# [Initiez-vous à Python pour l'analyse de données](https://openclassrooms.com/fr/courses/6204541-initiez-vous-a-python-pour-lanalyse-de-donnees)

Table des matières

[Initiez-vous à Python pour l'analyse de données 1](#_Toc62110420)

[Installez Python et Anaconda 5](#_Toc62110421)

[Tirez le meilleur parti de ce cours 5](#_Toc62110422)

[Découvrez Python 5](#_Toc62110423)

[Installez Python via Anaconda 5](#_Toc62110424)

[Faites vos premiers pas avec Jupyter Notebook 7](#_Toc62110425)

[Découvrez le notebook Jupyter 7](#_Toc62110426)

[Faites vos premiers pas avec Jupyter 8](#_Toc62110427)

[Documentez votre notebook 8](#_Toc62110428)

[**Déclarez une variable** 11](#_Toc62110429)

[**Qu'est-ce qu'une variable ?** 11](#_Toc62110430)

[**Comment créer une variable ?** 12](#_Toc62110431)

[**Qu'est-ce qu'une opération entre variables ?** 13](#_Toc62110432)

[**Au-delà de l’arithmétique** 15](#_Toc62110433)

[**En résumé** 16](#_Toc62110434)

[**Manipulez les types de variables** 18](#_Toc62110435)

[**Pourquoi avons-nous besoin des types de variables ?** 18](#_Toc62110436)

[**Les variables numériques** 18](#_Toc62110437)

[**Les chaînes de caractères** 20](#_Toc62110438)

[**En résumé** 21](#_Toc62110439)

[**Écrivez vos propres fonctions** 22](#_Toc62110440)

[**Découvrez les fonctions** 22](#_Toc62110441)

[**Écrivez vos propres fonctions** 22](#_Toc62110442)

[**Utilisez la fonction help en cas d'oubli** 25](#_Toc62110443)

[**En résumé** 26](#_Toc62110444)

[Programmez-en orienté objet 27](#_Toc62110445)

[**Les classes : des modèles d’objets** 27](#_Toc62110446)

[**Focus sur les méthodes** 28](#_Toc62110447)

[**Les méthodes de chaînes de caractères** 29](#_Toc62110448)

[En résumé 30](#_Toc62110449)

[Manipulez les fonctions et objets fondamentaux 31](#_Toc62110450)

[Compétences évaluées 31](#_Toc62110451)

[ Question 1 31](#_Toc62110452)

[ Question 2 31](#_Toc62110453)

[ Question 3 32](#_Toc62110454)

[ Question 4 32](#_Toc62110455)

[ Question 5 33](#_Toc62110456)

[ Question 6 33](#_Toc62110457)

[ Question 7 34](#_Toc62110458)

[ Question 8 34](#_Toc62110459)

[ Question 9 34](#_Toc62110460)

[ Question 10 35](#_Toc62110461)

[**Créez des tableaux pour stocker vos variables** 37](#_Toc62110462)

[**Déclarez une liste pour stocker vos éléments** 37](#_Toc62110463)

[**Ajoutez ou supprimez des éléments : les méthodes de listes** 39](#_Toc62110464)

[**Utilisez les dictionnaires** 42](#_Toc62110465)

[**En résumé** 44](#_Toc62110466)

[**Contrôlez votre code grâce aux structures conditionnelles** 45](#_Toc62110467)

[**Affichez des informations *si* elles sont disponibles** 45](#_Toc62110468)

[**Utilisez les booléens : le type sans demi-mesure** 46](#_Toc62110469)

[**Gérez un enchaînement de conditions** 50](#_Toc62110470)

[**En résumé** 51](#_Toc62110471)

[**Choisissez la boucle adéquate** 52](#_Toc62110472)

[**“Bouclez” un nombre de fois fixé avec la boucle FOR** 52](#_Toc62110473)

[**“Bouclez” selon une condition avec la boucle WHILE :** 54](#_Toc62110474)

[**Passez certaines instructions au sein de votre boucle** 56](#_Toc62110475)

[**En résumé** 57](#_Toc62110476)

[Structurez votre code au sein d’un projet 58](#_Toc62110477)

[Compétences évaluées 58](#_Toc62110478)

[ Question 1 58](#_Toc62110479)

[ Question 2 58](#_Toc62110480)

[ Question 3 59](#_Toc62110481)

[ Question 4 59](#_Toc62110482)

[ Question 5 60](#_Toc62110483)

[ Question 6 61](#_Toc62110484)

[ Question 7 62](#_Toc62110485)

[ Question 8 63](#_Toc62110486)

[ Question 9 63](#_Toc62110487)

[ Question 10 64](#_Toc62110488)

[**Prenez en main les modules et librairies Python** 66](#_Toc62110489)

[**Un module en Python** 66](#_Toc62110490)

[**Quand un module ne suffit pas : les packages** 68](#_Toc62110491)

[**Les packages dans l’analyse de données** 70](#_Toc62110492)

[**En résumé** 70](#_Toc62110493)

[**Manipulez des nombres aléatoires avec le module random** 71](#_Toc62110494)

[**Générez des nombres aléatoires** 71](#_Toc62110495)

[**Choisissez aléatoirement dans une liste : sous-échantillonnage** 73](#_Toc62110496)

[**Pour aller plus loin** 74](#_Toc62110497)

[**En résumé** 75](#_Toc62110498)

[Utilisez des librairies Python spécialisées 76](#_Toc62110499)

[Compétences évaluées 76](#_Toc62110500)

[ Question 1 76](#_Toc62110501)

[ Question 2 76](#_Toc62110502)

[ Question 3 77](#_Toc62110503)

[ Question 4 78](#_Toc62110504)

[ Question 5 80](#_Toc62110505)

[ Question 6 81](#_Toc62110506)

[ Question 7 82](#_Toc62110507)

[ Question 8 83](#_Toc62110508)

Pour créer des **programmes informatiques** ou construire des **algorithmes destinés à l’analyse de données,** vous devez maîtriser un **langage de programmation**.

Un programme dit à un ordinateur quoi faire, grâce à des données et des instructions que vous allez développer ! Alors, par où commencer pour écrire vos propres programmes en Python ?

Ce cours va vous apprendre les bases du **langage de programmation Python**, nécessaire pour évoluer dans le domaine de l’**analyse de données**.

Vous apprendrez notamment à **utiliser et écrire les fonctions** de base de Python. Vous manipulerez les **objets les plus courants** de ce langage, et mettrez en œuvre une**logique de programmation**. Votre programme fera exactement ce que vous lui direz de faire !

Prêt à vous initier à Python pour l’analyse de données ?

**Objectifs pédagogiques :**

* Mettre en place votre environnement de travail Python
* Manipuler les fonctions et objets fondamentaux
* Structurer votre code au sein d’un projet
* Utiliser des librairies Python spécialisées

**Prérequis :**

Aucun ! :soleil:

**Outils nécessaires :**

* Un terminal
* Anaconda
* Jupyter Notebook

## Installez Python et Anaconda

Bonjour et bienvenue ! Je m'appelle Alexis Perrier. Je suis un data scientist, c'est moi qui vais vous accompagner tout au long de ce cours. Envie d'en savoir plus ? Rendez-vous sur Twitter pour suivre mes réflexions relatives à la data : @alexip ^^

### Tirez le meilleur parti de ce cours

Dans ce cours, vous allez voir l’ensemble des bases du langage Python nécessaires avant de vous lancer dans l’analyse de données.

Ce cours comportera plusieurs consoles interactives en ligne qui vous permettront de tester les différentes notions apprises. Néanmoins, nous vous recommandons fortement d’installer Anaconda (incluant Python et Jupyter Notebook) pour tester les différents exemples de votre côté !

Vous allez donc voir dans un premier temps comment installer Anaconda.

### Découvrez Python

Python est un langage de programmation assez généraliste, c’est-à-dire qu’il est à peu près possible de tout faire avec : des sites et applications web, des applications mobiles, des scripts personnels, des applications de bureau, de l’analyse de données et même des jeux vidéo !

Par exemple :

* **Instagram** est codé en Python ;
* c’est l’un des langages principaux utilisés par les ingénieurs de chez **Google** ;
* **Netflix** utilise Python pour l’élaboration de ses algorithmes de recommandation ;
* l’application bureau de **Dropbox** est développée en Python ;
* **Reddit** est codé en Python.

Concernant la data, Python est le langage le plus utilisé, en particulier pour les traitements et manipulations de données, la data visualisation et le développement d’algorithmes d’intelligence artificielle.

### Installez Python via Anaconda

Anaconda est une distribution scientifique de Python : c’est-à-dire qu’en installant Anaconda, vous installerez Python, Jupyter Notebook (que nous présenterons plus en détail au prochain chapitre) et des dizaines de packages scientifiques, dont certains indispensables à l’analyse de données !

Pour commencer, téléchargez la distribution Anaconda correspondant à votre système d’exploitation, en Python version 3 : <https://www.anaconda.com/distribution/>

#### Installez Anaconda sur Windows ou Mac

* Téléchargez l’installeur Windows ou macOS, et double-cliquez pour lancer l’installation.
* Répondez aux différentes questions (les options par défaut suffisent !). Une fois l’installation terminée, vous pouvez vérifier que celle-ci s’est bien passée en lançant l’application Jupyter Notebook.

#### Installez Anaconda sur Linux

* Téléchargez l’installeur pour Linux.
* Entrez l’instruction suivante dans la commande :

bash ~/Downloads/Anaconda3-5.3.0-Linux-x86\_64.h

* L’installeur affiche : “In order to continue the installation process, please review the license agreement.” Affichez les termes de la licence d’utilisation, scrollez tout en bas et cliquez sur yes pour accepter.
* Répondez "yes" à la question qui suit :

Do you wish the installer to prepend the Anaconda3 install location

to PATH in your /home/ec2-user/.bashrc ? [yes|no]

* La commande précédente a ajouté au fichier .bashrc le chemin (le dossier) dans lequel se trouve Anaconda. Ainsi, vous pourrez lancer Anaconda directement en tapant seulement "anaconda" dans votre console ! Cette opération prendra effet au redémarrage de votre ordinateur, ou dès l’exécution de cette commande :

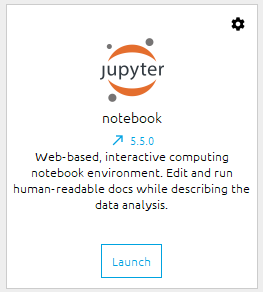
source .bashrc

* Vérifiez que tout s’est bien passé en lançant le programme Jupyter (voir ci-dessous).

#### Lancez Jupyter

* Sous macOS, lancez Anaconda Navigator via le Launchpad.
* Sous Linux, ouvrez une console et lancez : anaconda-navigator.
* Sous Windows, lancez Anaconda Navigator depuis votre liste de programmes.

Une fois cela fait, vous allez avoir plusieurs applications à disposition. Vous allez vous intéresser à celle s’appelant Jupyter Notebook :



Lancez votre premier notebook Jupyter !

Maintenant qu’Anaconda et Python sont installés, vous allez découvrir l’un des outils de programmation les plus utilisés en analyse de données. Simple d’utilisation, pratique et puissant, faites la connaissance de Jupyter Notebook.

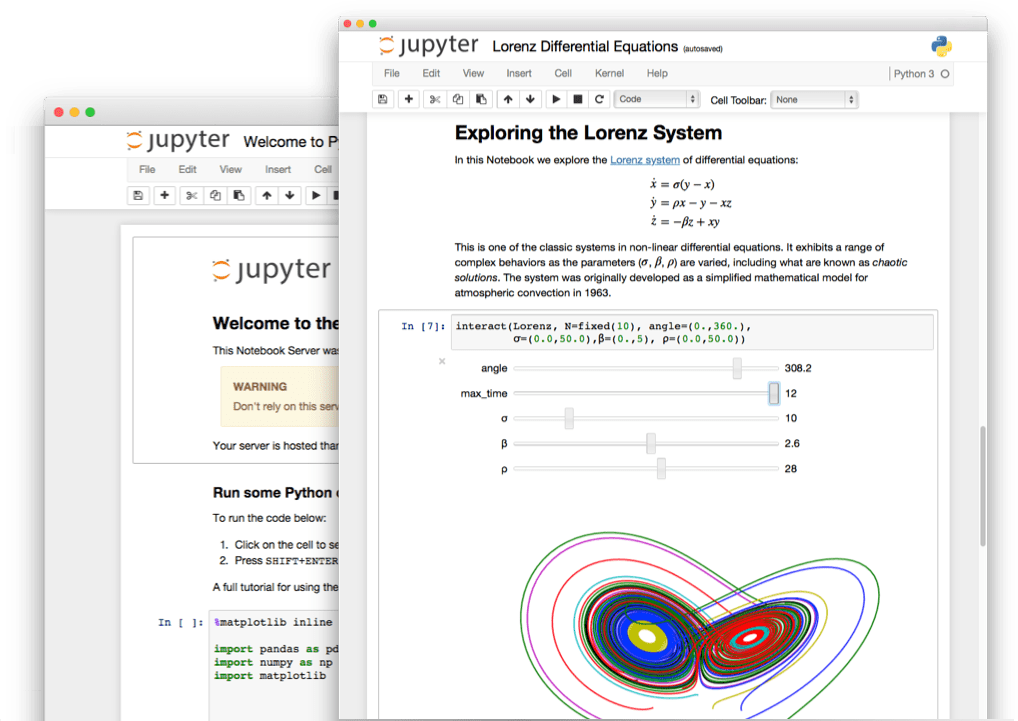
## Faites vos premiers pas avec Jupyter Notebook

Jupyter Notebook est une application web qui vous permet de stocker des lignes de code Python, les résultats de l’exécution de ces dernières (graphiques, tableaux, etc.) et du texte formaté. Cela peut être comparé à une page web... contenant du code Python ! Un peu comme ce cours, en fin de compte.

Il y a bien sûr d’autres moyens et applications permettant de travailler avec Python. Mais nous allons vous expliquer pourquoi Python est particulièrement bien adapté à l’analyse de données et pourquoi nous vous le recommandons !

### Découvrez le notebook Jupyter

Jupyter Notebook est un outil puissant qui permet aux utilisateurs du langage Python de créer et de partager des documents interactifs contenant du code dynamique et exécutable, des visualisations de contenus, des textes de documentation et des équations. Le terme "notebook" est lié au caractère intrinsèque de l’outil qui permet d’écrire des petits bouts de code exécutable (appelés "cellules"), de les documenter pour expliquer ce qu’ils font et d’afficher les données résultant de leur exécution. Tout cela est stocké dans un document partageable avec d’autres utilisateurs.



C’est donc particulièrement pratique pour prototyper des algorithmes ou tester des bouts de code, afin d’analyser les résultats et éventuellement de les ajouter à votre projet principal. Avec un notebook, pas besoin d’organiser son code en fonction, avec un main, etc. Vous avez juste besoin d’écrire votre code dans les blocs prédéfinis et de l’exécuter, pour "faire tourner du Python".

#### Jupyter Notebook pour l’analyse de données

Jupyter Notebook est une application très populaire pour l’ensemble des personnes travaillant dans l’analyse de données. Il permet à une équipe de pouvoir créer et partager des rapports sous forme de document/code, facilitant ainsi la productivité et la collaboration.

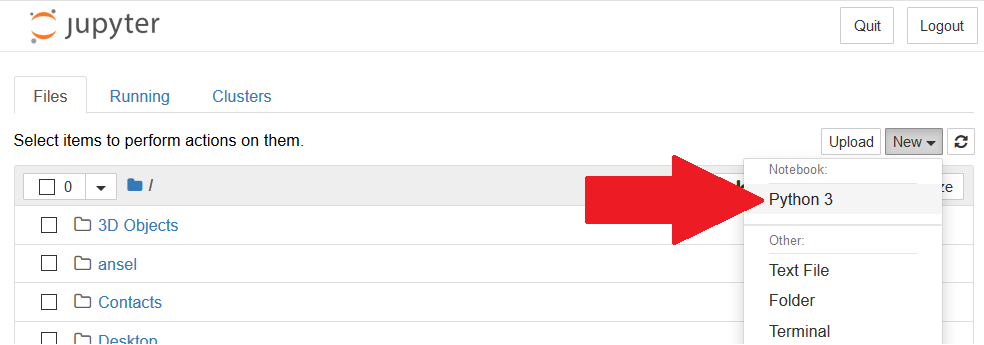
Par exemple, les data scientists de Netflix travaillent leurs données via Jupyter Notebook : leur algorithme de recommandation de films est codé à partir de Jupyter ! Si vous lisez un peu l’anglais, vous pouvez jeter un coup d’œil à cet article sur [Medium](https://medium.com/netflix-techblog/notebook-innovation-591ee3221233) qui en parle en détail.

