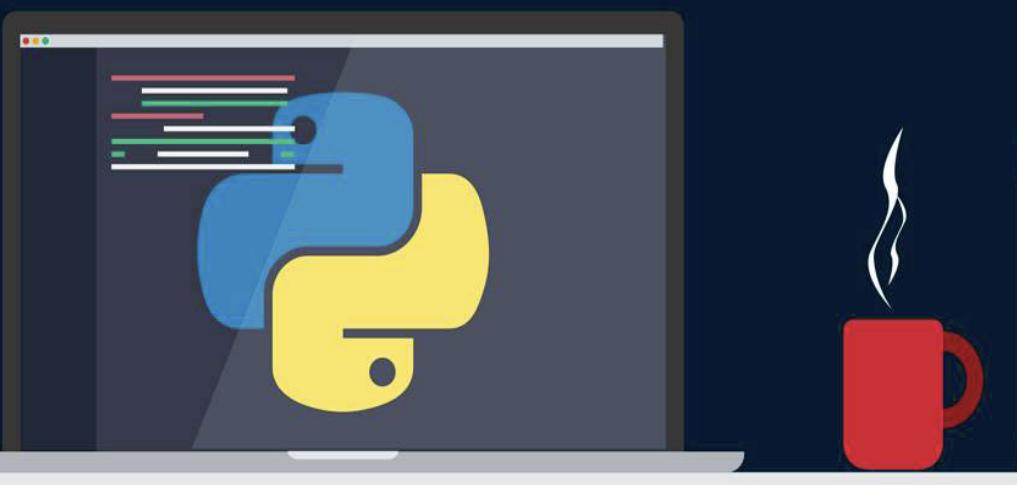
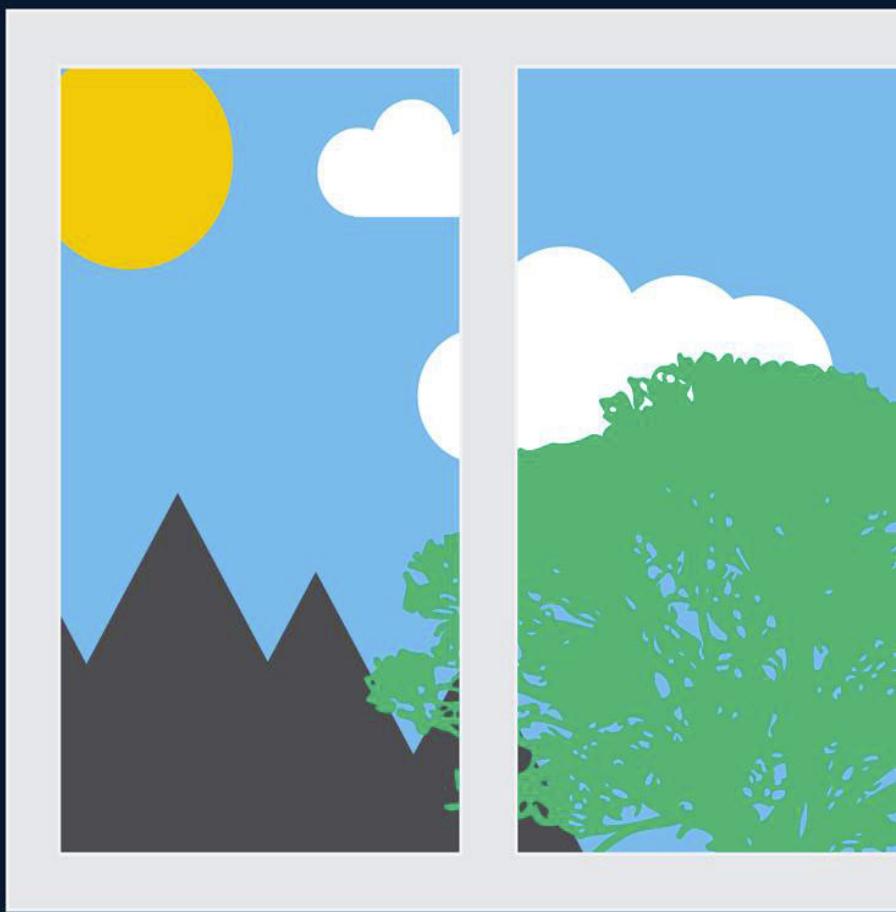


python

Par la pratique

Seconde édition



Par Honoré Hounwanou

Table des matières

Droits d'auteur	5
Dédicaces	6
À propos de l'auteur	7
Introduction	8
1.1. Envie de me soutenir?	8
1.2. Formation Vidéo	9
1.3. Pré-requis	9
1.4. Structure du livre	9
1.5. Conventions	10
1.6. Corrections et suggestions	11
1.7. Recommandations	11
1.8. Résumé	12
Jargon Informatique	13
2.1. Qu'est-ce que la programmation?	13
2.2. Qu'est-ce qu'un langage de programmation?	15
2.3. Quelques définitions	18
2.4. Le langage Python	19
2.5. Résumé	21
Installation de Python	22
3.1. Installation de Python sous Windows	22
3.2. Installation de Python sous Linux	25
3.3. Installation de Python sous macOS	26
3.4. La console d'interprétation de Python	28
3.5. Résumé	30
Notre premier jeu	31
4.1. Le principe du jeu	31
4.2. Les chaînes de caractères	34
4.3. Les variables	35

4.4. Les commentaires	38
4.5. Passons au code source !	39
4.6. Résumé	44
Les fonctions	46
5.1. Objectifs de la modularité	46
5.2. Les fonctions	46
5.3. Résumé	65
Conditions et Boucles	66
6.1. Les conditions	66
6.2. Les itérations ou boucles	74
6.3. Résumé	82
Types de données complexes (Partie 1/2)	83
7.1. Les tuples	83
7.2. Les listes	85
7.3. Les dictionnaires	87
7.4. Résumé	89
Types de données complexes (Partie 2/2)	90
8.1. Les types non scalaires	90
8.2. Les chaînes de caractères	90
8.3. Les tuples	97
8.4. Les listes	99
8.5. Les méthodes split et join	102
8.6. Résumé	103
Modules et Packages	104
9.1. Les modules	104
9.2. Les packages	114
9.3. Résumé	115
Jeu de capitales	116
10.1. Le principe du jeu	116
10.2. Le module random	117
10.3. Challenge	125
10.4. Exemple de Solution	125
10.5. Faites-vous confiance !	127

10.6. Petit exercice	129
10.7. Résumé	129
Fichiers et Exceptions	131
11.1. Les données sont très souvent externes	131
11.2. La fonction open	132
11.3. Contenu du fichier capitales.txt	135
11.4. Un code plus sûr	138
11.5. Jeu de capitales - Version 2	155
11.6. Petit exercice	157
11.7. Résumé	158
Conclusion	159
Révisions	160
Révision 2 (18/10/2024)	160
Révision 1 (15/02/2016)	160

Droits d'auteur

Python par la pratique couvrant Python 3.13.

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

SECONDE ÉDITION · © 2024 ParlonsCode.com.

Publié par Honoré Hounwanou, le 18 octobre 2024.

Ce livre est gratuit et il le restera. Les codes fournis dans ce livre peuvent être utilisés librement dans vos projets commerciaux ou autres.

Si tu veux me soutenir, tu peux m'acheter un petit café ici :).

Dédicaces

- Je tiens à remercier Guido van Rossum et toutes les personnes qui ont fait de Python une réalité. C'est vraiment un langage de programmation incroyable.
- À ma famille qui a toujours cru en moi. Papa, Maman, MERCI du fond du cœur. Il n'y a pas un seul jour qui passe, sans que je ne me sente chanceux d'avoir eu des parents prêts à investir autant sur mon éducation. Chaque ligne de ce livre vous est dédiée.
- À Moustapha Loum pour cette magnifique couverture. You rock!
- À ma conjointe pour son soutien inestimable et son amour inconditionnel ❤.
- À toute l'équipe des TEACHERS DU NET et à nos précieux(ses) abonné(e)s.

À propos de l'auteur



Honoré Hounwanou

Honoré Hounwanou est un développeur, formateur et entrepreneur béninois résidant au Canada où il occupe actuellement le poste d'ingénieur logiciel à Valence Labs, un centre de recherche qui vise à repousser les limites de l'IA pour trouver des traitements pour des maladies actuellement incurables.

Il est cofondateur des startups PayDunya (Solution de paiement en ligne) et SamaPass (Solution de billetterie en ligne).

Il a fondé il y a de cela 12 ans, la plateforme d'E-learning LES TEACHERS DU NET offrant des formations et tutoriels de catégories diverses expliqués de manière simple et chirurgicale. Parlons Code est une extension des TEACHERS DU NET offrant des formations payantes sur différents langages, technologies et frameworks.

Il parle couramment le Python, PHP, JavaScript, Java, Python, C++, SQL.

Vous pouvez le trouver en ligne sur <https://bento.me/honore-h>.

CHAPITRE 1

Introduction

Dans ce livre, nous apprendrons par la pratique à programmer en Python. Il s'agit aujourd'hui de l'un des langages de programmation les plus populaires, mais également celui ayant la croissance la plus rapide. Python est aussi reconnu pour être un langage idéal pour des personnes souhaitant découvrir le monde merveilleux de la programmation.

Très souvent, lorsque nous nous lançons dans l'apprentissage d'un nouveau langage de programmation, nous commençons par une phase théorique dans laquelle l'on essaie 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 pourrez l'utiliser pour créer 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 recommande de visiter la page [Histoires à succès 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 s'intéresser à des notions beaucoup plus avancées comme celles des structures de données (les listes, les tuples et les dictionnaires), des modules et packages, des fichiers et exceptions, du concept de programmation orientée objet en Python et bien plus.

Sacré programme n'est-ce pas? Alors prêt(e)? C'est parti!

1.1. Envie de me soutenir?

J'ai mis des années 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 petit don symbolique. Je pourrai utiliser cet argent pour m'acheter un petit café tout en ayant une pensée pour vous. Merci :).

1.2. Formation Vidéo

Certaines personnes préfèrent regarder des vidéos en lieu et place de lire. Si vous faites partie de ces personnes, vous serez heureux de savoir que tout le contenu dont regorge cet ouvrage (et plein d'autres contenus additonnels) est également disponible au format vidéo.

Chacune des vidéos a été montée afin de vous permettre de mieux assimiler les différentes notions abordées dans ladite vidéo.

La formation est disponible en accès prématué à 89\$ via ce lien:
<https://online.parlonscode.com/courses/masterclass-python>.

1.3. Pré-requis

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

1.4. Structure du livre

Après ce chapitre d'introduction, *Python par la pratique* est structuré en dix (10) autres chapitres.

Bien que vous puissiez choisir les chapitres à lire dans un ordre quelconque, sachez qu'il y a eu beaucoup de réflexion derrière la structure finale du livre. Chaque chapitre introduit un nouveau concept et renforce les enseignements passés. Ainsi, même si vous avez hâte de sauter des chapitres, je vous recommande fortement de lire les différents chapitres du livre dans l'ordre indiqué. Les chapitres suivants ne couvriront pas le matériel précédent dans la même profondeur que les chapitres précédents.

À la fin de ce livre, vous aurez une solide compréhension des concepts fondamentaux du langage de programmation Python, la capacité de créer des applications par vous-même et du contexte nécessaire pour tirer pleinement parti des ressources supplémentaires pour l'apprentissage des techniques Python intermédiaires et avancées.

1.5. Conventions

Il existe de nombreux exemples de code dans ce livre et ces derniers seront présentés comme suit:

`hello_world.py`

```
print("Hello, World!")
```

Chaque bloc de code aura en libellé le nom de fichier auquel il appartient (`hello_world.py` dans l'exemple ci-dessus). Ce nom de fichier sera cliquable et mènera vers un [gist Github](#) afin de vous permettre de très facilement copier son contenu. Essayez par exemple de cliquer sur le libellé `hello_world.py` du bloc de code ci-dessus et vous verrez par vous-même.

Par souci de concision, nous utiliserons les points de suspension “...” pour désigner un code existant qui reste inchangé. Les gists Github quant à eux contiendront toujours le code complet du fichier.

`jeu_des_capitales.py`

```
...
for i in range(0, NOMBRE_TOTAL_DE_QUESTIONS):
    pays = random.choice(liste_de_pays)

    while pays in pays_deja_choisis:
        pays = choice(liste_de_pays)

    pays_deja_choisis.append(pays)
...
```

Dans l'exemple ci-dessus, les points de suspension “...” ont été utilisés afin d'indiquer les parties du code qui restent inchangées. Si vous cliquez toutefois sur le libellé du bloc `jeu_des_capitales.py`, vous verrez que le gist Github associé contient le code complet du fichier.

Pour terminer, nous utiliserons fréquemment le shell interactif de Python pour exécuter des

bouts de code Python, à des fins d'expérimentation. Chaque session d'interaction avec le shell interactif de Python sera présentée comme suit:

```
Shell interactif de Python

>>> import random
>>> random.randint(2, 20)
13
>>>
```

1.6. Corrections et suggestions

1.6.1. Errata

Bien que j'ai pris toutes les précautions nécessaires pour garantir l'exactitude du contenu de ce livre, des erreurs peuvent s'y être glissées. Si vous trouvez des fautes d'orthographe ou de grammaire, des erreurs de programmation, je vous serai reconnaissant de bien vouloir me les signaler par mail à hello@parlonscode.com.

1.6.2. Piratage

Si vous trouvez sur Internet des copies illégales de ce livre sous quelque forme que ce soit, je vous serai reconnaissant de me fournir l'adresse de l'emplacement ou le nom du site Web à hello@parlonscode.com.

1.6.3. Questions générales

Si vous avez des questions générales sur un chapitre de ce livre, n'hésitez surtout pas à m'envoyer un courriel à hello@parlonscode.com.

Vous pouvez également me laisser un message sur Twitter à [@etsmo](#).

1.7. Recommandations

Juste quelques éléments importants avant de commencer:

1. Tout le monde apprend différemment. Dans ce livre, je vais essayer d'expliquer les choses le plus simplement possible. Assurez-vous de lire les sections de texte, les notes, les astuces, faire les challenges et surtout suivre à la lettre le programme du livre. L'apprentissage passif ne vous aidera en rien, alors soyez engagé(e), pratiquez!
2. Comme mentionné dans #1, peut-être la partie la plus importante de l'ensemble du livre est de travailler en suivant le programme du livre. Ce n'est pas un hasard, si les challenges vont être les parties les plus difficiles. Cela vous permettra d'apprendre par la pratique, et croyez-moi, vous serez fier(e) de vous.
3. Tout au long des différents chapitres, nous allons construire plusieurs projets complets. Il sera probablement avantageux de programmer avec moi pour être pleinement engagé(e). Si vous vous perdez ou que votre code ne fonctionne pas, vous pouvez télécharger le code source associé au chapitre au niveau du dépôt GitHub du livre.
4. Si vous vous trouvez toujours coincé ou avez des questions, vous pouvez me trouver sur Twitter à [@etsmo](#) ou par courriel à hello@parlonscode.com ou directement via le formulaire de contact de Parlons Code.

