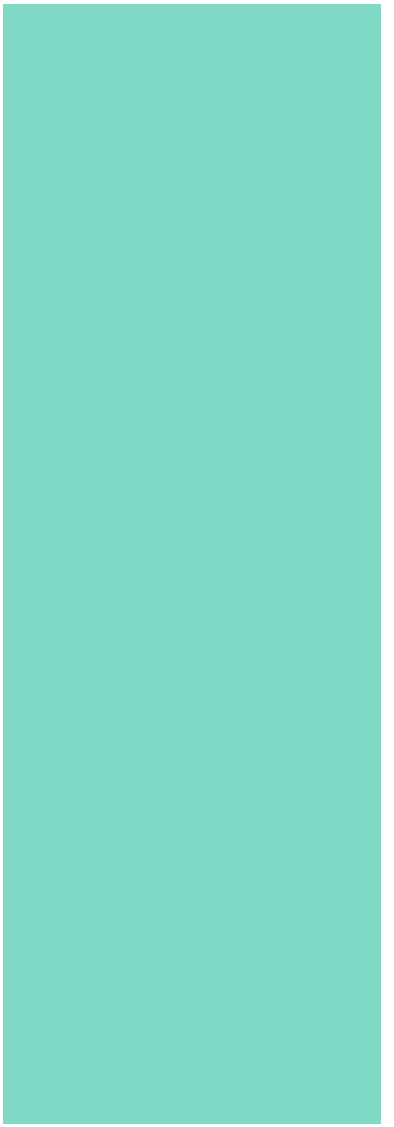
INTRODUCTION PRATIQUE À PYTHON

JEDHA BOOTCAMP

SOMMAIRE

INTRODUCTION



A QUI S'ADRESSE CE MANUEL ?

Ce manuel s'adresse à toutes personnes qui n'ont pas, ou très peu, d'expérience en programmation et qui souhaitent apprendre par la pratique. Nous irons à l'essentiel des connaissances du langage de programmation Python et éviter d'encombrer le lecteur avec des concepts qui ne lui serviront pas.

Python est un langage généraliste, que ce soit en développement web, en analyse de données ou encore dans l'hardware, les applications sont innombrables. Cependant, par souci de synthèse, nous nous concentrerons plutôt sur les applications liées à l'analyse de données et le développement web.

Les principes fondamentaux restent les mêmes mais nous ne couvrerons pas l'utilisation de certains outils comme Django ou Flask. Nous verrons cependant l'utilisation de certaines librairies comme Pandas ou Numpy pour manipuler nos données.

COMMENT UTILISER CE MANUEL ?

Nous tenons à préciser que ce manuel n'est pas un livre théorique qu'il est nécessaire de lire dans son entierété. Vous pouvez sauter des parties et en revoir d'autres puisque toutes sont indépendantes.

Il est tout à fait probable que vous ne compreniez pas tout dès la première lecture. C'est tout à fait normal en programmation. Rien ne sert de rester buté, passez à la suite et faites quelques exercices avant de revenir sur ce que vous n'avez pas compris. Vous allez voir que les choses vont se débloquer petit à petit.

Revenez sur ce manuel souvent, et servez-vous en comme d'une référence théorique une fois que vous vous êtes lancé dans le bain de la programmation.

QUI SONT LES AUTEURS ?

Antoine Krajnc-Rosenthal : Diplômé d'Audencia Business School et de UC Berkeley, Antoine a travaillé pendant plus de 3 ans en tant que Business Analyst à San Francisco et à Paris. Il a ensuite fondé sa première entreprise Evohé qu'il a vendu pour repartir dans la Silicon Valley et fonder le cours de Data Analytics de Product School, le plus grand bootcamp de Product Management des US, qu'il a enseigné pendant 2 ans. Il est aujourd'hui CEO et fondateur du cours de Data Science de Jedha.

Anaïs Armandy : Egalement diplômée d'Audencia Business School, Anaïs a commencé sa carrière chez Ubisoft en tant qu' Assitante Brand Manager puis chez L'Oréal en tant qu'International Project Manager Assistant. Elle a ensuite changé de voie pour aller dans de plus petites structures. Elle a commencé cette nouvelle carrière chez One Conciergerie, une startup dédiée à la concierge où elle exercera le poste de Chief Marketing Officer avant de rejoindre Jedha.

JEDHA ?

Chez Jedha, nous faisons le pari de faire de toute personne motivée débutante ou non, un expert en Data Science.

L'équipe porte en son coeur la volonté de transmettre un savoir complexe à ceux qui sont prêts à se donner les moyens d'apprendre. C'est d'ailleurs toujours une grande fierté de voir en chacun de nos élèves la preuve qu'il est possible d'apprendre et de maîtriser des concepts difficiles si l'on est volontaire.

Le domaine de la Data est encore nouveau et paraît trop souvent inatteignable. Ce n'est pourtant pas le cas. Comme toutes matières, il nécessite l'apprentissage de certains fondamentaux, comme la programmation ; mais rien n'est insurmontable. Il suffit simplement de croire en ses capacités et se dire qu'on peut y arriver. Après tout, si d'autres l'on fait, pourquoi pas vous ?

C'est dans cet état d'esprit que nous avons fondé Jedha ainsi que les deux

cours phares de l'école. Nous avons commencé par un cours de Data Science à temps partiel durant lequel nous apprenons aux étudiants les fondamentaux fin qu'ils découvrent toutes les possibilités qu'offre cette voie. Puis, nous avons ensuite ouvert un programme de 12 semaines durant lesquelles nos élèves vont plus loin dans l'apprentissage du domaine au point d'en devenir experts et d'être prêts à résoudre des problématiques complexes posées en entreprise.

UN PEU D'HISTOIRE

ORIGINE DU LANGAGE

Les origines de Python remontent à 1989. Son créateur, Guido van Rossum participe alors au développement du langage ABC au sein du CWI à Amsterdam. Alors que l'équipe en charge du système d'exploitation Amoeba à laquelle il est rattaché rencontre des problèmes de compatibilité entre les appels systèmes d'Amoeba et le Bourne Shell utilisé comme interface utilisateur; Guido réalise qu'un langage de script, inspiré d'ABC et ayant accès au système Amoeba pourrait interpréter les commandes pour Amoeba et ce, de manière beaucoup plus efficace.

Pendant les vacances de Noël 1989, il commence à travailler sur ce nouveau langage; ce qu'il continuera à faire sur son temps libre pendant plus d'un an. Tout en le développant, il le fait tester par ses collègues dans le cadre du projet Amoeba. L'aventure est un succès. Les retours de ses collègues lui permettent d'améliorer rapidement et efficacement ce nouveau langage.

En février 1991, après un peu plus d'un an de développement, il poste son travail sur USENET et ce sont les débuts publiques de Python.

Contrairement à la croyance populaire, Van Rossum n'a pas baptisé son langage d'après l'animal, mais d'après "Monthy Python's Flying Circus"; mais c'est le serpent qui l'a emporté comme le souligne le logo où deux serpents s'entremêlent.

LES APPLICATIONS DU LANGAGE

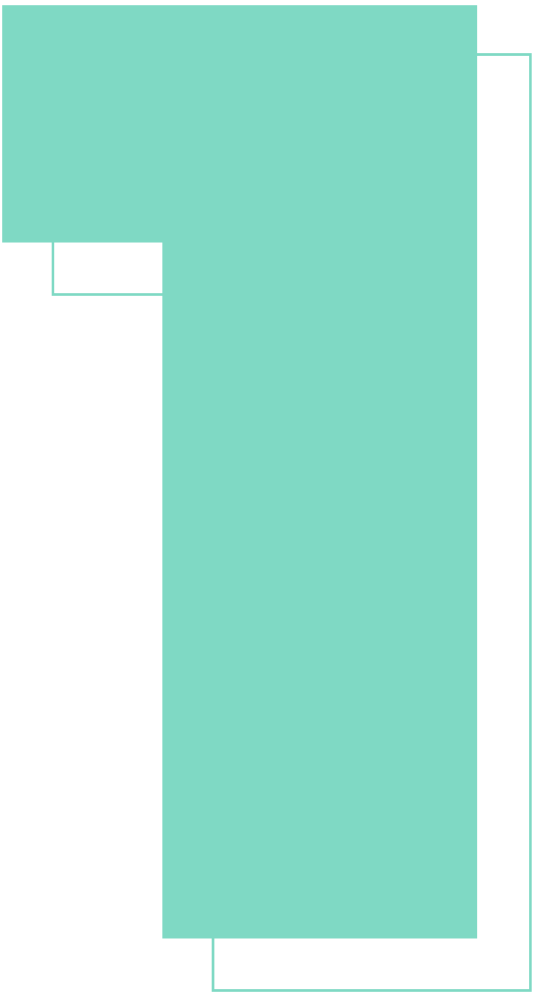
Du fait de son objectif généraliste ; les applications de Python sont multiples.

On peut tout d'abord créer des sites web en Python. De très bonnes technologies ont été inventées pour faciliter la vie des développeurs Python. Ce langage est donc rapidement devenu très apprécié des entreprises. Des organisations comme Dropbox, Buzzfeed, Sephora, Lego, ou encore Google utilisent Python dans le développement de leurs applications.

Des logiciels et mêmes des jeux vidéos sont développés en Python. Par exemple, les Sims 4 ou Battlefield 2 ou même World of Tanks l'utilisent pour certains éléments.

Dans les Data Sciences bien sûr, Python est très utilisé pour mettre en place des algorithmes de Machine Learning qui permettent de faire des prédictions très précises sur des événements donnés.

Ces trois applications concrètes vous montrent à quel point Python est devenu populaire. Bien sûr chacun de ces domaines requiert une expertise particulière et beaucoup de pratique. Il faut bien commencer quelque part, alors commençons à apprendre les fondamentaux.



AVANT DE COMMENCER À CODER

INSTALLER PYTHON

Et oui, avant de commencer à coder, il faut passer par la phase la moins amusante : ***l'installation.*** En l'occurrence : Python. Ne vous en faites pas, nous allons vous donner deux méthodes pour installer tout cela de la manière la plus simple possible.

Que vous soyez sur Mac ou PC, vous pouvez télécharger Python soit directement depuis le site python.org ou, si vous souhaitez vous lancer plutôt dans la Data Science, vous pouvez télécharger Anaconda qui vous installera Python.

POUR MAC

Python possède la version 2 de Python déjà pré-installée sur l'ordinateur. Cependant, comme nous l'avons précisé dans l'introduction, nous allons utiliser la version 3 de Python. Nous allons donc devoir quand même passer par la phase d'installation.

Quoiqu'il en soit, même si vous téléchargez Python 3, **N'EFFACEZ PAS PYTHON 2.** En effet, certaines applications sur Mac tournent encore sur Python 2 et ne sont pas compatible avec Python 3. C'est pour cela que vous devrez vivre avec les deux dans votre ordinateur.

INSTALLER PYTHON DIRECTEMENT DEPUIS PYTHON.ORG

Vous pouvez donc aller sur https://www.python.org/getit/ et cliquez sur Download Python 3.7.0[1]

Vous devriez avoir un dossier *.pkg* qui se télécharge. Une fois le téléchargement fait, vous n'avez plus qu'à l'ouvrir et commencer l'exécution de l'installation.

Aucune spécification d'installation n'est nécessaire, vous aurez simplement à

1. Dernière version de Python à l'écriture du manuel

accepter les conditions d'utilisation de Python avant de démarrer l'installation. Voilà, pas trop sorcier non ?

INSTALLER PYTHON DIRECTEMENT DEPUIS ANACONDA

Si vous vous tournez vers l'analyse de données plutôt que le développement web, autant télécharger directement Anaconda en plus de Python.

Allez donc sur https://www.anaconda.com/download/#macos, si vous scrollez en bas de la page, vous verrez que vous pouvez télécharger le logiciel en Python 3.6 ou Python 2.7[2]. Vous choisirez donc Python 3.6

Le téléchargement d'un fichier *.pkg* va aussi démarrer, cette fois un peu plus lourd que le premier. Une fois que le téléchargement est fait, vous pourrez ouvrir ce fichier *.pkg* et commencer l'installation.

Vérifiez bien que vous avez mis Anaconda dans vos applications pour que vous puissiez le retrouver facilement.

POUR WINDOWS

INSTALLER PYTHON DIRECTEMENT DEPUIS PYTHON.ORG

Allez sur le site officiel de Python : https://www.python.org/getit/, allez dans l'onglet Downloads et cliquez sur Download Python 3.7.0[2]. Une ceci fait, vous pourrez suivre les étapes d'installation classique.

Au moment de l'étape, "Customize Python", vous cocherez simplement "add python.exe to Path" si ceci n'est pas déjà précoché.

INSTALLER PYTHON DIRECTEMENT DEPUIS ANACONDA

Si vous vous lancez dans les Data Sciences, nous vous conseillons de télécharger directement Anaconda, car vous serez amené à l'utiliser plus tard.

2. Ceci sont les versions présentes sur le site au moment de l'écriture de manuel
3. Python 3.7.0 était la dernière version du programme à l'époque de l'écriture de ce manuel

Allez donc sur https://www.anaconda.com/download/#windows et choisissez de télécharger Anaconda sur Python 3.6[4].

Bravo ! Vous êtes maintenant fin prêt à coder. Vous devez surement être très pressé de commencer à pratiquer Python et à mettre les mains dans le cambouis. Vous avez peut-être déjà vu du code d'ailleurs écrit un peu partout mais vous vous êtes souvent demandé : "mais où diable écrivent-ils ce code ? Et d'ailleurs comment il s'exécute ce programme ?" Et bien expliquons cela maintenant.

De manière générale, lorsque vous codez, vous aurez besoin de deux outils :

Un éditeur de texte
Une console

L'ÉDITEUR DE TEXTE

QU'EST CE QUE C'EST ?

L'éditeur de texte est l'endroit où vous allez **écrire votre code.** Ce code prendra ensuite la forme d'un fichier que vous pourrez enregistrer sur votre machine puis exécuter via votre console. Depuis que vous avez ouvert pour la première fois un ordinateur, vous avez croisé déjà des tonnes de fichiers différents. Ils se caractérisent d'ailleurs par des extensions différentes, voici quelques exemples :

```
helloworld.html
Helloworld.json
Helloworld.php
Helloworld.xlsx
Helloworld.docx
Helloworld.js
Helloworld.pdf
helloworld.jpg
```

4. Il se peut que la version de Python ait évolué depuis. Assurez vous simplement de prendre la version de Python 3.X

Ce qui est avant le point correspond au nom de votre fichier et ce qui est après le point correspond à l'extension du fichier. L'extension vous donne des informations sur le code utilisé dans le fichier.Dans notre cas, nous écrirons sur des fichiers .py pour désigner python.
Par exemple :

```
helloworld.py
```

C'est à l'intérieur de ce fichier que vous pourrez écrire vos programmes et les sauvegarder pour les exécuter ensuite.

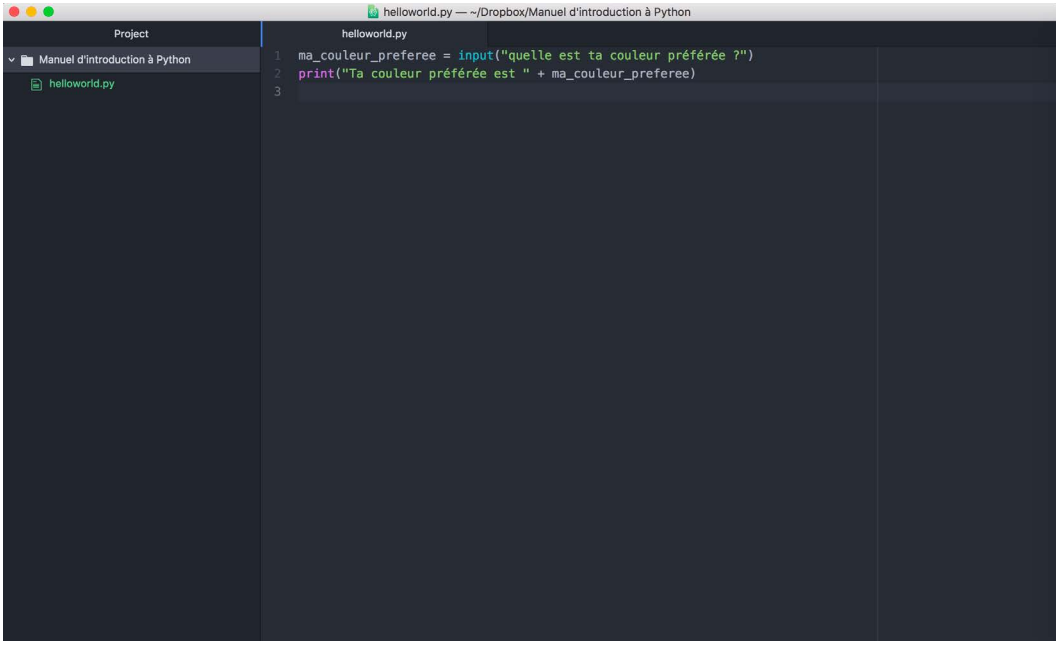
QUEL ÉDITEUR DE TEXTE CHOISIR ?

Techniquement, vous pouvez écrire du code sur n'importe quoi qui vous permet d'écrire des chaînes de caractères et des chiffres. Si cela vous amuse, vous pourriez tout à fait écrire du code sur un document Word ou Google Doc. Cependant, il y a des outils plus adaptés qui vous suggèrent par exemple des fonctions pré-enregistrées, des syntaxes etc.
Parmi les plus populaires on retrouve :

- ***Atom***
- ***Sublime Text***
- ***Support***
- ***TextWrangler***
- ***Texastic***
- ***Notepad++ (pour windows)***
- ***TextEdit (pour Mac)***

NB : Les deux derniers éditeurs ne sont pas forcément les plus adaptés et ni les plus robustes mais vous les croiserez nécessairement puisque ce sont ceux qui sont pré-installés dans les machines Windows et Mac.

Prenons un exemple de ce à quoi ressemble un éditeur de texte.
Nous utilisons Atom pour cette démonstration :



On voit plusieurs choses sur cette image. A gauche, vous pouvez apercevoir *helloworld.py*. Ce fichier est celui dans lequel nous écrivons notre code et sur lequel il sera donc sauvegardé.

A droite, vous pouvez apercevoir un premier programme que nous expliquerons plus tard dans l'ouvrage :

```
ma_couleur_preferee = input("quelle est ta couleur préférée ?")
print("Ta couleur préférée est " + ma_couleur_preferee)
```

Comme vous pouvez le constater, une partie du code est de couleur de différente. C'est l'un des grands avantages à utiliser un éditeur tel que Atom : cela va vous facilite à la fois l'écriture et la lecturee. Eh oui ! En tant que programmeur vous serez aussi souvent amené à lire du code qu'à en écrire !

Maintenant qu'un programme est écrit, ce n'est pas avec votre éditeur de texte que vous allez exécuter ce code mais avec votre ***console.***

LA CONSOLE

QU'EST CE QUE C'EST ?

Tout d'abord, la console porte plusieurs noms : *Terminal, Shell, Command-Prompt*. En fonction du langage de programmation sur lequel elle est basée, de l'ordinateur que vous utilisez etc. Vous adopterez un nom plutôt qu'un autre. Sur Mac par exemple, on appellera notre console le *Terminal* et sur *Windows*, on parlera plutôt de *PowerShell*.

Pour débuter rien ne sert de connaître les différences exactes entre chaque console. Par contre, il est bon de savoir à quoi elle sert.

C'est la console qui va vous permettre ***d'interpréter et d'exécuter le code que vous avez écrit dans votre éditeur de texte***. De manière générale, les consoles sont assez bien cachées dans votre ordinateur pour éviter que des néophytes aillent tout modifier et faire planter leur ordinateur.

Mais, puisque nous sommes des développeurs en herbe, tentons d'ouvrir notre console !

SUR MAC

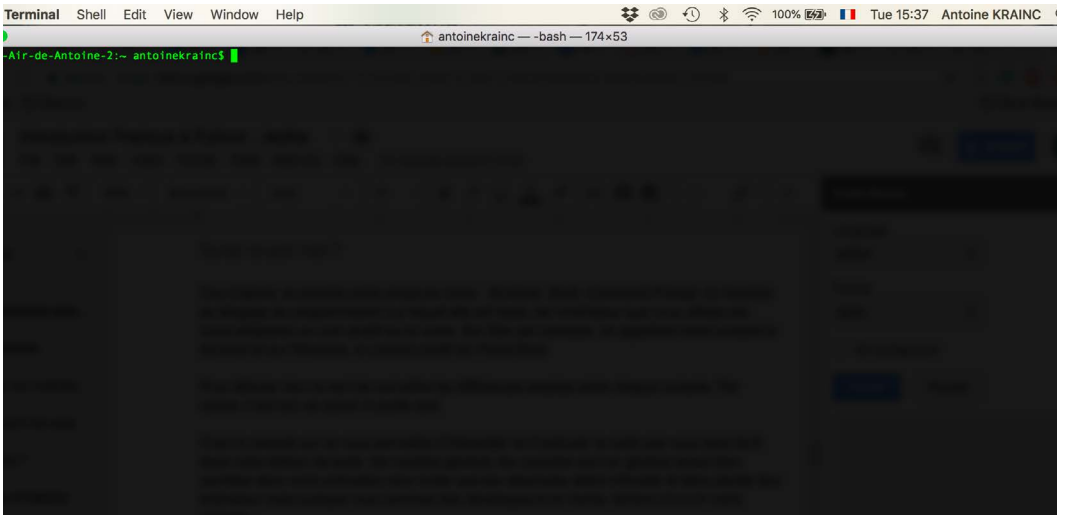
- Aller dans Application
- Aller dans Utilitaire
- Ouvrir Terminal

SUR WINDOWS :

- Ouvrir l'explorateur Windows ou le raccourcis Poste de Travail.
- Aller dans le répertoire c:\windows\system32
- Cliquer sur le fichier cmd.exe avec le bouton droit et choisir "Exécuter en tant qu'administrateur"
- Dans la fenêtre qui s'ouvre, saisir un nom de compte et le mot de passe d'un administrateur.
- Cliquer sur OK

Pour la suite des explications de cette partie, nous utiliserons le terminal MAC mais les commandes seront aussi valables sur Windows.

Voici donc à quoi ressemble votre terminal :



C'est à ce moment-là vous aurez l'impression d'entrer dans Matrix[5].

Tentons de comprendre un peu ce qu'il se passe.

On peut découper la phrase écrite en vert en deux parties : ***MacBook-Air-de-Antoine-2:~ antoinekrainc$***

Vous avez d'abord le nom de votre machine. En l'occurrence, j'ai eu deux Macbook Air et c'est sur le deuxième que je vous montre les exemples de ce livre. D'où le fait que vous voyiez :

MacBook-Air-de-Antoine-2:

La seconde partie désigne l'endroit où vous vous trouvez dans votre ordinateur. De la même manière que pourrait l'être votre Dropbox, Google Drive ou autre, votre ordinateur n'est en fait qu'une montagne de fichiers rangés dans des dossiers très bien organisés. On a fait une interface graphique et des systèmes d'exploitation pour rendre tout cela plus joli. **Mais fondamentalement, c'est ça un ordinateur : des fichiers rangés dans des dossiers.**

De fait, vous pouvez vous balader dans ces différents dossiers par le biais de votre console. En l'occurrence, je suis dans le dossier Utilisateurs de mon Mac nommé

antoinekrainc

Tout ce qui est après le signe *$* contiendra les commandes que vous allez entrer dans la console. Regardons celles qui vont nous être utiles

NB : Sur Windows, votre dossier racine sera C://

LES COMMANDES PRINCIPALES

Dans cette partie, nous allons utiliser les commandes qui nous seront le plus utiles à connaître. Les commandes varient selon que vous soyez sur Mac ou Windows. Les exemples suivants ont été réalisés sur Mac mais nous vous don-

nons les commades équivalentes sur Windows.

$ CD

La commande :

```
$ cd
```

Est l'abréviation de ***c***hange ***d***irectory . C'est la commande qui va vous permettre de naviguer dans vos dossiers. En effet, par défaut votre console vous amènera au dossier racine. Dans l'exemple du dessus, le dossier racine est *antoinekrainc*.
Vous pouvez donc changer de dossier en écrivant la commande *cd* puis le chemin vers lequel vous souhaitez aller. Par exemple si je cherche à aller dans mon dossier downloads, voici ce que je peux faire

```
$ cd Downloads
```

Je suis maintenant dans le dossier *Downloads* de mon ordinateur. Si je souhaitais aller dans un dossier qui est déjà dans le dossier *Downloads*, je peux le faire de la façon suivante :

```
$ cd Downloads/UN_DOSSIER
```

Il suffira en effet d'ajouter un slash (/) pour aller dans un dossier contenu dans un autre dossier.

Si vous êtes allé trop loin dans vos dossiers, il vous suffira de remettre la commande :

```
$ cd
```

Pour revenir à votre dossier racine.

NB : Ici pour Windows, cette commande est la même, donc aucun problème de compatibilité. ATTENTION : Si vous écrivez juste cd, vous obtiendrez le chemin pour accéder au dossier dans lequel vous êtes. Si vous voulez revenir au dossier racine, vous devrez écrire : **cd **

$ LS (OU DIR POUR WINDOWS)

La commande :

```
$ ls
```

Est utile pour regarder le contenu d'un dossier. Par exemple si je retournais dans mon dossier *UN_DOSSIER* et faisais la commande *$ ls* j'obtiendrai :

```
MacBook-Air-de-Antoine-2: UN_DOSSIER antoinekrainc$ ls
>>>> fichier_1.py fichier_2.py fichier_3.py
```

Vous voyez ici que j'ai trois fichiers dans mon dossier UN_DOSSIER qui sont :

fichier_1.py
fichier_2.py
fichier_3.py

NB : L'équivalent de ls pour Windows sera : dir

Bravo, vous connaissez maintenant les commandes principales pour naviguer dans votre ordinateur depuis votre console. Ceci va vous être utile pour la suite. Si vous souhaitez connaître plus de commandes, vous pouvez aller consulter les annexes de ce manuel pour en apprendre plus. Commençons à écrire et exécuter nos premiers programmes en Python !

UN PROGRAMME EN PYTHON

ÉCRIRE UN PREMIER PROGRAMME

Reprenons le programme que nous avions écrit plus haut. Vous pouvez écrire ce code sur l'éditeur de texte de votre choix :

```
ma_couleur_preferee = input("quelle est ta couleur préférée ?")
print("Ta couleur préférée est " + ma_couleur_preferee)
```

Vérifiez bien que vous ayiez bien écrit exactement la même chose que ce le code au dessus. Il se pourrait que vous ayez une erreur dans le cas inverse[6].

Qu'avons-nous donc fait dans ce programme ?

D'abord, nous avons créé une variable que l'on a nommé *ma_couleur_preferee*. Cette variable contient la fonction *input()* qui va demander à l'utilisateur : *quelle est ta couleur préférée ?*. Pour finir, le programme va ressortir une phrase disant : *Ta couleur préférée est*, avec la couleur préférée que l'utilisateur aura renseigné au programme.

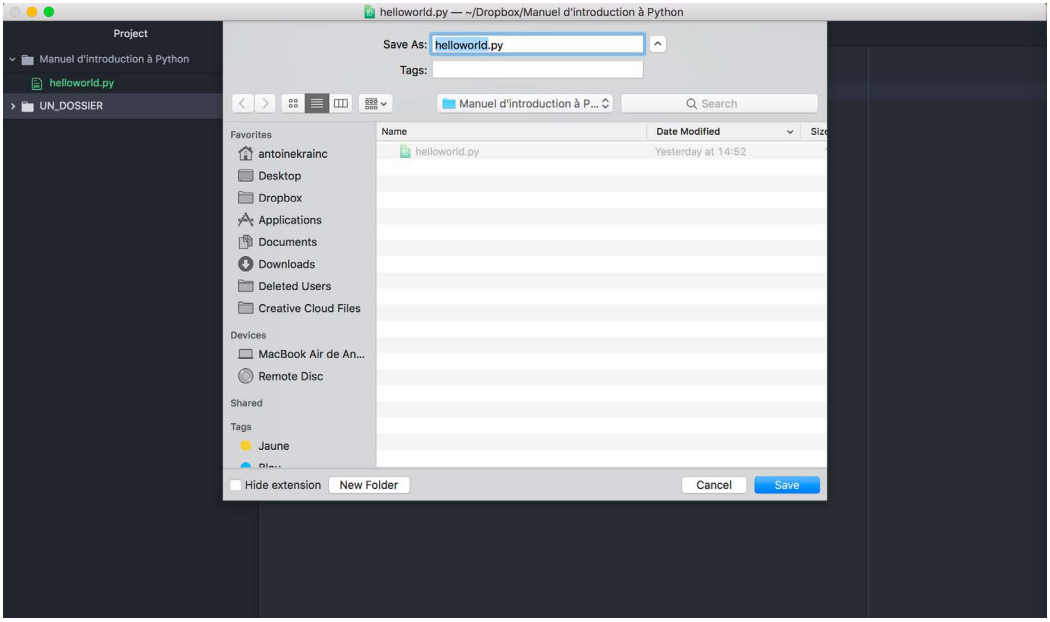
Si vous ne comprenez pas ces deux lignes de code dans le détail, c'est normal !

Nous n'avons pas parlé de beaucoup des concepts sous-jacents nécessaires à la compréhension complète. Le but ici est de simplement de comprendre l'idée générale d'un programme simple. Nous expliquerons plus tard dans chacune des parties ce qu'est une variable, une fonction et une chaîne de caractères.

