Dr@zielux
martinqt

Apprendre à programmer avec Python 3

12 août 2019

Table des matières

Vous souhaitez apprendre un langage de programmation ? À partir du début ? Alors bienvenue à vous.

Nous allons parler du langage Python, comme l'indique le titre. Aucune connaissance d'un quelconque autre langage de programmation n'est requise. Il est néanmoins conseillé de posséder un ordinateur afin de faciliter votre apprentissage.

Première partie

Les bases de Python

Cette partie regroupe les notions de bases pour commencer à manipuler Python.

1. Présentation

Apprendre Python c'est bien, mais qui est-il et où se situe-t-il par rapport à ses confrères ? Et même avant ça, pourquoi a-t-on besoin de telles bestioles et quel est vraiment leur rôle ?

Nous allons éclaircir ces différents points avant d'installer Python et de commencer à ~~suer~~ programmer.

1.1. Présentation

####Les langages de programmations

Les langages de programmation proviennent du besoin de pouvoir expliquer à l'ordinateur ce que l'on attend de lui. En effet, un ordinateur ne manipule que des 0 et des 1 : c'est *le binaire*. Tous les programmes, de l'éditeur de texte le plus basique au programme de simulation 3D de la NASA, sont traités comme une succession de 0 et de 1 par l'ordinateur. Autant dire que, pour nous, ce n'est pas très parlant. Il a donc fallu créer un intermédiaire, capable de traduire nos instructions en un langage que l'ordinateur puisse interpréter.

Il existe différents « niveaux » de langage. Les langages haut niveau, comme Python, permettent de communiquer avec l'ordinateur à un niveau plus élevé et de cacher les opérations élémentaires effectuées par l'ordinateur. Les langages bas niveau, par opposition, gèrent les opérations de l'ordinateur à un niveau plus fin et plus proche de ce qu'il se passe réellement dans l'ordinateur. On peut par exemple citer le C comme exemple de langage bas niveau. Le principal avantage des langages bas niveau est qu'ils donnent un contrôle fin des opérations effectuées par l'ordinateur, ce qui peut permettre une optimisation plus poussée. Les langages de haut niveau sont généralement plus simples à manier.

Nous ne rentrons pas plus dans les détails des différents niveaux de code qui existent, ce n'est pas l'objet de ce cours, mais si cela vous intéresse, il existe diverses ressources sur le net.

Python peut être employé pour diverses applications. Dans les exemples qui suivent, des liens vers des bibliothèques sont proposés. Une bibliothèque est un ensemble de code écris par d'autres développeurs dans le but de simplifier certaines tâches. Vous serez probablement amené à en utiliser plus tard. Pour le moment, ces exemples sont présents à titre informatif et vous concerneront surtout quand vous aurez acquis les bases de Python. Python peut donc notamment servir à :

— Créer des sites web, comme celui sur lequel vous êtes. Des bibliothèques comme Django ou Flask sont souvent utiliser.
— Des jeux, notamment grâce à PyGame .
— Du calcul scientifiques avec par exemple SymPy ou Numpy , et du tracer de graphique avec matplotlib .

Si vous êtes curieux sur l'histoire de Python, vous pouvez par exemple jetez un coup d'œil ici pour en avoir un aperçu.

Ce cours parlera de Python **3** et non pas de Python **2**. Outre le numéro de version, de nombreuses différences sont apparues entre ces deux versions majeures. Ainsi, un code fonctionnant sur Python 2 ne marchera pas forcément sur Python 3 et inversement. Soyez vigilants lorsque vous parcourez le web, pour poser une question ou chercher une réponse, en vérifiant la version de Python concernée, ou en l'indiquant si vous demandez de l'aide. Nous parlerons de Python 3 car, à terme, Python 2 deviendra obsolète. Ceci n'est néanmoins pas pour tout de suite, et les deux versions coexistent tant bien que mal. Vous serez également confronté à ce problème plus tard, lorsque que vous vous intéresserez à des librairies tierces : certaines fonctionnent sur les deux versions, d'autre proposent une version pour chaque version Python et certaines ne fonctionnent que pour une seule version de Python. Soyez donc vigilant.

1.2. Installation

À l'heure où sont écrites ces lignes, la version la plus récente de Python est la version 3.4.3. Ce tutoriel partira du principe que vous avez une version 3.4 et ne présentera donc pas de fonctionnalités n'apparaissant plus dans cette version ou prévues pour des versions futures.

Python possède une documentation en ligne ici . Celle-ci recense tout ce que Python peut faire dès son installation. Elle constitue une référence et vous sera utile plus tard, alors garder ce lien sous le coude. Soyez néanmoins prévenu, tout est en anglais.

####Windows

Rendez-vous ici et téléchargez `Python 3.4.a` où `a` est un nombre quelconque. Veillez surtout à ce que le premier chiffre soit bien **3** et non *2*.

Maintenant, lancez le programme d'installation que vous venez de télécharger (par défaut, son nom est `python-3.4.a.msi`)

Vous pouvez changer le répertoire d'installation mais le chemin ne **doit pas** contenir d'espace (évitez donc `Programme Files`). De façon générale, évitez tout ce qui est espace et accent dans vos répertoires de développement. Le répertoire par défaut est par exemple un bon choix.

Continuez jusqu'à l'étape suivante :

http://zestedesavoir.com/media/galleries/1783/

FIGURE 1.1. – Installation de Python - Étape Customize

Vous devez cliquer sur la croix et choisir l'option `Will be installed on local hard drive` comme montré ci-dessus. Continuez ensuite l'installation normalement jusqu'à ce que l'installateur se ferme.

Maintenant tapez Windows + R (Windows étant la touche en bas à gauche entre Ctrl et Alt). La fenêtre « Exécuter » s'ouvre. Saisissez `cmd` puis Entrée. Une fenêtre noir et blanche s'ouvre alors. Tapez alors `python` puis Entrée. Si un message d'erreur apparait disant que python n'est pas une commande reconnue, saisissez `chemin-installation\python\` où `chemin-installation` est l'endroit où vous avez installé Python (par défaut : `python34`).

Une ligne commençant par `>>` devrait être affichée si tout se passe bien.

####Linux

Sous Linux, vous avez généralement la chance de posséder des éditeurs supportant la coloration syntaxique déjà installés. Il s'agit, par exemple, de Kate sous KDE, ou de gedit sous Gnome.

Sous Linux, vous possédez déjà une version de Python, mais il est possible qu'il s'agisse de Python 2 et non 3. Il faudra alors installer la version 3 *en plus* de celle existante. Pour savoir quelle version vous possédez ouvrez un terminal et tapez `python` puis Entrée. La première ligne qui s'affiche va alors commencer soit par `Python 3` soit par `Python 2`. Si vous avez une erreur vous indiquant que cette commande n'est pas reconnue, ou si vous voyez écrit `Python 2`, installez Python 3.

Chaque distribution Linux possède ses particularités. Généralement, vous installerez Python 3 à l'aide de votre gestionnaire de paquets. Sous Debian et ses dérivés utilisez `sudo apt-get install python3`.

Il vous faudra ensuite utiliser la commande `python3` et non `python` tout au long de ce cours pour lancer Python 3. Si tout se passe bien, une ligne commençant par `>>` devrait être affichée.

Maintenant que tout est prêt, entrons dans le vif du sujet.

2. Première approche

Maintenant que la bestiole a gentiment intégré votre ordinateur, voyons ce qu'elle a dans le ventre.

2.1. Présentation de l'interpréteur

Je vais supposer que vous savez qu'exécuter ou valider une commande signifie appuyer sur Entrée et que vous pouvez lancer votre interpréteur comme expliqué dans le chapitre sur l'installation.

Une fois votre interpréteur lancé, Python vous informe qu'il est prêt et attend vos instructions en ajoutant `>>` au début de la ligne.

Si vous tentez de taper ce qu'il vous passe par la tête, vous allez très probablement vous retrouver avec une erreur. Par exemple :

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

Ceci est donc votre première erreur (et ce ne sera pas la dernière). Elle commence généralement par `Traceback`, ce qui signifie que ce qui suit est un récapitulatif de certaines actions qui ont eu lieu avant l'erreur. Vient ensuite le nom de l'erreur, dans ce cas `NameError`, suivi d'un message plus détaillé sur l'erreur, vous permettant de comprendre ce qui met Python en colère. Ici, Python nous informe que le nom `salut` ne correspond à rien de connu pour lui. Nous détaillerons les erreurs au fur et à mesure que nous les croiserons.

Nous allons donc apprendre à utiliser des instructions valides que Python saura interpréter.

2.1.1. Un peu de mathématiques

Mais non, ne partez pas, c'est lui qui va calculer. Nous allons commencer par la base, à savoir vérifier que nous pouvons saisir des valeurs numériques.

```
>>> 42
42
>>> 4.2
4.2
```

Ce coup-ci pas d'erreur en vue. Enfin, vous me direz peut être que le fait qu'il radote ne va pas dans ce sens. En réalité, Python ne radote pas, mais nous répond. On le sait, car la ligne ne commence pas par `>>`. Pour le moment, on lui a dit `42` et il nous répond `ça fait 42`.