Bonne chance! Mamma Mia!

1.8. Résumé

Dans ce premier chapitre, nous avons appris que:

- Python est actuellement l'un des langages de programmation les plus populaires, mais également celui ayant la croissance la plus rapide.
- Python est un langage de programmation facile d'apprentissage et idéal pour des personnes souhaitant découvrir le monde merveilleux de la programmation.
- Python est un langage polyvalent. On peut l'utiliser pour concevoir des applications variées: sites web, jeux vidéos, applications mobiles, etc.

CHAPITRE 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 libre de sauter ce chapitre et passer directement à la section consacrée à [l'installation de Python](#).

2.1. Qu'est-ce que la programmation?

Comme vous le savez probablement, il existe une pléthore de langages de programmation. Nous pouvons citer entre autres le C, C++, Java, Perl, PHP, Python et Ruby. Jugez-en par vous-même, [la liste est longue](#). Mon choix s'est 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 boîtes 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 $y * y = x$.

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

Étant donné que je sais très bien que ça peut arriver de temps à autre d'avoir de petites fuites de mémoire, nous allons essayer de détailler un peu tout ceci, histoire de nous assurer que nous sommes tous sur la même longueur d'onde.

Alors voilà:

- 4 est une racine carrée de 16 car $4 * 4 = 16$.
- 3 est une racine carrée de 9 car $3 * 3 = 9$, 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.

À priori, en suivant cette petite recette, nous pourrons déterminer 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 toujours avoir ces deux types de connaissances, du moins en fonction de ce qu'il souhaite réaliser comme programme.

Je m'explique. Si vous souhaitez 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 vous faudra avoir premièrement la connaissance déclarative associée à ce problème. C'est-à-dire savoir ce qu'on appelle intensité de la force électrostatique entre deux charges électriques.

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.

– Source: [Wikipédia](#)

Et deuxièmement, il vous faudra avoir également la connaissance impérative associée au problème. C'est-à-dire connaître la petite recette permettant de 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 de ce livre. Comprenez plutôt la logique qui est derrière cet exemple. Nous avons eu besoin des connaissances déclarative et impérative associées à notre problème précédent afin de le résoudre. C'est tout ce qu'il y a à comprendre!

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

Rappelons qu'un programme n'est rien d'autre qu'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 simple 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 dont la solution optimale est toujours recherchée.

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 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 claire (*sémantique*).

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, Kotlin, etc.

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 (*binary digits: 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, etc), nous pourrons écrire nos programmes (on dit aussi coder) plus aisément. Ce code sera ensuite traduit en langage machine (c'est-à-dire en une série de 0 et de 1) 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

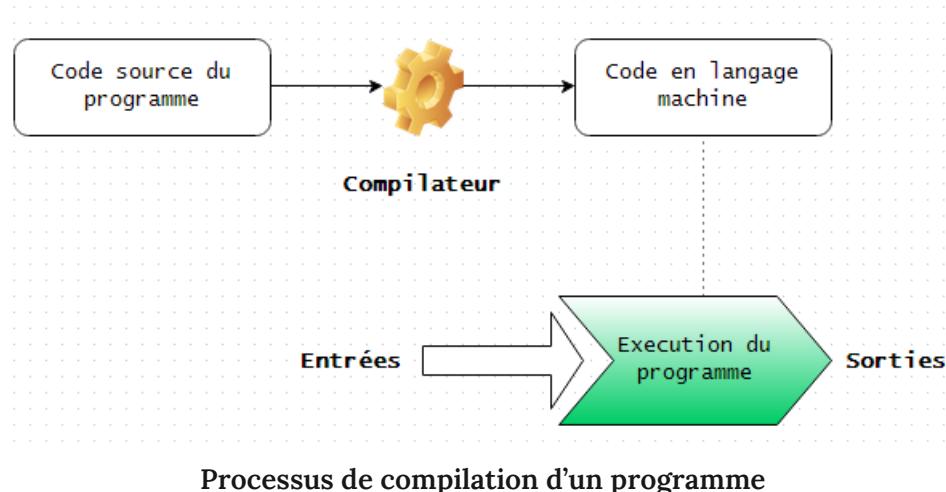
Il existe de manière générale deux moyens principaux pour traduire un code écrit dans un langage de programmation en langage machine. Un langage de programmation peut être soit compilé ou soit interprété.

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

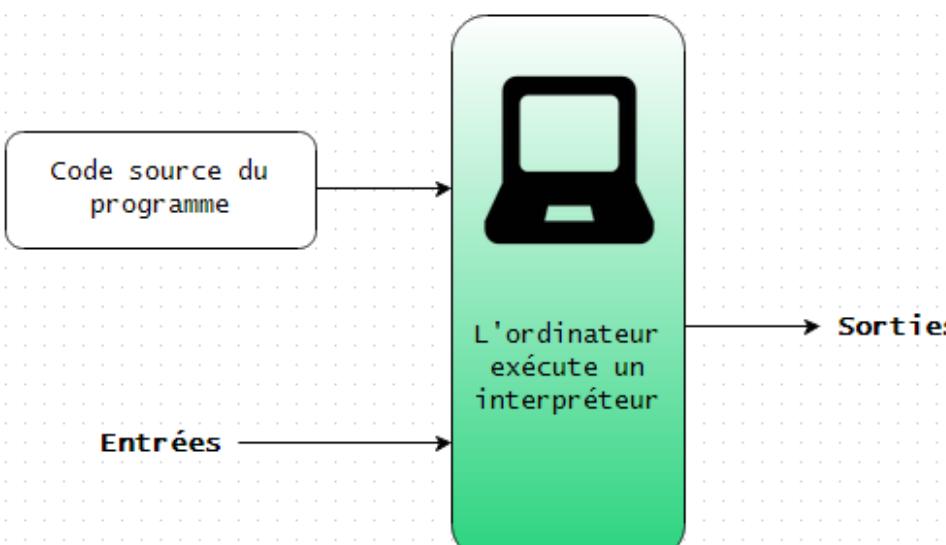
obtenu après traduction en code machine est un programme que l'ordinateur peut directement exécuter.



Interpréteur

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

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. Une recompilation sera nécessaire uniquement dans le cas où le code source du programme vient à changer. 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 développés 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 et/ou un interpréteur approprié.

2.3. Quelques définitions

2.3.1. 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. C'est en d'autres termes le nom que les informaticiens ont donné à la connaissance impérative.
- Un programme quant à lui est la traduction de votre algorithme dans un langage de programmation. C'est en effet un ensemble d'instructions exécutées par l'ordinateur.
- Une instruction est une tâche que doit exécuter l'ordinateur.

2.3.2. Syntaxe, sémantique statique, sémantique

Syntaxe

La syntaxe permet de définir les séquences de caractères et de symboles constitutants 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 n'est 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*. Soyez sans crainte, nous en reparlerons.
- `3 + "abc"` (Bonne syntaxe mais mauvaise sémantique statique) cela n'a aucun sens! On ne peut additionner des bananes et des pommes.

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 donc le temps de les assimiler. Ils nous seront plus tard utiles lorsque nous aurons des messages d'erreur comme:

`SyntaxError: invalid syntax.`

2.4. Le langage Python

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

2.4.1. 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, créateur du langage Python

Guido a décidé de baptiser son projet Python en référence à la série télévisée des Monty Python dont il en est un grand fan.

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

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

2.4.2. 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 aujourd'hui l'un des langages de programmation les plus populaires, mais également celui ayant la croissance la plus rapide.
- C'est un langage de programmation généraliste. Ce qui signifie que vous pouvez utiliser le langage Python pour concevoir plusieurs types de programmes: des scripts d'automatisation de tâches, des jeux vidéos, des sites web, des applications de bureau, etc.
- Python est un langage facile d'apprentissage.
- Python a une élégante syntaxe, ce qui facilitera grandement la lisibilité de tous vos programmes Python.
- Python est un langage concis et expressif, ce qui signifie que vous pouvez écrire des

programmes complexes avec une quantité de code relativement minimale.

- Python est interactif. Le shell interactif Python que nous utiliserons tout au long de ce livre nous permet d'exécuter différents bouts de code et avoir des réponses de manière instantanée. Nous verrons ceci en pratique, ne vous inquiétez donc pas si cela à l'air encore flou.
- Python est un langage supportant aussi bien le paradigme de la programmation orientée objet que celui de la programmation fonctionnelle.
- Il existe une communauté open source gigantesque autour de Python, développant des librairies super intéressantes et très puissantes dont nous pourrons nous servir afin d'éviter de réinventer la route.
- Pour terminer, le marché du travail pour Python est extrêmement dense en ce moment, en particulier pour toute personne ayant des compétences en science des données (data science) et en intelligence artificielle.

2.5. Résumé

Dans ce second chapitre, nous avons appris que:

- Un bon programmeur doit s'assurer de toujours avoir les connaissances déclarative et impérative associées au problème qu'il souhaite résoudre.
- 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, etc.
- De manière générale, il existe de 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 possède une élégante syntaxe et encourage grandement la lisibilité de votre code source.
- Le marché du travail pour Python est extrêmement dense en ce moment, en particulier pour toute personne ayant des compétences en science des données (data science) et en intelligence artificielle.

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 macOS.

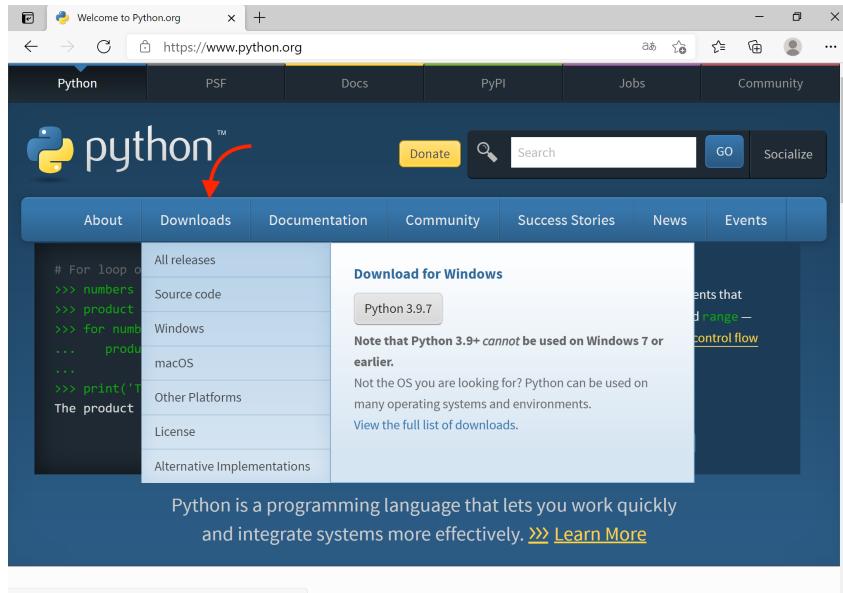
Quelque soit votre système d'exploitation, la première étape consiste à vous rendre sur le site officiel de Python: <https://python.org>.

Ne pas utiliser Python 2!

Python 2 est arrivé en fin de vie et ne devrait plus être utilisé pour de nouveaux projets Python. Pour les projets Python que nous aurons à concevoir dans ce livre, nous utiliserons Python 3, qui est aujourd'hui la version de Python recommandée si vous souhaitez démarrer de nouveaux projets Python.

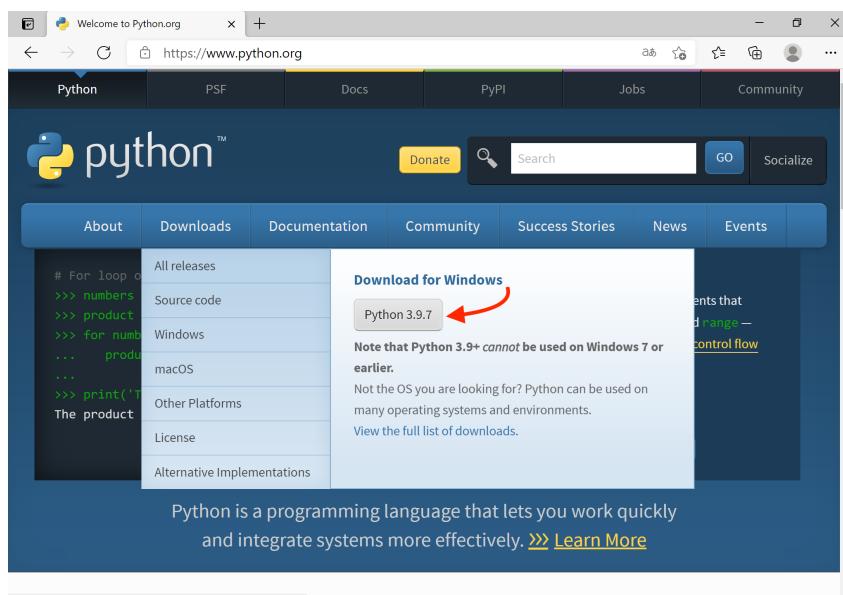
3.1. Installation de Python sous Windows

1. Survolez le lien *Downloads* présent au niveau du menu principal de [la page d'accueil du site officiel de Python](https://python.org).



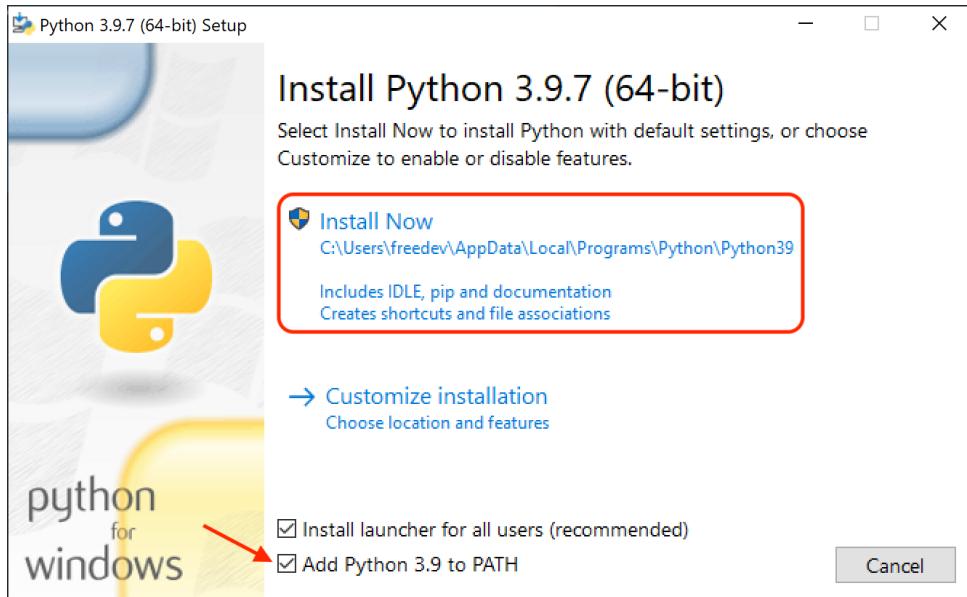
Site officiel de Python

2. Vous devriez normalement voir un bouton **Python 3.X.Y** pour télécharger la dernière version en date de Python. Au moment où j'écris ces lignes, il s'agit de la version **3.13.0**.