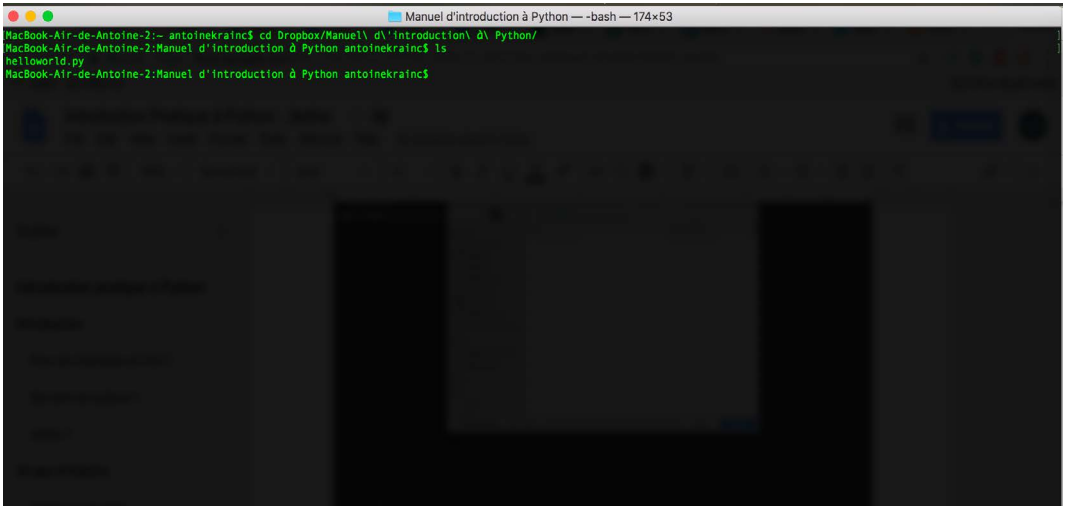
Une fois que vous avez écrit ces deux lignes de codes, vous pouvez sauvegarder votre fichier dans votre ordinateur. Dans notre cas, nous le ferons dans le dossier *Manuel d'introduction à Python* qui lui même se trouve dans une Dropbox.

Nous avons nommé ce fichier *helloworld.py.*

6. Nous parlons des erreurs les plus courantes dans les annexes de ce manuel



EXÉCUTER UN PROGRAMME AVEC LE REPL



Pour exécuter votre code, ouvrez votre console et mettez vous dans le dossier dans lequel réside votre fichier sur lequel vous avez écrit votre programme. Dans notre cas, nous sommes dans antoinekrainc > Dropbox > Manuel d'introduction à Python
La première chose que vous pouvez remarquer dans la console est qu'elle a du mal à gérer les espaces. C'est pour cela que, généralement, on essaie d'éviter les espaces lorsque l'on nomme des fichiers et utilisons plutôt ce caractère _ . Par exemple : UN_DOSSIER

Dans notre cas, les espaces sont remplacés par des *backslashs* : \

NB : Si, comme beaucoup de développeurs, vous êtes quelque peu paresseux, vous n'êtes pas nécessairement obligé d'écrire le nom entier de votre dossier. Vous pouvez en écrire qu'une partie et appuyer sur TAB pour que la console complète automatiquement la fin du nom.

Une fois que vous êtes dans votre dossier, vous pouvez exécuter la commande suivante :

```
$ python helloworld.py
```

NB : vous devrez faire précéder votre fichier de la mention python pour exécuter le fichier. Sinon cela ne va pas marcher.

ATTENTION : Si vous avez téléchargé python 3 pour MAC depuis python.org, vous devrez faire précéder votre fichier de python3 au lieu de python.

Donc comme suit :

```
$ python3 helloworld.py
```

Cette commande va faire exécuter votre code. Vous devriez donc voir apparaître :

```
Quelle est ta couleur préférée ?
```

Vous pourrez directement écrire sur votre console une couleur et la suite de votre programme s'exécutera :

```
Quelle est ta couleur préférée ?Bleu
Ta couleur préférée est Bleu
```

Dans l'exemple du dessus, nous avons donné la couleur bleue. C'est pour cela que le programme nous a retourné :

```
Ta couleur préférée est Bleu
```

Il est intéressant de voir que nous avons inséré *Bleu* avec une majuscule et que le programme l'a ressorti exactement comme nous l'avons inséré. Ceci est typique dans le code. On disait avant d'un ordinateur qu'il était "bête". Il y a du vrai là dedans dans le sens où, votre programme ne va pas essayer d'interpréter ce que vous voulez faire. Il va simplement **exécuter** le code inscrit dans l'éditeur de texte.

Lorsque vous avez exécuté le script sur votre console, vous avez utilisé ce qu'on appelle le *REPL* ou **Read Evaluate Print Loop** ou encore **Python Shell.** Ceci est simplement la console dédiée à Python.
C'est pour cela que nous avons commencé notre commande par :

```
$ python fichier_a_executer.py
```

Avec le **REPL,** vous pouvez exécuter des scripts mais vous pouvez aussi écrire directement du code. Il vous suffira d'exécuter la commande :

```
$ python
```

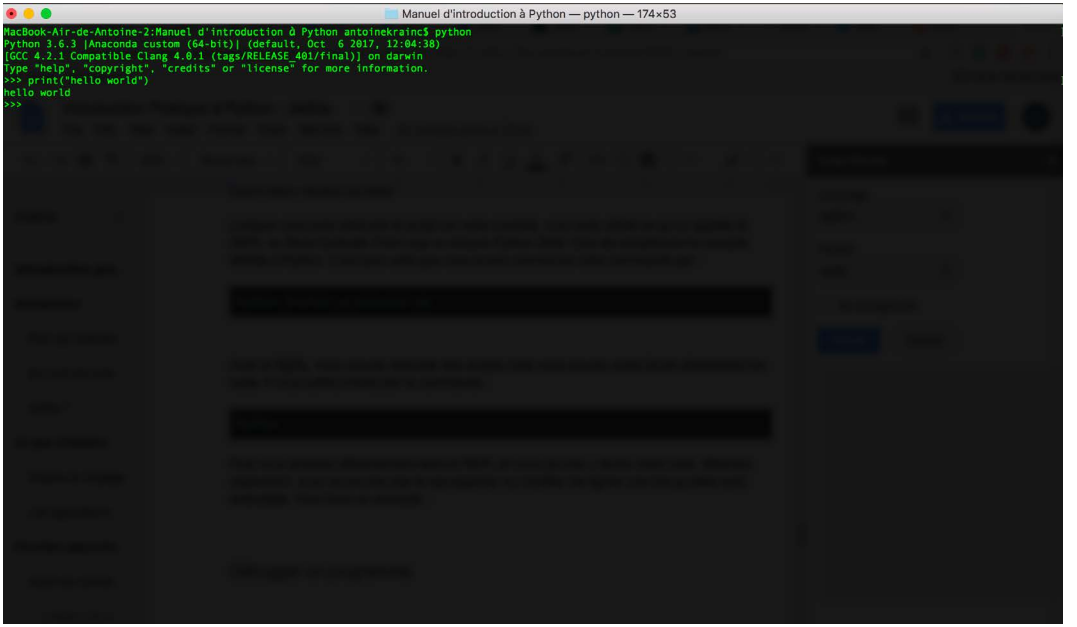
ATTENTION : Si vous avez téléchargé python 3 pour MAC depuis python.org, vous devrez écrire **python3 au lieu de python**. Sinon vous utiliserez la version 2 de python qui est celle utilisée par défaut sur MAC. Cependant, si vous avez téléchargé Anaconda, le logiciel va mettre automatiquement python 3 comme version par défaut donc vous n'aurez pas à écrire python3.

*Souvenez vous de ceci pour la suite des exercices et des explications car vous devrez tout le temps remplacer python par **python3**.*

```
$ python3
```

Une fois la commande exécutée, vous entrerez effectivement dans le REPL et vous pourrez y écrire votre code. Attention cependant, vous ne pourrez pas le sauvegarder ou modifier les lignes une fois qu'elles sont exécutées.

Voici donc un exemple :



Ici, nous avons écrit directement dans le REPL :

```
print("hello world")
```

Puis notre programme s'est exécuté directement.

Le REPL est très adapté pour exécuter du code simple. Par exemple, si vous souhaitez tester un bout de votre programme avant le mettre en production, vous pouvez le faire via cet outil.

Cependant, vous aurez la plupart du temps à écrire plusieurs dizaines de lignes de code dans lesquelles peut notamment se cacher une erreur de syntaxe. Le REPL vous demandera de réécrire tout votre code depuis le départ si vous vous trompez.
C'est pour cela que votre code sera plutôt écrit sur un éditeur de texte. La console sera uniquement utilisée pour exécuter le code qui est écrit sur votre éditeur, qu'on appellera *script.*

POUR ALLER PLUS LOIN : DÉBUGGER UN PROGRAMME

En tant que programmeur, il vous arrivera très souvent (surtout au début de votre carrière) de faire des erreurs.
Il y a plusieurs types d'erreurs, les plus connues sont les erreurs de *syntaxe* (vous écrivez une coquille dans votre programme) mais il y a aussi des erreurs de *noms de fonctions*, de définitions de *variables* etc.

Regardons deux erreurs que nous aurions pu faire dans le programme d'au dessus :

COMMENÇONS PAR UNE ERREUR DE NOM

```
pint("hello world")
```

Comme vous pouvez le remarquer, j'ai oublié un r dans ma fonction *print("hello world")*. Si j'exécute ce programme voici ce que m'indique mon REPL :

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

Même si à première vue, cette réponse peut faire un peu peur, ces trois lignes vous donnent des informations très utiles.

Lorsque votre REPL rencontre une erreur, il fait remonter la première erreur qu'il rencontre. D'où la première ligne:

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

Ensuite, le REPL nous donne des informations sur l'endroit où il a trouvé l'erreur d'où la ligne :

```
File "<stdin>", line 1, in <module>
```

En l'occurrence, l'erreur se trouve au niveau de la ligne 1 du programme.
Enfin, vous avez le message d'erreur qui s'affiche :

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

Ici, le REPL nous dit qu'il ne reconnaît pas la fonction *pint* que nous avons écrite. D'où le fait qu'elle ne soit pas *définie* ou *defined.*

Pour peu que vous écriviez un peu vite ou que vous soyez fatigué, il arrive souvent de faire des coquilles comme celle-ci.

Il se peut aussi que nous fassions de vraies erreurs de syntaxe où l'on ne respecte pas la "grammaire" du langage de programmation. Regardons un exemple :

```
print("hello world')
```

Ici, vous pouvez voir que nous avons mis d'un côté de la chaîne de caractères un *guillemet*, tandis que nous avons mis une *apostrophe* de l'autre côté.
Ces deux caractères sont tous les deux valides pour délimiter une chaîne de caractères mais on doit utiliser soit l'un soit l'autre et non pas les deux. Voici donc la réponse que vous obtenez avec cette erreur:

```
  File "<stdin>", line 1
    print("hello world')
                        ^
SyntaxError: EOL while scanning string literal
```

Cette fois, notre REPL a su détecter l'emplacement exact de notre erreur et nous l'indique d'une flèche. Vous avez un message d'erreur qui en lui-même ne vous donne pas beaucoup d'informations hormis que vous avez fait une erreur de syntaxe.

NB : EOL veut dire End of Line. Cela veut dire que Python est arrivé à la fin de la ligne alors qu'il essayait de lire une chaîne de caractère. Si cela arrive, il l'interprète comme une erreur.

Il y a beaucoup d'erreurs de syntaxe possibles. Nous en avons listé une partie dans les annexes de ce manuel. Cependant, il est tout à fait possible que vous en rencontriez d'autres. De manière générale, essayez de bien lire et de comprendre ce qu'essaie de vous dire votre console, elle vous donnera toujours des indications sur l'erreur que vous avez faite. Si toutefois ces informations ne sont pas utiles, essayez de changer votre code et de le réexécuter pour éventuellement faire apparaître d'autres erreurs et avancer à tâtons jusqu'à les résoudre.

C'est en faisant des erreurs que l'on apprend et c'est aussi vrai dans l'écriture de code. Même si cela peut être (très) frustrant parfois, il faut s'armer de patience et de courage lorsque vous êtes à la chasse d'une erreur dans votre programme.
Enfin dernier conseil dans la gestion d'erreurs, **lisez la documentation**.

Que ce soit en Python ou d'autres langages, il y existe de la documentation sur internet qui peut vous aider.

VOUS POUVEZ ALLER :

- Directement sur la documentation propre au langage avec lequel vous travaillez (ex : Mozilla Developer Network si vous cherchez des informations sur du HTML, CSS, JAVASCRIPT ou encore python.org pour tout ce qui concerne Python)

- Stack Overflow est un forum d'entraide où les développeurs postent des erreurs qu'ils ont faites pendant que d'autre les résolvent

- Tapez directement votre erreur dans Google. Vous serez redirigé vers des sites qui expliquent cette erreur si vous n'avez pas déjà trouvé votre bonheur avec les deux points du dessus

Bravo !
Vous pouvez déjà être fier de vous puisque vous avez écrit et débuggé votre premier programme en Python. C'est une bonne première étape de franchie. Nous allons maintenant gagner en profondeur et expliquer point par point les concepts à connaître lorsque l'on code en Python.

LES VARIABLES

QU'EST CE QU'UNE VARIABLE ?

Lorsque l'on code, il nous arrive très souvent de vouloir stocker des valeurs quelque part pour pouvoir lesréutiliser ensuite. Prenons un exemple simple, nous allons dans le REPL, nous pouvons écrire :

```
3+5
8
```

Mais nous pouvons aussi définir une variable, que nous appelerons *somme,* à laquelle nous faisons correspondre la somme de 3 et de 5 :

```
Somme = 3+5
print(Somme)
```

Nous obtenons alors le même résultat :

```
8
```

Mais alors à quoi cela sert-il de stocker 3+5 dans une variable que l'on appelle *somme* ?
La réponse est simple, si vous êtes par exemple amené à faire plusieurs fois la même opération dans votre code, au lieu de devoir la réécrire à chaque fois, vous pouvez simplement inscire votre variable.

Pour mieux comprendre, voici deux cas :

Cas n°1

```
3+5
print(3+5)
(3+5) - 2

>>>> 6
```

Cas n°2

```
somme = 3+5
print(somme)
somme - 2

>>>> 6
```

Dans le ***cas n°1***, si nous devions changer d'opération et faire par exemple : 3-5, nous devrions changer trois lignes de code :

```
3-5
print(3-5)
(3-5) - 2

>>>> -4
```

Dans le ***cas n°2***, nous ne devrions changer qu'une seule ligne :

```
somme = 3-5
print(somme)
somme - 2
>>>> -4
```

Sur trois lignes, le gain de temps est minime entre le cas n°1 et le cas n°2 mais imaginez sur 50 lignes.

De manière générale, on utilise donc très souvent les variables pour **"stocker"** une valeur afin de la réutiliser plus tard dans d'autres opérations.

La deuxième raison est que la valeur d'une variable peut être amenée à changer. C'est vous qui définissez votre variable. Si vous écrivez somme = 0; vous donnez la valeur 0 à votre variable somme. 0 reste la valeur de cette variable sauf si vous lui assignez une autre valeur, exemple 40. Comme dans l'exemple suivant :

```
somme = 0
print(somme)
somme = 40
print(somme)
```

Dans cet exemple, vous obtiendrez les résultats suivants :

```
0
40
```

C'est à dire que la première fonction *print()* a retourné la valeur 0 puisque la variable somme était paramétrée à 0 puis nous avons re-paramétré la variable à 40 et c'est pour cela que notre fonction *print()* a cette fois retourné 40.

EXERCICE : UN NOMBRE CHOUETTE

Créez une variable "pi" où vous assignerez la valeur 1.43

Mince ! Nous avons inversé des chiffres pour Pi, réassignez la vraie valeur de pi soit 3.14

SOLUTION

OPÉRATIONS ENTRE VARIABLES

Vous pouvez tout à fait **combiner** les variables comme dans l'exemple qui suit :

```
a = 1
b = 3

c = a + b

print(c)
```

Vous obtenez alors le résultat suivant :

```
4
```

C'est ici que cela devient très intéressant: Nous pouvons désormais changer la valeur de *c* simplement en changeant les valeurs pour *a* et *b*. Par exemple :

```
a = 20
b = 3

c = a + b

print(c)
```

Nous obtenons alors :

```
23
```

EXERCICE : L'ARBRE AU FRUIT MAGIQUE

Nous avons un arbre magique caché quelque part en France qui produit des fruits aux
propriétés incroyables (si vous trouvez l'arbre, vous connaîtrez ces priorités ;)) :
L'arbre produit 3 fruits par mois, créez une variable "fruits" et stockez cette valeur
Un enfant a trouvé l'arbre et a même mangé un fruit !
Créez une variable "fruit_mangé" et stockez cette valeur
Le jardinier qui s'occupe de cet arbre veut savoir combien de fruits il lui restera à la fin du mois.
Créez une troisième variable "fruits_restant" qui va permettre de calculer la valeur restante de fruits dans l'arbre

SOLUTION

DES CHIFFRES ; MAIS AUSSI DU TEXTE

Dans une variable, vous n'êtes pas obligé d'y insérer *uniquement des chiffres.* Vous pouvez tout aussi bien y stocker du texte.

Par exemple :

```
bonjour = "Bonjour, je m'appelle "
```

Cette fois, nous avons créé une variable que nommée *bonjour* dans laquelle il y a la chaîne de caractères : "Bonjour, je m'appelle "

Il est en effet tout à fait possible de stocker une grande variété de données dans une variable. Nous verrons dans la partie suivante les différents types de données en Python.

EXERCICE : NOTRE CHIEN PRÉFÉRÉ

Créez une variable nommée "chien" dans laquelle vous entrerez la valeur "berger allemand"
Réassignez une valeur pour "chien" à une autre race de chien
Si nous utilisons la fonction print() sur la variable chien, quel sera le résultat ?

SOLUTION

EST IL POSSIBLE DE CRÉER DES CONSTANTES ?

En Python, non.
Dans d'autres langages, vous verrez la possibilité de créer des constantes. Cela fonctionne de la même manière qu'une variable à la seule différence qu'une constante n'est pas vouée à changer de valeur au cours du programme.
L'utilisation des constantes étant tellement rare que les développeurs Python ont préféré les omettre.

POUR ALLER PLUS LOIN : CONCATÉNER UNE VARIABLE AVEC UNE CHAÎNE DE CARACTÈRES

Regardons cet exemple :

```
bonjour = "Bonjour, je m'appelle "
antoine = "Antoine"
```

```
print(bonjour + antoine)
```

Nous avons concaténé une chaîne de caractère via la variable *bonjour* et la variable *antoine* grâce au signe opératoire +.

Nous obtenons le résultat :

```
Bonjour, je m'appelle Antoine
```

Vous remarquerez que nous n'avons pas mis de guillemets (" ") autour du nom des variables. Si nous les avions mis alors la fonction print aurait considéré les deux mots comme **une chaîne de caractères et non une variable.** Nous aurions alors obtenu le résultat :

```
print("bonjour" + "antoine")
bonjour antoine
```

EXERCICE : UN ORDINATEUR POLI

Créez une variable "merci" contenant la chaîne de caractère "Merci beaucoup"
Créez une variable "prénom" contenant votre prénom
Demandez à votre console de vous remercier

SOLUTION

PROJET

COMPLIMENT DU JOUR

Nous avons déjà quelques connaissances avec Python que nous allons pouvoir mettre à profit.

Nous voulons que notre console nous fasse un compliment. Ce ne sera pour l'instant qu'un seul compliment.

Créez donc un programme où la console demandera votre prénom puis dira "Vous êtes rayonnant aujourd'hui prénom"

Indice : explorez la fonction *input(),* que nous avons utilisé au début de ce manuel

SOLUTION

LES DIFFÉRENTS TYPES DE DONNÉES EN PYTHON

COMME TOUS LES LANGAGES DE PROGRAMMATION, PYTHON GÈRE PLUSIEURS "TYPES" DE DONNÉES. NOUS AVONS COMMENCÉ À EN VOIR DANS LES DIFFÉRENTS EXEMPLES CITÉS AU DESSUS. MAIS IL Y EN A BEAUCOUP PLUS QUE CELA.

BIEN QU'INUTILE DE CONNAÎTRE CHACUN DES TYPES DE DONNÉES PAR COEUR, IL EST BON DE COMPRENDRE CE À QUOI ILS SERVENT CAR VOUS POURREZ ENSUITE COMPRENDRE QUELLES SONT LES DIFFÉRENTES OPÉRATIONS LOGIQUES POSSIBLES À RÉALISER.

DONNÉES NUMÉRIQUES

INTEGER

Ou Int pour faire plus court correspond à un nombre entier, positifs et négatifs. Exemple :

```
A = 2
B = 3058
C = 2 147 483 647
```

Sont trois variables de type int

NB : Dans Python 2, les int ne vont pas au delà de 2 147 483 647 dans les positifs et - 2 147 483 647 dans les négatifs. Si toutefois vous auriez besoin de d'aller au delà, vous seriez dans des données de types long. Ceci ne s'applique cependant plus avec Python 3.

FLOAT

Les données de type float sont des nombres décimaux, positifs ou négatifs.

Exemple :

```
A = 2.56
B = 1.05
C = 1.00
```

A noter que puisque les langages ont été développés sur la base anglo saxonne, les séparateurs de décimaux ne sont pas exprimés avec une virgule mais avec un point.

NB : Nous aurions pu écrire C = 1 ou C = 1.00 les deux fonctionnent. Cependant, dans le premier cas, C est une variable de type int *et dans le second cas, la variable C est de type* float*.*

Si vous faites une opération entre une variable de type *int* et une variable de type *float* vous obtiendrez un nombre de type float. Par exemple :

```
A = 1
B = 2.00

print(a+b)

>>>> 3.0
```

Ceci est parce que la donnée de type float garde toute l'information contenue dans les deux variables.

EXERCICE : C'EST PAS LE NOMBRE DE DÉCIMALES QUI COMPTE

Créez une variable int qui prend le nombre 2 001 291
Créez une variable float qui prend la valeur 3.14
Ajoutez les deux variables, quel est le type de données que vous obtenez ?

SOLUTION

DONNÉES TEXTES

STRING

Ou *str* dans le code. Cela correspond à des données texte comme :

```
b = "bonjour"
```

NB : Dans une chaîne de caractère, l'espace (' ') compte comme un caractère.

EXERCICE : DES CARACTÈRES DE NOMBRE

Créez un variable "ecole" qui prendra la valeur 42
Quelle est le type de donnée de la variable "ecole"
Utilisez la fonction str() sur la variable école, vous stockerez cette valeur dans une nouvelle variable qu'on appellera "str"
Quel est le type de donnée de "str"

SOLUTION

COLLECTION DE DONNÉES

Les collections de données sont des types de donnés sur lesquelles vous pouvez itérer. Elles sont très souvent utilisés pour créer des matrices, en Data Science notamment. Nous allons lister ici les plus courantes pour que vous puissiez vous y référer plus tard.

NB : Nous parlerons plus en détail des collections de données comme les dictionnaires, les tuples, listes et ensembles dans une autre partie du manuel pour que vous puissiez bien comprendre la différence.

TUPLE

En Français *N-uplet* est une collection de données ***hétérogène*** et ***immuable.***

Voici un exemple :

```
Tuple = (10, 20, "ceci est un n-uplet'", 3.14)
```

Vous pouvez accéder à chaque item de votre tuple de la façon suivante :

```
print(Tuple[0])
>>>>> 10
```

Ceci nous donnera donc le résultat **10**

NB : Attention, en informatique tout commence à 0 et non à 1. C'est pour cela que le premier item de notre tuple est à l'index 0.

LISTE

Une liste ressemble très fortement à un tuple à la différence qu'une liste est ***muable*** (donc changeable). Voici un exemple :

```
List = [1, 3, 10, "Ceci est une liste", 2.1095]
```

De la même manière qu'un tuple, vous pouvez accéder à chaque item de votre liste par en indiquant l'index de l'item que vous souhaitez sélectionner entre crochets :

```
print(List[1])
>>>> 3
```

ATTENTION : Comme nous vous le disions plus haut, la différence entre un tuple et une liste est que **le premier n'est pas modifiable** alors que le second l'est.

Voici un exemple :

```
Tuple[0] = 230
>>>> Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Alors que pour une liste :

```
List[0] = 230
print(List[0])
>>>> 230
```

Alors que le premier item de notre variable avait initialement la valeur 1, nous avons pu la modifier par la valeur 230. Ce que nous ne pouvons pas faire avec un tuple.

SET

Appelé aussi Ensemble en français, *set* ressemble fortement à une liste à ceci près que vous ne pouvez pas avoir de doublons dans un set et que la collection des données est non-ordonnée.

```
Set = set([1, 2, 3, 'hello world', (4,2,4), 'coder c'est chouette'])
```

Si vous ajoutez un item qui est déjà présent dans un ensemble, cela ne va pas l'ajouter en double :

```
Set.add(1)
Set
>>>> {1, 2, 3, 'hello world', 'coder c'est chouette', (4, 2, 4)}
```

NB : À première vue, la différence entre une liste et un ensemble peut paraître un peu floue, mais, retenez que, du fait qu'un ensemble n'est pas ordonné, vous pourrez plus rapidement retrouver un item dans un ensemble que dans une liste. De fait, si vous avez beaucoup d'items dans une collection, privilégiez plutôt le set qu'une liste.

DICTIONNAIRES

Enfin, un dictionnaire permet d'associer une valeur à une clé définie.

Voici un exemple :

```
dic = {"prénom": "Michel", "nom": "Delpeche"  }
```

L'avantage est que vous pouvez accéder à la clé ou aux valeurs correspondantes à cette clé.

```
dic.keys()
>>>> dict_keys(['prénom', 'nom'])

dic.values()
>>>> dict_values(['Michel', 'Delpeche'])

dic["prénom"]
>>>> 'Michel'
```

EXERCICE : COLLECTER DES DONNÉES

Essayons de voir quel type de collection de données il est préférable de choisir en fonction de la situation.

1. *Nous avons une application web qui est un site web. Dans ce site, nous demandons à l'utilisateur de donner ses coordonnées pour que nous puissions le rappeler. Ces coordonnées sont : Nom, Prénom, Email, Téléphone. Il arrive souvent que nous ayons à re-modifier les données entrées par l'utilisateur car il se peut qu'il / elle fasse une coquille. Nous souhaitons stocker ces données dans une des collections de données, à votre avis quel le meilleur type possible ?*

2. *Nous avons un fichier excel à analyser, puisque vous êtes un expert en Python, vous préférez faire vos analyses avec ce langage directement plutôt que sur excel. Vous décidez donc d'exporter chaque colonne de votre excel dans une*

collection de données. A votre avis, quel serait le meilleur type possible ?

3. *Nous avons une collection de numéros d'identification correspondant à une catégorie de produits. Cette catégorie doit être unique sinon nous pourrions avoir des problèmes de doublons que nous ne voulons pas gérer. A votre avis dans quelle collection devrions nous stocker ces données ?*

SOLUTION

AUTRES TYPES IMPORTANTS DE DONNÉES

BOOLÉEN

Le données de types Booléennes ne peuvent prendre que la valeur True ou False.
Par exemple:

```
45 < 50
>>>> True
```

A l'inverse

```
45 > 50
>>>> False
```

Même si cela comporte peu d'intérêt, on pourrait très bien stocker cette valeur dans une variable

```
bool = 45 > 50
print(bool)
>>>> False
```

Les valeurs booléennes nous seront plus utiles lorsque nous les combinerons

avec des conditions :

```
if variable_1 > variable_2:
      print("la variable_1 est plus grande que la variable_2")
else:
      print("la variable_2 est plus grande que la variable_1")
```

Ceci est un exemple d'une application concrète des valeurs booléennes ; nous aborderons en détail l'utilisation des conditions plus loin dans le manuel.

NONE

None représente le vide. Cela peut arriver très souvent lorsque vous avez des bases de données avec des cases vides dedans. On peut le représenter de la manière suivante :

```
rien = None
print(rien)
>>>>  None
```

Cela couvre tous les types de données que vous trouverez le plus souvent dans votre nouvelle carrière de codeur. Maintenant, ce qui sera utile de savoir est comment faire des opérations avec toutes ces données. Voyons donc les différents opérateurs possibles que nous offre Python.

EXERCICE : VIDE ET RIEN C'EST PAREIL ?

Tentons de voir si 0 et None sont la même chose. Créez une variable "nul" égale à 0
Créez une variable "vide" égale à None
Si vous écrivrez "nul == vide" qu'obtenez vous ?
Que peut-on en conclure ?

SOLUTION

PROJET

DES DONNÉES, DES DONNÉES ET ENCORE DES DONNÉES

Nous voudrions connaître le type de n'importe quelle donnée que l'utilisateur pourrait entrer

INDICE : On utilisera la fonction *input()* puis *type()*

Pouvez-vous voir quelque chose qui cloche dans notre programme ?

SOLUTION

LES OPÉRATEURS EN PYTHON

POUR LES VALEURS NUMÉRIQUES

OPÉRATIONS CLASSIQUES

Comme sur une calculatrice, vous pouvez faire toutes les opérations possibles que vous faisiez dans vos cours de mathématiques avec Python

```
3 + 5
>>>> 8

3 - 5
>>>> - 2

3/5
>>>> 0.6

3 * 5
>>>> 15
```

Vous pouvez faire aussi des opérations de puissance de la manière suivante

```
3 ** 5
>>>> 243
```

Vous pouvez faire aussi des divisions entières qui nous donnent uniquement un nombre entier comme résultat

```
12 // 5
>>>> 2
```

Si nous avions fait une division décimale

```
12 / 5
>>>> 2.4
```

Nous pouvons aussi obtenir le reste d'une division avec la fonction *modulo* représentée par le signe %.

