


Branch: master ▾

Find fileCopy path

data-wrangling-and-validation / Leçon 1-2-1 - Introduction à Jupyter Notebook et à la programmation en Python.ipynb

 **turukawa** Minor restructuring of Lesson 1.2.1
47f4628 6 days ago

1 contributor

<>

📄

RawBlameHistory

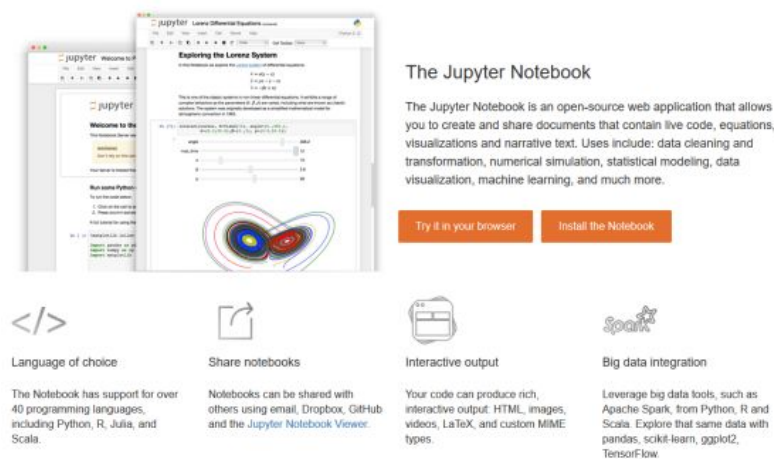
2115 lines (2115 sloc) 95.2 KB

2.1 Introduction à Jupyter Notebook et à la programmation en Python

A la fin de la formation, vous pourrez:

- Télécharger et installez Anaconda avec Jupyter Notebook.
- Ouvrir votre premier document Jupyter Notebook.
- Apprendre le contrôle des versions et la gestion du code avec Git et GitHub.
- Ecrire votre première ligne de code.
- Apprendre et appliquer les variables, types et opérateurs Python de base.
- Employer les variables, les types et les opérateurs dans des boucles et des conditions pour effectuer des tâches simples.
- Capturer et répondre efficacement aux exceptions.
- Utiliser des ensembles pour extraire des données uniques des listes.
- Développer et utiliser du code réutilisable en encapsulant les tâches dans des fonctions.
- Rassembler les fonctions en classes flexibles et extensibles.

2.1.1 Installation d'un ordinateur portable Python et Jupyter avec Anaconda

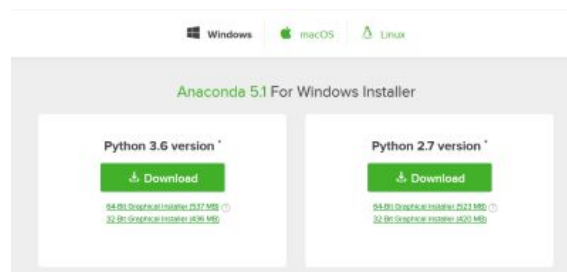


Anaconda est une suite de progiciels de sciences des données populaires pour Python et comprend Jupyter Notebook qui est un carnet interactif vous permettant de créer et de partager des documents contenant du code vivant. Il s'agit d'un Jupyter Notebook et vous pourrez suivre l'apprentissage au fur et à mesure.

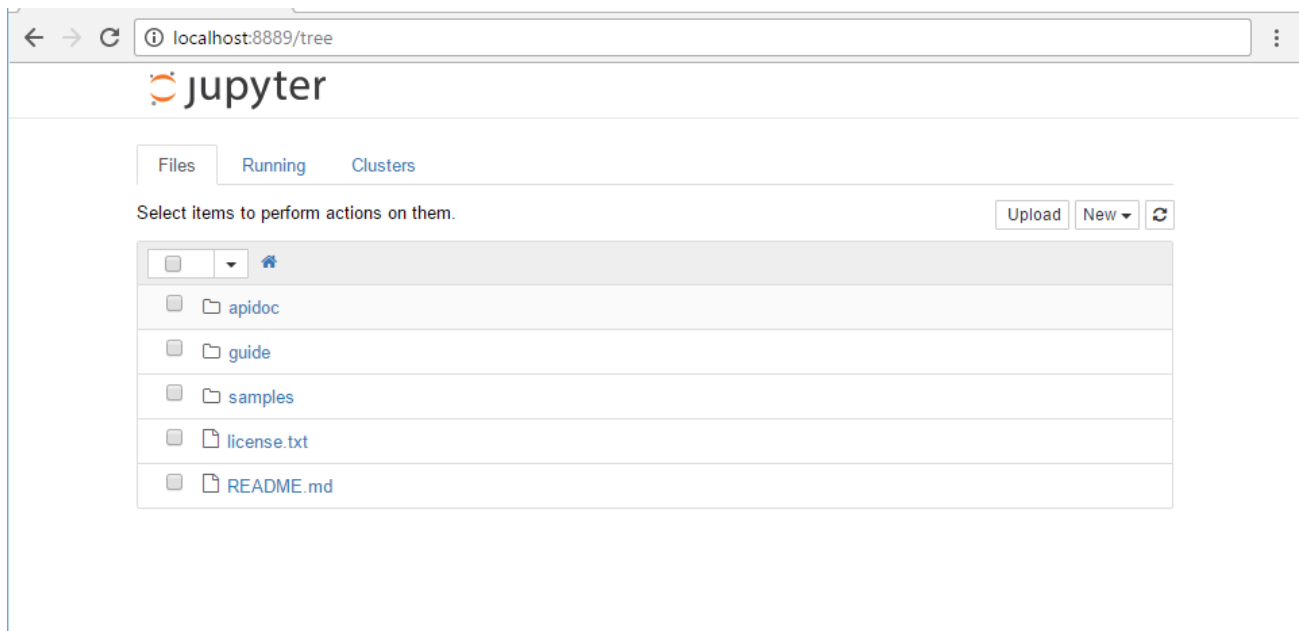
Notre premier objectif est d'installer Anaconda et de créer un Jupyter Notebook afin que vous puissiez démarrer ce tutoriel. Si vous voulez l'essayer dans un navigateur (y compris son propre tutoriel), alors [suivez ce lien \(https://jupyter.org/try\)](https://jupyter.org/try).

Pour installer Anaconda, et tous les paquets Python afin que vous puissiez travailler sur votre propre ordinateur:

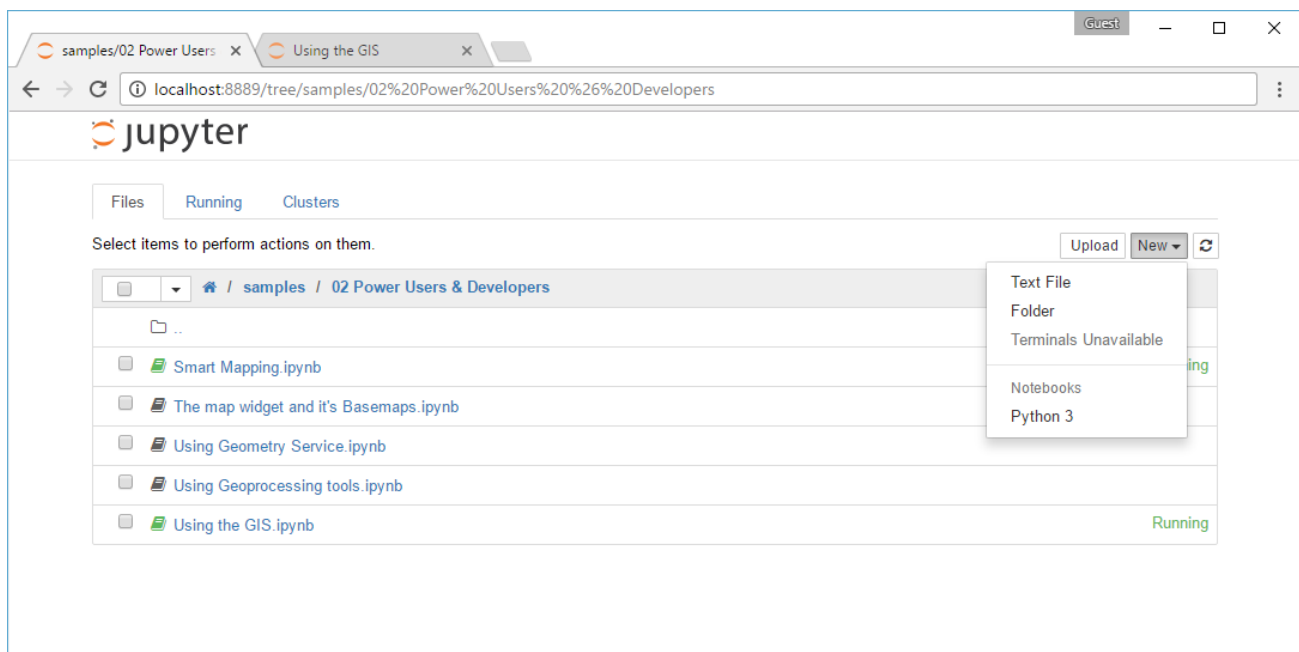
- Allez à la [page de téléchargement sur Anaconda \(https://www.anaconda.com/download/\)](https://www.anaconda.com/download/);



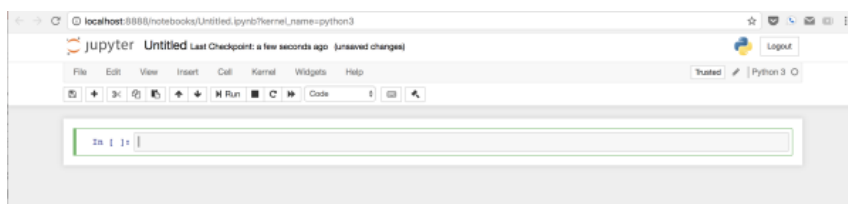
- Choisissez la version **Python 3.x** pour l'installation. Choisissez le paquet approprié pour votre ordinateur (Windows, MacOS ou Linux). Si vous êtes sous Windows, vous devrez également choisir entre 64 bits ou 32 bits. La plupart des ordinateurs supportent actuellement le 64 bits mais, si vous n'êtes pas sûr, choisissez la version 32 bits.
- Après le téléchargement, exécutez le logiciel et installez tous les paramètres par défaut.
- Vous verrez que vous avez la possibilité de lancer Anaconda, ou Jupyter Notebook. Pour l'instant, nous allons passer directement à Jupyter Notebook.
- La première page qui s'ouvre dans la vue du dossier (ou de l'arbre) de Jupyter Notebook. Ici, vous pouvez voir une liste de fichiers. Les Jupyter Notebooks se terminent par l'extension ".ipynb".