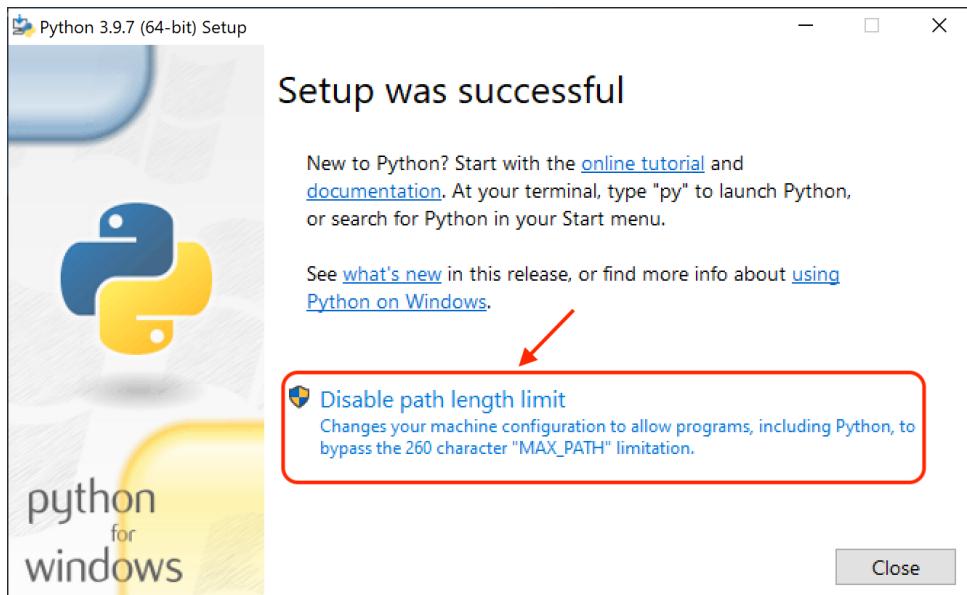
Bouton de téléchargement de Python depuis le site officiel

3. Cliquez à présent sur le bouton de téléchargement. Sans grande surprise, le téléchargement du fichier d'installation de Python débutera.
4. Une fois le téléchargement terminé, exécutez ensuite le fichier d'installation et suivez les différentes étapes. Il vous faut cocher la case **Add Python 3.x to PATH** lors de l'installation afin que Python soit accessible en ligne de commande. Cela s'avéra utile plus tard si vous souhaitez utiliser Python en ligne de commande (par exemple, si vous souhaitez créer des sites web en un temps record avec le framework web Django).



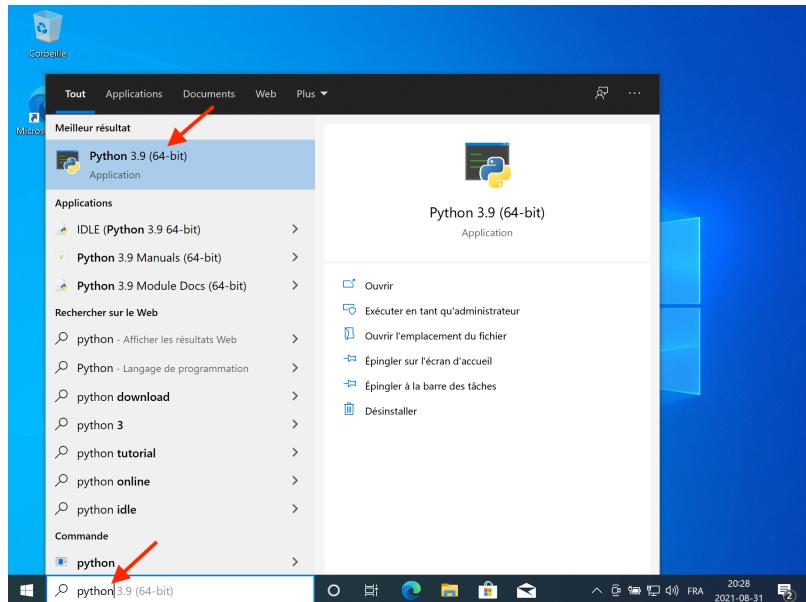
Ajout de l'exécutable de Python au PATH

- Il est également recommandé de désactiver la longueur limite de chemin une fois l'installation de Python réussie, car si Python est installé dans un répertoire avec une longueur de chemin supérieure à 260 caractères, son ajout au chemin pourrait échouer. Si vous n'avez rien compris de la phrase précédente, ce n'est pas bien grave. Contentez-vous simplement de cliquer sur le bouton *Disable path length limit* afin de désactiver la longueur limite de chemin. Cela vous fera potentiellement gagner du temps à l'avenir sur le débogage d'un problème évitable.



Désactivation de la longueur limite de chemin

- À cette étape, l'installation de Python est normalement terminée. Histoire d'avoir confirmation, recherchez `python` au niveau de la barre de recherche présente au niveau de la barre des tâches. Vous devriez normalement voir Python 3.13 affiché.



Confirmation d'installation de Python sous Windows

3.2. Installation de 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.

Comment afficher la version de Python installée?

Afin de connaître la version de Python installée, tapez dans un terminal la commande `python3 --version` ou `python3 -V` (avec un V majuscule).

Il est très probable que ce soit une version comme **3.8** ou **3.9**. Pour ce livre, nous utiliserons la dernière version en date de la branche **3.x** afin de profiter de toutes les nouvelles fonctionnalités intéressantes rajoutées au langage Python. Au moment où j'écris ces lignes, il s'agit de la version **3.13.0**.

Si vous avez une distribution Debian, exécutez les commandes suivantes:

Ligne de commande

```
$ sudo apt update
$ sudo apt install -y build-essential zlib1g-dev libffi-dev libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev
```

Cliquez ensuite sur [le lien suivant](#) afin de télécharger la dernière version en date de Python. J'ai eu à choisir l'archive `.tar.xz` (XZ compressed source tarball).

Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive:

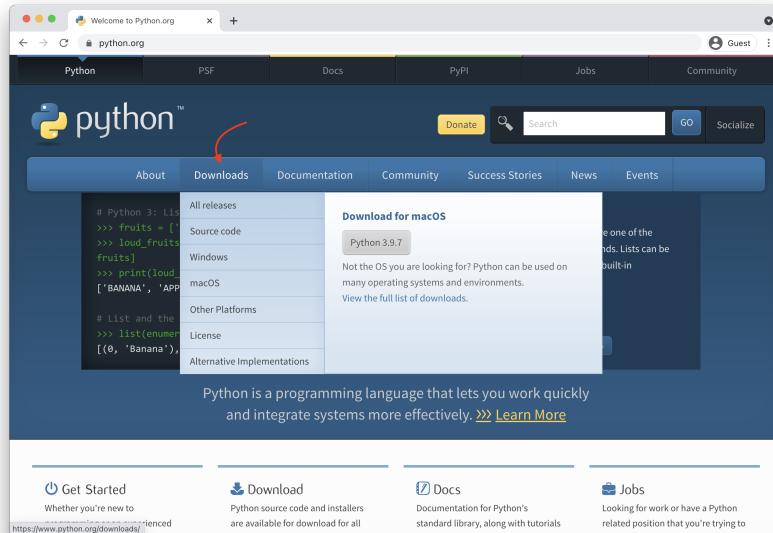
1. Décompressez l'archive en tapant: `tar xf Python-3.13.0.tar.xz` (cette commande est bien entendu à adapter selon la version et le type de compression du fichier téléchargé).
2. Une fois la décompression terminée, vous devez vous rendre dans le dossier qui vient d'être créé dans le répertoire courant (`Python-3.13.0` dans mon cas): `cd Python-3.13.0`.
3. Il vous faudra ensuite lancer le script de configuration en exécutant la commande `./configure --enable-optimizations` au niveau de votre terminal.
4. Une fois la configuration terminée, il n'y a plus qu'à lancer la compilation en exécutant la commande `make`.
5. La dernière étape consiste à exécuter la commande `sudo make altinstall` (en tant que root, super-utilisateur) pour installer Python à proprement dit.

Histoire d'avoir confirmation du fait que Python soit bel et bien installé, redémarrez votre terminal (fermez et réouvrez-le) afin que les changements apportés lors de l'installation soient pris en compte. Dans le nouveau terminal, exécutez la commande `python3.13 --version` ou `python3.13 -V` (avec un `V` majuscule) afin de voir la version de Python installée au niveau de votre ordinateur. Cela devrait afficher quelque chose comme `Python 3.13.0`.

3.3. Installation de Python sous macOS

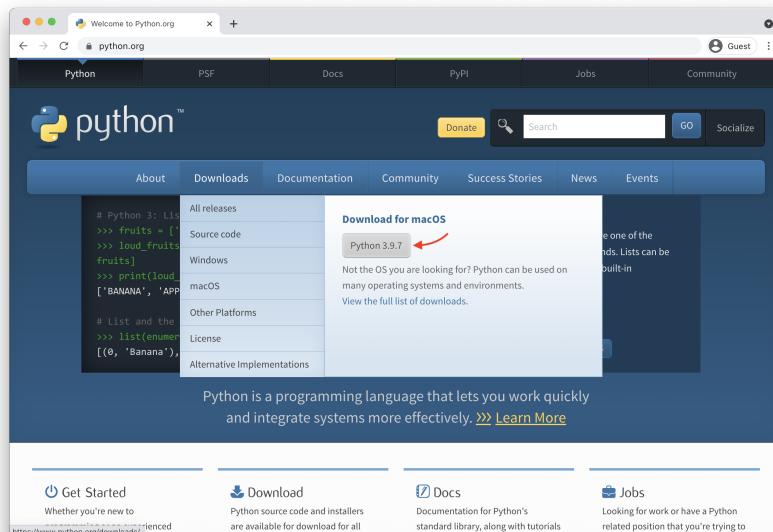
Sur le site officiel de Python, vous trouverez des paquetages pour macOS similaires à ceux proposés sous Windows.

1. Survolez le lien *Downloads* présent au niveau du menu principal de [la page d'accueil du site officiel de Python](#).



Site officiel de Python

2. Vous devriez normalement voir un bouton **Python 3.X.Y** pour télécharger la dernière version en date de Python. Au moment où j'écris ces lignes, il s'agit de la version **3.13.0**.



Bouton de téléchargement de Python depuis le site officiel

3. Cliquez à présent sur le bouton de téléchargement. Sans grande surprise, le téléchargement du fichier d'installation de Python débutera. Dans mon cas, il s'agit d'un fichier **python-3.13.0-macosx10.9.pkg**.
4. Une fois le téléchargement terminé, exécutez le fichier d'installation et suivez les étapes de l'assistant d'installation.
5. À cette étape, vous devriez normalement avoir Python installé au niveau de votre précieux Mac :).



Confirmation d'installation de Python sous macOS

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?

3.4.1. Sous Windows

Recherchez `python` au niveau de la barre de recherche présente au niveau de la barre des tâches. Vous devriez normalement voir Python 3.13 affiché. Cliquez là dessus afin de voir cette sublime console:

A screenshot of a Windows command prompt window titled "Python 3.9 (64-bit)". It displays the Python 3.9.7 interactive shell. The text in the window reads:

```
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> -
```

The window has a standard Windows title bar and scroll bars on the right side.