```
10 % 3
>>>> 1
```

OPÉRATIONS PUIS AFFECTATION

Lorsque vous ferez des calculs sur des variables, il se peut que vous souhaitiez incrémenter ces dernières par une certaine valeur.
Par exemple :

```
X = 0
X = X + 1
print(X)
>>>> 2
```

Une manière plus élégante d'arriver au même résultat est de faire :

```
X = 0
X += 1
print(X)
>>>> 2
```

Même si les additions sont les plus courantes, vous pouvez faire la même chose avec les autres opérateurs classiques:

```
X = 1
X -= 3
print(X)
>>>> -2

X = 1
X *= 3
print(X)
>>>> 3
```

```
X = 1
X /= 3
print(X)
>>>> 0.3333333333333333
```

EXERCICE : L'ESTHÉTISME NE TIENT QU'À UN NOMBRE

On commence à pouvoir faire de belles choses avec Python, commençons à utiliser tous les signes opératoires que nous avons vus.

1. Calculez ½ et stockez cette valeur dans une variable qu'on appellera "x"
2. Stocker une valeur y qui sera égale à 1
3. Plus difficile maintenant, calculez la racine carrée de 5 et stockez cette valeur dans une variable "z" (INDICE : Une racine carrée, est l'équivalent d'une puissance ½)
4. Calculez x * (y+z). Quel nombre obtenez vous ?

SOLUTION

COMPARAISONS

Vous pouvez aussi comparer des nombres avec les opérateurs suivants :

COMPARATEURS STRICTS

Les comparateurs stricts n'incluent pas les valeurs égales dans la comparaison :

```
10 > 5
>>>> True

10 > 10
>>>> False
```

```
5 < 10
>>>> True

5 < 5
>>>> False
```

COMPARATEURS NON STRICTS

Les comparateurs non stricts incluent les valeurs égales dans la comparaison:

```
10 >= 5
>>>> True

10 >= 10
>>>> True

5 <= 10
>>>> True

5 <= 5
>>>> True
```

En fonction de votre problématique, vous définirez de quels opérateurs vous aurez besoin d'utiliser.

ÉGAL OU DIFFÉRENT

Vous pouvez aussi vouloir voir si des valeurs sont égales ou différentes.
Voici comment nous procédons:

```
5 == 5
>>>> True

4 != 5
>>>> True
```

```
5 == 6
>>>> False

5 != 5
>>>> False
```

Le signe différent se représente donc par le signe **!=** alors que le comparateur d'égalité se fait par le signe **==**

NB : Faites attention, en code, il y a une différence entre = et == . Dans le premier cas, on assigne une valeur à une variable comme par exemple :

```
X = 5
```

Dans ce cas de figure, nous avons assigné la valeur 5 à X. Si je veux maintenant comparer X, il faut que j'utilise ==

```
X == 7
>>>> False
```

Si, nous avions écrit :

```
X = 7
```

Alors nous aurions réassigné la valeur 7 à X et donc

```
X == 7
>>>> True
```

Il est donc bien important de comprendre la différence entre assigner une valeur à une variable avec le signe = et comparer la valeur d'une variable avec le signe ==.

EXERCICE : LES DALTONS

Les braquages de Bob, Grat, Bill et Emmett Daltons sévissent dans l'ouest américains des années 1890. Mais les frères s'écharpent sur un problème beaucoup plus important ! Qui est le plus grand des Daltons ?

1. Ecrivez un programme qui permette de demander à Bill et Grat de donner leur taille
2. Donnez des valeurs pour la taille de Bill et de Grat
3. Est ce que Bill est plus grand que Grat (montrez le de manière programmatique)
4. Bob et Emmett disent qu'ils font respectivement 10cm de plus que Bill et Grat. Créez un calcul qui vous permette de trouver la taille de Bob et Emmett. Voyez-vous un problème arriver ?
5. Utilisez la fonction int() pour transformer les variables Bill et Grat en valeurs numériques
6. Calculez maintenant la taille de Bob et Emmett
7. Est ce que la taille de Bob est différente de cette d'Emmett ? (montrez le de manière programmatique)

SOLUTION

AUTRES OPÉRATEURS BOOLÉENS

De la même manière que lorsque nous avons présenté les types de données, les opérateurs booléens peuvent prendre uniquement des valeurs *True* ou *False*. Voici les plus courants :

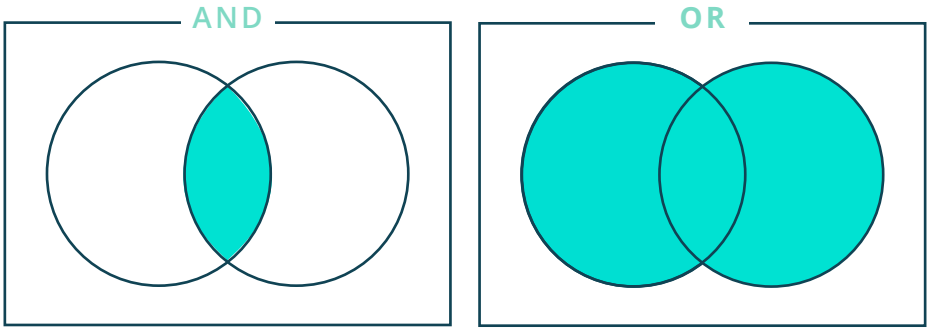
AND / OR

Les opérateurs AND et OR servent le plus souvent à combiner des conditions. On peut faire :

```
X = 5
Y = 8
Z = 10

X < Y and X < Z
```

```
>>>> True

Y < X or Y < Z
>>>> True

Y < X and Y < Z
>>>> False
```

Comme vous pouvez le voir, il y a une différence majeure entre l'opérateur AND et l'opérateur OR.
Le premier nécessite que les deux conditions soient vraies pour être vraies tandis que l'autre ne nécessite qu'une seule des conditions vraie pour être vrai. Voici un schéma qui explique le tout :



NOT

Not correspond à une négation

```
X = 5
Y = not X
not X == y
>>>> True
```

IS

Is sert de test d'identification et fonctionne de la même manière que ==. Par exemple

```
X = 5
X is 5
>>>> True
```

IN$

Enfin In sert de testeur d'appartenance :

```
A = [1, 2, 3, 4]
4 in A
>>>> True
```

EXERCICE : LUCKY LUKE EST PERDU

Lucky Luke n'est pas bon en orthographe. De fait, il a du mal à écrire le prénom de chacun des daltons. Aidons le

1. Est ce que la lettre "o" est dans le prénom "bob" ?
2. Est ce que la lettre "b" est dans le prénom "bob" et "bill" ?
3. Est ce qu'il y a un a dans "grat" ou "emmett"

SOLUTION

DES SPÉCIFICITÉS POUR LES VALEURS TEXTE

Les valeurs de type texte fonctionnent de manière un peu différente que les valeurs numériques (on pouvait s'en douter). Voici les principaux opérateurs qu'on peut utiliser :

LES OPÉRATEURS CLASSIQUES

Voici des opérations que vous serez assez souvent amené à faire lorsque vous gérerez des chaînes de caractères:

```
t = "hello"
u = " world"

t + u
>>>> 'hello world'

t * 4
>>>> 'hellohellohellohello'
```

De la même manière que les opérateurs numériques, nous pouvons de manière élégante concaténer et affecter une valeur à une variable de la manière suivante :

```
t  += " you"
t
>>>> 'hello you'
```

ACCÉDER À DES CARACTÈRES DANS LA CHAÎNE

On peut aussi vouloir accéder à un caractère spécifique dans la chaîne de caractère. Nous le faisons de la même manière que lorsqu'on souhaite accéder à un item d'une liste :

```
t[0]
>>>> 'h'
```

Pour atteindre plusieurs caractères, nous pouvons créer un intervalle de la façon suivante :

```
t[0:3]
>>>>> 'hel'
```

```
T[5:8]
>>>> ' yo'
```

NB : Notez que les espaces comptent comme un caractère dans une chaîne. De fait t[5] vaut ' '

TEST D'APPARTENANCE

Enfin, on peut tester si une chaîne de caractère est comprise dans une autre chaine de caractère avec in :

```
"hello" in t
>>>> True

' ' in t
>>>> True

'H' in t
>>>> False
```

Nous pouvons aussi faire le test inverse avec *not in* :

```
"Michel" not in t
>>>> True

"hello" not in t
>>>> False
```

ALLER À LA LIGNE

Il n'est pas rare d'avoir besoin d'aller à la ligne, dans un texte par exemple. Cela se symbolise par *\n*.

Voici un exemple :

```
t = "hello \nworld"
print(t)
>>>>
hello
world
```

NB : Attention, pour voir vraiment votre chaîne de caractère aller à la ligne vous devrez utiliser la fonction print. Si vous appelez la variable t simplement, vous obtiendrez le résultat suivant :

```
t = "hello \nworld"
t
>>>> 'hello \nworld'
```

TABULATION

De la même manière que le retour à la ligne, on peut faire un tabulation par le caractère \t :

```
t = 'hello \tworld'
print(t)
>>>> hello  world
```

Et comment je fais si j'ai besoin de faire un backslash ?

Bonne question ! Il suffit d'écrire \\ :

```
t = "hello \\ world"
print(t)
>>>>'hello \ world'
```

QUELQUES FONCTIONS UTILES

Il y a des fonctions pour qui peuvent vous aider grandement dans la gestion des chaînes de caractères. Il est donc bon d'avoir les principales en tête :

.upper()

Cette fonction permet de mettre tous les caractères de la chaîne en majuscule

```
t = "hello world"
t.upper()
>>>> 'HELLO WORLD'
```

.lower()

La même fonction existe pour mettre tous les caractères d'une liste en minuscule. Cette fonction est très utilisée pour normaliser les chaînes de caractères et simplifier les comparaisons.

```
t = "HeLlO WorLd"
t.lower()
>>>>'hello world'
```

str()

Cette fonction permet de convertir n'importe quel type de donnée en une donnée de type texte.

```
str(345)
>>>> '345'
```

NB : Ce type de fonction existe généralement pour les autres types de variables

```
int(34.506)
>>>> 34
```

```
float(34)
>>>> 34.0
```

.title()

La fonction *title()* va vous permettre de capitaliser tous les mots de votre titre.

```
t = "ceci est un titre"
t.title()
>>>> 'Ceci Est Un Titre'
```

.capitalize()

A la différence de *title(), capitalize()* va mettre une majuscule uniquement sur le premier mot de la chaîne de caractères.

```
t = "voici un autre titre. il est un peu plus long que le premier titre"
t.capitalize()
>>>> 'Voici un autre titre. il est un peu plus long que le premier titre'
```

.split()

La fonction *split()* va permettre de séparer votre chaîne de caractère en plusieurs parties en fonction d'un séparateur que vous définissez en tant que paramètre.

```
t = "j'aime les pâtes, les lasagnes, les spaghetti, le jambon"
t.split(",")
>>>> ["j'aime les pâtes", ' les lasagnes', ' les spaghetti', ' le jambon']
```

Dans cet exemple, nous avons utilisé la virgule comme séparateur. On aurait pu tout à fait utiliser un autre séparateur.

```
t = "hello world"
t.split(" ")
>>>> ['hello', 'world']
```

.find()

La fonction *find()* permet de retrouver une chaîne de caractère à l'intérieur d'un chaîne de caractère. La fonction donnera l'index du premier caractère contenu dans le mot.

```
t = "hello world"
t.find("world")
>>>> 6
```

NB : Attention, s'il y a plusieurs fois le même caractère dans la chaîne, la fonction retourne l'indice où la caractère apparaît le premier.

```
t = "hello world"
t.find("o")
>>>> 4
```

.join()

Il peut vous arriver d'avoir des chaînes de caractères contenus dans une liste. Vous pouvez les joindre pour en faire une phrase avec la fonction *join()*. Cette fonction fonctionne de manière inverse à *split()*. Vous devrez définir un séparateur puis joindre par ce séparateur :

```
separateur = " "
t = ["voici", "une", "liste", "de", "caractères"]
separateur.join(t)
>>>> voici une liste de caractères
```

De la même manière, on aurait pu choisir n'importe quel autre séparateur

```
separateur = ","
t = ["voici", "une", "liste", "de", "caractères"]
separateur.join(t)
>>>> voici,une,liste,de,caractères
```

.count()

La fonction *count()* permet de compter le nombre de caractères présents dans une une chaîne de caractères.

```
t = "hello world"
t.count("o")
>>>> 2
```

NB : count() permet de connaitre le nombre de fois qu'un caractère apparaît. Mais si vous souhaitez connaître le nombre total de caractères présents dans une chaîne, vous utiliserez plutôt la fonction len()

```
len(t)
>>>> 11
```

.format()

Cette fonction est extrêmement pratique puisqu'elle vous permet d'insérer des valeurs dans une chaîne de caractères.

```
t = "bonjour {}"
t.format("Michel")
>>>> bonjour Michel
```

Vous pouvez insérer plusieurs chaînes de cette manière :

```
t = "bonjour, je m'appelle {} et ma couleur préférée est {}"
t.format("Michel", "le vert")
>>>> bonjour, je m'appelle Michel et ma couleur préférée est le vert
```

help()

Pour finir, dans votre REPL, vous avez une fonction *help()* qui vous ouvrira une session d'aide et pourra vous donner de la documentation relative à ce que vous cherchez

```
help(str)

 |
 |  translate(...)
 |      S.translate(table) -> str
 |
 |      Return a copy of the string S in which each character has been
mapped
 |      through the given translation table. The table must implement
 |      lookup/indexing via __getitem__, for instance a dictionary or list,
 |      mapping Unicode ordinals to Unicode ordinals, strings, or None. If
 |      this operation raises LookupError, the character is left untouched.
 |      Characters mapped to None are deleted.
 |
 |  upper(...)
 |      S.upper() -> str
 |
 |      Return a copy of S converted to uppercase.
 |
 |  zfill(...)
 |      S.zfill(width) -> str
 |
 |      Pad a numeric string S with zeros on the left, to fill a field
 |      of the specified width. The string S is never truncated.
 |
 |  ----------------------------------------------------------------------
```

```
 |  Static methods defined here:
 |
 |  maketrans(x, y=None, z=None, /)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary mapping
Unicode
 |      ordinals (integers) or characters to Unicode ordinals, strings or
None.
 |      Character keys will be then converted to ordinals.
 |      If there are two arguments, they must be strings of equal length,
and
 |      in the resulting dictionary, each character in x will be mapped to
the
 |      character at the same position in y. If there is a third argument,
it
 |      must be a string, whose characters will be mapped to None in the
result.
```

EXERCICE : UN PEU DE POÉSIE

Quelqu'un a écrit un très joli Haiku :

"Au bout de sa langue
Il cache des paysages -
L'étranger."

1. Créez une première variable *"vers1"* contenant le premier vers. Mettez à la fin un caractère qui permette d'aller à la ligne
2. Soyez certain que la première lettre soit capitalisée
3. Faites de même avec le deuxième et troisième vers. On assignera les valeurs respectivement à la variable "*vers2*" et "*vers3*"
4. Joignez *vers1*, *vers2* et *vers3* dans une seule variable qu'on appellera haiku

SOLUTION

POUR ALLER PLUS LOIN : FORMATER DES NOMBRES DANS UNE CHAÎNE DE CARACTÈRES

Il n'est pas rare de vouloir formater les nombres à l'intérieur d'une chaîne de caractères pour qu'ils apparaissent de manière plus élégante. Cela peut extrêmement nécessaire notamment lorsque vous faites des opérations qui résultent sur des nombres décimaux très longs

Pour formater un nombre, on utilisera le symbole **%**

```
X = 5.29320190958
print("si on veut que deux chiffres après la virgule, on peut écrire de la
manière suivante %3.2f" % x)

>>>> si on veut que deux chiffres après la virgule, on peut écrire de la
manière suivante 5.29
```

Dans cet exemple, nous avons décidé d'avoir 3 chiffres au total et 2 chiffres après la virgule pour la variable x. Ceci est représenté dans la façon suivante :

```
"%3.2f" % x
```

Pour généraliser:

```
%a.bf % une_variable_contenant_un_chiffre_a_formater
```

Où *a* correspond au nombre total de chiffres que l'on veut retourner, *b* correspond au nombre de chiffres après la virgule et *f* indique que l'on veut que le format du nombre sorti soit celui d'un réel.

On aurait pu aussi choisir d'autres formats. Voici une liste :

d -> entier relatif
e -> nombre réel au format exponentiel
s -> chaîne de caractères

```
x = 10000000
print("%1.1e" % x)
>>>> 1.0e+05
```

```
x = "une chaine de caractère"
print("%1.10s" % x)
>>>> une chaine
```

```
x = -2.42321
print("%1.1d" % x)
>>>> -2
```

EXERCICE : Y'A TROP DE CHIFFRES

Votre patron n'est pas matheux du tout, la vue des chiffres l'horrifie à un niveau qui dépasse presque l'entendement. Vous devez pourtant lui rendre votre rapport où il y a quelques calculs

1. Calculez la valeur 1/3 , stockez la dans une variable que vous souhaitez. Faites sortir uniquement les 2 premiers chiffres
2. Faites ressortir uniquement les deux premiers chiffres de la variable

SOLUTION

PROJET
TEXT MINING

Si vous n'avez jamais fait d'analyse de données, l'expression text mining peut faire peur. C'est normal mais ce n'est pas si sorcier. Le but de ce concept est d'analyser des données textes et d'en tirer des informations.

1. Prenez le texte suivant et assignez le à la variable *"text"* : "Chaque Homme sur terre a un trésor qui l'attend, lui dit son coeur. Nous, les coeurs, en parlons rarement, car les Hommes ne veulent plus trouver ces trésors. Nous n'en parlons qu'aux petits enfants. Ensuite, nous laissons la vie se charger de conduire chacun vers son destin. Malheureusement, peu d'Hommes suivent le chemin qui leur est tracé, et qui est le chemin de la Légende Personnelle et de la félicité. La plupart voient le monde comme quelque chose de menaçant et, pour cette raison même, le monde devient en effet une chose menaçante."

2. Pour commencer, mettons tous les mots en minuscule

3. En text mining, nous ne voulons garder uniquement les mots qui ont un sens intrinsèque. Les prépositions, pronoms personnels, et la ponctuation sont à enlever. Ce processus est très long compte tenu des connaissances que nous avons mais tentons quand même une méthode.

4. Séparez votre variable *“text”* en fonction des virgules présentes dans le texte. On stockera cette séparation dans une variable nommée separation

5. Prenez le premier objet de la variable separation et stockez dans une variable mots_positifs

6. Trouvez l’index du mot *“trésor”*

7. Stockez le mot *“trésor”* dans une variable que l’on nommera *mot_positif*

8. *Nous avons réussi à tirer un mot positif de tout ce texte de Coelho. Dans du vrai text mining, nous aurions dû répéter cette étape pour chaque mot positif que nous aurions croisé. Il existe heureusement des fonctions qui nous permettent de le faire automatiquement mais, cela dépasse le programme de ce manuel.*

SOLUTION

CONDITIONS & BOUCLES

PLUS VOUS AVANCEZ DANS VOTRE VIE DE PROGRAMMEUR, PLUS VOUS ALLEZ CRÉER DES PROGRAMMES COMPLEXES. VOUS AUREZ ALORS BESOIN DE CONNAÎTRE DEUX CHOSES : LES CONDITIONS ET LES BOUCLES.

CONDITIONS

IF

La première instruction que l'on peut apprendre est la façon dont est créée une condition. C'est dans cette partie que les opérateurs de comparaison que nous avons vu plus haut vont être utiles. Commençons donc par un exemple :

```
a = 5
b = 10

if a < b:
    print("a est inférieur à b")
```

Voici la structure simple d'une condition en Python. Elle commence donc par le *if* puis une condition, les ":" puis à la ligne, vous intégrez votre condition.

Les espaces ont une importance cruciale en dessous de votre condition. L'espacement correspond à une tabulation ou 4 espaces. Si vous ne mettez pas ces espaces, python ne va pas comprendre que votre ligne de code fait partie de la condition. Par exemple :

```
a = 5
b = 10

if a < b:
print("a est inférieur à b")
```

```
>>>>>  File "<stdin>", line 2
    print("a est inférieur à b")
        ^
IndentationError: expected an indented block
```

Ici, nous avons de la chance car notre REPL nous indique une erreur et qu'il attendait une *indentation* (espacement).

ELSE

Il se peut que vous ayez deux actions possibles que vous souhaitiez que votre programme fasse en fonction de la condition que vous insérez. Une manière peu élégante de le faire est :

```
a = 5
b = 10

if a < b:
    print("a est inférieur à b")

if a > b:
    print("a est supérieur à b")
```

Si nous changeons les valeurs pour a, notre REPL nous retourne : *a est supérieur à b.*

Cependant, cette façon d'écrire est peu élégante et surtout n'est pas la plus lisible. Il est plus clair de ne garder un seul *if* par catégorie d'action que nous souhaitons faire. C'est pour cela que l'on utilise *else.*

```
a = 15
b = 10
```

```
if a < b:
    print("a est inférieur à b")
else:
    print("a est supérieur à b")
```

Else sert à indiquer à votre programme ce qu'il doit faire quand la condition n'est pas remplie (ou *False*).

Dans le cas du dessus, a < b est *False* donc le programme peut sortir a *est supérieur à b.*

ELIF

Le dernier concept utile pour les boucles est le *Elif*. Il vous sera utile si vous avez plusieurs conditions à poser pour un même type d'action. Par exemple :

```
a = 15
b = 10

if a < b:
    print("a est inférieur à b")
else:
    print("a est supérieur à b")
```

Dans cet exemple, nous essayons de comparer la variable *a* à la variable *b*. Nous avons trois possibilités :

a est inférieur à b
a est égal à b
a est supérieur à b

C'est pour cela que *elif* nous est pratique.

Si vous avez plus de trois conditions, ce qui n'est pas rare, vous pouvez multiplier les *elif*.

Voici un exemple :

```
a = input("entrez une valeur pour a")
b = 10

if a < b:
    print("a est inférieur à b")
elif a == b:
    print("a est égal à b")
elif type(a) is str:
    print("a est une chaîne de caractère, je ne peux pas comparer les deux
variables")
else:
    print("a est supérieur à b")
```

Dans ce programme, nous avons demandé à l'utilisateur d'entrer une valeur pour *a*, au lieu de la définir nous même. L'utiliseur peut tout à fait être malicieux et entrer une chaîne de caractères dans la variable. Il faut donc prévoir cela dans notre code. C'est pour cela qu'on a ajouté un *elif type(a) is str.*

CONCATÉNER LES CONDITIONS

Le dernier scénario que vous pouvez rencontrer lorsque vous créez une condition est d'avoir une action qui dépend de plusieurs conditions en même temps. Par exemple, si vous chercher à savoir si quelqu'un peut conduire, il faut que la personne :

a- Ait le permis de conduire
b- Qu'elle n'ait pas bu

Lorsque l'on rencontre ce genre de problématique, on utilise les opérateurs logiques *AND* et *OR* :

```
permis = True
a_bu = True

if permis == True and a_bu == False:
    print("Cette personne peut conduire")
else:
    print("Cette personne ne peut pas conduire")
```

Souvenez-vous donc bien de la différence entre AND et OR. Dans le cas ci-dessus, si nous avions remplacé AND par OR, nous aurions eu le résultat suivant :

```
>>>> Cette personne peut conduire
```

EXERCICE : AVEC DES SI, ON POURRAIT METTRE PARIS EN BOUTEILLE

On commence à avoir un peu de connaissances en la matière, faisons donc un quizz !

1. Demandez à l'utilisateur combien de fois la France a gagné la coupe du monde
2. Ecrivez maintenant une condition : Si la réponse est la bonne, le programme devra dire "Bravo ! C'est la bonne réponse" Si ce n'est pas la bonne réponse, le programme devra dire "Dommage, tente une autre question"
3. Répétez ceci jusqu'à ce que vous ayez assez de questions pour faire un quizz

SOLUTION

BOUCLES

Une boucle est un **processus itérati**f qui vous permet de répéter une action un certain nombre de fois. Dans notre vie, nous passons tous les jours par des boucles. Par exemple, “Tant que je n’ai pas fini la tâche 1, je ne commence pas la tâche 2”, “Tant que je ne connais pas par coeur ma leçon, je continue de l’apprendre”, “Tant que je n’ai pas fait 200 photos, je continue de photographier”.

Vous avez deux types de boucles: *While* et *For*.

BOUCLES WHILE

La boucle While vous permet de faire tourner votre programme tant qu’une condition, que vous avez vous même paramétrée est vraie. Voici un exemple :

```
a = 0
while a < 10:
     print("a est égal à".format(a))
     a += 1

>>>>
a est égal à 0
a est égal à 1
a est égal à 2
a est égal à 3
a est égal à 4
a est égal à 5
a est égal à 6
a est égal à 7
a est égal à 8
a est égal à 9
```

On voit tout d’abord que la structure de la boucle *while* reste la même qu’une condition *if*. Ce qui nous facilite la tâche en terme de structure.

Voyons ensuite ce qu'il se passe dans la boucle :

En français, tant que a est inférieur à 10, renvoie *"a est égal à valeur_de_a"* puis nous incrémentons a de 1

Voici comment procède un ordinateur :

Itération 0 :
a = 0
a < 10 → VRAI
>>>> a est égal à 0
a = 1

Itération 1:
a = 1
a < 10 → VRAI
>>>> a est égal à 1
a = 2

Itération 2:
a = 2
a < 10 → VRAI
>>>> a est égal à 2

.
.
.
.

Itération 9:
a = 10
a < 10 → FAUX
STOP

Dans la description de ce processus, nous voyons plus clairement que tant que la condition a < 10 est VRAIE alors la boucle continue.

Ceci nous amène à un point très important, ***la boucle est while est dangereuse*** car elle peut vous faire créer des fonctions infinies et donc faire planter votre navigateur ou console. Par exemple, si nous n'avions jamais incrémenté a de 1, la condition a < 10 serait toujours VRAIE et donc on entrerait dans une boucle infinie.