### Faites vos premiers pas avec Jupyter

Maintenant que Jupyter est installé et que nous avons un peu plus appréhendé son utilité, il est temps de faire tourner nos premiers codes Python.

Tout au long de ce cours, vous trouverez des consoles Python interactives. Vous pouvez naturellement les utiliser pour réaliser les différents exercices, mais nous vous conseillons fortement d’ouvrir Jupyter en parallèle pour tester par vous-même les codes qui vous sont présentés.

Premièrement, ouvrez Jupyter Notebook : cela va ouvrir un nouvel onglet dans votre navigateur Internet appelé **Home**. Commencez par créer un nouveau notebook en cliquant sur New puis Python 3 :



Cela va normalement encore ouvrir un nouvel onglet dans lequel se trouvent, de haut en bas : le nom de votre notebook (que vous pouvez modifier en cliquant dessus), une barre de menu, une barre d’outils et une cellule vide. Essayez d’écrire dans cette cellule votre première ligne de code et cliquez ensuite sur Run. Dire "Bonjour" serait par exemple un bon début. ;)

print('Hello World !')

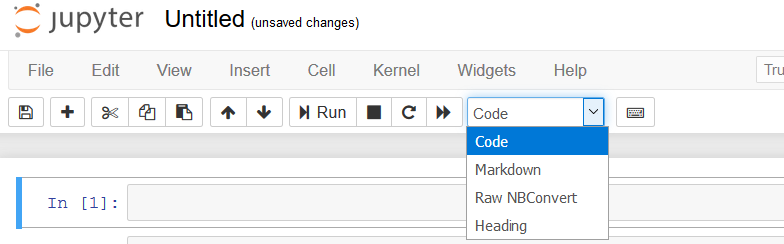
"Hello World !" apparaît normalement en dessous de votre cellule : c’est le résultat de l’exécution de cette dernière. Félicitations, vous venez de faire tourner votre tout premier code Python !

Saviez-vous qu’afficher ou écrire (ici print) "Hello World" est la première chose que vous apprenez lors de l’apprentissage de n’importe quel langage de programmation ?

### Documentez votre notebook

Avant de vraiment plonger dans la programmation, vous devez savoir qu’il existe 4 types de cellules différentes avec Jupyter : **code, markdown, row nbconvert** et **heading**.

Lorsque vous cliquez sur une cellule, vous pouvez changer le type simplement en le sélectionnant dans la liste, comme dans l’exemple ci-dessous :



Voyons plus en détail chaque type :

* **Code** – la cellule de code classique. Celle-ci est réservée pour l’écriture et l’exécution de code Python ! Vous pouvez exécuter votre code en cliquant sur le bouton Run, comme vu précédemment.
* **Heading** – ce type est légèrement obsolète. Jusqu’alors, il servait pour définir des titres, mais les Markdown couvrent aujourd’hui cette fonctionnalité. Il est amené à disparaître dans les prochaines versions de Jupyter.
* **Raw nbconvert** – permet de contrôler le formatage du document lors d’une conversion du notebook dans un autre format. Vous n’allez pas être amené à l’utiliser dans le cadre de ce cours.
* **Markdown** – cellule de texte qui sert essentiellement à la documentation du notebook, pour y rédiger des commentaires, des titres, des équations, etc. Ce type permet de structurer votre texte en utilisant les balises HTML ou la syntaxe Markdown.

Que ce soit pour du Markdown ou du code, il suffit d’écrire votre texte/code et de l’exécuter via le bouton Run pour voir le résultat.

On pourrait par exemple créer un titre. Pour cela, il suffit d’ajouter un # devant votre texte. De façon similaire à ce que l’on peut avoir avec un logiciel de traitement de texte, un seul # correspondra à un titre de niveau 1, deux # à un titre de niveau 2, etc.

Vous pouvez aussi décider de mettre votre texte en gras ou en italique. Pour cela, il suffit tout simplement d’entourer le texte souhaité :

* par \*\* pour mettre en gras ;
* par une simple \* pour mettre en italique.

Par exemple, la cellule Markdown suivante :

# Un titre de niveau 1

Voici un peu de texte pour présenter un markdown

## titre de niveau 2

\*\*Ceci\*\* est très important

## autre titre de niveau 2

Mais \*cela\* également !

donnera le résultat suivant :



Si vous souhaitez aller plus loin dans les fonctionnalités des cellules Markdown, je vous conseille de suivre [ce lien](https://guides.github.com/features/mastering-markdown/).

Maintenant que votre environnement Python est configuré, vous êtes prêt à rentrer dans le vif du sujet !

**Déclarez une variable**

Vous savez afficher un texte à l’écran. Très bien. C’est un début, mais vous serez capable de faire bien plus après avoir vu ce que l’on appelle **les *variables*** en programmation.

**Qu'est-ce qu'une variable ?**

Les variables sont l’un des concepts qui se retrouvent dans la totalité des langages de programmation. Autant dire que sans variable, on ne peut pas programmer, et ce n’est pas une exagération.

Comme mentionné dans la vidéo, considérez une variable comme une sorte de boîte contenant une **valeur**. Cette boîte est elle-même stockée sur une étagère parmi tant d’autres, dans un gigantesque entrepôt. Néanmoins, l’emplacement de chaque boîte est très précisément répertorié, tout comme votre ordinateur enregistre l’emplacement de votre variable en mémoire.



Boîtes stockées dans un entrepôt

Une **valeur** est une partie de vos données que vous allez stocker dans une variable. Pour revenir à l’analogie avec l’entrepôt, plusieurs boîtes permettent de stocker différentes valeurs. Par exemple, dans l’optique où vous travaillerez dans une banque, on peut imaginer que l’on veuille stocker dans différentes boîtes des informations à propos d’un client, comme le solde de son compte épargne ou celui de son compte courant. On va également avoir besoin d’appliquer différentes opérations sur ces boîtes comme les vider, ajouter de l’argent, transférer le contenu de l’une à l’autre, etc. Les variables vont vous permettre de faire cela !

Pour vous rappeler du contenu de chaque boîte, vous aurez besoin de les étiqueter. Le procédé est similaire en programmation : à chaque variable est attribué un **nom**.



Bocaux étiquetés contenant vos économies

De la même façon qu’avec l’étiquetage des boîtes, le nom d’une variable doit toujours représenter son contenu. Voici quelques recommandations générales concernant le choix des noms de vos variables :

* **Utiliser des noms de variables clairs**  
  Cela paraît fastidieux à faire, mais c’est réellement profitable pour vous et les personnes avec qui vous serez amené à partager votre code : cela permet la facilitation de la relecture et la maintenance de votre code. Par exemple, compteEpargne et compteCourant sont des noms bien plus explicites que compte1 et compte2.
* **Utiliser des noms de variables explicites**  
  Évitez les abréviations et acronymes si possible, même si une abréviation peut sembler évidente. Par exemple, revenuAnnuel est mieux que revAnnuel.
* **Suivre une convention typographique**  
  Une des conventions typographiques les plus communes est appelée **camel case** (aussi connu sous le nom de *camel caps*) – elle consiste à écrire les noms de variables contenant plusieurs mots sans espace ni ponctuation : le premier mot est écrit en minuscule, ensuite chaque mot est écrit avec la première lettre en majuscule, comme présenté ci-dessus.

**Comment créer une variable ?**

Avant de pouvoir l’utiliser, il faut forcément créer votre variable ! Il suffit d’associer une valeur à un nom pour créer une variable ; Python se charge tout seul de créer une boîte de la bonne taille. Pratique, non ?

Il y a plusieurs types de variables en Python, représentatifs de la réalité : du texte, des valeurs numériques, des tableaux, etc. Les variables numériques sont déclarées en associant un nom à une **valeur numérique**. Par exemple :

compteCourant = 500

compteEpargne = 1000

Vous avez ici déclaré deux variables : compteCourant et compteEpargne, en y stockant respectivement les valeurs 500 et 1.000.

Vous n’avez utilisé ici que des entiers ! Mais il est tout à fait possible de stocker des nombres décimaux.

Si vous stockez une nouvelle valeur dans une variable existante, le type de la variable va changer de lui-même en fonction de la valeur stockée dedans.

**Qu'est-ce qu'une opération entre variables ?**

Comme son nom le suggère, une variable peut varier, ou plutôt **la valeur d’une variable peut changer**. Vous pouvez faire cela via différentes **opérations**. En considérant les deux variables précédemment déclarées dans le dernier exemple, vous pourriez :

* ajouter un peu d’argent à votre compte épargne ;
* en retirer de votre compte courant ;
* calculer combien de temps il vous faudrait pour atteindre 5 000 € si vous économisez 500 € tous les mois ;
* calculer à combien serait votre compte courant si vous y ajoutiez 30 € tous les jours pendant une semaine ;
* calculer à combien serait votre compte courant si vous dépensiez 10 € tous les jours ;
* etc.

En bref, des problèmes concrets résolubles via Python. Chaque opération va faire appel à des opérateurs arithmétiques :

* + : addition ;
* - : soustraction ;
* \* : multiplication ;
* / : division.

Les règles de calcul arithmétique s’appliquent en Python, en particulier [l’ordre des opérations](https://fr.wikipedia.org/wiki/Ordre_des_op%C3%A9rations). Mais comme en mathématiques classiques, vous pouvez utiliser des parenthèses pour réorganiser l’ordre des calculs. Voyez comment réaliser cela en Python :

# ajouter 100 à notre épargne (Yeah!)

compteEpargne = compteEpargne + 100

# enlever 50 de notre compte courant (Snif)

compteCourant = compteCourant - 50

# calculer le nombre de jours à économiser pour atteindre 5000€

nombreJoursEconomiser = (5000 - compteCourant) / 500

# mettre à jour notre compte courant (encore) après les gains/pertes journaliers

compteCourant = compteCourant + (30 - 10) \* 7

Cela forme un chouette bout de code, non ? Si vous prêtez attention, vous noterez que la couleur n’est pas la même partout. C’est parce que votre code est composé de **commentaires** et **d’expressions** :

* les lignes commençant avec un # sont des **commentaires**. Elles sont utilisées pour documenter votre code et aider d’autres personnes à le comprendre ;
* les autres lignes de codes (qui permettent d’exécuter des opérations, déclarations, etc.) sont des **expressions**. Elles permettent de *dire* à l’ordinateur quoi faire.

Ici, chacune des expressions **affecte** une valeur à une variable. L’opérateur d’affectation est =.

1. À la droite de l’opérateur d’affectation se trouve l’expression qui va créer/calculer une **valeur**.
2. À la gauche de l’opérateur d’affectation, vous écrivez**le nom de la variable** à laquelle vous allez assigner la valeur correspondante.

Pour résumer tout cela, pour affecter une valeur à une variable, vous écrivez une expression. Cette expression se construit avec le nom de la variable, suivi de l’opérateur d’affectation et finalement de la valeur à associer.

Il existe également d’autres opérateurs arithmétiques un peu plus complexes, mais pas moins utiles pour autant, comme :

* % modulo : retourne le reste de la division euclidienne ;
* \*\* puissance : permet d’élever un nombre à une certaine puissance ;
* // division entière : calcule la division entière (chiffre arrondi à l’inférieur).

print(compteEpargne % 500) # -> 100

# 1100 = 500 \* 2 + 100, donc 1100 % 500 = le reste = 100

print(9 \*\* 3) # -> 729, 9\*9\*9 = 729

print(compteEpargne // 500) # -> 2

# 1100 = 500 \* 2 + 100, donc 1100 // 500 = résultat division entière = 2

**Essayez par vous-même :**

compte = 200

economies = 3000

# A FAIRE: modifiez la ligne suivante avec l'opérateur simplifié +=

# economies = economies + 500

economies += 500

print("Vos économies sont de: ", economies)

#### Essayez par vous-même :

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=f9f0d7ba-e456-4365-87ed-c96bd8b38f69>

**Écrivez des codes plus courts avec des opérateurs simplifiés**

Lorsque vous voulez changer une variable en modifiant la valeur initiale via un opérateur basique, vous pouvez utiliser une version plus courte. Autrement dit, vous pouvez utiliser **des opérateurs simplifiés**! Par exemple, plutôt que d’utiliser l’expression compteEpargne = compteEpargne + 100pour ajouter 100 € à votre variable, vous pouvez utiliser une sorte de double opérateur arithmétique/affectation += :

# version explicite

compteEpargne = compteEpargne + 100

# équivalent à version courte :

compteEpargne += 100

Il se décline naturellement pour chaque opérateur vu précédemment :

* -= pour la soustraction ;
* \*= pour la multiplication ;
* /= pour la division ;
* %= pour le reste de la division entière ;
* etc.

**Au-delà de l’arithmétique**

Dans l’exemple ci-dessus, toutes les variables permettent de stocker des montants d’argent, des numériques (entiers ou décimaux).

Existe-t-il d’autres types de variables que les numériques ?

Naturellement ! En fait, il est possible de stocker n’importe quel type de données dans une variable.

Jusque-là, vous avez vu qu’une variable est composée d’un nom et d’une valeur qui va définir automatiquement son type. Pourtant, pour revenir à votre analogie de l’entrepôt, on peut imaginer que stocker de l’argent et stocker un livre ou une voiture demande des boîtes, des conteneurs, même, de tailles différentes. En Python, l’espace de stockage d’une variable s’adapte automatiquement à son contenu, presque par magie ! :magicien:

À présent, on peut considérer qu’avoir uniquement le montant sur chaque compte de votre client ne suffira probablement pas. On va très probablement avoir également besoin de son nom, son activité bancaire exprimée en pourcentage, son adresse, etc., avec une nouvelle variable pour chacune de ces informations. On aurait donc besoin de stocker du texte dans nos variables.

Vous pourriez déclarer ces trois variables de la façon suivante :

prenom = "Benjamin"

pourcentageActivite = 72.3

adresse = "7 Cité Paradis, 75010 Paris"

Notez que vous avez besoin d’utiliser des doubles **quotes** (") pour définir des chaînes de caractères – variables textuelles – en Python, sinon vous risquez d’être confronté à une erreur ! Vous pouvez cependant utiliser des simples (') ou doubles **quotes** pour déclarer des variables textes comme vu ci-dessous :

adresse = "7 Cité Paradis, 75010 Paris"

# équivalent à

adresse = '7 Cité Paradis, 75010 Paris'

Vous allez rentrer plus en détail dans les différents types de variables en Python dans les prochains chapitres, mais vous avez pour le moment vu 3 types de variables différentes :

* les variables entières **(int)**;
* les variables décimales **(float)** ;
* les chaînes de caractères **(string)**.

**En résumé**

Dans ce chapitre, vous avez découvert les fondamentaux sur les variables :

* Une **variable** est composée de deux éléments : son **nom** et sa **valeur**.
* L’attribution d’une valeur à une variable s’appelle une **affectation**.
* La **valeur** d’une variable peut être **modifiée**.
* Le **type** d’une variable dépend de sa **valeur**.
* Les noms de vos variables doivent être **clairs**,**explicites** et doivent suivre une **convention typographique**.

Dans le prochain chapitre, vous aborderez plus en détail les types de variables !

**Manipulez les types de variables**

**Pourquoi avons-nous besoin des types de variables ?**

Vous avez exploré rapidement différents types de variables lors du précédent chapitre, mais il y a encore bien d’autres choses à savoir sur le sujet !

Les types rencontrés jusqu’à présent sont surnommés les**types primitifs**. Ils existent de base en Python – un peu comme les atomes. Ce sont les types de variables les plus simples ; ils représentent la fondation de toutes les opérations et tous les programmes informatiques. Vous pouvez néanmoins les combiner pour créer des types de variables bien plus complexes, comme vous pourrez le voir dans les prochaines parties de ce cours. Pour le moment, vous allez explorer un peu plus en profondeur les **types numériques** et les**chaînes de caractères**. Let’s go !

**Les variables numériques**

Les variables numériques peuvent être décomposées en deux types bien distincts :

* **les entiers**, qui correspondent à l’ensemble des nombres entiers positifs ou négatifs (1, 2, 0, 123, -3, etc.) ;
* **les nombres décimaux**, qui, en plus des entiers, incluent l’ensemble des nombres décimaux (2,50 ; 5,99 ; -1,20 ; 2/3 ; etc.).