Shell interactif de Python

3.4.2. Sous Linux

Pendant l'installation de Python, un lien vers l'interpréteur Python devrait normalement avoir été créé. Ce dernier aura comme libellé `python3.X` (où X représente le numéro de la version installée). Si, par exemple, vous avez installé Python 3.13, vous pourrez y accéder grâce à la commande `python3.13` comme présenté ci-dessous:

```
Ligne de commande

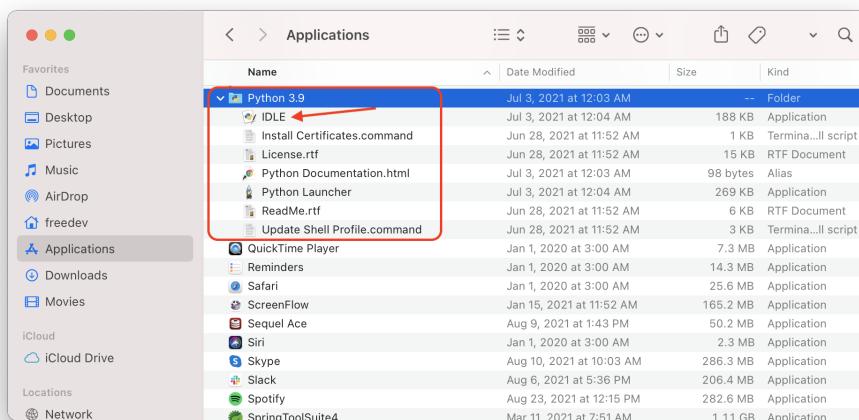
$ python3.13
Python 3.13.0 (default, Oct 7 2024, 00:37:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

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 3.13**. Une fois que vous l'avez trouvé, il suffira d'ouvrir l'application **IDLE** présente dans ce répertoire afin de lancer Python.

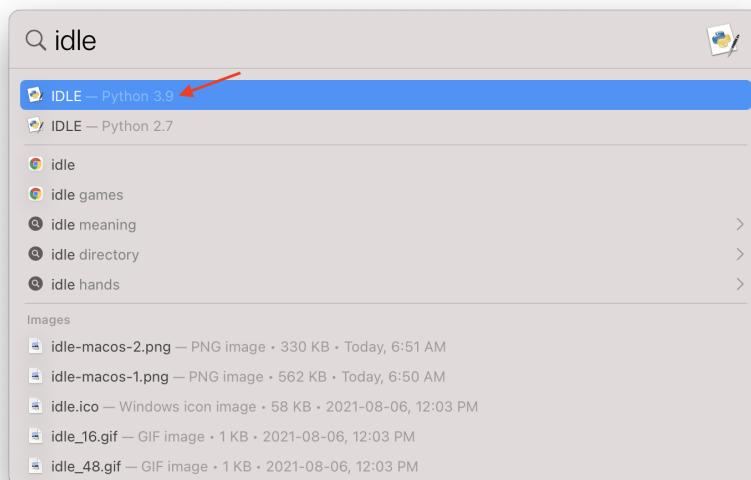


Trouver l'application IDLE via Finder



Trouver l'application IDLE via Finder

Il est également possible d'ouvrir l'application **IDLE** en utilisant la recherche Spotlight. Utilisez le raccourci clavier CMD + Espace et entrez IDLE au niveau du champ de recherche. Vous pouvez à présent cliquer sur l'application IDLE associée à Python 3.13 dans les résultats de recherche.



Trouver l'application IDLE via Spotlight

3.5. Résumé

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

Notre premier jeu

Dans ce chapitre, nous créerons notre premier programme Python. Il s'agira d'un programme assez simple qui nous permettra de nous familiariser assez rapidement avec les concepts de bases de la programmation.

4.1. Le principe du jeu

Le jeu que nous allons concevoir dans ce chapitre est extrêmement simple. Comme je vous l'avais mentionné au tout début, nous commencerons 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 au clavier et afficher par la suite *Mamma Mia* suivi de son nom.

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

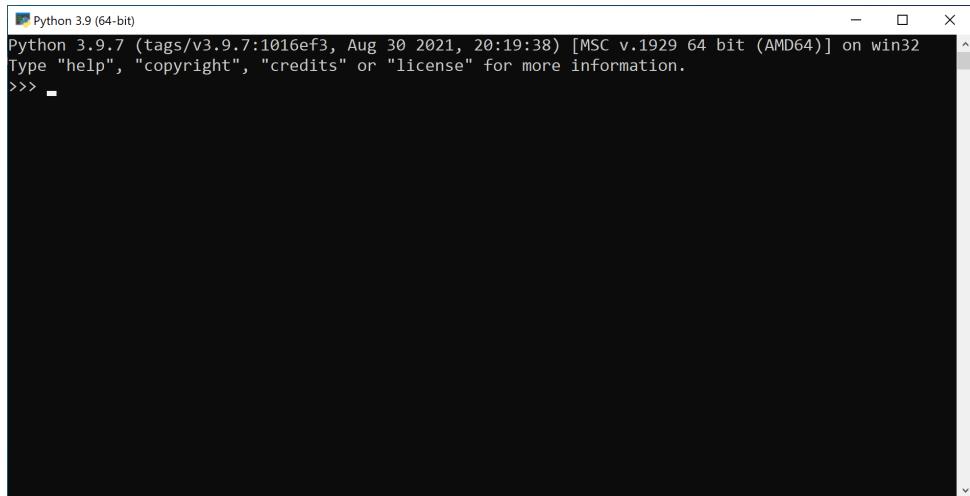
```
Entrez votre nom:  
Honoré Hounwanou  
Mamma Mia Honoré Hounwanou  
>>>
```

Bien avant de se lancer à tête baissée dans l'écriture de notre programme, 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 (encore appelé *Shell interactif de Python*) si ce n'est pas déjà fait!

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



Shell interactif de Python

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

Shell interactif de Python

```
>>> 2  
2  
>>> 9  
9  
>>> 14  
14  
>>>
```

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 l'utiliser comme une simple calculatrice:

Shell interactif de Python

```
>>> 12 + 6  
18  
>>> 11 - 2  
9  
>>> 39 * 2  
78  
>>> 5 / 2  
2.5  
>>> 5 // 2  
2  
>>> 9 % 2  
1  
>>>
```

Vous connaissez sans doute les quatre opérations arithmétiques de base `+`, `-`, `*`, `/`. Mais c'est quoi ces autres 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.

Shell interactif de Python

```
>>> 12 // 5  
2  
>>> 15 // 2  
7  
>>> 9 // 2  
4  
>>>
```

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 est pair s'il est divisible par 2. Ainsi, si `5 % 2` est égal à alors `5` est pair, dans le cas contraire `5` est impair. Aussi simple que cela!

$$\begin{array}{r}
 5 \quad | \quad 2 \\
 - \\
 4 \quad | \quad 2 \\
 \hline
 1
 \end{array}$$

$5 \% 2 = 1$.
 5 est donc impair.

L'opérateur modulo

4.2. Les chaînes 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:

```
Shell interactif de Python

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

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 les messages d'erreur sont 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 simple, double ou triple quotes comme ceci:

Shell interactif de Python

```
>>> 'toto'  
'toto'  
>>> "toto"  
'toto'  
>>> """je peux avoir  
mon texte  
sur plusieurs lignes :)"""  
'je peux avoir\nmon texte\nsur plusieurs lignes :)'  
>>>
```

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 est 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, etc) de manière permanente. Pour cela, il vous faudra utiliser un autre type de mémoire dite *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, etc. 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 petit 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 "Mamma Mia" 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 "Mamma Mia" suivi de ce dernier 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 sachiez 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.

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:

```
Shell interactif de Python

>>> age_de_toto = 21
>>> date_anniversaire_de_toto = "16/05/1993"
>>> 1tata = 3
SyntaxError: invalid syntax
>>>
```

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 déclaré deux variables `age_de_toto` et `date_anniversaire_de_toto` auxquelles nous avons respectivement affectées 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 *undescore* encore appelé *blanc souligné* `_`. Cette manière de nommer les choses est ce qu'on appelle la notation `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 sur ce sujet, je vous inviterai à visiter cette page [PEP 0008 - Style Guide for Python Code](#).

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 également d'éviter d'utiliser les caractères accentués (Ex: é, à, è, ô) dans vos noms de variables même si cela ne générera pas forcément des erreurs.

Shell interactif de Python

```
>>> nom_de_mon_père = "Pierre" # À éviter
>>> nom_de_mon_pere = "Léonard" # Super!
>>> nom_de_ma_mère = "Reine" # Super!
>>>
```

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 recommande fortement d'éviter d'utiliser des caractères accentués dans vos noms de variables afin de ne pas avoir de mauvaises surprises plus tard.

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 les noms des constantes en majuscules histoire d'indiquer aux autres programmeurs 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 [# À éviter](#) et [# Super!](#) 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. Grossso 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 et ne sera donc pas interprété.

Profitez-en c'est gratuit, mais n'en abusez surtout pas comme dans l'exemple ci-dessous:

Shell interactif de Python

```
>>> nom_de_mon_pere = "Léonard" # Je déclare une variable nom_de_mon_pere  
qui a pour contenu la chaîne de caractères "Léonard"  
>>> nom_de_ma_mère = "Reine" # Je déclare une variable nom_de_ma_mère qui  
a pour contenu la chaîne de caractères "Reine"  
>>>
```

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 un langage *sensible à la casse*. Ainsi, pour Python, les variables **toto**, **ToTo**, **TOTo** 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 en théorie utiliser n'importe lequel des éditeurs de texte. Par exemple, le Bloc Notes de Windows ou encore TextEdit sous macOS feront l'affaire. Toutefois, en tant que programmeurs nous préférerons utiliser des éditeurs de code ou des IDEs (Environnements de développement intégré) qui sont des logiciels spécifiquement conçus pour écrire du code.

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 installer Python.

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

Ligne de commande

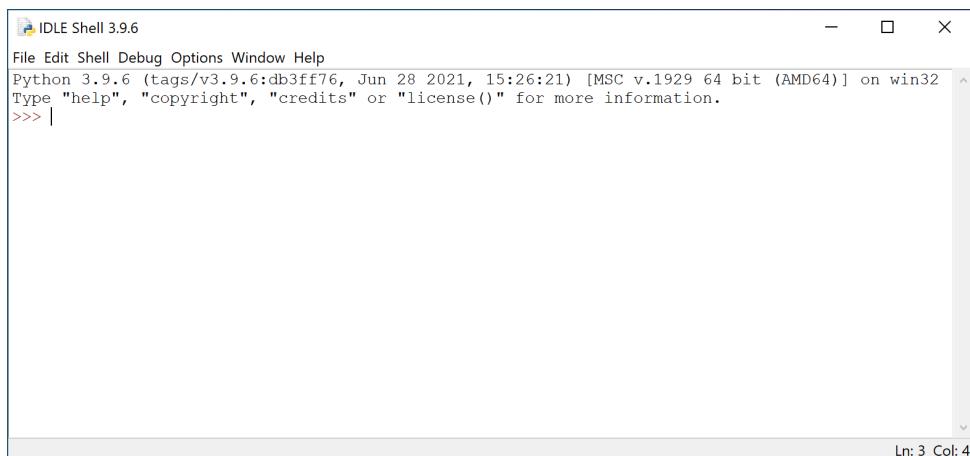
```
$ python3.13 nom_du_fichier.py
```

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. Crédation du fichier jeu1.py

Il nous faudra premièrement ouvrir IDLE. Pour ce faire, on peut passer par les menus Démarrer > Python 3.13 > IDLE (Python 3.13)

Vous devez normalement voir apparaître un truc de ce genre:



IDLE (Integrated Development Environment)

Pour créer ensuite notre programme, 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**.

jeu1.py

```
# Python par la pratique  
# -----  
# Notre premier jeu  
  
print("Entrez votre nom: ")  
nom = input()  
print("Mamma Mia " + nom)
```

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

Notre programme devrait normalement s'exécuter sans aucun problème.

```
>>>  
Entrez votre nom:  
Honoré Hounwanou  
Mamma Mia Honoré Hounwanou  
>>>
```

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 présentement des lignes dans notre code que vous ne comprenez pas, mais ne vous inquiétez surtout pas car je vais expliquer chacune de ces lignes. À partir de maintenant, ce ne sera que de la pratique. Je vous présenterai un challenge, le code qui ira avec et on essayera de l'expliquer instruction après instruction.

jeu1.py

```
1 # Python par la pratique
2 # -----
3 # Notre premier jeu
4
5 print("Entrez votre nom: ")
6 nom = input()
7 print("Mamma Mia " + 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 l'écran. Si je veux afficher *J'aime python.*, alors j'écrirai:

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

Remarquez que là j'utilise les doubles quotes en lieu et place des simples quotes, faute de quoi j'aurai eu droit à une belle erreur.

En effet si j'avais écrits:

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

L'interpréteur se serait dit que ma chaîne de caractère est '**J**' et tout le reste lui aurait été incompréhensible. Si vous voulez à tout prix utiliser des 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.

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

Magnifique!

Revenons à notre programme:

- Au niveau de la ligne **6**, nous utilisons la fonction `input`. Cette fonction permet de récupérer une valeur saisie par l'utilisateur. Nous récupérons donc ce que l'utilisateur aura saisi au clavier et nous stockons cette valeur au niveau de la variable `nom`.
- Il ne reste plus qu'à afficher à présent, "Mamma Mia" 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 bout à bout. 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:

```
print("Mamma Mia nom")
```

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

Et voilà! C'est tout pour ce chapitre. Toutefois, avant de terminer, je vais vous montrer une autre manière d'écrire notre code précédent.

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`.

```
jeu.py

# Python par la pratique
# -----
# Notre premier jeu

nom = input("Entrez votre nom: ")
print("Mamma Mia " + 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!

```
# Python par la pratique  
# -----  
# Notre premier jeu  
  
print("Mamma Mia " + input("Entrez votre nom: "))
```

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

Notre programme est composé d'une seule ligne de code et cette ligne de code demande à afficher "Mamma Mia " + input("Entrez votre nom: ").

L'interpréteur Python verra qu'il peut afficher le texte "Mamma Mia" mais par contre pour afficher l'autre partie faisant usage de la fonction `input`, il faudra demander à l'utilisateur de saisir une valeur au clavier. La fonction `input` entre donc en action, elle recueille le nom de l'utilisateur tout en s'assurant d'afficher bien avant le prompt "Entrez votre nom:".

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

Bien vrai que les trois programmes que nous avons vu produisent le même résultat à l'exécution, j'ai tendance ici à préférer le second qui me semble plus clair et plus lisible. On ne peut pas en dire de même pour le troisième script.

Voici donc notre programme final:

```
jeu.py  
  
# Python par la pratique  
# -----  
# Notre premier jeu  
  
nom = input("Entrez votre nom: ")  
print("Mamma Mia " + nom)
```

4.6. Résumé

Dans ce chapitre, nous avons appris que:

- Le shell interactif de 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.
- La séquence d'échappement `\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 et ne sont pas interprétés.
- Python est un langage sensible à la casse. C'est-à-dire qu'il fait une distinction entre les lettres majuscules et minuscules.
- La fonction `print` permet d'afficher quelque chose au niveau de l'écran.
- 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.

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 le concept de modularité. 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 contenir des fonctions, des classes, etc). 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 soucie pas forcément de comment le code fonctionne mais on l'utilise tout simplement. Par exemple, lorsque nous avons utilisé la fonction `print`, nous ne nous sommes pas souciés de savoir comment est-ce que Python faisait pour afficher à l'écran le texte mis entre les parenthèses. 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 Étape 1, Étape 2, Étape 3 et ainsi de suite.

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

- Étape 1
- Étape 2
- Étape 3
- ...

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

- Étape 1
- Étape 2
- Étape 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:

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

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

```
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. Définition d'une fonction

En lieu et place de dire *créer* une fonction, les programmeurs vous diront *définir une fonction*. Voici donc un exemple de définition de fonction:

```
def nom_de_la_fonction(parametre_1, parametre_2, ..., parametre_n):
    """Docstring de la fonction"""
    Instruction 1
    Instruction 2
    ...
```

5.2.2. Explication

Nous avons 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 fonction sont les mêmes que celles d'une variable.
- 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:

```
jeu1.c

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char nom[30];
    printf("Entrer votre nom: ");
    scanf("%s", nom);
    printf("Mamma Mia %s", nom);

    return 0;
}
```

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, etc). 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.

```
int main(int argc, char **argv) {  
    char nom[30];  
    printf("Entrer votre nom: ");  
    scanf("%s", nom);  
    printf("Mamma Mia %s", nom);  
  
    return 0;  
}
```

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:

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char **argv){  
    char nom[30]; printf("Entrer votre nom: ");scanf("%s",  
nom);printf("Mamma Mia %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,

```
def faire_du_cafe():
    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 et ainsi de suite.