Notez bien l'utilisation du point et non de la virgule pour les nombres à virgule. Nos amis anglo-saxons étant passés par là, il va falloir utiliser leur convention. La virgule a une autre signification que nous verrons plus loin.

Maintenant les opérations de base :

```
>>> 3.2 + 1 # Addition
4.2
>>> 7 - 3 # Soustraction
4
>>> 6 * 5 # Multiplication
30
>>> 1 / 3 # Division
0.3333333333333333
```

Vous avez sans doute remarqué le `#` suivi de texte. Ceci s'appelle un *commentaire*. Le dièse informe Python que tout ce qui se trouve sur la *même ligne* après lui ne le concerne pas : il va donc simplement l'ignorer. L'utilité non négligeable est que vous pouvez ainsi laisser des notes dans votre code pour vous souvenir de ce qu'il fait. Ceci deviendra de plus en plus utile au fur et à mesure que vos projets grandiront. Je vous recommande donc de prendre l'habitude de commenter votre code dès maintenant. Vous pouvez également écrire un commentaire sur plusieurs lignes en l'encadrant par `"""` au début et à la fin, comme ceci :

```
"""
Un commentaire
sur plusieurs lignes que Python ignorera
"""
```

Remarquez que la dernière commande renvoie un résultat approché. En effet, la représentation par défaut des nombres à virgule ne peut pas comporter un nombre de décimales infini. Ces erreurs d'approximation peuvent se cumuler et donner des résultats surprenant. Par exemple :

```
>>> 9.3 - 8.5
0.8000000000000007
```

Rassurez-vous, il existe des moyens de contourner ces problèmes d'arrondi, mais pour des raisons de simplicité, nous ne les aborderons pas ici. Gardez à l'esprit que travailler avec des nombres à virgule peut réserver des surprises à cause des arrondis, même si vous ne les remarquerez pas la plupart du temps.

2.1.1.1. D'autres opérations

Vous ne pensiez pas que c'était déjà fini ? Nous allons voir trois autres opérations : la division entière, le modulo et l'élévation à une puissance.

Les deux premières correspondent au quotient et au reste des divisions que l'on posait à la main. Par exemple :

```
>>> 13 // 3 # Le quotient
4
>>> 13 % 3 # Le reste
1
```

Et l'on trouve bien `13 = 3 * 4 + 1`.

La puissance, quant à elle, se note comme ceci :

```
>>> 2 ** 3
8
```

Cette opération est parfois notée `^` dans certains logiciels, tels que Excel. Néanmoins le `^` possède une autre signification en Python qui ne nous intéresse pas pour le moment. Soyez donc prudent car Python ne vous retournera pas forcément une erreur.

Python suit l'ordre conventionnel de priorité des opérations mathématiques (pour rappel, l'exponentiation en premier, suivit de la multiplication et la division, et finalement de l'addition et la soustraction). Vous pouvez néanmoins changer cet ordre en utilisant des parenthèses pour donner la priorité à une opération, comme en mathématiques.

2.2. Ecrire son code dans des fichiers

L'interpréteur est bien pratique pour tester de petits bouts de code, mais il présente plusieurs inconvénients, l'un d'entre eux étant que quand vous le fermez, tout votre code disparait. C'est alors qu'interviennent les fichiers, qui nous permettrons de sauvegarder notre travail.

Tout d'abord, je vous recommande d'utiliser un éditeur de texte un peu avancé et spécialisé dans l'écriture de code. La principale fonctionnalité qui nous sera utile pour le moment est la *colorisation syntaxique*. Si vous possédez déjà un éditeur de texte avec cette fonctionnalité, pas besoin d'en installer un autre. Si ce n'est pas le cas, un premier choix vous sera indiqué dans la partie réservée pour votre système d'exploitation. Plus tard vous pourrez peut-être vous tourner vers un *EDI* (ou *IDE* en anglais), qui propose plusieurs fonctionnalités avancées pour vous assister dans vos grands projet, mais qui sont souvent déroutantes quand on débute.

Prenez l'habitude de nommer vos fichiers avec des noms clairs, sans espace ou caractères spéciaux. Cela vous évitera des mauvaises surprises plus tard.

Cette procédure différant quelque peu selon votre système d'exploitation, veuillez-vous reporter à la partie vous concernant.

####Windows

Vous pouvez utiliser Notepad++ pour commencer. Sachez qu'il existe de nombreux choix, libre à vous d'utiliser un autre logiciel si celui-ci n'est pas à votre convenance.

Une fois dans votre éditeur, tapez votre code puis enregistrez votre fichier. Au moment de l'enregistrement, spécifiez l'extension `.py`. Ouvrez votre explorateur de fichier et rendez vous dans le dossier où vous avez sauvegardé votre fichier.

Maintenant, pour exécuter votre code, copiez le chemin d'accès du répertoire. Faites ensuite shift + clic droit. Vous aurez alors accès à une ligne s'intitulant `Ouvrir une fenêtre de commandes ici`. Cliquez dessus, et vous aurez alors une console directement située dans ce dossier.

http://zestedesavoir.com/media/galleries/1783/

Si par malheur vous ne parvenez pas à effectuer l'opération présentée si dessus, voici une méthode plus longue, mais qui devrait être exempt de la plupart des caprices de votre OS. Tapez Windows + R puis `cmd` puis Entrée. Nous revoilà dans la console mais les choses diffèrent un peu maintenant. Tapez `cd chemin` où `chemin` est le chemin d'accès du répertoire, puis Entrée. Pour coller le chemin d'accès, effectuez un clic droit puis `coller` (Ctrl + V ne fonctionnera pas).

Maintenant, tapez `python nom-fichier.py`. Si dans les chapitres précédant vous deviez lancer Python en spécifiant son nom de chemin complet, remplacer alors `python` par le nom du chemin que vous utilisiez.

Si vous avez l'impression que rien ne se passe et que vous n'avez pas d'erreur, c'est normal. Rendez vous plus bas, où les explications communes reprennent.

2.2.1. Quelques modifications

Nous allons voir quelques points pour que votre séjour en compagnie de vos fichiers se passe dans les meilleures conditions.

2.2.1.1. L'encodage

L'encodage est la façon dont les ordinateurs se représentent le texte. Il existe de nombreux encodages différents, et donc des représentations différentes. Python a besoin de savoir dans quel encodage vous travaillez pour pouvoir lire correctement votre fichier. Les problèmes sont généralement le plus visible quand vous commencez à utiliser des accents par exemple. Généralement, si vous êtes sous Windows, vous travaillez vraisemblablement en Latin-1, alors que si vous êtes sous Linux ou Mac, il est plus probable que vous utilisiez UTF-8. Pour spécifier votre encodage, ajoutez `# -*-coding:encodage -*` en haut de votre fichier. Remplacez `encodage` par celui que vous utilisez, par exemple :

```
1 # -*-coding:Latin-1 -*
2 # Généralement sous Windows
3
4 # -*-coding:Utf-8 -*
5 # Généralement sou Linux ou Mac
```

Cette ligne doit apparaitre *au début de tous* vos fichiers.

Par défaut, Python va considérer que vos fichiers sont encodés en Utf-8. Sous Linux ou Mac, vous pouvez donc omettre cette ligne puisque qu'il est très probable que l'encodage par défaut de votre fichier soit aussi Utf-8. Néanmoins, si vous travaillez sur des ordinateurs avec des OS différents, il est préférable d'ajouter cette ligne afin d'éviter les mauvaises surprises.

2.2.1.2. Écrire dans la console

Vous vous êtes probablement rendu compte que Python ne nous parle plus. C'est normal. Il se contente d'exécuter le code dans notre fichier. N'étant plus chez lui, il est beaucoup moins bavard et ne parle que si on le lui demande. Comment lui dire de parler ? Comme ceci :

```
1 print(5 + 3)
```

Je vous l'accorde, il y a un peu de changement, mais ça sera plus simple pour la suite. Vous êtes dans un fichier donc plus de `>>` car vous êtes ici chez vous. Python ne viendra pas vous répondre ici, mais dans la console et quand nous lui demanderons (ou si il rencontre une erreur). Tout ce qui se trouve dans le fichier sera interprété au moment où vous l'exécuterez.

Pour l'étrangeté de la ligne de code donnée, `print`, qui signifie « imprimer » en anglais, est ce que l'on appelle une *fonction*. Pour le moment, retenez juste que Python va afficher dans la console le résultat de ce qui se trouve entre les parenthèses de `print`. Dans notre exemple, le résultat de `5 + 3` sera affiché dans la console. Ne vous inquiétez pas, nous allons nous servir régulièrement de `print` et vous allez vite vous y habituer. Nous reviendrons sur les fonctions plus tard.