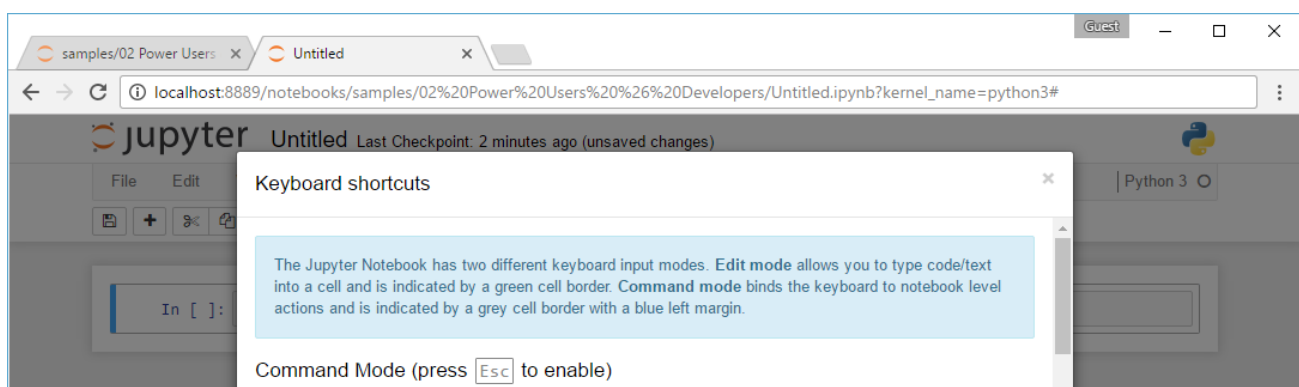
- Créez un nouveau bloc-notes en cliquant sur "New" et sélectionnez "Python 3" comme noyau que vous souhaitez exécuter.

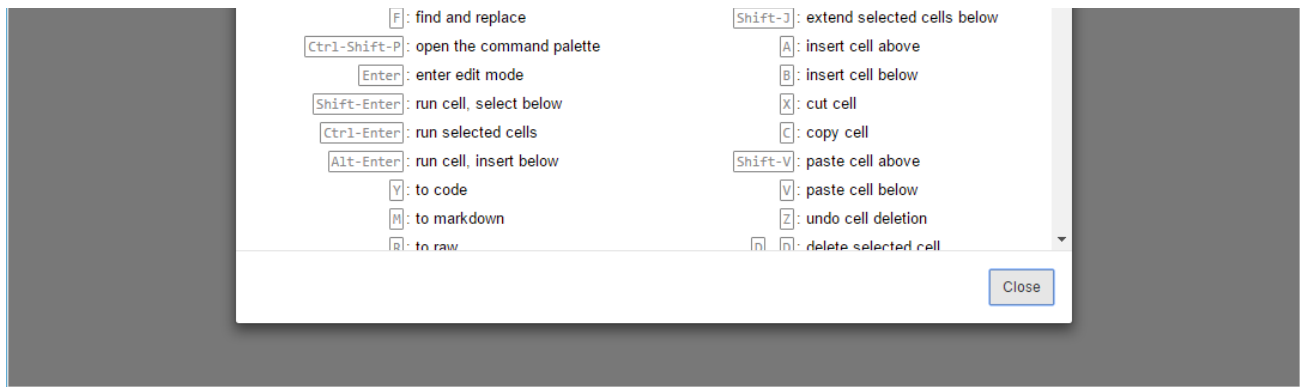


- C'est un nouveau carnet et c'est ce que vous devriez voir :



- Vous pouvez faire un tour de l'interface du portable à partir de "Aide -> Tour de l'interface utilisateur", ou vous pouvez apprendre quelques raccourcis à partir de "Aide -> Raccourcis clavier".





Vous devrez lancer Jupyter Notebooks chaque fois que vous voudrez travailler sur ces tutoriels. N'oubliez pas d'enregistrer vos fichiers `.ipynb` où vous pourrez les trouver facilement. Vous pouvez également télécharger tous les carnets à partir de ce dépôt et, si vous les exécutez dans Jupyter Notebook, vous pouvez exécuter le code ici, ou cliquer sur le texte pour voir comment la version Markdown du HTML est écrite pour coder ce texte.

Dans la section suivante, vous apprendrez comment cloner ce dépôt.

Références:

- [Anaconda \(https://www.anaconda.com/\)](https://www.anaconda.com/)
- [Project Jupyter \(https://jupyter.org/\)](https://jupyter.org/)
- [Conda Cheatsheet \(https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf\)](https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf)

2.1.2 Contrôle des versions et gestion des codes avec Git et GitHub

Le tutoriel suivant est adapté [du tutoriel "Hello World" de GitHub \(https://guides.github.com/activities/hello-world/\)](https://guides.github.com/activities/hello-world/) et de leur manuel "[Git Handbook \(https://guides.github.com/introduction/git-handbook/\)](https://guides.github.com/introduction/git-handbook/)". GitHub est une implémentation particulière de Git. Il se trouve que c'est là que ces leçons sont hébergées, mais vous pouvez utiliser tout autre service, tel que [GitLab \(https://about.gitlab.com/\)](https://about.gitlab.com/), [Bitbucket \(https://bitbucket.org/\)](https://bitbucket.org/) ou même le vénérable [SourceForge \(https://sourceforge.net/\)](https://sourceforge.net/).

[Git \(https://git-scm.com/\)](https://git-scm.com/) est un système de contrôle de version distribué, libre et gratuit, conçu pour gérer tout type de projet, du plus petit au plus grand, avec rapidité et efficacité. Les systèmes de contrôle de version suivent l'historique des changements au fur et à mesure que les personnes et les équipes collaborent sur les projets. Au fur et à mesure de l'évolution du projet, les équipes peuvent effectuer des tests, corriger des bugs et contribuer au nouveau code avec la certitude que toute version peut être récupérée à tout moment.

Les développeurs peuvent consulter l'historique du projet pour savoir :

- Quelles modifications ont été apportées ?
- Qui a effectué les changements ?
- Quand les modifications ont-elles été apportées ?
- Pourquoi les changements étaient-ils nécessaires ?

Git lui-même fonctionne en ligne de commande (c'est-à-dire dans une fenêtre de terminal, et en dehors de l'interface graphique plus familière que vous utilisez pour interagir avec le système d'exploitation de votre ordinateur). Il y a de nombreuses commandes à apprendre, mais nous ne passerons en revue que le sous-ensemble le plus basique dont vous avez besoin pour l'instant. Git dispose d'une [série de vidéos de base \(https://git-scm.com/video/get-going\)](https://git-scm.com/video/get-going) que vous pouvez suivre pour en savoir plus.

Un *repository*, ou projet Git, englobe l'ensemble des fichiers et des dossiers associés à un projet, ainsi que l'historique des révisions de chaque fichier. L'historique des fichiers apparaît sous la forme d'instantanés dans le temps appelés *commits*, et les commits existent sous la forme d'une relation de liste liée, et peuvent être organisés en plusieurs lignes de développement appelées *branches*. Comme Git est un système de contrôle de version distribué, les dépôts sont des unités autonomes et toute personne possédant une copie du dépôt peut accéder à l'ensemble de la base de code et à son historique. En utilisant la ligne de commande ou d'autres interfaces faciles d'utilisation, un dépôt git permet également : l'interaction avec l'historique, le clonage, la création de branches, le commit, la fusion, la comparaison des changements entre les versions du code, et plus encore.

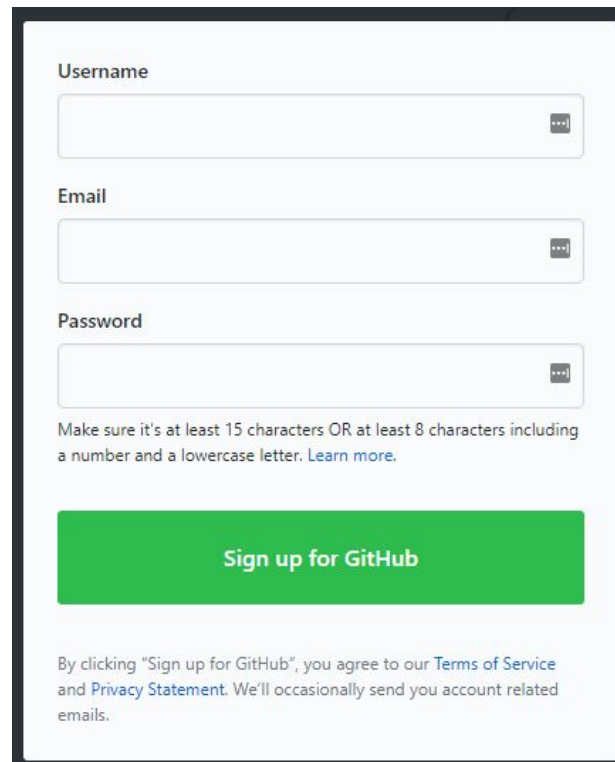
Travailler dans des dépôts permet de garder les projets de développement organisés et protégés. Les développeurs sont encouragés à corriger les bugs ou à créer de nouvelles fonctionnalités, sans craindre de faire dérailler les efforts de développement de la ligne principale. Git facilite cela grâce à l'utilisation de branches thématiques : des pointeurs vers des engagements dans l'histoire qui peuvent être facilement créés et dépréciés lorsqu'ils ne sont plus nécessaires.

Grâce à des plateformes comme GitHub, Git offre également davantage de possibilités de transparence et de collaboration sur les projets. Les dépôts publics aident les équipes à travailler ensemble pour construire le meilleur produit final possible.

Cette leçon, par exemple, existe dans un dépôt GitHub public et open-source avec contrôle de version Git.

[GitHub \(https://github.com\)](https://github.com) est un dépôt d'hébergement Git qui fournit aux développeurs des outils pour améliorer le code grâce à des fonctionnalités en ligne de commande, de gestion de problèmes (discussions en fil de discussion), des demandes d'extraction et la révision du code. Vous pouvez en savoir plus sur [GitHub's flow ici \(https://guides.github.com/introduction/flow/\)](https://guides.github.com/introduction/flow/).

Pour compléter la partie suivante de ce tutoriel, vous aurez besoin d'un compte [GitHub.com](http://github.com/) (<http://github.com/>) et d'un accès à Internet. Vous n'avez pas besoin de savoir comment coder, utiliser la ligne de commande ou installer Git.