Commencez avec celui avec lequel vous êtes déjà familier : **les entiers**. Les entiers sont déclarés comme n’importe quelle variable, en associant une valeur à un nom de variable.

compte = 10

Vous avez ici la valeur 10 associée à la variable compte. 10 étant un entier, compte est automatiquement de type entier (int).

Pour les chiffres décimaux, Python utilise le type float. Vous pouvez le définir de la même façon que les entiers, en ajoutant simplement explicitement la virgule :

longueur = 1876.79

largeur = 870.0

Tant que la valeur associée est un nombre décimal, Python va automatiquement considérer la variable comme un type float. Ceci est vrai, même si le chiffre après la virgule est un 0 – comme ci-dessus avec la variable largeur.

**Mixez plusieurs variables numériques**

Il est important de garder en mémoire la façon dont les **différents types numériques peuvent être mixés ensemble** et quelles sont les potentielles conséquences. Si vous mixez différents types, le plus *complexe* sera forcément celui conservé pour le résultat final. Par exemple, un int peut être float– comme vu ci-dessus avec la variable largeur, mais le contraire n’est pas forcément vrai ! Le float est donc le type le plus complexe : si vous mixez un int avec un float, le résultat sera toujours un float, peu importent l’opération effectuée et le résultat.

a = 7.5

b = 3

c = a/b

c

# cela va afficher 2.5, ce qui est bien un float

Écrire le nom d’une variable comme vous l’avez fait ci-dessus va l’afficher dans la console, mais ce n’est pas la façon la plus propre de le faire ! La meilleure façon est d’utiliser la fonction **print**, comme vous pourrez le voir dans le prochain exemple.

Si le résultat d’une opération entre deux entiers est censé être un nombre décimal, Python va automatiquement le convertir en **float**. De plus, la division (même si le résultat est censé être entier) renverra forcément un **float** également :

a = 10

b = 5

c = a/b

print(c)

# c est un float

Cependant, vous pouvez forcer la conversion d’une variable dans un type bien défini. Ceci est appelé du ***typecasting***, car en faisant ainsi, vous remodelez (***cast***en anglais) le type d’une variable. Pour ce faire, vous aurez besoin des fonctions correspondantes :

* int(): pour les entiers ;
* float(): pour les décimaux.

a = 14.0

# a est un float

a = int(a)

print(a)

# a est à présent un entier : il affiche 14 et non plus 14.0

**Les chaînes de caractères**

Vous allez à présent explorer un peu plus **les chaînes de caractères**, qui permettent de stocker du texte dans vos variables. Tout d’abord, un peu de sémantique : nous appelons cela des *chaînes de caractères*, car Python ne considère pas ces variables comme du texte en tant que tel, mais comme un ensemble de caractères mis bout à bout. Voilà comment vous pouvez définir des chaînes de caractères en Python :

ville = "Paris"

film = 'intouchables'

stringVide = ""

Les chaînes de caractères sont de type string en Python.

Assembler plusieurs strings ensemble est une des opérations les plus courantes que vous aurez à réaliser lorsque vous manipulerez des strings : on appelle cette opération une **concaténation**. Voyez comment faire cela en Python :

villeFavoriteUne = "Reims"

villeFavoriteDeux = "Strasbourg"

favoris = villeFavoriteUne + villeFavoriteDeux

print(favoris) # => "ReimsStrasbourg"

Notez qu’il n’y a pas d’espace entre les deux. Faites-en sorte de rendre votre code plus lisible... en concaténant vos variables avec d’autres chaînes de caractères :

villeFavoriteUne = "Reims"

villeFavoriteDeux = "Strasbourg"

favoris = "Mes villes favorites sont " + villeFavoriteUne + " et "+ villeFavoriteDeux

print(favoris) # -> "Mes villes favorites sont Reims et Strasbourg"

C’est bien mieux comme cela, n’est-ce pas ? Vous ne pouvez cependant pas concaténer d’autres types avec des **strings** – comme des variables numériques : cela renverrait une erreur. Pour remédier à cela, vous aurez besoin de *caster* votre variable numérique en **string**, via la fonction str():

ville = "Sydney"

nombreVoyages = 5

histoire = "J'ai déjà été à " + ville + " " + str(nombreVoyages) + " fois"

print(histoire) # => "J'ai déjà été à Syndey 5 fois"

Vous avez vu que l’opérateur + peut avoir différents buts en fonction des types de variables que vous manipulez :

* avec des **types numériques**, il sert à **additionner** ;
* avec des **chaînes de caractères**, il sert à **concaténer**.

**En résumé**

Dans ce chapitre, vous avez rencontré trois types primitifs de variables, indispensables à tous programmes/analyses :

* entiers **(int)** ;
* décimaux **(float)** ;
* chaînes de caractères **(string)**.

Vous avez également vu comment manipuler ces différents types :

* vous pouvez réaliser des **opérations numériques** sur des variables numériques de types différents ;
* vous pouvez **caster** des variables pour forcer la transformation du type d’une variable en un autre spécifique, en fonction de vos besoins ;
* des chaînes de caractères peuvent être assemblées ensemble : c’est la **concaténation**.

Dans le prochain chapitre, vous verrez comment écrire et utiliser des fonctions.

**Écrivez vos propres fonctions**

Depuis le début de ce cours, vous avez utilisé différentes fonctions, comme la fonction print() ou encore les différentes fonctions de **cast** comme int() ou encore str(). Vous allez à présent prendre le temps de bien définir ce qu’est une fonction, à quoi elle sert et comment vous pouvez créer vos propres fonctions : les fonctions n’auront plus aucun secret pour vous !

**Découvrez les fonctions**

Lors de vos analyses de données, vous allez régulièrement être amené à utiliser plusieurs fois des groupes d’instructions dans un but très précis. Un des principes fondamentaux pour tout programmeur informatique est d’**avoir un maximum de résultats pour un minimum d’efforts**: un dicton dit même *qu’un bon programmeur est un programmeur fainéant*. C’est grâce à ce principe un peu "paresseux", mais sacrément efficace qu’est venue l’idée des fonctions. Les fonctions**permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom**.

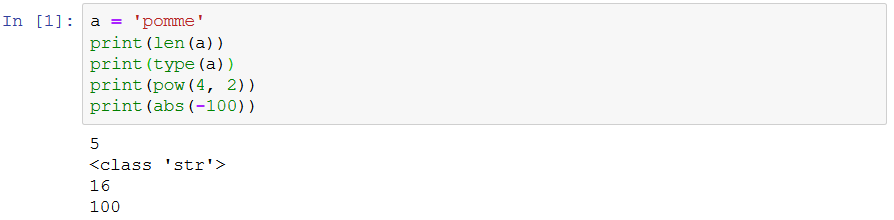
Les fonctions ne sont pas spécifiques à Python, elles sont présentes dans tous les langages informatiques. Elles permettent en particulier :

* de réutiliser une portion de code déjà écrite via le nom de la fonction – il n’est donc pas nécessaire de réécrire toute la portion de code à chaque fois ;
* d’alléger un code et de le rendre plus lisible !

Il existe de nombreuses fonctions préexistantes en Python ! En plus de celles déjà vues, il y a par exemple :

* len() : une fonction qui renvoie la longueur d’un élément. Vous vous souvenez des chaînes de caractères ? Utiliser cette fonction sur une chaîne de caractères permet par exemple de savoir combien de caractères cette dernière contient ;
* type() : permet d’afficher le type d’une variable ;
* pow(a, b) : permet de calculer a puissance b. Elle est équivalente à l’écriture a\*\*b;
* abs() : retourne la valeur absolue d’un nombre.

Voici quelques exemples pour illustrer l’utilisation de ces fonctions :



Fonctions de base en Python

**Écrivez vos propres fonctions**

Supposez à présent qu’il vous soit demandé de développer un programme faisant intervenir des notions de géométrie. Vous avez à utiliser de nombreux triangles dont vous avez la longueur des trois côtés et vous souhaitez afficher leur périmètre. Vous pourriez naturellement réaliser l’opération pour chaque triangle à la main, mais n’oubliez pas : vous voulez un maximum de résultat pour le moins d’efforts possible. Vous allez donc créer une fonction qui va afficher le périmètre d’un triangle en fonction de la longueur de ses côtés !

Une fonction est définie via le mot clé def suivi du nom de la fonction. L’ensemble des instructions associées à cette fonction seront ensuite écrites à la suite des deux points.

def nomFonction():

# instructions

# pouvant aller

# sur plusieurs

# lignes

Notez que l’ensemble des instructions associées à votre fonction sont décalées dans le code : c’est ce que l’on appelle l’***indentation***. Python est un langage structuré en blocs : chaque groupe ou *bloc de programme* doit donc être indenté. Par exemple, l’ensemble des instructions indentées à la suite des : seront uniquement disponibles dans nomFonction. L’indentation démarre le bloc et la "désindentation" le termine (la première ligne non indentée ne sera donc plus incluse dans nomFonction).

L’**indentation** est un principe fondamental en Python, que vous allez encore rencontrer de nombreuses fois à l’avenir. Généralement, il se traduit par 4 espaces ou une tabulation.

Définissez à présent une fonction qui répondrait à votre problématique :

def afficherPerimetre():

cote1 = 6

cote2 = 4

cote3 = 3

perimetre = cote1 + cote2 + cote3

print(perimetre)

afficherPerimetre() # => 13

Cette fonction est correcte, mais un peu inutile telle quelle : vos triangles n’ont pas tous la même dimension.

**Définissez des paramètres**

Pour aller au-delà de cette limitation, vous devez faire accepter à votre fonction des nombres extérieurs. Vous pouvez faire cela en définissant des **paramètres**.

En Python, les paramètres, tout comme le nom de la fonction, sont définis lors de l’écriture de cette dernière. Voilà ce que cela donnerait avec l’exemple ci-dessus :

def afficherPerimetre(cote1, cote2, cote3):

perimetre = cote1 + cote2 + cote3

print(perimetre)

Les paramètres sont des variables définies lors de la déclaration de la fonction, spécifiées à l’intérieur des parenthèses. À présent, vous pouvez utiliser votre fonction avec n’importe quel triangle existant :

afficherPerimetre(10, 11, 4) # => 25

afficherPerimetre(2, 2, 3.5) # => 7.5

Chaque valeur est affectée à un paramètre, dans l’ordre où elles ont été définies. Par exemple, dans le premier essai :

* la variable cote1 aura pour valeur 10 ;
* la variable cote2 aura pour valeur 11 ;
* la variable cote3 aura pour valeur 4.

La fonction va ensuite réaliser l’ensemble des opérations avec ces valeurs-là.

Les paramètres correspondent aux variables déclarées dans une fonction. Les valeurs qui sont passées en paramètres sont, elles, appelées des ***arguments***.

C’est top, vous avez ajouté quelques fonctionnalités à votre fonction !

À présent, qu’est-ce que je peux faire avec le résultat ?

Souvent, lorsque vous utilisez une fonction dans un code, vous attendez une réponse que vous pouvez réutiliser pour avancer dans le code. Cette réponse peut être apportée via la **valeur retournée** par une fonction.

**Définissez une valeur retournée**

Pour définir une valeur retournée, il faut explicitement utiliser le mot clé **return** à la fin de votre fonction.

Vous pourriez changer votre fonction afficherPerimetre en calculPerimetre qui retournera l’aire d’un triangle, en fonction de la longueur de ses trois côtés, pour pouvoir être réutilisé après coup :

def calculPerimetre(cote1, cote2, cote3):

perimetre = cote1 + cote2 + cote3

return perimetre

Une fois votre fonction définie, vous pouvez l’utiliser autant de fois que nécessaire :

perimetre1 = calculPerimetre(6, 4, 3)

perimetre2 = calculPerimetre(10, 3, 11)

print("Le périmetre de mon premier triangle est", perimetre1, "et celle de mon second est", perimetre2)

Et en analysant ces lignes, vous vous rendrez compte qu’à chaque fois que vous utilisiez la fonction print, vous lui passiez en paramètres les éléments à afficher.

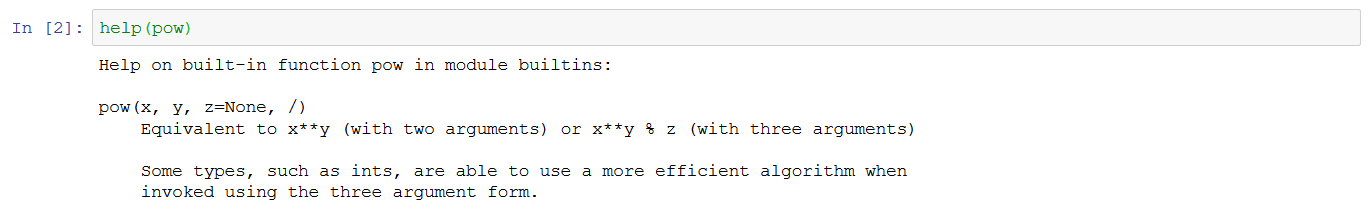
Notez que vous pouvez passer plusieurs arguments à la fonction print et le résultat sera la concaténation de tous les arguments. En faisant ainsi, vous n’avez pas besoin de **caster** les numériques en **string**.

**Utilisez la fonction help en cas d'oubli**

Dans votre expérience dans l’analyse de données, il va vous arriver régulièrement de vous rappeler du nom d’une fonction, mais pas forcément de ce qu’elle fait ou pas forcément de ces arguments, etc. Pas de panique ! La fonction help est là pour ça ! Si vous exécutez help(nomFonction), cela affichera la documentation de cette fonction résumant :

* son but ;
* des recommandations d’utilisation ;
* la liste et la description des paramètres ;
* parfois même des exemples.

Voici un exemple avec la fonction puissance (pow) vue ci-dessus :



Documentation d’aide de la fonction Pow

Ouf, vous avez assimilé pas mal de nouvelles connaissances en peu de temps. Quelle progression !

**En résumé**

Dans ce chapitre, vous avez vu que :

* les fonctions peuvent avoir des **paramètres** et des **valeurs de retour** ;
* une **valeur de retour** est le résultat de l’exécution de la fonction. Elle est retournée au code qui a appelé la fonction, pour ensuite être utilisée au besoin ;
* les **paramètres** sont les données nécessaires à une fonction pour être exécutée et générer un résultat ;
* les paramètres sont des variables définies par un **nom**. Ils sont **spécifiés dans la déclaration de la fonction** ;
* lorsque l’on utilise une fonction, vous lui passez différentes valeurs en paramètres. Ces valeurs sont appelées ***arguments*** ;
* vous pouvez utiliser la fonction **help** pour afficher la documentation d’une fonction donnée.

Dans le prochain chapitre, vous aborderez la notion de programmation orientée objet et ce que cela change pour vous en Python.

## Programmez-en orienté objet

Python est un langage de programmation orienté objet : cela signifie qu’en Python, tout est objet ! Vous allez voir dans ce chapitre ce que cela signifie et comment vous pouvez l’exploiter de façon concrète. Commencez par regarder quelques objets issus de la vie de tous les jours, comme des stylos, livres, smartphones, ordinateurs, etc.



Des objets bien différents !

Des objets peuvent avoir des formes et caractéristiques bien diverses, mais vous pouvez classer différentes versions d’un même objet dans une catégorie ou un groupe. C’est pourquoi vous n’avez aucun mal à reconnaître une chaise dans un magasin, par exemple, bien qu’elle puisse avoir une apparence (forme, couleur, etc.) très variable d’un modèle à un autre.

C’est en observant les points communs entre différents objets que vous êtes capable, mentalement, de classer les objets dans un même groupe ou une même catégorie !

Par exemple, il y a différents types de livres, mais ils ont tous un titre, un auteur, une quatrième de couverture, etc. Tous les livres partagent différents attributs qui vous permettent de les classer dans une catégorie bien identifiée : celle des livres.

**Les classes : des modèles d’objets**

En programmation, cette notion de groupe ou de catégorie d’objet est appelée **une *classe***. Une classe est en quelque sorte le schéma de construction d’un objet qui va définir les caractéristiques de tous les objets de ce type et leurs fonctionnalités. À partir de cette classe, vous allez pouvoir créer différents modèles d’un objet.

Prenons un exemple concret avec une classe **Voiture**. Le plan d’une voiture peut être défini par :

* ses caractéristiques, appelées ***attributs***: elle a forcément 4 roues, une couleur, une forme, une puissance moteur, etc. ;
* ses fonctionnalités, appelées ***méthodes***: elle peut rouler, freiner, etc.