Maintenant que nous pouvons sauvegarder nos petits ~~bébés~~ programmes, nous allons pouvoir les complexifier sans craindre de devoir tout retaper la fois suivante (pourvu que vous pensiez à sauvegarder bien sûr).

3. Les variables

Nous savons donc effectuer des opérations mathématiques avec Python. Mais nos résultats se perdent à la fin de chaque calcul sauf si on les affiche. Et même dans ce cas, nous ne pouvons pas récupérer le résultat pour le réutiliser. C'est ici qu'intervient la notion de *variable*.

3.1. Présentation

Mettons tout cela en contexte. Votre ordinateur possède de la *mémoire* où il *stocke* des informations. Nous pouvons représenter la mémoire comme un ensemble de cases, chacune contenant une valeur *différente*. Lorsque nous créons une variable, nous lui assignons un *nom*, qui sera tel une étiquette que l'ordinateur collera sur cette case.

####Les variables en Python

Voyons comment tout ceci se traduit en Python. D'abord, du code :

```
1 a = 5
```

Ici, `a` est une variable. Comment ça c'est tout ? Oui, on vient de dire à Python que la variable `a` correspond à la valeur `5`. C'est une *déclaration* (on a créé une variable) suivi d'une *affectation* (on a modifié sa valeur).

? Je peux mettre ce que je veux comme nom de variable ?

Presque. Il y a quelques règles à suivre :

— Vos noms de variable ne commenceront pas par un chiffre.
— Vos noms de variable peuvent avoir des lettres, en minuscule ou majuscule et des chiffres.
— Vos noms de variable ne seront pas un mot-clé réservé par Python.

Les points suivant sont d'avantages de l'ordre de la recommandation :

— Vos noms de variable peuvent contenir `_` (un « underscore ») mais *éviterons* d'autres caractères spéciaux (accents compris).
— Vos noms de variable *éviteront* d'être le nom d'une fonction prédéfinie de Python.
— Vos noms de variable seront *si possible* en anglais (afin que votre code soit facilement compréhensible par le plus grand nombre).

Voici la liste des mots-clés réservés par Python :

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	async
def	for	lambda	return	await

`async` et `await` ne seront des mots reservés qu'à partir de la version `3.5`. Prenez néanmoins l'habitude de ne pas les utiliser si possible.

Ainsi `bon_Jour1` est un nom valide alors que `1.bonJour` ne l'est pas. Il faut également noter que Python fait la différence entre les majuscules et les minuscules : vous pouvez donc avoir une variable `bonjour` et une autre `bonJour` de valeurs différentes.

De manière générale, préférez des noms de variables clairs, évocateurs et pas trop longs. Vous serez bien content quand vous commencerez à cumuler les variables de pouvoir vous y retrouver facilement.

Vous avez peut-être essayé de faire un `print` de cette ligne. Python vous a alors renvoyé une erreur de type `TypeError`. Vous comprendrez mieux pourquoi plus tard, mais pour l'instant, représentez-vous ceci : `print` affiche un résultat, mais l'affectation de la variable se contente de stocker, elle n'a rien à nous dire. En réalité, il s'agit d'un problème de syntaxe que nous verrons plus loin.

3.1.0.1. Manipulons nos variables

Commençons :

```
1 a = 5
2 print(a) # Affiche 5
3
4 b = 8.2
5 print(a + b) # Affiche 13.2
```

Nous avons d'abord déclaré une variable puis affiché sa valeur. En effet, quand `print` reçoit le nom d'une variable, il va écrire la valeur de cette variable dans la console. Puis nous créons une deuxième variable. Nous tentons alors de faire une opération entre variables et d'afficher le résultat. Et tout se passe bien (pour une fois).

Détaillons un peu cette dernière ligne. Python a repéré l'opération entre les variables. Il est alors parti chercher les valeurs des variables. Il a ensuite effectué le calcul, puis il a affiché le résultat. Voyons comment modifions nos variables maintenant.

```
print(a) # Affiche 5
a = 10
print(a) # Affiche 10
a = b + 1
print(a) # Affiche 9.2
```

Comme vous le voyez, un simple égal suffit pour indiquer que l'on souhaite remplacer la valeur de la variable. Vous remarquerez au passage que `a` était un entier mais qu'il s'agit maintenant d'un nombre à virgule. La valeur peut être le résultat d'un calcul, pouvant impliquer d'autres variables, voir la variable que l'on modifie elle-même. Exemple :

```
c = 1
c = c + 3 # c vaut 4
```

Vous pouvez faire un `print` pour vous en convaincre. Python a commencé par évaluer l'expression à droite du signe `=`, allant chercher l'ancienne valeur de `c`. Fort du résultat de son calcul, il l'a ensuite stocké dans la variable `c`. Ainsi, Python n'a eu aucun problème pour effectuer cette opération puisqu'il a commencé par calculer la nouvelle valeur avant de la stocker.

3.1.0.2. Un peu de sucre

Non, ce n'est pas encore l'heure du dessert. Quoique. Je vais ici vous donner quelques notations pour alléger votre code. Ces notations sont des raccourcis et leur action peut être effectuée avec les éléments vus précédemment. De tels raccourcis sont appelés *sucre syntaxique*.

Vous vous rendrez compte que l'on est souvent amené à écrire des instructions du type `i = i + 1`, par exemple pour compter le nombre de fois qu'une action est effectuée. Pour ne pas avoir à recopier le `i` deux fois, de nouveaux opérateurs ont été introduits :

Opérateur	Équivalent
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a **= b`	`a = a ** b`
`a /= b`	`a = a / b`
`a //= b`	`a = a // b`
`a %= b`	`a = a % b`

Comme vous le constatez, il ne s'agit que de raccourci, rien de véritablement nouveau. `b` représente une variable, mais une variable peut être le résultat d'un calcul quelconque. Vous pouvez donc remplacer `b` par ce calcul si vous le désirez, au lieu de passer par une autre variable.

Voyons maintenant un deuxième raccourci :

```
1 a, b = 4, 5.2 # a vaut 4 et b vaut 5.2
```

Vous pouvez mettre plus de deux variables/valeurs, séparées par des virgules. Ceci permet d'affecter plusieurs variables sur une ligne. Ainsi, la première variable de gauche, prend la première valeur de droite et ainsi de suite. Cette écriture présente un avantage supplémentaire : l'échange de valeur de variables.

Supposons que l'on souhaite intervertir les valeurs de `a` et `b`. La première idée qui pourrait surgir est :

```
1 a = b
2 b = a
```

Malheureusement, nous nous sommes peut-être un peu précipités. En effet, après la première ligne, `a` vaut la même valeur que `b`. A la ligne suivante, `a` et `b` ont donc la même valeur : nous avons perdu la valeur de `a`. La deuxième affectation ne changera donc rien. Pour contourner ce problème, il va falloir créer un troisième pour stocker la valeur de `a` le temps de l'échange.

```
1 c = a
2 a = b
3 b = c
```

Cette fois-ci, tout se passe bien car nous avons gardé la valeur de `a` en mémoire et à l'abri dans `c`. Cette écriture est néanmoins un peu lourde, surtout si vous voulez échanger plus de deux variables. Mais vous voyez peut-être un raccourci pointer le bout de son nez :

```
1 a, b = b, a
```

Ceci marche de façon similaire à `a = a + b`. La valeur à droite est d'abord calculée puis affectée, de sorte qu'il n'y ait aucun conflit. Dans ce cas, Python voit qu'il va devoir stocker le couple `b, a`. Il calcul donc ces valeurs, puis va les stocker dans le couple `a, b` et dans cet ordre. Cette méthode fonctionne également pour plus de deux variables.

Vous pouvez vous amuser et vous familiariser avec les différentes opérations vues dans ce chapitre. Il est d'ailleurs recommandé de faire des tests soi-même pour bien assimiler le concept. Encore une fois, n'hésitez à faire des `print` pour contrôler.

Manipuler des nombres c'est bien gentil, mais c'est tout ce que connait Python ? Non, et c'est ce que nous allons découvrir.

3.2. Les types de données

En réalité, nous avons manipulé déjà deux types différents : les entiers et les nombres à virgule. Nous n'avons pas vu de différence car Python sait passer d'un type à l'autre quand le calcul le nécessite. Pourquoi différencier deux types en apparence aussi proches ? Car ils ne sont pas stockés de la même façon en mémoire. Python doit donc savoir de quel type est la variable pour pouvoir lire correctement sa valeur. Je vous sens bien peu convaincu. Demandons au principal intéressé ce qu'il en pense :

```
print(type(1)) # Affiche <class 'int'>
print(type(1.0)) # Affiche <class 'float'>
```

La méthode `type` permet de récupérer le type de la variable tel que Python le voit. Voyons maintenant ce que nous apprend l'affichage de ce résultat. Dans un premier temps, l'information `class` ne nous concernera pas vraiment. Nous aborderons cette notion plus tard, quand nous aurons déjà un peu plus d'expérience. Vient ensuite l'information qui nous intéresse ici : `int` et `float`. Il y a donc bien deux types différents. Pour votre culture, `int` est un diminutif pour « integer » qui signifie entier en anglais. `float` est une abréviation de « floating », signifiant flottant. En effet, les nombres à virgule sont aussi appelés nombre à virgule *flottante*.