The screenshot shows the GitHub sign-up page. It has three input fields for 'Username', 'Email', and 'Password', each with a 'show/hide' icon. Below the password field is a note: 'Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)'. A large green button labeled 'Sign up for GitHub' is centered. At the bottom, there is a disclaimer: 'By clicking "Sign up for GitHub", you agree to our [Terms of Service](#) and [Privacy Statement](#). We'll occasionally send you account related emails.'

Créer un dépôt de tutoriels

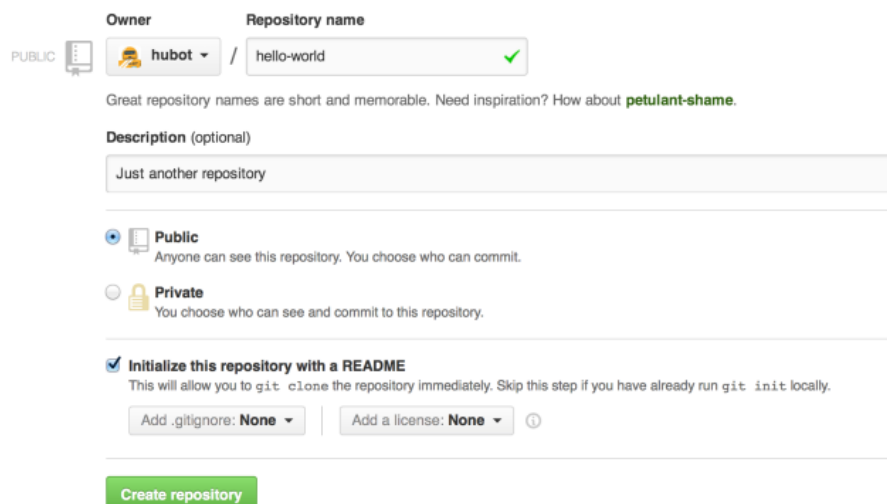
Un *repository* est généralement utilisé pour organiser un seul projet. Les dépôts peuvent contenir des dossiers et des fichiers, des images, des vidéos, des feuilles de calcul et des ensembles de données - tout ce dont votre projet a besoin. Nous vous recommandons d'inclure un README, ou un fichier contenant des informations sur votre projet. GitHub permet d'en ajouter un facilement en même temps que vous créez votre nouveau dépôt. Il offre également d'autres options courantes telles qu'un fichier de licence.

Votre dépôt "hello-world" peut être un endroit où vous stockez des idées, des ressources, ou même où vous partagez et discutez avec d'autres. Vous pouvez l'appeler comme vous voulez, mais vous aurez besoin d'un dépôt pour cette leçon et les tutoriels suivants. Profitez-en pour créer un premier dépôt. Vous pouvez l'appeler par exemple "cours sur la préparation de données".

Pour les comptes GitHub gratuits, tout ce que vous faites est public. Ne stockez pas d'informations personnelles importantes, et n'exprimez pas de comportements qui pourraient vous embarrasser ou mettre votre sécurité en danger.

Pour créer un nouveau dépôt, vous devez avoir créé votre compte GitHub. Ensuite, vous devez vous connecter à votre compte GitHub :

- Dans le coin supérieur droit, à côté de votre avatar ou identicon, cliquez sur **+** puis sélectionnez **Nouveau dépôt**.
- Donnez à votre dépôt le nom de data-wrangling-course (ou tout autre nom que vous préférez).
- Rédigez une brève description.
- Sélectionnez **Initialisez ce dépôt avec un LISEZ-MOI**.



The screenshot shows the 'Create repository' form on GitHub. It includes a 'PUBLIC' toggle, an 'Owner' dropdown set to 'hubot', and a 'Repository name' field containing 'hello-world' with a green checkmark. A tip suggests repository names should be short and memorable, with an example 'petulant-shame'. There is a 'Description (optional)' field with the text 'Just another repository'. Below this are radio buttons for 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone can see this repository. You choose who can commit.' The 'Private' option is described as 'You choose who can see and commit to this repository.' There is a checked checkbox for 'Initialize this repository with a README', with a note: 'This will allow you to git clone the repository immediately. Skip this step if you have already run git init locally.' Below this are two dropdowns: 'Add .gitignore: None' and 'Add a license: None'. A green 'Create repository' button is at the bottom.

- Cliquez sur **Create repository**.

Créer une branche

La *Branching* est la façon de travailler sur différentes versions d'un dépôt à la fois.

Par défaut, votre dépôt possède une branche nommée `master` qui est considérée comme la branche définitive. Nous utilisons les branches pour expérimenter et faire des modifications avant de les livrer à `master`.

Lorsque vous créez une branche à partir de la branche `master`, vous faites une copie, ou un instantané, de `master` tel qu'il était à ce moment là. Si quelqu'un d'autre apportait des modifications à la branche `master` pendant que vous travaillez sur votre branche, vous pourriez faire ces mises à jour.

Ce diagramme montre :

- La branche `master`
- Une nouvelle branche appelée `feature` (parce que nous faisons du 'feature work' sur cette branche)
- Le voyage qu'effectue la branche `feature` avant de se fondre dans le `master`.



Avez-vous déjà enregistré différentes versions d'un fichier ? Quelque chose comme :

- `story.txt`
- `story-joe-edit.txt`
- `story-joe-edit-reviewed.txt`

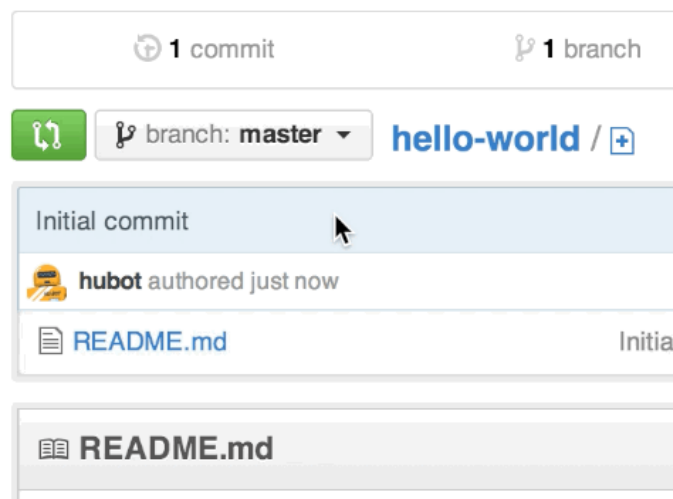
Les branches accomplissent des objectifs similaires dans les dépôts GitHub.

Sur GitHub, les développeurs, les rédacteurs et les concepteurs utilisent les branches pour garder les corrections de bugs et le travail sur les fonctionnalités séparés de la branche `master` (production). Lorsqu'une modification est prête, ils fusionnent leur branche avec la branche `master`.

Pour créer une nouvelle branche :

- Allez à votre nouveau dépôt `data-wrangling-course`.
- Cliquez sur le menu déroulant en haut de la liste des fichiers qui indique **branche : master**.
- Tapez un nom de branche, `readme-edits`, dans la boîte de texte de la nouvelle branche.
- Sélectionnez la case bleue **Create branch** ou appuyez sur `Enter` sur votre clavier.

Just another repository — Edit



Vous avez maintenant deux branches, `master` et `readme-edits`. Elles sont exactement les mêmes, mais pas pour longtemps ! Ensuite, nous allons ajouter nos changements à la nouvelle branche.

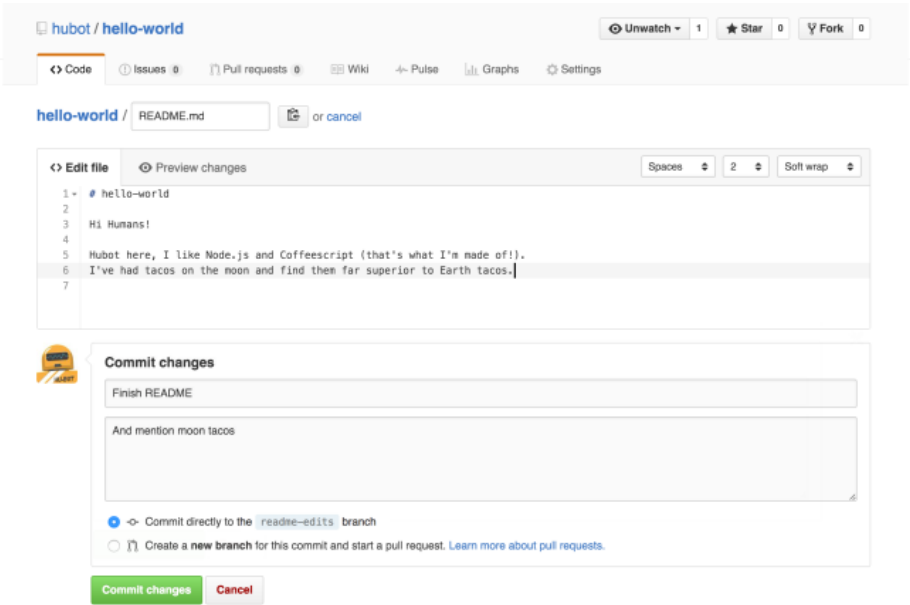
Faites et validez les changements

Maintenant, vous êtes sur la vue du code de votre branche `readme-edits`, qui est une copie de `master`. Faisons quelques modifications.

Sur GitHub, les changements sauvegardés sont appelés *commits*. Chaque commit a un *commit message* associé, qui est une description expliquant pourquoi un changement particulier a été fait. Les messages de commit capturent l'historique de vos changements, afin que les autres contributeurs puissent comprendre ce que vous avez fait et pourquoi.

Effectuez et validez des changements :

- Cliquez sur le fichier README .md.
- Cliquez sur l'icône représentant un crayon dans le coin supérieur droit de la vue du fichier à modifier.
- Dans l'éditeur, écrivez un peu sur vous-même.
- Rédigez un message de validation qui décrit vos modifications.
- Cliquez sur le bouton **Commit changes**.



Ces changements seront apportés au fichier README sur votre branche readme-edits, donc maintenant cette branche contient un contenu différent de master.

Ouvrir une demande de fusion (pull request)

Maintenant que vous avez des changements dans une branche de master, vous pouvez ouvrir une *pull request*.

Les Pull Requests sont le coeur de la collaboration sur GitHub. Lorsque vous ouvrez une *pull request*, vous proposez vos changements et vous demandez à quelqu'un de revoir et de tirer votre contribution et de les fusionner dans sa branche. Les pull request montrent les *diffs*, ou différences, du contenu des deux branches. Les modifications, ajouts et soustractions sont indiqués en vert et en rouge.