```
a = 0
while a < 10:
    print("ceci est une boucle infinie")
```

Dans l'ancien temps des ordinateurs, il aurait fallu éteindre l'ordinateur pour sortir de la boucle. Aujourd'hui, vous pouvez appuyer sur ***ctrl + c*** pour sortir d'une boucle infinie.

Un type de boucle un peu moins dangereux est donc la boucle *For*, voyons la particularité de cette boucle.

BOUCLES FOR

La boucle *For* fonctionne presque de la même manière qu'un boucle *While* sauf qu'***au lieu de tourner tant qu'une condition est vraie, elle va plutôt tourner tant qu'il y a un item présent dans l'objet sur lequel vous souhaitez faire votre itération.*** Pas très clair ?

Essayons plutôt de réécrire le code ci-dessus avec une boucle For :

```
for a in range(0,10):
    print("voici la valeur pour a: {}".format(a))

>>>>
voici la valeur pour a: 0
voici la valeur pour a: 1
voici la valeur pour a: 2
```

```
voici la valeur pour a: 3
voici la valeur pour a: 4
voici la valeur pour a: 5
voici la valeur pour a: 6
voici la valeur pour a: 7
voici la valeur pour a: 8
voici la valeur pour a: 9
```

Ici nous avons utilisé la fonction *range()* qui permet de donner un intervalle entre le premier paramètre et le second paramètre. En l'occurrence de 0 à 10.

En termes de résultats, nous arrivons à la même chose qu'avec la boucle while. Cependant, avec la boucle *For* nous avons donné ***une limite dans l'itération*** de a tandis que pour la boucle *while*, nous avons utilisé une condition pour arrêter l'itération.

Le vrai pouvoir de la boucle *For* réside dans la possibilité d'itérer sur **plein d'autres types de données**, et non uniquement des nombres. On peut en effet itérer des chaînes de caractères ou même des listes de la façon suivante :

```
chaine = "hello world"

for caractere in chaine:
      print(caractere.upper())

>>>>
H
E
L
L
O
```

```
W
O
R
L
D
```

Dans cet exemple, nous avons itéré sur chaque caractère de la variable *chaine* et l'avons mis en majuscule. Expliquons la structure de la boucle :

```
for item in une_variable:
    du code
```

Item correspond à chaque élément de la variable *une_variable*. Vous pouvez nommer *item* comme vous le souhaitez. Dans l'exemple ci-dessus, nous avons appelé notre item *caractere* mais nous aurions tout à fait pu écrire n'importe quel autre nom. Le tout est de rester cohérent avec le reste du code.
Si nous avons utilisé *caractere* comme nom de notre item, nous l'utilisons ensuite lorsque nous souhaitons appliquer la fonction *upper()*

Voici un autre exemple sur une liste :

```
a = [1, 3, "ceci est une liste", 3.10]

for item in a:
    print(item)

>>>>
1
3
ceci est une liste
3.10
```

Dans notre exemple, nous avons fait ressortir chaque item de notre liste a. Nous avons appelé ces items *item.*

DOUBLE-BOUCLE

Pour finir avec les boucles, vous pouvez créer un boucle à l'intérieur d'une boucle. Voici un exemple :

```
liste = [[1,2,3], [4,5,6], [7,8,9], [10, "hello", "world"]]

for i in range(0, len(liste)):
    for item in liste[i]:
        print(item)

>>>>
1
2
3
4
5
6
7
8
9
10
hello
world
```

Le processus est un peu complexe donc expliquons le par étape:

- Tout d'abord nous avons créé une liste que nous avons appelé *liste*. Elle-même contient 4 listes différentes .
- L'objectif est : "comment est ce qu'on peut itérer sur chacun des items présents dans les 4 sous-listes ?" → une double boucle
- Nous allons donc d'abord itérer sur chaque liste présente dans liste puis nous allons itérer sur chaque item présent dans la liste
- Pour cela nous créons une première boucle for *i in range(0, len(liste))*: (En

l'occurrence *len(liste)* vaut 4)

- Puis nous créons une seconde boucle qui va itérer sur les items de chacune des listes for *item in liste[i]*. Ici le *i* correspond au *i* que nous avons mis dans la première boucle *for*.
-

Si cela n'est toujours pas clair pour vous, essayons de reprendre le processus itératif comme nous l'avons fait tout à l'heure :

```
Itération 0:
i = 0
for item in liste[0]:
    print(item)

>>>>
1
2
3

Itération 1:
i = 1
for item in liste[1]:
    print(item)

>>>>
4
5
6

Itération 2:
i = 2
for item in liste[2]:
    print(item)
>>>>
7
8
9
```

Itération 3:
i = 3
for item in liste[3]:
    print(item)

>>>>
10
hello
world

Voici ce que vient de faire votre ordinateur de manière très concrète.

Techniquement, **vous pouvez créer autant de concaténations de boucle que vous le souhaitez** mais vous perdez énormément en lisibilité. A priori, s'il vous faut trois boucles concaténées, c'est qu'il y a de fortes chances que vous pouvez trouver un meilleur moyen d'arriver à vos fins.

De manière générale, on évitera de trop concaténer les boucles pour éviter de donner des migraines à vos lecteurs.

BREAK

Il se peut parfois que vous ayez à sortir de votre boucle avant que celle-ci ne se finisse. Pour cela, on utilise *break.*

```
nb_essais = 0
password = input("quel est le mot de passe")

while password != "1234":
    if nb_essais == 3:
        print("Vous avez essayé trop de fois, retentez plus tard")
        break
    else:
        nb_essais+=1
        password = input("quel est le mot de passe")
```

Dans cet exemple du mot de passe, nous avons ajouté une fonctionnalité. Si l'utilisateur tente plus de trois fois le mot de passe sans succès alors on sort de la boucle et le programme dit de retenter plus tard.

EXERCICE : UN QUIZ PLUS COMPLIQUÉ

Continuons sur notre quiz ; ajoutons-y de la complexité.
Tant que la personne n'a pas donné la bonne réponse, l'algorithme demandera de redonner une réponse.

1. Ecrivez une première question via *input()*
2. Ecrivez une condition qui imprime "Bravo ! Vous avez trouvé la bonne réponse" si la réponse est effectivement correct et "Dommage ! Tente à nouveau"
3. Mettez cette condition dans une boucle qui montre que tant que la réponse n'est pas bonne, le programme repose la question à l'utilisateur

SOLUTION

PROJET

UN QUIZ VRAIMENT MIEUX

Améliorons notre quiz. Cette fois, nous voudrions que notre utilisateur ait un total de trois chances pour réussir un quiz de trois questions. Si l'utilisateur utilise toutes ses chances, le programme devra s'arrêter et dire à l'utilisateur qu'il a perdu. Si l'utilisateur arrive à la fin du quiz avec toutes les bonnes réponses, le programme devra le féliciter.

Ce projet est beaucoup moins facile que les précédents.
Voici quelques conseils pour bien commencer

1. Modélisez votre problème d'abord avec une seule question. Une fois que vous aurez bien compris comment cela marche, vous n'aurez pas de problèmes à écrire les autres
2. La place de vos *if else* ont une importance capitale ici pour bien faire fonctionner le programme
3. Réfléchissez bien au type de boucle que vous voulez faire : *while* ou *for* ?
4. Il n'y a pas qu'une seule manière d'arriver au résultat !

SOLUTION

FONCTIONS

JUSQU'À MAINTENANT, NOUS AVONS UTILISÉ DES FONCTIONS PRÉ-ÉCRITES COMME *UPPER()*, *LOWER()*, *FORMAT()* ETC.

MAIS IL SE PEUT QUE VOUS AYEZ BESOIN DE CRÉER VOS PROPRES FONCTIONS ; NOTAMMENT SI VOUS VOYEZ QUE VOUS ÊTES EN TRAIN DE RÉÉCRIRE SOUVENT LE MÊME CODE.

LA MEILLEURE DES PRATIQUES EST DE NE PAS SE RÉPÉTER (DRY : DON'T REPEAT YOURSELF). VOYONS DONC COMMENT LES FONCTIONS PEUVENT NOUS ÊTRE UTILES.

QU'EST CE QUE C'EST QU'UNE FONCTION ?

Une fonction permet de stocker une instruction et de l'utiliser autant de fois que vous le souhaitez plus tard. Par exemple :

```
bonjour = "hello world"
nom = "Michel"

bonjour.upper()
nom.upper()
```

Ici nous avons utilisé deux fois la fonction *upper()* mais celle-ci vous la connaissez déjà. Essayons de voir une autre fonction que nous créons nous-mêmes.

CRÉER UNE FONCTION

```
def carre(nombre):
    nombre = nombre ** 2
    return nombre
```

```
a = 12
b = 2

carre(a)

>>>> 144

carre(b)
>>>> 4
```

Ici nous venons de créer une fonction qui retourne le carré d'un nombre.
La structure d'une fonction se fait de la manière suivante :

```
def nom_de_la_fonction(paramètre_1, paramètre_2, ...., paramètre_n):
      du code
```

On commencera toujours par le *def* puis on nommera toujours la fonction.
Les paramètres sont en revanche facultatifs. Il se peut que vous n'en ayez pas besoin.

```
return
```

Représente ce que la fonction va retourner comme résultat. Dans l'exemple ci-dessus elle va retourner la variable nombre que nous avons définie en paramètre de la fonction *carre*.

A ce stade de la lecture, vous devez sûrement vous poser une question : "pourquoi je ne fais pas *a.carre()* au lieu de *carre(a)*". C'est tout à fait normal ; cela vient de ce qu'on appelle la ***programmation Orientée Objets*** qui est un peu différente de la programmation que nous montrons dans ce chapitre. Nous en parlerons cependant plus tard dans le manuel.

EXERCICE : VOILÀ COMMENT ÇA FONCTIONNE

Créez une fonction qu'on appellera *sqrt* qui retourne la racine carré d'un nombre, la fonction prendra un seul paramètre que l'on appellera "nombre".
INDICE : l'équivalent d'une racine carrée est de mettre le nombre à puissance ½

SOLUTION

AJOUTER UN PARAMÈTRE PAR DÉFAUT

Dans certains cas, il est judicieux de définir un paramètre par défaut dans votre fonction pour que cela évite à l'utilisateur de remettre toujours le même paramètre s'il doit réutiliser votre fonction plusieurs fois d'affilée.

Pour ce faire :

```
def bonjour(prenom = None):
    if prenom == None:
        print("Je ne connais pas votre nom mais bonjour quand même")
    else:
        print("Bonjour {}".format(prenom))

bonjour()
>>>> Je ne connais pas votre nom mais bonjour quand même

bonjour("Michel")
>>>> Bonjour Michel
```

Ici ce que nous avons fait est que, par défaut, nous avons paramétré l'argument *prenom* comme *None*. De telle manière à ce que si notre utilisateur utilise simplement la fonction *bonjour()* sans mettre d'argument, elle ressortira quand même un résultat.

En l'occurrence : *Je ne connais pas votre nom mais bonjour quand même*

EXERCICE : 1ERE CLASSE OU 2ND CLASSE

La sncf voudrait informer ses voyageurs de la possibilité d'avoir un repas dans leur train. Obtenir un repas dépendra de la classe dans laquelle le voyageur se trouve (1ere classe ou seconde classe). Si un voyageur se trouve en 1ere classe, un repas lui sera servi. Si un voyageur se trouve en seconde classe alors il devra aller en wagon-bar pour pouvoir se restaurer.

Créez donc une fonction qui prendra un seul argument : classe. Par défaut cette argument sera paramétré en "seconde_classe". Puis écrivez la fonction qui permettra d'informer le voyageur de sa possibilité d'avoir un repas ou non.

SOLUTION

GÉRER DES EXCEPTIONS

QU'EST CE QUE C'EST ?

Si votre code est fait pour être partagé et utilisé par d'autres utilisateurs, il faut que tous les scénarios possibles soient pris en compte dans votre code. Ceci inclut les erreurs possibles qu'un utilisateur va pouvoir faire en utilisant votre code. Par exemple, si nous reprenons la fonction de la partie précédente, il est tout à fait possible qu'un utilisateur entre une chaîne de caractères au lieu d'un nombre. En programmation, ce type d'erreur s'appelle *exceptions*.

Voyons donc comment nous pouvons gérer cette exception avec notre fonction carré.

```
def carre(nombre):
    try:
        nombre = nombre ** 2
    except:
        print("cette valeur n'est pas valide, veuillez entrer un nombre")
    else:
        return nombre
```

```
carre(2)
>>>> 4

carre("hello")
>>>> "cette valeur n'est pas valide, veuillez entrer un nombre"
```

Vous pouvez donc voir que les exceptions sont gérées par:

try:
except:
else:

C'est la structure normale. On demande à notre programme "d'essayer" le code qu'on lui propose ; s'il rencontre une erreur à ce moment là il fera tourner le code en dessous du *except:*. Si tout se passe bien, il fera tourner le code en dessous du *else:*

NB : Nous aurions tout à fait pu écrire le code de la manière suivante :

```
def carre(nombre):
    try:
        nombre = nombre ** 2
        return nombre

    except:
        print("cette valeur n'est pas valide, veuillez entrer un nombre")

carre(2)
>>>> 4

carre("hello")
>>>> "cette valeur n'est pas valide, veuillez entrer un nombre"
```

Le code fonctionnera de la même manière. Cependant le premier code est plus clair pour un lecteur qui souhaiterait relire notre programme puisque ce

que l'on veut vraiment "essayer" est de faire l'opération nombre = nombre ** 2 et non pas de retourner la variable *nombre.*

Vous pouvez aussi gérer des types d'erreurs bien précis de la manière suivante :

```
def carre(nombre):
    try:
        nombre = nombre ** 2
    except TypeError:
        print("le paramètre entré n'est pas valide")
    else:
        return nombre
```

TypeError veut dire qu'il y a un problème lors du paramétrage.
Si nous écrivons par exemple :

```
carre("hello")
>>>> le paramètre entré n'est pas valide
```

GÉRER PLUSIEURS ERREURS À LA FOIS

On peut aussi gérer plusieurs exceptions à la fois. Tentons une fonction assez classique de "partage d'addition" qui nous permettrait de connaître le montant que chacun doit payer en fonction de l'addition totale et du nombre de personnes à table.

```
def partage_addition():
    try:
        total_addition = float(input("quel est le montant total de
l'addition"))
        total_personnes = float(input("quel est le nombre total de
personnes"))
        montant_par_personne =  total_addition / total_personnes
```

```
    except ValueError:
        print("Ne mettez qu'un nombre dans le montant total de l'addition
et le nombre total de personnes")
    except ZeroDivisionError:
        print("On ne peut pas diviser par 0")

    else:
        return print("le montant par personne est de
{:4.2f}".format(montant_par_personne))
```

Expliquons le code. Notre fonction *partage_addition()* demande à l'utilisateur le nombre total de personnes qui sont venues manger puis le montant total de l'addition. Ces deux informations sont respectivement stockées dans la variable *total_personnes* et *montant_par_personne*. Ensuite, on va diviser ces deux variables pour obtenir le montant par personne.

NB : la fonction input() enregistre les valeurs entrées par l'utilisateur en tant que **string***. C'est pour cela qu'on utilise la fonction* **float()** *pour convertir ces données en un nombre décimal.*

Dans cet exemple, notre utilisateur peut éventuellement faire deux erreurs :

1. Ecrire une chaîne de caractères (ex: "Nous étions 10")
2. Ecrire 0 dans le nombre total de personnes (Si vous vous souvenez vos cours de mathématiques, il est impossible de diviser par 0)

La première erreur est gérée par l'exception *ValueError* où nous la console retourne à l'utilisateur *"Ne mettez uniquement qu'un nombre dans le montant total de l'addition et le nombre total de personnes"*

La seconde erreur est gérée par l'exception *ZeroDivisionError*, où l'on dit simplement à l'utilisateur qu'il ne peut pas diviser par zéro.

Nous pourrions trouver encore d'autres erreurs possibles que l'utilisateur pourrait faire en entrant les différentes valeur mais le but de cet exemple est

vous montrer qu'il est possible de gérer plusieurs exceptions à la fois, simplement en les superposant les unes aux autres.

DES EXCEPTIONS HORS DES FONCTIONS

Les exceptions ne doivent pas nécessairement être imbriquées dans des fonctions. Vous pouvez tout à fait gérer plusieurs exceptions directement dans votre code ou même dans une boucle :

```
while password != 1234:
    try:
        password = input("entrez un mot de passe pour arrêter la boucle")
        password = int(password)
    except ValueError:
        print("ceci n'est pas un nombre")
```

De manière générale, pensez que s'il y a une possibilité d'erreur alors un utilisateur va la faire.
C'est pour cela que vous devez prendre en compte ce scénario dans votre fonction.

DONNER UN ALIAS À VOS EXCEPTIONS

On peut leur donner des alias et faire ressortir leur description dans votre *print*. Ceci peut être extrêmement utile car certaines erreurs sont très bien gérées de manière inhérente à votre console. Vous pouvez alors mettre une simple indication et ne pas avoir à réécrire la même explication qu'aurait donnée votre exception par défaut. Reprenons l'exemple d'au dessus :

```
while password != 1234:
    try:
        password = input("entrez un mot de passe pour arrêter la boucle")
        password = int(password)
```

```
    except ValueError as erreur:
        print("ceci n'est pas un nombre. Plus de détails en dessous \n",
erreur)
```

Dans cet exemple, nous avons ajouté un alias que nous avons appelé *erreur* :

```
except ValueError as erreur:
```

Nous avons ensuite ajouté cette erreur dans notre fonction *print()*. De fait, lorsqu'une erreur de type *ValueError* arrive, nous obtenons :

```
ceci n'est pas un nombre. Plus de détails en dessous
invalid literal for int() with base 10: 'une chaîne de caractère'
```

POUR ALLER PLUS LOIN : CRÉER VOS PROPRES EXCEPTIONS

Il se peut parfois que la console ne voie pas d'erreur dans votre code mais que, dans l'application que vous êtes en train de créer, il y ait des scénarios impossibles qui devraient soulever une erreur.

Si nous reprenons l'exemple du partage d'addition tel que nous l'avons créé dans la partie *Gérer plusieurs exceptions à la fois,* notre utilisateur peut mettre un nombre négatif de personnes présentes au restaurant ou une addition négative. Ceci n'a aucune logique dans notre application et nous devrions faire en sorte de créer une erreur si notre utilisateur entre un nombre négatif.

Voici comment faire :

```
def partage_addition():
    try:
        total_addition = float(input("quel est le montant total de
l'addition"))
        total_personnes = float(input("quel est le nombre total de
personnes"))
```

```
        if total_addition < 0 or total_personnes < 0:
            raise ValueError("On ne peut pas avoir moins de 0 personnes à

        montant_par_personne =  total_addition / total_personnes

    except ValueError as erreur:

        print("Ne mettez qu'un nombre valide dans le montant total de
l'addition et le nombre total de personnes")
        print(erreur)

    except ZeroDivisionError:
        print("On ne peut pas diviser par 0")

    else:
        return print("le montant par personne est de
{:4.2f}".format(montant_par_personne))
```

Qu'avons-nous fait de différent ?
Presque rien. Nous avons ajouté une condition dans notre *try:* qui dit que si l'une de nos variables *total_addition ou total_personnes* est négative alors notre programme doit soulever une erreur.

Pour soulever une erreur, il suffit d'utiliser le mot clef *raise* suivi de l'erreur qui correspond à ce que vous souhaitez faire. Dans notre cas, nous avons une *ValueError* puisque nous considérons qu'un nombre en dessous de 0 n'est pas une valeur valide. Dans la parenthèse, nous ajoutons le message que nous souhaitons que l'utilisateur voie.

Ensuite, nous avons simplement ajouté un *alias* dans l'exception *ValueError* pour faire ressortir l'erreur que nous avons soulevée dans notre condition *if.*

Le tour est joué, vous savez maintenant comment créer vos propres exceptions ! Cela peut s'avérer extrêmement utile lorsque vous aurez à collaborer sur des applications et que vous avez besoin que vos collègues comprennent plus facilement les erreurs qu'ils font.

EXERCICE : EXCEPTIONNELLEMENT POUR VOUS

Une entreprise de billetterie en ligne voudrait faire appel à vos services pour programmer leur outil. Le programme devra :

1. Demander à l'utilisateur combien de billet il souhaite prendre
2. L'utilisateur devra mettre uniquement un nombre de tickets
3. Chaque billet coûte 25 euros pour tout le monde. Personne ne dispose de tarif réduit
4. L'utilisateur ne pourra prendre moins de 1 ticket
5. L'utilisateur ne pourra pas prendre plus de 15 tickets d'un coup directement sur la plateforme, il faudra qu'il appelle le service pour obtenir un tarif de groupe
6. Modéliser ceci via une fonction que l'on appellera "acheter_son_billet". Cette fonction ne prendra pas d'arguments mais retournera le prix total de la commande

SOLUTION

PROJET

COMBIEN VOUS RAPPORTE VOTRE LIVRET A ?

Il parait que si Christophe Colomb avait mis 1€ de côté qu'il avait laissé à 2% d'intérêt par an, aujourd'hui ses héritiers n'auraient plus besoin de travailler pour vivre leur vie. Essayons de calculer la même chose pour notre livret A.

1. Créeons donc une fonction qui va nous permettre de savoir combien d'argent un utilisateur aura au total au bout d'un certain nombre d'années
2. L'utilisateur devra pouvoir appeler la fonction qui ensuite lui demander :
    - La somme totale qu'il souhaite placer
    - Le nombre d'année pendant lesquelles il souhaite placer son argent
    - Le taux d'intérêt auquel il a accès
3. Faites attention aux erreurs que peut rentrer l'utilisateur
    - S 'il entre une chaîne de caractères plutôt qu'un nombre, une erreur devra ressortir
    - Si l'utilisateur met des chiffres négatifs, une erreur devra sortir de même
    - Enfin, il est très probable que l'utilisateur écrive le taux d'intérêt en pourcentage (ex: 10%). Levez une erreur si tel est le cas pour le prévenir de mettre un nombre décimal
4. Pour terminer, le programme devra ressortir qu'un chiffre avec au maximum deux chiffres après la virgule.

INDICE : La formule pour calculer un taux d'intérêt est :

Retour sur capital investi =capital de départ (1 + intérêt)nombre d'années

SOLUTION

MANIPULATION AVANCÉE DES COLLECTIONS DE DONNÉES

DANS VOTRE CARRIÈRE DE PROGRAMMEUR, VOUS ALLEZ ÉNORMÉMENT MANIPULER DE COLLECTIONS DE DONNÉES.

C'EST POUR CELA QUE NOUS SOUHAITONS DÉVELOPPER UN PEU PLUS CES TYPES DE DONNÉES ET VOUS DONNER QUELQUES ASTUCES LORSQUE VOUS LES MANIPULEZ.

LISTES

La grande spécificité des listes est leur caractère muable. Ceci peut être d'une grande utilité comme vous poser quelques problèmes. Voyons comment manipuler des listes

INDEXATION

AJOUTER DES ITEMS

Pour ajouter des items vous disposez de plusieurs possibilités :

+=

```
list = [1,2,3,"a"]
list += [43]

print(list)
>>>> [1,2,3,"a", 43]
```

Avec l'opérateur +=, vous pouvez ajouter des items de manière assez simple. Faites attention cependant, si vous ajoutez des nombres, il faudra les mettre entre ***braquets [ ]*** pour que votre console reconnaisse le nombre comme un nouvel item de votre liste. Sinon il va tenter de faire une opération.

De la même manière, si vous souhaitez ajouter une liste de caractères, faites bien attention à utiliser les ***braquets [ ]*** sinon python va considérer chaîne caractère de la chaîne comme un nouvel item :

```
list += "hello"
print(list)
>>>> [1, 2, 3, 'a', 43, 'h', 'e', 'l', 'l', 'o']
```

```
list += ["hello"]
print(list)

>>>> [1, 2, 3, 'a', 43, 'h', 'e', 'l', 'l', 'o', 'hello']
```

.append()

Cette fonction marche presque de la même manière que += si ce n'est qu'elle est plus rapide d'exécution. Cette fonction va ajouter l'item à la fin de votre liste

```
list = [3,2,0,9]
list.append("hello world")
print(list)
>>>> [3,2,0,9, "hello world"]
```

L'autre avantage de cette fonction est que vous n'aurez pas à gérer les problèmes de chaîne de caractères comme avec +=

Vous pouvez aussi insérer une autre liste grâce à la fonction *.append()*

```
list = [3,2,0,9]
list.append([2,2])
print(list)
>>>> [3,2,0,9,[2,2])
```

Nous constatons un problème avec cette fonction : comment pouvons-nous ajouter plusieurs items d'un coup ? Cette fonctionnalité est prise en charge par la fonction *.extend()*

.extend()

Cette fonction permet en effet d'ajouter plusieurs items à la fois sans avoir à copier/coller plusieurs fois la même ligne. Voyez plutôt :

```
list = [3,2,0,9]
list.extend([2,30,31])
print(list)
>>>> [3,2,0,9,2,30,31]
```

.insert()

La fonction *.insert()* permet d'ajouter des items ailleurs qu'à la fin d'une liste.

```
list = [3,2,0,9]
list.insert(3, "un nouvel item")
print(list)
>>>> [3, 2, 0, 'un nouvel item', 9]
```

Comme vous pouvez le voir dans l'exemple ci-dessus, la fonction *.insert()* prend deux arguments :

- L'index dans lequel vous voulez insérer votre nouvel item
- L'item que vous souhaitez insérer

En l'occurrence nous avons inséré à l'index 3 la chaîne de caractère "un nouvel item".

ENLEVER DES ITEMS

Ajouter des items sur une liste c'est bien mais parfois, nous allons avoir besoin d'en enlever. Voici donc quelques méthodes utiles

del()

Le premier moyen de supprimer un item est par la fonction *del().*

```
list = [2,3,2,"hello world"]
del(list[-1])
print(list)
>>>> [2,3,2]
```

NB : Vous l'avez déjà vu dans la partie précédente mais vous voyez en quoi les index négatifs sont utiles dans cet exemple.

.remove()

La fonction *.remove()* va vous permettre de spécifier l'item que vous souhaitez enlever.

```
list = [3,1,"hello world"]
list.remove("hello world")
print(list)
>>>> [3,1]
```

Cependant, faites bien attention à mettre l'item exact dans votre *.remove()* sinon la fonction sortira une erreur.

.pop()

Le principal désavantage des deux fonctions présentées ci-dessus est qu'elles détruisent votre item et qu'il est, de fait, définitivement perdu. Grâce à la fonction *.pop(),* vous pouvez stocker la valeur que souhaitez enlever dans une variable.

```
list = [3,1, "hello world"]
item_supprime = list.pop()
print(item_supprime)
>>>> "hello world"
```

Par défaut, *.pop()* prendra le dernier item de votre liste. Cependant, vous pouvez spécifier l'index que vous voulez enlever

```
list = [3,1, "hello world"]
item_supprime = list.pop(1)
print(item_supprime)
>>>> 1
```

Enfin, ce que nous permet *.pop()* de faire est de déplacer un item dans une liste.

```
list = [3,1,"hello world"]
list.insert(1, list.pop())
print(list)
>>>> [3,"hello world",1]
```

EXERCICE : LISTE DE COURSES

Vous êtes chargé des courses de la maison cette semaine :

1. Créez une liste contenant les produits suivants à acheter :
   - Salade
   - Sel
   - Poivre
   - Viande
   - Pâtes
2. Oh non le petit a oublié de vous demander d'acheter du déodorant, ajoutez le sur votre liste
3. Finalement, ce soir au lieu de manger une salade, la famille voudrait prendre du Quinoa, remplacez le dans la liste
4. Enfin, si l'on mange du Quinoa, on ne va pas manger des pâtes. Enlever les de la liste de cette semaine MAIS, on aura besoin de pâtes la semaine prochaine donc ajoutez les dans la liste de la semaine prochaine.

SOLUTION

SLICES

QU'EST CE QUE C'EST ?

PRINCIPE GÉNÉRAL

Souvent, il se peut que vous soyez amené à gérer une sous-liste. C'est à dire qu'une partie d'une liste. En Data Science par exemple, vos bases de données sont souvent représentées par des listes que vous devez couper.
C'est exactement ce à quoi servent les ***slices***.

```
liste = list(range(100))
print(list)
>>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99]


moitie_de_liste = liste[0:50]
print(moite_de_liste)
>>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Que voit-on dans cet exemple ? Nous avons déjà vu la fonction *range()* qui permet de créer un intervalle entre 0 et le paramètre que vous avez décidé (ici : 100).

Si nous voulons simplement que les 50 premiers items de notre liste, il suffit de créer une *slice* qui est matérialisée par ***les braquets et les deux points***. Dans notre exemple, nous avons pris l'index de 0 à 50. Nous aurions tout à fait pu

prendre d'autres indexes.

De manière générale, la structure est la suivante :

Variable_slice[index_debut : index_fin]

Ainsi, si nous avions voulu les 50 derniers items nous aurions pu faire:

```
seconde_moitie_de_liste = liste[50:100]
print(seconde_moitie_de_liste)
>>>> [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Une manière encore plus élégante de faire serait la suivante

```
seconde_moitie_de_liste = liste[50:]
print(seconde_moitie_de_liste)
>>>> [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Si vous ne spécifiez pas l'index de fin, python va comprendre de lui même qu'il doit aller jusqu'à la fin de la liste. Cela fonctionne d'ailleurs aussi avec l'index de début

```
premiere_moitie_de_liste = liste[:50]
print(premiere_moitie_de_liste)
>>>> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Et donc, vous l'aurez deviné, si vous ne mettez pas d'index de début, ni d'index de fin, vous aurez la liste entière

```
liste_entiere = liste[:]
print(liste_entiere)
>>>> [0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13. 14. 15. 16. 17. 18. 19.
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99]
```

ATTENTION : Comme vous pouvez le voir, le dernier index n'est pas pris en compte dans votre intervalle. Par exemple, dans liste[0:50], le résultat s'arrête à l'item qui est à l'indice 49 et non l'indice 50. L'intervalle est donc un intervalle ouvert. Faites bien attention à cela lorsque vous faites vos slices.

SLICE À INDEX NÉGATIF

Ce concept est un peu difficile à comprendre mais il est très utile. Il arrive souvent que vous ne connaissiez pas la longueur de votre liste. La raison principale est que vous pouvez être amené à gérer des listes qui peuvent changer de taille.

Nous avons déjà vu comment prendre simplement le dernier item d'une liste en mettant avec *liste[-1]*. Mais comment peut-on faire pour par exemple obtenir les deux derniers items ?