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:

```
def dire_bonjour():
    print("Bonjour Maman")
    print("Bonjour Papa")
    print("Bonjour Gloradie") # Ma grande sœur adorée

    print("Je ne fais pas partie du bloc d'instructions.")
```

- Il faudra mettre premièrement 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:

```
def dire_bonjour():
    print("Bonjour Maman")
    print("Bonjour Papa")
    print("Bonjour Gloradie")

    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:

```
def dire_bonjour():
    print("Bonjour Maman")
    print("Bonjour Papa")
    print("Bonjour Gloradie")

    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 plus 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, etc.

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

```
def dire_bonjour_a(nom):
    """Permet de saluer la personne ayant le nom donné en paramètre"""
    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*. Tâchez 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'appeler!

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

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("Honoré")`, `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 avions 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, nous avons donc notre fonction qui devra exécuter cette instruction:

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

Pour vous, que signifie `"Salut " + 2`? 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 nous avons ceci:

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

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

```
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, si ce n'est pas une chaîne de caractères qui est passée en argument, nous aurons 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 lorsque nous nous sommes 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 faudra 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:

Shell interactif de Python

```
>>> type(2)
<type 'int'>
>>> type("2")
<type 'str'>
>>> type('2')
<type 'str'>
>>>
```

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

Nous aurions pu aussi avoir le type en utilisant l'attribut `__class__`:

Shell interactif de Python

```
>>> (2).__class__
<type 'int'>
>>> "2".__class__
<type 'str'>
>>>
```

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

Shell interactif de Python

```
def dire_bonjour_a(nom):
    """Permet de saluer la personne ayant le nom donné en paramètre"""
    print("Salut " + nom)
>>> dire_bonjour_a.__doc__
'Permet de saluer la personne ayant le nom donné en paramètre'
>>>
```

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

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

```
>>> dire_bonjour_a("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, un entier en réel, etc. Passons sans plus tarder à une petite phase pratique pour ainsi sortir de cette atmosphère d'abstraction.

Shell interactif de Python

```
>>> int("3") # Permet de convertir la chaîne de caractères "3" en entier
3
>>> int(3.0) # Permet de convertir le réel 3.0 en entier
3
>>> int(3.5) # Permet de convertir le réel 3.5 en entier
3
>>> str(3) # Permet de convertir l'entier 3 en chaîne de caractères
'3'
>>> int('123') # Permet de convertir la chaîne de caractères '123' en
entier
123
>>> int('maman')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    int('maman')
ValueError: invalid literal for int() with base 10: 'maman'
>>> float(3)
3.0
>>> int('123.09')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    int('123.09')
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:

```
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:

Shell interactif de Python

```
>>> float('123.09')
123.09
>>>
```

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, on peut ...?

Shell interactif de Python

```
>>> int(float('123.09'))
123
>>>
```

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.

La solution que je vous propose est la suivante:

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

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

```
>>> dire_bonjour_a(2)  
Salut 2  
>>> dire_bonjour_a("Honoré")  
Salut Honoré  
>>>
```

Paramètres avec des 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 afficher *Salut suivi du nom*;
- Dans le cas contraire, elle devra afficher tout simplement Salut Boss.

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

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

Notre fonction `dire_bonjour_a` a donc maintenant une valeur par défaut "`Boss`". Grossièrement, il s'agit d'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.

```
>>> dire_bonjour_a(2)
Salut 2
>>> dire_bonjour_a()
Salut Boss
>>>
```

Vous pouvez donc donner 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.

```
def carre():
    """Première fonction chargée d'afficher une figure"""
    print("""
        *****
        *   *
        *****""")

carre() # Affichera notre joli petit carré

def carre(nombre):
    """Seconde fonction écrasant la première et qui affiche le carré
    d'un nombre passé en argument"""
    resultat = nombre * nombre
    print("Le carré de " + str(nombre) + " est " + str(resultat))

carre(2) # Affichera 4
carre(4) # Affichera 16
carre() # Affichera une erreur
```

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

```
*****
*
*****
Le carré de 2 est 4
Le carré de 4 est 16

Traceback (most recent call last):
  File "C:/Users/honore.h/Desktop/toto.py", line 18, in <module>
    carre()
TypeError: carre() takes exactly 1 argument (0 given)
```

Nous avons une erreur lorsque nous appellons 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.

Affichage d'un dessin

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:

```
def est_pair(nombre):
    """Renvoie True si nombre est pair et False dans le cas contraire"""
    if nombre % 2 == 0:
        return True
    else:
        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 `nombre1` et `nombre2` et retourne le maximum de ces deux nombres. Pour ce faire, nous allons 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, nous allons également donner des valeurs par défaut à `nombre1` et `nombre2`, respectivement 0 et 100.

Notre fonction se présentera donc comme suit:

```
def maximum(nombre1=0, nombre2=100):
    return max(nombre1, nombre2)
```

Nous pourrons donc appeler notre fonction comme suit:

```
maximum() # Retournera 100
maximum(40, 300) # Retournera 300
```

Et si maintenant je veux uniquement préciser la valeur de nombre2? Comment est-ce que je fais?

En effet, si je fais ceci:

```
maximum(500)
```

Notre fonction va considérer **500** comme la valeur affectée à **nombre1** et **nombre2** 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 **nombre2**. Eh bien je n'ai qu'à le dire à ma fonction comme suit:

```
maximum(nombre2=500)
```

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

```
def maximum(nombre1=0, nombre2=100):
    print("Nombre 1:", nombre1)
    print("Nombre 2:", nombre2)

    return max(nombre1, nombre2)
```

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.

```
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:

```
>>> maximum(500)
Nombre 1: 500
Nombre 2: 100
500
>>> maximum(nombre2=500)
Nombre 1: 0
Nombre 2: 500
500
>>>
```

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.

Conditions et 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 souhaiterez 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
!=	Different 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` ou `False`.

Attention

Il faudrait remarquer que le T de `True` et le F de `False` sont en MAJUSCULES.
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:

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

Exemple:

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

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**, nous verrons 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).

```
nombre_de_mains = 3

if nombre_de_mains == 2:
    print("Vous êtes un humain")

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:

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

Exemple:

```
nombre_de_mains = 3

if nombre_de_mains == 2:
    print("Vous êtes un humain")

else:
    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.

Nous verrons donc affiché 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:

```
if condition1:  
    instruction  
    ...  
elif condition2:  
    instruction  
    ...  
elif condition3:  
    instruction  
    ...  
else:  
    instruction  
    ...
```

Exemple:

```
note = 12  
  
if note > 18:  
    print("Vous êtes excellent.")  
elif note > 15:  
    print("Très bien.")  
elif note > 10:  
    print("Pas mal.")  
else:  
    print("Quel nullard :).")
```

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

Ainsi si l'on devait traduire notre programme en français, nous aurions obtenu ceci:

```
note = 12

si note > 18:
    print("Vous êtes excellent.")
sinon si note > 15:
    print("Très bien.")
sinon si note > 10:
    print("Pas mal.")
sinon:
    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 la fonction **est_pair** définie dans le chapitre précédent:

```

def est_pair(nombre):
    """Renvoie True si nombre est pair et False dans le cas contraire"""

    if nombre % 2 == 0:
        return True
    else:
        return False

```

Ce programme utilise donc la forme moyenne **if-else**. Grossièrement, nous demandons 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. Tâchez de vous en souvenir.

```

def petit_test():
    print("Je serai affichée")
    return True
    print("Je suis cool")

```

6.1.8. Les opérateurs logiques

Très souvent l'on est 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.

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

Shell interactif de Python

```
>>> a, b = 4, 5 # a reçoit 4 et b reçoit 5. Assez cool non?  
>>> a == 4 or b == 9 # Vu que l'une des conditions est vraie, nous avons  
donc True  
True  
>>>
```

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.

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

Il faudra donc que nos deux conditions soient vraies pour qu'on ait `True`.

Shell interactif de Python

```
>>> a, b, c = 4, 5, 9 # a reçoit 4 et b reçoit 5, c reçoit 9.  
>>> a == 4 and b == 5 and c == 6 # Vu que la dernière condition est  
fausse, nous obtenons False  
False  
>>>
```

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 donner `True`.

Voyons cela en pratique:

Shell interactif de Python

```
est_malade = True

if not est_malade:
    print("Vous n'êtes pas malade.")
else:
    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, mais `if not est_malade` peut se lire comme: *S'il n'est pas malade.*

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:

```
print(0)
print(1)
print(2)
print(3)

...
# 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 deux (2) **boucles** encore appelées **structures itératives**:

- 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:

```
while condition:  
    instruction  
    ...
```

Ainsi, avec la boucle `while`, pour afficher tous les nombres de 0 à 100, on écrira:

```
i = 0 # On commence à zéro  
  
# Tant qu'on a pas encore atteint 100, on continue  
while i <= 100:  
    print(i)  
    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.

```
i = i + 1
```

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

```
i += 1
```

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

```
i = 10
i = i - 1 # i vaut maintenant 10 - 1 = 9
i -= 1 # i vaut maintenant 9 - 1 = 8 (Forme courte)
print(i) # Affiche 8
```

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

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

Ainsi notre programme précédent pouvait également être écrit comme suit:

```
i = 0

while i <= 100:
    print(i)
    i += 1
```

Notons que sans l'incrémentation de la valeur de la variable `i` à chaque tour de boucle, nous aurions 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. Ainsi lorsque vous écrivez vos boucles, assurez-vous que ces dernières prendront fin à un moment donné. Soyez prudents!

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:

```
for element in sequence:  
    instruction  
    ...
```

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

```
nom = "Honoré"  
for lettre in nom:  
    print(lettre)
```

Affichera:

```
H  
o  
n  
o  
r  
é
```

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:

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

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

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

On aura comme résultat:

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

Comme vous pouvez le voir avec le paramètre `end`, nous pouvons préciser n'importe lequel des séparateurs que nous souhaitons. En gros le paramètre `end` permet de dire après avoir affiché la ou les valeurs passées entre parenthèses mets ceci (le séparateur que vous aurez choisi) à la suite.

Voyons un deuxième exemple avant de clôturer ce chapitre.

```
# Affiche tous les nombres de 0 à 100
for i in range(101):
    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 => 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.

Shell interactif de Python

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

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.

Shell interactif de Python

```
>>> range(15)
range(0, 15)
>>>
```

Le mot-clé `in`

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

En voici une illustration:

```
chaine= "Hello World"

for letter in chaine:
    if letter in "aeiouyAEIOUY":
        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.

```
for i in range(0, 15):
    if i == 2:
        break
    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`.

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

Ce programme affichera:

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

Rappelez-vous, j'ai dit que le `break` arrêtait la boucle appellante 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.
- À 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.

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

Le mot-clé `continue` comme nous venons 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.

Types de données complexes

(Partie 1/2)

Nous avons vu que les variables représentaient un excellent moyen de stocker une valeur en mémoire. De plus, si nous le souhaitons, 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 qui de surcroît ne change pas avec le temps? Disons par exemple, les noms des mois de l'année. Nous pouvons aussi vouloir stocker une longue liste d'informations qui elle change au fil du temps. Par exemple, les noms de tous vos chats. Vous pouvez adopter de nouveaux chats, certains peuvent mourir (chose que je ne souhaite pas), d'autres peuvent par contre se transformer en dîner (nous devrions alors discuter recettes lol). 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 contraint 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 peuvent être stockés à l'aide un tuple à moins que plus tard Février vient à être renommé Mamma Mia?

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 :

```
mois = ("Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
"Août", "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.

Dans la grande majorité des cas, les parenthèses utilisées pour délimiter le début et la fin de notre tuple sont optionnelles. Nous n'étions donc pas obligés de les mettre dans l'exemple ci-dessus. Je vous conseille toutefois de toujours les mettre afin de ne pas vous compliquer la vie.

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.

```
# Je suis également un tuple croyez-moi :)
mois = "Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
"Août", "Septembre", "Octobre", "Novembre", "Décembre"
```

```
# Déclaration de notre tuple
mois = ("Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet",
"Août", "Septembre", "Octobre", "Novembre", "Décembre")

print(mois[0]) # Affiche Janvier
print(mois[2]) # Affiche Mars
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 l'on souhaite accéder.

7.2. Les listes

Les listes comme le nom l'indique permettent de stocker une liste de valeurs. À la différence des tuples, après création d'une liste, vous pourrez ajouter, modifier et supprimer ses valeurs comme bon vous semble. Encore une fois, chaque valeur aura un indice et on commencera à compter à partir de zéro et non 1. Voici un exemple de liste contenant les noms de vos nombreux chats.

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

Comme vous pouvez le voir, le code est exactement le même que celui de la déclaration d'un tuple, sauf que toutes les valeurs sont mises entre crochets, et non entre 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.

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

La liste ci-dessus comporte 3 éléments. Les indices varient donc de 0 à 2. Ainsi, lorsque nous tentons 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.

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

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

```
Shell interactif de Python

>>> nom_de_mes_chats = ["Panpidou", "Milou", "Mamole"] # Déclaration de notre liste
>>> nom_de_mes_chats.append("Bobby") # Ajout d'un élément à la liste
>>> nom_de_mes_chats
['Panpidou', 'Milou', 'Mamole', 'Bobby']
>>> nom_de_mes_chats.remove("Milou") # Suppression d'un élément grâce à sa valeur
>>> nom_de_mes_chats
['Panpidou', 'Mamole', 'Bobby']
>>> nom_de_mes_chats[1] = "Toto" # Modification d'une valeur
>>> nom_de_mes_chats
['Panpidou', 'Toto', 'Bobby']
>>> del nom_de_mes_chats[1] # Suppression d'un élément via son indice
>>> nom_de_mes_chats
['Panpidou', 'Bobby']
>>>
```

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'aurais pu rajouter l'entier `5` à ma liste de chats et personne ne m'aurait blâmé, 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 l'une ou l'autre de ces deux options dépendamment du contexte dans lequel je me trouve.

7.3. Les dictionnaires

Les dictionnaires comme le nom indique représentent des dictionnaires. 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.

L'annuaire téléphonique sera un exemple parfait de dictionnaire.

```
contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000", "Dad" :  
"32283904484"}
```

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 ?