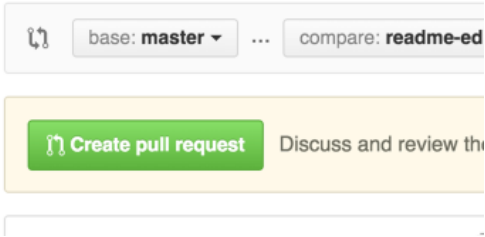
Dès que vous faites un commit, vous pouvez ouvrir une pull request et lancer une discussion, avant même que le code ne soit terminé.

En utilisant le système @mention de GitHub dans votre message de pull request, vous pouvez demander un retour d'information à des personnes ou des équipes spécifiques, qu'elles soient au bout du couloir ou à dix fuseaux horaires de distance.

Vous pouvez même ouvrir des pull request dans votre propre dépôt et les fusionner vous-même. C'est un excellent moyen d'apprendre le fonctionnement du GitHub avant de travailler sur des projets plus importants.

Ouvrez une demande de modification du fichier README:.

Step	Screenshot
Cliquez sur l'onglet Pull Request , puis à partir de la page Pull Requet, cliquez sur le bouton vert New Pull Request .	
Dans la case Example Comparisons , sélectionnez la branche que vous avez créée, readme-edits, pour la comparer avec master (l'original).	
Regardez vos changements dans les différences sur la page Comparaison, assurez-vous qu'ils correspondent à ce que vous voulez soumettre.	

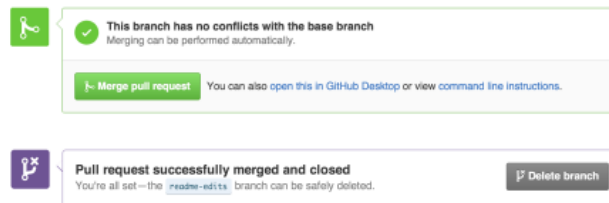
	
Lorsque vous êtes convaincu que ce sont les changements que vous voulez soumettre, cliquez sur le gros bouton vert Create Pull Request .	
Donnez un titre à votre demande de tirage et rédigez une brève description de vos changements.	

Lorsque vous avez terminé votre message, cliquez sur **Create pull request!**

Fusionnez vos changements

Lors de cette dernière étape, il est temps de rassembler vos changements - en fusionnant votre branche readme-edits avec la branche master.

- Cliquez sur le bouton vert **Merge pull request** pour fusionner les modifications dans la branche master.
- Cliquez sur **Confirm Merge**.
- Continuez et supprimez la branche, puisque ses modifications ont été incorporées, avec le bouton **Delete Branch** dans la case violette.



Et cela complète le tutoriel GitHub. Voici ce que vous avez fait :

- Création d'un dépôt open source
- Création et gestion d'une nouvelle branche
- Modification d'un fichier et commit de ces modifications sur GitHub
- Ouverture et fusion d'une demande d'intégration (pull request)

Exercice:

GitHub dispose d'une application de bureau (<https://desktop.github.com/>) que vous pouvez utiliser pour gérer vos dépôts et votre code directement depuis votre ordinateur. Elle est dotée d'une interface utilisateur facile à utiliser et facilitera le clonage, l'édition, l'insertion et l'extraction de modifications dans vos dépôts.

Pour cet exercice:

- Etudier le tutoriel de 'application de bureau de GitHub par vous-même. (<https://help.github.com/en/desktop/getting-started-with-github-desktop>).
- Installer l'application. (<https://desktop.github.com/>)
- Vous n'êtes pas obligé, mais il vous sera peut-être plus facile de installer Notepad++ (<https://notepad-plus-plus.org/downloads/>) pour modifier le code et le texte.
- Refaites ce tutoriel GitHub, mais directement via l'application et sur votre propre bureau. Vous pouvez éditer votre fichier 'README' dans Notepad++, puis l'enregistrer et pousser vos modifications dans votre dépôt en utilisant l'app.
- Vous devez créer un répertoire de codes pour sauvegarder tous vos projets de codes. Il est essentiel d'être très soigné et organisé en ce qui concerne les répertoires de code pour gérer vos dépôts. Prenez-en l'habitude dès maintenant.

Dans la section suivante, vous commencerez à apprendre à coder en Python.

Tutoriels et références:

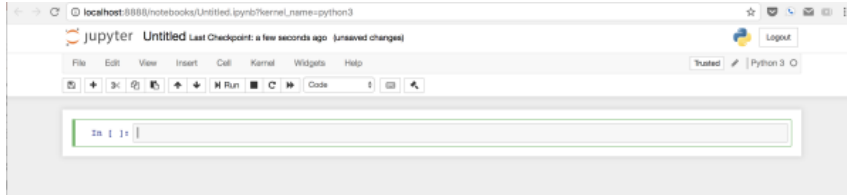
- GitHub's 'Hello World' tutorial (<https://guides.github.com/activities/hello-world/>)

- [Git Basics Get Going \(https://git-scm.com/video/get-going/\)](https://git-scm.com/video/get-going/).
- [GitHub Guides \(http://guides.github.com/\)](http://guides.github.com/).
- [GitHub YouTube channel \(http://youtube.com/githubguides\)](http://youtube.com/githubguides).
- [GitHub Desktop App \(https://desktop.github.com/\)](https://desktop.github.com/).

2.1.3 Les bases du python

Python est un langage interprété avec une syntaxe très simple. La première étape dans l'apprentissage de tout nouveau langage est de coder une application qui ne fera qu'écrire "Bonjour, le monde !".

Vous aurez ouvert un nouveau carnet dans **Jupyter Notebook** :



Vous devez enregistrer ce cahier dans un dossier dédié à vos tutoriels pour ce cours. Donnez-lui un nom, comme Tutoriel de codage Python 1, ou tout ce qui vous convient. Vous pourriez vous retrouver avec une arborescence de répertoires qui ressemble à celle-ci :

```
documents/github-code/data-wrangling-course/
```

Notez deux choses :

- L'élément de menu déroulant qui dit code
- La disposition de la zone de saisie de texte en dessous (qui commence par Dans []:)

Ceci précise que ce bloc doit être utilisé pour le code. Vous pouvez également l'utiliser pour des blocs de texte structurés comme celui-ci en sélectionnant markdown dans la liste de menu. Ce tutoriel n'enseignera pas la markdown, mais vous pouvez en apprendre davantage [ici \(https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet\)](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).

Le tutoriel qui suit est similaire à [Learn Python's \(https://www.learnpython.org/\)](https://www.learnpython.org/), bien qu'ici vous pourrez utiliser les exemples sur votre propre ordinateur.

Bonjour, le monde !

Pour l'instant, commençons par "Bonjour, le monde !".

Tapez le texte ci-dessous dans le bloc de code de votre ordinateur portable et appuyez sur Ctrl-Enter pour exécuter le code :

```
print("Bonjour, le monde !")
```

```
In [1]: print("Bonjour, le monde !")  
Bonjour, le monde !
```

Vous venez de lancer votre premier programme. Voyez comment Jupyter effectue pour vous la surbrillance de code, en changeant les couleurs du texte selon la nature de la syntaxe utilisée. Le terme print est un terme protégé et est mis en évidence en vert.

Cela montre également à quel point Jupyter Notebook peut être utile. Vous pouvez le traiter comme un document, en enregistrant le fichier et en stockant les résultats de votre programme dans le notebook. Vous pouvez même l'envoyer par e-mail à quelqu'un d'autre et, si cette personne a le Jupyter Notebook, elle peut l'utiliser pour voir ce que vous avez fait. Si vous sauvegardez votre travail dans GitHub, vous pouvez partager les liens avec d'autres personnes sur ce cours et obtenir de l'aide ou des conseils si vous êtes bloqué.

À partir de maintenant, nous allons simplement passer par le tutoriel de codage Python et apprendre la syntaxe et les méthodes de codage.

Indentation

Python utilise l'indentation pour indiquer les parties du code qui doivent être exécutées ensemble. Les tabulations et les espaces (généralement quatre par niveau) sont pris en charge, et ma préférence va aux tabulations. Ceci fait l'objet d'un débat, mais ne vous en faites pas. Quoi que vous décidiez de faire, c'est très bien, mais **ne mélangez en aucun cas l'indentation par tabulation avec l'indentation par espace**.

Dans cet exercice, vous allez attribuer une valeur à une variable, vérifier si une comparaison est vraie, puis - en fonction du résultat - imprimer.

Tout d'abord, repérez le petit symbole + dans la barre de menu juste après le symbole enregistrer. Cliquez dessus et vous obtiendrez une nouvelle boîte dans laquelle vous pourrez taper le code suivant. Lorsque vous avez terminé, appuyez sur la touche Ctrl-Entrée ou sur le bouton Exécuter.

```
In [2]: x = 1  
        if x == 1:
```

```
# Indenté ... et remarquez comment Jupyter Notebook s'est automatiquement indenté pour vous
print("x est égal à 1")

x est égal à 1
```

Tout terme de texte non protégé peut être une variable. Votre x aurait pu facilement être votre nom. Habituellement, nous donnons aux variables un nom aussi descriptif que possible afin de pouvoir lire les algorithmes comme du texte (c'est-à-dire que le code se décrit lui-même).

Syntaxe

- Pour attribuer une variable avec une valeur spécifique, utilisez =
- Pour tester si une variable a une valeur spécifique, utilisez les opérateurs booléens:
 - égalité : ==
 - pas égal : !=
 - plus grand que : >
 - moins que : <
- Vous pouvez également ajouter des commentaires utiles à votre code avec le symbole #. Toute ligne commençant par un # n'est pas exécutée. Je trouve très utile de prendre des notes détaillées sur mes réflexions car, souvent, lorsque vous revenez au code plus tard, vous ne vous souvenez pas pourquoi vous avez fait ce que vous avez fait, ou même ce que votre code est censé faire.

Variables et types

Python n'est pas "statiquement typé". Cela signifie que vous n'avez pas à déclarer toutes vos variables avant de pouvoir les utiliser. Vous pouvez créer de nouvelles variables quand vous le souhaitez. Python est également "orienté objet", ce qui signifie que chaque variable est un objet. Cela deviendra de plus en plus important au fur et à mesure que vous gagnerez en expérience.