```
liste = list(range(10))
deux_derniers_items = liste[-2:]
print(deux_derniers_items)
>>>> [8,9]
```

Et voilà, le tour est joué !

Dans notre exemple, nous avons considéré que notre index de départ était à la position -2. Ce qui correspond à l'avant-dernier item. Nous allons ensuite jusqu'à la fin de notre liste en mettant les deux points.

NB : Attention, cette façon d'écrire n'est pas équivalente à

```
deux_derniers_items = liste[-2:-1]
```

Ceci nous aurait donné

```
print(deux_derniers_items)
>>>> [8]
```

Car, comme nous l'avons mentionné plus haut, l'intervalle est ouvert.

LES SLICES FONCTIONNENT AUSSI POUR LES CHAÎNES DE CARACTÈRES

Vous pouvez tout à fait faire une slice sur une chaîne de caractères :

```
chaine = "hello world"
print(chaine[0:5])
>>>>"hello"
```

SLICER PAR INTERVALLE

Dans quelques cas, il se peut que vous ayez besoin de prendre, non pas chaque item de l'intervalle de votre slice mais plutôt 1 sur 2 ou 1 sur 3. Par exemple, nous pourrions vouloir ne prendre que les items qui ont un index pair. Voici comment faire :

```
liste = ["pair", "impair", "pair", "impair", "pair", "impair"]
liste[0::2]
>>>> ['pair', 'pair', 'pair']
```

En fait, vous êtes en train de comprendre qu nous pouvons donner le nombre d'item pris dans l'intervalle. Par défaut, python prend tous les items possibles dans l'intervalle.

Voici donc la structure complète des slices:

Variable_slice[index_debut : index_fin : item_a_prendre]

Dans l'exemple ci-dessus, nous avons donc pris de l'index 0 à la fin de la liste et nous sautons un item dans l'intervalle à chaque fois.

Nous aurions aussi renverser l'ordre des items de la façon suivante :

```
liste = [1,2,3,4,5]
liste = [::-1]
>>>> [5,4,3,2,1]
```

ENLEVER ET REMPLACER DES ITEMS

Pour finir avec les slices, nous pouvons aussi enlever et remplacer des items. Commençons par en enlever :

```
liste = [5,4,3,2,1, "j'aime pas cet item", "ni celui là", "et encore moins
celui-ci"]
del(liste[-3:])
print(liste)
>>>> [5,4,3,2,1]
```

Pas mal non ?

Voyons comment nous pouvons remplacer des items dans notre liste:

```
liste = [10,9,3,2,1]
liste[0:2] = [5,4]
print(liste)
>>>> [5,4,3,2,1]
```

Le tour est joué !

ATTENTION : Si vous ajoutez une chaîne de caractère de la façon suivante

```
liste[-1:] = "hello"
```

Vous obtiendrez le résultat suivant :

```
print(liste)
>>>> [5, 4, 3, 2, 'h', 'e', 'l', 'l', 'o']
```

Si vous souhaitez ajouter toute la chaîne de caractère en tant qu'un seul item faites :

```
liste[-1:] = ["hello"]
print(liste)
>>>> [5, 4, 3, 2, 'h', 'e', 'l', 'l', 'hello']
```

EXERCICE : UNE LISTE PAIRE

1. Créez la liste suivante
   list = [1,2,3,4,5,6,7,8,9,10]
2. Créez une nouvelle variable qui ne contiendra que les nombre pairs de *"list"*
3. Enlevez le 2e et 3e item de la nouvelle liste.
4. Pour terminer, nous aimerions avoir notre nouvelle liste ordonnée par ordre décroissant. Créez une troisième variable dans laquelle nous aurons la nouvelle liste de nombre pairs classés du plus grand au plus petit.

SOLUTION

DICTIONNAIRES

Nous avons vu ce qu'était un dictionnaire. La spécificité du dictionnaire est qu'il contient des clés associées à une ou plusieurs valeurs. Cela arrive assez

souvent d'avoir ce type de données à gérer notamment lorsque l'on traite avec des APIs en Python.

MANIPULER UNE CLÉ

AJOUTER UNE CLÉ

La première chose que vous voudriez sûrement faire est d'ajouter une clé à votre dictionnaire. Voici comment faire :

```
dic = {"prénoms": ["Michel", "Elisabeth", "Jean"], "noms": ["Dupont",
"Durand", "Dugenoux"]}
dic["professions"]= ["Chanteur", "Ecrivain", "Acteur"]
print(dic)
>>>>
{'prenoms': ['Michel', 'Elisabeth', 'Jean'], "noms":  ["Dupont", "Durand",
"Dugenoux"], 'profession': ['Chanteur', 'Ecrivain', 'Acteur']}
```

CHANGER UNE CLÉ

On peut utiliser la fonction *.update()* pour changer une clé:

```
dic.update({"prénoms": ["Adrien", "Lola", "Myriam"]})
print(dic)

>>>> {'prénoms': ['Adrien', 'Lola', 'Myriam'], "noms":  ["Dupont",
"Durand", "Dugenoux"], 'profession': ['Chanteur', 'Ecrivain', 'Acteur']}
```

La fonction *.update()* peut aussi vous permettre d'ajouter des clés:

```
dic.update({"couleur_preferee": ["rouge", "bleu", "vert"]})
print(dic)
>>>> {'prenoms': ['Adrien', 'Lola', 'Myriam'],"noms":  ["Dupont", "Durand",
"Dugenoux"], 'profession': ['Chanteur', 'Ecrivain', 'Acteur'],
'couleur_preferee': ['rouge', 'bleu', 'vert']}
```

SUPPRIMER UNE CLÉ

Pour supprimer une clé, vous pouvez utiliser la fonction *del()* de la même manière que les listes:

```
del(dic["noms"])
print(dic)
>>>> {'prenoms': ['Adrien', 'Lola', 'Myriam'], 'profession': ['Chanteur',
'Ecrivain', 'Acteur'], 'couleur_preferee': ['rouge', 'bleu', 'vert']}
```

NB : Si vous supprimez une clé, vous supprimez aussi les valeurs qui sont associées à cette clé.

BOUCLE AVEC LES DICTIONNAIRES

Itérer sur un dictionnaire peut paraître un peu compliqué au départ parce qu'on ne sait pas comment itérer sur les clés ou sur les valeurs des clés.
Voici donc comment faire.

ITÉRER SUR LES CLÉS D'UN DICTIONNAIRE

Commençons par le plus simple: itérer sur une clé fonctionne exactement de la même manière qu'itérer sur les items d'une liste. Par exemple:

```
dic = {'prenoms': ['Adrien', 'Lola', 'Myriam'], 'profession': ['Chanteur',
'Ecrivain', 'Acteur'], 'couleur_preferee': ['rouge', 'bleu', 'vert']}
for key in dic:
      print(key)

>>>>
prenoms
Profession
couleur_preferee
```

Rien de bien méchant jusqu'ici.
Cela se corse lorsque l'on veut itérer sur les items d'une clé.

ITÉRER SUR LES VALEURS D'UNE CLÉ

A première vue, cela peut paraitre complexe mais c'est en fait très simple grâce à la fonction *.values()*.

```
for value in dic.values():
      print(value)

>>>>
['Adrien', 'Lola', 'Myriam']
['Chanteur', 'Ecrivain', 'Acteur']
['rouge', 'bleu', 'vert']
```

ITÉRER SUR LES VALEURS D'UNE CLÉ ET LA CLÉ

Parfois il est utile de pouvoir sortir de notre console la clé ainsi que ses valeurs associées. Nous pouvons utiliser la fonction *.items()* pour cela

```
for item in dic.items():
      print(item)

>>>>
('prenoms', ['Adrien', 'Lola', 'Myriam'])
('profession', ['Chanteur', 'Ecrivain', 'Acteur'])
('couleur_preferee', ['rouge', 'bleu', 'vert'])
```

POUR ALLER PLUS LOIN : LES KWARGS

Kwargs veut dire : **Key Word Argument**. Il arrive très souvent que dans des fonctions, au lieu de se limiter à un simple paramètre, nous voulions entrer un

dictionnaire. Cela arrive très souvent dans les fonctions un peu complexes qui peuvent avoir plusieurs catégories d'arguments possibles.
Voyons plutôt:

```
def keyword(**kwargs):
      print(kwargs)

keyword(prénom="Michel", nom="Delpeche", profession="Chanteur")

>>>> {'prenom': 'Michel', 'nom': 'Delpeche', 'profession': 'Chanteur'}
```

Ici, nous venons de créer une fonction qui peut prendre plusieurs valeurs associées à plusieurs mots clés en tant qu'arguments. D'ailleurs on peut voir que la fonction nous retourne un dictionnaire.

*NB : N'oubliez pas les ** avant les kwargs car c'est ce qui vous permet de définir le fait que votre argument est un kwarg*

Ceci nous permet donc rassembler plusieurs variables dans un seul endroit. Ce qui peut être extrêmement pratique dans certains cas.

A l'inverse, on peut aussi entrer des kwargs en tant qu'argument de notre fonction.

```
def bonjour(prenom, nom):
if prenom and nom:
    print("bonjour {} {}".format(prenom, nom))
else:
    print("bonjour très cher inconnu")

bonjour(**{"nom":"Delpeche", "prenom":"Michel"})

>>>> bonjour Michel Delpeche
```

Ma fonction a reconnu mon dictionnaire comme étant des arguments de ma fonction. Et vous n'avez même pas de besoin de mettre le tout dans le bon ordre ! Lorsque vous aurez à gérer des données qui vous sont données par des APIs, ces kwargs peuvent être d'une grande utilité.

EXERCICE : UN PEU PLUS DE TEXTE MINING

Prenons la phrase suivante : "Dans la vie on ne fait pas ce que l'on veut mais on est responsable de ce que l'on est."

Utilisons les dictionnaires pour compter combien de fois chaque mot apparait dans la phrase. Le résultat devra être le suivant:

{'dans': 1, 'la': 1, 'vie': 1, 'on': 2, 'ne': 1, 'fait': 1, 'pas': 1, 'ce': 2, 'que': 2, "l'on": 2, 'veut': 1, 'mais': 1, 'est': 1, 'responsable': 1, 'de': 1, 'est.': 1}

1. On créera une fonction que l'on appellera *frequence_de_mots()* et prendra "phrase" comme argument
2. Mettez tous les mots contenus dans "phrase" en minuscule
3. Séparez chaque mot de cette phrase dans une liste qu'on appellera "liste"
4. Dans une boucle for, itérez sur chaque item de "liste" et comptez le nombre de fois qu'il apparait. Vous stockerez ceci dans un dictionnaire qu'on appellera "dic".

SOLUTION

TUPLES

Comme on l'a vu dans la partie *Types de données*, les tuples sont immuables. C'est à dire que l'on ne peut pas changer leur valeur. Cela limite donc beaucoup les choses qu'on peut faire avec. Par exemple, on ne pourrait pas utiliser *.pop()* ou *.del()*. Mais nous allons pouvoir faire bien d'autres choses !

ECHANGER LES VALEURS DANS UN TUPLE

On peut tout à fait échanger les valeurs dans un tuple

```
a = 100
b = 1000

a, b = b, a
print("a égal {}".format(a))
print("b égal {}".format(b))

>>>>
a égal 1000
b égal 100
```

A quoi cela sert-il me direz vous ? Et bien grâce à cette fonctionnalité, nous allons pouvoir créer des boucles encore plus puissantes qu'avant.
En effet, cette fonctionnalité nous permet d'assembler et désassembler des tuples.

ITÉRER AVEC DES TUPLES

Prenons une liste de tuples

```
liste_tuples = [("antoine", 24), ("léa", 22), ("margaux",20)]
```

Cette liste de tuples représente un prénom et l'âge de la personne associée à ce prénom.

Grâce à la possibilité d'assemblage / désassemblage des tuples, nous pouvons itérer sur le prénom ainsi que l'âge en même temps :

```
for prenom, age in liste_tuples:
    print("{} a {} ans".format(prenom, age))
```

```
>>>>
antoine a 24 ans
léa a 22 ans
margaux a 20 ans
```

POUR ALLER PLUS LOIN : ARGS

Dans les dictionnaires, on a des Kwargs et dans les tuples on a des Args. Ils fonctionnent de la même manière sauf que le premier va créer des dictionnaires alors que le second va créer des tuples. Voyons une fonction :

```
def prenoms(*prenoms):
    nb_de_prenoms = 0
    print("Tes prénoms sont :")
    for prenom in prenoms:
        nb_de_prenoms += 1
        print(prenom)
    print("Ce qui fait {} jolis prénoms !".format(nb_de_prenoms)

prenoms("Michel", "Vincent", "Jean-Jacques")

>>>>
Tes prénoms sont :
Michel
Vincent
Jean-Jacques
Ce qui fait 3 jolis prénoms !
```

Le gros avantage de ce type de notation est que vous n'avez pas besoin de préciser à l'avance le nombre d'arguments total que doit avoir votre fonction !

EXERCICE : CALCULER UNE MOYENNE AVEC DES ARGS

Grâce aux args, nous avons un moyen simple de calculer une moyenne
1. Créez une fonction *moyenne()* qui prendra *chiffres comme argument
2. Faite en sorte que la fonction retourne la moyenne des nombres

SOLUTION

SETS

Même si les Sets sont moins utilisés, il est nécessaire de savoir comment les manipuler. Voici quelques méthodes qui peuvent s'avérer utiles.

.UNION()

```
set1 = {1, 2,4, 12}
set2 = {0,1,2,4,102,120}

set1.union(set2)
>>>> {0,1,2,4,12,102,120}
```

.union() fonctionne de la même manière qu'un *join()* pour des sets. La seule différence est que puisque les sets ne peuvent pas avoir deux valeurs similaires, la fonction ne va retourner que les valeurs uniques de chaque set.

Un autre système de notation pour *union()* est le | (shift + alt + L) :

```
set1 | set2
>>>> {0,1,2,4,12,102,120}
```

.DIFFERENCE()

On peut vouloir uniquement les items qui sont uniques pour l'un des deux sets. Ceci se fait avec la fonction *.difference().*

```
set1.difference(set2)
>>>> {12}
```

Comme vous pouvez le voir, nous n'avons qu'un seul chiffre qui est appartient seulement à set1: le chiffre 12.

A l'inverse si on veut connaître les valeurs uniques à set2, il suffira d'écrire:

```
set2.difference(set1)
>>>> {0, 120, 102}
```

On peut aussi écrire la *.difference()* de la manière suivante:

```
set1 - set2
>>>> {12}
```

.SYMETRIC_DIFFERENCE()

On peut vouloir avoir uniquement les valeurs uniques de chacun de deux sets en même temps. C'est ce qu'on appelle la différence symétrique :

```
set1.symmetric_difference(set2)
>>>> {0, 102, 120, 12}
```

On obtient donc uniquement les valeurs uniques dans les deux sets.
On peut aussi écrire de la façon suivante :

```
set1^set2
>>>> {0,102,120,12}
```

.INTERSECTION()

Pour finir, on peut aussi prendre uniquement les valeurs communes entre

deux sets avec la fonction *.intersection()*

```
set1.intersection(set2)
>>>> {1,2,4}
```

On peut aussi écrire :

```
set1 & set2
>>>> {1,2,4}
```

EXERCICE : QUE FAUT-IL POUR ÊTRE DATA SCIENTIST ?

Voyons voir si vous pourriez devenir Data Scientist

1. Créez une fonction nommée devenir_data_scientist() qui prendra “set” comme argument
2. Dans la fonction, créez un set nommé “competences_a_avoir” qui contiendra {“SQL”, “Python”, “Machine Learning”, “Statistiques”, “Big Data”, “A/B testing”, “Excel”, “Cloud Computing”, “Deep Learning”, “Spark”, “Hadoop”}
3. Maintenant, comparez les compétences inscrites dans “set” et “competences_a_avoir”. Demandez à la fonction de ne ressortir uniquement les compétences en commun via une variable qu’on appelle “competences_en_commun”
4. La console devra dire à l’utilisateur : “Vous avez déjà X Y Z comme compétences nécessaires pour devenir Data Scientists, continuez comme ca !”

SOLUTION

PROJET
TO-DO LIST

Pratiquons un peu les collections de données en créant une to-do list.
Une fois le programme lancée il devra :

1. Montrer la To-do list en cours
2. Demander si l'utilisateur veut ajouter une to-do
3. Demander si l'utilisateur veut enlever une to-do
4. Enlever ou ajouter la to-do correspondante

SOLUTION

UTILISER DES LIBRAIRIES EN PYTHON

QU'EST CE QU'UNE LIBRAIRIE ?

DÉFINITION

Vous avez surement déjà reçu comme conseil : "il faut marcher sur les épaules du géant". Et bien, les programmeurs ont bien compris ce principe et l'ont appliqué à Python.

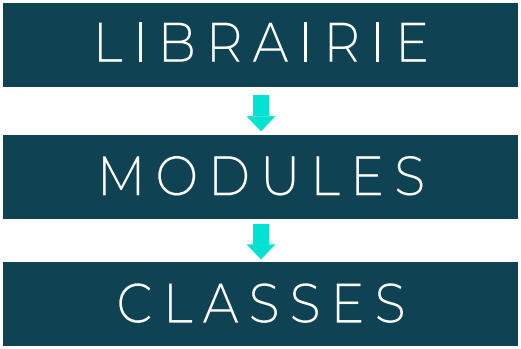
Une libraire est en fait du code pré-écrit que vous allez pouvoir utiliser à votre compte pour votre propre programme. Ce code pré-écrit est composé de fonctions qui vont vous faciliter la vie en tant que développeur.

Parmi les librairies les plus connues on retrouve:v ***Numpy, Pandas, Matplotlib, Scikitlearn, math, datetime***, et d'autres.

COMMENT SE STRUCTURE UNE LIBRAIRIE ?

Une librairie est composée de modules qui eux-mêmes sont composés de classes. Nous verrons cela plus en détail lorsque nous parlerons de programmation orientée objet.

Pour l'instant, vous avez juste à retenir les terme:.



ATTRIBUTES

⬇

METHODS == FUNCTIONS

COMMENT IMPORTER UNE LIBRAIRIE ?

```
import nom_du_module
```

Cette instruction va importer toute la librairie d'un coup dans votre code et vous pourrez commencer à l'utiliser.

Parfois certaines librairies sont assez lourdes et vous n'aurez besoin d'utiliser qu'une partie de celles-ci. Vous pouvez donc importer uniquement qu'un module de cette librairie de la façon suivante

```
from nom_du_module import nom_de_la_classe
```

COMMENT UTILISER UNE LIBRAIRIE ?

De manière générale vous allez utiliser une libraire de la manière suivante:

```
import nom_du_module
Une_variable = nom_du_module.nom_de_la_classe()
```

Il y a plusieurs choses sur lesquelles se concentrer. D'abord, le fait que l'on "appelle" toujours le **nom_du_module** puis le **nom de la classe** qu'on veut utiliser en le séparant d'un point .

Ensuite, une classe en elle-même ne peut pas être utilisée.

En revanche, on peut utiliser ce que l'on appelle une ***instance*** de cette classe. C'est pour cela qu'il est important de mettre les parenthèses après le nom de votre classe. Cela spécifie à Python que l'on créé cette fameuse instance.

Si tout cela semble un peu flou pour l'instant, c'est normal. Cela s'éclaircira lorsque l'on va voir l'utilisation d'une librairie avec un exemple.

Du fait de la diversité des librairies, nous ne pouvons pas toutes vous les présenter mais nous avons choisi deux qui vont vous être grandement utiles :

- ***Datetime q*ui va vous permettre de gérer des données temporelles**
- ***csv, json, excel* qui vont vous permettre d'importer et d'exporter des fichiers excel, json et csv**

GÉRER LES DONNÉES TEMPORELLES AVEC DATETIME

Dans votre vie de programmeur, vous allez être amené à gérer des données temporelles ; voyons comment la librairie **Datetime** peut nous aider.

PREMIERE APPROCHE

Regardons une première utilisation de notre libraire:

```
import datetime
aujourdhui = datetime.date.today()
print(aujourdhui)

>>>> 2018-07-19
```

Qu'a-t-on fait ici ? Nous avons utilisé le module datetime dans lequel il y a une classe date à laquelle nous avons appliqué la fonction *today()*.
Cela nous a permis d'avoir la date du jour (NDLR : Cette partie a été écrite le 19

Juillet 2018) que nous avons stocké dans la variable *aujourdui*)

NB : Vous remarquez que les dates sont mises au format américain. C'est à dire AAAA - MM - JJ. Faites bien attention car cela peut prêter à confusion.

En ayant stocké cette date dans notre variable nous pouvons maintenant utiliser de nombreux attributs de notre instance :

```
aujourdhui.month
>>>> 7

aujourdhui.day
>>>> 19

aujourdhui.year
>>>> 2018
```

Nous avons utilisé aussi un autre terme qui est celui d'attribut un attribut est simplement ce qui a après le.
Cela peut par exemple être une fonction.

Ici nous avons trois attributs plutôt utiles :

- .month
- .day
- .year

Ceux-ci vous permettent respectivement d'obtenir le mois d'une date, le jour d'une date et l'année d'une date.

QUELQUES FONCTIONS UTILES POUR LES DATES

Il y a pas mal de choses que vous pouvez faire avec la classe *date*. Voyons

quelques attributs utiles à avoir dans votre arsenal de connaissances en Python.

.REPLACE()

Vous pouvez changer une partie d'une date grâce à la fonction *replace().*

```
aujourdhui = datetime.date.today()
demain = aujourdhui.replace(day=20)
print(demain)
>>>> datetime.date(2018,7,30)
```

Dans cet exemple, nous avons remplacé le jour de la variable aujourdhui par 20, qui correspond au lendemain, 20 Juillet 2018.

On peut aussi utiliser pour des mois et des années :

```
mois_prochain = aujourdhui.replace(month=8)
anne_prochaine = aujourdhui.replace(year= 2019)
print(mois_prochain)
print(anne_prochaine)

>>>>
datetime.date(2018, 8, 20)
datetime.date(2019, 7, 20)
```

.WEEKDAY()

On peut connaître le jour de la semaine auquel tombre votre date grâce à la fonction *.weekday().*

```
aujourdhui = datetime.date.today()
aujourdhui.weekday()
>>>> 1
```

NB : Comme vous pouvez le voir, la fonction retourne un numéro qui correspond à un jour de la semaine comme suit :

0 --- > Lundi
1 --- > Mardi
etc

.ISOFORMAT()

Cette fonction retournera une date en tant que chaîne de caractères :

```
iso = datetime.date(2018,7,9).isoformat()
print(iso)
>>>> '2018-07-09'
```

NB : Attention le format iso retourne AAAA - MM - JJ et non le format français JJ - MM - AAAA

.STRFTIME()

Si vous souhaitez retourner une date un peu plus personnalisée, vous pouvez utiliser la fonction *.strftime()* où vous pouvez spécifier exactement le format que vous souhaitez utiliser.

```
une_date = datetime.date(2018,7,10)
une_date.strftime("%d/%m/%y")

>>>> ''10/07/18'
```

Ici nous avons retourné le jour d'abord puis le mois puis l'année, respectivement représentés par *%d, %m, %y.*

Il existe d'autres formats que vous pouvez utiliser.
Voici un tableau récapitulatif:

ECRITURE	DÉFINITION	EXEMPLE
%a	*Retourne le jour de la semaine abrégé*	*“Mon” / “Tue” / “Wed*
%A	*Retourne le jour de la semaine complet*	*“Sunday” / “Monday”*
%d	*Retourne le jour de la semaine en chiffre*	*01, 02, 03, 04…*
%b	*Retourne le mois de l’année en abrégé*	*“Jan” / “Feb” …*
%B	*Retourne le mois de l’année complet*	*“January” / February”*
%m	*Retourne le mois de l’année en chiffre*	*01, 02, 03, 04…*
%y	*Retourne l’année à deux chiffre*	*98, 99, 00, 01*
%Y	*Retourne l’année à quatre chiffres*	*1998, 1999, 2000, 2001*
%c	*Retourne la date et l’heure*	*Tue Aug 16 21:30:00 1988*
%x	*Retourne juste la date*	*08/16/88*
%X	*Retourne juste l’heure*	*21:30:00*

NB : A chaque fois, votre console va essayer de s'adapter aux formats locaux. Si vous écrivez "%A" et que votre console sait que vous vous situez en France. Elle retournera "Lundi" "Mardi" etc.

APPLIQUER CES MÊMES FONCTIONS POUR LES HEURES ET LES HORODATAGES

Tout ce que vous venez de voir pour les dates s'applique aussi pour les heures et les horodatages (qui sont les dates combinés aux heures).
Voyons quelques exemples:

POUR LES HEURES

```
une_heure = datetime.time(20, 32, 43)
une_heure.hour
>>>> 20

une_heure.minute
>>>> 32

une_heure.second
>>>> 43
```

APPLICATIONS DES FONCTIONS LIÉES AUX HEURES

```
une_heure = une_heure.replace(hour = 18)
print(une_heure)
>>>> 18:32:43

une_heure.isoformat()
>>>> '18:32:43'

une_heure.isoformat()
'18:32:43'
```

```
une_heure.strftime("%Ih %Mmin %Ssec")
'06h 32min 43sec'
```

NB : voici une liste des formats utiles pour les heures

ECRITURE	DÉFINITION	EXEMPLE
%H	Retourne l'heure au format 24h	"01" "03" "22"
%I	Retourne l'heure au format 12h	"01" "03" "12"
%M	Retourne les minutes	"01" "58" "59" "00"
%S	Retourne les secondes	"01" "58" "59" "00"

POUR LES HORODATAGES

Faites attention avec les horodatages car nous utilisons une classe qui s'appelle *datetime*. Ceci peut porter à confusion car elle porte le même nom que le module ***datetime.*** *Ne confondez pas les deux.*

Le module ***datetime*** contient les classes *date, time* ET *datetime*. Vous utiliserez donc la classe *datetime* de la même manière que les autres.

Voyons un exemple

```
un_horodatage = datetime.datetime.today()
print(un_horodatage)
>>>> 018-07-23 15:47:31.433580
```

```
un_horodatage.year
>>>> 2018

un_horodatage.month
>>>> 7

un_horodatage.day
>>>> 23

un_horodatage.hour
>>>> 15

un_horodatege.minute
>>>> 47

un_horodatage.second
>>>> 31
```

Comme on peut le voir ici, la librairie datetime combine à la fois les dates et les heures. Ce qui peut être très utile.

DES FONCTIONS SPÉCIFIQUES AUX HORODATAGES

Vous aurez toujours les fonctions *.replace()*, *.isoformat()* ou *strftime()* dans votre classe datetime mais vous avez aussi quelques fonctions qui peuvent s'avérer utiles et qui sont spécifiques à cette classe.

.date() ou .time()

Ces deux fonctions retournent la date d'un horodatage ou respectivement, l'heure d'un horodatage.

```
un_horodatage.date()
>>>> datetime.date(2018, 7, 23)
```

```
un_horodatage.time()
>>>>datetime.time(15, 47, 31, 433580)
```

.combine()

Il est possible que vous ayez une variable *heure* et une variable *date* et que vous souhaitiez combiner les deux. Et bien voici:

```
une_heure = datetime.time(19,32,00)
une_date = datetime.date(2018,7,1)
combinaison = datetime.datetime.combine(une_date,une_heure)
print(combinaison)
>>>> 2018-07-01 19:32:00
```

.strptime()

Souvent, il vous arrivera d'avoir à transformer des chaînes de caractères qui correspondent à des dates en format *datetime* pour que vous puissiez faire vos calculs. C'est exactement ce que la fonction *.strptime()* vous aide à faire.

```
une_date = "20/11/18"
date_format = datatime.datetime.strptime(une_date, "%d/%m/%y")
print(date_format)
>>>> 2018-11-20 00:00:00
```

Dans cet exemple, nous n'avons pas mis d'heure mais nous aurions très bien pu le faire.

```
un_horodatage = "20/11/18 18:30"
date_format = datatime.datetime.strptime(un_horodatage, "%d/%m/%y %H:%M")
print(date_format)
>>>> 2018-11-20 18:30:00
```

Cette fonction est extrêmement puissante et mérite d'avoir une belle place dans votre arsenal car vous serez amené à l'utiliser souvent.

GÉRER DES CALCULS AVEC DES DONNÉES TEMPORELLES

TIMEDELTA

Faire des calculs sur des dates n'est jamais une tâche aisée. C'est pour cela que le module ***datetime*** vous facilite la vie avec la classe *timedelta*. Cette classe représente la différence en jours et secondes entre deux dates.
Par exemple :

```
date1 = datetime.date(2018, 7, 10)
date2 = datetime.date(2018, 8, 10)

delta = date1 - date2
print(delta)

>>>> -31 days, 0:00:00
```

Ici on peut voir qu'il y a une différence de 31 jours entre *date1* et *date2*.

Nous n'avons pas l'impression de voir timedelta en oeuvre. Pourtant, si on regarde le type de donnée de la variable delta.

```
type(delta)
<class 'datetime.timedelta'>
```

Nous voyions ici que c'est une variable de type *timedelta*. C'est donc grâce à cette classe que nous pouvons faire nos calculs.

OPÉRATION AVEC TIMEDELTA

Si vous avez besoin d'enlever ou d'ajouter des jours, des mois, des années ou

même des heures à une date, vous pouvez le faire de la façon suivante :

```
date1 = date1 - datetime.timedelta(10)
print(date1)
>>>> 2018-06-30
```

Dans cet exemple, nous avons enlevé 10 jours à date1

Timedelta se structure de la manière suivante :
datetime.timedelta(jours, secondes, microsecondes, minutes, heures, semaines)

Donc si nous souhaitons enlever 10 jours et 2 heures, nous pouvons écrire:

```
horodatage1 = datetime.datetime.now()
horodatage2 = horodatage1 - datetime.timedelta(10, 0, 0, 0, 2, 0)

print(horodatage1)
>>>> 2018-07-23 16:57:24.051602

print(horodatage2)
>>>> 2018-07-13 16:55:24.051602
```

Grâce à *timedelta()*, on peut aussi ajouter, et comparer des dates:

```
horodatage2 = horodatage1 + datetime.timedelta(10, 0, 0, 0, 2, 0)
print(horodatage2)
>>>> 2018-08-02 16:59:24.051602

horodatage2 < horodatage1
>>>> False
```

EXERCICE : RAPPEL DE TÂCHE

Le samedi 7 Juillet 2018 à 15h30, vous avez un match de Tennis très important avec votre boss. Vous ne voulez surtout pas le louper du coup vous décidez d'écrire un programme qui va vous permettre de vous le rappeler.

1. Créez un programme qui permette de vous rappeler la chose 3 heures avant que le match ne commence
2. Vous n'avez pas loupé le match! Bravo. Du coup votre boss veut vous inviter pour tous les samedis du mois du Juillet. Mettez vous un rappel pour chacune de ces dates.

ATTENTION : Vous ne pouvez pas itérer avec une boucle for sur l'objet datetime

SOLUTION

LIRE ET CRÉER DES FICHIERS AVEC PYTHON

MÉTHODE GÉNÉRALE DE LECTURE ET ÉCRITURE D'UNE FICHIER

Dans les exemples qui vont suivre, nous utiliserons un fichier se nommant ***list.txt*** qui représente simplement une liste de prénoms.

LECTURE D'UN FICHIER

Commençons par voir la façon dont nous pouvons lire un fichier de manière générale.

La méthode est très simple

```
with open("list.txt") as file:
    for line in file:
        print(line)

>>>>
Sandrine

Didier

Thien

Jean-Luc

Charles

Julien

Léa
```

La nouveauté ici est que nous avons fait une lecture structurée du fichier list.txt :

with open*(“nom_fichier_a_ouvrir ”)* as *un_alias*:
   votre_code

On peut simplement avoir des informations sur le fichier

```
with open("list.txt") as file:
        print(file)

>>>> <_io.TextIOWrapper name='list.txt' mode='r' encoding='UTF-8'>
```

Nous pouvons également faire quelques manipulations utiles:

```
with open("list.txt") as file:
    list = []
    for line in file:
        list += [line]
    print(list)

>>>> ['Sandrine\n', 'Didier\n', 'Thien\n', 'Jean-Luc\n', 'Charles\n',
'Julien\n', 'Léa\n']
```

Pouvoir lire un fichier est donc essentiel car cela peut vous donner une première base pour des manipulations plus importantes.
Cependant, il peut être frustrant de vous limiter à la lecture d'un fichier. Voyons comment nous pouvons exporter nos manipulations, et donc, écrire un fichier.

ECRITURE D'UN FICHIER

Pour écrire un fichier, il est nécessaire d'ajouter un argument à la fonction *open()*.