Shell interactif de Python

```
>>> contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000",  
"Dad": "32283904484"}  
  
>>> contacts["Marie"] = "4579303332" # Ajout d'un nouveau contact  
  
>>> contacts  
{'Maman': '77562024848', 'Ma princesse': '000000000', 'Dad': '32283904484',  
'Marie': '4579303332'}  
  
>>> contacts["Maman"] = '30393974' # Modifier un contact  
  
>>> contacts  
{'Maman': '30393974', 'Ma princesse': '000000000', 'Dad': '32283904484',  
'Marie': '4579303332'}  
  
>>> contacts.keys() # Affiche toutes les clés  
dict_keys(['Maman', 'Ma princesse', 'Dad', 'Marie'])  
  
>>> contacts.values() # Affiche toutes les valeurs  
dict_values(['30393974', '000000000', '32283904484', '4579303332'])  
  
>>> del contacts["Ma princesse"] # Suppression d'un contact  
  
>>> contacts  
{'Maman': '30393974', 'Dad': '32283904484', 'Marie': '4579303332'}  
  
>>> mes_cles = list(contacts.keys()) # On convertit notre dict_keys en  
liste et nous stockons le résultat dans la variable mes_cles  
  
>>> mes_cles.sort() # Nous ordonnons ensuite la liste de clés  
  
>>> mes_cles  
['Dad', 'Maman', 'Marie']  
  
>>>
```

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

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

Ainsi très facilement nous déclarons notre dictionnaire **contacts**.

```
contacts = {"Maman" : "77562024848", "Ma princesse" : "000000000", "Dad":  
"32283904484"}
```

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 remarquer que les dictionnaires conservent l'ordre d'insertion par défaut. Ceci n'est vrai que depuis Python 3.6. Avant Python 3.6, les éléments d'un dictionnaire étaient insérés dans un ordre quelconque.

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 méthodes que nous pouvons appliquer aux listes et aux dictionnaires. Malheureusement, nous 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](#) si vous voulez avoir une liste exhaustive des ces méthodes. Je crois bien que c'est le moment de prendre une petite pause café afin de digérer tout ce que nous avons vu jusque-là parce que dans la section suivante nous aborderons un autre gros morceau.

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.
- À la différence des tuples, nous pouvons ajouter, modifier et supprimer des valeurs d'une liste comme bon nous semble.
- Les dictionnaires nous permettent de stocker des paires de clé - valeur.
- Les valeurs d'un dictionnaire ne sont pas numérotées comme c'est le cas pour les listes et les tuples.

Types de données complexes

(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 rentrer un peu plus en profondeur.

Je répéterai volontairement certaines notions que nous avons vu dans le chapitre précédent pour question de pédagogie. Comme on le dit, la répétition est pédagogique.

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 chaînes 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, etc.

Shell interactif de Python

```
>>> chaine = "Python est cool"  
>>> print(chaine[0])  
P  
>>> print(chaine[-1])  
l  
>>>
```

Comme vous pouvez le voir, nous pouvons 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 avec les tuples et les listes.

Shell interactif de Python

```
>>> ma_liste = [1, 2, 3, 4]  
>>> ma_liste[-1]  
4  
>>> mon_tuple = (1, 2, 3, 4)  
>>> mon_tuple[-2]  
3  
>>>
```

Les chaînes de caractères sont dites **immuables** étant donné qu'une fois créées, il est impossible de changer la valeur d'un élément spécifique.

Shell interactif de Python

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

Pour rappel, les tuples sont également immuables. Pour preuve:

Shell interactif de Python

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

8.2.1. Les tranches

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

La syntaxe est la suivante: `sequence[indice_debut : indice_fin]`, pour exprimer le fait que nous souhaitons 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.

Shell interactif de Python

```
>>> chaine = "Python est cool"
>>> chaine[3:6]
'hon'
>>> chaine[4:8]
'on e'
>>> chaine[1:2]
'y'
>>>
```

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

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

la taille de votre séquence.

Cela revient donc à dire que le code ci-dessous:

```
Shell interactif de Python

>>> chaine = "Python est cool"
>>> chaine[::]
'Python est cool'
>>>
```

permettra de récupérer la séquence entière. Il est courant d'utiliser cette technique pour créer une copie d'une liste. Nous y reviendrons un peu plus tard.

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

```
Shell interactif de Python

>>> chaine = "Python est cool"
>>> chaine[4:]
'on est cool'
>>> chaine[:-1]
'Python est coo'
```

8.2.2. Modification d'une chaîne de caractères

À 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 créée!

Vous avez tout à fait raison. Et je maintiens ma position! Le fait est qu'avec l'utilisation du slicing, nous pourrons faire semblant de modifier une chaîne de caractères en trichant un tout petit peu.

Shell interactif de Python

```
>>> chaine = 'baba'  
>>> chaine = 'd' + chaine[1:]  
>>> chaine  
'daba'  
>>> chaine = 'P' + chaine[1] + 'p' + chaine[3:]  
>>> chaine  
'Papa'  
>>>
```

Les chaînes de caractères sont immuables!

Attention, cela ne veut pas dire que les chaînes de caractères sont muables, ce serait alors se contredire. Nous créerons de nouvelles chaînes de caractères en concaténant différentes chaînes de caractères.

8.2.3. Parcourir une chaîne de caractères

Nous pouvons parcourir tous les éléments d'une chaîne de caractères aussi bien avec la boucle **for** qu'avec 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

```
chaine = "coucou"  
  
for caractere in chaine:  
    print(caractere)
```

Avec la boucle while

```
chaine = "coucou"

i = 0
while i < len(chaine):
    print(chaine[i])
    i += 1
```

La fonction `len()`

La fonction `len()` nous renvoie la taille d'une séquence de données. En d'autres termes, le nombre d'éléments qui la composent.

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:

Shell interactif de Python

```
>>> chaine = "Qui est-ce qui n'aime pas PYTHON?"  
>>> chaine.lower() # Conversion des caractères en minuscules  
"qui est-ce qui n'aime pas python?"  
>>> chaine.upper() # Conversion des caractères en majuscules  
"QUI EST-CE QUI N'AIME PAS PYTHON?"  
>>> chaine.swapcase()  
"qUI EST-CE QUI N'AIME PAS python?" # Devinez :)  
>>> chaine.capitalize() # Conversion du premier caractère en majuscule  
"Qui est-ce qui n'aime pas python?"  
>>> chaine.title() # Conversion du premier caractère de chaque mot en  
majuscule.  
"Qui Est-Ce Qui N'Aime Pas Python?"  
>>> chaine = " j'ai des espaces "  
>>> chaine.strip() # Suppression des espaces à gauche et à droite  
"j'ai des espaces"  
>>> chaine.find('e') # Recherche un sous-chaine dans une chaîne de  
caractères. Si la sous-chaine a été trouvée, l'indice de sa première  
occurrence est retourné. Dans le cas contraire la valeur -1 est retournée.  
7  
>>> chaine.find("z")  
-1  
>>> chaine.strip().capitalize().replace('espaces', 'femmes') # Cet exemple  
parle de lui-même :)  
"J'ai des femmes"  
>>> chaine = "BARACK OBAMA"  
>>> chaine.replace("A", "*")  
'B*R*CK OB*M*'>>>
```

Comme vous l'avez remarqué avec `chaine.strip().capitalize().replace('espaces', 'femmes')`, nous avons la possibilité de chaîner différentes méthodes. Ici, nous supprimons premièrement les espaces à gauche et à droite grâce à la méthode `strip()`. Ensuite, nous mettons le premier caractère en majuscule via la méthode `capitalize()` et pour terminer pour le bon plaisir, nous remplaçons `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 découvrir plus de méthodes, je vous inviterai à consulter la documentation.

Attention! les méthodes appliquées à la chaîne de caractères ne la modifie en aucun cas. Elles retournent à chaque fois une nouvelle chaîne de caractères.

Shell interactif de Python

```
>>> chaine = "TEST"  
>>> chaine.lower()  
'test'  
>>> chaine  
'TEST'  
>>> chaine = chaine.lower()  
>>> chaine  
'test'  
>>>
```

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, nous ne pouvons pas les modifier.

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

Shell interactif de Python

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

Nous avons toutefois la possibilité de concaténer plusieurs tuples.

Shell interactif de Python

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

Shell interactif de Python

```
>>> tuple_test = (1, 2, 3, 4)
>>> tuple_new = (5,) # Sans la virgule, ce ne serait plus un tuple mais
l'entier 5.
>>> tuple_test = tuple_test + tuple_new
>>> tuple_test
(1, 2, 3, 4, 5)
>>> tuple_test = tuple_test[0:2] + (2.5,) + tuple_test[3:]
>>> tuple_test
(1, 2, 2.5, 4, 5)
>>> len(tuple_test)
5
>>>
```

8.3.1. Parcourir un tuple

Avec la boucle while

```
tuple_test = (1, 'abc', 3.45)

i = 0
while i < len(tuple_test):
    print(tuple_test[i])
    i += 1
```

Avec la boucle for

```
tuple_test = (1, 'abc', 3.45)

for element in tuple_test:
    print(element)
```

```
tuple_test = (1, 'abc', 3.45)

for i in range(len(tuple_test)):
    print(tuple_test[i])
```

8.4. Les listes

Les listes sont muables, ce qui veut dire que nous pouvons les modifier une fois créées. Dans bien de contextes, cela représente un avantage considérable par rapport aux tuples.

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

Shell interactif de Python

```
>>> liste = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> len(liste)
10
>>>
```

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

Shell interactif de Python

```
>>> liste = [3] * 10
>>> liste
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> len(liste)
10
>>>
```

Nous pouvons utiliser à peu près le même principe avec les chaînes de caractères. Admirez:

Shell interactif de Python

```
>>> "cou" * 2
'coucou'
>>> "cou" * 4
'coucoucoucou'
>>>
```

Le principe du slicing reste également valide:

Shell interactif de Python

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

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:

Shell interactif de Python

```
>>> liste1 = ['Paris', 'Rabbat', 'Abidjan']
>>> liste2 = liste1 # liste1 et liste2 sont maintenant identiques.
Modifier liste2 impliquera également la modification de liste1
>>> liste3 = liste1[:] # Nous copions juste les valeurs de liste1 dans
liste3. Ce sont donc deux listes distinctes
>>> liste2[1] = 'Dakar' # liste1 sera aussi modifiée
>>> liste1
['Paris', 'Dakar', 'Abidjan']
>>> liste2
['Paris', 'Dakar', 'Abidjan']
>>> liste3 # liste3 n'est pas du tout modifiée
['Paris', 'Rabbat', 'Abidjan']
>>>
```

Les alias permettent de donner de multiples noms à un objet. Dans l'exemple ci-dessus `liste2` est un alias de `liste1`.

8.5. Les méthodes `split` et `join`

La méthode `split()` permet de convertir une chaîne de caractères en liste.

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

Shell interactif de Python

```
>>> chaine = "Salut les amis"
>>> chaine.split()
['Salut', 'les', 'amis']
>>> liste = ['Salut', 'Monsieur']
>>> ' '.join(liste)
'Salut Monsieur'
>>>
```

Vous avez peut-être remarqué que j'employais le terme **méthodes** en lieu et place de

fonctions. 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 créés, nous ne pouvons 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 après création. Dans bien de contextes, 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.

Modules et Packages

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

9.1. Les modules

9.1.1. Définition

Un module Python n'est rien d'autre qu'un *fichier Python*. Dans un module, nous pouvons regrouper plusieurs fonctions, variables et classes ayant un rapport entre elles. Par exemple dans un module `calcul`, nous pouvons 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, lorsque nous voudrons travailler avec les fonctionnalités offertes par le module `calcul`, il suffira de *l'importer*.

9.1.2. L'instruction 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:

```
import nom_du_module
```

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

```
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.

Le dossier pycache

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 rapidement 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 depuis 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:

```
import math

print(math.ceil(1.5))  # Affichera 2
print(math.floor(1.5)) # Affichera 1
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.

Nous pouvons également afficher la documentation d'une fonction en tapant:

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

Exemple:

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

```
sqrt(x)
```

```
Return the square root of x.
```

```
>>>
```

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:

Shell interactif de Python

```
>>> help("math")
```

```
Help on built-in module math:
```

NAME

```
math
```

DESCRIPTION

```
This module is always available. It provides access to the mathematical functions defined by the C standard.
```

FUNCTIONS

```
acos(...)
```

```
acos(x)
```

```
Return the arc cosine (measured in radians) of x.
```

```
acosh(...)
```

```
acosh(x)
```

```
Return the hyperbolic arc cosine (measured in radians) of x.
```

```
asin(...)
```

```
asin(x)
```

```
Return the arc sine (measured in radians) of x.
```

```
asinh(...)
```

```
asinh(x)
```

```
...
```

```
>>>
```

Shell interactif de Python

```
>>> help("math.ceil")  
Help on built-in function ceil in module math:  
  
math.ceil = ceil(...)  
    ceil(x)  
  
    Return the ceiling of x as an int.  
    This is the smallest integral value >= x.  
>>>
```

Les fameux underscores

Lorsque nous aborderons la notion de programmation orientée Objet en Python, vous comprendrez davantage pourquoi j'ai eu à surnommer Python le langage des underscores.

9.1.5. Créer un alias d'espace de noms

Via l'instruction `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 certain préfixe.

Pour changer l'espace de noms sous lequel le module sera importé, il faudra procéder comme suit:

```
import nom_du_module as nouveau_nom
```

Les fonctions du module seront dorénavant accessibles via `nouveau_nom.fonction`

```
import math as m
print(m.fabs(-0.5)) # Affiche 0.5

print(math.fabs(-0.5)) # Affiche une erreur vu que le namespace math
n'existe plus!
```

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

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

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

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

Avec cette seconde forme d'importation, il est possible d'indiquer de manière explicite les fonctions que nous souhaitons importer.

Si nous souhaitons 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:

```
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 faudra plus préfixer la fonction avec le nom du module concerné.

Voyons ceci avec des exemples pratiques:

```
Shell interactif de Python

>>> from math import ceil, sqrt
>>> sqrt(25)
5.0
>>> ceil(5.7)
6
>>> fabs(12) # Cette fonction n'a pas été importée
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fabs(12)
NameError: name 'fabs' is not defined
>>> math.ceil(0.5) # Plus besoin de préfixe math
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    math.ceil(0.5)
NameError: name 'math' is not defined
>>>
```

Bien que fonctionnelle, il est généralement recommandé dans la mesure du possible d'éviter les importations de la forme `from nom_du_module import *`, étant donné que ces derniers peuvent causer des effets de bord.

9.1.7. Créer son propre module

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

```

"""
Module contenant des fonctions pour les 4 opérations arithmétiques de base
"""

def addition(a, b):
    return a + b

def soustraction(a, b):
    return a - b

def multiplication(a, b):
    return a * b

def division(a, b):
    if b != 0:
        return a / b
    else:
        print("Division impossible")

```

Vu que notre fichier a pour nom `calcul.py`, le nom de notre module sera `calcul`. Le nom du module est déduit de manière implicite en se basant sur le nom de votre fichier.

Maintenant que notre module a été créé, nous pouvons l'exécuter comme si nous avions affaire à un programme Python classique.

Ensuite au niveau du shell interactif, tapez par exemple:

Shell interactif de Python

```
>>> help("calcul")  
Help on module calcul:
```

NAME

```
calcul - Module contenant des fonctions pour les 4 opérations  
arithmétiques de base
```

FUNCTIONS

```
addition(a, b)  
  
division(a, b)  
  
multiplication(a, b)  
  
soustraction(a, b)
```

FILE

```
C:\Users\honore.h\Desktop\calcul.py
```

```
>>>
```

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.