Passons en revue les principaux types de variables :

Nombres

Python supporte deux types de nombres : les nombres entiers et les nombres flottants. Les nombres entiers sont des nombres entiers (par exemple 7), tandis que les nombres flottants sont des nombres fractionnaires (par exemple 7.321). Vous pouvez également convertir des entiers en nombres flottants, et vice versa, mais vous devez être conscient des risques que cela comporte.

Suivez le code :

```
In [4]: myint = 7
        print(myint)
```

```
7
```

```
In [5]: myfloat = 7.0
        print(myfloat)
        # Ou vous pourriez convertir l'entier que vous avez déjà
        myfloat = float(myint)
        # Notez que le terme "float" est vert. C'est un terme protégé.
        print(myfloat)
```

```
7.0
```

```
7.0
```

```
In [6]: # Maintenant, voyez ce qui se passe quand vous convertissez un float en un int
        myint = int(7.3)
        print(myint)
```

```
7
```

Remarquez comment vous avez perdu de la précision lorsque vous avez converti un float en un int ? Faites toujours attention, car cela pourrait être la différence entre une porte qui s'adapte à son cadre et une porte qui est beaucoup trop petite.

Chaînes de caractères

Les chaînes de caractères sont le terme Python pour le texte. Vous pouvez les définir entre guillemets simples ou doubles. Je vais utiliser des guillemets doubles (puisque vous utilisez souvent une seule citation à l'intérieur des phrases de texte).

Essayez ces exemples :

```
In [9]: mystring = "Bonjour, le monde !"
        print(mystring)
        # et en montrant comment utiliser une apostrophe dans une chaîne
        mystring = "'Parlons des apostrophes...'"
        print(mystring)
```

```
Bonjour, le monde !
```

```
'Parlons des apostrophes...'
```

Vous pouvez également appliquer des opérateurs simples à vos variables, ou assigner plusieurs variables simultanément.

```
In [10]: one = 1
```

```
two = 2
three = one + two
print(three)

hello = "Bonjour,"
world = "le monde !"
helloworld = hello + " " + world
print(helloworld)

a, b = 3, 4
print(a, b)
```

```
3
Bonjour, le monde !
3 4
```

Notez cependant que le mélange des types de variables pose des problèmes.

```
In [11]: print(one + two + hello)
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-11-a2961e4891f2> in <module>
----> 1 print(one + two + hello)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python lancera une erreur quand vous ferez une erreur comme celle-ci et l'erreur vous donnera autant de détails que possible sur ce qui vient de se passer. C'est extrêmement utile lorsque vous essayez de "débuguer" votre code.

Dans ce cas, on vous dit (traduction): `TypeError : type(s) d'opérande non pris en charge pour + : 'int' et 'str'`.

Et le contexte doit indiquer clairement que vous avez essayé de combiner deux variables entières avec une chaîne.

Vous pouvez également combiner des chaînes de caractères avec des caractères de remplacement pour les variables :

Syntaxe

- Ajouter des variables à une chaîne de caractères avec des 'F-strings', par exemple `F"Variable {x}"` remplacera le `{x}` dans la chaîne par la valeur de la variable `x`
- Les nombres à virgule flottante peuvent échapper à tout contrôle (imaginez que vous incluez un nombre avec 30 décimales dans une chaîne), et vous pouvez formater cela avec `F"Variable {x:10.4f}"` où `10` est l'espace alloué à la partie entière (utile si vous devez aligner une colonne de nombres ; vous pouvez également laisser cela de côté pour inclure tous les chiffres significatifs) et le `.4f` est le nombre de décimales après le point. Adaptez le format selon vos besoins.

```
In [12]: variable = 1/3 * 100
print(F"Variable non formatée : {variable}%")
print(F"Variable formatée : {variable:.3f}%")
print(F"Variable formatée en espace : {variable:10.1f}%")
```

```
Variable non formatée : 33.33333333333333%
Variable formatée : 33.333%
Variable formatée en espace :          33.3%
```

Listes

Lists est une liste ordonnée de tout type de variable. Vous pouvez combiner autant de variables que vous le souhaitez, et elles peuvent même être de plusieurs types. En règle générale, à moins que vous n'ayez une raison particulière de le faire, les listes contiennent des variables d'un seul type.

Vous pouvez également itérer sur une liste (utiliser chaque élément d'une liste dans l'ordre).

Une liste est placée entre crochets : `[]`

```
In [13]: mylist = []
mylist.append(1)
mylist.append(2)
mylist.append(3)
# Chaque élément d'une Liste peut être abordé directement.
# La première adresse dans une Liste Python commence à 0
print(mylist[0])
# Le dernier élément d'une Liste Python peut être adressé comme -1.
# Ceci est utile lorsque vous ne savez pas quelle sera la longueur de la Liste.
print(mylist[-1])
# Vous pouvez également sélectionner des sous-ensembles de données dans une Liste comme celle-ci
print(mylist[1:3])

# Vous pouvez aussi faire une boucle dans une liste en utilisant une instruction "for".
# Notez que `x` est une nouvelle variable qui prend la valeur de chaque élément de la Liste dans l'ordre.
for x in mylist:
    print(x)
```

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

Si vous essayez d'accéder à un élément d'une liste qui ne s'y trouve pas, vous obtiendrez une erreur.

```
In [13]: print(mylist[10])
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-13-46c3ae90a572> in <module>
----> 1 print(mylist[10])

IndexError: list index out of range
```

Voyons cela avec un exemple un peu plus complexe. Mais d'abord, une nouvelle syntaxe :

Syntaxe

- Vérifiez quel type de variable vous avez avec `isinstance`, par exemple `isinstance(x, float)` sera `True` si `x` est un `float`
- Vous avez déjà vu `for`, mais vous pouvez obtenir le décompte de boucle en enveloppant votre liste dans le terme `enumerate`, par exemple `for count, x in enumerate(mylist)` vous donnera un décompte pour chaque élément de la liste (la variable 'count' contient l'index et `x` l'élément de la liste à cet index)
- Trier une liste par ordre numérique ou alphabétique peut être fait avec `sort`
- Pour obtenir le nombre d'éléments dans une liste, il suffit de demander `len(list)`
- Si vous voulez compter le nombre de fois qu'une variable particulière apparaît dans une liste, utilisez `list.count(x)` (où `x` est la variable qui vous intéresse)

Essayez-le vous-même.

```
In [14]: # Imaginons que nous ayons une liste de noms non ordonnés qui, d'une manière ou d'une autre, comporte des
#         numéros aléatoires.
# Pour cet exercice, nous voulons imprimer la liste alphabétique des noms sans les numéros.
# Ce n'est pas la meilleure façon de faire l'exercice, mais cela illustrera tout un tas de techniques.
names = ["John", 3234, 2342, 3323, "Eric", 234, "Jessica", 734978234, "Lois", 2384]
print(F"Nombre de noms dans la liste : {len(names)}")
# D'abord, débarrassons-nous de tous les entiers bizarres.
new_names = []
for n in names:
    if isinstance(n, str):
        # Vérifier si n est une chaîne de caractères
        # Et notez que nous sommes maintenant deux fois en retrait dans cette nouvelle composante
        new_names.append(n)
# Nous ne devrions maintenant avoir que des noms dans la nouvelle liste. Faisons un tri..
new_names.sort()
print(F"Nombre de noms nettoyés dans la liste : {len(new_names)}")
# Enfin, imprimons-les.
for i, n in enumerate(new_names):
    # En utilisant à la fois i et n dans une chaîne de caractères formatée
    # Ajout de 1 à i car les listes commencent à 0
    print(F"{i+1}. {n}")
```

```
Nombre de noms dans la liste : 10
Nombre de noms nettoyés dans la liste : 4
1. Eric
2. Jessica
3. John
4. Lois
```

Dictionnaires (Dictionaries)

Les dictionnaires sont l'un des types de données les plus utiles et les plus polyvalents en Python. Ils sont similaires aux tableaux, mais se composent de paires clé:valeur. Chaque valeur stockée dans un dictionnaire est accessible par sa clé, et la valeur peut être n'importe quel type d'objet (chaîne de caractères, nombre, liste, etc.).

Cela vous permet de créer des enregistrements structurés. Les dictionnaires sont placés dans des `{}`.

```
In [15]: phonebook = {}
phonebook["John"] = {"Phone": "012 794 794",
                    "Email": "john@email.com"}
phonebook["Jill"] = {"Phone": "012 345 345",
                    "Email": "jill@email.com"}
phonebook["Joss"] = {"Phone": "012 321 321",
                    "Email": "joss@email.com"}
print(phonebook)
```

```
{'John': {'Phone': '012 794 794', 'Email': 'john@email.com'}, 'Jill': {'Phone': '012 345 345', 'Email': 'j...
```

```
ill@email.com'}, 'Joss': {'Phone': '012 321 321', 'Email': 'joss@email.com'}}}
```

Notez que vous pouvez imbriquer des dictionnaires et des listes. Le tableau ci-dessus vous montre comment vous pouvez ajouter des valeurs à un dictionnaire existant ou créer des dictionnaires avec des valeurs.

Vous pouvez itérer sur un dictionnaire tout comme sur une liste, en utilisant le terme `.items()`. En Python 3, le dictionnaire maintient l'ordre dans lequel les données ont été ajoutées, mais les anciennes versions de Python ne le font pas.

```
In [16]: for name, record in phonebook.items():
         print(F"Le numéro de téléphone de {name} est le {record['Phone']} et leur adresse électronique est {re
cord['Email']}")
```

```
Le numéro de téléphone de John est le 012 794 794 et leur adresse électronique est john@email.com
Le numéro de téléphone de Jill est le 012 345 345 et leur adresse électronique est jill@email.com
Le numéro de téléphone de Joss est le 012 321 321 et leur adresse électronique est joss@email.com
```

Vous ajoutez de nouveaux enregistrements comme indiqué ci-dessus, et vous supprimez les enregistrements avec `del` ou `pop`. Ils ont chacun un effet différent.