Ainsi, à partir de ce plan, vous pouvez créer différents modèles de voiture :

* une familiale classique, de couleur verte, de puissance moyenne (110 ch) ;
* une voiture de sport, rouge, relativement puissante (180 ch) ;
* une petite voiture citadine bleue, peu puissante (90 ch) ;
* etc.

Et peu importe le modèle de la voiture, elles sont toutes capables de rouler ou freiner, mais pas avec la même performance !

En résumé, **une classe** correspond au plan d’un objet, définissant ses **attributs** et ses **méthodes**. À partir d’une même classe, on peut donc créer plusieurs objets d’un même type, mais aux attributs différents : on appelle cela **des instances de classe**.

**Focus sur les méthodes**

Comme nous l’avons dit précédemment, en Python, tout est objet. C’est-à-dire que, sans le savoir, depuis le début de ce cours, vous manipulez des objets ! Considérons les lignes de code suivantes pour illustrer cela :

var1 = 14

var2 = 1031

Vous avez ici déclaré deux variables nommées var1 et var2, contenant les valeurs 14 et 1.031. En réalité, vous avez créé deux instances de la classe **int**, deux objets ayant chacun **un seul attribut** : sa **valeur**. Il en est de même pour les **floats** ou les **strings** : chaque fois que vous créez une variable de l’un de ces types, vous créez en réalité des objets en Python ayant pour attribut la valeur que vous leur assignez.

Nous avons parlé jusque-là des attributs, il est temps à présent de voir à quoi correspondent les méthodes. Une méthode de classe est une fonction qui n’est disponible que pour les différentes instances de cette classe. Si par exemple on considère la classe *Voiture* présentée ci-dessus ayant une méthode *rouler()*, et une classe *Avion* ayant pour méthode *voler()*, vous conviendrez assez logiquement qu’un avion ne peut pas rouler et une voiture voler. Il en va de même pour nos différents objets !

L’utilisation d’une méthode est toujours réalisée via la notation nomVariable.méthode(). Par exemple, les **strings** ont une méthode appelée lower() qui va transformer l’ensemble du texte contenu dans un objet, en minuscules. Voilà comment l’utiliser :



La méthode de string : lower

De la même façon qu’avec les fonctions, les méthodes de classes peuvent prendre des paramètres.

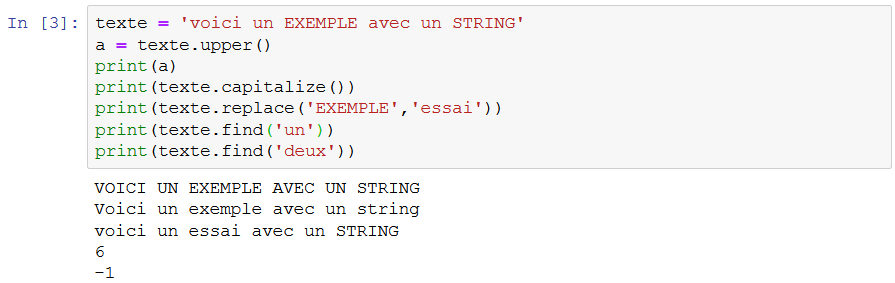
**Les méthodes de chaînes de caractères**

Lors des différentes analyses de données que vous serez amené à réaliser, vous allez forcément à un moment donné être confronté à des variables textuelles. Vous avez déjà vu comment changer votre string en minuscules, mais vous pouvez également avoir besoin de remplacer certains mots spécifiques, de formater le texte d’une certaine façon, etc.

Python a mis en place de nombreuses méthodes pour nous permettre de faire tout cela. Voici les plus courantes :

* upper() : retourne l’ensemble du texte en majuscules ;
* capitalize() : retourne l’ensemble du texte en minuscules avec la première lettre en majuscule ;
* replace(old, new) : cette méthode prend deux arguments. *old* correspond au(x) caractère(s) remplacé(s), *new* correspond au(x) caractère(s) remplaçant(s) ;
* find(chaîne) : retourne soit l’indice de la première occurrence de la chaîne passée en argument, soit -1 si elle ne la trouve pas.

Voici quelques exemples d’utilisation de ces méthodes :



Les méthodes de strings

Comme vous pouvez le noter ici, notamment avec les lignes qui concernent la variable a, les méthodes vues ci-dessus ne modifient pas l’objet initial ! Elles ne font que retourner le résultat de la méthode appliquée à l’objet. Vous aurez régulièrement à réaffecter ce résultat à la variable initiale, lorsque vous souhaiterez la modifier directement.

**Essayez par vous-même :**

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=47e75459-2980-4f16-8f5d-9e6a1188242b>

maPhrase = 'JE SUIS ENERVE !!'

# TODO : mettre maPhrase en miniscule

maPhrase\_lowercase = maPhrase.lower()

print(maPhrase\_lowercase)

# TODO : remplacez les ... par quelque chose permettant d'afficher : je suis calme !!

print('je suis calme !!')

### En résumé

* Une **classe** est un plan de construction d’un objet.
* Une variable est une **instance de classe**, ou **un objet**.
* Un objet est défini par ses **attributs**.
* Toutes les instances d’une classe ont accès aux mêmes **méthodes**via la notation..
* Une méthode, tout comme une fonction, ne modifie généralement pas l’objet initial.

Dans la prochaine partie, nous verrons un peu plus en détail comment organiser votre code via différentes structures et objets complexes.

# Manipulez les fonctions et objets fondamentaux

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Manipuler les fonctions et objets fondamentaux

### Question 1

**Quelles composantes définissent une variable en programmation Python ?**

* + 

Sa description et sa valeur

* + 

Son type et son but

* + 

Son nom et sa valeur

* + 

Son label et son contenu

*Les noms "officiels" pour les composantes sont : son nom et sa valeur. Le type, lui, est automatiquement défini par la valeur !*

### Question 2

**Quelles affirmations suivantes sont correctes concernant les noms des variables ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Le nom d'une variable peut commencer par un underscore.

* + 

Le nom d'une variable peut commencer par un chiffre.

* + 

Les mots clés ne peuvent être utilisés comme noms de variable.

* + 

Les noms de variable peuvent contenir des symboles comme : @, #, $, etc.

*Les noms de variables peuvent commencer par des underscores, mais pas par des chiffres. Dans tous les cas, ils ne peuvent pas contenir de symboles tels que : @, #, $, etc.*

*Les mots-clés étant réservés en Python, il est impossible de les utiliser pour quoi que ce soit d'autre que ce pour quoi ils ont été créés (incluant nommer une variable par l'un d'eux).*

### Question 3

**Parmi les éléments de la liste ci-dessous, lequel n'est pas un type primitif en Python ?**

* + 

int (integer)

* + 

str (string)

* + 

decimal

* + 

float

*int et float sont des types numériques. string correspond aux chaînes de caractères. Les nombres décimaux étant représentés par des float en Python, le type decimal n'existe pas.*

### Question 4

**Quelles valeurs ont les variables x et y une fois le code suivant exécuté ?**

x = 10

x += 12

y = int(x/4)

x = x + y

* + 

x = 27, y = 5.5

* + 

x = 27.5, y = 7

* + 

x = 27, y = 5

* + 

x = 27.5, y = 5.5

*Voilà le déroulé des opérations pas à pas :*

* + *x = 10*
  + *x = x + 12 = 10 + 12 = 22*
  + *y = int(x/4) = int(22/4) = int(5.5) = 5 : la transformation en entier supprime les chiffres après la virgule*
  + *x = 22 + 5 = 27*

*Donc x = 27 et y = 5*

### Question 5

**Parmi les options suivantes, laquelle permet de concaténer deux chaînes de caractères ?**

* + 

L'opérateur .

* + 

L'opérateur +

* + 

La fonction strcat()

* + 

L'opérateur ^

*Seul l'opérateur + permet de concaténer plusieurs strings ensemble. Par exemple, a = 'a' + 'concatenate' + 'string' stockera*'a concatenate string'*dans a.*

### Question 6

**Parmi les affirmations suivantes, laquelle est vraie ?**

* + 

Les fonctions sont utilisées pour créer des objets en Python.

* + 

Les fonctions permettent à notre programme de tourner plus vite.

* + 

Une fonction est un bloc de code qui est écrit pour réaliser une tâche spécifique.

*Le mot clé pour déclarer des objets est class, différent de celui pour créer une fonction (def). Une fonction ne fait pas nécessairement tourner notre programme plus rapidement, bien qu'elle le puisse dans certains cas !*

### Question 7

**Quels points suivants doivent être présents à la création d'une fonction ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Le nom de la fonction (suivi de parenthèses)

* + 

La liste des paramètres

* + 

La valeur à retourner

* + 

Le mot clé def

*Pour définir une fonction, vous devez absolument spécifier le mot clé de création de fonction (def), suivi du nom de la fonction. En revanche, il n'est pas nécessaire qu'elle ait des paramètres ou qu'elle retourne une valeur.*

### Question 8

**Lesquelles des affirmations suivantes sont vraies ?**

*Attention, plusieurs réponses sont possibles.*

* + 

Les méthodes sont des sortes de fonctions uniquement disponibles pour un objet donné.

* + 

Les méthodes sont des sortes de fonctions uniquement pour les chaînes de caractères.

* + 

Une méthode ne peut pas modifier l'objet original.

* + 

Vous pouvez utiliser une méthode via l'opérateur ..

*Les méthodes sont disponibles/possibles pour n'importe quel type/objet.  
Les méthodes peuvent retourner un objet ou une valeur, ou changer l'objet original.*

### Question 9

**Vous disposez de la variable a suivante :**

a = "This is A TEST"

**Quel code parmi ceux proposés permet d'avoir la sortie suivante ?**

**This is a test**

* + 

a.capitalize()

* + 

a.upper()

* + 

a.lower()

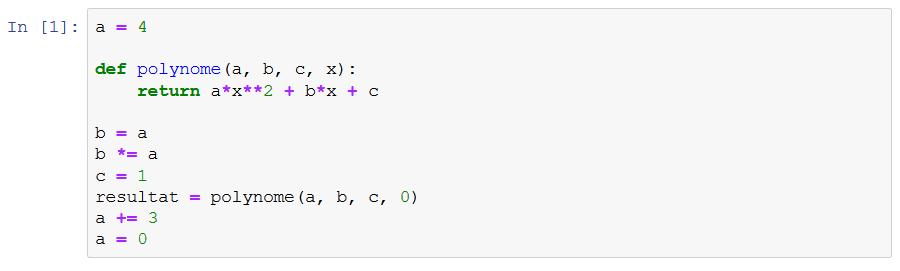
* + 

a.firstCapitalize()

*upper : retourne la chaîne en majuscules.  
lower : retourne la chaîne en minuscules.  
capitalize : tout en minuscules, sauf la première lettre.  
firstCapitalize n'existe pas !*

### Question 10

**Soit le code suivant :**

****

**Quelles sont les valeurs de résultat a, b et c ?**

* + 

resultat = 0, a = 0, b = 4, c = 1

* + 

resultat = 0, a = 7, b = 16, c = 1

* + 

resultat = 1, a = 7, b = 1, c = 1

* + 

resultat = 1, a = 0, b = 16, c = 1

*Déroulé des opérations :*

* + *a = 4*
  + *b = a = 4*
  + ***b****\*= a équivalent à b = b \* a = 4 \* 4****= 16***
  + ***c = 1***
  + ***resultat =****polynome(4, 16, 1, 0) = 4 \* (0\*0) + 16 \* 0 + 1 = 0 + 0 + 1****= 1***
  + *a + = 3 équivalent à a = a + 3 = 4 + 3 = 7*
  + ***a = 0***

*On a ainsi :*

*resultat = 1, a = 0, b = 16, c = 1*

**Créez des tableaux pour stocker vos variables**

Imaginez à nouveau que vous travailliez dans une banque. Ce que vous aviez vu en partie une était bien pour traiter chaque client individuellement, mais dans la réalité, il n’y a pas qu’un seul client à analyser...

Si jamais vous voulez analyser plusieurs clients en une fois, vous pourrez donc imaginer qu’il faille une variable pour chaque client. Cela pourrait donner, pour les noms :

nomClient1 = 'Georges Dupont'

nomClient2 = 'Luc Martin'

nomClient3 = 'Lucas Anderson'

nomClient4 = 'Alexandre Petit'

# etc...

Si jamais vous avez 10 clients à analyser, ne serait-il pas plus simple de pouvoir tous les stocker dans **une seule variable qui contiendrait toutes les informations** ?

Petit veinard ! Python offre une structure, une **classe** capable de pouvoir stocker plusieurs informations comme une sorte de tableau. Cette structure est appelée une ***liste***. Voyez un peu comment la manipuler.

**Déclarez une liste pour stocker vos éléments**

Les listes sont des collections. En fait, leur nom est plutôt explicite, puisque ce sont des objets capables de contenir d’autres objets de n’importe quel type. On peut avoir une liste contenant plusieurs nombres entiers (1, 2, 50, 2 000 ou plus, peu importe), une liste contenant des flottants, une liste contenant des chaînes de caractères... et une liste mélangeant ces objets de différents types.

Les listes sont des **objets ordonnés**, c’est-à-dire qu’à chaque élément de la liste est associé un nombre correspondant à son ordre dans la liste. Ce nombre est appelé ***indice*** et il démarre à 0 (et non à 1 !). Le premier élément est donc associé à l’indice 0, le second à l’indice 1, etc.

Déclarer une liste est assez similaire à la déclaration de n’importe quelle variable vue jusque là : via un **nom** auquel on associe **une liste d’éléments à stocker** dans ce nom.

Par exemple, voici la liste contenant les noms de 4 de vos clients :

nomClient = ['Georges Dupont', 'Luc Martin', 'Lucas Anderson', 'Alexandre Petit']

Maintenant que votre liste est créée, vous pouvez effectuer deux opérations basiques :

* **accéder** à une valeur à un indice donné ;
* **changer** la valeur à un indice donné.

Dans les deux cas, l’écriture se compose du nom de la variable suivi par [, la valeur de l’indice et ].

Si par exemple vous avez fait une erreur sur le nom du premier client et que vous voulez corriger son nom :

# assigner la valeur 'Georges Dupond' au premier nom dans notre liste

# c'est donc l'indice 0, car les indices commencent à 0 en python !

nomClient[0] = 'Georges Dupond'

Pour l’afficher, vous pouvez écrire la ligne suivante :

print(nomClient[0])

Python permet également d’utiliser des indices négatifs pour accéder à un élément ou le modifier. L’indice -1 correspond au dernier élément de la liste, -2 à l’avant-dernier, et ainsi de suite. Vous pouvez également accéder à un intervalle d’indices en utilisant l’opérateur :, 1:3 permettra par exemple d’accéder aux éléments 2 à 4.

# afficher le dernier élément

print(nomClient[-1])

# accéder du second élément au 4ème

print(nomClient[1:3])

# accéder à tous les éléments du début au 3ème

print(nomClient[:2])

Vous avez ici manipulé des listes de chaînes de caractères, mais vous pouvez faire la même chose avec le montant sur le compte de chaque individu :

montantCompte = [10000, 150, 300, 1800.74]

Vous l’aurez probablement noté, mais dans notre dernière liste, les trois premiers éléments sont des entiers, alors que le dernier est un décimal. Comme nous l’avons dit un peu plus haut, vous pouvez très bien stocker plusieurs objets différents dans une même liste.

Par exemple, la liste suivante est totalement valide :

listeEtrange = [4, 10.2, "Georges Dupond", ["une autre liste", 1]]

# afficher le 4ème élément de la liste

print(listeEtrange[3])

**Essayez par vous-même :**

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=cd75755d-1707-44da-a8c3-a657887f869f>

# TODO étape 1 - Déclarez une liste couleurs, contenant les 5 "couleurs" demandées

couleurs=['rouge', 'jaune', 'orange', 'vert' , 'bleu']

# TODO étape 2 - remplacez la couleur "vert", par "émeraude"

couleurs[3]='émeraude'

# affichez le contenu de la liste

for couleur in couleurs :

print(couleur)

Que faire à présent si un nouveau client vient s’ajouter à notre analyse ? Lorsque j’essaie de faire nomClient[4] = '...', cela me renvoie une erreur !

Pas de panique ! On ne peut en effet pas accéder à un indice qui n’existe pas déjà dans une liste en Python... en revanche, les listes disposent de nombreuses méthodes vous permettant d’y remédier.

