument.
- La fonction `sqrt` (SQuare RooT ou racine carrée) renvoie la racine carrée du nombre passé en argument.

9.1.4 La fonction `help`

La fonction `help` vous permet d'avoir de la documentation que ce soit sur un module ou une fonction. Elle prend en argument la fonction ou le module sur lequel vous souhaitez obtenir de l'aide.

On peut également afficher la spécification d'une fonction en tapant :

```
1 print(nom_fonction.__doc__)
```

Vous vous rappelez du quiz haha :) ?

Exemple :

```
1 import math
2 print( math.sqrt.__doc__ )
```

```
1 sqrt(x)
2
3 Return the square root of x.
4 >>>
```

En effet, la fonction **help** vous sera utile pour connaître l'ensemble des fonctions existantes pour un module et l'utilité de chacune d'elles.

Exemple :

```
1 >>> help("math")
2 Help on built-in module math:
3
4 NAME
5     math
6
7 DESCRIPTION
8     This module is always available. It provides access to the
9     mathematical functions defined by the C standard.
10
11 FUNCTIONS
12     acos(...)
13         acos(x)
14
15         Return the arc cosine (measured in radians) of x.
16
17     acosh(...)
18         acosh(x)
19
20         Return the hyperbolic arc cosine (measured in radians) of x.
21
22     asin(...)
23         asin(x)
24
25         Return the arc sine (measured in radians) of x.
26
27     asinh(...)
28         asinh(x)
29 ...
30 >>>
```

```

1 >>> help("math.ceil")
2 Help on built-in function ceil in math:
3
4 math.ceil = ceil(...)
5     ceil(x)
6
7     Return the ceiling of x as an int.
8     This is the smallest integral value >= x.
9 >>>

```



Les fameux underscores :)

Lorsqu'on abordera la notion de programmation orientée Objet en Python, faites-moi confiance, vous serez habitués à tous ces fameux underscores présents à gauche et à droite...

9.1.5 Créer un alias d'espace de noms

En tapant `import math`, nous créons un espace de noms dénommé **math**, contenant les variables et les fonctions du module `math`.

Un **espace de noms** ou **namespace** permet de regrouper certaines fonctions et variables sous un préfixe spécifique.

Pour changer l'espace de noms sous lequel le module sera importé il suffit de taper :

```

1 import nom_du_module as nouveau_nom

```

Les fonctions du module seront donc maintenant accessibles grâce à **nouveau_nom.fonction**

```

1 import math as m
2 print( m.fabs(-0.5) ) #Affiche 0.5
3
4 print( math.fabs(-0.5) ) #Affiche une erreur vu que le namespace math n'existe p\
5 lus!

```

```
1  Traceback (most recent call last):
2    File "<pyshell#2>", line 1, in <module>
3      print( math.fabs(-0.5) )
4  NameError: name 'math' is not defined
5  >>>
```

La fonction **fabs** du module **math** vous retourne la valeur absolue du nombre passe en argument. La valeur retournée est donc toujours positive.



Attention !

Pour rappel, le délimiteur de la partie entière et la partie décimale d'un nombre à virgule n'est pas une virgule mais un **point**!

9.1.6 Importation de fonction spécifiques

```
1  from nom_du_module import fonction1, fonction2, ...
```

Avec cette instruction, tout le module ne sera pas importé mais seulement les fonctions dont on aura indiqué le nom.

Si l'on veut importer tout un module avec cette syntaxe (ce qui revient à faire **import nom_du_module** avec tout de même une petite différence que nous verrons plus tard), il faudra taper :

```
1  from nom_du_module import *
```

L'étoile ***** signifie **tout**. En d'autres termes, depuis le module **nom_du_module** importer tout.

La seule différence entre les techniques **import** et **from-import**, est que si nous utilisons la méthode **from – import** pour importer des fonctions d'un module celles-ci sont mises directement dans l'espace de noms principal, ce qui revient à dire qu'il ne serait plus question de préfixer la fonction avec le nom du module concerné.

Voyons ceci avec des exemples pratiques :


```

1  >>> from math import ceil, sqrt
2  >>> sqrt(25)
3  5.0
4  >>> ceil(5.7)
5  6
6  >>> fabs(12) #Cette fonction n'a pas été importée
7  Traceback (most recent call last):
8    File "<pysHELL#3>", line 1, in <module>
9      fabs(12)
10 NameError: name 'fabs' is not defined
11 >>> math.ceil(0.5) #Plus besoin de préfixe math
12 Traceback (most recent call last):
13   File "<pysHELL#4>", line 1, in <module>
14     math.ceil(0.5)
15 NameError: name 'math' is not defined
16 >>>

```

9.1.7 Créer son propre module

Comme je vous l'avais dit, un module n'est rien d'autre qu'un simple fichier python. Créez donc un fichier `calcul.py` et ajoutez y le code suivant :

```

1  """Module contenant des fonctions pour les 4 opérations arithmétiques de base"""
2
3  def addition(a, b):
4      return a + b
5
6  def soustraction(a, b):
7      return a - b
8
9  def multiplication(a, b):
10     return a * b
11
12 def division(a, b):
13     if(b != 0):
14         return a / b
15     else :
16         print("Division impossible")

```

Vu que notre fichier a pour nom `calcul.py`, le nom de notre module sera `calcul`. Souvenez vous que ce nom est implicite et est fonction du nom que vous donnerez à votre fichier.

Maintenant que notre module a été créé, vous pouvez lancer votre programme comme si vous aviez affaire à un programme python habituel.

Ensuite au niveau de l'interpréteur de commandes, tapez par exemple :

```

1  >>> help("calcul")
2  Help on module calcul:
3
4  NAME
5      calcul - Module contenant des fonctions pour les 4 opérations arithmétiques \
6  de base
7
8  FUNCTIONS
9      addition(a, b)
10
11     division(a, b)
12
13     multiplication(a, b)
14
15     soustraction(a, b)
16
17  FILE
18     c:\users\honore.h\desktop\calcul.py
19
20  >>>

```

9.1.8 Faire des tests à l'intérieur de son module

Il faudra rajouter une structure conditionnelle pour pouvoir faire des tests d'exécution dans le module lui-même.

```

1  # test de la fonction addition
2  if(__name__ == "__main__"):
3      addition(1,5)

```

La variable `__name__` existe dès le lancement de l'interpréteur. Si elle vaut `__main__`, cela signifie que le fichier appelé est le fichier exécuté. Le code de test ne sera donc exécuté que lorsque qu'on double-cliquera sur le fichier contenant le module.

Pour des raisons d'efficacité, chaque module est importé une seule fois par ouverture de session de l'interpréteur. Ainsi si vous changez vos modules, vous devez redémarrer l'interpréteur ou si vous voulez juste tester un seul module de manière interactive, utilisez `imp.reload()`.

```
1 import imp
2 imp.reload(nom_du_module)
```

Encore une fois tout ce qui a été dit plus haut est purement technique et peut paraître très compliqué. Ne vous efforcez pas à vouloir tout retenir :).

Ce que j'aimerais que vous puissiez retenir par contre est que ce code :

```
1 # test de la fonction addition
2 if(__name__ == "__main__"):
3     addition(1,5)
```

sera exécuté si et seulement si c'est notre fichier **calcul.py** qui est exécuté. Dans le cas où on importe notre module, à partir d'un autre fichier, ce code ne sera pas exécuté.

Vous trouverez la plupart du temps sur le net des programmes Python basiques se présentant comme ceci :

```
1 def main():
2     print("Hello, World")
3
4 if(__name__ == "__main__"):
5     main()
```

Ne soyez donc pas du tout paniqué. Vous êtes maintenant en mesure de tout comprendre :).

- On a tout simplement défini une fonction **main()** contenant notre code principal.
- Ensuite, on spécifie que si le fichier appelé est le fichier exécuté (en gros si notre fichier est exécuté directement et non via importation) alors il faudra exécuter le contenu de la méthode **main()**.

Si cela vous semble un peu compliqué, donnez vous le temps d'assimiler tout ceci. Ne soyez pas trop dur envers vous-même. Vous pouvez faire une petite pause et pourquoi pas relire cette partie. La technique que j'utilise lorsque je n'arrive pas à comprendre quelque chose, c'est de refaire cette même chose plusieurs fois. A un moment donné cela devient un automatisme. Je ne veux surtout pas dire qu'il faudrait apprendre le code par cœur sans le comprendre, loin de là :). Ce que je veux dire, c'est que la répétition est très souvent un bon moyen d'apprentissage.

9.2 Les packages

9.2.1 Définition

- Les modules permettent de regrouper comme on l'a vu précédemment des fonctions, des classes, des variables...
- Un package quant à lui permet de regrouper un ou plusieurs modules, de préférence ayant un rapport entre eux. Ils permettent entre autres, de hiérarchiser nos programmes.
- Un package peut contenir plusieurs autres packages.
- A la différence des modules qui sont des fichiers, les packages quant à eux sont des répertoires (des dossiers). Certains programmeurs appellent les packages, des bibliothèques.

9.2.2 Importer un package

Un package s'importe de la même manière qu'un module. On utilise donc soit les mots-clés **from-import** combinés ou soit le mot-clé **import** tout seul.

```
1 import nom_package
2 from nom_package import mon_module
```

9.2.3 Créer un package

On commence par créer un répertoire dans le même dossier que notre programme python.