Je sens votre inquiétude grandissante : on avait dit autre chose que des nombres. On y arrive, mais il est bon de se rendre compte que nous avons déjà manipulé plusieurs types et il est important de les connaitre.

3.2.1. Les chaînes de caractères

Python sait aussi manipuler du texte, sous forme de chaîne de caractères. Pour indiquer à Python qu'il s'agit d'une chaîne de caractère, il faut utiliser des délimiteurs. Il en existe trois : `'`, `"` et `"""`. Les délimiteurs se placent de part et d'autre de la chaîne, comme ceci :

```
chaine = 'Et voilà du texte'
chaine2 = "Et encore du texte"
chaine3 = """Toujours plus de texte"""
```

Ces trois types de délimiteurs différents ont une utilité :

```
chaine = 'Aujourd'hui' # Erreur
chaine2 = "J'ai cassé Python"
```

```
chaine3 = """Demain,
je le réparerai"""
```

La première ligne va créer une erreur : en effet, l'apostrophe dans `aujourd'hui` est interprété comme la fin de la chaîne, et Python ne sait alors que faire de `hui'`. Ce problème est résolu dans la deuxième ligne car le délimiteur n'est plus `'` mais `"`. Si on avait mis des guillemets, comme pour un dialogue, il aurait fallu utiliser `'` pour ne pas gêner Python. Rassurez-vous, cela ne signifie pas que vous ne pouvez pas mettre des apostrophes et des guillemets simultanément. Il existe un caractère bien utile qui va nous sauver : le *backslash* `\`.

Le backslash indique à Python que le caractère qui le suit doit être *échappé*, c'est à dire qu'il fait partie du texte et non code. Ainsi :

```
chaine = 'Nous l\'avons'
chaine2 = "\"réparé\""
```

Aucune erreur ici, puisque Python sait maintenant qu'il ne s'agit en fait pas de la fin de la chaîne.

Le backslash peut, selon le caractère qui le suit, avoir une signification particulière. Par exemple, `print('Bonjour,\nVisiteur')` va afficher `Bonjour,` puis `Visiteur` sur une autre ligne. `\n` signifie donc « retour à la ligne » pour Python.
Si vous souhaitez juste afficher un backslah, il faudra alors utiliser `\\` afin que Python sache qu'il doit échapper le second `\`, et le considérer comme du texte.

Venons-en à l'utilité du dernier délimiteur. Celui-ci permet contrairement à ses deux confrères d'écrire le texte de notre chaîne sur plusieurs lignes. De plus, si vous effectuez un retour à la ligne, vous n'avez pas besoin d'ajouter un `\n` pour que celui-ci s'affiche avec `print`.

Il existe d'autres types de données mais nous nous contenterons de cela pour l'instant. La manipulation des chaînes de caractères sera abordée dans un autre chapitre.

3.2.2. Conversion

Python est capable de convertir certains types de données. Voyons ceci avec un bout de code :

```
var = 10
print(type(var)) # Affiche <class 'int'>
var = str(var)
print(type(var)) # Affiche <class 'str'>
var = float(var)
print(type(var)) # Affiche <class 'float'>
```

Les conversions sont bien effectuées. Attention à ne pas demander des conversions improbables à Python non plus : par exemple `int('a')` va retourner une `ValueError`.

Une utilité ? Allez, je vais vous montrer un petit truc, mais le dites pas aux autres, ils vont être jaloux.

3.2.2.1. Poser des questions

Nous savons afficher des informations dans la console, voyons comment demander à l'utilisateur d'en saisir. Pour cela, on utilise `input` :

```
1 reponse = input() # Une ligne vide apparait et attend que
      l'utilisateur entre une information
2 age = input("Age : ") # "Age : " est affiché en début de ligne puis
      attend une information
3 # `age` et `reponse` contiennent ce que l'utilisateur a entré
```

Nous pouvons donc communiquer avec l'utilisateur. Cependant, les variables `age` et `réponse` sont de type `str` même si l'utilisateur a entré un nombre. Il ne sera donc pas possible, par exemple, de soustraire un nombre à l'âge récupéré. Ah moins que ... ah bah si, il suffit de convertir `age` en un `int`.

Prochaine étape, modifier le comportement de son programme en fonction de conditions.

4. Conditions et notion de boucles

Effectuer des calculs, manipuler des variables c'est bien, pouvoir prendre des décisions en fonction des résultats, c'est plus marrant.

4.1. Les structures conditionnelles

Nous allons commencer par un exemple. Notre but est de demander l'âge de l'utilisateur et de déterminer si il/elle a plus de 16 ans par exemple.

```
1 age = int(input("Quel est votre âge ? ")) # Souvenez-vous, il faut
      convertir en un entier
2
3 if age > 16: # Si l'âge est strictement supérieur à 16 (ans)
4     print("Vous avez plus de 16 ans :)")
```

Voyons ce qui s'est passé. Nous avons commencé par demander son âge au visiteur tout en nous assurant de récupérer un entier. Nous avons ensuite effectué un test : si la variable `age` est supérieur à 16, on affiche un message. Dans le cas contraire rien ne se passe. Nous verrons les différents tests que l'on peut effectuer dans la suite de ce chapitre, mais pour l'instant concentrons-nous sur la structure.

Vous avez sans doute remarqué la présence d'espaces devant le `print`. On appelle *indentation* le fait de décaler une ou plusieurs lignes de code à l'aide d'espaces, généralement 4 comme recommandé par Python, ou d'une tabulation (touche Tab à gauche de la touche A). Cette indentation est requise pour le bon fonctionnement de la condition. En effet, Python a besoin de savoir ce qui doit être exécuté uniquement si la condition est vérifiée et ce qui sera toujours exécuté. Les lignes indentées après le `if` forment les instructions qui seront uniquement exécutées si la condition est vérifiée. Les lignes alignées avec le `if` seront, elles, toujours exécutées.

A partir de maintenant, vous êtes susceptibles de rencontrer une `IndentationError`. Ce type d'erreur indique que vous avez un problème d'indentation. Il peut s'agir d'un `if` qui n'est suivi d'aucun bloc indenté ou d'un nombre incohérent d'espace utilisé tout au long de votre programme : si vous utilisez 4 espaces pour indenter la première fois, n'en mettez pas 5 au `if` suivant, sinon Python va râler. Si vous utilisez Tab, vous avez probablement oublié ou mis une tabulation de trop.

Maintenant, ajoutons quelques éléments :

```python
age = int(input("Quel est votre âge ? "))

if age > 16: # Si l'âge est strictement supérieur à 16 (ans)
    print("Vous avez plus de 16 ans :)")
elif age < 0: # Si l'âge est strictement inférieur à 0
    print("Tu te moquerais pas de moi ?")
else: # Dans tous les autres cas
    print("Tu es encore un peu jeune")

print("Au revoir")
```

Détaillons les trois mots clé introduits :

- `if`, c'est-à-dire « si », marque le début de la structure conditionnelle. Il est suivi d'une condition. Il n'y a qu'un seul `if` dans une même structure conditionnelle. C'est également la première condition à être vérifiée.
- `elif`, qui correspond à « sinon si », est aussi suivi d'une condition. Il peut y avoir plusieurs `elif` dans une même structure de condition. La condition ne sera testée que si aucune des conditions précédentes se trouvant dans la même structure n'est vérifiée. De plus, les `elif` seront testés dans l'ordre dans lequel ils se trouvent dans votre fichier.
- `else` correspond à « sinon » ou « dans tous les autres cas ». Il n'est pas suivi d'une condition. Il n'y a forcément qu'un seul `else` par structure. En effet, son rôle est d'être nécessairement exécuté si **aucune** des conditions au-dessus de lui n'a été vérifiée.

Les instructions `elif` et `else` peuvent ou non faire partie de votre structure.

Détaillons maintenant notre exemple. Une fois l'âge récupéré, plusieurs scénarios sont possibles :

- si `age` est strictement plus grand que 16, `print("Vous avez plus de 16 ans :)")` est exécuté.
- sinon si `age` est strictement plus petit que 0, `print("Tu ne te moquerais pas de moi ?")` est exécuté.
- dans tous les autres cas, ce qui correspond ici à un âge entre 0 et 16, `print("Tu es encore un peu jeune")` est exécuté.
- Dans tous les cas, c'est-à-dire peut importe l'âge, `print("Au revoir")` est exécuté.

Dès qu'une condition est vérifiée, le bloc d'instructions associé est exécuté et le programme reprend son exécution à la sortie de la condition, c'est-à-dire dans notre exemple à `print("Au revoir")`. Ainsi, les conditions suivantes ne seront pas testées et directement ignorées.