**Ajoutez ou supprimez des éléments : les méthodes de listes**

Considérez à présent que voulez lister les animaux par ordre de *mignoncité* (du plus mignon vers le "moins" mignon). On démarre facilement avec une liste de quatre animaux : le renard, le koala, la chouette et la loutre. Maintenant, en surfant sur Internet, vous tombez inévitablement sur cette image :



Celui-ci mérite une place au sommet, non ?

Et vous souhaitez à présent ajouter le chat en première position !

Bonne nouvelle pour notre petit chat, les listes sont totalement modifiables, que ce soit le nombre d’éléments, l’ordre de ceux-ci, etc. Grâce aux différentes méthodes de listes, on peut :

* **chercher** un élément spécifique dans la liste ;
* **ajouter** un nouvel élément à la fin ;
* **insérer** un nouvel élément à un indice spécifique ;
* **supprimer** un élément de la liste.

**Ajoutez des éléments à une liste**

Vous pouvez très bien créer une liste vide en Python et ensuite ajouter les éléments un à un via la méthode **append :**

liste = []

liste.append(7)

liste.append(5)

print(liste) # => [7, 5]

1. La première instruction crée une liste vide nommée très originalement liste.
2. Vous ajoutez ensuite l’entier 7 à la fin de la liste. Python va donc l’ajouter à l’indice 0.
3. Finalement, vous ajoutez l’entier 5, qui sera stocké à la suite, à l’indice 1.

Vous remarquerez que vous n’avez pas eu besoin d’écrire liste = liste.append(...). Contrairement aux méthodes de strings vues précédemment, celle-ci modifie bien l’objet original.

Voici d’autres méthodes qu’il est indispensable de connaître autour des listes :

* insert : pour insérer un nouvel élément à une position spécifique. Par exemple, liste.insert(1, 12) insérera l’entier 12 à l’indice 1, déplaçant l’ancien élément 1 à l’indice 2 et ainsi de suite ;
* extend : similaire à append, mais avec une autre liste. D’une certaine façon, cela permet de concaténer plusieurs listes ensemble ;
* remove : cherche l’élément donné dans la liste et supprime la première occurrence rencontrée. Par exemple, si vous souhaitez supprimer 5 de votre liste, vous pouvez utiliser :liste.remove(5);
* index : cette méthode permet de trouver l’indice de la première occurrence d’un élément à chercher dans notre liste ;
* mot clé del : pour supprimer un élément selon son indice.

Voyez à présent à quoi tout cela ressemble :

liste = []

liste.append(7) # -> [7]

liste.append(5) # -> [7, 5]

liste.insert(1,12) # [7, 12, 5]

liste[0] = 4 # -> [4, 12, 5]

liste.remove(12) # [4, 5]

liste.index(5) # affiche 1

liste.extend([1, 2, 3]) # [4, 5, 1, 2, 3]

del liste[3] # [4, 5, 1, 3]

Décomposons ces quelques lignes :

* les trois premières lignes correspondent à ce qui a été vu avant ;
* vous ajoutez ensuite l’entier 12 à l’indice 1. L’ancienne valeur en position 1 est déplacée en position 2 ;
* vous remplacez ensuite la valeur à l’indice 0 par 4 ;
* avec la méthode .remove(), vous retirez l’entier 12 de notre liste ;
* vous demandez ensuite l’indice du premier élément 5 dans notre liste (ici en seconde position, donc retourne 1) ;
* vous ajoutez la liste [1, 2, 3] à la suite de notre liste initiale ;
* et vous supprimez finalement l’élément situé en position 4 dans notre liste.

Cela vous laisse finalement avec la liste finale : [4, 5, 1, 3].

**Gardez le contrôle de votre liste**

La fonction len() vous permet de récupérer la taille de votre liste :

liste = [1, 2, 3]

len(liste) # affichera 3

Cela permet de garder en tête les opérations qui ont été réalisées, et d’avoir une idée plus précise des indices auxquels d’éventuels nouveaux éléments seraient placés, sans avoir à afficher toute la liste !

La fonction len est extrêmement utilisée, notamment lorsque vous avez besoin de parcourir les différents éléments avec une **boucle**, comme vous le verrez dans le prochain chapitre !

**Essayez par vous-même :**

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=c327ece4-ccaf-4da0-b3ea-bab6bb7db086>

# TODO étape 1 - Remplacez les ?? par le code adéquat pour créer une liste vide

invites = []

# TODO étape 2 - Ajoutez 3 invités à la liste: Joey, Martin et Marie

invites.extend(['Joey', 'Martin', 'Marie'])

print(invites)

# TODO étape 3 – Affichez la taille de la liste

len(invites)

print('Nbre : ', len(invites))

# TODO étape 4 - Remplacez Martin par John dans la liste

invites[1]='John'

print(invites)

# TODO étape 5 - Supprimez Joey de la liste

del invites[0]

print(invites)

# affichez le contenu de la liste

for invite in invites :

print(invite)

**Utilisez les dictionnaires**

Revenez à présent à notre problématique avec les noms de nos clients bancaires et le compte en banque associé à chacun. Avec la méthode présentée ci-dessus, il vous faudrait deux listes. À chaque fois qu’une nouvelle personne est ajoutée à notre analyse, il faudrait ajouter son nom et son compte en banque dans les listes correspondantes.

Les dictionnaires sont un autre type d’objet, similaire aux listes, mais qui vont vous permettre de faire cela avec une seule variable ! En effet, un **dictionnaire** est une liste d’éléments organisés via **un système de clés**. Avec un vrai dictionnaire, vous regardez à un nom pour accéder à sa définition. En programmation, ce nom correspond à la clé et la définition à la valeur qui y est associée. C’est une association **clé-valeur**. On pourrait ainsi avoir :

|  |  |  |  |
| --- | --- | --- | --- |
| **Georges Dupont** | **Luc Martin** | **Lucas Anderson** | **Alexandre Petit** |
| 10000 | 150 | 300 | 1800.74 |

**Chaque clé** dans un dictionnaire doit être **unique**. On utilise généralement des chaînes de caractères pour définir les clés, mais ce n’est pas une obligation en soi !

**Déclarez un dictionnaire**

Les listes et dictionnaires sont déclarés de façon similaire, à la différence qu’un dictionnaire utilise des **accolades** au lieu des crochets, et qu’il faut déclarer les associations clé-valeur :

comptes = {"Georges Dupont": 10000, "Luc Martin": 150, "Lucas Anderson": 300, "Alexandre Petit": 1800.74}

print(comptes["Luc Martin"]) # -> 150

La dernière ligne va afficher la valeur associée à la clé "Luc Martin", qui est bien 150.

**Manipulez les éléments d’un dictionnaire**

Voici les opérations fréquemment réalisées avec des dictionnaires :

* **accéder** à la valeur d’un élément ;
* **ajouter** un nouvel élément (valeur-clé) ;
* **supprimer** un élément via sa clé.

On peut accéder à une valeur ou la modifier via la même notation qu’avec les listes. Cette notation vous permet même d’ajouter des éléments avec les dictionnaires, contrairement aux listes.

Voyez cela avec l’exemple suivant :

comptes['Georges Dupont'] -= 2000 # je soustrais 2000 au compte de Georges

comptes['Cyril Andreje'] = 1000 # j'ajoute un nouvel individu dans mon dictionnaire

print(comptes['Cyril Andreje']) # j'affiche la valeur du compte de Cyril

Finalement, vous pouvez supprimer un élément via la méthode pop(), en précisant la clé de l’élément que vous voulez supprimer.

comptes.pop('Luc Martin') # supprime Luc Martin de notre dictionnaire

Pour finir, de la même façon qu’avec les listes, vous pouvez utiliser la fonction len() pour garder le contrôle sur l’évolution de votre dictionnaire :

len(comptes) # -> 3

**En résumé**

Dans ce chapitre, vous avez appris l’ensemble des bases de l’utilisation de différents "types de stockage" :

* les **listes** : un tableau **ordonné**, où chaque élément est associé à un **indice** ;
* les dictionnaires : un tableau **non ordonné**, où chaque élément est associé à une **clé** ;
* les actions les plus souvent réalisées avec ces objets sont :
  + **accéder** à un élément,
  + **ajouter** un élément,
  + **supprimer** un élément,
  + **modifier** un élément,
  + **compter** le nombre d’éléments stockés ;
* on peut réaliser ces différentes actions via des **méthodes**.

Le type à utiliser dépend de la tâche à accomplir. Au fur et à mesure que vous progresserez dans votre carrière, vous serez en mesure de mieux identifier la structure la plus adaptée à votre situation !

Nous allons à présent voir comment organiser notre code via des structures conditionnelles.

**Contrôlez votre code grâce aux structures conditionnelles**

Au fur et à mesure que vous allez travailler sur des projets plus en plus complexes, écrire un ensemble de lignes qui vont s’exécuter une à une ne sera pas suffisant ! C’est là que les structures conditionnelles rentrent en jeu.

Dans nos tout premiers chapitres, vous avez vu comment dire "Hello World". Ne serait-il pas mieux de modifier légèrement ce programme pour qu’il soit un peu plus spécifique et dise bonjour à une personne en particulier ?

**Affichez des informations *si* elles sont disponibles**

Lorsque vous démarrez votre programme, vous ne connaissez pas forcément à l’avance le nom de l’utilisateur. Que diriez-vous d’un programme qui puisse :

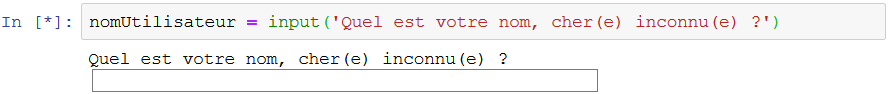
* dire bonjour à un utilisateur en particulier,**si** vous connaissez son nom ;
* **sinon**, continuer à dire bonjour à tout le monde ?

Voici votre première **condition**, qui vous permettra de construire votre première structure conditionnelle.

Comment pouvez-vous avoir le nom d’une personne en premier lieu ?

Vous souvenez-vous des fonctions ? Bonne déduction, vous pouvez faire cela via une fonction : la fonction input. Cette dernière va demander à l’utilisateur du notebook (autrement dit... vous !) d’entrer une chaîne de caractères qui sera ensuite stockée dans une variable.

Voyons ensemble un exemple :



Fonction input avec Jupyter !

Vous avez un espace pour pouvoir répondre à la question et votre réponse sera stockée sous forme de string dans la variable nomUtilisateur.

Concevez à présent le code qui vous permettra de dire bonjour à votre utilisateur :

1. Demander à l’utilisateur son nom et stocker sa réponse dans une variable nom.
2. Vérifier si la variablenom contient bien une valeur (au cas où l’utilisateur ne répondrait rien). La fonction len est votre amie pour réaliser cette tâche !
3. **Si** c’est le cas, dire *Hello* à votre utilisateur avec son nom.
4. **Sinon**, continuer à dire *Hello* au monde entier. :)

Voici le code correspondant :

nom = input('Quel est ton nom, cher(e) inconnu(e) ?')

if len(nom) > 0:

print("Hello", nom, "!")

else:

print("Hello World !")

Cela marche bien ! Tout ce qui est indenté en dessous du **if** est exécuté si **la condition est vraie**,**sinon** le programme exécute tout ce qui est indenté en dessous du **else**.

Comme vous pouvez le voir dans l’exemple ci-dessus, la construction en blocs utilisant l’opérateur : et l’**indentation** sont des notions indispensables et omniprésentes en Python ! Soyez bien vigilant sur l’organisation de votre code à ce niveau, pour éviter que certaines lignes s’exécutent alors qu’elles ne le doivent pas, et inversement.

Regardons un peu plus en détail comment la structure **if** fonctionne concrètement.

**Utilisez les booléens : le type sans demi-mesure**

En Python, pour valider une condition, vous utilisez un type (ou objet ;)) un peu spécial appelé ***booléen***. Une variable de type booléen ne peut contenir que deux valeurs : **vrai** ou **faux**. C’est finalement un type assez simple, mais ô combien utile !

En Python, le booléen peut concrètement prendre les valeurs **True** et **False**. Voyez à présent comment déclarer des booléens en Python :

ceCoursEstGenial = True

sonAuteurEstTresHumble = False

Facile, non ?

Pour revenir à votre structure conditionnelle **if**, je pense que vous l’aurez compris, mais il faut absolument que **ce qui suit le mot clé *if* aboutisse à un booléen**. Cela peut être réalisé via :

* une valeur True ou False. Par exemple, if True : ;
* une variable de type booléen. Par exemple, if maVariable: où maVariable est un booléen ;
* une **expression** qui aboutit à une valeur booléenne, comme dans l’exemple vu ci-dessus.

Par exemple :

meteo = "La météo est chouette !"

meteo.startswith("La météo") # -> True

startswith est une méthode de la classe **string**, qui retourne True lorsque la chaîne de caractères commence exactement par le string passé en paramètre ;False, sinon. Vous pourriez par exemple utiliser cette expression pour réaliser une action si une phrase commence bien par un mot en particulier.

Pour produire des booléens, vous pouvez également utiliser des **opérateurs de comparaison**.

**Opérateurs de comparaison**

Comme le nom le suggère, les opérateurs de comparaison sont utilisés pour comparer deux valeurs. Il y en a six principaux :

* == égal à (deux valeurs sont exactement pareilles) ;
* != différent de ;
* < inférieur à ;
* <= inférieur ou égal ;
* > supérieur à ;
* >= supérieur ou égal.

Voici quelques exemples avec des variables numériques :

2 == 2 # -> True

2 == 3 # -> False

4 != 4 # -> False

4!= 5 # -> True

1 < 2 # -> True

1 < 1 # -> False

1 <= 1 # -> True

3 > 4 # -> False

5 > 4 # -> True

5 >= 4 # -> True

On peut assigner le résultat de ces opérations à une variable :

age=15

if age>=21:

# Faire quelquechose si l'age est supérieur ou égal à 21

Parfois, vous allez avoir besoin de conditions plus élaborées, où la condition va être le résultat de la combinaison de plusieurs expressions. C’est là qu’interviennent les **opérateurs logiques**.

**Opérateurs logiques**

Ces opérateurs vont vous permettre de mixer plusieurs valeurs booléennes : des valeurs booléennes spécifiques ou des résultats d’expression. Il y en a 3 :

* and : l’opérateur **ET**.  
  Le résultat final est vrai seulement lorsque toutes les expressions/valeurs sont vraies. Par exemple : le résultat de expression1 and expression2 sera à True seulement si expression1 est vraie **ET** expression2 est également vraie ;
* or : l’opérateur **OU**.  
  Le résultat final est vrai lorsqu’au moins une des expressions/valeur est vraie. Par exemple : le résultat de expression1 or expression2 sera à True si expression1 est vraie **OU**expression2 est vraie ;
* not : l’opérateur **N’EST PAS**.  
  Cela inverse simplement le résultat de l’expression donnée. Par exemple, le résultat de not(expression) est vrai lorsque expression est faux.

Voici quelques exemples avec les résultats en commentaire :

True and True # True

True and False # False

False and False # False

True or False # True

True or True # True

False or False # False

not(True) # False

not(False) # True

On peut également mixer plus de deux expressions/valeurs :

True and True and True # True

True and True and False # False

True or False or False # True

False or False or False # False

Comme avec les opérations numériques, les opérateurs logiques respectent les priorités d’opérations : l’opérateur not est réalisé en premier, ensuite l’opérateur and puis l’opérateur or. Par exemple :

False or True and True # True

not(False) and True or False # True

Vous pouvez également utiliser des parenthèses pour changer l’ordre :

(True and False) or True # True

not(True and False or not(True)) # True

La forme générale d’une structure conditionnelle**if** est if condition: où la **condition** peut être soit un booléen, soit une variable de type booléen, soit le résultat d’une expression aboutissant à un résultat booléen.

**L’opérateur "in"**

Un autre opérateur logique utile en Python est l’opérateur in. Cela renvoie True lorsqu’une valeur est trouvée dans une séquence (un string ou une liste) ;False, sinon.

Par exemple :

maListe = [4, 2, 3, 2, 10]

maListeDeString = ["a", "b", "c", "d"]

monString = "La météo est vraiment bien aujourd'hui !"

4 in maListe # True

0 in maListe # False

0 in maListeDeString # False

"c" in maListeDeString # True

"e" in maListeDeString # False

"météo" in monString # True

"vraiment" in monString # True

"pluie ?" in monString # False

Dans votre exemple "Hello World", vous avez défini **une seule alternative**. Comment faire à présent si vous avez **plus d’une seule alternative** ?