Pour que Python sache que ce répertoire est un package, il faudra créer à l'intérieur de ce dernier un fichier nommé `__init__.py`. Le contenu de ce fichier peut être vide mais il est obligatoire qu'il puisse exister pour que Python sache qu'il s'agit d'un package.

Vous pouvez par la suite créer dans ce répertoire vos modules ou créer d'autres sous-packages qui devront également contenir un fichier `__init__.py`.

Voilà c'est ici que se termine notre exploration du concept de modularité.

9.3 Résumé

- Une fonction vous permet de regrouper un bloc d'instructions que vous pourrez appeler comme vous le souhaitez et autant de fois que vous le souhaitez.
- Un module n'est rien d'autre qu'un fichier qui va nous permettre de regrouper nos fonctions, variables et classes ayant un rapport entre elles.
- Un package est un dossier qui va permettre de regrouper un ensemble de modules ayant un rapport entre eux.
- Un package peut contenir d'autres packages.

10. Jeu de capitales

10.1 Le principe du jeu

Le jeu que nous allons réaliser dans ce chapitre sera un tout peu plus fun que ceux des chapitres précédents. Il consistera à afficher une série aléatoire de pays et demander à l'utilisateur de nous fournir les capitales respectives de ces pays. Si l'utilisateur nous donne la bonne réponse, nous afficherons tout simplement "Bonne réponse" et incrémenterons son score actuel qui lui sera affiché à la fin de la série. Si par contre, la réponse donnée est fausse, ce sera l'occasion pour nous de lui démontrer notre super niveau en culture générale en lui affichant la réponse qu'il aurait fallu donner.

Ainsi si je choisis comme pays la **France** et que je vous demande sa capitale, vous me répondrez je parie **Paris**. (Belles rimes j'avoue :))

Pour que ce soit un peu plus clair, je vous présente un exemple d'exécution de notre programme :

```
1  >>>
2  Quelle est la capitale de ce pays: Gabon?
3  Libreville
4  Bonne réponse!
5  Quelle est la capitale de ce pays: Burkina Faso?
6  OuAGADOUgou
7  Bonne réponse!
8  Quelle est la capitale de ce pays: Mozambique?
9  Je ne sais pas :)
10 Mauvaise réponse! Il fallait répondre: Maputo
11 Quelle est la capitale de ce pays: France?
12 Paris
13 Bonne réponse!
14 Quelle est la capitale de ce pays: Sénégal?
15 Dakar
16 Bonne réponse!
17 C'est terminé! <<Score : 4/5 >>
18 >>>
```

Alors comme dans le [chapitre 4](#), bien avant de se lancer à tête baissée dans l'écriture de notre script, il va nous falloir répondre dans un premier temps à un bon nombre de questions.

Comment stocker en mémoire nos différents pays et leurs capitales respectives ? Comment choisir une valeur de manière aléatoire en Python ? Comment comparer deux valeurs (Nous en aurons besoin afin de déterminer si la réponse entrée par l'utilisateur est exacte ou non) ?

Ce qui est intéressant dans ces questions, c'est que vous avez **quasiment** toutes les réponses :).

- Comment stocker en mémoire nos différents pays et leurs capitales respectives ? Un dictionnaire pourquoi pas fera l'affaire.
- Comment comparer deux valeurs ? Les opérateurs de comparaison nous seront d'une grande utilité.
- Comment choisir une valeur de manière aléatoire en Python ? Le module **random**.



Jamais tu ne nous as parlé de ce module random ?

Regardez le titre de la prochaine section. Ah ah :)

10.2 Le module random

Lors du précédent chapitre, nous nous sommes attardés sur un seul module : le module **math**. A l'instar de ce module, Python met à votre disposition un tas d'autres modules : le module **datetime** vous permettant de gérer le temps (date, heure...), le module **fractions** vous permettant de gérer les fractions, le module **ftplib** pour vous amuser un tout petit peu avec le protocole ftp et bien d'autres.



Liste de tous les modules

Pour avoir une liste exhaustive des modules python disponibles visiter ce lien <https://docs.python.org/3/py-modindex.html>¹⁵

Dans cette section, nous nous intéresserons particulièrement au module **random** qui comme vous l'aurez peut être deviné va nous permettre de générer des nombres pseudo-aléatoires.

En théorie, un ordinateur ne connaît pas le hasard. Mais tout ce que vous devez savoir, c'est qu'il fera de son mieux pour simuler ce monde du hasard. Des algorithmes hyper-sophistiqués ont donc déjà été mis en place pour gérer tous ces mushinishis (des valeurs de manière aléatoire). Vous n'aurez donc pas à vous en soucier. Utilisez la méthode appropriée et vous aurez un résultat approprié. Aussi simple que cela !

Assez parlé, découvrons par la pratique ce fameux module random.

Bien avant de faire nos petits tests, je tiens tout d'abord à vous faire une petite déclaration :

La documentation sur le module random est disponible à [cette adresse](#)¹⁶ Celle-ci contient également des exemples concrets d'utilisation.

15. <https://docs.python.org/3/py-modindex.html>

16. <https://docs.python.org/3/library/random.html#module-random>

Expliquer donc en long et en large chacune des fonctions de ce module, serait à mon avis anti-pédagogique ! Si un jour vous postez sur un forum un message du genre : A quoi sert la fonction `choice` du module `random` ? ou encore combien d'arguments prend la fonction `randrange` du module `random` ? Vous aurez sans risque de me tromper cette réponse : **RTFM (Read The Fucking Manual)**. Petite traduction : Merci de bien vouloir lire la documentation SVP. (Ceux qui comprennent l'anglais se mettront à rire :))

Ce que je veux donc dire, c'est que vous devez toujours vous servir de la documentation en premier recours si vous bloquez sur quoi que ce soit. En général, vous y trouverez votre réponse.

10.2.1 La fonction `random`

Synopsis: `random.random()`

La fonction `random` du module `random` nous retourne une valeur aléatoire (ici un nombre réel) entre 0.0 et 1.0. 1.0 étant exclus. Ainsi mathématiquement parlant on a cet intervalle $[0.0, 1.0[$.

```
1 >>> import random #On importe le module random
2 >>> random.random()
3 0.6879157636029883
4 >>> random.random()
5 0.739628459893686
6 >>> random.random()
7 0.10960434825810839
8 >>> random.random()
9 0.9443851801331767
10 >>> random.random()
11 0.22761731499433735
12 >>> random.random()
13 0.9221146293989106
14 >>> random.random()
15 0.40612880708816845
16 >>> random.random()
17 0.5208338016201847
18 >>>
```

10.2.2 La fonction `randrange`

Synopsis: `random.randrange(stop)` ou `random.randrange(start, stop[, step])`

La fonction `randrange` fonctionne à peu près de la même manière que la fonction `range` que nous avons eu à étudier lors du chapitre consacré aux boucles. Pour preuve, elles ont les mêmes paramètres : le début, la fin, le pas.

Voici quelques exemples :

```

1  >>> import random #Toujours importer le module avant de l'utiliser
2  >>> random.randrange(3) #génère un nombre aléatoire entre 0,1,2
3  1
4  >>> random.randrange(3)
5  2
6  >>> random.randrange(3)
7  2
8  >>> random.randrange(4, 9) #génère un nombre aléatoire entre 4,5,6,7,8
9  7
10 >>> random.randrange(0, 1) #génère un nombre aléatoire entre 0 et 1 [1 exclus]. \
11 On aura donc toujours 0
12 0
13 >>> random.randrange(0, 1)
14 0
15 >>> random.randrange(0, 10, 3) #génère un nombre aléatoire entre 0,3,6,9
16 0
17 >>> random.randrange(0, 10, 3)
18 6
19 >>> random.randrange(0, 10, 3)
20 9
21 >>>

```

10.2.3 La fonction randint

Synopsis: `random.randint(a, b)`

La fonction `randint` vous permet de générer une valeur entre `a` et `b`, mais cette fois-ci `b` est inclus dans l'intervalle. L'intervalle sera donc `[a, b]`.

```

1  >>> import random
2  >>> random.randint(0,1)
3  1
4  >>> random.randint(0,1)
5  0
6  >>> random.randint(10,100)
7  47
8  >>> random.randint(10,100)
9  62
10 >>>

```

10.2.4 La fonction choice

Synopsis: `random.choice(sequence)`

La fonction `choice` que nous allons utiliser dans notre programme pour le jeu de capitales vous retourne une valeur aléatoire à partir d'une séquence de valeurs non vide fournie en argument. On pourra donc passer en argument un tuple, une liste, une chaîne de caractères...



Pourquoi une chaîne de caractères ?