Vous pouvez, à la suite de la première condition, en ajouter une deuxième. Celle-ci commencera par un `if` et sera indépendante de la précédente. Comme toujours, n'hésitez pas à faire des tests.

Maintenant que vous connaissez la structure des conditionnelle, nous allons aborder les différentes conditions que vous pouvez utiliser.

4.2. Les opérations logiques

4.2.1. Un nouveau type variable

Et devinez quoi, on s'en est servi sans même le voir. Je vous présente donc le type *booléen* (*bool* pour les intimes) :

```
1 vrai = True # Notez bien la majuscule
2 faux = False # Ici aussi
3 print(type(vrai)) # Affiche <class 'bool'>
```

Et voilà. Mesdames et messieurs merci pour vos applaudissements. Plus sérieusement, le type bool ne prend que deux valeurs, vrai (`True`) et faux (`False`). Vous l'avez peut être compris, il s'agit donc du résultat d'un test. Ainsi, un `if` est considéré comme vérifié si la condition renvoie `True`. Un peu de vocabulaire, la condition est appelée le *prédicat*, et nous emploierons ce terme à présent.

4.2.2. Les comparaisons

La description parle d'elle-même, je passerai donc rapidement sur ce point, mais comme toujours, n'hésitez pas à tester.

Opérateur	Description
`a > b`	`a` *strictement* supérieur à `b`
`a < b`	`a` *strictement* inférieur à `b`
`a >= b`	`a` supérieur ou *égal* à `b`
`a <= b`	`a` inférieur ou *égal* à `b`
`a != b`	`a` différent de `b`
`a == b`	`a` égal à `b`

Attention à ne pas confondre `=` et `==`. Le premier effectue une affectation, le second une comparaison.

Le prédicat, après évaluation, donne une variable de type `bool`. Ainsi, si `a` est un booléen, `if a:` équivaut à `if a == True:`.

4.2.2.1. Combinaison

Vous pouvez utiliser plusieurs de ces opérateurs dans une même condition :

```
print(7 > 5 > 1) # Affiche True
print(7 > 5 < 9 != 10) # Affiche True
```

Vous pouvez bien entendu utiliser des variables. Attentions toutefois à ne pas abuser de ce type de conditions, vous pourriez vous retrouver avec des conditions rapidement peu incompréhensibles, surtout si vous devez vous relire après plusieurs mois. On retiendra néanmoins qu'il est facile de tester des encadrements, notamment de variables, en utilisant cette méthode.

4.2.3. Les opérateurs logiques

Il est possible d'imbriquer des conditions en écrivant un `if` dans un `if` comme ceci :

```
if couleur == "rouge":
    if poids > 10:
        # La couleur est rouge et le poids supérieur à 10
```

Bien qu'utile dans certains cas, dans cet exemple, il serait plus pratique de *combiner* des prédicats. Ça tombe bien, c'est justement ce dont nous allons parler.

4.2.3.1. Le ET logique

Supposons que nous avons deux prédicats A et B. Nous allons nous intéresser au *ET* logique. Le tableau qui suit est appelé *table de vérité* de ET. Le tableau se lit comme suit : pour une ligne donnée, les deux premières colonnes indiquent la *valeur de vérité* du prédicat A et B (c'est-à-dire si oui ou non ils sont vérifiés). La troisième colonne donne alors la valeur de vérité de l'opération logique étudiée. Ainsi par exemple, la première ligne de la table suivante signifie : « si le prédicat A n'est pas vérifié et si le prédicat B n'est pas vérifié, alors le prédicat A ET B n'est pas vérifié ». Si vous vous faîtes des recherches sur le net, vous trouverez peut-être des tables de vérités utilisant 1 pour représenter `True` et 0 pour `False`.

A	B	A ET B
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

Je vais vous montrer la syntaxe Python en reprenant l'exemple précédent :

```python
if couleur == "rouge" and poids > 10:
    # La couleur est rouge et le poids supérieur à 10
```

Vous pouvez combiner plus de deux prédicats en utilisant plusieurs `and`.

4.2.3.2. Le OU logique

Voici tout d'abord sa table de vérité :

A	B	A OU B
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Cette opération est aussi appelée *OU inclusif*. C'est à dire que si vos deux prédicats sont être vrais, le OU retournera vrai aussi. Il ne s'agit pas du *OU exclusif*, qui n'autorise qu'un seul des deux prédicats à être vrai à la fois. Le ou exclusif est celui utilisé dans « fromage ou dessert » : vous n'avez pas le droit aux deux petits gourmands.

Et voici sa notation Python :

```python
if couleur == "verte" or poids == 15:
    # La couleur est verte ou le poids égal à 15 ou les deux en
        même temps
```

De la même façon que pour `and` vous pouvez utiliser plusieurs `or` dans un prédicat, et même combiner des `and` et des `or`. Il faut néanmoins bien noter que le ET sera *prioritaire* sur le OU, tout comme la multiplication est prioritaire sur l'addition. Vous pouvez utiliser des parenthèses, comme pour les opérations, afin de changer la priorité.

4.2.3.3. Le NON logique

Si vous souhaitez vérifier le contraire d'un prédicat, vous pouvez utiliser le NON logique, noté `not` en Python. Ce dernier diffère des deux précédents car il se place devant un prédicat un n'affecte *qu'un seul* prédicat.

Voici donc sa table de vérité :

A	NON A
`False`	`True`
`True`	`False`

```python
if not (couleur == "bleu" and poids > 20):
    # Si la condition est vérifiée, alors on ne peut avoir
        simultanément couleur == bleu et poids > 20
```

4.2.4. Un petit exercice

Souvenez-vous, nous avons parlé du OU exclusif. Pour raccourcir les notations, le OU désigne le ou inclusif, et XOR (de l'anglais « exclusive or ») le OU exclusif. Vous avez peut-être remarqué que je ne vous ai pas donné d'opérateur pour cette opération. C'est normal, Python ne définit pas cette opération par défaut. Rassurez-vous, on peut effectuer cette opération avec les trois opérations que je viens de vous présenter. Et bonne nouvelle, vous allez essayer de la faire vous-même. Si, si, ça sera un bon exercice très court.

Si vous ne savez pas par où commencer, je vous propose de subdiviser le problème : essayé d'écrire sa table de vérité dans un premier temps, puis de traduire cette table en langage naturel et enfin d'écrire le code associé. Oui, il y a une correction pour chaque étape, mais essayez de chercher avant de vous jeter dessus.

Contenu masqué n°1

Contenu masqué n°2

Contenu masqué n°3

Vous pouvez effectuer des tests afin de vérifier que nous suivons bien la table de vérité attendue.

Si vous souhaitez en apprendre plus sur les prédicats, vous pouvez regarder du côté de l'Algèbre de Boole.

4.3. Première boucle

Pour mieux illustrer l'utilité des boucles, nous allons chercher à calculer le *PGCD* de deux nombres. Pour rappel, le PGCD de deux nombres est le plus grand entier divisant simultanément ces deux entiers. Par exemple, le PGCD de 15 et 12 est 3 car $15 = 5 \times 3$ et $12 = 4 \times 3$. Pour le calculer, nous allons utiliser l'algorithme d'Euclide. Celui-ci repose sur le principe suivant :

— Le PGCD de `0` et d'un entier `a` est l'entier `a`
— Le PGCD de deux entiers `a` et `b` est égal au PGCD de `b` et de `a mod b`, c'est à dire $\text{PGCD}(a, b) = \text{PGCD}(b, a \bmod b)$

`mod` représente l'opération modulo de Python, c'est à dire `%`. Cette méthode fonctionne car `a mod b` est strictement plus petit que `a` et reste positif. Ainsi, en répétant la deuxième étape un certain nombre de fois, nous allons pouvoir nous ramener à la première étape et obtenir notre PGCD.

Facile ? Vous pouvez-essayer mais vous allez avoir un petit problème : combien de fois faut-il exécuter la deuxième étape ? Eh bien, autant de fois qu'il le faut. Le nombre d'étapes dépend de `a` et `b`, et puis si je vous avais dit 50 étapes, vous auriez fait 50 copier-coller de la deuxième étape ? Nous allons donc utiliser une boucle pour effectuer le travail.

Une boucle `while`, qui signifie « tant que » en anglais, permet d'exécuter un bloc d'instructions tant qu'un certain prédicat est vérifié. Voyons comment utiliser cette fameuse boucle en Python.

Figure 4.1. – Schéma illustrant la structure d'une boucle

4.3.1. Notre première boucle

Sans surprise, nous allons utiliser le mot-clé `while` :

```
1 while predicat:
2     # Instructions
```

Tout comme pour le `if`, les instructions indentées sont à *l'intérieur* de la boucle, c'est à dire que ce sont les instructions répétées tant que `predicat` est vrai. `predicat` peut être remplacé par n'importe quel prédicat vu précédemment.