```
In [17]: # Première `del`
del phonebook["John"]
for name, record in phonebook.items():
    print(F"Le numéro de téléphone de {name} est le {record['Phone']} et leur adresse électronique est {re
cord['Email']}")
```

```
# Pop retourne l'élément, et le supprime
jill_record = phonebook.pop("Jill")
print(jill_record)
for name, record in phonebook.items():
    # Vous pouvez voir que seul Joss est encore dans le système
    print(F"Le numéro de téléphone de {name} est le {record['Phone']} et leur adresse électronique est {re
cord['Email']}")
```

```
# Si vous essayez de supprimer un enregistrement qui n'est pas dans le dictionnaire, vous obtenez une erre
ur
del phonebook["John"]
```

```
Le numéro de téléphone de Jill est le 012 345 345 et leur adresse électronique est jill@email.com
Le numéro de téléphone de Joss est le 012 321 321 et leur adresse électronique est joss@email.com
{'Phone': '012 345 345', 'Email': 'jill@email.com'}
Le numéro de téléphone de Joss est le 012 321 321 et leur adresse électronique est joss@email.com
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-17-c6500625ef15> in <module>
    12
    13 # Si vous essayez de supprimer un enregistrement qui n'est pas dans le dictionnaire, vous obtenez
une erreur
--> 14 del phonebook["John"]
```

```
KeyError: 'John'
```

Une chose à faire est de prendre l'habitude de tester les variables avant de supposer qu'elles présentent les caractéristiques que vous recherchez. Vous pouvez tester un dictionnaire pour voir s'il a un enregistrement, et renvoyer une réponse par défaut s'il n'en a pas.

Vous faites cela avec le terme `.get("key", default)`. La réponse par défaut peut être n'importe quoi, y compris une autre variable, ou simplement `True` ou `False`. Si vous laissez le terme `default` vide (c'est-à-dire `.get("key")`), le résultat sera automatiquement `False` s'il n'y a pas d'enregistrement.

```
In [19]: # True et False sont des termes spéciaux en Python qui vous permettent de définir des tests
jill_record = phonebook.get("Jill", False)
if jill_record: # c'est-à-dire si vous avez un enregistrement dans l'étape précédente
    print(F"Le numéro de téléphone de Jill est {jill_record['Phone']}, et leur adresse électronique est {j
ill_record['Email']}")
else: # L'alternative, si "if" renvoie False
    print("Aucun enregistrement trouvé.")
```

```
Aucun enregistrement trouvé.
```

Opérateurs de base

Les opérateurs sont les différents symboles algébriques (tels que `+`, `-`, `*`, `/`, `%`, etc.). Une fois que vous avez appris la syntaxe, la programmation est surtout des mathématiques.

Opérateurs arithmétiques

Comme vous vous en doutez, vous pouvez utiliser les différents opérateurs mathématiques avec les nombres (entiers et flottants).

```
In [20]: number = 1 + 2 * 3 / 4.0
         # Essayez de prédire quelle sera la réponse ... Python suit-il la hiérarchie des opérations d'ordre ?
         print(number)
```

```
# Le modulo (%) renvoie le reste entier d'une division
remainder = 11 % 3
print(remainder)

# Deux multiplications sont équivalentes à une opération de puissance
squared = 7 ** 2
print(squared)
cubed = 2 ** 3
print(cubed)

2.5
2
49
8
```

Liste des opérateurs

```
In [21]: even_numbers = [2, 4, 6, 8]
# Un de mes premiers professeurs à l'école m'a dit : "Les gens sont bizarres. Les chiffres sont inégaux."
# Il a aussi dit : "Cecil John Rhodes mangeait toujours des tas de cacahuètes non décortiquées au parlemen
t du Cap."
# "Vous saviez qu'il était au parlement grâce à l'énorme tas de coquilles sur le sol. Il ne portait jamais
non plus de chaussettes".
# "Vous n'oubliez jamais cela." Et je n'ai pas oublié. Je ne sais pas si c'est vrai.
uneven_numbers = [1, 3, 5, 7]
all_numbers = uneven_numbers + even_numbers
# Que pensez-vous qu'il va se passer ?
print(all_numbers)

# Vous pouvez également répéter des séquences de listes
print([1, 2, 3] * 3)

[1, 3, 5, 7, 2, 4, 6, 8]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Nous pouvons rassembler tout cela en un petit projet.

```
In [22]: x = object() # Un objet Python générique
y = object()

# Modifier ce code pour que x_list et y_list aient chacun 10 objets répétitifs
# et concat_list est la concaténation de x_list et y_list
x_list = [x]
y_list = [y]
concat_list = []

print(f"x_list contient {len(x_list)} objets")
print(f"y_list contient {len(y_list)} objets")
print(f"big_list contient {len(concat_list)} objets")

# Testez vos listes
if x_list.count(x) == 10 and y_list.count(y) == 10:
    print("On y est presque...")
if concat_list.count(x) == 10 and concat_list.count(y) == 10:
    print("Super !")

x_list contient 1 objets
y_list contient 1 objets
big_list contient 0 objets
```

Opérateurs de strings

Vous pouvez faire un nombre surprenant de choses avec des opérateurs sur des strings.

```
In [23]: # Vous avez déjà vu des enchaînements arithmétiques de cordes
helloworld = "Hello," + " " + "World!"
print(helloworld)

# Vous pouvez également multiplier les chaînes de caractères pour former une séquence répétitive
manyhellos = "Hello " * 10
print(manyhellos)

# Mais ne vous emportez pas. Tout ne fonctionnera pas.
nohellos = "Hello " / 10
print(nohellos)

Hello, World!
Hello Hello Hello Hello Hello Hello Hello Hello Hello
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-23-92892e606944> in <module>
      8
      9 # Mais ne vous emportez pas. Tout ne fonctionnera pas.
--> 10 nohellos = "Hello " / 10
```

```
11 print(nohellos)
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Il faut garder à l'esprit que les chaînes de caractères sont des listes de caractères. Cela signifie que vous pouvez effectuer un certain nombre d'opérations de liste sur des chaînes de caractères. Et quelques nouvelles opérations.

Syntaxe

- Obtenez l'index de la première occurrence d'une lettre spécifique avec `string.index("l")` où l est la lettre que vous recherchez
- Comme dans les listes, comptez le nombre d'occurrences d'une lettre spécifique avec `string.count("l")`
- Obtenir des portions de chaînes avec `string[start:end]`, par exemple `string[3:7]`. Si vous n'êtes pas sûr de la fin d'une chaîne, souvenez-vous que vous pouvez utiliser des nombres négatifs pour compter à partir de la fin, par exemple `string[:-3]` pour obtenir une portion du premier caractère au troisième à partir de la fin
- Vous pouvez également "passer" dans une chaîne de caractères avec `string[start:stop:step]`, par exemple `string[2:6:2]` qui sautera un caractère entre les caractères 2 et 5 (c'est-à-dire que 6 est la limite)
- Vous pouvez utiliser un "pas" négatif pour inverser l'ordre des caractères, par exemple `string[::-1]`
- Vous pouvez convertir des chaînes de caractères en majuscules ou en minuscules avec `string.upper()` et `string.lower()`
- Tester si une chaîne commence ou se termine par une sous-chaîne avec:
 - `string.startswith(substring)` qui renvoie True ou False
 - `string.endswith(substring)` qui renvoie True ou False
- Utilisez `in` pour tester si une chaîne contient une sous-chaîne, ainsi `substring in string` retournera True ou False
- Vous pouvez diviser une chaîne en une véritable liste avec `.split(s)` où s est le caractère spécifique à utiliser pour la division, par exemple `s = ","` ou `s = " "`. Vous pouvez voir comment cela peut être utile pour fractionner un texte qui contient des données numériques.

```
In [24]: a_string = "Hello, World!"
print(f"String length: {len(a_string)}")
# Vous pouvez obtenir un index de la première occurrence d'une lettre spécifique
# Rappelez-vous que les listes Python sont basées sur 0 ; la première lettre est l'index 0
# Notez également l'utilisation de guillemets simples à l'intérieur des guillemets doubles
print(f"Index for first 'o': {a_string.index('o')}")
print(f"Count of 'o': {a_string.count('o')}")
print(f"Slicing between second and fifth characters: {a_string[2:6]}")
print(f"Skipping between 3rd and 2nd-from-last characters: {a_string[3:-2:2]}")
print(f"Reverse text: {a_string[::-1]}")
print(f"Starts with 'Hello': {a_string.startswith('Hello')}")
print(f"Ends with 'Hello': {a_string.endswith('Hello')}")
print(f"Contains 'Goodbye': {'Goodbye' in a_string}")
print(f"Split the string: {a_string.split(' ')}")
```

```
String length: 13
Index for first 'o': 4
Count of 'o': 2
Slicing between second and fifth characters: llo,
Skipping between 3rd and 2nd-from-last characters: l,Wr
Reverse text: !dlrow ,olleH
Starts with 'Hello': True
Ends with 'Hello': False
Contains 'Goodbye': False
Split the string: ['Hello,', 'World!']
```

Conditions

Dans la section [Indentation](#), vous avez vu l'instruction `if` et l'ensemble des opérateurs booléens qui vous permettent de tester différentes variables les unes par rapport aux autres.

À cette liste d'opérateurs booléens s'ajoute un nouvel ensemble de comparaisons : `and`, `or` et `in`.

```
In [26]: # Des tests booléens simples
x = 2
print(x == 2)
print(x == 3)
print(x < 3)

# Utilise `and`
name = "John"
print(name == "John" and x == 2)

# Utilise `or`
print(name == "John" or name == "Jill")

# Utilise `in` sur listes
print(name in ["John", "Jill", "Jess"])

True
False
True
```

```
True
True
True
```

Ceux-ci peuvent être utilisés pour créer des comparaisons nuancées en utilisant des "if. Vous pouvez utiliser une série de comparaisons avec if, elif et else.

N'oubliez pas que le code doit être correctement indenté, sinon vous obtiendrez un comportement inattendu.

```
In [25]: # Des résultats inattendus
x = 2
if x > 2:
    print("Testing x")
print("x > 2")
# Formé correctement
if x == 2:
    print("x == 2")
```

```
x > 2
x == 2
```

```
In [26]: # Démonstration de tests plus complexes
x = 2
y = 10
if x > 2:
    print("x > 2")
elif x == 2 and y > 50:
    print("x == 2 and y > 50")
elif x < 10 or y > 50:
    # Mais, souvenez-vous, vous ne savez pas QUELLE condition était vraie
    print("x < 10 or y > 50")
else:
    print("Rien n'a fonctionné.")
```

```
x < 10 or y > 50
```

Deux cas particuliers sont not et is.

Syntaxe

- not est utilisé pour obtenir l'inverse d'un test booléen particulier, par exemple not(False) renvoie True
- is semblerait, superficiellement, être similaire à ==, mais il vérifie si les objets réels sont les mêmes, et non si les valeurs que les objets reflètent sont égales.

Une démonstration rapide.