Souvenez-vous qu'une chaîne de caractères n'est rien d'autre qu'une séquence de caractères.

```

1  >>> random.choice('123456')
2  '5'
3  >>> random.choice('123456')
4  '4'
5  >>> random.choice('123456')
6  '5'
7  >>> random.choice('123456')
8  '2'
9  >>> random.choice(1,2,4) #Ici les parenthèses sont obligatoires pour un tuple
10 Traceback (most recent call last):
11   File "<pyshell#21>", line 1, in <module>
12     random.choice(1,2,4)
13 TypeError: choice() takes 2 positional arguments but 4 were given
14 >>> random.choice((1,2,4))
15 2
16 >>> random.choice((1,2,4))
17 4
18 >>> random.choice(["toto","tata","mama"])
19 'mama'
20 >>> random.choice(["toto","tata","mama"])
21 'toto'
22 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
23 >>> random.choice(data)
24 Traceback (most recent call last):
25   File "<pyshell#4>", line 1, in <module>
26     random.choice(data)
27   File "C :\Python34\lib\random.py", line 256, in choice
28     return seq[i]
29 KeyError: 0
30 >>> random.choice(data.keys())
31 Traceback (most recent call last):
32   File "<pyshell#13>", line 1, in <module>
33     random.choice(data.keys())

```

```

34 File "C : \Python34 \lib \random.py", line 256, in choice
35     return seq[i]
36 TypeError: 'dict_keys' object does not support indexing
37 >>> data.keys()
38 dict_keys(['Gabon', 'Chine', 'France'])
39 >>> data.values()
40 dict_values(['Libreville', 'Pekin', 'Paris'])
41 >>> random.choice(list(data))
42 'Gabon'
43 >>> random.choice(list(data.keys()))
44 'Chine'
45 >>> random.choice(list(data.values()))
46 'Libreville'

```

Les premiers exemples sont comme vous pouvez le voir, assez faciles à comprendre. Les choses commencent à se compliquer à la ligne où nous déclarons notre fameux dictionnaire **data**.

En tapant :

```

1 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
2 >>> random.choice(data)

```

on a une erreur parce que tout simplement un dictionnaire n'est pas une séquence de valeurs. Rappelez-vous je vous avais dit qu'un dictionnaire n'était rien d'autre qu'un ensemble de couples clé-valeur. L'erreur est donc dans ce cas justifiée.

Par contre ici :

```

1 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
2 >>> random.choice(data.keys())

```

C'est un tout petit peu bizarre qu'on ait une erreur n'est-ce pas ? En effet, on demande à récupérer la liste des clés de notre dictionnaire. Cette liste sera donc bel et bien une séquence de valeurs (chaque valeur représentera une clé de notre dictionnaire). Mais l'on s'aperçoit très rapidement avec l'instruction suivante, du pourquoi de cette erreur.

En tapant :

```

1 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
2 >>> data.keys()
3 dict_keys(['Gabon', 'Chine', 'France'])

```

Ce qui est retourné n'est pas tout simplement une liste, mais un fameux type special **dict_keys**. Ainsi, pour résoudre ce problème, il nous faudra convertir notre **dict_keys** en liste.

Pour ce faire, il suffira d'utiliser la fonction **list** mise à notre disposition gratuitement par Python.

```

1 >>> un_tuple = (1,2,3,4)
2 >>> un_tuple_devient_une_liste = list(un_tuple)
3 >>> un_tuple_devient_une_liste
4 [1, 2, 3, 4]
5 >>> list({1: "un", 2: "deux"})
6 [1, 2]

```

Comme vous le voyez, la fonction `list` appliquée directement à un dictionnaire ne récupère que les clés et non les valeurs.

Il existe de même des fonctions `tuple` et `dict` qui vous l'aurez deviné permettront de ...

```

1 >>> tuple([1,2,3,4])
2 (1, 2, 3, 4)
3 >>> tuple({1: "un", 2: "deux"})
4 (1, 2)
5 >>> un_dictionnaire_vide = dict()
6 >>> une_liste_vide = list()
7 >>> un_tuple_vide = tuple()

```

Vous pouvez maintenant comprendre aisément pourquoi les instructions suivantes ont fonctionné :

```

1 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
2 >>> random.choice(list(data))
3 'Gabon'
4 >>> random.choice(list(data.keys())) #Similaire à l'instruction précédente
5 'Chine'
6 >>> random.choice(list(data.values()))
7 'Libreville'

```

10.2.5 La fonction `sample`

Synopsis: `random.sample(population, k)`

La fonction `sample` retourne une liste de taille `k` composée d'**éléments uniques** choisis de façon aléatoire au niveau de la séquence **population**. Voyez un peu cette méthode comme un moyen de récupérer des échantillons dans une séquence donnée. Encore une fois, si vous voulez avoir beaucoup plus d'informations, n'oubliez pas que la documentation n'attend que vous :).

```
1 >>> random.sample(range(10), 3)
2 [8, 3, 2]
3 >>> random.sample([2,345, 5, "toto"], 3)
4 [2, 5, 'toto']
5 >>> random.sample("tatattoto", 4)
6 ['t', 't', 'o', 'a']
7 >>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
8 >>> random.sample(data.keys(), 2) #Cette fois-ci on peut utiliser directement da\
9 ta.keys()
10 ['Chine', 'Gabon']
11 >>> random.sample(list(data), 2)
12 ['Gabon', 'France']
```

Je crois que nous pouvons nous arrêter ici pour ce qui est de notre petite exploration du module random. J'espère que vous avez apprécié cette courte visite guidée et tout le personnel ose espérer vous revoir pour une prochain voyage avec Python Airways...

10.3 Challenge

A présent vous avez toutes les cartes en main pour écrire notre jeu de capitales. Ce que je vous propose c'est d'écrire votre propre programme et après de le comparer au mien. Il existe plusieurs solutions à un problème donné. Suivez donc votre logique et produisez moi un programme mielleux. Je vous fais confiance.

N'hésitez surtout pas a relire le principe du jeu afin de vous rappelez de ce dont il était question. Bonne chance !

10.4 Exemple de Solution

J'espère que vous avez eu une solution à notre problème ou au mieux vous avez essayé. Si tel n'est pas le cas, je vous conseille de le faire. Il n'y a qu'en forgeant que l'on devient forgeron. Vous devez donc pratiquer afin d'assimiler toute cette théorie. J'ose espérer que j'ai été assez convaincant cette fois-ci pour les derniers rescapés.

Voici la solution que je vous propose. Bien sûr, elle n'est pas parole d'Évangile :)

```
1  # Jeu de capitales
2  # Python par la pratique
3
4  # Nous aurons besoin uniquement de la fonction sample, pas besoin
5  # alors d'importer tout le module
6  from random import sample
7
8  #Ensemble de pays et leurs capitales respectives
9  capitales = {
10     "Senegal" : "Dakar",
11     "Nigeria": "Lagos",
12     "France": "Paris",
13     "Gabon" : "Libreville",
14     "Burkina Faso": "Ouagadougou",
15     "Allemagne" : "Berlin",
16     "Belgique" : "Bruxelles",
17     "Qatar" : "Doha",
18     "Zimbabwe" : "Harare",
19     "Perou" : "Lima",
20     "Mozambique" : "Maputo"
21 }
22
23
24 # On utilise ici une constante.
25 # Ainsi, si nous voulons modifier plus tard le
26 # nombre de questions, nous le ferons à cet unique
27 # emplacement et notre changement sera répercuté partout ailleurs
28 NBRE_TOTAL_DE_QUESTIONS = 5
29
30 #Contiendra le score de l'utilisateur
31 score = 0
32
33 #On récupère une liste aléatoire de [NBRE_TOTAL_DE_QUESTIONS] pays
34 liste_pays = sample(list(capitales), NBRE_TOTAL_DE_QUESTIONS)
35
36 for pays in liste_pays:
37     print("Quelle est la capitale de ce pays : " + pays + " ?")
38     reponse = input() #on récupère la réponse de l'utilisateur
39
40     #La méthode lower() nous permet de convertir une chaine de caractères
41     #en minuscules. On le fait pour qu'ainsi si l'utilisateur tape OuagadoUgou e\
42     n lieu
```

```

43     #et place de Ouagadougou, cela fonctionnera également.
44     if(reponse.lower() == capitales[pays].lower()):
45         print("Bonne réponse!")
46         score += 1 #On incrémente son score
47     else:
48         print("Mauvais réponse! Il fallait répondre : " + capitales[pays])
49
50 print("C'est terminé ! << Score : ", score, "/", NBRE_TOTAL_DE_QUESTIONS, " >>")

```

10.5 Faites vous confiance !

Comme vous l'avez vu, je n'ai pas utilisé la fonction `choice` comme je l'avais indiqué. C'était en effet un piège, histoire de voir si vous allez vous efforcer à utiliser cette fonction alors qu'elle n'est pas forcément la plus adaptée dans notre cas de figure !