```
with open("list.txt", "a") as file:
    file.write("Yves")
>>> 4
```

La console nous a renvoyé le nombre de caractères présents dans "Yves". C'est pour cela que l'on voit le chiffre 4. Mais, si on ouvre à nouveau notre fichier list.txt, on verra le nom de Yves apparaître.

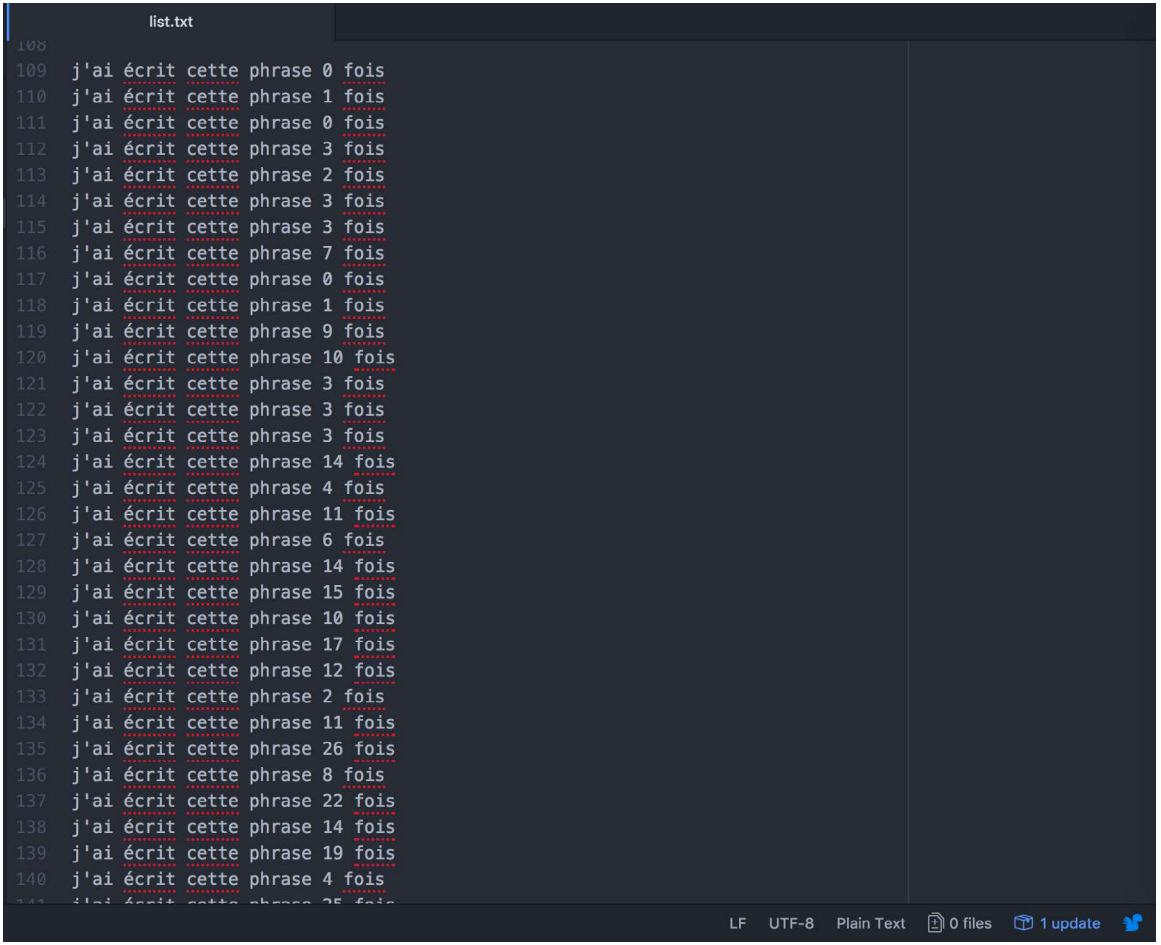
La plupart du temps, on inclura ce code dans une fonction pour pouvoir le réutiliser et garder une trace de ce que nous avons fait :

```
import random
def nb_alea():
    for i in range(100):
        with open("list.txt", "a") as file:
            file.write("\nj'ai écrit cette phrase {}
```

```
fois".format(random.randint(0,i)))

nb_alea()
```

Cette fonction *nb_alea()* permet de sortir un nombre aléatoire avec la phrase : "*j'ai écrit cette phrase i fois"* où i est un nombre aléatoire, et nous allons à la ligne avant chaque nouvelle phrase.

Maintenant si on ouvre notre fichier "list.txt" on voit :



A chaque fois que nous réutiliserons la fonction *nb_alea()*, nous obtiendrons 100 nouvelles lignes avec *"j'ai écrit cette phrase i fois"* où i est un nombre aléatoire.

Quelle est donc la différence entre la lecture et l'écriture ? Elle réside simplement dans le second argument que vous allez renseigner dans votre fonction *open()*. En fonction de l'argument entré, python saura quoi faire avec le fichier.

Voici la liste des différents arguments :

CARACTÈRE	SIGNIFICATION
"r"	Lecture simple du fichier. Cet argument est l'argument par défaut. C'est pour cela que nous n'avons eu besoin de le mettre lorsque nous parlions de lecture de fichiers
"w"	Efface tout ce qu'il y a dans le fichier avant d'écrire quelque chose
"x"	Crée et écrit sur un fichier nouveau. Attention, la fonction retourne une erreur si le fichier existe déjà
"a"	Ecrit sur un fichier à la suite de ce qui existe déjà

Dans l'exemple du dessus, nous avons utilisé l'argument *"a"* car nous avions besoin d'écrire des nouvelles lignes sans effacer les anciennes. Si nous avions eu besoin d'effacer d'abord ce que contenait le fichier, nous aurions utilisé *"w"*.

LIRE DES FICHIERS CSV, JSON ET EXCEL

Maintenant que nous avons compris comment lire et écrire des fichiers de manière générale, il existe des librairies qui servent uniquement à gérer certains types de fichiers. Les principaux sont les csv, les fichiers json et les fich-

iers excel. Voyons les un par un.

NB : Avant de commencer vos manipulations, soyez certains que vous enregistrez votre fichier csv en UTF-8 sinon votre console aura du mal à interpréter certains caractères spéciaux.

CSV

Les fichiers csv sont les fichiers les plus standards lorsqu'il s'agit d'importation et d'exportation de données. C'est pourquoi nous allons commencer par celui-ci. Nous allons utiliser un fichier *notes.csv* qui contient 15 élèves et des notes sur 20 réparties de manière aléatoire.

.reader()

Pour lire un fichier csv, on utilisera la fonction *csv.reader()* :

```
import csv
with open("notes.csv", newline="") as csvfile:
    notes = csv.reader(csvfile, delimiter=",")
    for row in notes:
        print(row)

>>>>
['Prénom', 'Note']
['Sandrine', '0']
['Didier', '3']
['Thien', '4']
['Jean-Luc', '2']
['Charles', '18']
['Julien', '10']
['Léa', '1']
['Yves', '13']
['Michel', '20']
['Arthur', '20']
```

```
['Jacques', '15']
['Adèle', '19']
['Dalida', '8']
['Myriam', '20']
```

Même si la structure de lecture de fichiers standards reste presque identique, nous retrouvons quelques modifications sur lesquelles nous allons nous attarder :

Tout d'abord dans notre fonction *open()*, nous avons ajouté l'argument *newline=""*.
Il est toujours bon d'ajouter cet argument lorsque vous ouvrez un fichier. Dans notre cas, ne pas mettre l'argument ne change rien mais de manière générale si newline="" n'est pas mis, vous risquez d'avoir des problèmes lorsque vous aurez des lignes qui comportent des guillemets.

Nous avons ensuite utilisé la fonction *.reader()* dans laquelle nous avons ajouté les arguments :

- **csvfile**
- **delimiter = ","**

Le premier argument désigne le fichier que nous voulons lire. En l'occurrence, nous avions appelé ce fichier *csvfile* dans notre boucle.

Ensuite, nous devons désigner le caractère qui va nous servir de séparateur entre chaque colonne. Ici, nous avons choisi la virgule.

On obtient donc chaque ligne sous forme de liste séparée par le séparateur que nous avons choisi.

.DictReader()

Le seul désavantage de *.reader()* est qu'il est plutôt difficile de séparer le nom des colonnes de leur contenu.

C'est pour cela qu'on utilise aussi. *DictReader()*

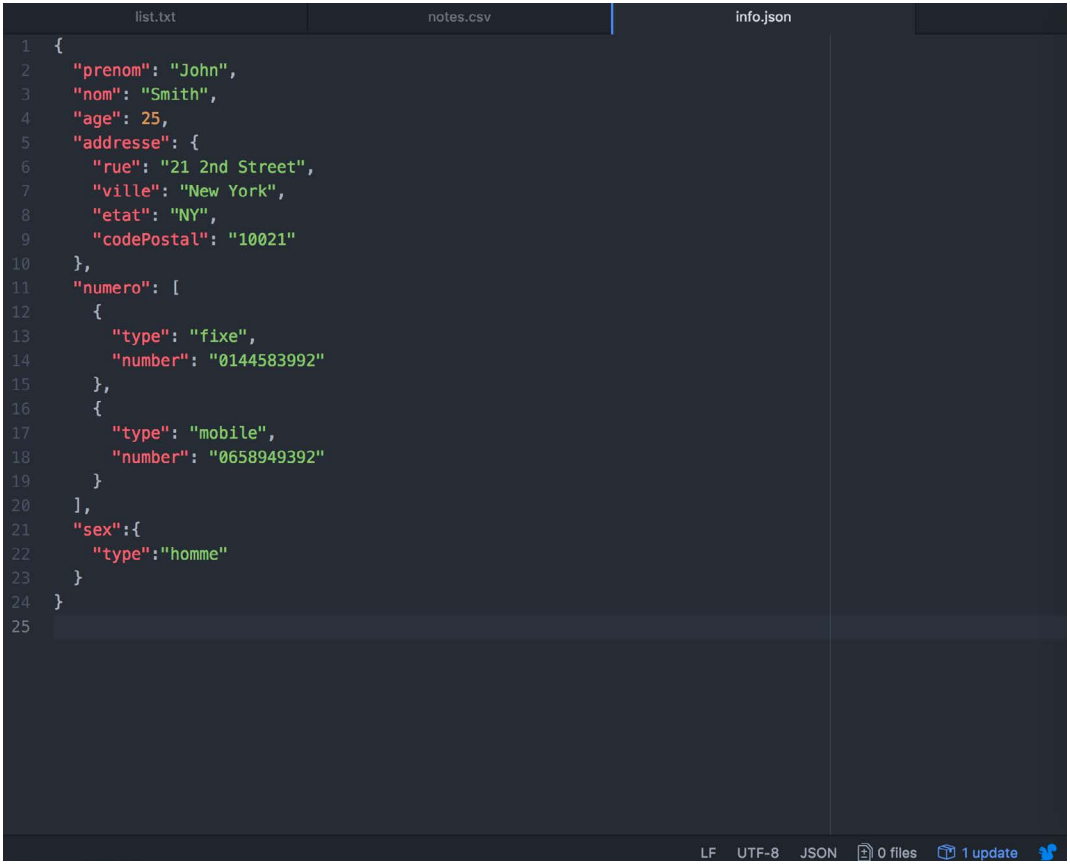
```
import csv
with open("notes.csv", newline="") as csvfile:
      notes = csv.DictReader(csvfile)
      for row in notes:
            print(row['Prénom'], row['Note'])

>>>>
Sandrine 0
Didier 3
Thien 4
Jean-Luc 2
Charles 18
Julien 10
Léa 1
Yves 13
Michel 20
Arthur 20
Jacques 15
Adèle 19
Dalida 8
Myriam 20
François 14
```

Maintenant, notre programme arrive à reconnaître la colonne *Prénom* et la colonne *Note* et nous pouvons manipuler plus facilement leur contenu.

JSON

L'acronyme JSON signifie **JavaScript Object Notation.** Bien que nous n'écrivons pas en Javascript, il arrive très souvent que vous ayez à manipuler ce type de fichiers car ils sont très présents dans l'univers du web. Heureusement pour nous, Python possède aussi une librairie pour cela.

Si vous n'avez aucune idée de ce à quoi peut ressembler un fichier JSON, voici un exemple.



Cela s'apparente presque à un Dictionnaire en Python. Nous avons une clé associée à une ou plusieurs valeurs.

Pour la suite des exemples, nous utiliserons ce fichier que nous appellerons *info.json.*

Pour lire un fichier, rien de plus facile, on garde toujours la même structure de lecture de fichier :

```
import json
with open("info.json") as info:
      file = json.load(info)
      print(file["adresse"])

>>>>
{'rue': '21 2nd Street', 'ville': 'New York', 'etat': 'NY', 'codePostal':
'10021'}
```

Qu' y a-t-il de différent ?
Ici, nous avons utilisé la fonction *.load()* qui permet de lire le fichier json.

Ensuite, nous avons demandé à la console de renvoyer les informations liées à l'adresse de l'utilisateur.

Voici comment nous pouvons lire des données sur un fichier JSON.

Finissons par Excel pour que vous ayez un bon aperçu des fichiers que nous pouvons lire avec Python.

EXCEL

Dans cette partie, nous allons utiliser une librairie moins populaire que *csv* et *json*. Elle peut cependant vous être utile.

Cette librairie s'appelle *openpyxl* et vous permet donc de lire et écrire des fichiers excel. Pour la suite, nous utiliserons un fichier *notes.xlsx* qui ressemble au même fichier que celui que nous avons utilisé pour notre partie sur la librairie csv.

```
import openpyxl
excel = load_workbook(filename="notes.xlsx")
worksheet = excel.active

for row in worksheet.rows:
    for cell in row:
        print(cell)

>>>>
Prénom
Note
Sandrine
0
Didier
3
Thien
4
Jean-Luc
2
Charles
18
Julien
10
Léa
1
Yves
13
Michel
20
Arthur
20
Jacques
15
Adèle
19
Dalida
8
```

```
Myriam
20
François
14
```

Pour lire un fichier, nous allons cette fois utiliser la fonction *load_workbook()* qui va nous permettre de lire un fichier excel.

Ensuite, nous pouvons accéder à la cellule d'une feuille excel en accédant d'abord à la ligne. C'est pour cela qu'on utilise une double-boucle : cela nous a permet d'accéder à chaque cellule de notre fichier excel.

EXPORTER DES DONNÉES EN FICHIER CSV, JSON OU EXCEL

Maintenant que nous savons comment accéder à des données sur chacun de ces fichiers, nous pouvons voir comment exporter ces données.
La méthode est similaire à celle que nous avons vu plus haut.

CSV

Pour écrire sur un fichier csv, nous aurons besoin des fonctions *.DictWriter(), .writeheader(), .writerow().*

```
import csv
with open("notes.csv", "a") as csvfile:
    notewriter = csv.DictWriter(csvfile, fieldnames=["Prénom", "Note"])
    notewriter.writerow({
    "Prénom":"Morgane",
    "Note":"18"})
```

Si nous ouvrons notre fichier, on peut maintenant voir que nous avons *Morgane* avec une note de *18* ajouté à notre fichier csv.

La fonction *.DictWriter()* est très utile car elle nous permet de spécifier sur quel fichier vous souhaitez exporter vos données et sous quel nom de colonne. Ceci est matérialisé par les deux premiers arguments que nous avons vu ci-dessus.

Nous aurions pu aussi utiliser la fonction *.writer()* mais cette fonction pose le même problème que *.reader()*, nous n'avons pas accès aux noms de colonnes. C'est pour cela que nous utiliserons que *.Dictwriter()* dans cet exemple.

NB : Nous avons parlé au dessus de .writeheader() mais nous ne l'avons pas utilisé dans l'exemple du dessus. Cette fonction sert en effet à écrire les noms des colonnes dans le fichier csv. Ce qui n'aurait pas eu de sens ici.

JSON

Lorsqu'il s'agit de fichiers JSON, nous avons deux fonctions qui vont nous être utiles : *.dump()* & *.dumps().*

```
import json
couleur_pref = {'couleur_preferee':'bleu'}
with open("info.json", "a") as info:
    json.dump(couleur_pref, info)
```

Avec cette méthode, nous avons pu exporter la variable *couleur_pref* dans notre fichier *info.json.*

Nous évoqué plus haut la fonction *.dumps()*, celle-ci fonctionne de la même manière que la fonction *.dump()* à cela près qu'elle formate les chaînes de caractères au format JSON.
En effet, alors que la plupart du temps les guillemets et les apostrophes ne font pas de différence pour désigner une chaîne de caractères, en JSON, les guillemets sont la forme valide pour les écrire. Il est donc nécessaire de garder cela en tête si vous commencez à manipuler beaucoup de fichiers JSON.

```
json.dumps(couleur_pref)
>>>> '{"couleur_preferee": "bleu"}'
```

EXCEL

Terminons les écritures de fichiers avec Excel sur *openpyxl*. Cette librairie nous offre un certain nombre de fonctions qui peuvent nous être utiles pour exporter nos données.

Changer le titre d'une feuille

On peut changer le titre d'une feuille de calcul grâce à l'attribut *.title*

```
from openpyxl import load_workbook
excel = load_workbook("notes.xlsx")
sheet = excel.active
sheet.title = "Notes"
excel.save(filename="notes.xlsx")
```

Si vous ouvrez maintenant votre fichier excel, votre feuille de calcul sera nommée "Notes"

ATTENTION : vous devrez toujours finir votre code par .save(filename= "nom_du_fichier.xlsx") pour que le code que vous avez écrit se reflète sur votre fichier.

Créer une nouvelle feuille et insérer des valeurs

On peut aussi créer une nouvelle feuille et y ajouter des valeurs de la façon suivante :

```
sheet2 = excel.create_sheet(title="une_nouvelle_feuille")
for row in range(1,100):
    for col in range(1,10):
        _ = sheet2.cell(column= col, row = row, value=row)
excel.save(filename="notes.xlsx")
```

Voici un bon exemple où nous avons rempli les 100 premières lignes des 10 premières colonnes de valeurs allant de 1 à 100.

Nous aurions pu simplement utiliser la fonction *.cell()* et spécifier une valeur différente pour une colonne et une ligne données sans utiliser la double-boucle mais cet exemple est ici pour vous montrer différentes fonctionnalité de cette librairie.

Openpyxl étant très vaste, nous ne serons pas exhaustifs dans l'explication de toutes les fonctionnalités. Si vous êtes amené à manipuler plus de fichiers excel, nous vous conseillons de vous référer à la documentation : https://openpyxl.readthedocs.io/

EXERCICE : LISTE D'INVITÉS

C'est l'anniversaire de votre meilleure amie. Pour l'occasion, elle a prévu d'inviter tout le monde dans une boite de nuit branchée de Paris. Cependant la boîte de nuit demande la liste ordonnée par ordre alphabétique des personnes qui vont venir à la soirée

1. Demandez à l'utilisateur les noms des personnes présentes à la fête
2. Ordonnez cette liste par ordre alphabétique
3. Exportez cette liste au format csv. On mettra en première ligne le nom "invites".

SOLUTION

COMMENT FAIRE POUR LES AUTRES LIBRAIRIES ?

Nous vous avons montré quelques exemples de différentes librairies mais il en existe beaucoup d'autres que vous pouvez utiliser en Python.
Cependant, il faut savoir faire le tri entre les bonnes librairies et les librairies dont on peut se passer. Aussi, il y aura toujours des choses nouvelles que vous

devrez apprendre dans chacune des librairies. C'est pourquoi nous voulions vous donner deux conseils.

REGARDEZ LA DOCUMENTATION DES LIBRAIRIES QUE VOUS UTILISEZ

Le premier conseil que l'on peut vous donner est de ***lire la documentation relative à la librairie ou au module que vous souhaitez utiliser***. Plus les librairies sont connues, plus vous aurez de chances qu'elles soient bien faites.

Dans la plupart des cas, vous aurez tout ce qu'il vous faut pour trouver les solutions à vos problèmes. Donc n'hésitez pas à avoir plusieurs onglets ouverts sur votre ordinateur avec les différentes documentations des librairies que vous utilisez.

Il se peut toutefois que certaines documentations soient peu ou pas claires. Dans ces moments là, vous pouvez taper votre problème sur google. Souvent, vous aurez des articles de blog traitant exactement de votre problème ou sinon vous tomberez sur un forum comme ***StackOverflow*** où vous aurez quelqu'un qui a déjà eu le problème avant vous.

TOUTE LIBRAIRIE N'EST PAS BONNE À UTILISER

Souvent, les librairies peu ou pas claires le sont parce qu'elles sont nouvelles et que les personnes qui les ont développées les utilisent pour des problématiques bien précises, voire trop précises. Et, le fait que ces librairies ne soient pas très populaires peut vous poser des problèmes de compatibilité lorsque vous souhaitez industrialiser votre application.

Loin de nous l'idée de vous déconseiller de vous appuyer sur le travail de quelqu'un d'autre pour faire le vôtre. Simplement, il peut arriver que ce soit un cadeau empoisonné que d'utiliser une librairie qui n'est pas compatible avec tous les outils que vous devez utiliser par la suite.

Si vous commencez dans la programmation, nous vous conseillons donc plutôt de rester sur les libraires populaires qui devraient déjà résoudre beaucoup de vos problèmes.

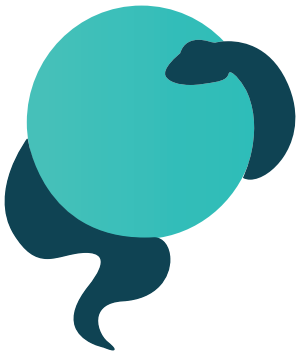
PROJET

UNE MEILLEURE TO-DO

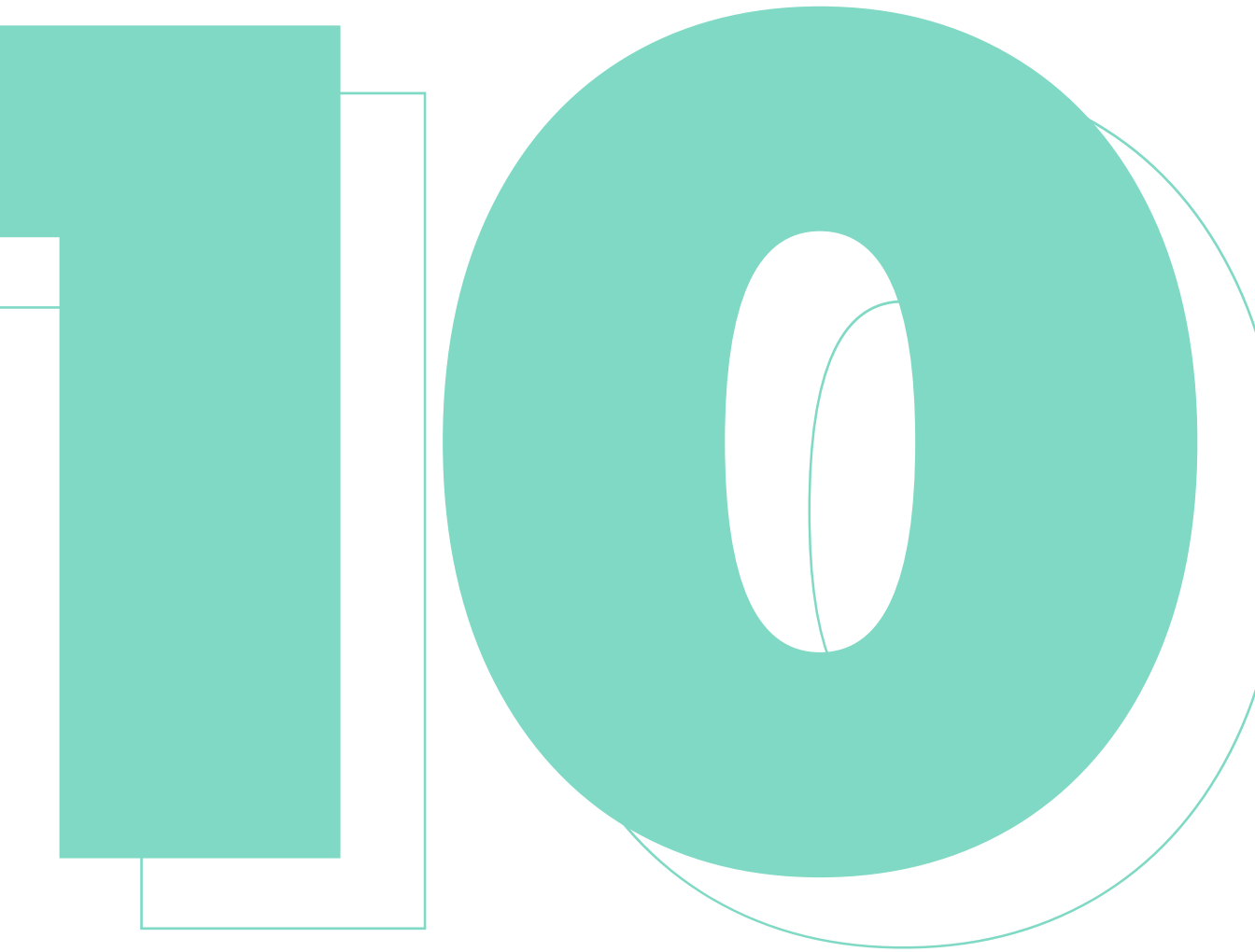
Le projet de la to-do que nous avons fait plus haut était vraiment bien mais notre utilisateur pouvait être un peu frustré car, à chaque que l'on sortait du programme, la liste s'effacait. On ne pouvait donc pas garder de trace de notre to-do.

1. Reprenons donc notre code et exportons cette to-do dans un fichier csv pour que nous puissions garder une trace de ce que nous devons faire.
2. Vous devrez pouvoir écrire les nouvelles to-do dans votre csv et lire les anciennes to-do depuis votre csv

SOLUTION



PROGRAMMATION ORIENTÉE OBJET



NOUS NE POUVIONS PAS ÉCRIRE CE MANUEL SANS EXPLIQUER CE QU'EST LA PROGRAMMATION ORIENTÉE OBJET.

NOUS EN AVONS FAIT SANS QUE VOUS LE SACHIEZ DANS CHACUNE DES PARTIES DU MANUEL CAR PYTHON EST UN LANGAGE FONDÉ SUR CE CONCEPT.

ÉTANT EXTRÊMEMENT VASTE, NOUS ALLONS ALLER À L'ESSENTIEL DANS L'EXPLICATION, NOTRE BUT EST DE VOUS FAIRE COMPRENDRE LES TERMES ET LA FAÇON DE PENSER DERRIÈRE CECI.

POURQUOI FAIRE DE LA PROGRAMMATION ORIENTÉE OBJET ?

La **programmation orientée objet** est apparue dans les années 80 dans le but de faciliter la vie des programmeurs. Grâce à l'**OOP** (*Object Oriented Programming*), on a du code plus robuste, flexible et réutilisable.

L'OOP repose sur 4 principes fondamentaux :

1. ***Encapsulation***
2. ***Abstraction***
3. ***Héritage***
4. ***Polymorphisme***

Connaître la définition de ces concepts sur le bout des doigts ne servirait pas à grand chose pour un débutant. Mais voici une explication sommaire des quatre principes pour les curieux.

Encapsulation : Comme nous l'avons vu ci-dessus, nous avons des classes et dans ces classes, nous avons des attributs. C'est le principe de l'encapsulation. Tous les objets communiquent entre eux selon une certaine règle logique.

Abstraction : C'est le principe de pouvoir cacher le code qui n'est pas forcément utile à voir pour tous les développeurs. De la même manière que, sur votre smartphone, vous n'avez qu'un seul bouton qui vous permet de faire uniquement les actions les plus essentielles, les autres fonctions moins "utiles" de votre téléphone sont cachées dans vos réglages. Pour le code, c'est pareil. On tente de cacher toutes les formules compliquées pour faire en sorte qu'aucun novice ne puisse casser le code.

Héritage : Ceci veut simplement dire qu'on peut avoir une hiérarchie dans les objets. Par exemple, on peut avoir un classe "parents" qui peut donner des attributs réutilisables pour les classes "enfants".

Polymorphisme : Pour terminer le polymorphisme est le concept le plus complex à comprendre. Alors que l'héritage permet d'avoir des attributs réutilisables pour classes "enfants". Le polymorphisme veut dire, qu'en plus, ces attributs s'adaptent à la classe "enfants". Autrement dit, on aura des fonctions "générales" qui vont s'adapter à plusieurs situations différentes. Ces situations étant les classes "enfants".

LE JARGON

Vous avez surement entendu parler de beaucoup de mots de vocabulaire dans l'OOP. Nous en avons déjà défini la plupart mais nous souhaitons vous faire la liste complète, voici donc la structure

Un objet : N'importe quoi est un objet, une classe est un objet, une fonction est un objet etc. Lorsqu'on connaît pas le type de données auxquelles on a affaire, on va parler d'objet. La plupart du temps cependant, on sait de quoi on parle

Une classe : Une classe est une contient une collection de méthodes, de variables etc. qui vont permettre d'exécuter une certain nombre d'actions.

Un attribut : Ce sont les données contenues dans une classe. La plupart du temps, ce sont des fonctions

Une méthode : Lorsque l'on construit des fonctions, dans une classe, nous par-

lons en fait de méthodes. Dans ce manuel, nous avons parlé de fonctions mais, si nous avions été extrêmement rigoureux, nous aurions parlé de méthodes. Ceci a été fait exprès par soucis de clarté et de vulgarisation mais maintenant vous saurez vous saurez faire la différence.

Une instance : Nous en avons déjà parlé au dessus, une instance est donc un "exemple" de votre classe. C'est à dire que vous utilisez votre classe dans un cas précis

Ce sont les cinq mots de vocabulaire qu'il est bon de connaître lorsque l'on parle d'OOP. Faisons un peu de pratique en construisant une classe et quelques méthodes.

CRÉER UNE CLASSE

Supposons que nous gérons un garage et que nous souhaitions modéliser ce garage par du code. Dans un garage, nous avons des employés, des voitures à réparer et un nombre total de client qui sont passés par ce garage. Par souci de simplification, on dira qu'il n'y a que cela dans le garage, même si nous ne vous conseillons pas de tenter de lancer une modélisation simplement avec ces trois attributs.