```
In [27]: # Utilise `not`
name_list1 = ["John", "Jill"]
name_list2 = ["John", "Jill"]
print(not(name_list1 == name_list2))

# Utilise `is`
name2 = "John"
for n in name_list1:
    print(f"{name2} is {n}: {name2 is n}")
print(name_list1 == name_list2)
print(name_list1 is name_list2)
```

```
False
John is John: True
John is Jill: False
True
False
```

Boucles

Les boucles (loops) itèrent sur une séquence donnée, et - ici - il est essentiel de s'assurer que votre indentation est correcte, sinon vous obtiendrez des résultats inattendus pour ce qui est considéré comme étant à l'intérieur ou à l'extérieur de la boucle.

Il existe deux types de loop en Python :

Syntaxe

- les boucles for passent en boucle dans une liste. Il y a également une nouvelle syntaxe à utiliser dans les boucles for:
 - Pour les listes vous avez vu enumerate, ce qui vous permet de compter le nombre de boucles
 - Range crée une liste d'entiers sur laquelle on peut faire une boucle, range(start, stop) crée une liste d'entiers entre start et stop, ou range(num) crée une liste de zéros jusqu'à num, ou range(start, stop, step) parcourt une liste par incréments de step
- Les boucles while, qui s'exécutent alors qu'une condition particulière est True. Et une nouvelle syntaxe pour while:

- while est une déclaration conditionnelle (elle nécessite un test pour retourner True), cela signifie que nous pouvons utiliser else dans une boucle while (mais pas for)

```
In [28]: # For Loops

for i, x in enumerate(range(2, 8, 2)):
    print(f"{i+1}. Range {x}")

# While Loops
count = 0
while count < 5:
    print(count)
    count += 1 # Une sténographie pour count = count + 1
else:
    print("Fin de la boucle")

1. Range 2
2. Range 4
3. Range 6
0
1
2
3
4
Fin de la boucle
```

Faites bien attention à l'indentation de cette boucle while. Que se passerait-il si le `count += 1` était en dehors de la boucle ?

Que se passe-t-il si vous devez quitter les boucles plus tôt ou si vous manquez une étape ?

Syntaxe

- `break` sort d'une boucle while ou for immédiatement
- `continue` saute la boucle de courante et revient à la boucle conditionnelle

```
In [29]: # Break and while conditional
print("Break et while conditionnelle")
count = 0
while True:
    # Vous pensez peut-être que ça va durer éternellement, mais...
    print(count)
    count += 1
    if count >= 5:
        break

# Continue
print("Continue")
for x in range(8):
    # Vérifier si x est irrégulier
    if (x+1) % 2 == 0:
        continue
    print(x)

Break et while conditionnelle
0
1
2
3
4
Continue
0
2
4
6
```

List comprehensions

L'une des tâches courantes du codage consiste à parcourir une liste d'éléments, à modifier ou à appliquer une forme d'algorithme et à renvoyer une nouvelle liste.

L'écriture de longues portions de code pour accomplir cette tâche est fastidieuse et prend du temps. Les 'list comprehensions' est un moyen efficace et concis d'y parvenir.

Par exemple, imaginez une phrase dans laquelle nous voulons compter la longueur de chaque mot, mais sans les "les" :

```
In [30]: sentence = "car la chanson et l'épée sont des droits de naissance vendus à un usurier, mais je suis le der
nier bandit de grand chemin solitaire et je suis le dernier aventurier"
words = sentence.split()
word_lengths = []
for word in words:
```

```

for word in words:
    if word not in ["la", "le"]:
        word_lengths.append(len(word))
print(word_lengths)

# La même chose peut être réalisée avec une liste de compréhension
word_lengths = [len(word) for word in sentence.split(" ") if word not in ["la", "le"]]
print(word_lengths)

[3, 7, 2, 6, 4, 3, 6, 2, 9, 6, 1, 2, 8, 4, 2, 4, 7, 6, 2, 5, 6, 9, 2, 2, 4, 7, 10]
[3, 7, 2, 6, 4, 3, 6, 2, 9, 6, 1, 2, 8, 4, 2, 4, 7, 6, 2, 5, 6, 9, 2, 2, 4, 7, 10]

```

Traitement des exceptions

Pour le reste de cette section du tutoriel, nous allons nous concentrer sur une syntaxe et une méthodologie plus avancées.

Dans Strings, nous vous avons présenté comment l'instruction de concaténer une chaîne avec un entier en utilisant l'opérateur + a entraîné une erreur :

```

In [32]: print(1 + "bonjour")

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-32-c2d1fcd3159a> in <module>
----> 1 print(1 + "bonjour")

TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

En Python, cela est connu comme une exception. L'exception particulière ici est une `TypeError`. L'obtention d'exceptions est essentielle au codage, car elle vous permet de corriger les erreurs de syntaxe, de détecter et éviter les petits problèmes lorsque vous passez les mauvaises variables, ou que votre code se comporte de manière inattendue.

Cependant, une fois que votre code est mis en production, les erreurs qui arrêtent complètement votre programme sont frustrantes pour l'utilisateur. Le plus souvent, il n'y a aucun moyen d'exclure les erreurs et parfois la seule façon de le savoir est d'essayer et de voir si la fonction provoque une erreur.

Beaucoup de ces erreurs sont entièrement attendues. Par exemple, si vous voulez qu'un utilisateur saisisse un nombre entier, vous voulez l'empêcher de taper du texte. Ou - à l'ère du piratage de masse - vous voulez empêcher un utilisateur d'essayer d'inclure son propre code dans un champ de texte.

Lorsque cela se produit, vous voulez pouvoir essayer d'exécuter votre code et ensuite détecter les exceptions prévues en toute sécurité.

Syntaxe

- Tester et récupérer les exceptions avec `try` et `except`
- Capturez des erreurs spécifiques, plutôt que toutes les erreurs, car vous devez toujours être au courant de tout ce qui est inattendu ... sinon vous pouvez passer des heures à essayer de trouver une erreur qui est délibérément ignorée par votre programme.
- Chaîne d'exceptions avec, par exemple, `exception (IndexError, TypeError)`. Voici un lien vers toutes les exceptions courantes (<https://docs.python.org/3/library/exceptions.html>).

```

In [33]: # Une "Erreur d'index" est lancée lorsque vous essayez d'adresser un index dans une liste qui n'existe pas
# Dans cet exemple, nous allons attraper cette erreur et faire autre chose

def print_list(l):
    """
    Pour une liste `l` donnée, de longueur inconnue, essayez d'imprimer la première
    10 éléments de la liste.

    Si la liste est inférieure à 10, complétez les éléments restants par `0`.
    """
    for i in range(10):
        try:
            print(l[i])
        except IndexError:
            print(0)

print_list([1,2,3,4,5,6,7])

1
2
3
4
5
6
7
0
0
0

```

Vous pouvez également déclencher délibérément une exception avec `raise`. Pour aller plus loin et rédiger vos propres types d'exceptions,

vous pouvez également déclencher délibérément une exception avec `raise`. Pour aller plus loin et définir vos propres types d'exceptions, consultez [cette explication](https://stackoverflow.com/a/26938914) (<https://stackoverflow.com/a/26938914>).

```
In [34]: def print_zero(zero):
          if zero != 0:
              raise ValueError("Not Zero!")
          print(zero)

          print_zero(10)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-34-a5b85de8b30c> in <module>
      4     print(zero)
      5
----> 6 print_zero(10)

<ipython-input-34-a5b85de8b30c> in print_zero(zero)
      1 def print_zero(zero):
      2     if zero != 0:
----> 3         raise ValueError("Not Zero!")
      4     print(zero)
      5

ValueError: Not Zero!
```

Sets

Les set sont des listes sans doublons. Vous pourriez probablement écrire un algorithme de tri, ou un dictionnaire, pour arriver au même résultat, mais les ensembles sont plus rapides et plus souples.

```
In [35]: # Extraire tous les termes uniques dans cette phrase

print(set("la pluie est humide et humide est la pluie".split()))

{'pluie', 'humide', 'et', 'la', 'est'}
```

Syntaxe

- Créer un ensemble unique de termes avec `set`
- Pour obtenir les membres d'un ensemble commun aux deux ensembles, utilisez `set1.intersection(set2)`
- Obtenez les membres uniques de chacun des ensembles et des autres, utilisez `set1.symmetric_difference(set2)`
- Pour obtenir les membres uniques d'une liste (celle sur laquelle s'applique la fonction `.`) par rapport à une autre (le paramètre de la fonction), utilisez `set1.difference(set2)`
- Pour obtenir tous les membres de chacune des deux listes, utilisez `set1.union(set2)`

```
In [40]: set_one = set(["Alice", "Carol", "Dan", "Eve", "Heidi"])
          set_two = set(["Bob", "Dan", "Eve", "Grace", "Heidi"])

          # Intersection
          print(f"Set One intersection: {set_one.intersection(set_two)}")
          print(f"Set Two intersection: {set_two.intersection(set_one)}")

          # Symmetric difference
          print(f"Set One symmetric difference: {set_one.symmetric_difference(set_two)}")
          print(f"Set Two symmetric difference: {set_two.symmetric_difference(set_one)}")

          # Difference
          print(f"Set One difference: {set_one.difference(set_two)}")
          print(f"Set Two difference: {set_two.difference(set_one)}")

          # Union
          print(f"Set One union: {set_one.union(set_two)}")
          print(f"Set Two union: {set_two.union(set_one)}")

          Set One intersection: {'Eve', 'Dan', 'Heidi'}
          Set Two intersection: {'Eve', 'Dan', 'Heidi'}
          Set One symmetric difference: {'Alice', 'Carol', 'Grace', 'Bob'}
          Set Two symmetric difference: {'Alice', 'Carol', 'Grace', 'Bob'}
          Set One difference: {'Alice', 'Carol'}
          Set Two difference: {'Grace', 'Bob'}
          Set One union: {'Alice', 'Dan', 'Heidi', 'Bob', 'Eve', 'Carol', 'Grace'}
          Set Two union: {'Alice', 'Dan', 'Heidi', 'Bob', 'Eve', 'Carol', 'Grace'}
```

Et cela marque la fin de cette section du tutoriel.

2.1.4 Python intermédiaire

Le code de la section 2.1.3 Les bases du python a été limité à de courts extraits. La résolution de problèmes plus complexes implique un code plus

Le code de la section [2.1.3 Les bases du python](#) a été limité à de courts extraits. La résolution de problèmes plus complexes implique un code plus complexe s'étendant sur des centaines, voire des milliers, de lignes, et - si vous voulez réutiliser ce code - il n'est pas pratique de le copier-coller plusieurs fois. Pire encore, toute erreur est amplifiée, et toute modification devient fastidieuse à gérer.