Si nous avions utilisé la fonction `choice`, nous aurons été dans l'obligation de gérer pas nous-mêmes le fait qu'une même question ne puisse pas être affichée plus d'une fois dans une série de questions. Ainsi, notre code source final en sera un tout petit alourdi. Je vous laisse juger par vous-même.

```

1  #On utilise cette fois-ci la fonction choice
2  from random import choice
3
4  #Ensemble de pays et leurs capitales respectives
5  capitales = {
6      "Senegal" : "Dakar",
7      "Nigeria": "Lagos",
8      "France": "Paris",
9      "Gabon" : "Libreville",
10     "Burkina Faso": "Ouagadougou",
11     "Allemagne" : "Berlin",
12     "Belgique" : "Bruxelles",
13     "Qatar" : "Doha",
14     "Zimbabwe" : "Harare",
15     "Perou" : "Lima",
16     "Mozambique" : "Maputo"
17 }
18
19 NBRE_TOTAL_DE_QUESTIONS = 5
20 score = 0
21
22 #Contiendra la liste des pays déjà choisis afin de ne pas
23 #afficher plus d'une fois la même question dans une même série

```

```

24 paysDejaChoisis = []
25
26 #On récupère la liste des pays
27 listePays = list( capitales.keys() )
28
29 #On répète ce bloc [NBRE_TOTAL_DE_QUESTIONS] fois
30 for i in range(0, NBRE_TOTAL_DE_QUESTIONS):
31     pays = choice(listePays) #On choisi un pays de manière aléatoire
32
33     #Si le pays fait déjà partie de la liste, on en choisi un autre, et ce,
34     #tant que nous n'avons pas trouvé un pays qui ne fait pas déjà partie de la l\
35 iste.
36     while pays in paysDejaChoisis:
37         pays = choice(listePays)
38
39     #On ajoute le pays qui vient d'être sélectionné
40     #à notre liste de pays déjà choisis pour ne
41     #pas avoir à le choisir une fois de plus
42     paysDejaChoisis.append(pays)
43
44     print("Quelle est la capitale de ce pays : " + pays + " ?")
45     reponse = input()
46
47     #La méthode lower() nous permet de convertir une chaine de caractères
48     #en minuscules. On le fait pour qu'ainsi si l'utilisateur tape OuagadoUgou e\
49 n lieu
50     #et place de Ouagadougou, cela fonctionnera également.
51     if(reponse.lower() == capitales[pays].lower()):
52         print("Bonne réponse!")
53         score += 1 #On incrémente son score
54     else:
55         print("Mauvais réponse! Il fallait répondre : " + capitales[pays])
56
57 print("C'est terminé ! << Score : ", score, "/", NBRE_TOTAL_DE_QUESTIONS, " >>")

```

J'ai eu à documenter le code source afin que vous puissiez comprendre chacune des lignes. Si vous avez bien suivi jusque-là, comprendre ce code devrait être en réalité chose aisée pour vous.

10.6 Petit exercice

Modifier notre programme afin de pouvoir donner la possibilité à l'utilisateur de pouvoir rejouer. Ainsi, si l'utilisateur termine une partie, on lui demande s'il souhaite rejouer ou non. S'il répond par

l'affirmative alors on lui prépare une nouvelle partie, dans le cas contraire on lui dira gentiment **Bye Bye**.

Vous pouvez donc utiliser ce petit test avec encore une fois notre fameuse méthode `lower`. Nous en reparlerons au niveau du chapitre suivant. Ne vous inquiétez donc pas.

```
1 reponse = input("Voulez-vous rejouer (O/N) ?")
2
3 if(reponse.lower() == 'o'):
4     #L'utilisateur veut rejouer
5 else:
6     #On lui dit Bye Bye
```

En utilisant la méthode `lower`, cela nous évite d'avoir à faire deux tests.

```
1 reponse = input("Voulez-vous rejouer (O/N) ?")
2
3 if(reponse == 'O' or reponse == 'o'):
4     #L'utilisateur veut rejouer
5 else:
6     #On lui dit Bye Bye
```

Rendez ensuite votre code modulaire en le découpant en de petites fonctions.

10.7 Résumé

Dans ce chapitre, nous avons appris que :

- En théorie, un ordinateur ne connaît pas le hasard. Il fait de son mieux pour simuler le monde du hasard.
- Le module `random` nous permet de générer des valeurs pseudo-aléatoires selon une variété de distribution.
- Nous avons la possibilité d'utiliser les fonctions `list`, `tuple`, `dict` pour créer respectivement des listes, des tuples et des dictionnaires.
- Vous devez toujours vous servir de la documentation en premier recours si vous bloquez sur quoi que ce soit.
- Une constante est une variable dont le contenu ne sera pas amené à changer au cours de l'exécution d'un notre programme.
- Il existe toujours plusieurs solutions à problème donné.

11. Les fichiers et les exceptions

Travailler sur des listes, des tuples... c'est bien ! Mais parfois ce n'est pas suffisant. Très souvent vos données proviendront d'un fichier, d'une base de données, depuis Internet etc.

Je peux comprendre qu'il soit un tout petit peu difficile à l'heure actuelle pour vous de percevoir l'utilité des fichiers vu que nous avons travaillé jusque-là que sur de petites données, mais croyez-moi vous vous en rendrez compte au fur et à mesure que nous avancerons dans ce chapitre.

Il sera très facile de manipuler les fichiers avec Python et ceci n'est pas une surprise avec Python :). Python met à notre disposition tout un arsenal de fonctions très simples nous permettant de manipuler les fichiers. Toutefois vu que la manipulation des fichiers est un peu délicate, vous serez amenés à rencontrer quelques petits problèmes dus à des situations exceptionnelles (vous avez spécifié par exemple un nom de fichier incorrect par exemple) et il nous faudra gérer tout ceci. Heureusement grâce à la gestion des exceptions présentes au niveau du langage Python, cela sera un vrai jeu d'enfants.

11.1 Les données sont très souvent externes

Comme je vous le disait en introduction, généralement, la plupart de nos programmes recevront des données provenant de sources externes. Ces données seront donc récupérées, traitées, stockées, affichées, imprimées, transférées... En gros tout dépendra de vous !

A ce stade, j'ose supposer que vous êtes maintenant des experts en ce qui concerne le traitement de données internes. Qu'en est-il alors des données provenant d'une source externe ? Comment récupérer le contenu d'un fichier par exemple ?

Pour bien comprendre l'utilité des fonctions qui vont suivre, je vais vous demander de vous faire une image de la procédure de manipulation des fichiers d'un point de vue graphique.

Si je veux lire le contenu d'un fichier, qu'est-ce que je dois faire ?

- Premièrement chercher le fichier et l'ouvrir,
- Ensuite lire son contenu.
- Faire ce que je veux avec ce contenu (le copier, le couper, le modifier...)
- Et si je juge qu'il ne me sera plus d'une grande utilité, je peux décider de le fermer.

C'est exactement de cette manière que Python procède. Il mettra donc à votre disposition une fonction pour chacune des ces étapes là.

11.2 La fonction open

Le contenu d'un fichier n'est rien d'autre qu'un ensemble de lignes. En effet, lorsque vous lisez le contenu d'un fichier en Python, ce dernier vous sera fourni ligne par ligne. Vous lirez donc qu'une seule ligne à la fois.

La fonction `open` comme son nom l'indique nous permettra d'ouvrir un fichier. Si elle est ensuite combinée avec la boucle `for` la lecture du contenu en est largement facilitée.

Si vous voulez ouvrir un fichier, il suffira d'écrire : `open(chemin_menant_au_fichier)`

```
1 >>> fichier = open("c :/Users/honore.h/Desktop/mon_fichier.txt")
```

Lorsque vous travaillez sur les fichiers la procédure sera la suivante :

```
1 fichier = open("c :/Users/honore.h/Desktop/mon_fichier.txt")
2 # Faire quelque chose avec les données récupérées depuis le fichier
3 fichier.close() #Fermer le fichier
```



N'oubliez jamais de fermer le fichier après avoir terminé le traitement des données. Cela vous permettra de libérer un tant soit peu de la ressource mémoire. Please guys, don't forget it !

Essayons ensuite de voir ce que contient la variable `fichier`

```
1 >>> fichier = open("c :/Users/honore.h/Desktop/mon_fichier.txt")
2 >>> fichier
3 <_io.TextIOWrapper name='c :/Users/honore.h/Desktop/mon_fichier.txt' mode='r' enc\
4 oding='cp1252'>
5 >>>
```

Tout ce que vous devez remarquer ici c'est que notre variable `fichier` ne contient pas pour l'instant le contenu de notre fichier. Nous disposons de ce qu'on appelle un **iterator** qui va nous permettre de pouvoir parcourir notre fichier ligne après ligne. Qui dit parcourir fait penser automatiquement aux boucles. Mais ne soyez pas pressés, voyons les choses étape par étape.

Premièrement voyons voir comment faire pour seulement spécifier le nom de notre fichier et non ce long chemin "c :/Users/honore.h/Desktop/mon_fichier.txt".