```
# test de la fonction addition  
if __name__ == "__main__":  
    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.

Encore une fois tout ce qui a été dit plus haut est purement technique et peut paraître très compliqué. Ce que j'aimerais que vous puissiez retenir par contre est que ce code:

```
# Test de la fonction addition

if __name__ == "__main__":
    addition(1,5)
```

sera exécuté si et seulement si c'est notre fichier `calcul.py` est exécuté directement. Dans le cas où nous importons notre fichier comme un module, ce code ne sera pas exécuté.

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

```
def main():

    print("Hello, World")

if __name__ == "__main__":
    main()
```

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

- Nous avons tout simplement défini une fonction `main()` contenant notre code principal.
- Ensuite, nous spécifions 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. À un moment donné cela deviendra 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, etc.
- 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.
- À 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. Il faudra donc utiliser soit l'une des deux formes d'importation: `from-import` ou `import`.

```
import nom_package  
from nom_package import mon_module
```

9.2.3. Créer un package

Pour créer un package, on commence par créer un dossier. Pour que Python sache que ce dossier est un package, il faudra créer à l'intérieur de ce dernier un fichier spécial nommé `__init__.py`. Le contenu de ce fichier peut être vide, mais sa présence est obligatoire 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 Python 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.

Jeu de capitales

10.1. Le principe du jeu

Le jeu que nous allons réaliser dans ce chapitre sera un tout petit 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:

```
Quelle est la capitale de ce pays: Gabon?  
Libreville  
Bonne réponse!  
Quelle est la capitale de ce pays: Burkina Faso?  
OuAGADOUgou  
Bonne réponse!  
Quelle est la capitale de ce pays: Mozambique?  
Je ne sais pas :)  
Mauvaise réponse! Il fallait répondre: Maputo  
Quelle est la capitale de ce pays: France?  
Paris  
Bonne réponse!  
Quelle est la capitale de ce pays: Sénégal?  
Dakar  
Bonne réponse!
```

C'est terminé! <<Score: 4/5 >>

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>.

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 hypersophistiqués ont donc déjà été mis en place pour gérer tous ces mushibishis (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](#). Celle-ci contient également des exemples concrets d'utilisation.

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 $[0.0, 1.0[$.

```
>>> import random # On importe le module random
>>> random.random()
0.6879157636029883
>>> random.random()
0.739628459893686
>>> random.random()
0.10960434825810839
```

```
>>> random.random()
0.9443851801331767
>>> random.random()
0.22761731499433735
>>> random.random()
0.9221146293989106
>>> random.random()
0.40612880708816845
>>> random.random()
0.5208338016201847
>>>
```

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:

```
>>> import random # Toujours importer le module avant de l'utiliser
>>> random.randrange(3) # Génère un nombre aléatoire entre 0, 1, 2
1
>>> random.randrange(3)
2
>>> random.randrange(3)
2
>>> random.randrange(4, 9) # Génère un nombre aléatoire entre 4, 5, 6, 7
et 8
7
>>> random.randrange(0, 1) # Génère un nombre aléatoire entre 0 et 1 [1
exclus]. On aura donc toujours 0
0
>>> random.randrange(0, 1)
```

```
0
>>> random.randrange(0, 10, 3) # Génère un nombre aléatoire entre 0, 3, 6
et 9
0
>>> random.randrange(0, 10, 3)
6
>>> random.randrange(0, 10, 3)
9
>>>
```

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].

```
>>> import random
>>> random.randint(0,1)
1
>>> random.randint(0,1)
0
>>> random.randint(10,100)
47
>>> random.randint(10,100)
62
>>>
```

10.2.4. La fonction choice

Synopsi: `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.

```
>>> random.choice('123456')
'5'
>>> random.choice('123456')
'4'
>>> random.choice('123456')
'5'
>>> random.choice('123456')
'2'
>>> random.choice(1, 2, 4) # Ici les parenthèses sont obligatoires pour un tuple
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    random.choice(1, 2, 4)
TypeError: choice() takes 2 positional arguments but 4 were given
>>> random.choice((1, 2, 4))
2
>>> random.choice((1, 2, 4))
4
>>> random.choice(["toto", "tata", "mama"])
'mama'
>>> random.choice(["toto", "tata", "mama"])
'toto'
>>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
>>> random.choice(data)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    random.choice(data)
  File "C:\Python34\lib\random.py", line 256, in choice
```

```

    return seq[i]

KeyError: 0

>>> random.choice(data.keys())

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    random.choice(data.keys())
  File "C:\Python34\lib\random.py", line 256, in choice
    return seq[i]

TypeError: 'dict_keys' object does not support indexing

>>> data.keys()

dict_keys(['Gabon', 'Chine', 'France'])

>>> data.values()

dict_values(['Libreville', 'Pekin', 'Paris'])

>>> random.choice(list(data))

'Gabon'

>>> random.choice(list(data.keys()))

'Chine'

>>> random.choice(list(data.values()))

'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:

```

>>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
>>> 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:

```

>>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}
>>> 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:

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

Ce qui est retourné n'est pas tout simplement une liste, mais un fameux type spécial `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.

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

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

Vous pouvez maintenant comprendre aisément pourquoi les instructions suivantes ont

fonctionné:

```
>>> data = {"France": "Paris", "Gabon": "Libreville", "Chine": "Pekin"}  
>>> random.choice(list(data))  
'Gabon'  
>>> random.choice(list(data.keys())) # Similaire à l'instruction  
 précédente  
'Chine'  
>>> random.choice(list(data.values()))  
'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.

```
>>> random.sample(range(10), 3)  
[8, 3, 2]  
>>> random.sample([2,345, 5, "toto"], 3)  
[2, 5, 'toto']  
>>> random.sample("tatattoto", 4)  
['t', 't', 'o', 'a']  
>>> data = {"France" : "Paris", "Gabon" : "Libreville", "Chine" : "Pekin"}  
>>> random.sample(data.keys(), 2) # Cette fois-ci on peut utiliser  
 directement data.keys()  
['Chine', 'Gabon']  
>>> random.sample(list(data), 2)  
['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 à relire le principe du jeu afin de vous rappeler 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 parfaite d'Évangile.

```
# Jeu de capitales
# Python par la pratique

# Nous aurons besoin uniquement de la fonction sample, pas besoin
# alors d'importer tout le module
from random import sample

# Ensemble de pays et leurs capitales respectives
capitales = {
    "Senegal" : "Dakar",
    "Nigeria": "Lagos",
    "France": "Paris",
    "Gabon" : "Libreville",
    "Burkina Faso": "Ouagadougou",
    "Allemagne" : "Berlin",
    "Belgique" : "Bruxelles",
    "Qatar" : "Doha",
```

```

    "Zimbabwe" : "Harare",
    "Perou" : "Lima",
    "Mozambique" : "Maputo"
}

# On utilise ici une constante.

# Ainsi, si nous voulons modifier plus tard le
# nombre de questions, nous le ferons à cet unique
# emplacement et notre changement sera répercuté partout ailleurs
NBRE_TOTAL_DE_QUESTIONS = 5

# Contiendra le score de l'utilisateur
score = 0

# On récupère une liste aléatoire de [NBRE_TOTAL_DE_QUESTIONS] pays
liste_pays = sample(list(capitales), NBRE_TOTAL_DE_QUESTIONS)

for pays in liste_pays:
    print("Quelle est la capitale de ce pays: " + pays + " ?")
    reponse = input() # On récupère la réponse de l'utilisateur

    # La méthode lower() nous permet de convertir une chaîne de caractères
    # en minuscules. On le fait pour qu'ainsi si l'utilisateur tape
    # OuagadoUgou en lieu
    # et place de Ouagadougou, cela fonctionnera également.

    if reponse.lower() == capitales[pays].lower():
        print("Bonne réponse!")
        score += 1 # On incrémente son score
    else:
        print("Mauvaise réponse! Il fallait répondre: " + capitales[pays])

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 aurions été dans l'obligation de gérer par 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 peu alourdi. Je vous laisse juger par vous-même.

```
# On utilise cette fois-ci la fonction choice

from random import choice

# Ensemble de pays et leurs capitales respectives
capitales = {
    "Senegal" : "Dakar",
    "Nigeria": "Lagos",
    "France": "Paris",
    "Gabon" : "Libreville",
    "Burkina Faso": "Ouagadougou",
    "Allemagne" : "Berlin",
    "Belgique" : "Bruxelles",
    "Qatar" : "Doha",
    "Zimbabwe" : "Harare",
    "Perou" : "Lima",
    "Mozambique" : "Maputo"
}

NBRE_TOTAL_DE_QUESTIONS = 5
score = 0

# Contiendra la liste des pays déjà choisis afin de ne pas
# afficher plus d'une fois la même question dans une même série
pays_deja_choisis = []
```

```

# On récupère la liste des pays
liste_pays = list(capitales.keys())

# On répète ce bloc [NBRE_TOTAL_DE_QUESTIONS] fois
for i in range(0, NBRE_TOTAL_DE_QUESTIONS):
    pays = choice(liste_pays) #On choisi un pays de manière aléatoire

    # Si le pays fait déjà partie de la liste, on en choisi un autre, et ce,
    # tant que nous n'avons pas trouvé un pays qui ne fait pas déjà partie
    # de la liste.

    while pays in pays_deja_choisis:
        pays = choice(liste_pays)

    # On ajoute le pays qui vient d'être sélectionné
    # à notre liste de pays déjà choisis pour ne
    # pas avoir à le choisir une fois de plus
    pays_deja_choisis.append(pays)

    print("Quelle est la capitale de ce pays: " + pays + " ?")
    reponse = input()

    # La méthode lower() nous permet de convertir une chaîne de caractères
    # en minuscules. On le fait pour qu'ainsi si l'utilisateur tape
    # OuagadoUgou en lieu
    # et place de Ouagadougou, cela fonctionnera également.

    if reponse.lower() == capitales[pays].lower():
        print("Bonne réponse!")
        score += 1 # On incrémente son score
    else:
        print("Mauvaise réponse! Il fallait répondre: " + capitales[pays])

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.

```
reponse = input("Voulez-vous rejouer (O/N)?")\n\nif reponse.lower() == 'o':\n    # L'utilisateur veut rejouer\nelse:\n    # On lui dit Bye Bye
```

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

```
reponse = input("Voulez-vous rejouer (O/N)?")\n\nif reponse == 'O' or reponse == 'o':\n    # L'utilisateur veut rejouer\nelse:\n    # On lui dit Bye Bye
```

Rendez ensuite votre code modulaire en le découplant 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.

Fichiers et 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 disais 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 est largement facilitée.

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

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

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

```
fichier = open("C:/Users/honore.h/Desktop/mon_fichier.txt")
# Faire quelque chose avec les données récupérées depuis le fichier
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`

```
>>> fichier = open("C:/Users/honore.h/Desktop/mon_fichier.txt")
>>> fichier
<_io.TextIOWrapper name='C:/Users/honore.h/Desktop/mon_fichier.txt'
mode='r' encoding='cp1252'>
>>>
```

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:

```
Dakar-Sénégal  
Lagos-Nigeria  
Paris-France  
Libreville-Gabon  
Ouagadougou-Burkina Faso  
Berlin-Allemagne  
Bruxelles-Belgique  
Doha-Qatar  
Harare-Zimbabwe  
Amsterdam-Pays-Bas  
Lima-Pérou  
Maputo-Mozambique  
Monaco  
Monrovia-Liberia  
Panama  
Rome-Italie
```

Un peu de pratique

```
>>> import os # On importe le module OS depuis la librairie standard  
>>> os.getcwd() # On détermine le répertoire courant  
'C:\\\\Users\\\\honore.h'  
>>> os.chdir("C:/Users/honore.h/Desktop/") # On choisit le répertoire qui  
contient notre fichier capitales.txt comme nouveau répertoire de travail  
>>> os.getcwd() # Nous confirmons que nous sommes maintenant dans le bon  
répertoire  
'C:\\\\Users\\\\honore.h\\\\Desktop'
```

```
>>> fichier = open("capitales.txt") # On ouvre notre fichier et on affecte la valeur rentrée à une variable fichier  
>>> print(fichier.readline()) # On utilise la méthode "readline()" pour lire une ligne au niveau de notre fichier, puis on utilise la fonction "print()" pour afficher la ligne récupérée.
```

Sénégal-Dakar

```
>>> print(fichier.readline(), end="") # Pour ne pas avoir cette ligne vide de trop, on rajoute l'argument end
```

Nigeria-Lagos

```
>>> fichier.seek(0) # Utilisons la méthode "seek()" pour placer le curseur au début de notre fichier.
```

0

```
>>> for ligne in fichier: # Juste une boucle for pour lire toutes les lignes au niveau de notre fichier  
    print(ligne, end="")
```

Dakar-Sénégal

Lagos-Nigeria

Paris-France

Libreville-Gabon

Ouagadougou-Burkina Faso

Berlin-Allemagne

Bruxelles-Belgique

Doha-Qatar

Harare-Zimbabwe

Amsterdam-Pays-Bas

Lima-Pérou

Maputo-Mozambique

Monaco

Monrovia-Liberia

Panama

Rome-Italie

```
>>> fichier.close() # N'oubliions pas de fermer notre fichier vu que nous avons terminé.
```

>>>

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>.

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.

```
>>> texte = "Maman: Je t'aime mon fils."
>>> texte.split(":")
['Maman', ' Je t'aime mon fils.']
>>> texte = "Paris-France"
>>> texte.split("-")
['Paris', 'France']
>>>
```

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.

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

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

retournée.