Si votre prédicat est *toujours* vrai, alors votre boucle ne s'arrêtera **pas** et vous aurez alors une *boucle infinie*. Même s'il est vrai qu'une telle boucle peut être utile, méfiez-vous des boucles infinies indésirables. Python ne se charge pas de déterminer si vous avez

programmé ou non ce que vous vous vouliez, il exécute. Donc si vous lui demandez de faire une boucle infinie, il le fera.
Il vous arrivera sûrement de créer de telles boucles par mégarde, mais n'ayez crainte, si cela vous arrive, tapez Ctrl + C dans votre console. Cette combinaison, de manière générale, va interrompre brusquement l'exécution de Python. Vous pouvez alors corriger votre programme et relancer votre code.

Maintenant ~~que je vous ai fait peur~~ que vous êtes averti, essayez de réaliser une boucle de façon à ce qu'à sa sortie, c'est à dire quand elle se termine, `a` soit égal au PGCD que vous afficherez.

Contenu masqué n°4

Tout comme pour un `if`, vous pouvez mettre ce que vous voulez dans le bloc d'instructions de la boucle.

4.3.1.1. Deux mots-clés supplémentaires

Nous avons vu que le prédicat d'une boucle doit devenir faux à un moment pour que la boucle s'arrête. Il y a néanmoins une autre façon d'interrompre une boucle avec le mot-clé `break`. Quand Python rencontre cette instruction, il sort immédiatement de la boucle et continue l'exécution du code située après celle-ci. Par exemple, notre code pour trouver le PGCD peut s'écrire :

```
1 while True: # Cette boucle ne s'arrêtera pas sauf si ....
2     a, b = b, a%b
3     if b == 0:
4         break # ... on utilise un break pour la stopper
5     print(b) # Affiche l'évolution de b
6
7 print(a)
```

Cette méthode rend néanmoins plus difficile la lecture de longues boucles : on voit d'abord une boucle infinie, mais on ne sait pas à quelle condition celle-ci s'arrête. Il faut alors trouver le, ou les `break`. Remarquez que quand `b == 0`, l'instruction `print(b)` ne sera pas exécutée, car située après le `break`.

Vous serez tenté d'utiliser cette approche si votre prédicat est trop compliqué à exprimer. Néanmoins, si vous souhaitez éviter le comportement du `break`, qui ne va pas exécuter les instructions suivantes dans la boucle, vous pouvez utiliser une variable. Une telle variable est souvent appelée *drapeau* ou *flag* en anglais. Par exemple :

```
1 drapeau = True
2 while drapeau:
```

```
    a, b = b, a%b
    if b == 0:
        drapeau = False
    print(b)
```

Cette fois ci, l'instruction `print(b)` est exécutée même quand `b` vaut `0`.

Le deuxième mot-clé que je voulais évoquer est `continue`. Quand Python rencontre cette instruction, il recommence à la boucle à partir du prédicat, sans exécuter la fin du passage en cours. Un exemple pour éclaircir tout ça :

```
i = 0
while i < 10:
    i += 1 # Pour ne pas avoir une boucle infinie
    if i%2 == 0: # Si i est pair
        continue
    print(i)
```

Ici, seuls les valeurs impaires de `i` seront affichées. En effet, si `i` est pair, `continue` est exécuté et Python reprend le programme à partir de `while i < 10`. Notez que *l'incrémentation*, le fait d'augmenter de façon répétée la valeur de `i`, est placé au début. Si nous l'avions ajoutée après le `continue`, nous aurions eu une boucle infinie : l'incrémentation n'aurait plus lieu dès que `i` est pair, ce qui arrive dès le début avec `i = 0`.

Maintenant que nous commençons à pouvoir écrire des programmes un peu plus complexes, nous allons regarder comment mieux nous organiser.

Contenu masqué

Contenu masqué n°1

Seule la dernière ligne diffère de la table du OU :

A	B	A XOR B
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`False`

Retourner au texte.

Contenu masqué n°2

Pour la traduction en langage naturel, qui est probablement le plus dur, vous devez obtenir quelque chose *d'équivalent* à :

> Il faut que A soit vrai et pas B ou que B soit vrai et pas A pour que le prédicat soit vérifié

Retourner au texte.

Contenu masqué n°3

Je vais noter `A` et `B` deux variables de type `bool`. Elles peuvent bien sûr être remplacées par un prédicat.

```
(A and not(B)) or (B and not(A))
```

Retourner au texte.

Contenu masqué n°4

Je vous mets la correction, mais cherchez avant de sauter dessus :

```
a = int(input("a:"))
b = int(input("b:"))

while b != 0:
    a, b = b, a%b

print(a)
```

Retourner au texte.

5. Les fonctions

Vous commencez à écrire vos premiers programmes et ces derniers deviennent de plus en plus complexes. Vous rendre compte que vous voulez parfois effectuer plusieurs fois la même opération en différents endroits de votre code. Vous avez alors probablement été tenté de faire un copier-coller de ces parties redondantes.

Et là, malheur : vous vous rendez compte après plusieurs copier-coller que ce bout de code contient un bug. Il va alors falloir retrouver et corriger chaque utilisation de ce code dans tout votre code pour le rendre fonctionnel. Un vrai cauchemar en perspective. Mais voici venir la solution.

5.1. Les fonctions

Les fonctions sont des bouts de code d'un programme qui sont exécutés à chaque fois qu'ils sont appelés. Ainsi, par exemple, vous avez plusieurs fois besoin d'afficher une liste de choix dans votre console. Vous allez alors écrire ce code une première fois et le marquer comme fonction : on dit que l'on *définit* une fonction. Vous pourrez alors, à chaque fois que cela est nécessaire, *appeler* cette fonction. Ceci aura pour effet de déclencher le code présent dans ladite fonction et, dans notre cas, d'afficher notre superbe menu.

Que se passe-t-il si nous définissons plusieurs fonctions ? Comment les différencier ? Eh bien, on appelle une fonction par son *nom*. Ce dernier est défini au moment de la création de la fonction. Les noms des fonctions suivent les mêmes règles que les noms des variables. Il faut de plus éviter de nommer une variable et une fonction de la même façon. Tout comme pour les variables, donnez des noms clairs et explicites à vos fonctions. Par exemple, `affiche_menu` est un nom clair décrivant le rôle de cette fonction et facile à retenir comparer à un nom plus court donné à la va-vite tel que `f1`. Par convention, les noms de fonction sont écrits en minuscules et la séparation entre mots est délimité par un `_`.

Revenons sur les avantages de cette méthode : votre code est maintenant plus court, car l'affichage de votre menu est défini à un endroit unique au lieu de se répéter plusieurs fois. De plus, si vous voulez modifier les choix de votre menu ou corriger un bug, il vous suffit maintenant de modifier la *déclaration*, aussi appeler *définition*, de la fonction. Ce changement se répercutera alors automatiquement sur tous vos appels à cette fonction.

Il faut également noter que rien n'empêche une fonction d'en appeler une autre. Par exemple `affiche_menu` pourrait d'abord appeler une fonction `efface_console`. Une fonction peut même s'appeler elle-même : c'est la *récursivité*.

5.1.1. Les fonctions en Python

Abordons maintenant l'implémentation en Python de ce concept de fonction. Nous allons continuer sur notre exemple précédent et créer une fonction `affiche_menu`. La *définition* d'une fonction suit la structure suivante :

```
# Code hors fonction
def nom_fonction():
    # Code de la fonction
# Code hors fonction
```

Détaillons un peu tout ça. `def` est le mot clé indiquant à Python que l'on est en train de définir une fonction. `nom_fonction` est le nom de la fonction. C'est celui que nous utiliserons pour l'apeller. Viens ensuite le code de la fonction qui sera exécuté à chaque appel. Notez bien l'indentation du code dans la fonction. La réduction d'un niveau d'indentation délimite la fin de la fonction. C'est le même principe que pour les conditions ou les boucles.

Voyons maintenant ce que pourrais donner notre exemple de fonction :

```
def affiche_menu():
    print("Menu :")
    print("* Action 1")
    print("* Action 2")
    # Et ainsi de suite
```

Maintenant, à chaque fois que vous voulez afficher ce menu, il suffit d'utiliser :

```
affiche_menu()
```

Il est néanmoins très probable qu'après l'appel de cette fonction, vous utilisiez `input` pour récupérer le choix de l'utilisateur. Cette instruction pourrait faire partie de la fonction. Cependant, le traitement de la réponse ne devrait pas se trouver dans la fonction car il se peut qu'il ne soit pas partout le même. Il serait alors pratique de pouvoir renvoyer un résultat, un peu à la manière de `input` justement.