11.2.1 Fichier capitales.txt

Veuillez créer un fichier `capitales.txt` au niveau d'un dossier de votre choix et mettez y comme contenu :

```

1  Dakar-Sénégal
2  Lagos-Nigeria
3  Paris-France
4  Libreville-Gabon
5  Ouagadougou-Burkina Faso
6  Berlin-Allemagne
7  Bruxelles-Belgique
8  Doha-Qatar
9  Harare-Zimbabwe
10 Amsterdam-Pays-Bas
11 Lima-Pérou
12 Maputo-Mozambique
13 Monaco
14 Monrovia-Liberia
15 Panama
16 Rome-Italie

```

Un peu de pratique :)

```

1  >>> import os #On importe le module OS depuis la librairie standard
2  >>> os.getcwd() #On détermine le répertoire courant
3  'C :\\Python34'
4  >>> os.chdir("c :/Users/honore.h/Desktop/") #On choisit le répertoire qui contien\
5  t notre fichier capitales.txt comme nouveau répertoire de travail
6  >>> os.getcwd() #Nous confirmons que nous sommes maintenant dans le bon réperto\
7  ire
8  'c :\\Users\\honore.h\\Desktop'
9  >>> fichier = open('capitales.txt') #On ouvre notre fichier et on affecte la val\
10 eur retournée à une variable fichier
11 >>> print(fichier.readline()) #On utilise la méthode "readline()" pour lire une \
12 ligne au niveau de notre fichier, puis on utilise la fonction "print()" pour aff\
13 icher la ligne récupérée.
14 Sénégal-Dakar
15
16 >>> print(fichier.readline(), end='') #Pour ne pas avoir cette ligne vide de tro\
17 p, on rajoute l'argument end
18 Nigeria-Lagos
19 >>> fichier.seek(0) #Utilisons la méthode "seek()" pour placer le curseur au débu\
20 t de notre fichier.
21 0
22 >>> for ligne in fichier: #Juste une boucle for pour lire toutes les lignes au n\
23 iveau de notre fichier

```

```
24     print(ligne, end=' ')
25
26
27     Dakar-Sénégal
28     Lagos-Nigeria
29     Paris-France
30     Libreville-Gabon
31     Ouagadougou-Burkina Faso
32     Berlin-Allemagne
33     Bruxelles-Belgique
34     Doha-Qatar
35     Harare-Zimbabwe
36     Amsterdam-Pays-Bas
37     Lima-Pérou
38     Maputo-Mozambique
39     Monaco
40     Monrovia-Liberia
41     Panama
42     Rome-Italie
43 >>> fichier.close() #N'oublions pas de fermer notre fichier vu que nous avons te\
44 rminé.
45 >>>
```



Le module os

Le module `os` est un module qui vous permet de manière portable d'utiliser des fonctionnalités propres aux systèmes d'exploitations. Si vous voulez avoir plus de détails sur ce dernier, comme des grands, vous pouvez vous servir de la documentation officielle de Python <https://docs.python.org/2/library/os.html>¹⁷.

11.3 Contenu du fichier `capitales.txt`

Si vous ouvrez le fichier `capitales.txt`, vous remarquerez que sur chaque ligne nous avons tout simplement le nom d'un pays suivi de celui de sa capitale. Toutefois, il est important de noter qu'il suit un format bien précis :

pays-capitale

Avec ce format, nous pouvons par exemple récupérer uniquement le pays en utilisant la méthode `split()` que nous avons déjà étudiée.

17. <https://docs.python.org/2/library/os.html>

```
1 >>> texte = "Maman : Je t'aime mon fils."
2 >>> texte.split(":")
3 ['Maman', " Je t'aime mon fils."]
4 >>> texte = "Paris-France"
5 >>> texte.split("-")
6 ['Paris', 'France']
7 >>>
```

Très simple n'est-ce pas ? Vous n'avez qu'à fournir en argument le séparateur et la méthode `split()` fera le reste du travail. Par défaut, si aucun argument n'est fourni l'espace sera utilisé comme séparateur.

```
1 >>> texte = "Je suis un homme sans femme"
2 >>> texte.split()
3 ['Je', 'suis', 'un', 'homme', 'sans', 'femme']
4 >>>
```

Précédemment lorsque nous avons utilisé les deux points comme séparateur, cette liste était retournée.

```
1 ['Maman', " Je t'aime mon fils."]
```

Pour récupérer le nom et le message, on peut donc procéder ainsi :

```
1 >>> texte = "Maman : Je t'aime mon fils."
2 >>> donnees = texte.split(":") #Les parenthèses sont optionnelles
3 >>> nom = donnees[0]
4 >>> message = donnees[1]
5 >>> nom
6 'Maman'
7 >>> message
8 " Je t'aime mon fils."
9 >>>
```

Encore mieux, nous pouvons faire d'une pierre deux coups et récupérer à la fois le nom et le message en une seule ligne comme ceci :

```
1 >>> texte = "Maman : Je t'aime mon fils."
2 >>> (nom, message) = texte.split(":") #Les parenthèses sont optionnelles
3 >>> nom
4 'Maman'
5 >>> message
6 " Je t'aime mon fils."
7 >>>
```

Si nous appliquons ce que nous venons d'apprendre au contenu de notre fichier capitales.txt, nous pouvons produire quelque chose comme ceci :

```
1 >>> fichier = open('capitales.txt')
2 >>> for ligne in fichier:
3     capitale, pays = ligne.split("-")
4     print(capitale, end=' est la capitale de ')
5     print(pays, end='')
6
7
8 Dakar est la capitale de Sénégal
9 Lagos est la capitale de Nigeria
10 Paris est la capitale de France
11 Libreville est la capitale de Gabon
12 Ouagadougou est la capitale de Burkina Faso
13 Berlin est la capitale de Allemagne
14 Bruxelles est la capitale de Belgique
15 Doha est la capitale de Qatar
16 Harare est la capitale de Zimbabwe
17 Traceback (most recent call last):
18   File "<pyshell#37>", line 2, in <module>
19     capitale, pays = ligne.split("-")
20 ValueError: too many values to unpack (expected 2)
21 >>>
```



Eh oui, je sais très bien que nos phrases violent un tout petit peu la langue de Molière, mais ça restera entre nous :).

Comme vous pouvez le voir, nous avons droit à une belle erreur et cela est tout à fait normal si l'on jette un petit coup d'œil à notre fichier :

```
1  Dakar-Sénégal
2  Lagos-Nigeria
3  Paris-France
4  Libreville-Gabon
5  Ouagadougou-Burkina Faso
6  Berlin-Allemagne
7  Bruxelles-Belgique
8  Doha-Qatar
9  Harare-Zimbabwe
10 Amsterdam-Pays-Bas
11 Lima-Pérou
12 Maputo-Mozambique
13 Monaco
14 Monrovia-Liberia
15 Panama
16 Rome-Italie
```

La ligne numéro **10** contient deux **tirets**. Ainsi lorsqu'on fera le `split("-")`, nous n'aurons pas en résultat une liste à deux éléments mais plutôt une liste contenant 3 éléments et vu que notre programme ne sait quoi faire avec cette troisième valeur une exception de type **ValueError** sera levée.

Pour preuve :

```
1  >>> texte = "Amsterdam-Pays-Bas"
2  >>> texte.split("-")
3  ['Amsterdam', 'Pays', 'Bas']
4  >>>
```

11.4 Un code plus sûr

Pour résoudre notre problème, nous pouvons avoir comme première idée de changer pourquoi pas de séparateur : utiliser **les deux points** en lieu et place du **tiret**. Cela est tout à fait exact je vous l'accorde, mais avouez que tout ce qu'on aura fait, c'est fuir le problème et non le résoudre :).

Jetons plutôt un coup d'œil au niveau de la documentation sur la méthode `split()` afin de savoir si oui ou non nous avons la possibilité de préciser le nombre de découpage maximum.

```

1 >>> texte = "Amsterdam-Pays-Bas"
2 >>> help(texte.split)
3 Help on built-in function split:
4
5 split(...) method of builtins.str instance
6     S.split(sep=None, maxsplit=-1) -> list of strings
7
8     Return a list of the words in S, using sep as the
9     delimiter string.  If maxsplit is given, at most maxsplit
10    splits are done.  If sep is not specified or is None, any
11    whitespace string is a separator and empty strings are
12    removed from the result.
13
14 >>>

```

La méthode **split()** prend en effet un second argument **max_split** qui nous permettra de préciser le nombre de découpage maximum. Ainsi dans notre cas, nous renseignerons comme valeur 1 afin de n'avoir qu'un seul découpage et donc deux (2) informations.

```

1 >>> texte = "Amsterdam-Pays-Bas"
2 >>> texte.split("-", 1)
3 ['Amsterdam', 'Pays-Bas']
4 >>> len( texte.split("-", 1) )
5 2
6 >>>

```

En utilisant donc ce second paramètre, notre **premier** problème est résolu.

```

1 >>> fichier = open("capitales.txt")
2 >>> for ligne in fichier:
3     capitale, pays = ligne.split("-", 1)
4     print(capitale, end=' est la capitale de ')
5     print(pays, end='')
6
7
8 Dakar est la capitale de Sénégal
9 Lagos est la capitale de Nigeria
10 Paris est la capitale de France
11 Libreville est la capitale de Gabon
12 Ouagadougou est la capitale de Burkina Faso
13 Berlin est la capitale de Allemagne
14 Bruxelles est la capitale de Belgique

```



```
15 Doha est la capitale de Qatar
16 Harare est la capitale de Zimbabwe
17 Amsterdam est la capitale de Pays-Bas
18 Lima est la capitale de Pérou
19 Maputo est la capitale de Mozambique
20 Traceback (most recent call last):
21   File "<pyshell#53>", line 2, in <module>
22     capitale, pays = ligne.split("-", 1)
23 ValueError: need more than 1 value to unpack
24 >>>
```



Astuce

IDLE vous donne accès à l'entière documentation python via le menu Help -> Python Docs

Comme vous pouvez vous en rendre compte, nous ne sommes pas à la fin de nos ennuis ! Nous avons encore un nouveau problème à résoudre. Normalement, si vous ouvrez votre fichier **capitales.txt** et que vous regardez attentivement chacune des lignes, vous saurez d'où vient cette fameuse erreur.