**Gérez un enchaînement de conditions**

Pour accorder un prêt, une banque se base (entre autres) sur l’état des comptes de ses utilisateurs. Par exemple, une règle de décision naïve pourrait être :

* si le client a plus de 10 000 € sur son compte, on lui attribue son prêt d’office ;
* s’il a entre 100 € et 10 000 €, on fait une étude de son dossier ;
* sinon on lui refuse.

On pourrait utiliser deux **if** imbriqués, mais Python permet d’enchaîner plusieurs conditions grâce au mot clé elif (contraction d’*else* et *if*). Voici la forme générale :

if condition1:

# instructions

elif condition2:

# instructions

else:

# instructions

Voilà le code correspondant à l’exemple présenté ci-dessus :

compte = input("Quel est le soldes de votre compte ?")

compte = int(compte) # transformer la réponse en entier

if compte >= 10000:

print("Prêt accordé !")

elif compte >= 100 and compte < 10000:

print("Prêt en cours de validation : à l'étude")

else:

print("Prêt refusé")

**Essayez par vous-même :**

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=1e239e1d-5120-40a1-a857-a319ada883e8>

a = 5

b = 10

c = 7

if c < a :

print("c est plus petit que a")

if c > b :

print("c est plus grand que b")

if c > a and c < b :

print("c est entre a et b")

if c < a :

print("c est plus petit que a")

elif c > b :

print("c est plus grand que b")

else:

print("c est entre a et b")

**En résumé**

* Les conditions vous permettent d’exécuter un bloc de code lorsqu’un **booléen**, une **variable** ou une **expression** est vrai (True).
* Les expressions utilisent de l’**arithmétique booléenne**, incluant des **opérateurs logiques** et des **opérateurs de comparaison**.
* Vous pouvez réaliser plusieurs conditions avec les enchaînements de**if/elif/else**.

Dans le prochain chapitre, vous verrez un autre moyen de contrôler le code via des **boucles**.

**Choisissez la boucle adéquate**

Imaginez que vous avez un bloc de code que vous voulez répéter plusieurs fois. Vous pouvez naturellement le stocker dans une fonction et appeler cette fonction autant de fois que nécessaire. Cela fonctionnerait, mais reviendrait un peu à chasser une mouche avec un bazooka : ça marche, mais ce n’est pas forcément la solution la plus optimale ;). D’autant plus que, généralement, on ne sait pas forcément à l’avance le nombre de fois que l’on va avoir besoin de répéter ledit bloc.

Les boucles **permettent de résoudre cette problématique** ! En programmation, une boucle est une structure qui permet de répéter une ou plusieurs instructions, sans avoir à les réécrire à chaque fois. Il existe deux types de boucles (**for** et **while**), que vous allez à présent expliciter.

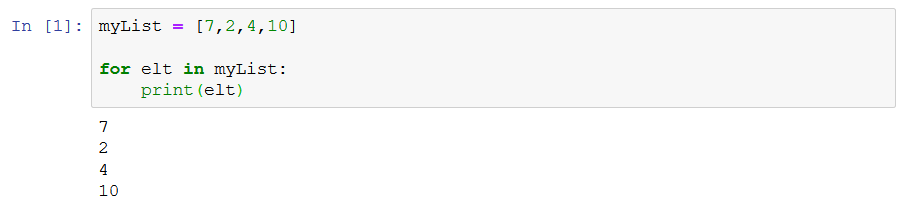
**“Bouclez” un nombre de fois fixé avec la boucle FOR**

Les boucles **for** vont être utilisées lorsque l’on sait par avance le nombre de fois où une action va être répétée.

Néanmoins, contrairement à la majorité des autres langages, en Python, une boucle **for** va forcément **itérer via une collection** (liste, dictionnaire, string, etc.).

**La boucle for sur une collection**

L’utilisation classique des boucles en Python se fait en utilisant directement les différentes valeurs d’une collection. Voici un exemple avec une liste :

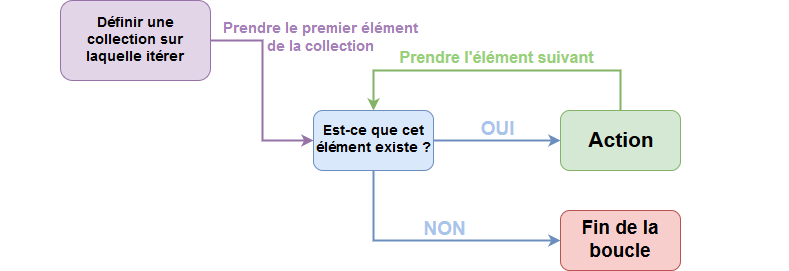


Parcours d’une liste via une boucle

Le résultat affiché correspond à chaque élément de la liste pris un à un. Détaillez un peu ce qui a été réalisé dans le code ci-dessus :

1. Vous avez créé une liste myList contenant 4 éléments : 7, 2, 4 et 10.
2. La boucle va stocker la première valeur de la liste (ici 7) dans la variable elt.
3. Ensuite, l’ensemble du bloc de code associé à la boucle **for** (défini via l’indentation... encore !) est exécuté avec elt ayant la première valeur. Ici, ce bloc consiste juste à afficher elt.
4. Une fois cela fait, elt va prendre la seconde valeur de la liste (ici 2) et l’on réexécute le bloc de code.
5. La boucle va continuer ainsi jusqu’à ce que l’ensemble des valeurs de votre liste aient été stockées dans la variable elt et que les instructions relatives à la boucle aient été exécutées pour chacune d’elles.

Voici un diagramme pour comprendre la logique de Python derrière la boucle :



Comprendre la logique de la boucle !

Vous pouvez également itérer via un **string** ! Rappelez-vous que l’on appelle les ***strings*** des "chaînes de caractères" en français, justement, car ils représentent une collection de caractères, de façon similaire à une liste.

monString = "Eléments"

for elt in monString:

print(elt)

Dans ce cas, elt va prendre successivement chaque caractère de votre string.

**La boucle for via une valeur itérative entière**

Si l’on veut vraiment boucler via une valeur itérative entière en Python pour revenir à une boucle **for** plus classique en termes de programmation, on va en réalité devoir contourner le problème. Vous allez itérer au travers d’une liste qui contiendra les différentes valeurs de votre entier itératif. Pour cela, vous utiliserez la fonction range(début, fin, pas), qui va générer une liste de nombres selon trois paramètres :

* début : le premier nombre de la séquence;
* fin : correspond au dernier nombre de la séquence non inclus. La fonction va générer des nombres de début à fin- 1;
* pas : le pas entre chaque nombre généré.

Tous les paramètres ne sont pas nécessaires. Par exemple :

for i in range(0, 5, 1):

print(i) # -> affiche de 0 à 4 par pas de 1 (fin - 1)

for i in range(0, 5):

print(i) # -> affiche de 0 à 4 également (le pas par défaut est 1)

for i in range(5):

print(i) # -> affiche de 0 à 4 également (le début par défaut est 0)

for i in range(0, 5, 2):

print(i) # -> affiche 0, 2 puis 4

La variable itérative peut prendre n’importe quel nom. Lorsque l’on itère sur un nombre entier, on utilise généralement des valeurs comme i, j ou k. Sinon, il est préférable d’utiliser des noms explicites, comme vu ci-dessus (elt étant l’abréviation d’élément).

La **boucle for** est parfaitement adaptée lorsque vous avez à réaliser une action un certain nombre de fois connu à l’avance ou une action pour chaque élément d’une collection. Pour l’ensemble des autres cas, on peut réaliser une**boucle conditionnelle** : une boucle qui n’itère pas via une collection, mais selon une condition.

**“Bouclez” selon une condition avec la boucle WHILE :**

La boucle conditionnelle est la boucle**while** en Python.

"While" en anglais signifie "tant que", et comme vous pouvez le deviner par le nom, la boucle va tourner **tant qu’une condition est réalisée**. C’est une sorte de combinaison entre une **boucle for** et une**structure if**. Le nombre de répétitions n’est pas défini à l’avance, mais via une condition à réaliser, comme avec un **if**. On appelle cela une *boucle conditionnelle*.

Voici comme se présente la syntaxe :

while expressionLogique:

# bloc à exécuter

Elle peut être interprétée comme : *tant que mon expression logique est vraie, exécute le bloc d’instruction.*

Voilà comment cela fonctionne :

1. Le programme vérifie que expressionLogique est égal à True.
2. Si c’est le cas, les instructions indentées à la suite des : sont exécutées. Une fois que cela est fait, on retourne à l’étape 1.
3. Sinon, le programme sort de la boucle sans exécuter les instructions.

Essayez avec l’exemple ci-dessous :

nombreArbres = 0

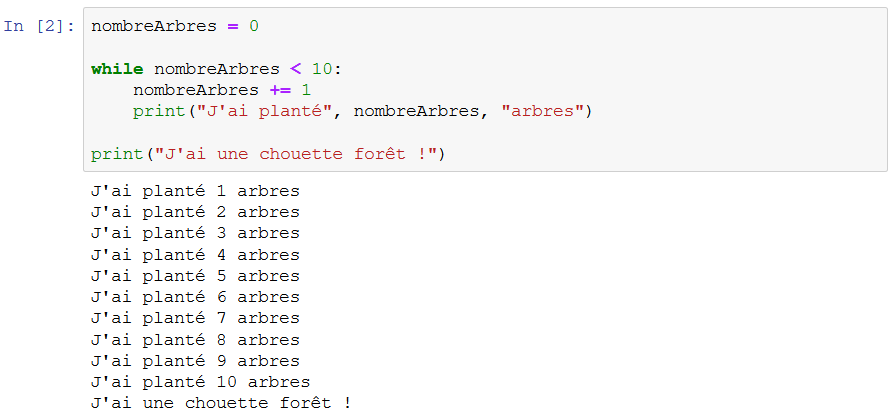
while nombreArbres < 10:

nombreArbres += 1

print("J'ai planté", nombreArbres, "arbres")

print("J'ai une chouette forêt !")

Cela va produire le résultat suivant :



Plantez une forêt !

À chaque tour de boucle, nombreArbres est incrémenté de 1. Lorsque la variable atteint la valeur 10, l’expression nombreArbre < 10 n’est plus vraie ! À ce moment, la boucle se termine et exécute le reste du programme à la suite. Dans ce cas précis, cela affiche : "J’ai une chouette forêt !"

❗️ Il est indispensable de bien garder en mémoire qu’une boucle **while** mal utilisée peut faire planter votre programme ! ☠️ Lorsque la condition reste toujours vraie, le programme reste bloqué dans la boucle. On est, dans ce que l’on appelle en programmation, une **boucle infinie**.

Voici un exemple à ne surtout pas reproduire chez vous (celui-ci a été réalisé par un professionnel...) :

leSoleilBrille = True

while leSoleilBrille:

print("Reste éveillé... pour toujours !")

# leSoleilBrille ne change jamais, donc la condition est toujours vraie

# nous n'atteignons donc jamais cette ligne

print("Il est temps d'aller dormir !")

C’est une erreur courante et malheureusement très vite arrivée. Donc **soyez vigilant** !

**Essayez par vous-même**

<https://api.next.tech/api/v1/publishable_key/72CE394E0D458BCE9188491ED894E88A?content_id=d740f82b-2866-4b1c-b7b3-46fbbe0892dc>

# TODO étape 1 - Créez une boucle for qui va afficher : 'OpenClassRooms est vraiment top !' 5 fois

print('-----------------------------------------')

print('-- BOUCLE FOR --')

print('-----------------------------------------')

for i in range(5):

print('OpenClassRooms est vraiment top !')

# TODO étape 2 - Changez votre boucle for par une boucle while. Est-ce une bonne idée ?

print('-----------------------------------------')

print('-- BOUCLE WHILE --')

print('-----------------------------------------')

i=1

while i<=5:

print('OpenClassRooms est vraiment top !')

i+=1

**Passez certaines instructions au sein de votre boucle**

Peu importe le type de boucle, il y aura des situations où vous allez vouloir passer certaines itérations au sein de votre boucle, voire interrompre cette dernière prématurément.

Par exemple, vous voulez répéter quelque chose 10 fois, mais passer (au moins partiellement) lorsque la valeur est égale à 2 ou 5. En Python, pour passer une itération dans une boucle, vous utilisez le mot clé continue :

for i in range(10):

# instructions réalisées à chaque itération

print(i)

if (i == 2) or (i == 5):

print("Cas spécial")

continue

# instructions non exécutées si i == 2 or 5

print("i != 2 & i != 5")

Vous pouvez aussi décider d’interrompre la boucle, lorsque par exemple on cherche un élément particulier dans une liste. Pour cela, vous utiliserez le mot clé break :

panier = ["pomme", "orange", "banane"]

for fruit in panier:

if fruit == "orange":

print("J'ai une", fruit, "!")

break

Une fois que le fruit a été trouvé dans votre panier, vous finissez la boucle.

**En résumé**

Dans ce chapitre, vous avez découvert 2 types de boucles :

* celle pour répéter une action un certain nombre de fois fixé, ou selon une séquence : la **boucle for** ;
* la boucle vous permettant de répéter une action tant qu’une condition est vraie : la **boucle while**.
* Il y a une erreur courante à ne pas commettre avec la boucle while : la **boucle infinie** !
* Vous pouvez choisir de passer certains tours de boucle via le mot clé **continue**.
* Les cycles de la boucle peuvent être interrompus via la commande **break**.

Maintenant que vous avez vu comment organiser votre code via différentes structures, vous allez à présent ouvrir votre horizon avec la découverte des modules et des librairies !

# Structurez votre code au sein d’un projet

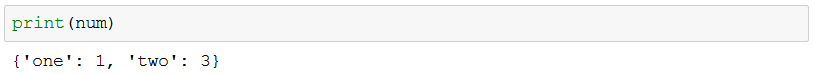
Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

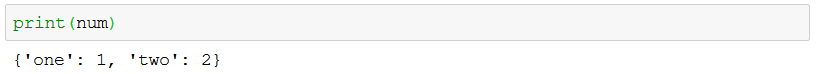
* Structurer votre code au sein d’un projet

### Question 1

**Comment pouvez-vous modifier la variable num suivante ?**

****

**En :**

****

* + 

num[2] = 'two'

* + 

num[1] = 'two'

* + 

num['two'] = 2

* + 

num['two'] = '2'

*Pour obtenir le second résultat, vous devez changer la valeur associée à la clé 'two' à 2. Le seul moyen de le faire est via l'écriture : num['two'] = 2.*

### Question 2

**Quelle affirmation parmi les 4 suivantes est vraie, vis-à-vis des listes en Python ?**

* + 

Tous les éléments d'une liste doivent être du même type.

* + 

Une liste peut contenir tous types d'objets à l'exception d'une autre liste.

* + 

Ces deux listes sont équivalentes : ['a', 'b', 'c'] et ['c', 'a', 'b'].

* + 

Une liste n'a pas de limite conceptuelle !