5.1.1.1. Renvoyer un résultat

Renvoyer ou *retourner* un résultat s'effectue à l'aide du mot-clé `return` suivit de ce que vous désirez renvoyer. Quelques exemples :

```
return True # Renverra toujours la même valeur.
return var # Où var est une variable définit précédemment. La valeur renvoyée peut alors être différente.
return input("Sélection: ") # On peut également renvoyer le résultat d'une autre fonction.
```

Notre exemple devient alors :

```
def affiche_menu():
    print("Menu :")
    print("* Action 1")
    print("* Action 2")

    return input("Choix: ") # Renvoie le choix de l'utilisateur
```

i

Il faut noter que `return` interrompt la fonction, c'est à dire que tout code se trouvant après un `return` ne sera pas exécuté.

```
return False
print("Non")  # ne sera pas exécuté
```

Cependant :

```
if "toto" == input():
    return False
print("Non")  # Sera exécuté si le texte tapé est différent de 'toto'
```

On obtient alors :

```
choix = affiche_menu()
```

5.1.1.2. Les paramètres

Vous commencez maintenant à avoir plusieurs fonctions mais certaines d'entre elles ont des comportements très semblables. Par exemple, supposons que nous avons une fonction `dire_bonjour` et une autre `dire_au_revoir`. Vous voulez maintenant créer une fonction `dire_une_blague`. Toutes ces fonctions se ressemblent au niveau de leur code et ne diffèrent donc que peu. Il semble alors que nous soyons à nouveau en train d'effectuer des copier-coller.

C'est alors qu'entrent en scène les paramètres. Ceux-ci permettent de *passer en argument* des informations variables à votre fonction pour que celle-ci puisse modifier son comportement. Voici un exemple pour y voir plus clair :

```
def dire(texte):
    print('# ' + texte)

dire('Bonjour') # Affiche `# Bonjour`
dire('Au revoir') # Affiche `# Au revoir`
```

Ainsi, nous pouvons nous débarrasser des fonctions `dire_bonjour` et `dire_au_revoir` et n'avoir plus qu'une seule fonction. On dit que `texte` est un paramètre. Une fois dans le corps de votre fonction `texte` peut être utilisé comme une variable ordinaire : vous pouvez lui appliquer des conditions, la modifier, la passer en paramètre à une autre fonction, ...

Vous pouvez avoir plusieurs paramètres pour une seule fonction. Il vous suffit de les séparer par des virgules dans la déclaration de votre variable :

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

addition(10, 5) # Renvoie 15
addition(10)
```

Le dernier appel renvoie l'erreur suivante :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: addition() missing 1 required positional argument: 'b'
```

En effet, vous *devez* passer à votre fonction le nombre de paramètre attendu par celle-ci. C'est pourquoi le deuxième appel a retourné une erreur : nous n'avons passé qu'un seul paramètre au lieu de deux. Dans le cas de notre fonction, cela semble logique car pour faire une addition, nous avons besoin de deux valeurs. Cependant, ce comportement peut être parfois gênant. Imaginons la fonction `saluer` qui prend en paramètre le nom de la personne à saluer. Il serait pratique de pouvoir l'appeler sans paramètre dans le cas où nous ne connaissons pas le nom de l'utilisateur. Nous allons voir comment réaliser une telle fonction.

5.1.1.3. Paramètres optionnels

Reprenons notre fonction :

```
def saluer(nom):
    print('Bonjour ' + nom)
```

Nous allons maintenant rendre le paramètre `nom` *optionnel*. Pour ce faire, on utilise la syntaxe `param = valeur` dans la définition de la fonction, où `valeur` est la *valeur par défaut* du paramètre. Un exemple :

```
def saluer(nom = 'Visiteur'):
    print('Bonjour ' + nom)

saluer('Clem') # Affiche `Bonjour Clem`
saluer() # Affiche `Bonjour Visiteur`
```

Vous pouvez ajouter plusieurs paramètres optionnels, et même combiner paramètres optionnels et obligatoires. La seule condition est que tous vos paramètres obligatoires doivent se trouver au début, et tous ceux facultatifs à la fin, par exemple `def fonction(a, b, c = 1)` et non `def fonction(a, c = 1, b)`. On appelle *signature* la combinaison formée du nom de la fonction et des paramètres, optionnels ou non, attendus par celle-ci.

5.2. La portée des variables

Vous avez peut-être essayé de faire passer des variables entre une fonction et votre programme sans utiliser les arguments ou `return`. Il se peut que certains comportements vous aient alors troublé. Nous allons essayer de percer ces mystères ensemble.

Commençons par un premier exemple :

```
a = 42

def affichage():
  print(a)

affichage() # Affiche 42
```

Il se peut que vous soyez surpris du fait que malgré l'absence de paramètre, `a` soit accessible dans la fonction `affichage`. Pour Python, nous avons déclaré une variable `a` puis notre fonction a cherché à afficher une variable nommée `a`. Python a alors utilisé la variable qu'il connaissait à défaut de la trouver une en paramètre.

Continuons un peu notre exploration :

```python
def modifier():
  print(a)
  a = 0

modifier()
```

Et là, c'est le drame :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in modifier
UnboundLocalError: local variable 'a' referenced before assignment
```

? Mais quel est cette erreur ? Que se passe-t-il ?

Décryptons tout d'abord ce message d'erreur. Le type `UnboundLocalError` peut se traduire par `Erreur Locale D'affectation`. Le message nous informe que la variable `a` a été utilisée avant d'être définie. Or la ligne d'avant affiche `a` sans problème. Un mot qui pourrait passer inaperçu mais qui va cependant être la clé de ce mystère est `local`.

Pour Python, le *contexte local* est l'intérieur de la fonction dans laquelle il se trouve. Il essaye alors de modifier une variable locale appelée `a` qui n'existe pas car celle-ci a été déclarée en dehors de la fonction. Pour l'affichage, Python utilise la variable définie à l'extérieur. Python applique le principe suivant : vous avez le droit de lire des variables qui ne sont pas locales mais vous n'avez pas le droit de les modifier.

On pourrait s'attendre à ce que Python crée une nouvelle variable lors de l'affection. C'est ce qui se produira si vous enlevez la ligne `print(a)`. En effet, `a` n'ayant pas été définie localement, la variable `a` extérieure a été importée pour pouvoir être lue, mais en lecture seule, empêchant toute modification.

Mais ce n'est pas tout :

```python
a = 42

def change(valeur):
    a = valeur

print(a) # Affiche `42`
change(10)
print(a) # Affiche ... `42` encore ? C'est universel ?
```

Tout ceci est normal et découle de ce qui a été dit au-dessus. Dans la fonction `change`, Python ne nous autorise pas à modifier une variable extérieure. Etant donné qu'il n'a pas été obligé

d'effectuer un import en lecture seule, le nom `a` est toujours libre. Il créé alors une variable *locale* `a` ayant pour valeur le paramètre de la fonction. J'insiste sur le *locale* : en effet, à la sortie de la fonction, le *contexte local* est détruit. Ainsi notre variable locale `a = 10` a été détruite. On aurait néanmoins pu la récupérer à l'aide d'un `return`.

5.2.1. Les variables globales

Il se peut que vous souhaitiez passer outre ce comportement de Python et modifier la valeur d'une variable extérieure. Pour ce faire, nous allons utiliser le mot-clé `global`.

```
a = 42

def change(valeur):
    global a
    a = valeur

print(a) # Affiche `42`
change(10)
print(a) # Affiche `10`
```

Vous devez utiliser ce mot-clé avant toute référence à `a` dans la fonction. Quand Python rencontre l'instruction `global a`, il va chercher dans le contexte extérieur une variable `a` puis va l'importer dans le contexte local tout en *autorisant sa modification*.

Il faut néanmoins noter qu'il ne s'agit pas d'un remplacement de la combinaison *paramètre/return*. Les variables globales sont un bon choix dans certains cas très particuliers, mais si vous les utilisez trop sans vraiment les maitriser, vous risquez de vous retrouver avec des comportements inexpliqués. Évitez donc de les utiliser.

5.3. Les fonctions anonymes

Nous allons aborder une autre façon de définir une fonction. La syntaxe `def` est vraisemblablement celle que vous utiliserez le plus mais, dans certains cas, elle peut être un peu longue et lourde. À cet effet, le mot-clé `lambda` permet une définition plus courte mais avec un comportement différent.

La première particularité est que ces fonctions ne seront pas nommées, donc *anonymes*. Voyons un premier exemple :

```
fonction = lambda a, b: a**b
print(fonction(2,3))  # Affiche 8
```

Commençons par la première ligne. On définit notre fonction anonyme à l'aide de `lambda`. Les paramètres se situent entre le mot-clé et `:`, séparés par des virgules, comme pour les fonctions que nous connaissons. Vous pouvez même ajouter une valeur par défaut aux paramètres tout comme pour les fonctions définies avec `def`.

Remarquez que nous n'avons pas besoin d'utiliser `return` pour indiquer que nous souhaitons renvoyer un résultat. Le résultat de la dernière instruction de la fonction sera automatiquement renvoyé. Vous remarquerez que les instructions possibles dans ce type de fonction sont limitées. Si vous vous retrouvez bloqué par ces limitations, c'est probablement qu'un `def` serait préférable, même si il existe parfois un moyen de passer outre cette limitation.

Malgré l'absence de nom, nous parvenons à utiliser cette fonction car nous l'avons stockée dans une variable. Par exemple, vous pouvez notamment passer cette fonction en paramètre à une autre. Voyons un exemple, certes simple, mais qui vous aidera peut-être à mieux visualiser l'utilité de ce type de fonction.