La première chose à remarquer c'est que l'erreur que nous avons est différente de celle que nous avions précédemment. Avant on nous disait que nous avions trop de valeurs, mais maintenant c'est le fait que nous n'ayons pas assez de valeurs qui pose problème : **ValueError : need more than 1 value to unpack**.

En effet si on analyse de nouveau notre fichier comme je le disais, on peut remarquer que certaines lignes ne sont pas du même format que la grande majorité des autres lignes. C'est le cas par exemple de la ligne sur laquelle se trouve **Monaco** ou encore celle où nous voyons marqué **Panama**. En effet, vu que la capitale de Monaco est Monaco, de même que la capitale du Panama est Panama, j'ai jugé judicieux de ne pas avoir à me répéter. (Pour dire vrai, je cherchais à avoir ce problème afin de vous montrer quelques petites astuces :)).

Donc si on récapitule, notre problème vient du fait que certaines lignes ne contiennent pas de tirets, ce qui pose problème lorsqu'on appelle la méthode **split()**.

Généralement lorsque nous devons traiter des situations exceptionnelles comme celle que nous avons, nous avons deux choix :

- Ajouter une surcouche de logique afin de gérer ladite situation exceptionnelle.
- Laisser l'erreur se produire, mais préparer à l'avance un moyen de la résoudre.

L'objectif sera de choisir la méthode qui marche le mieux en fonction de notre cas de figure. Je sais que cela peut paraître un peu confus, raison pour laquelle je vous invite à voir cela en pratique.

11.4.1 Ajouter encore plus de logique

En plus de la méthode `split()`, toute chaîne de caractères sous Python dispose également d'une méthode `find()`. On peut demander à la méthode `find()` de rechercher une sous-chaîne dans une chaîne de caractères (rechercher par exemple la sous-chaîne "son" dans la chaîne de caractères "maison"). Si la méthode `find()` ne trouve pas la sous-chaîne demandée alors elle retournera la valeur -1. Par contre, si la méthode `find()` trouve la sous-chaîne demandée, elle retournera l'index de la sous-chaîne dans la chaîne.

```

1 >>> ligne = "Monaco"
2 >>> ligne.find("-")
3 -1
4 >>> ligne = "Paris-France"
5 >>> ligne.find("-")
6 5
7 >>>

```

- **P** a pour indice 0
- **a** a pour indice 1
- **r** a pour indice 2
- **i** a pour indice 3
- **s** a pour indice 4
- et notre fameux "-" a été bel et bien trouvé à l'indice 5.

Nous pouvons à présent résoudre très facilement notre problème en procédant comme suit :

```

1 >>> fichier = open("capitales.txt")
2 >>> for ligne in fichier:
3     if( ligne.find("-") == -1 ): #Si on ne trouve pas de tiret sur la ligne
4         capitale = pays = ligne #Alors la ligne correspond à la fois à la ca\
5 pitale et au pays
6     else:
7         capitale, pays = ligne.split("-", 1)
8
9     print(capitale, end=' est la capitale de ')
10    print(pays, end='')
11
12
13 Dakar est la capitale de Sénégal
14 Lagos est la capitale de Nigeria
15 Paris est la capitale de France

```

```

16 Libreville est la capitale de Gabon
17 Ouagadougou est la capitale de Burkina Faso
18 Berlin est la capitale de Allemagne
19 Bruxelles est la capitale de Belgique
20 Doha est la capitale de Qatar
21 Harare est la capitale de Zimbabwe
22 Amsterdam est la capitale de Pays-Bas
23 Lima est la capitale de Pérou
24 Maputo est la capitale de Mozambique
25 Monaco
26     est la capitale de Monaco
27 Monrovia est la capitale de Liberia
28 Panama
29     est la capitale de Panama
30 Rome est la capitale de Italie
31 >>> fichier.close()
32 >>>

```



capitale = pays = ligne équivaut à écrire capitale = ligne puis pays = ligne

Vous remarquez que nous avons un petit retour à la ligne après Monaco et Panama qui déforme un tant soit peu l’affichage de notre texte. Pour résoudre ce problème, on peut utiliser la méthode **strip()** qui permet d’enlever les espaces inconsistants. Comme d’habitude, si vous voulez en savoir plus sur cette méthode, je vous invite à vous servir de la documentation.

On obtient donc le script suivant :

```

1 >>> fichier = open("capitales.txt")
2 >>> for ligne in fichier:
3     if( ligne.find("-") == -1 ):
4         capitale = pays = ligne
5     else:
6         capitale, pays = ligne.split("-", 1)
7
8     # C'est ici que la magie s'opère
9     print(capitale.strip(), end=' est la capitale de ')
10    print(pays, end='')
11
12
13 Dakar est la capitale de Sénégal
14 Lagos est la capitale de Nigeria

```

```
15 Paris est la capitale de France
16 Libreville est la capitale de Gabon
17 Ouagadougou est la capitale de Burkina Faso
18 Berlin est la capitale de Allemagne
19 Bruxelles est la capitale de Belgique
20 Doha est la capitale de Qatar
21 Harare est la capitale de Zimbabwe
22 Amsterdam est la capitale de Pays-Bas
23 Lima est la capitale de Pérou
24 Maputo est la capitale de Mozambique
25 Monaco est la capitale de Monaco
26 Monrovia est la capitale de Liberia
27 Panama est la capitale de Panama
28 Rome est la capitale de Italie
29 >>> fichier.close()
30 >>>
```

Ajouter une surcouche de logique, comme on vient de le confirmer, nous permet de résoudre notre problème. Toutefois, cette méthode présente de nombreux inconvénients. On peut citer entre autres le fait que :

- Si le format du texte contenu dans notre fichier **capitales.txt** change, il faudra automatiquement penser à changer la condition et cela conduira très souvent à un code plus complexe.
- La condition utilisée au niveau de notre instruction **if** sera à mon avis un peu difficile à lire et surtout à comprendre pour un autre développeur (ou même pour nous après plusieurs mois de repos :)).
- Le code est encore fragile, et nous aurons droit à une autre erreur si une nouvelle situation exceptionnelle se présente.

11.4.2 Gérer les exceptions

Tous les programmes informatiques sont conçus pour être utilisés, et de surcroît par des personnes n'ayant certainement aucune connaissance en programmation. Ainsi pour ces derniers, il peut paraître troublant de voir des erreurs s'afficher lors de l'exécution de votre programme. Comment gérer au mieux ce problème ? Une seule réponse : **les exceptions** qui viennent en amont aux structures conditionnelles.

Vous avez remarqué que lorsqu'on avait quelque chose qui ne tournait pas rond au niveau de notre programme, l'interpréteur Python affichait ce qu'on appelle un *traceback* suivi d'un message d'erreur un peu plus explicite.

```

1  Traceback (most recent call last):
2    File "<pyshell#53>", line 2, in <module>
3      capitale, pays = ligne.split("-", 1)
4  ValueError: need more than 1 value to unpack

```

Le **traceback** est un moyen pour Python de nous informer du fait que quelque chose d'inattendu s'était produit durant l'exécution de notre programme. Ces erreurs à l'exécution sont ce qu'on appelle **exceptions** dans le monde Python.

Vous avez en effet deux choix :

- Ignorer les exceptions lorsqu'elles ont lieu, mais soyez sur que votre programme s'arrêtera de manière brutale. On parle de crash du programme dans le jargon informatique (un peu comme un crash d'avion :)).
- Capturer ces exceptions, ce qui vous donnera la chance de gérer par vous-même ces exceptions et ne pas faire ainsi crasher votre programme.

Dans la suite de cette section, je vous montrerai comment capturer les exceptions afin de contrôler le comportement de votre programme à l'exécution. Cela permettra de rendre vos programmes robustes face aux différentes erreurs pouvant survenir à l'exécution.

Les types d'exception

A l'exécution de votre programme, lorsque Python rencontre une erreur dans votre code ou dans une opération que vous lui demandez de faire, il lève une exception. Une exception ressemble à ceci :

```

1  >>> 5 / 'abc'
2  Traceback (most recent call last):
3    File "<pyshell#0>", line 1, in <module>
4      5 / 'abc'
5  TypeError: unsupported operand type(s) for /: 'int' and 'str'
6  >>>

```

Python a ici levé une exception de type **TypeError**, et nous donne des informations supplémentaires pour que nous puissions comprendre d'où vient l'erreur. Il nous dit qu'on ne peut diviser un entier par une chaîne de caractères.

En plus des exceptions de type *TypeError*, il y a également les exceptions de type *ValueError*, *ZeroDivisionError*, *NameError*, *IndexError*, *AssertionError* et bien d'autres...