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

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

```
>>> texte = "Maman: Je t'aime mon fils."
>>> donnees = texte.split(":") # Les parenthèses sont optionnelles
>>> nom = donnees[0]
>>> message = donnees[1]
>>> nom
'Maman'
>>> message
" Je t'aime mon fils."
>>>
```

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:

```
>>> texte = "Maman: Je t'aime mon fils."
>>> nom, message = texte.split(":") # Les parenthèses sont optionnelles
>>> nom
'Maman'
>>> message
" Je t'aime mon fils."
>>>
```

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

```
>>> fichier = open('capitales.txt')
>>> for ligne in fichier:
    capitale, pays = ligne.split("-")
    print(capitale, end=" est la capitale de ")
    print(pays, end="")
```

```
Dakar est la capitale de Sénegal  
Lagos est la capitale de Nigeria  
Paris est la capitale de France  
Libreville est la capitale de Gabon  
Ouagadougou est la capitale de Burkina Faso  
Berlin est la capitale de Allemagne  
Bruxelles est la capitale de Belgique  
Doha est la capitale de Qatar  
Harare est la capitale de Zimbabwe  
Traceback (most recent call last):  
  File "<pyshell#37>", line 2, in <module>  
    capitale, pays = ligne.split("-")  
ValueError: too many values to unpack (expected 2)  
>>>
```

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:

```
Dakar-Sénégal  
Lagos-Nigeria  
Paris-France  
Libreville-Gabon  
Ouagadougou-Burkina Faso  
Berlin-Allemagne  
Bruxelles-Belgique  
Doha-Qatar  
Harare-Zimbabwe  
Amsterdam-Pays-Bas  
Lima-Pérou  
Maputo-Mozambique
```

```
Monaco
Monrovia-Liberia
Panama
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:

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

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.

```
>>> texte = "Amsterdam-Pays-Bas"
>>> help(texte.split)
Help on built-in function split:

split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string. If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
```

```
whitespace string is a separator and empty strings are  
removed from the result.
```

```
>>>
```

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.

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

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

```
>>> fichier = open("capitales.txt")  
>>> for ligne in fichier:  
    capitale, pays = ligne.split("-", 1)  
    print(capitale, end=" est la capitale de ")  
    print(pays, end="")
```

```
Dakar est la capitale de Sénégal  
Lagos est la capitale de Nigeria  
Paris est la capitale de France  
Libreville est la capitale de Gabon  
Ouagadougou est la capitale de Burkina Faso  
Berlin est la capitale de Allemagne  
Bruxelles est la capitale de Belgique  
Doha est la capitale de Qatar  
Harare est la capitale de Zimbabwe  
Amsterdam est la capitale de Pays-Bas
```

```
Lima est la capitale de Pérou  
Maputo est la capitale de Mozambique  
Traceback (most recent call last):  
  File "<pyshell#53>", line 2, in <module>  
    capitale, pays = ligne.split("-", 1)  
ValueError: need more than 1 value to unpack  
>>>
```

Astuce

IDLE vous donne accès à l'entièvre 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.

```
>>> ligne = "Monaco"
>>> ligne.find("-")
-1
>>> ligne = "Paris-France"
>>> ligne.find("-")
5
>>>
```

- **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:

```
>>> fichier = open("capitales.txt")
>>> for ligne in fichier:
    if ligne.find("-") == -1: #Si on ne trouve pas de tiret sur la
ligne
        capitale = pays = ligne #Alors la ligne correspond à la fois à
la capitale et au pays
    else:
        capitale, pays = ligne.split("-", 1)

    print(capitale, end=" est la capitale de ")
```

```
print(pays, end="")  
  
Dakar est la capitale de Sénégal  
Lagos est la capitale de Nigeria  
Paris est la capitale de France  
Libreville est la capitale de Gabon  
Ouagadougou est la capitale de Burkina Faso  
Berlin est la capitale de Allemagne  
Bruxelles est la capitale de Belgique  
Doha est la capitale de Qatar  
Harare est la capitale de Zimbabwe  
Amsterdam est la capitale de Pays-Bas  
Lima est la capitale de Pérou  
Maputo est la capitale de Mozambique  
Monaco  
    est la capitale de Monaco  
Monrovia est la capitale de Liberia  
Panama  
    est la capitale de Panama  
Rome est la capitale de Italie  
>>> fichier.close()  
>>>
```

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:

```
>>> fichier = open("capitales.txt")
```

```

>>> for ligne in fichier:
    if ligne.find("-") == -1:
        capitale = pays = ligne
    else:
        capitale, pays = ligne.split("-", 1)

    # C'est ici que la magie s'opère
    print(capitale.strip(), end=" est la capitale de ")
    print(pays, end="")

```

Dakar est la capitale de Sénégal
 Lagos est la capitale de Nigeria
 Paris est la capitale de France
 Libreville est la capitale de Gabon
 Ouagadougou est la capitale de Burkina Faso
 Berlin est la capitale de Allemagne
 Bruxelles est la capitale de Belgique
 Doha est la capitale de Qatar
 Harare est la capitale de Zimbabwe
 Amsterdam est la capitale de Pays-Bas
 Lima est la capitale de Pérou
 Maputo est la capitale de Mozambique
 Monaco est la capitale de Monaco
 Monrovia est la capitale de Liberia
 Panama est la capitale de Panama
 Rome est la capitale de Italie
 >>> fichier.close()
 >>>

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.

```
Traceback (most recent call last):
  File "<pyshell#53>", line 2, in <module>
    capitale, pays = ligne.split("-", 1)
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:

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

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

```
try:  
    instruction  
    ...  
except:  
    instruction  
    ...
```

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

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

À l'exécution, nous obtenons la sortie suivante:

```
>>>  
Division par zéro illégale!
```

```
>>>
```

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.

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

À l'exécution, nous obtenons la sortie suivante:

```
>>>  
Division par zéro illégale!  
>>>
```

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.

```
try:  
    #Bloc d'instructions  
except TypeDeNotreException as exception_retournee:  
    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:

```
try:  
    calcul = 3 / 0  
except ZeroDivisionError as exception:
```

```
print("Voici l'erreur:", exception)
```

À l'exécution, on obtient le résultat suivant:

```
>>>  
Voici l'erreur: division by zero  
>>>
```

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.

```
try:  
    resultat = numerateur / denominateur  
except NameError:  
    print("La variable numérateur ou dénominateur n'a pas été définie.")  
except TypeError:  
    print("La variable numérateur ou dénominateur possède un type  
incompatible avec la division.")  
except ZeroDivisionError:  
    print("La variable dénominateur est égale à 0.")  
else:  
    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 survenait, 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.

```
try:  
    # Test d'instruction(s)  
  
except TYPE_D_EXCEPTION:  
    # Traitement en cas d'erreur  
  
finally:  
    # 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:

```
try:  
    fichier = open("toto.txt")  
    # Quelques traitements  
    ...  
  
except:  
    # Traitement en cas d'erreur  
  
finally:  
    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.

```
try:  
    # Test d'instruction(s)  
  
except Type_d_instruction:  
    # Rien ne doit se passer en cas d'erreur  
    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:

```
def une_fonction_qui_fera_quelque_chose():
    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.

```
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.

```
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:

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

À l'exécution, vous aurez ceci:

```
>>> que_des_valeurs_paires(2)
Good Job.

>>> que_des_valeurs_paires(14)
Good Job.
```

```

>>> que_des_valeurs_paires(1)

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    que_des_valeurs_paires(1)
  File "C:\Users\honore.h\Desktop\game.py", line 5, in que_des_valeurs_paires
    raise ValueError("Seules les valeurs paires sont autorisées. Merci.")
ValueError: Seules les valeurs paires sont autorisées. Merci.

>>>

```

Ainsi, avec la gestion des exceptions, nous pourrons écrire un bien meilleur programme comme celui-ci:

```

try:
    que_des_valeurs_paires(1)
except ValueError as exception:
    print(exception)

>>>
Seules les valeurs paires sont autorisées. Merci.

>>>

```

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:

```

>>> fichier = open("capitales.txt")
>>> for ligne in fichier:
    capitale, pays = ligne.split("-", 1)
    print(capitale, end=" est la capitale de ")

```

```
print(pays, end="")  
  
Dakar est la capitale de Sénégal  
Lagos est la capitale de Nigeria  
Paris est la capitale de France  
Libreville est la capitale de Gabon  
Ouagadougou est la capitale de Burkina Faso  
Berlin est la capitale de Allemagne  
Bruxelles est la capitale de Belgique  
Doha est la capitale de Qatar  
Harare est la capitale de Zimbabwe  
Amsterdam est la capitale de Pays-Bas  
Lima est la capitale de Pérou  
Maputo est la capitale de Mozambique  
Traceback (most recent call last):  
  File "<pyshell#53>", line 2, in <module>  
    capitale, pays = ligne.split("-", 1)  
ValueError: need more than 1 value to unpack  
>>>
```

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.

```
fichier = open("capitales.txt")
```

```

for ligne in fichier:
    try:
        capitale, pays = ligne.split("-", 1)
    except ValueError:
        capitale = pays = ligne
    finally:
        print(capitale.strip(), end=" est la capitale de ")
        print(pays, end="")

fichier.close()

```

Et comme résultat, au niveau de la console d'interprétation on a:

```

>>>
Dakar est la capitale de Sénégal
Lagos est la capitale de Nigeria
Paris est la capitale de France
Libreville est la capitale de Gabon
Ouagadougou est la capitale de Burkina Faso
Berlin est la capitale de Allemagne
Bruxelles est la capitale de Belgique
Doha est la capitale de Qatar
Harare est la capitale de Zimbabwe
Amsterdam est la capitale de Pays-Bas
Lima est la capitale de Pérou
Maputo est la capitale de Mozambique
Monaco est la capitale de Monaco
Monrovia est la capitale de Liberia
Panama est la capitale de Panama
Rome est la capitale de Italie
>>>

```

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:

```
Dakar est la capitale de Sénégal
Traceback (most recent call last):
  File "C:\Users\honore.h\Desktop\game.py", line 2, in <module>
    for ligne in fichier:
ValueError: I/O operation on closed file.

>>>
```

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.

```
>>> import os
>>> os.path.exists("capitales.txt")
True
>>>
```

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.

```
# Méthode 1: if - else

import os

if os.path.exists("capitales.txt"):
    fichier = open("capitales.txt")

    for ligne in fichier:
        if ligne.find("-") == -1:
            capitale = pays = ligne
```

```

else:
    capitale, pays = ligne.split("-", 1)

    print(capitale.strip(), end=" est la capitale de ")
    print(pays, end="")

fichier.close()

else:
    print("Le fichier de données est manquant.")

```

```

# Méthode 2: Exceptions

# Comme vous le voyez, nous n'avons plus besoin d'importer le module os

try:
    fichier = open("capitales.txt")

    for ligne in fichier:
        try:
            capitale, pays = ligne.split("-", 1)
        except ValueError:
            capitale = pays = ligne

            print(capitale.strip(), end=" est la capitale de ")
            print(pays, end="")

    fichier.close()

except FileNotFoundError:
    print("Le fichier de données est manquant.")

```

Pour savoir qu'il fallait capturer une exception de type `FileNotFoundException`, 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.

```
>>> open("fichier_de_ouf.txt")
```

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    open("fichier_de_ouf.txt")
FileNotFoundError: [Errno 2] No such file or directory:
'fichier_de_ouf.txt'
>>>
```

11.5. Jeu de capitales - 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**. À 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:

```
Sénégal:Dakar
Nigeria:Lagos
France:Paris
Gabon:Libreville
Burkina Faso:Ouagadougou
Allemagne:Berlin
Belgique:Bruxelles
Qatar:Doha
Zimbabwe:Harare
Pays-Bas:Amsterdam
Pérou:Lima
Mozambique:Maputo
Monaco:Monaco
```

Liberia:Monrovia
Panama:Panama
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.

```
# Jeu de capitales v2

# Python par la pratique

from random import sample

try:
    fichier = open("capitales_game.txt")

    # Contiendra l'ensemble des pays et leurs capitales respectives
    capitales = {}

    for ligne in fichier:
        # On récupère chaque pays et sa capitale associée
        parts = ligne.split(":")
        pays = parts[0].strip()
        capitale = parts[1].strip()

        # On ajoute progressivement les données à notre dictionnaire
        capitales[pays] = capitale

    fichier.close() #Fermeture du fichier
```

```

# Nombre total de questions par série
NBRE_TOTAL_DE_QUESTIONS = 6

# Contiendra le score de l'utilisateur
score = 0

# On récupère une liste aléatoire de [NBRE_TOTAL_DE_QUESTIONS] pays
liste_pays = sample(list(capitales), NBRE_TOTAL_DE_QUESTIONS)

for pays in liste_pays:
    print("Quelle est la capitale de ce pays: " + pays + " ?")
    reponse = input()

    if reponse.lower() == capitales[pays].lower():
        print("Bonne réponse!")
        score += 1 # On incrémente son score
    else:
        print("Mauvaise réponse! Il fallait répondre: " +
capitales[pays])

    print("C'est terminé ! << Score: ", score, "/", 
NBRE_TOTAL_DE_QUESTIONS, " >>")

except FileNotFoundError:
    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`.

```
reponse = input("Voulez-vous rejouer (O/N)?")
```

```
if reponse.lower() == "o":  
    # L'utilisateur veut rejouer  
  
else:  
    # On lui dit Bye Bye
```

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

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

Rendez ensuite votre code modulaire en le découplant 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

Waouh, félicitations! Quelle aventure!

Merci énormément d'avoir pris de votre temps pour lire ce livre. J'espère de tout cœur qu'il vous a été bénéfique et que vous avez apprécié découvrir avec moi le fabuleux langage qu'est Python.

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. Vous êtes fin prêts à démarrer vos projets Python tout en utilisant les meilleures pratiques de développement actuelles.

L'ultime secret pour s'approprier un langage, c'est la pratique. Ainsi, à cette étape, je vais vous recommander de pratiquer, pratiquer et pratiquer. Si vous avez des idées de projets que vous souhaitez réaliser, c'est le moment de le faire. Rien qu'en pratiquant, cela vous permettra de découvrir de nouvelles techniques, de nouvelles bibliothèques, et grâce à tout ceci vous verrez que petit à petit vous deviendrez des pros Python.

Si vous avez appris quelque chose grâce à ce livre ou s'il vous a inspiré d'une quelconque manière, j'apprécierai énormément si vous le partagiez avec votre entourage. Un petit mot de votre part sur les réseaux sociaux, aidera d'autres personnes à trouver ce livre. Sentez-vous libre de me taguer sur Twitter à [@etsmo](#).

Mon adresse e-mail hello@parlonscode.com reste ouverte si vous avez des questions, des commentaires ou des suggestions. Ce sera toujours un plaisir de vous répondre.

D'ici là, portez-vous bien et pensez à faire un tour sur la chaîne Youtube des [TEACHERS DU NET](#) ou sur [Parlons Code](#), pour apprendre encore plus sur le monde de la programmation.

Ce livre est gratuit et il le restera. Si tu veux me soutenir, tu peux [m'acheter un petit café ici](#) :).

Révisions

Révision 2 (18/10/2024)

- Passage de Python 3.4 à 3.13.
- Correction de quelques fautes d'orthographe et de grammaire.
- Ajout d'un thème sombre pour le livre.

Révision 1 (15/02/2016)

- Première version du livre.