```
class garage:
    employes = 5
    clients = 150
    voitures_a_reparer = 10

garage = garage()
garage.employes
>>>> 5
```

Ici nous avons créé une classe simple dans laquelle nous avons ajouté trois attributs *: employes, clients, voitures_a_reparer*

Ensuite, nous avons créé une instance de la classe *garage*, que nous avons

stockée dans une variable du même nom : *garage*.
Lorsque l'on crée une instance d'une classe, on utilise les parenthèse ()

Nous avons pu utiliser l'attribut *employes* par le biais de la notation ***"."*** : *garage.employes*

EXERCICE GÉRER SON BAR

Nous avons un tenant de bar qui souhaite avoir des informations sur son entreprise et les gérer informatiquement

1. Créez une classe bar
2. Créez trois attributs : employes, surface, nb_max_de_personnes
3. Assignez des valeurs pour ces trois attributs
4. Créez une instance de la classe bar et afficher les valeurs que vous avez souhaitez pour chaque attribut

SOLUTION

CRÉER UNE MÉTHODE

Nous comprenons un peu mieux la façon dont sont construites des classes et pourquoi nous avions à manipuler nos librairies de la manière dont nous l'avons fait plus haut. Essayons d'aller plus loin et construisons une méthode pour notre classe garage. Créons un nouvel attribut qui va être le nombre voiture réparées et créons une méthode que l'on appellera *voiture_reparee()* qui va permettre de dire que nous avons réparé une nouvelle voiture

```
class garage:
    employes = 5
    clients = 150
    voitures_a_reparer = 10
    voitures_reparees = 0
    def voiture_reparee(self):
        garage.voitures_a_reparer -= 1
```

```
        garage.voitures_reparees += 1

garage = garage()
garage.voitures_a_reparer
>>>> 10

garage.voiture_reparee()
garage.voitures_a_reparer
>>>> 9
```

Que fait notre méthode ? Elle ajoute une voiture réparée et enlève une voiture à réparer dans notre garage. Tout simple mais elle nous fait voir quelque chose d'utile : Qu'est ce que ce "self" ?

Ceci se rapporte à l'instanciation d'une **classe**. Comme nous l'avons dit plus haut, on utilise jamais une classe en elle-même, nous utilisons une instance. Dès lors, chaque méthode dans une classe prend au moins un argument qui est l'instance. C'est pour cela que nous devons au moins mettre *self* dans une classe.

Rendons notre fonction un peu plus complexe. Il se peut en effet que le garage répare plusieurs voitures à la fois. Au lieu d'appeler notre méthode plusieurs fois, ajoutons un argument qui va nous permettre de donner le nombre de voitures réparées.

```
class garage:
    employes = 5
    clients = 150
    voitures_a_reparer = 10
    voitures_reparees = 0
    def voiture_reparee(self, nb_de_voitures):
        garage.voitures_a_reparer -= nb_de_voitures
        garage.voitures_reparees += nb_de_voitures

garage = garage()
garage.voiture_reparee(3)
garage.voitures_reparees
>>>> 3
```

Et voici comment on ajoute des arguments à une méthodes. Finalement, rien n'est nouveau par rapport aux fonctions normales (fort heureusement). N'oubliez simplement pas de garder toujours l'argument *self* dans la méthode à cause de l'instanciation.

EXERCICE : DES MATHS FACILES

Les mathématiques, en général, on n'aime pas en faire. Surtout lorsqu'il s'agit de répéter des opérations, encore, encore et encore. Soyons de vrais paresseux et créons une classe qui s'occupera de faires les opérations qu'on veut pour nous.

1. Créez une classe qu'on appellera "math"
2. Cette classe ne prendra que des méthodes comme attribut
3. Créez une méthode qui calculera la racine carrée de n'importe quel nombre
4. Créez une méthode qui calculera la moyenne de n'importe quelle liste de nombres
5. Créez une méthode pour savoir si un nombre est pair ou impair
6. Enfin créez un méthode qui donnera la somme totale d'une liste de nombres

SOLUTION

LA FONCTION __INIT__()

Maintenant que vous comprenez un peu mieux ce que sont une classe et une méthode, il faut que nous parlions de l'une d'elle en particulier qui est la fonction *__init__().* Cette fonction est celle qui va initialiser votre classe lorsque vous créez une instance. Vous aurez besoin de l'utiliser la plupart du temps.

Reprenons l'exemple du garage, imaginons maintenant que nous n'avons pas qu'un seul garage à gérer et que ceux-ci sont de taille différente en termes d'employés, de clients et de voitures à réparer. On va pouvoir utiliser la classe *__init__()* de cette manière :

```
class garage():
    def __init__(self, employes, clients, voitures_a_reparer,
voitures_reparees = 0):
        self.employes = employes
        self.clients = clients
        self.voitures_a_reparer = voitures_a_reparer
        self.voitures_reparees = voitures_reparees

    def voiture_reparee(self, nb_de_voitures):
        garage.voitures_a_reparer -= nb_de_voitures
        garage.voitures_reparees += nb_de_voitures

garage1 = garage(10, 300, 15, 0)
garage2 = garage(15, 400, 20, 4)
garage3 = garage(5, 150, 10, 8)
```

Comme vous pouvez le constater, nous pouvons désormais avoir plein de garages différents avec des attributs que nous pouvons paramétrer à l'avance.

Cela couvre ce que nous voulions voir de l'OOP. Ceci n'est encore une fois pas l'entièreté du concept mais vous en avez maintenant une bonne vision et pouvez comprendre de quoi on vous parle si vous êtes amené à travailler avec des experts.

PROJET

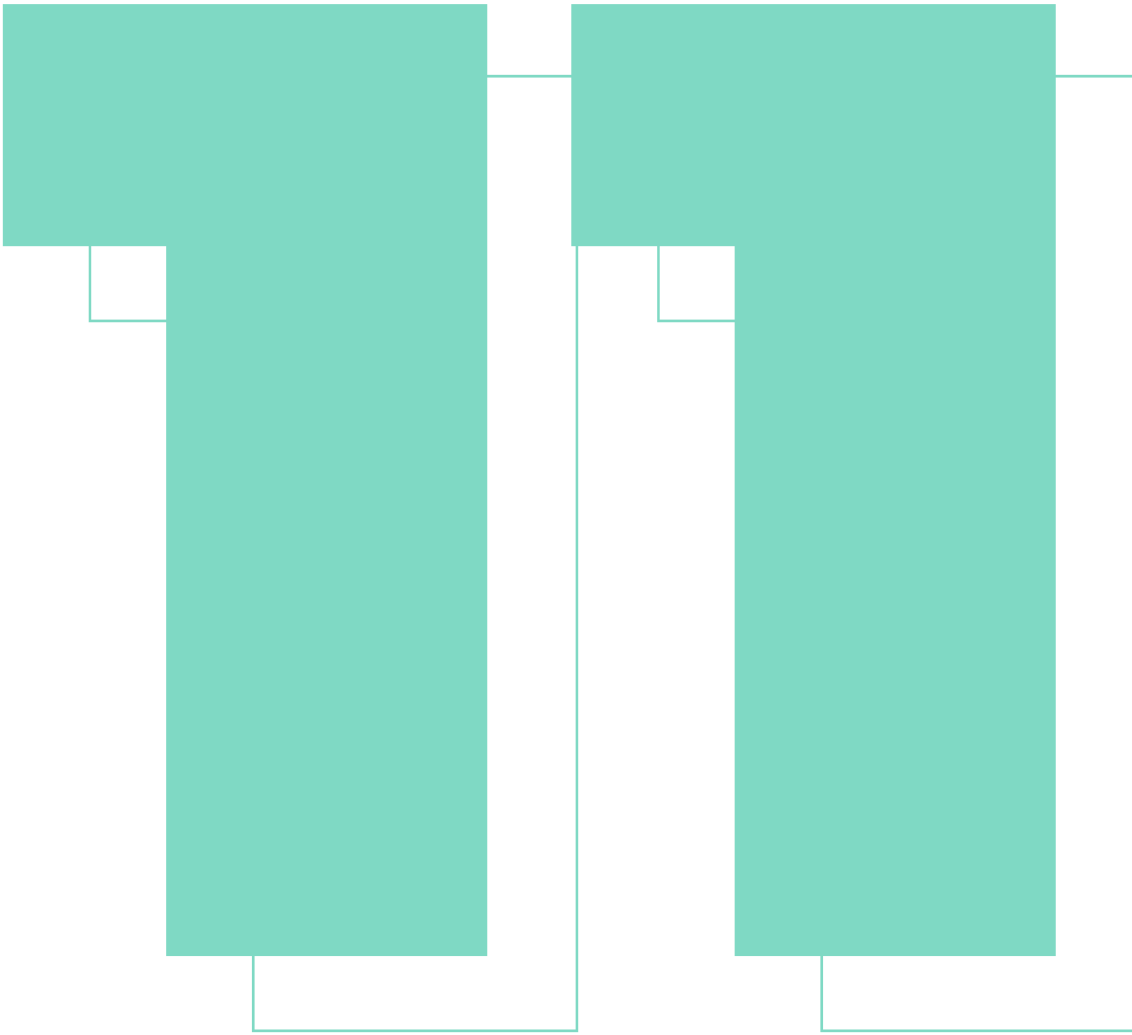
DEVINETTE

Utilisons la programmation orientée objet pour faire un jeu de devinette. L'ordinateur va choisir un nombre au hasard entre 1 et 10 puis l'utilisateur devra deviner ce que l'ordinateur a choisi.

1. Importez la librairie random pour pouvoir utiliser random.randint(1,10) qui vous permettra de sortir un nombre aléatoire entre 1 et 10
2. Créez une classe devinette qui aura trois fonctions :
   - *__init__(self) qui initialisera le nombre d'essais et le nombre à deviner*
   - *deviner(self) qui demandera au joueur de deviner un nombre*
   - *rejouer(self) qui demandera au joueur s'il souhaite rejouer une partie*
3. Mettez en place un programme qui va permettre de jouer à ce jeu de devinette

SOLUTION

INTEGRATED DEVELOPMENT ENVIRONMENT (OU IDE)



AUX BONS OUVRIERS LES BONS OUTILS ET QUAND ON PARLE DE CODE, IL EST UTILE DE CONNAÎTRE LES OUTILS QUI VONT VOUS PERMETTRE D'ÉCRIRE VOTRE CODE DE MANIÈRE EFFICACE.

C'EST POUR CELA QUE NOUS VOUDRIONS DÉDIER NOTRE DERNIÈRE PARTIE SUR LES ENVIRONNEMENTS QUE VOUS POUVEZ UTILISER POUR VOUS FACILITER LA VIE.

APRÈS AVOIR DÉFINI CE QU'EST UN IDE, NOUS ALLONS FAIRE UN TOUR D'HORIZON DES AVANTAGES ET DÉFAUTS DE CHACUN POUR QUE VOUS PUISSIEZ CHOISIR LEQUEL TÉLÉCHARGER EN TEMPS VOULU.

QU'EST CE QU'UN IDE ?

IDE ou Integrated Development Environment est un outil qui rassemble la console et l'éditeur de texte en un seul endroit. En fonction des objectifs de ce pourquoi l'IDE a été conçu, vous aurez d'autres outils en plus du simple éditeur de texte et de la console.

Les IDE sont de multiples variétés. Certains sont multi-langage et assez généralistes ; ils servent à coder tous types d'applications. D'autres peuvent être spécialisés dans un langage et / ou dans un secteur.

Voici quelques exemples d'IDE populaires :

- *Xcode* (pour développer des applications mobiles sur IOS)
- *Android Studio* (pour développer des applications mobiles sur Android)
- *PyCharm*, *Spyder* et *Jupyter Notebook* (pour Python)
- *Cloud9* (IDE généraliste et cloud)

L'avantage principal d'un IDE est qu'il intègre tout ce dont vous avez besoin dans un seul endroit et cela peut vous faire gagner en clarté pour comprendre votre code. Encore mieux, certains IDE sont intégrés directement dans le cloud

donc vous n'avez même pas besoin d'installer quoique ce soit sur votre machine en local. Donc pas de problème d'installation, de compatibilité ou autre problème d'IT compliqué !

Le désavantage majeur est souvent que l'on perd en vitesse d'exécution puisque le système intègre souvent plus de choses que ce dont vous avez réellement besoin.

PYCHARM

LES AVANTAGES DE PYCHARM

Commençons par l'IDE le plus généraliste en Python : ***Pycharm***. Avec cet outil vous allez pouvoir faire du web development comme de l'analyse de données de manière très efficace. Pycharm intègre en effet tous les frameworks dont vous avez besoin en développement de sites internet comme ***Django*** ou ***Flask*** et permet de vous faire des aperçus live de votre code.
De la même manière, il intègre les librairies les plus utilisées en Data Sciences comme ***Matplotlib*** ou ***Numpy*** ; ce qui vous évite de les télécharger indépendamment.

Vous l'avez compris, l'avantage principal de Pycharm est qu'il a été très bien pensé pour **être généraliste** sans pour autant perdre en clarté.

UN DÉFAUT ?

Pycharm *est un outil à deux vitesses,* vous avez une version payante et une version gratuite. Vous aurez besoin de la version payante pour avoir accès aux features intéressantes dont vous aurez de toute façon besoin. L'outil est tellement bien fait que le prix n'est pas problématique, mais il vaut mieux le savoir à l'avance pour ne pas être surpris.

Pour la suite, nous vous présenterons deux IDE dédiés aux Data Sciences, mais qui eux, sont gratuits.

JUPYTER NOTEBOOK

Si, avant d'ouvrir ce manuel, vous avez exploré l'univers technologique autour de python, vous avez surement dû voir passer ce qu'on appelle un ***notebook***.

En effet, vous pouvez écrire du code de deux manières. D'abord sur votre éditeur de texte et faire tourner votre code via la console, comme ce que nous avons fait jusqu'ici. Le fichier où votre code est écrit s'appelle alors un *script.* Mais si vous vous tournez vers les Data Sciences, un outil très utilisé est ce qu'on appelle le *Notebook* qui rassemble l'éditeur de texte et la console dans un même endroit et qui vous permet de faire tourner une séquence de votre code à la fois.

Vous pouvez rencontrer deux noms possibles pour le *Notebook* : ***Jupyter Notebook*** ou ***IPython Notebook.*** Le premier étant l'appellation la plus récente, utilisée depuis les dernières mises à jours du projet IPython.

LES AVANTAGES DU NOTEBOOK

Tout d'abord, Jupyter est conçu pour l'analyse de données. Vous avez donc les librairies et packages les plus utilisés déjà pré-installés sur votre notebook, Ce qui vous évite de devoir télécharger vos packages à la main sur votre machine pour pouvoir faire fonctionner votre code.

Au delà de Python, Jupyter inclut d'autres langages de programmation comme *R*, *Julia* et *Scala*. Si dans votre parcours, vous êtes amené à apprendre d'autres langages, vous ne serez pas obligé de changer complètement d'environnement de développement. Ce qui vous fera gagner beaucoup de temps d'adaptation.

Les notebooks sont assez facilement partageables puisque légers.
Vous pouvez utiliser des outils comme Dropbox, Github ou même envoyer votre code par email.

Enfin Jupyter notebook gère très bien toutes les problématiques Big Data en incluant des outils comme Spark pour Python.

Tout ceci a fait qu'aujourd'hui, Jupyter notebook est extrêmement populaire et largement utilisé dans le domaine de l'analyse de données.

UN DÉFAUT TOUT DE MÊME

Si vous commencez dans l'analyse de données, cet outil peut paraître abscon à première vue. En effet, l'interactivité de l'outil n'est pas vraiment adaptée aux débutants et vous pouvez rapidement vous retrouver perdu.

C'est pour cela que dans un premier temps, nous vous conseillons d'utiliser des IDE comme *Spyder* ou *Pycharm* qui vont vous paraître plus simples à prendre en main.
Une fois que vous vous sentez à l'aise, n'hésitez pas à repasser sur Jupyter Notebook.

SPYDER

AVANTAGES DE SPYDER

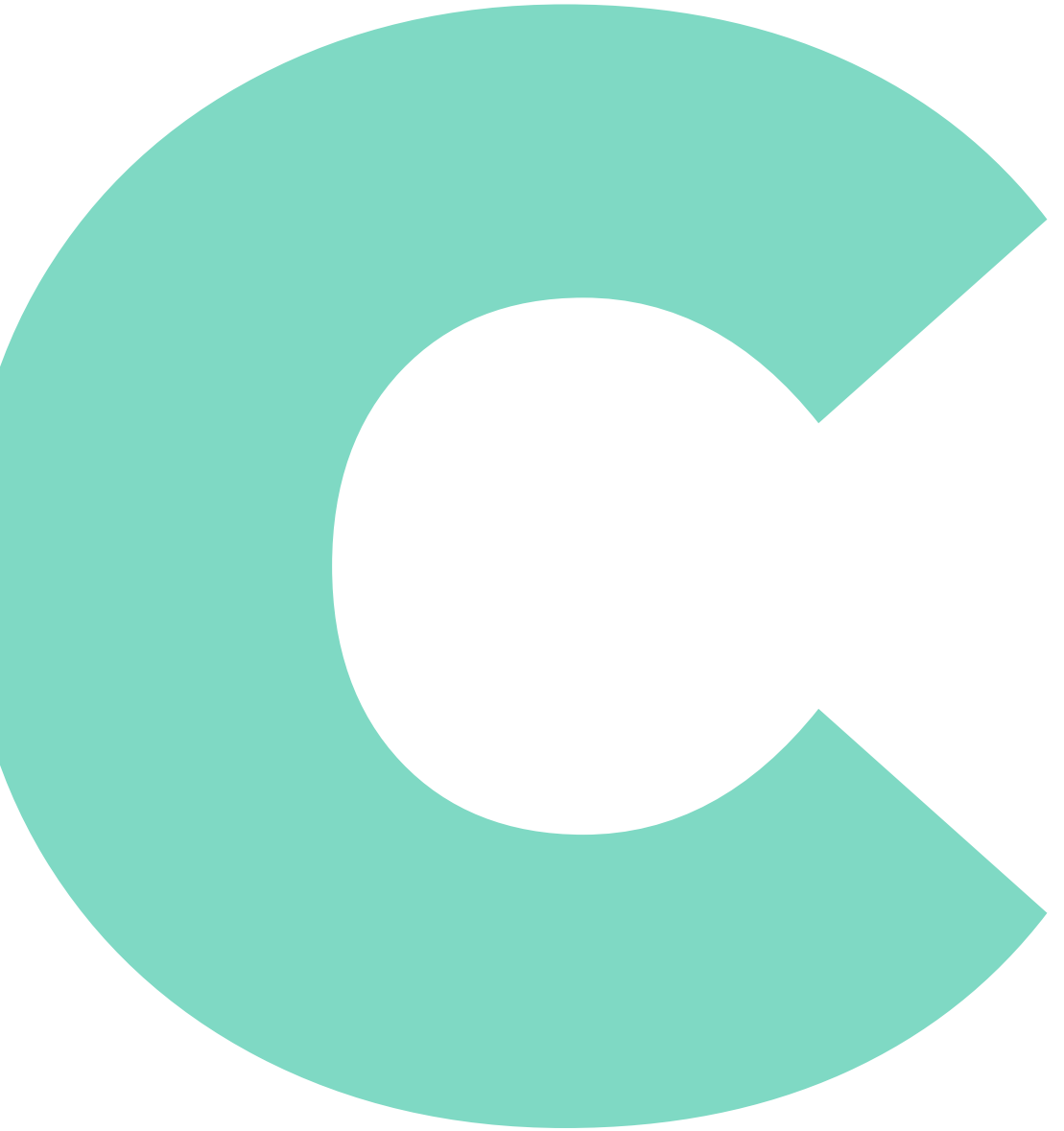
Spyder fonctionne de la même manière que Pycharm à la différence que Spyder a été uniquement pensé pour l'analyse de données et cela se ressent dans la façon dont est présenté l'IDE.
Vous avez donc d'un côté votre éditeur de texte, d'un autre votre console IPython mais vous avez en plus un lecteur de variables et une possibilité d'aide interactive très utile

L'avantage majeur de Spyder est donc sa clarté. Si vous commencez en Data Science, il est très agréable d'utiliser cet outil car vous allez comprendre rapidement ce qu'il se passe. Nous vous conseillons donc de commencer par cet outil si vous commencez votre carrière car vous allez pouvoir plus simplement débugger vos programmes et vous pourrez interagir plus facilement avec vos variables, algorithme etc.

ET ALORS POURQUOI CELA NE SERAIT PAS BIEN ?

Le désavantage de Spyder est la vitesse d'exécution du code. Si vous travaillez sur des algorithmes simples, il est peu probable que vous constatiez une différence. Si à l'inverse, vous commencez à utiliser des algorithmes de deep learning demandant beaucoup de mémoire, l'IDE va commencer à ralentir.

CONCLUSION



Nous avons couvert beaucoup de concepts dans ce manuel. Si vous en êtes arrivé à cette conclusion en ayant compris chacune de ces parties, vous avez alors acquis des bases solides dans la programmation du langage Python. Vous verrez que vous serez amené à pratiquer plus certains concepts que d'autres et qu'il est fortement probable que vous oubliez une partie de ce que vous avez appris. ***Ne vous inquiétez pas, c'est tout à fait normal.***

Dans le code, le but n'est pas d'apprendre par coeur puisqu'on peut tout copier / coller. Gardez ce livre comme une **référence** sur laquelle vous pouvez vous appuyer et revenir dès que vous rencontrez des problématiques qui vous donnent du fil à retordre. La pratique est encore et toujours le secret de l'apprentissage.

Vous pourriez aussi vous demander : "Maintenant qu'est ce que je fais avec nouvelles connaissances ?" et vous avez tout à fait raison.
C'est à ce moment là qu'il faut vous demander quel domaine dans l'informatique vous intéresserait. Le Web Development est parfait pour les personnes qui souhaitent avoir un pied dans les Computer Sciences. L'analyse de données est aussi un domaine très prometteur dans lequel Python excelle. Vous retrouverez des similitudes mais aussi quelques différences en termes de compétences dans les deux domaines. Les Data Sciences demanderont, par exemple, des connaissances statistiques et un esprit analytique plus fort alors que le Web Development demandera une connaissance des technologies liées au web comme d'autres langages : HTML, CSS, JavaScript.

Quelle que soit la voie que vous allez choisir, il est bon de commencer à pratiquer sur des projets concrets car c'est cela qui va vous permettre d'avancer et de garder votre motivation au plus haut. Appuyez vous sur les ressources dont vous disposez. Restez ouvert et curieux des choses qui se passent dans les technologies car c'est un milieu très mobile qui évolue très rapidement.

N'hésitez donc pas à continuer à vous former, que ce soit par le biais de MOOC, de bootcamps ou de livres. Vous ne perdrez jamais votre temps à apprendre des nouveaux concepts.

Enfin, nous terminerons ce livre en disant que quelque soit le milieu d'où vous

venez, l'apprentissage du code est compliqué et n'est pas inné. Beaucoup se disent "non, c'est trop compliqué pour moi" comme si les programmeurs étaient des mathématiciens au cerveau supérieur.

Ce n'est pas vrai, tout le monde peut apprendre les fondamentaux de la programmation. Beaucoup se disent aussi qu'il faut absolument un master en ingénierie pour pouvoir prétendre être codeur et que ce n'est pas en apprenant tout seul qu'on va pouvoir décrocher un poste. Ceci s'appelle le syndrôme de l'imposteur et il est ressenti chez beaucoup de personnes qui prennent des chemins de traverse pour se former. De fait, ceux-ci pensent qu'ils ne sont jamais "assez bons" pour être ce qu'ils veulent devenir. Souvenez vous que c'est la pratique et vos accomplissements qui feront que vous êtes assez bons et non pas un diplôme. **C'est votre expérience qui vous différenciera et qui fera de vous un expert.** Ce domaine n'en est pas une exception et, même si apprendre le code peut être frustrant parfois, il est tout à fait possible d'y entamer une carrière sans être passé par les meilleures écoles d'ingénieur, qui ne sont qu'un moyen de parvenir à cet objectif. Il en existe **beaucoup d'autres.**

Restez donc motivé et dites vous que, lorsque les choses deviennent difficiles, c'est à ce moment précis que vous apprenez.

SOLUTIONS

LES VARIABLES

UN NOMBRE CHOUETTE

```
pi = 1.43
pi = 3.14
print(pi)
>>>> 3.14
```

L'ARBRE AUX FRUITS MAGIQUES

```
fruits = 3
fruits_manges = 1
fruits_restant = fruits - fruits_manges
print(fruits_restant)
>>>> 2
```

MON CHIEN PRÉFÉRÉ

```
chien = "berge allemand"
chien = "husky"
print(chien)
>>>>
husky
```

UN ORDINATEUR POLI

```
merci = "Merci beaucoup"
prénom = "Antoine"
print(merci + prénom)
>>>> Merci beaucoup Antoine
```

PROJET : COMPLIMENT DU JOUR

```
prenom = input("quel est votre prénom ? ")
>>>> Quel est votre prénom ? Antoine
print("Vous êtes rayonnant aujourd'hui " + prenom)
>>>>
Vous êtes rayonnant aujourd'hui Antoine
```

TYPES DE DONNÉES

C'EST PAS LE NOMBRE DE DÉCIMALES QUI COMPTE

```
int = 2001291
float = 3.14
result = int + float
type(result)
>>>> <class 'float'>
```

On obtient donc un nombre de type float

DES CARACTÈRES DE NOMBRE ?

```
ecole = 42
type(ecole)
>>>> <class 'int'>
str = str(ecole)
type(str)
>>>> <class 'str'>
```

Ceci nous montre qu'on peut aussi avoir des chiffres considérés comme des caractères !

COLLECTER DES DONNÉES

Cette fois, plutôt que d'écrire du code, nous avons voulu prendre un peu de recul pour bien comprendre dans quel(s) cas utiliser nos données

1. Dans le premier cas, puisque nous savons que nous avons les coordonnées : Nom, Prénom, Email, Téléphone. Nous aurons donc plusieurs Noms, Prénoms, Emails et numéros de téléhone. De fait, nous pouvons associer une clé Nom aux différentes valeurs possibles que cette clé peut prendre et ainsi de suite. On va donc préférer utiliser un dictionnaire dans ce cas. Voici un exemple:

```
{"Nom" : ["Dupont", "Durant", "Dugenou"], "Prénom": ["Michel", "Myriam",
"Clément", ....], "Email":["michel@dupont.me", "myriam@durant.com",
"clement@gmail.com", "téléphone": [065949394, 068392320, 00391933,
078392194]}
```

2. Dans le second cas, nous avons plutôt des données que nous allons analyser et potentiellement changer. Autant donc utiliser une liste pour chaque colonne

```
colonne1 = [1,302,138,1039,103,...]
colonne2 = [131,20109,23819,201]
.
.
.
```

3. Enfin dans le troisième cas, si nous voulons être certain que chaque valeur est unique ; autant utiliser des sets.

```
id = set([0291D, 0210A, 2010A,...])
```

VIDE ET RIEN C'EST PAREIL ?

```
nul = 0
vide = None
nul == vide
>>>> False
```

On peut conclure que 0 et None ne sont pas la même chose. Zéro représente simplement un chiffre alors que None représente le vide.

PROJET : DES DONNÉES, DES DONNÉES, ENCORE DES DONNÉES

```
data = input("entrez votre donnée, nous vous en dirons son type")
>>>> entrez votre donnée, nous vous en dirons son type {"bonjour":"hello"}
print(type(data)
>>>> <class 'str'>
```

Ceci nous montre qu'on peut aussi avoir des chiffres considérés comme des caractères !

LES OPÉRATEURS EN PYTHON

L'ESTHÉTISME NE TIENT QU'À UN NOMBRE

```
x = 1/2
y = 1
z = 5 ** (1/2)
nb_or = x*(y+z)
print(nb_or)
>>>> 1.618033988749895
```

LES DALTONS

```
bill = input("Quelle est ta taille Bill ?")
grat = input("Quelle est ta taille Grat ?")