Forme minimale

```
1 try:
2     instruction
3     ...
4 except:
5     instruction
6     ...
```

On teste le bloc d'instruction du `try`. Si une erreur survient, alors on exécute le bloc d'instructions contenu dans `except`.

```
1 try:
2     test = 3 / 0
3 except:
4     print('Division par zéro illégale!')
```

A l'exécution, nous obtenons la sortie suivante :

```
1 >>>
2 Division par zéro illégale!
3 >>>
```

Forme complète

Dans l'exemple précédent, nous avons simplement dit que si une erreur survenait durant l'exécution du bloc `try` (n'importe laquelle des erreurs), il faudra afficher *Division par zéro illégale!*. Ce que je ne vous ai pas dit, c'est que vous avez la possibilité d'être beaucoup plus explicite par rapport aux types d'exceptions que vous souhaitez capturer. Je sais que cela peut paraître confus, raison pour laquelle je vous propose de voir cela dans un exemple pratique.

```
1 try:
2     test = 3 / 0
3 except ZeroDivisionError:
4     print('Division par zéro illégale!')
```

A l'exécution, nous obtenons la sortie suivante :

```
1 >>>
2 Division par zéro illégale!
3 >>>
```

Récupérer le message d'erreur par défaut

Vous avez la possibilité de capturer une exception et afficher directement le message d'erreur par défaut si vous avez la flemme d'en écrire le votre.

```
1 try:
2     #Bloc d'instructions
3 except TypeDeNotreException as exception_retournee:
4     print("Voici l'erreur : ", exception_retournee)
```

Comme vous pouvez le voir, seul le mot-clé `as` fait la différence.

On peut avoir l'exemple suivant :

```
1 try:
2     calcul = 3 / 0
3 except ZeroDivisionError as exception:
4     print("Voici l'erreur :", exception)
```

A l'exécution, on obtient le résultat suivant :

```
1 >>>
2 Voici l'erreur : division by zero
3 >>>
```

Le mot-clé `else`

Dans un bloc `try`, le mot-clé `else` vous permet d'exécuter une action si aucune n'exception n'a été levée dans le bloc.

```
1 try:
2     resultat = numerateur / denominateur
3 except NameError:
4     print("La variable numérateur ou dénominateur n'a pas été définie.")
5 except TypeError:
6     print("La variable numérateur ou dénominateur possède un type incompatible a\
7 vec la division.")
8 except ZeroDivisionError:
9     print("La variable dénominateur est égale à 0.")
10 else:
11     print("Le résultat obtenu est : ", resultat)
```

En gros, dans ce code, on indique que si il y a une exception, on la capture et on en informe l'utilisateur, dans le cas contraire on affiche gentiment le résultat de la division.

Le mot-clé `finally`

Le mot-clé `finally` vous permet d'exécuter du code après un bloc `try`, quelque soit le résultat de l'exécution dudit bloc.

Vous pouvez être tentés de le demander à quoi cela pourrait bien servir ? Eh bien je vais vous donner un exemple très simple. Supposons que vous avez un programme qui traite certaines valeurs provenant d'un fichier. Il peut arriver que lors de l'exécution de votre programme, une erreur survient et le programme crashe et que le bloc `except` soit exécuté dans le cas où vous avez décidé de gérer les exceptions.

Nous avons toutefois un petit problème. Nous devons en effet nous assurer que le fichier soit toujours fermé après traitement, qu'il y ait eu erreur ou non. Si nous fermons notre fichier à la fin du bloc `try`, cette instruction risque de ne pas être exécutée au cas où une erreur survient, vu que notre bloc `except` sera automatiquement exécuté. Nous ne pouvons pas également mettre l'instruction de fermeture du fichier dans le bloc `except`, dans la mesure où lorsqu'il n'y aura pas d'erreur ce bloc ne sera jamais exécuté et notre fichier ne sera pas ainsi fermé. C'est justement là que le bloc `finally` fait son entrée en fanfare. Qu'il y ait erreur ou non, ce bloc sera toujours exécuté.

```
1  try:
2      # Test d'instruction(s)
3  except Type_d_instruction:
4      # Traitement en cas d'erreur
5  finally:
6      # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

Si nous appliquons cela à notre petit exemple énoncé plus haut, nous obtenons le code suivant :

```
1  try:
2      fichier = open("toto.txt")
3      #quelques traitements
4      ...
5  except:
6      # Traitement en cas d'erreur
7  finally:
8      fichier.close()
```

Le mot-clé `pass`

Le mot-clé `pass` est utilisé lorsque vous n'avez pas pour l'instant d'implémentation concrète pour un bloc donné.


```
1 try:
2     # Test d'instruction(s)
3 except Type_d_instruction:
4     # Rien ne doit se passer en cas d'erreur
5     pass
```

Il peut être utilisé pour n'importe lequel des bloc d'instructions if, while...

Vu qu'une fonction, regroupe également un bloc d'instructions, vous pouvez avoir ce type de déclaration :

```
1 def une_fonction_qui_fera_quelque_chose():
2     pass
```

Vous pourrez ainsi revenir après et implémenter cette fonction. L'utilité du mot-clé *pass* réside dans le fait que grâce à ce dernier, vous n'aurez pas d'erreur, alors qu'une déclaration de fonction de ce genre aurait généré une erreur.

```
1 def une_fonction_qui_fera_quelque_chose():
```

Lever une exception

Il peut être parfois indispensable de lever soi-même des exceptions. Pour ce faire, il suffit de taper le mot-clé **raise** suivi du type d'exception à lever et entre parenthèses le message à afficher.

```
1 raise TypeDeLException("message à afficher")
```

Supposons que vous souhaitiez créer une fonction **que_des_valeurs_paires** qui sera censée recevoir que des valeurs paires. Au cas où, une valeur impaire est passée en argument, une exception de type `ValueError` sera levée. Comme vous allez le voir ce sera très simple en procédant comme suit :

```
1 def que_des_valeurs_paires(valeur):
2     if( valeur % 2 == 0):
3         print("Good Job.")
4     else:
5         raise ValueError("Seules les valeurs paires sont autorisees. Merci.")
```

A l'exécution, vous aurez ceci :

```

1  >>> que_des_valeurs_paires(2)
2  Good Job.
3  >>> que_des_valeurs_paires(14)
4  Good Job.
5  >>> que_des_valeurs_paires(1)
6  Traceback (most recent call last):
7    File "<pyshell#3>", line 1, in <module>
8      que_des_valeurs_paires(1)
9    File "C : \Users\honore.h\Desktop\game.py", line 5, in que_des_valeurs_paires
10      raise ValueError("Seules les valeurs paires sont autorisees. Merci.")
11  ValueError: Seules les valeurs paires sont autorisees. Merci.
12  >>>

```

Ainsi, avec la gestion des exceptions, nous pourrons écrire un bien meilleur programme comme celui-ci :

```

1  try:
2      que_des_valeurs_paires(1)
3  except ValueError as exception:
4      print(exception)

1  >>>
2  Seules les valeurs paires sont autorisees. Merci.
3  >>>

```

Cool n'est-ce pas ?

11.4.3 Revenons à nos moutons

Maintenant que vous êtes à l'aise avec la gestion des exceptions, nous pouvons utiliser tout ce que nous avons vu jusque là pour résoudre notre problème précédent avec le contenu du fichier `capitales.txt`.

Pour rappel, le problème que nous avions était le suivant :

```
1 >>> fichier = open("capitales.txt")
2 >>> for ligne in fichier:
3     capitale, pays = ligne.split("-", 1)
4     print(capitale, end=' est la capitale de ')
5     print(pays, end='')
6
7
8 Dakar est la capitale de Sénégal
9 Lagos est la capitale de Nigeria
10 Paris est la capitale de France
11 Libreville est la capitale de Gabon
12 Ouagadougou est la capitale de Burkina Faso
13 Berlin est la capitale de Allemagne
14 Bruxelles est la capitale de Belgique
15 Doha est la capitale de Qatar
16 Harare est la capitale de Zimbabwe
17 Amsterdam est la capitale de Pays-Bas
18 Lima est la capitale de Pérou
19 Maputo est la capitale de Mozambique
20 Traceback (most recent call last):
21   File "<pyshell#53>", line 2, in <module>
22     capitale, pays = ligne.split("-", 1)
23 ValueError: need more than 1 value to unpack
24 >>>
```

Nous l'avons résolu précédemment en utilisant une combinaison if - else. Ce que je vous propose, c'est de résoudre le même problème en utilisant cette fois-ci les exceptions. Saurez-vous le faire ? Je dirais OUI à 100%. Alors au travail !



Indice

Comme nous pouvons le voir dans le traceback que nous fournit l'interpréteur Python, l'exception levée est de type `ValueError`.

Long moment de réflexion...

J'espère que vous avez trouvé une solution à notre problème. J'avoue qu'il n'y avait rien de bien compliqué. Je vous propose rapidement ma solution.