### Question 3

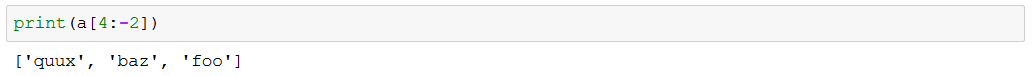
**Soit la liste suivante :**

a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

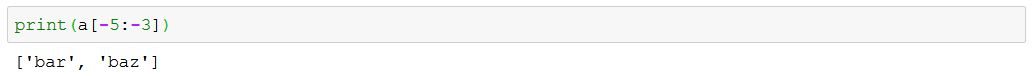
**Quels sont les affichages corrects parmi les 4 suivants ?**

*Attention, plusieurs réponses sont possibles.*

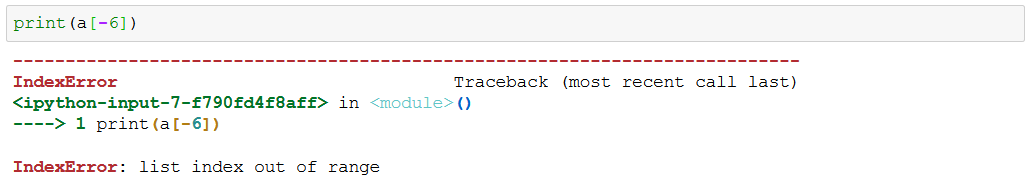
* + 



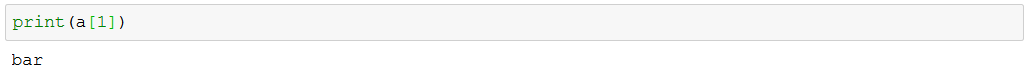
* + 



* + 



* + 



*Pour aller d'un indice positif à un indice négatif, vous devez utiliser 2 :, l'écriture correcte serait donc : print(a[4::-2]).*

*Lorsque vous accédez à un élément d'une liste via un indice négatif, l'indice du dernier élément est -1. print(a[-6]) n'est donc pas une erreur, elle affiche foo.*

### Question 4

**Les codes suivants concernent différentes utilisations des listes. Sélectionnez les blocs corrects parmi ceux ci-dessous :**

*Attention, plusieurs réponses sont possibles.*

* + 
  + colors = []
  + colors.append('Red')
  + colors.append('Green')
  + colors.append('Yellow')
  + print(colors)
  + 
  + colors = []
  + colors.insert('Red')
  + colors.insert('Green')
  + colors.insert('Yellow')
  + print(colors)
  + 
  + frequency = []
  + frequency[0] = 'Monday'
  + frequency[1] = 'friday'
  + 
  + frequency = ['sunday', 'monday', 'tuesday','wednesday']
  + frequency.remove('monday')

*Pour ajouter un élément à une liste, nous utilisons la méthode .append. .insert permet juste d'insérer un élément dans un indice fixé (mais vous devez spécifier l'indice... par exemple : colors.insert(0, 'Red') serait correct !). De plus, l'écriture list[indice] = ... sert uniquement à modifier un élément existant.*

*Dans tous les cas, .remove supprime l'élément sélectionné de la liste.*

### Question 5

**Comment délimiter la fin d'un bloc d'instructions en Python ?**

* + 

}

* + 

Un commentaire

* + 

Une ligne qui est moins indentée que la précédente

* + 

Le mot clé end

*En Python, seule l'indentation permet de délimiter les différents blocs de code. Ainsi, la fin d'un bloc de code est délimitée par une ligne moins indentée que la précédente.*

### Question 6

**Quelle est la sortie du code suivant ?**

if 'bar' in {'foo': 1, 'bar': 2, 'baz': 3}:

print(1)

print(2)

if 'a' in 'qux':

print(3)

print(4)

* + 

1  
2  
4

* + 

4

* + 

1  
2  
3  
4

* + 

Cela ne va générer aucune sortie !

*'bar' se trouve bien dans le dictionnaire placé après l'opérateur in, donc on rentre dans la structure conditionnelle :*

* + *affichage de 1 ;*
  + *affichage 2 ;*
  + *'a' n'est pas dans la chaîne de caractères 'qux', donc il n'affiche pas 3.*

*4 sera forcément affiché, car en dehors de la boucle.*

*L'affichage final est donc :*

*1  
2  
4*

### Question 7

**Que va-t-il se passer avec le code suivant ?**

d = {'a': 0, 'b': 1, 'c': 0}

if d['a'] > 0:

print('yeah !')

elif d['b'] > 0:

print('yeah !')

elif d['c'] > 0:

print('ok')

elif d['d'] > 0:

print('ok')

else:

print('not ok')

* + 

Il va afficher ok.

* + 

Il va afficher yeah !.

* + 

Il va afficher not ok.

* + 

Le code va lever une erreur de type KeyError à cause de l'accès à une clé non existante !

*Il y a bien une potentielle erreur dans le code, car la clé 'd' n'existe pas. Néanmoins, comme le programme suit les conditions dans l'ordre :*

* + *Il teste si la valeur associée à la clé 'a' est supérieure à 0 : ce n'est pas le cas, donc il passe à l'itération suivante.*
  + *Il teste si la valeur associée à la clé 'b' est supérieure à 0 : c'est le cas, donc il réalise les instructions correspondantes (afficher 'yeah !') et sort complètement de la structure conditionnelle.*

*Le programme n'atteint donc jamais la ligne où se trouve l'erreur.*

### Question 8

**La boucle while a été conceptualisée initialement lorsque :**

* + 

On connaît le nombre d'itérations.

* + 

À chaque fois que l'on souhaite répéter un bloc de code plusieurs fois.

* + 

On ne connaît pas forcément le nombre d'itérations.

* + 

On souhaite réaliser une boucle infinie.

*La boucle while peut en effet être utilisée à chaque fois que l'on souhaite répéter un bloc de code plusieurs fois, incluant les cas où l'on connaît le nombre d'itérations. Néanmoins, elle a été initialement créée pour les cas où l'on ne connaît pas le nombre d'itérations !*

### Question 9

**Qu'affiche le code suivant ?**

a = ['foo', 'bar', 'baz', 'qux', 'corge']

while a:

if len(a) < 3:

break

print(a.pop())

print('Done.')

* + 

corge  
qux  
baz  
Done.

* + 

corge  
qux  
baz

* + 

corge  
qux  
baz  
bar  
foo  
Done.

* + 

Le code ne génère aucun affichage !

*Cette écriture est un peu particulière :*

* + *le while a: va exécuter la boucle tant que a n'est pas vide ;*
  + *ensuite, on ajoute une condition d’arrêt de la boucle dès que la longueur de la liste a est inférieure à 3 ;*
  + *dans tous les cas, à chaque tour de boucle, on supprime le dernier élément de la liste et on l'affiche.*

*Ainsi, si on matérialise la boucle :*

* + *len(a) = 5, on supprime et affiche corge*
  + *len(a) = 4, on supprime et affiche qux*
  + *len(a) = 3, on supprime et affiche baz*
  + *len(a) = 2, on sort de la boucle, et on affiche Done.*

### Question 10

**Comment peut-on interrompre une boucle prématurément ?**

* + 

Via le mot clé continue

* + 

Via l'opérateur :

* + 

Via le mot clé stop

* + 

Via le mot clé break

*L'opérateur : permet de définir le début d'un bloc de code.*

*Le mot clé continue permet juste de sauter un tour de boucle.*

*stop n'est en revanche pas un mot clé Python !*

**Prenez en main les modules et librairies Python**

Supposez à présent que vous ayez besoin de calculer la racine carrée d’un nombre dans le cadre d’un de vos notebooks. Il n’existe pas de fonction racine carrée native en Python. Vous pourriez naturellement l’écrire vous-même, mais hey, il y a sûrement eu un tas de personnes qui se sont déjà posé la même question. Et devinez quoi ? L’un d’eux a déjà écrit la fonction et l’a enregistrée dans un module !

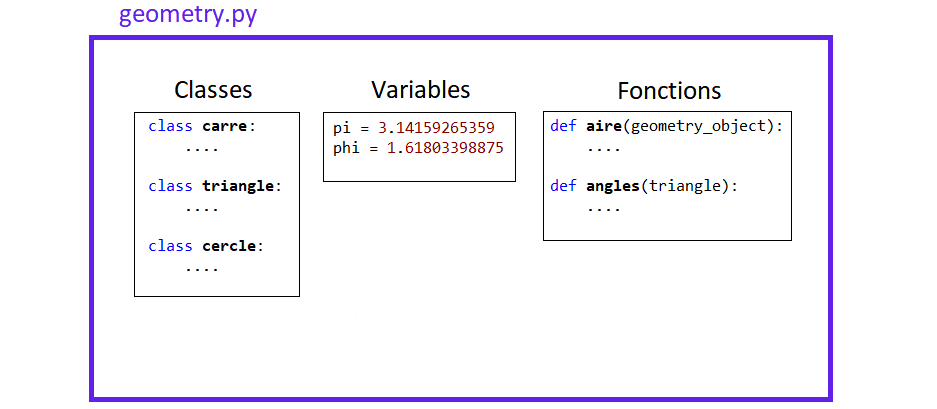
**Un module en Python**

Un **module** est un fichier Python contenant un ensemble de **fonctions**, de **classes** et de **variables** prédéfinies et fonctionnelles, que vous pouvez utiliser comme bon vous semble dans votre code !

Par exemple, si vous travaillez sur une problématique faisant intervenir de la géométrie, vous pourriez avoir besoin de :

* **classes :**
  + carré – défini par la longueur de son côté,
  + triangle – défini par la longueur de ses trois côtés,
  + cercle – défini par son rayon,
  + etc. ;
* **variables :**
  + pi : constante indispensable pour calculer l’aire d’un cercle, égale à 3,1415...,
  + phi : constante représentant le nombre d’or, égale à 1,6180... ;
* **fonctions :**
  + aire : qui prend en paramètre un objet géométrique (carré, triangle, etc.) et calcule son aire,
  + angles : qui prend en paramètre un triangle, et calcule les angles internes de ce dernier,
  + etc.

Vous pouvez naturellement définir toutes ces choses dans votre notebook, mais cela ne ferait que l’alourdir. Le mieux est de stocker tout cela dans un fichier Python externe, que vous allez ensuite importer dans votre notebook : c’est un module !



Votre module geometry

Voici un exemple simplifié d'un module geometry :

'''

Module geometry.py

'''

# variables

pi = 3.14159265359

phi = 1.6180

# fonction qui calcule l'aire

def aire(obj):

if type(obj) == carre:

return obj.a\*\*2

# definitions de quelques classes

class carre(object):

def \_\_init\_\_(self,a):

self.a = a

class triangle(object):

def \_\_init\_\_(self,a,b,c):

self.a = a

self.b = b

self.c = c

Pour pouvoir **importer un module**, vous allez avoir besoin du mot clé import. Voici un exemple avec notre module *geometry* :

import geometry

Après avoir fait cela, vous pouvez utiliser les différents éléments définis dans votre module :

car = geometry.carre(4)

tri = geometry.triangle(3, 6, 5)

print(geometry.pi) # -> 3.14159265359

geometry.aire(car) # -> 16

Tous les éléments inclus dans le module *geometry* peuvent être utilisés via la notation nomModule.fonction() ou encore nomModule.variable. Si vous ne souhaitez pas réécrire *geometry* à chaque fois, vous avez deux autres options :

* soit donner un **alias** au nom de votre module, pour n’avoir à écrire que l’alias :

import geometry as geo # on peut maintenant accéder à geo.aire() ou geo.pi

* soit importer des fonctions spécifiques que vous pourrez ensuite utiliser comme des fonctions/variables Python natives (sans la notation .) :

from geometry import pi

print(pi) # -> 3.14159265359

Un cas particulier de cette dernière méthode est d’importer en une ligne tous les objets contenus dans un module via la notation \*. Néanmoins, ce n’est pas la méthode préconisée, afin d’éviter par exemple les conflits entre plusieurs modules qui pourraient avoir un nom de fonction identique.

from geometry import \*

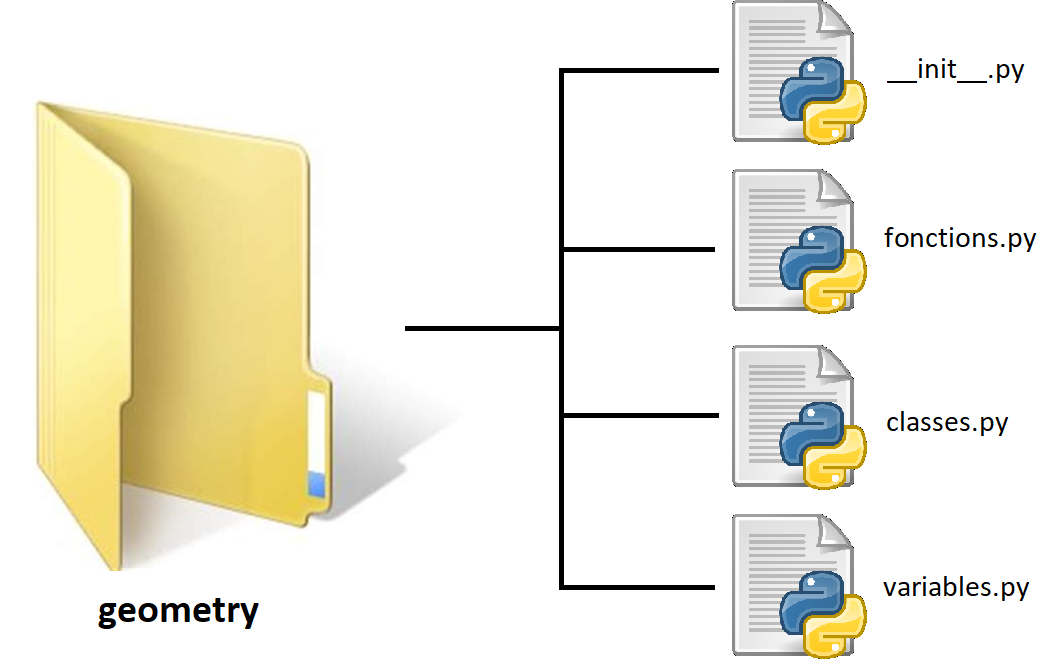
**Quand un module ne suffit pas : les packages**

Un **package** (appelé parfois ***librairie***) est une collection, un **ensemble de modules Python**. Comme vous l’avez vu ci-dessus, un module est un fichier Python. Un package est simplement un dossier contenant plusieurs fichiers Python (.py) et un fichier additionnel nommé \_\_init\_\_.py. Ce dernier différencie un package d’un dossier lambda contenant uniquement des codes Python.

Par exemple, vous auriez pu stocker votre module *geometry* dans trois fichiers différents au lieu d’un seul :

* un pour les classes : classes.py ;
* un pour les variables : variables.py ;
* un pour les fonctions : fonctions.py.

On aurait dans ce cas le dossier suivant :



Organisation du package geometry

Vous aurez besoin d’utiliser l’opérateur . pour accéder au module, après avoir importé le package :

import geometry # import all the geometry package

print(geometry.variables.pi) # -> 3.1415...

car = geometry.classes.carre(4)

geometry.fonctions.aire(car) # -> 16

Ou vous pouvez également importer seulement un module du package :

import geometry.variables as var # importer uniquement ce qui est défini dans variables.py

print(var.pi) # -> 3.1415...

**Les packages dans l’analyse de données**

Les packages sont omniprésents dans l’analyse de données avec Python. En effet, de nombreux packages ont été créés spécifiquement pour répondre aux problématiques du domaine. Au fur et à mesure de votre parcours, vous allez être amené à :

* **manipuler vos données** pour en faciliter l’analyse ;
* réaliser**différents graphiques** pertinents représentant le comportement de vos données ;
* utiliser des **méthodes statistiques** ;
* faire tourner des**algorithmes de machine learning** plus ou moins compliqués ;
* etc.

Et pour réaliser tout cela, il vous sera indispensable de maîtriser les différents objets et fonctions issus des packages correspondants.

Si vous souhaitez en savoir plus sur les packages permettant de faire ces différentes tâches, je vous redirige vers [**ce cours**](https://openclassrooms.com/fr/courses/4452741-decouvrez-les-librairies-python-pour-la-data-science) qui présente plus en détail les packages les plus utilisés dans l’analyse de données.

Pour revenir à votre problématique initiale (avoir une fonction racine carrée), il y a par exemple le package **numpy** qui en propose une – et bien d’autres choses !

import numpy as np

np.sqrt(16) # -> 4.0

Je vous propose de voir un exemple concret d’utilisation des packages math et random dans le prochain chapitre.

**En résumé**

Dans ce chapitre, nous avons vu ensemble les bases de l’utilisation des modules et des packages :

* un module est un **fichier** contenant du code Python (extension .py) qui peut définir des **fonctions**, des **classes** et/ou des **variables** ;
* vous pouvez **importer** n’importe quel module Python via le mot clé import;
* pour utiliser une fonction d’un module, ou une classe ou une variable, il faut utiliser l’opérateur .;
* un **package** est un ensemble de plusieurs modules Python ;
* il existe de nombreux packages spécifiquement créés pour l’analyse de données.

**Manipulez des nombres aléatoires avec le module random**

Il existe de nombreux phénomènes dus au hasard dans la réalité, et il peut être utile, parfois numériquement, de pouvoir modéliser ces phénomènes pour balayer les différents cas/scénarios possibles. En Python, le module **random** contient plusieurs fonctions pour pouvoir générer des nombres ou des suites de nombres aléatoires.

Si le sujet vous intéresse, les différentes fonctions du module random utilisent un générateur de nombres pseudo-aléatoires très performant et populaire, appelé [**Mersenne Twister**](https://fr.wikipedia.org/wiki/Mersenne_Twister).