Quelle est ta taille Bill ?185
Quelle est ta taille Grat ?187

bill > grat
>>> false
```

```
bob = bill + 10
emmett = grat + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

On obtient une erreur car les valeurs pour Grat et Bill ont été stockées en tant que chaîne de caractères. Il faut qu'on change ce type de données.

```
bill = int(bill)
grat = int(grat)

bob = bill + 10
emmett = grat + 10

Cette fois tout fonctionne bien

bob != emmett
>>>> True
```

LUCKY LUKE EST PERDU

```
"o" in "bob"
>>>> True

"b" in ("bob" and "bill")
>>>> True
"a" in ("grat" or "emmett")
>>>> True
```

UN PEU DE POÉSIE

```
vers1 = "au bout de sa langue \n"
vers1 = vers1.capitalize()
```

```
vers2 = "il cache des paysages - \n"
vers2 = vers2.capitalize()

vers3 = "l'étranger."
vers3 = vers3.capitalize()

haiku = vers1 + vers2 + vers3
print(haiku)

>>>>
Au bout de sa langue
Il cache des paysages -
L'étranger.
```

Y A TROP DE CHIFFRES

```
x = 1/3
print("%2.2f" % x)
>>>> 0.33
```

PROJET : TEXT MINING

```
text = "Chaque Homme sur terre a un trésor qui l'attend, lui dit son coeur.
Nous, les coeurs, en parlons rarement, car les Hommes ne veulent plus
trouver ces trésors. Nous n'en parlons qu'aux petits enfants. Ensuite, nous
laissons la vie se charger de conduire chacun vers son destin.
Malheureusement, peu d'Hommes suivent le chemin qui leur est tracé, et qui
est le chemin de la Légende Personnelle et de la félicité. La plupart
voient le monde comme quelque chose de menaçant et, pour cette raison même,
le monde devient en effet une chose menaçante."

text = text.lower()

separation = text.split(",")

mots_positifs = separation[0]
mots_positifs.find("trésor")
```

```
>>>> 28
mot_positif = mots_positifs[28:34]

print(mot_positif)
>>>> 'trésor'
```

CONDITIONS ET BOUCLES

AVEC DES SI, ON POURRAIT METTRE PARIS EN BOUTEILLE

```
question1 = input("Combien de fois la France a gagné la coupe du monde ?")
if question1 == "2":
    print("Bravo ! C'est la bonne réponse")
else:
    print("Dommage, tente une autre question")

question2 = input("Quand est-ce qu'a été fondé Apple ?")

if question2 == "1976":
    print("Bravo ! C'est la bonne réponse")
else:
    print("Dommage, tente une autre question")

question3 = input("Combien de vie(s) possède un chat ?")

if question3 == "9":
    print("Bravo ! C'est la bonne réponse")
else:
    print("Dommage, tente une autre question")
```

UN QUIZ PLUS COMPLIQUÉ

```
question1 = input("Combien de fois la France a gagné la coupe du monde ?")
```

```
while question1 != "2":
    print("Dommage ! Retente ta chance")
    question1 = input("Combien de fois la France a gagné la coupe du monde
?")

print("Bravo ! C'est la bonne réponse")
```

PROJET : UN QUIZ VRAIMENT MIEUX

```
nb_de_chances = 3

print("Voici notre quiz, tu as trois chances !")
question1= input("Combien de fois la France a gagné la coupe du monde ?")

while question1 != "2":
    if nb_de_chances == 0:
        print("Oh non ! Tu as perdu le jeu...")
        break
    else:
        nb_de_chances -= 1
        print("Dommage ! Il te reste {} chances".format(nb_de_chances))
        question1 = input("Combien de fois la France a gagné la coupe du
monde ?")

if nb_de_chances > 0:
    question2 = input("Quand a été fondé Apple")
    while question2 != "1976":
        if nb_de_chances == 0:
            print("Oh non ! Tu as perdu le jeu...")
            break
        else:
            nb_de_chances -=1
            print("Dommage ! Il te reste {} chances".format(nb_de_chances))
            question2 = input("Quand a été fondé Apple ?")
```

```
if nb_de_chances >0:
    question3 = input("Qui a fondé SpaceX")
    question3 = question3.lower()
    while question3 != "elon musk":
        if nb_de_chances == 0:
            print("Oh non ! Tu as perdu le jeu...")
            break
        else:
            nb_de_chances -=1
            print("Dommage ! Il te reste {} chances".format(nb_de_chances))
            question3 = input("Qui a fondé SpaceX")
            question3 = question3.lower()
    if question3 == "elon musk":
        print("Bravo ! Tu as gagné le quiz !")
```

Expliquons un peu le code ici. Procédons par étape :

1. Nous avons d'abord définit une variable *nb_de_chances* à laquelle nous avons assigné 3 chances
2. Ensuite, nous présentons notre quiz, d'où le premier *print()*
3. Ensuite pour la première question, nous reprenons l'idée du code que nous avions utilisé pour construire le quiz précédent. La seule chose étant que si l'utilisateur se trompe, il faut que celui-ci perde une chance et que nous affichions le nombre chances restantes et que nous reposions la question.
4. Cependant, pour que l'utilisateur puisse rejouer, il faut s'assurer qu'il lui reste assez de chances. S'il n'en a plus, il faut qu'il ait perdu le jeu. D'où le fait d'utiliser une condition à l'intérieur de la *boucle while* où nous vérifions que le nombre de chances de l'utilisateur n'est pas égal à 0, sinon on sort de la boucle et notre programme annonce à l'utilisateur que malheureusement, il a perdu le jeu.
5. On peut reprendre cette même structure pour les autres questions. Si ce n'est, qu'il faut prendre quelque chose de plus en compte.
6. Si l'utilisateur s'est trompé et n'a plus de chances dès les premières questions, il ne faut pas que le programme lui repose une nouvelle question. C'est pour cela que nous allons utiliser une nouvelle condition vérifiant que l'utilisateur possède au moins 1 chance. Sinon, on ne pose pas la question.
7. Un dernier conseil. Pour la troisième question, vous voyez que nous avons utilisé la fonction *.lower()* pour mettre question3 en minuscule. Ceci est une technique

très utilisée pour prévenir le fait qu'un utilisateur puisse écrire "Elon Musk" ou "ELON MUSK" ou encore "elon Musk". Ces trois réponses sont valides mais si nous avions laissé "elon musk" dans notre condition, notre programme aurait considéré que tout autre façon d'écrire Elon Musk n'aurait pas été bonne.

Si vous avez réussi à arriver jusqu'ici, bravo ! Ce projet n'est pas facile à mettre en place. Si vous n'avez pas le code qu'il y a au dessus mais que tout fonctionne, bravo tout de même. Il n'y a pas qu'une seule façon d'arriver au même résultat.

FONCTIONS

VOILÀ COMMENT ÇA FONCTIONNE

```
def sqrt(nombre):
    nombre = nombre ** (1/2)
    return nombre

sqrt(9)
>>>> 3.0
```

PREMIÈRE OU SECONDE CLASSE ?

```
def repas(classe = "seconde_classe"):
    if classe == "seconde_classe":
        print("Pour obtenir votre repas, vous devrez aller en wagon-bar qui
se situe voiture 14")
    elif classe == "premiere_classe":
        print("Un repas vous sera servi à votre place, bon voyage")
    else:
        print("Si vous n'êtes ni en première, ni en seconde classe, êtes
vous sûre d'avoir pris votre billet ?")
```

EXCEPTIONNELLEMENT POUR VOUS

```
def acheter_son_billet():
    try:
        nb_de_billets = input("Bonjour, combien de billets souhaitez vous
acheter ?")
        nb_de_billets = int(nb_de_billets)

        if nb_de_billets < 1:
            raise ValueError("Vous devez prendre au moins un billet")

        if nb_de_billets > 14:
            raise ValueError("Vous avez pris beaucoup de billets, vous
devriez appeler notre service pour bénéficier d'un tarif de groupe")

    except ValueError as erreur:
        print("Ne mettez uniquement un nombre rond s'il vous plait \n")

    else:
        prix_total = 25 * nb_de_billets
        print("Le prix total de votre commande est de
{}".format(prix_total))

acheter_son_billet()
```

PROJET : COMBIEN VOUS RAPPORTE LE LIVRET A ?

```
def interet():
    try:

        total = 0

        somme = float(input("Donnez-nous la somme totale que vous souhaitez
```

```
        if somme < 0:
            raise ValueError("Vous ne pouvez pas investir un nombre négatif
d'argent")
        elif somme == 0:
            raise ValueError("Vous n'avez pas investi d'argent !")

        nb_annees = int(input("combien d'années allez-vous placer cet
argent ?\n"))

        if nb_annees < 0:
            raise ValueError("Les années ne peuvent pas prendre une valeur
négative")
        elif nb_annees == 0:
            raise ValueError("Vous allez bien attendre au moins un an, non
?")

        interet = input("A quel taux souhaitez vous voir les intérêts ?\n
ATTENTION : Vous devrez mettre une valeur décimale \n Ex: 10% --> 0.10\n")

        if "%" in interet:
            raise ValueError("Merci de mettre une valeur décimale
uniquement, pas de signe de pourcentage \nEx 10% ---> 0.10")

        else:
            interet = float(interet)

        if interet < 0:
            raise ValueError("Vous ne pouvez pas avoir un taux d'intérêt
négatif")
        elif interet == 0:
            raise ValueError("Vous devriez avoir un taux d'intérêt")

        total = somme*(1+interet)**(nb_annees)

    except ValueError as erreur:
        print("La valeur insérée n'est pas valide \n")
        print(erreur)
```

```
    else:
        print("La somme totale dont vous disposerez après avoir déposé {}
au bout de {} ans sera de {:.2f}".format(somme,nb_annees, total))


interet()
```

Dans ce projet, toute la difficulté est de trouver où lever les exceptions dans la fonction *interet()*. En effet, on veut par exemple qu'une erreur arrive à la moindre faute entrée par l'utilisateur. D'où le fait qu'on ajoute des conditions de vérification après chaque *input().* On vérifie aussi que la personne mette un nombre décimal plutôt qu'un taux d'intérêt exprimé en pourcentage.

Pour terminer, on formate les nombres pour n'obtenir que deux chiffres après la virgule.

MANIPULATION AVANCÉE DES COLLECTIONS DE DONNÉES

LISTE DES COURSES

```
semaine1 = ["Salade", "Sel", "Poivre", "Viande", "Pâtes "]
semaine1.append("Déodorant")
semaine1[0] = "Quinoa"
semaine2 = semaine1.pop(-2)
```

UNE LISTE PAIRE

```
list = [1,2,3,4,5,6,7,8,9,10]
list_paire = list[1::2]
del(list_paire[1:3])
list_inverse = list_paire[::-1]
```

UN PEU PLUS DE TEXT MINING

```
def frequence_de_mots(phrase):
    phrase = phrase.lower()
    liste = phrase.split(" ")
    dic = {}
    for item in liste:
        dic[item] = liste.count(item)
    return dic
```

NB : Attention, nous n'avons pas mis de guillemet dans dic[item] ni dans liste. count(item) car, dans le cas inverse, nous aurions la chaîne de caractères "item" au lieu de l'item présent dans liste.

CALCULER UNE MOYENNE AVEC DES *ARGS

```
def moyenne(*chiffres):
    somme = 0
    nb_de_chiffres = 0
    for chiffre in chiffres:
        somme += chiffre
        nb_de_chiffres += 1

    moyenne = somme/nb de chiffres
    return moyenne
```

QU'EST CE QU'IL FAUT POUR DEVENIR DATA SCIENTIST ?

```
def devenir_data_scientist(set):
    competences_a_avoir = {"SQL", "Python", "Machine Learning",
"Statistiques", "Big Data", "A/B testing", "Excel", "Cloud Computing",
"Deep Learning", "Spark", "Hadoop"}
    competences_en_commun = set & competences_a_avoir
    format = ""
    for competence in competences_en_commun:
        competences_formatees += "{}, ".format(competence)
```

```
    return print("Vous avez déjà {} comme compétences nécessaires pour
devenir Data Scientists, continuez comme ca
!".format(competences_formatees))
```

PROJET : TO-DO LIST

```
todo = []
quit_program = ""

def show_todo():
    print("\nVoici votre to-do du jour \n ------- \n")
    for item in todo:
        print(item)
    print("\n")

def ajout_item(todo):
    item = input("Que devez vous faire ? \n Si vous devez ajouter plusieurs
todo à la fois, séparez les par une virgule")
    new_items = item.split(",")
    for item in new_items:
        todo += [item]
    return todo

def enlever_item(todo):
    item = input("Qu'avez vous fait ? \n Si vous avez fait plusieurs todo
to à la fois, séparez les par une virgule")
    remove_items = item.split(",")
    for item in remove_items:
        todo.remove(item)
    return todo

while quit_program != "oui":

    show_todo()

    ajouter = input("Voulez vous ajouter un item --> Ecrivez OUI ou NON
\n")
```

```
    ajouter = ajouter.lower()

    if ajouter == "oui":
        ajout_item(todo)
        show_todo()
    elif ajouter == "non":
        enlever = input("Voulez vous enlever un item --> Ecrivez OUI ou NON
\n")
        enlever = enlever.lower()
        if enlever == "oui":
            enlever_item(todo)
            show_todo()
        else:
            print("Okay, votre to-do reste la suivante \n {}".format(todo))
    else:
        print("Je ne suis pas sur d'avoir compris")

    quit_program = input("Souhaitez)vous sortir de l'application ? Tapez
OUI pour sortir ou NON pour revoir votre TODO")
    quit_program = quit_program.lower()
```

Voici un programme pas facile à faire tourner. Il est bon de prendre le problème par étape :

1. On écrit d'abord le coeur du programme. A savoir : demander à l'utilisateur d'ajouter, d'enlever des éléments de sa todo mais aussi de la présenter. On utilisera une boucle *while* pour ne pas sortir du programme tant que l'utilisateur ne l'a pas décidé. Car dans le cas inverse, nous perdrions toutes nos todos à chaque fin de programme.
2. Maintenant, nous allons écrire des fonctions que nous allons appeler à chaque que l'utilisateur fait une action. C'est à dire dès qu'il a besoin d'ajouter un élément dans sa todo, nous allons utiliser la fonction *ajout_item()*. Dès qu'il aura besoin d'enlever un élément, on utilisera *enlever_item()*. Pour finir, on présentera la todo avec la fonction *show_todo()*
3. Puisque nous utilisons la fonction *input(),* il faut être certain que l'utilisateur mette les bonnes valeurs (OUI / NON) sinon, il peut faire bugger l'application. D'où le fait d'intégrer cette fonctionnalité dans votre comme dernier else.

Si vous avez réussi à coder ce programme, bravo ! Vous avez vraiment bien compris comment fonctionne les listes, les fonctions, les boucles et les conditions ! Si vous n'avez pas réussi, ce n'est pas grave. Il ne faut pas se décourager. Lisez bien le code et tenter de comprendre chacune des étapes. Si vous souhaitez améliorer ce programme, vous pouvez penser à mieux gérer les exceptions dans vos fonctions pour encore mieux guider votre utilisateur.

UTILISER DES LIBRAIRIES EN PYTHON

RAPPEL DE TÂCHES

PARTIE 1

```
import datetime

def rappel():
    print("N'oublie pas le Tennis !")

aujourdhui = datetime.datetime.today()
tennis = datetime.datetime(2018, 7, 7, 15,30,00)

if aujourdhui + timedelta(0,0,0,0,0,3) == tennis:
    rappel()
```

PARTIE 2

```
import datetime

def rappel():
    print("N'oublie pas le Tennis !")

aujourdhui = datetime.datetime.today()
first_tennis = datetime.datetime(2018, 7, 7, 15,30,00)
last_tennis = datetime.datetime(2018, 7, 28, 15,30,00)

while first_tennis < last_tennis:
    first_tennis += timedelta(7)
    if aujourdhui + timedelta(0,0,0,0,0,3) == first_tennis:
        rappel()
```

On ne peut pas itérer avec des boucles for sur les datetime. En revanche, en donnant une date de début et une date de fin, on peut itérer avec une boucle while.

Le reste du code reste le même que dans la partie 1.

LISTE D'INVITÉS

```
invites= input("Qui sont les invités de la soirée ? Ecrivez Prénoms Noms et
séparez chaque invité par une virgule")
liste_invites = invites.split(",")
liste_invites.sort()

import csv
with open("invites.csv", "w") as csvfile:
    notewriter = csv.writer(csvfile, delimiter=",")
    notewriter.writerow(["Invites"])
    for invite in liste_invites:
        notewriter.writerow([invite])
```

Rien de nouveau lorsqu'il s'agit de préparer les noms et prénoms et les ordonner. En revanche pour écrire le fichier csv, nous avons décidé d'effacer tous les anciens noms à chaque fois que nous exportons le fichier pour que cela nous évite d'avoir des doublons.

Ensuite, il faudra bien faire attention à mettre les invités comme des listes. Sinon le programme va séparer chaque caractère de votre chaîne pour donner quelque chose comme : "A,n,t,o,i,n,e, ,K,r,a,j,n,c".

PROJET : UNE MEILLEURE TODO

```
import csv
todo= []
quit_program = ""
```

```
with open("todo.csv") as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        todo+=[row["todo"]]

def show_todo():
    print("\nVoici votre to-do du jour \n ------- \n")
    for item in todo:
        print(item)
    print("\n")

def ajout_item(todo):
    item = input("Que devez vous faire ? \n Si vous devez ajouter plusieurs
todo à la fois, séparez les par une virgule")
    new_items = item.split(",")
    for item in new_items:
        todo += [item]
    return todo

def enlever_item(todo):
    item = input("Qu'avez vous fait ? \n Si vous avez fait plusieurs todo
to à la fois, séparez les par une virgule")
    remove_items = item.split(",")
    for item in remove_items:
        todo.remove(item)
    return todo

while quit_program != "oui":

    show_todo()

    ajouter = input("Voulez vous ajouter un item --> Ecrivez OUI ou NON
\n")
    ajouter = ajouter.lower()

    if ajouter == "oui":
        ajout_item(todo)
        show_todo()
```

```
    elif ajouter == "non":
        enlever = input("Voulez vous enlever un item --> Ecrivez OUI ou NON
\n")
        enlever = enlever.lower()
        if enlever == "oui":
            enlever_item(todo)
            show_todo()
        else:
            print("Okay, votre to-do reste la suivante \n")
            show_todo()
    else:
        print("Je ne suis pas sur d'avoir compris")

    quit_program = input("Souhaitez)vous sortir de l'application ? Tapez
OUI pour sortir ou NON pour revoir votre TODO")
    quit_program = quit_program.lower()
    if quit_program == "oui":
        with open("todo.csv", "w") as csvfile:
            write = csv.writer(csvfile, delimiter=',')
            write.writerow(["todo"])
            for item in todo:
                write.writerow([item])
```

Le code principal de notre première todo reste inchangé. En revanche, il fallait ajouter la possibilité de lire le fichier todo.csv pour pouvoir modifier la variable todo directement et que notre programme puisse reprendre là où nous nous étions arrêté.

Ensuite, une fois que nous avons fait nos modifications, il faut que le programme réécrive les nouvelles données dans le csv. C'est pour cela que nous l'ajoutons à la fin.

LA PROGRAMMATION ORIENTÉE OBJET

GÉRER SON BAR

```
class bar:
    employes = 10
    surface = 100
    nb_max_de_personnes = 100


bar = bar()
print("le nombre d'employés du bar est {}".format(bar.employes))
print("la surface du bar est de {} m2".format(bar.surface))
print("le bar peut contenir un maximum de {}
personnes".format(bar.nb_max_de_personnes))
```

DES MATHS FACILES

```
class math:
    def sqrt(self,nombre):
        nombre = nombre ** (1/2)

        print(nombre)

    def moyenne(self,*chiffres):
        somme = 0
        nb_de_chiffres = 0
        for chiffre in chiffres:
            somme += chiffre
            nb_de_chiffres += 1

        moyenne = somme/nb_de_chiffres
        print(moyenne)

    def parite(self,nombre):
        if type(nombre/2) == 'int':
            print("{} est pair".format(nombre))
        else:
            print("{} est impair".format(nombre))
```

```
    def sum(self,*chiffres):
        somme = 0
        for chiffre in chiffres:
            somme += chiffre
        print(somme)


math = math()
math.sqrt(9)
math.moyenne(10,30,20,10)
math.parite(4)
math.parite(10.3)
math.parite(11)
math.sum(10,30,10,40,20)
```

DEVINETTE

```
import random

class devinette():

    def __init__(self):
        self.essais = 3
        self.nombre_a_deviner = random.randint(1,10)

    def deviner(self):
        nb_joueur = input("Devinez le nombre que j'ai choisi. \n Il est
compris entre 1 et 10\n")
        nb_joueur = int(nb_joueur)
        return nb_joueur

    def rejouer(self):
        rejouer = input("Voulez-vous rejouer ? OUI / NON")
        rejouer = rejouer.lower()
        while True:
            if rejouer == "oui" or rejouer == "non":
                break
```

```
            else:
                print("Je n'ai pas bien compris \n")
                rejouer = input("Voulez-vous rejouer ? OUI / NON")
                rejouer = rejouer.lower()
        return rejouer

devinette = devinette()
nb_joueur = devinette.deviner()
nb_essais = devinette.essais

while True:
    if nb_essais > 0:
        if nb_joueur > devinette.nombre_a_deviner:
            print("Le nombre à deviner est plus petit")
            nb_essais -= 1

        elif nb_joueur < devinette.nombre_a_deviner:
            print("Le nombre à deviner est plus grand")
            nb_essais -= 1

        elif nb_joueur == devinette.nombre_a_deviner:
            print("Bravooo, vous avez trouvé le nombre")
            rejouer = devinette.rejouer()
            if rejouer == "oui":
                devinette = devinette()
                nb_joueur = devinette.deviner()
                nb_essais = devinette.essais
            elif rejouer == "non":
                break

        print("Il vous reste {} essais".format(nb_essais))
        nb_joueur = devinette.deviner()

    elif nb_essais == 0:
        print("Dommage vous n'avez plus d'essais....\n")
        rejouer = devinette.rejouer()
        if rejouer == "oui":
            devinette = devinette()
```

```
            nb_joueur = devinette.deviner()
            nb_essais = devinette.essais
        elif rejouer == "non":
            break
```

ANNEXES

LES COMMANDES PRINCIPALES DE VOTRE CONSOLE

```
$ mkdir
$ cp
$ mv
$ rm
$ rmdir
$ pwd
$ clear
$ sudo
$ touch
```

$ touch

La commande touch vous sert à créer un fichier.
Vous pourrez donc faire:

```
$ touch le_nom_dun_fichier.son_extension
```

Prenons un exemple concret :

```
$ touch fichier_1.py
```

Avec cette commande, nous venons de créer un fichier nommé *fichier_1.py*.

$ mkdir

Cette commande vous permet de créer un nouveau dossier sur votre ordinateur.

```
$ mkdir nom_de_votre_dossier
```

Voici un exemple :

```
$ cd Downloads
$ mkdir manuel
```

Si nous allons maintenant dans notre dossier Téléchargements, nous voyons un dossier manuel dans lequel nous pouvons mettre des fichiers.

$ cp

Vous pouvez aussi copier des fichiers avec la commande *cp.*

```
$ cp tablestyling.html tablestyling2.html
```

Vous devrez donner un nom à votre nouveau fichier car on ne peut pas avoir deux fichiers avec le même nom. C'est pour cela que dans cet exemple, nous avons renommé *tablestyling.html* en *tablestyling2.html*.

On peut aussi copier et déplacer le fichier dans un autre dossier directement

```
$ cp tablestyling.html /Users/antoinekrainc/Downloads/tablestyling2.html
```

$ mv

A la différence de cp, mv vous permet simplement de déplacer ou renommer un fichier sans le copier

```
$ mv tablestyling.html /Users/antoinekrainc/Documents/
```

Dans cette exemple, nous avons déplacé le fichier *tablestyling.html* dans le dossier *v*.

De manière générale voici la structure

```
$ mv nom_de_votre_fichier.son_extension
chemin_vers_lequel_vous_voulez_déplacer_votre_fichier
```

Vous pouvez aussi utiliser mv pour renommer votre fichier

```
$ mv tablestyling.html stylingtable.html
```

Ici, nous avons donc renommé mon fichier *tablestyling.html* en s*tylingtable.html*

Vous pouvez même maintenant déplacer et renommer votre fichier en précisant le nom du nouveau fichier dans votre chemin

```
$ mv stylingtable.html /Users/antoinekrainc/Downloads/tablestyling.html
```

Ici nous avons déplacé le fichier *stylingtable.html* dans notre dossier Téléchargements et nous l'avons renommé à son d'origine tablestyling.html

$ rm

Maintenant qu'on sait créer et déplacer des fichiers et dossier, on peut aussi les supprimer. Pour supprimer des fichiers, on utilisera la commande *rm.*

```
$ rm tablestyling2.html
```

ATTENTION : utiliser la commande rm ne déplace pas votre fichier dans la corbeille, elle le supprime définitivement de votre ordinateur donc soyez certain que vous ne voulez plus de ce fichier.

$ rmdir

On peut faire la même chose avec les dossiers

```
$ rmdir dossier_a_enlever
```

ATTENTION : ce qui vaut pour rm vaut aussi pour rmdir, vous ne pourrez pas récupérer votre dossier dans la corbeille une fois effacé avec cette commande.

$ pwd

Il peut être utile de connaître le chemin entier dans lequel votre dossier se

trouve. On peut utiliser la commande pwd pour cela

```
$ cd Dropbox/Data_Sciences/
$ pwd
/Users/antoinekrainc/Dropbox/Data_Sciences
```

$ clear

Si vous commencez à écrire beaucoup de code dans votre console et que cela devient illisible, vous pouvez faire un peu de ménage avec la commande clear

$ sudo

Parfois, votre ordinateur ne vous laissera pas faire tout ce que vous voulez avec votre console car il veut être certain que vous n'êtes pas un novice qui ne sait pas ce qu'il fait. Si cela vous arrive et que vous avez effectivement besoin d'exécuter votre commande, vous aurez à la faire précéder de la commande sudo. Votre console vous demandera aussi le mot de passe que vous utilisez lorsque vous allumez votre machine.

Bonus : Faire dire n'importe quoi à votre ordinateur

Si vous êtes sur mac, vous pouvez utiliser la commande say pour faire dire ce que vous voulez à votre mac

```
$ say bonjour je suis gentil
```

Il existe beaucoup d'autres commande sur votre console mais celles-ci sont les principales à maîtriser.

LISTE DES ERREURS CLASSIQUES

Il existe beaucoup d'exceptions possibles en Python, voici une liste qui va vous permettre d'y voir plus clair et d'éventuellement vous aider dans le débuggage de vos programmes.

AttributeError

Si vous utilisez un attribut qui n'existe pas, vous aurez des chances d'obtenir cette erreur.

```
class hello:     prenom = "michel"
     def bonjour(self):
             print("bonjour {}".format(prenom))

hello = hello()
hello.test
>>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'hello' object has no attribute 'test'
```

Comme vous pouvez le voir dans notre classe hello, nous n'avons pas créé d'attribut test d'où l'erreur *AttributeError.*

ImportError

Il se peut que vous vous trompiez dans l'importation de vos modules. Si tel est le cas, vous tomberez sur un ImportError

```
from csv import Michel
>>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'Michel'
```

Ici nous n'avons pas de classe qui s'appelle Michel. D'où l'erreur

I ndentationError

Il n'est pas rare de lever ce genre d'erreur en Python. Elle correspond au fait que vous n'ayez pas respecté le bon espacement lorsque vous avez défini une fonction, une classe ou une boucle.

```
class hello:
prenom = 'michel'
def bonjour(self):

IndentationError: expected an indented block
```

Ici nous n'avons pas mis d'espace après la classe *hello*.

IndexError

Si vous essayez de faire trop d'itérations sur une boucle par rapport à la taille de votre objet.

```
list = [1,2,3]
for i in range(4):
    print(list[i])

>>>>
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

KeyError

Si vous avez un dictionnaire de données et que vous utilisez une clé qui n'est pas présente dans le dictionnaire, vous leverez une *KeyError.*

```
dic = {"prenom":"Adele", "nom":"Blanchet"}
dic["Age"]
>>>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Age'
```

Ici dans la variable dic, nous n'avons pas de clé Age d'où l'erreur.

NameError

NameError arrive très souvent lorsque vous utilisez des variables ou fonctions que vous n'avez pas définies auparavant.

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

Ici, nous n'avions jamais défini la variable prenom auparavant d'où le fait que nous ayions un *NameError.*

SyntaxError

Tout programmeur commence sa carrière en faisant beaucoup de ce genre d'erreurs. Elle arrive lorsque vous n'avez pas respecté la "grammaire" d'un langage

```
for i range(0):
  File "<stdin>", line 1
    for i range(0):
              ^
SyntaxError: invalid syntax
```

Ici nous avons simplement oublié d'écrire in dans for i range(0). D'où le fait que nous ayons une erreur de syntaxe.

TypeError

TypeError arrive si vous faites des opérations avec des types de données qui ne vont pas ensemble .

```
2 + 'hello'
>>>>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

UnicodeEncodeError

Cette erreur arrive souvent lorsque vous utilisez des caractères spéciaux que votre programme ne comprend pas. Ou alors, que vous essayez d'écrire des fichiers avec un mauvais encodage.

```
'hello ç'.encode("ascii")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xe7' in position
6: ordinal not in range(128)
```

Ici, nous avons fait exprès d'encoder notre chaîne de caractères contenant le ç pour lever cette erreur d'encodage. Mais vous saurez maintenant où est le problème si vous voyez ce type d'erreur arriver.

NB : La petite soeur de cette erreur UnicodeDecodeError. Au lieu de faire une erreur dans votre encodage, cette fois vous avez des problèmes dans votre décodage. Cela arrive très souvent lorsque vous souhaitez ouvrir un fichier qui n'est pas au bon format.

ValueError

A la différence de *TypeError,* ValueError arrive lorsque vos données sont du bon type mais ne peuvent pas exister dans votre fonction.

```
def password():
    password = input("entrez votre mot de passe")
    password = int(password)

password()
entrez votre mot de passe hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in password
ValueError: invalid literal for int() with base 10: 'hello'
```

ZeroDivisionError

Cette erreur arrive lorsque vous tentez de diviser un nombre 0

```
3/0
>>> ZeroDivisionError: division by zero
```