```
1 fichier = open("capitales.txt")
2 for ligne in fichier:
3     try:
4         capitale, pays = ligne.split("-", 1)
5     except ValueError:
6         capitale = pays = ligne
7     finally:
8         print(capitale.strip(), end=' est la capitale de ')
9         print(pays, end='')
10
11 fichier.close()
```

Et comme résultat, au niveau de la console d'interprétation on a :

```
1 >>>
2 Dakar est la capitale de Sénégal
3 Lagos est la capitale de Nigeria
4 Paris est la capitale de France
5 Libreville est la capitale de Gabon
6 Ouagadougou est la capitale de Burkina Faso
7 Berlin est la capitale de Allemagne
8 Bruxelles est la capitale de Belgique
9 Doha est la capitale de Qatar
10 Harare est la capitale de Zimbabwe
11 Amsterdam est la capitale de Pays-Bas
12 Lima est la capitale de Pérou
13 Maputo est la capitale de Mozambique
14 Monaco est la capitale de Monaco
15 Monrovia est la capitale de Liberia
16 Panama est la capitale de Panama
17 Rome est la capitale de Italie
18 >>>
```

Je n'ai pas mis l'instruction **fichier.close()** au niveau du bloc **finally** parce que tout simplement cela aurait fermé notre fichier bien que nos traitements sur ce dernier ne soient pas terminés. et nous aurons eu droit à une belle erreur de ce type :

```

1  Dakar est la capitale de Sénégal
2  Traceback (most recent call last):
3    File "C : \Users\honore.h\Desktop\game.py", line 2, in <module>
4      for ligne in fichier:
5  ValueError: I/O operation on closed file.
6  >>>

```

11.4.4 Une dernière chose à gérer

Notre programme est quasi parfait, mais nous n'avons toutefois pas gérer une dernière éventualité. Vous devinez laquelle ? Qu'est-ce qui se passera si le fichier `capitales.txt` n'existe pas ? Nous aurons donc tenté d'ouvrir un fichier fantôme et bang nous aurons droit à une erreur. Heureusement au niveau du module `os` nous avons une méthode qui nous permet de vérifier l'existence d'un fichier.

```

1  >>> import os
2  >>> os.path.exists("capitales.txt")
3  True
4  >>>

```

Ainsi nous pouvons soit utiliser un bloc `if - else` pour gérer ce dernier cas où utiliser tout simplement les exceptions que nous aimons tant. Je vous propose les deux méthodes pour ne pas faire de jaloux. Après, vous pourrez par vous-même voir la méthode que vous sied le mieux.

```

1  #Methode 1 : if - else
2
3  import os
4
5  if os.path.exists('capitales.txt'):
6      fichier = open("capitales.txt")
7
8      for ligne in fichier:
9          if( ligne.find("-") == -1 ):
10             capitale = pays = ligne
11         else:
12             capitale, pays = ligne.split("-", 1)
13
14         print(capitale.strip(), end=' est la capitale de ')
15         print(pays, end='')
16
17     fichier.close()
18 else:
19     print("Le fichier de données est manquant.")

```

```

1  #Methode 2 : Exceptions
2  #Comme vous le voyez, nous n'avons plus besoin d'importer le module os
3
4  try:
5      fichier = open("capitales.txt")
6
7      for ligne in fichier:
8          try:
9              capitale, pays = ligne.split("-", 1)
10             except ValueError:
11                 capitale = pays = ligne
12
13             print(capitale.strip(), end=' est la capitale de ')
14             print(pays, end='')
15
16         fichier.close()
17 except FileNotFoundError:
18     print("Le fichier de données est manquant.")

```

Pour savoir qu'il fallait capturer une exception de type `FileNotFoundError`, j'ai tout simplement fait crasher volontairement notre programme en donnant à la fonction `open()` le nom d'un fichier qui n'existe pas et comme vous pouvez le voir l'interpréteur Python nous a gentiment retourné ce que nous cherchons.

```

1  >>> open("fichier_de_ouf.txt")
2  Traceback (most recent call last):
3      File "<pyshell#0>", line 1, in <module>
4          open("fichier_de_ouf.txt")
5  FileNotFoundError: [Errno 2] No such file or directory: 'fichier_de_ouf.txt'
6  >>>

```

11.5 Jeu de capites - Version 2

Notre jeu de capitales du chapitre précédent qui stockait les données directement au niveau d'un dictionnaire avait un tout petit problème. Un utilisateur lambda qui ne maîtrise pas la programmation n'aurait pas pu par lui-même rajouter de nouveaux pays et leurs capitales respectives au dictionnaire de peur de bousiller le programme.

Pour ce faire, notre utilisateur vous a donc fait appel et vous a fourni un fichier `capitales_game.txt` contenant une liste de pays et leurs capitales respectives.

Le fichier suit le format suivant Pays :Capitale. A noter que même pour les pays qui ont le même nom que leur capitale (comme Monaco et Panama), ce même format sera toujours utilisé.

Je vous présente un aperçu du contenu de notre fichier afin que tout soit clair dans votre esprit :

```
1  Sénégal:Dakar
2  Nigeria:Lagos
3  France:Paris
4  Gabon:Libreville
5  Burkina Faso:Ouagadougou
6  Allemagne:Berlin
7  Belgique:Bruxelles
8  Qatar:Doha
9  Zimbabwe:Harare
10 Pays-Bas:Amsterdam
11 Pérou:Lima
12 Mozambique:Maputo
13 Monaco:Monaco
14 Liberia:Monrovia
15 Panama:Panama
16 Italie:Rome
```

Vous avez si je m'en abuse, toutes les cartes en main pour résoudre ce problème et rendre heureux votre très cher ami. Ma solution vous sera proposée comme d'habitude tout juste en bas, mais je vous invite à chercher par vous-même et avoir votre propre solution. Bonne chance !

11.5.1 Exemple de solution

Ce code, j'ose l'espérer n'aura pas besoin d'explication supplémentaire :).

```
1  # Jeu de capitales v2
2  # Python par la pratique
3
4  from random import sample
5
6  try:
7      fichier = open("capitales_game.txt")
8
9      #Contiendra l'ensemble des pays et leurs capitales respectives
10     capitales = {}
11
12     for ligne in fichier:
13         #On récupère chaque pays et sa capitale associée
14         parts = ligne.split(":")
```

```

15     pays = parts[0].strip()
16     capitale = parts[1].strip()
17
18     # On ajoute progressivement les données à notre dictionnaire
19     capitales[pays] = capitale
20
21     fichier.close() #Fermeture du fichier
22
23     #Nombre total de questions par série
24     NBRE_TOTAL_DE_QUESTIONS = 6
25
26     #Contiendra le score de l'utilisateur
27     score = 0
28
29     #On récupère une liste aléatoire de [NBRE_TOTAL_DE_QUESTIONS] pays
30     liste_pays = sample(list(capitales), NBRE_TOTAL_DE_QUESTIONS)
31
32     for pays in liste_pays:
33         print("Quelle est la capitale de ce pays : " + pays + " ?")
34         reponse = input()
35
36         if(reponse.lower() == capitales[pays].lower()):
37             print("Bonne réponse!")
38             score += 1 #On incrémente son score
39         else:
40             print("Mauvaise réponse! Il fallait répondre : " + capitales[pays])
41
42     print("C'est terminé ! << Score : ", score, "/", NBRE_TOTAL_DE_QUESTIONS, " >\
43 >")
44 except FileNotFoundError:
45     print("Le fichier de données est manquant.")

```

11.6 Petit exercice

Modifier notre programme afin de pouvoir donner la possibilité à l'utilisateur de pouvoir rejouer. Ainsi, si l'utilisateur termine une partie, on lui demande s'il souhaite rejouer ou non. S'il répond par l'affirmative alors on lui prépare une nouvelle partie, dans le cas contraire on lui dira gentiment **Bye Bye**.

Vous pouvez donc utiliser ce petit test avec encore une fois notre fameuse méthode `lower`.


```
1 reponse = input("Voulez-vous rejouer (O/N) ?")
2
3 if(reponse.lower() == 'o'):
4     #L'utilisateur veut rejouer
5 else:
6     #On lui dit Bye Bye
```

En utilisant la méthode `lower`, cela nous évite d'avoir à faire deux tests.

```
1 reponse = input("Voulez-vous rejouer (O/N) ?")
2
3 if(reponse == 'O' or reponse == 'o'):
4     #L'utilisateur veut rejouer
5 else:
6     #On lui dit Bye Bye
```

Rendez ensuite votre code modulaire en le découpant en de petites fonctions.

11.7 Résumé

Etant donné qu'il s'agit du dernier chapitre, essayez par vous-même de le parcourir afin d'en faire un résumé.

Je vous fais confiance :).

Conclusion

Comme j'ai voulu le démontrer tout au long de cet ouvrage, Python est non seulement un langage assez simple à apprendre mais il regorge également d'une pléthore de fonctionnalités. Malheureusement, ce que nous avons vu dans ce livre ne représente que le seizième de ce dont Python est capable de faire.

Mais soyez-en rassurés, ce n'est pas la fin ! En fait, nous ne faisons que commencer. Je n'ai pas écrit *Python par la pratique* pour le publier et ensuite le jeter aux oubliettes, loin de là. Ce livre sera amené à être continuellement mis à jour dans le courant de l'année 2016, pour refléter non seulement les meilleures pratiques de développement en Python mais également aborder d'autres notions qui n'ont pas eu la chance d'être étudiées dans cette première édition (on peut citer entre autres les interfaces graphiques, le développement web en Python, l'interaction avec les bases de données, la programmation orientée objet en Python, la gestion des réseaux avec Python...).

D'ici là portez vous bien, et n'hésitez surtout pas à faire un tour sur la chaîne Youtube des [TEACHERS DU NET](#)¹⁸, pour apprendre encore plus sur le monde de la programmation.

Faire un [don](#)¹⁹.

18. <http://youtube.com/hounwanou1993>

19. https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=LGFM78CBR2J2Q