Il serait de loin préférable d'écrire un module pour résoudre cette tâche, l'obtenir de manière absolument parfaite, et l'appeler chaque fois que vous voulez que le même problème soit résolu ou que la même tâche soit exécutée.

Dans les logiciels, ce processus de regroupement de fonctions individuelles dans leur propre code modulaire est appelé "abstraction". Un système logiciel complet est constitué d'un certain nombre de modules qui interagissent ensemble pour produire une application intégrée.

En Python, ces modules sont appelés fonctions (functions) et une suite complète de fonctions regroupées autour d'un ensemble de tâches connexes est appelée librairie (library) ou module. Les bibliothèques vous permettent d'hériter d'une grande variété de solutions logicielles puissantes développées et maintenues par d'autres personnes.

Python est [open source](https://en.wikipedia.org/wiki/Open-source_software) (https://en.wikipedia.org/wiki/Open-source_software), ce qui signifie que son code source est publié sous une licence qui permet à quiconque d'étudier, de modifier et de distribuer le logiciel à n'importe quel but. De nombreuses bibliothèques Python parmi les plus populaires sont également à code source ouvert. Il existe des milliers de bibliothèques partagées que vous pouvez utiliser et, peut-être, lorsque vous vous sentez suffisamment confiant, auxquelles vous pouvez contribuer avec votre propre code.

Fonctions

Les fonctions sont des modules de code appelables, certains avec des paramètres ou des arguments (variables que vous pouvez passer à la fonction), qui effectuent une tâche et peuvent renvoyer une valeur. Elles constituent un moyen pratique d'empaqueter du code en blocs indépendants, ce qui rend l'ensemble de votre programme plus lisible, réutilisable et permet de gagner du temps.

Vous pouvez aussi facilement partager vos fonctions avec d'autres personnes, ce qui leur fait également gagner du temps. J'en ai glissé un dans la section sur la **gestion des exceptions** tout à l'heure.

Syntaxe

- Vous structurez une fonction en utilisant `def`, comme suit:

```
def nom_de_la_fonction(paramètres) :
    code
    return response
```
- `return` est optionnel, mais vous permet de renvoyer les résultats de toute tâche effectuée par la fonction au point où la fonction a été appelée
- Pour vérifier si un objet est une fonction (c'est-à-dire s'il peut être appelé), utilisez `callable`, par exemple
`callable(nom_de_la_fonction)` renverra `True`

```
In [36]: # Une fonction simple et sans arguments
def say_hello():
    print("Hello, World!")

# L'appeler est aussi simple que cela
say_hello()

# Et vous pouvez tester que c'est un appel
print(callable(say_hello))

Hello, World!
True
```

Un argument peut être n'importe quelle variable, telle que des entiers, des chaînes de caractères, des listes, des dictionnaires ou même d'autres fonctions. C'est là que vous commencez à réaliser l'importance de laisser des commentaires et des explications dans votre code, car vous devez vous assurer que toute personne utilisant une fonction sait quelles variables la fonction attend, et dans quel ordre.

Les fonctions peuvent également effectuer des calculs et les renvoyer à ce qu'on appelle les fonctions.

```
In [37]: # Une fonction avec deux arguments de string
def say_hello_to_user(username, greeting):
    # Renvoie un message de salutation à un nom d'utilisateur
    print(f"Salut, {username}! J'espère que vous aurez un grand {greeting}.")

# Appelez-Le
say_hello_to_user("Jill", "jour")

# Effectuer un calcul et le renvoyer
def sum_two_numbers(x, y):
    # Retourne la somme de x + y
    return x + y

sum_two_numbers(5, 10)

Salut, Jill! J'espère que vous aurez un grand jour.
```

Out[37]: 15

Vous pouvez voir que l'échange `username` et `greeting` dans la fonction `say_hello_to_user` serait déroutant, mais l'échange des chiffres dans `sum_two_numbers` ne poserait pas de problème.

Non seulement vous pouvez appeler des fonctions à partir de fonctions, mais vous pouvez aussi créer des variables qui sont des fonctions, ou le résultat de fonctions.

```
In [38]: def number_powered(number, exponent):
# Renvoie Le nombre à La puissance de L'exposant
return number ** exponent

# Jupyter garde à disposition des fonctions qui ont été appelées dans d'autres cellules
# Cela signifie que `sum_two_numbers` est toujours disponible
def sum_and_power(number1, number2, exponent):
# Renvoie deux nombres additionnés, puis à un exposant
summed = sum_two_numbers(number1, number2)
return number_powered(summed, exponent)

# Appelez-Le `sum_and_power`
print(sum_and_power(2, 3, 4))

625
```

Grâce à une dénomination soignée et à de nombreux commentaires, vous pouvez voir comment vous pouvez rendre votre code extrêmement lisible et explicite.

Une meilleure façon d'écrire les commentaires dans les fonctions est appelée docstrings.

Syntaxe

- Docstrings s'écrit comme un texte structuré entre trois séries de guillemets, par exemple `""" Il s'agit d'une chaîne documentaire """`
- Vous pouvez accéder à la chaîne de documents d'une fonction en appelant `function.__doc__`

```
In [39]: def docstring_example():
"""
Un exemple de fonction qui renvoie `True`.
"""
return True

# Impression de La docstring
print(docstring_example.__doc__)

# Appelez-La
print(docstring_example())

Un exemple de fonction qui renvoie `True`.

True
```

Classes et objets

Un objet Python complet est une encapsulation de variables et de fonctions en une seule entité. Les objets obtiennent leurs variables et leurs fonctions à partir de classes.

Les classes sont l'endroit où la plupart des actions se déroulent en Python et le codage consiste, en grande partie, à produire et à utiliser des classes pour effectuer des tâches.

Une classe très basique ressemblerait à ceci :

```
In [40]: class myClass:
"""
Une classe de démonstration.
"""
my_variable = "Regardez, une variable !"

def my_function(self):
"""
Une fonction de classe de démonstration.
"""
return "Je suis une fonction de la classe !"

# Vous appelez une classe en créant un nouvel objet de classe
new_class = myClass()

# Vous pouvez accéder aux variables de classe ou aux fonctions avec un appel pointillé, comme suit
print(new_class.my_variable)
print(new_class.my_function())

# Accéder à La classe docstrings
print(myClass.__doc__)
print(myClass.my_function.__doc__)

Regardez, une variable !
```

Je suis une fonction de la classe !

Une classe de démonstration.

Une fonction de classe de démonstration.

Résumons la nouvelle syntaxe.

Syntaxe

- Vous instanciez une classe en l'appelant `class()`. Si vous n'appellez que `class`, sans les parenthèses, vous accédez à l'objet lui-même. C'est également utile, et cela signifie que vous pouvez faire circuler les classes comme vous le feriez avec des variables.
- Toutes les variables et fonctions d'une classe sont accessibles par l'appel, `.function()` ou `.variable`. Vous pouvez même ajouter de nouvelles fonctions et variables à une classe que vous avez créée. N'oubliez pas, cependant, qu'elles n'existeront pas dans les nouvelles classes que vous créez puisque vous n'avez pas modifié le code sous-jacent.
- Les fonctions au sein d'une classe nécessitent un argument de base qui, par convention, est appelé `self`. Il y a une explication complexe à la nécessité de `self`, mais - brièvement - considérez que c'est l'instance de l'objet lui-même. Ainsi, à l'intérieur de la classe, `self.function` est la façon dont la classe appelle ses composants fonctions.
- Vous pouvez également accéder aux docstrings comme vous le feriez auparavant.

```
In [41]: # Ajouter une nouvelle variable à une instance de classe
new_class1 = myClass()
new_class1.my_variable2 = "Salut, Bob!"
print(new_class1.my_variable2, new_class1.my_variable)

# Mais, essayer d'accéder à ma_variable2 dans new_class provoque une erreur
print(new_class.my_variable2)

Salut, Bob! Regardez, une variable !
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-41-e834e768c0bc> in <module>
      5
      6 # Mais, essayer d'accéder à ma_variable2 dans new_class provoque une erreur
----> 7 print(new_class.my_variable2)

AttributeError: 'myClass' object has no attribute 'my_variable2'
```

Les classes peuvent s'initialiser avec un ensemble de variables disponibles. Cela rend le référencement `self` plus explicite, et vous permet également de transmettre des arguments à votre classe pour fixer les valeurs initiales.

Syntaxe

- Initialiser une classe avec la fonction spéciale `def __init__(self)`
- Passez les arguments à vos fonctions avec `__init__(self, arguments)`
- Nous pouvons également différencier les arguments, et les arguments par mots clés:
 - **arguments** : ils sont transmis de la manière habituelle, sous la forme d'un seul terme, par exemple `ma_fonction(argument)`.
 - **argument_mot_clé** : ils sont transmis de la manière dont on pense à un dictionnaire, par exemple `ma_fonction(argument_mot_clé = valeur)`. C'est aussi une façon d'initialiser un argument avec une valeur par défaut. Si vous laissez de côté l'argument lorsqu'il a une valeur par défaut, il s'appliquera sans que la fonction ne soit défaillante.
 - Les fonctions ont souvent besoin qu'on leur transmette de nombreux arguments et mots-clés, et cela peut devenir compliqué. Vous pouvez aussi penser à une liste d'arguments comme une liste, et une liste d'arguments de mots-clés comme un dictionnaire. Une façon plus simple de traiter ce problème est de référencer vos arguments et mots-clés comme ceci : `ma_fonction(*args, **kwargs)` où `*args` sera disponible pour la fonction sous forme de liste ordonnée, et `**kwargs` sous forme de dictionnaire.

```
In [42]: # Une démonstration de tous ces nouveaux concepts

class demoClass:
    """
    Une classe de démonstration avec une fonction __init__, et une fonction qui prend Les args et Les kwar
    gs.
    """

    def __init__(self, argument = None):
        """
        Une fonction qui est appelée automatiquement Lorsque La demoClass est initialisée.
        """
        self.demo_variable = "Hello, World!"
        self.initial_variable = argument

    def demo_class(self, *args, **kwargs):
        """
```

```
.....  
Une classe de démo qui passe en boucle tous Les args et kwargs fournis et Les imprime.  
"""  
for i, a in enumerate(args):  
    print("Arg {}: {}".format(i+1, a))  
for k, v in kwargs.items():  
    print("{} - {}".format(k, v))  
if kwargs.get(self.initial_variable):  
    print(self.demo_variable)  
return True  
  
demo1 = demoClass()  
demo2 = demoClass("Bob")  
  
# Qu'est-ce qui a été initialisé dans chaque objet de démonstration ?  
print(demo1.demo_variable, demo1.initial_variable)
```