**Générez des nombres aléatoires**

Premièrement, importez votre module random. Le nom du package en Python est... **random :**

import random

La fonction de base de génération de nombre aléatoire s’appelle...**random()** également (quelle originalité ;)). Elle va générer un **float aléatoire compris entre 0 et 1 non inclus**. Réalisez un exemple simple en affichant 3 nombres aléatoires :

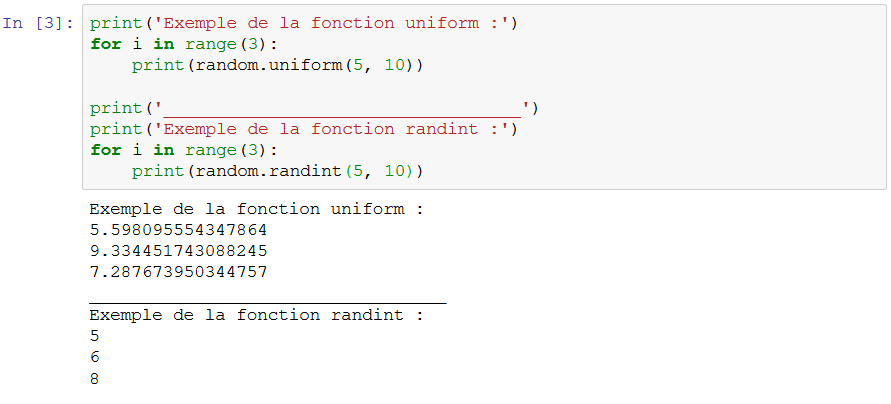


Naturellement, si vous réalisez le même code chez vous, vous obtiendrez des résultats différents !

Mais avoir un nombre entre 0 et 1 uniquement, c’est un peu limité... non ?

Tout à fait ! Mais ceux qui ont créé le package random ont heureusement pensé à tout. Vous avez d’autres fonctions vous permettant de générer un nombre aléatoire dans un intervalle donné :

* uniform(a, b) : va générer un float aléatoire compris entre a et b;
* randint(a, b) : comme son nom le suggère, celle-ci est similaire à uniform, sauf que le nombre aléatoire généré est cette fois-ci un entier !

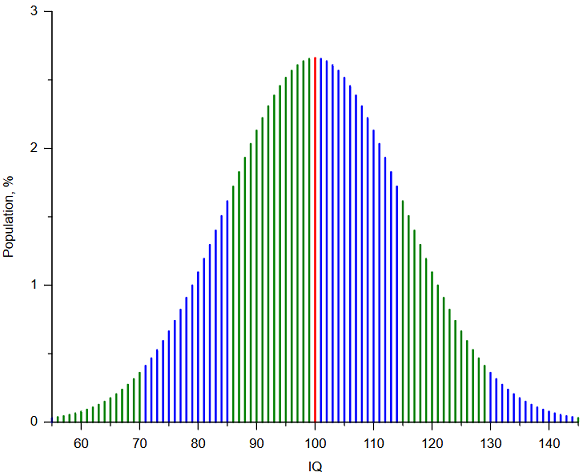


Vous pourrez utiliser l’une ou l’autre en fonction de vos besoins !

**Générez un nombre aléatoire selon une distribution donnée**

Le module random permet également de générer un nombre aléatoire suivant une distribution. L’une des plus connues est la loi gaussienne (ou *normale*). Si vous ne la connaissez pas encore, laissez-moi vous la présenter !

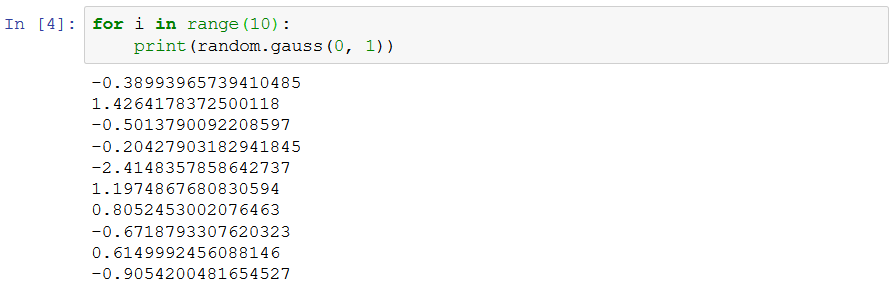
La **loi normale** est l’une des lois de probabilité les plus adaptées pour modéliser des phénomènes naturels issus de plusieurs événements aléatoires. Ce sont l’ensemble de ces phénomènes où la majeure partie des individus se situent autour d’une moyenne, avec des proportions décroissantes en dessous et au-dessus de cette moyenne. Voici un exemple très parlant, avec la répartition de la population par quotient intellectuel :



Proportion de la population par QI - Alessio Damato, Mikhail Ryazanov

Le module random vous permet de générer des nombres aléatoires selon cette loi : c’est-à-dire que l’on a beaucoup plus de chances d’avoir des valeurs proches de la moyenne (avec l’exemple ci-dessus, entre 85 et 115) que des valeurs extrêmes (proches de 70 ou 130). La fonction correspondante s’appelle gauss(moyenne, écart-type).

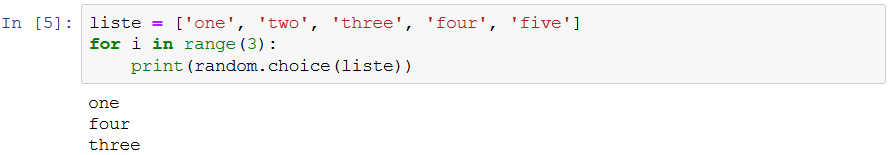
Voici un exemple avec une loi centrée en 0 et avec un écart-type de 1 (loi normale "classique") :



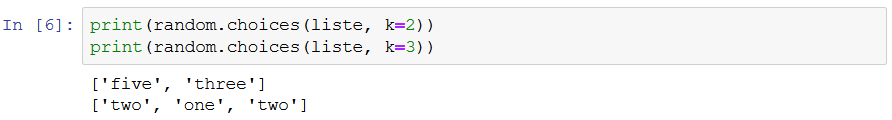
On visualise bien ici, avec 10 valeurs, que la majorité des valeurs sont proches de 0.

**Choisissez aléatoirement dans une liste : sous-échantillonnage**

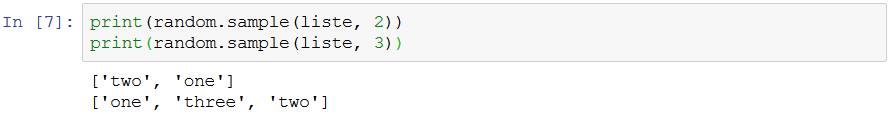
Comme vous le savez déjà, pour sélectionner un élément dans une liste, il faut le faire via son indice. Si vous voulez sélectionner un élément aléatoirement dans une liste, une solution un peu naïve pourrait être de tirer l’indice aléatoirement. Le module **random** va un peu plus loin en proposant une fonction permettant de faire la sélection directement sur la liste : la fonction choice.



L’évolution de celle-ci est la fonction choices, permettant cette fois-ci de sélectionner un échantillon de la liste initiale, **avec remise :**



On parle alors de ***sous-échantillonnage***. La fonction correspondante, pour un échantillon **sans remise**, est sample :



En analyse de données, cette notion de sous-échantillonnage est primordiale, elle permet de sélectionner un échantillon d’une population initiale. En statistique, un échantillon est un ensemble d’individus représentatifs d’une population. Le recours à un sous-échantillonnage répond en général à une contrainte pratique (manque de temps, de place, coût financier…) ne permettant pas l’étude exhaustive de toute la population.

**Pour aller plus loin**

Le module random propose plus de fonctions que celles présentées dans ce cours, même si vous avez néanmoins parcouru les plus utilisées en pratique. Si vous souhaitez aller plus loin, vous pouvez consulter [la documentation officielle du module random](https://docs.python.org/3/library/random.html), qui liste toutes les possibilités offertes par ce dernier.

Sachez également que le package **numpy**, que nous avons rapidement évoqué lors du chapitre précédent, reprend également le module random. Vous y retrouverez l’ensemble des fonctions vues ci-dessus. L’ensemble des fonctions sont accessibles via la ligne (par exemple) :

import numpy.random as random

**En résumé**

Vous avez vu les principales fonctionnalités disponibles via le module random. Vous pouvez à présent :

* générer un nombre aléatoire, **entier** ou **décimal**, **dans un intervalle donné** ;
* générer un nombre aléatoire**suivant une distribution donnée** ;
* choisir aléatoirement **un ou plusieurs éléments dans une liste**, avec ou sans remise.

# Utilisez des librairies Python spécialisées

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Utiliser des librairies Python spécialisées

### Question 1

**Cochez les nombres que l'on peut obtenir si l'on exécute le code suivant :**

import random

n = random.uniform(10,19)

print(n)

*Attention, plusieurs réponses sont possibles.*

* + 

6

* + 

10.0

* + 

14.4

* + 

23.0

* + 

10.5

* + 

19.0

*Ici, la fonction uniform va générer des nombres aléatoires décimaux, entre 10 et 19, ceux-ci inclus. Toutes les réponses sont donc correctes, sauf 6 et 23.*

### Question 2

**Par quoi remplacer les ... pour que le programme précédent affiche un des prénoms de la liste au hasard ?**

import random

panier = ["Pomme", "Poire", "Banane", "Ananas", "Orange"]

resultat = ...

print(resultat)

*Attention, plusieurs réponses sont possibles.*

* + 

panier[random.randint(0, 4)]

* + 

random.choice(panier)

* + 

random.random(panier)

* + 

random.shuffle(panier)

* + *random.randint(0, 4) va sélectionner un entier pris aléatoirement entre 0 et 4. Ce nombre pris aléatoirement va ensuite nous servir d'indice pour sélectionner au sein du panier.*
  + *La fonction choice permet par défaut de sélectionner un élément aléatoirement dans une liste.*
  + *Ceci renverra une erreur. La fonction random ne peut pas prendre une liste en paramètre.*
  + *Cette fonction permet de mélanger une liste.*

### Question 3

**Considérons l'expérience suivante :**

**Soit 10.000 lancers d'un dé équilibré classique (à 6 faces). Parmi ces 10.000 lancers, on en prend 1.000 aléatoirement.**

**On réalise 5 fois cette expérience, et on note m la moyenne des fréquences de 6 obtenus sur les 10.000 lancers et n la moyenne des fréquences de 4 obtenus dans le sous-échantillon.**

**Quelles seraient les valeurs de m et n à l'issue de cette expérience ?**

* + 

L'aléatoire rend les résultats trop variables pour répondre strictement à cette question.

* + 

m et n seront proches de 1/6.

* + 

m sera proche de 1/6, n sera proche de (1/6)/10 = 1/60.

* + 

m sera proche de 1/6 et il n'est pas possible de savoir pour n à cause de l'aléatoire.

*Ce morceau de code devrait pouvoir vous aider à répondre à cette question :*

*import random*

*m = 0*

*n = 0*

*for i in range(5):*

*listeDe = []*

*j = 0*

*while j < 10000:*

*listeDe.append(random.randint(1, 6))*

*j += 1*

*sousEchntillonDe = random.sample(listeDe, 1000)*

*m += listeDe.count(6)/10000*

*n += sousEchantillonDe.count(4)/1000*

*print("m =", m/5, "et n =", n/5)*

### Question 4

**On considère deux jeux de hasard :**

* + Le jeu A est un jeu de pile ou face avec une pièce biaisée (pile avec une probabilité de p=0.49). On lance la pièce. Si l'on obtient pile, on gagne un euro, sinon on perd un euro.
  + Le jeu B est un jeu avec deux pièces biaisées. La pièce 1 donne pile avec une probabilité p1 = 0.09 et la pièce 2 donne pile avec une probabilité p2 = 0.74. Si la somme en jeu de K euros est un multiple de 3, on lance la pièce 1, sinon on lance la pièce 2. Comme dans le jeu A, si l'on obtient pile, on gagne un euro, sinon on perd un euro.

**Le jeu A est clairement perdant. Le jeu B l'est aussi (vous pourrez le vérifier). À présent, on va mixer les deux ! En effet, à chaque tour, on lance une pièce (cette fois-ci...) équilibrée ! Si l'on a pile, on joue au jeu A, sinon on joue au jeu B.**

**On suppose que le joueur a 0 euros comme capital de départ.  
Après avoir joué 1.000.000 de parties, quel est le statut du jeu, du point de vue du joueur ?**

**Je vous conseille fortement d'organiser votre code en plusieurs fonctions, pour pouvoir en simplifier l'organisation et la relecture.**

* + 

Jeu gagnant

* + 

Jeu perdant

* + 

Jeu neutre (les gains sont proches de 0 en fin de compte)

*Dans un premier temps, définissons une fonction pour chaque pièce que nous allons potentiellement lancer (il y en a 4 au total) :*

*import random*

*def pieceEquilibree():*

*if random.uniform(0,1) <= 0.5:*

*return 1*

*else:*

*return -1*

*def pieceJeuA():*

*if random.uniform(0,1) <= 0.49:*

*return 1*

*else:*

*return -1*

*def pieceJeuB1():*

*if random.uniform(0,1) <= 0.09:*

*return 1*

*else:*

*return -1*

*def pieceJeuB2():*

*if random.random() <= 0.74:*

*return 1*

*else:*

*return -1*

*Et ensuite un exemple de programme utilisant ces fonctions et permettant de déduire le gain au bout de 1.000.000 de parties :*

*moy = 0*

*for i in range(10):*

*n = 1000000*

*gain = 0*

*for i in range(n):*

*jeu = pieceEquilibree()*

*if jeu == 1:*

*gain += pieceJeuA()*

*else:*

*if gain % 3 == 0:*

*gain += pieceJeuB1()*

*else:*

*gain += pieceJeuB2()*

*moy += gain*

*print("le gain net est de :", moy /10, "€")*

*Le gain est donc bel et bien positif, autour des 6.000 €. C'est ce que l'on appelle le paradoxe de Parrondo !*

### Question 5

**Si l'on reprend le jeu précédent (présenté en question 4), à combien s'élèvent les gains ou pertes au bout de 1.000.000 de parties ?**

* + 

Proche de 0 €

* + 

Proche de 100 €

* + 

Proche de 13 000 €

* + 

Proche de 6 000 €

*Cf. réponse à la question 4. Faites tourner le code plusieurs fois, vous vous rendez compte que les gains sont proches de 6.000 €.*

### Question 6

**Comment générer N tickets de loterie, soit N numéros entre 0 et 999, qui soient uniques?**

**On peut utiliser la fonction range(p) pour produire une suite d'entiers de 0 a p-1**

# range(p) produit une suite d'entiers de 0 a p-1

population = range(1000)

# Nombre de tickets souhaités

N = 10

* + 

random.choices(population, k=N)

* + 

tickets = []

for n in range(N):

tickets.append(random.choice(population))

* + 

random.sample(population, N)

* + 

tickets = []

for n in range(N):

tickets.append(random.randint(0,1000))

*Seul la fonction sample() permet d'échantillonner sans remplacement. C'est à dire que chaque échantillon de la population est pris de façon unique.*

*Pour s'en assurer, il suffit de lire la documentation de la fonction avec*

*?random.sample*

*qui indique en effet que*

*Signature: random.sample(population, k)  
Docstring:  
Chooses k unique random elements from a population sequence or set.*

*Les autres fonctions utilisées  choices(), choice(), randint() échantillonnent toutes "*avec remplacement*". Ce qui veut dire qu'un échantillon est remis en jeu dans la population après l'échantillonage.*

### Question 7

**Nous essayons de générer un float aléatoire avec la fonction random.**

**Quelle est la meilleure façon d'importer le bon module et d'appeler la bonne fonction pour cela?**

* + 

from numpy import random as rnd

x = rnd.random()

* + 

import numpy as np

x = np.random.random()

* + 

import random

x = random.random()

* + 

Toutes les versions sont équivalentes

*Le module random étant accessible soit directement par import random ou en tant que sous module du module numpy, les 3 versions du code reviennent strictement au même.*

### Question 8

**Quelle affirmation est vraie?**

*Attention, plusieurs réponses sont possibles.*

* + 

Un package est constitué de un ou plusieurs fichiers python et d'un fichier \_\_init\_\_.py

* + 

Un module doit lister ses propres fonctions dans un autre fichier \_\_init\_\_.py

* + 

Les packages et les modules sont importés de la même facon:

import nom\_du\_package

import nom\_du\_module

* + 

Un module est un package constitué d'un seul fichier python

*Un module est un simple fichier python (.py) qui contient des classes, des fonctions ou des variables initialisées.*

*Un package est en ensemble de fichiers python organisés hiérarchiquement et qui contient un fichier \_\_init\_\_.py qui liste les éléments accessibles du package.*