```
1  def test(f, a, b=None):
2      if b != None:
3          r = f(a, b)
4      else:
5          r = f(a)
6
7      if r:
8          print("Test passé avec succès :)")
9      else:
10         print("Echec du test :(")
11
12 pair = lambda a: a % 2 == 0
13 divise = lambda a, b: a % b == 0
14
15 test(pair, 6)
16 test(divise, 6, 3)
```

Vous pouvez également passer en paramètre des fonctions définies à l'aide de `def`, ce n'est donc pas une particularité des fonctions anonymes. Celles-ci sont souvent utilisées pour représenter des fonctions mathématiques, mais ce n'est qu'un exemple.

Vous savez maintenant comment mieux organiser votre code grâce aux fonctions. Nous allons continuer notre organisation avec la notion de modules.

6. Découper son code

Nous allons aborder un niveau d'organisation à plus grande échelle.

6.1. Plusieurs fichiers

Au risque de casser l'ambiance, je vous annonce tout de suite qu'un *module* n'est rien d'autre qu'un *fichier*. Nous allons donc voir comment écrire notre programme dans plusieurs fichiers. En plus de fournir un nouveau moyen d'organisation, que vous apprécierez largement quand vos programmes grossiront, vous pourrez mettre en commun certaines de vos fonctions entre plusieurs programmes en les plaçant dans un module commun.

Vous savez créer un fichier, vous savez donc en créer plusieurs. Nous allons nous intéresser à l'utilisation de ces modules. En effet, malgré ses qualités, Python n'est pas devin : il va falloir lui dire où se cachent nos modules et lesquels nous souhaitons utiliser.

Créez deux fichiers, par exemple `programme.py` et `commun.py`, dans un même répertoire.

N'oubliez pas de spécifier l'encodage de **tous** vos fichiers, y compris ceux qui sont partagés, comme `commun.py` dans cet exemple.

Souvenez-vous du OU exclusif que nous avions codé dans le chapitre sur les conditions. Vous décidez (même si vous ne le savez pas encore) de le placer dans une fonction, appelée `xor` par exemple, dans le fichier `commun.py` afin de pouvoir la réutiliser rapidement dans vos divers programmes.

Notre programme se situera dans `programme.py` dans cet exemple. Voici donc comment utiliser notre fonction `xor`. Essayez donc d'écrire tout seul le code de `commun.py`.

 Contenu masqué n°5

```python
# programme.py
import commun

print(commun.xor(True, 2 != 2))
```

Nous avons commencé par indiquer à Python qu'il devait charger le module `commun`. De façon générale, `import nom` permet de charger le fichier `nom.py` situé dans le même dossier que

votre programme. Pour importer plusieurs modules, vous pouvez, soit écrire plusieurs `import`, soit séparer le nom des modules par une virgule. Prenez l'habitude de placer vos `import` au début de vos fichiers, mais en dessous de l'encodage tout de même. Nous avons ensuite utilisé `commun.xor` au lieu de `xor`. Nous avons dit à Python d'aller chercher la fonction `xor` dans le module `commun`.

Cette notation peut vous sembler lourde, mais elle est bien utile. Supposez que vous récupériez un module sur internet pour réaliser une certaine tâche. Si ce module défini une fonction portant le même nom que l'une des vôtres, Python va écraser la première fonction du même nom qu'il a rencontrée par la deuxième. Cela est problématique si les deux fonctions ne sont pas identiques dans leurs paramètres et leurs valeurs renvoyées, ce qui est généralement le cas. L'utilisation du nom de module comme *préfixe* du nom des fonctions permet à Python de savoir quelle fonction nous souhaitons utiliser.

Rassurez-vous, il existe des moyens de faire plus court, mais vous devez toujours faire attention à ce que plusieurs fonctions portant le même nom n'entrent pas en conflit.

```
1 import commun as com
2 print(com.xor(True, 2 != 2))
```

L'ajout de `as` crée un *alias*, c'est à dire que le module `commun` est maintenant connu sous le nom de `com`. Il existe également un moyen d'appeler une fonction sans spécifier son module :

```
1 from commun import xor
2 print(xor(True, 2 != 2))
```

La syntaxe générale est, comme vous l'avez peut-être deviné, `from module import fonction1, fonction2`. Vous n'importez alors que les fonctions spécifiées. Si vous souhaitez tout importer, vous pouvez utiliser `from module import *`.

Pour des modules de taille conséquente, il n'est pas recommandé de tout importer, surtout si vous n'utilisez que quelques fonctions, ce qui est le cas dans la grande majorité des cas. En effet, vous allez vous retrouver avec un nombre conséquent de noms utilisés par ce module, et vous aurez du mal à déterminer rapidement si un nom donné n'est pas déjà pris.

Vous pouvez également combiner cette dernière syntaxe avec `as` afin de changer le nom des fonctions importées, ce qui peut permettre d'éviter un conflit de nommage. Par exemple `from module import fonction1 as f1, fonction2 as f2` importera les deux fonctions sous le nom de `f1` et `f2`. Il faudra donc utiliser le nom avec lequel la fonction a été *importée*, par exemple `f1`, et non le nom avec lequel elle a été définie, dans ce cas `fonction1`.

Vous pouvez également importer des variables depuis un module. Pour pouvoir importer une telle variable, il vous suffit de la définir en dehors de toute fonction dans votre module. En ce qui concerne son utilisation, nous allons nous intéresser au module `math` de Python qui définit

une variable `pi`. Nous pouvons l'importer de manière similaire aux fonctions. Le commentaire correspond à la façon d'accéder à `pi` pour chaque `import`.

```
1 import math # print(math.pi)
2 import math as m # print(m.pi)
3 from math import pi # print(pi)
4 from math import * # print(pi)
```

Notez que la dernière ligne n'importe pas seulement la variable `pi`, mais l'intégralité du module.

Il se peut que vous souhaitiez écrire un fichier qui serait un programme à lui seul, mais que vous souhaitiez pouvoir importer les fonctions de ce dernier sans lancer l'exécution du programme. Pour ce faire, nous allons utiliser une variable spéciale de Python :

```
1 def fonction(a):
2     return a + 1
3
4 if __name__ == "__main__":
5     a = 1
6     while a < 5:
7         a = fonction(a)
```

La variable `__name__` contient le nom du module courant, c'est à dire le nom du module de ce fichier. Néanmoins, `__name__` peut prendre une valeur spéciale, `__main__`, qui signifie « principal », pour indiquer que ce module a été exécuté depuis la console. Ainsi dans notre exemple, la boucle `while` ne sera prise en compte que si le fichier est exécuté et non importer.

De façon générale, les variables ou fonctions de la forme `__abc__` correspondent à des fonctions ou variables ayant une signification spéciale pour Python.

6.2. Des répertoires

Nous allons maintenant encore étendre notre organisation en conquérant les répertoires. Bien entendu, on va leur donner un petit nom, celui de *package*, soit « paquet » en français. Tout comme un répertoire peut contenir plusieurs fichiers et plusieurs sous-dossiers, un package peut contenir plusieurs modules ainsi que d'autres packages.

Il s'agit d'un niveau de découpage encore plus grand que celui des modules. Nous allons illustrer ceci par un exemple. Imaginez que vous soyez en train de programmer un logiciel de gestion de la maison du futur ~~pas si lointain~~. Vous seriez par exemple amené à écrire un programme `gestion_courses`. Celui-ci utiliserait les modules `frigo` et `placard` afin de savoir ce qu'il vous manque. Continuons avec l'arrosage. Vous auriez alors un programme `arrosage_automatique` qui utiliserait le module `arrosage`. Et ainsi de suite. Nous allons nous retrouver sous

une montagne de modules et de programmes mélangés, même s'ils visent le but commun de gérer une maison. On pourrait alors utiliser des packages. La structure suivante représente une organisation possible de votre répertoire de travail :

- `arrosage_automatique.py`, `gestion_courses.py`, nos programmes
- `exterieur/`, un grand package
  - `jardin/`, un premier sous-package
    - `arrosage.py`, `piscine.py`, et bien d'autres modules
- `interieur/`, un autre grand package avec plusieurs sous-packages
  - `cuisine/`
    - `frigo.py`, `placard.py`, ...
  - `salon/`
    - `tv.py`, ...

Les `/` marquent les dossiers, et donc les packages, le `.py` les modules. Le nom du package est le nom du répertoire, sans `/`. Remarquez que, tout comme pour les fichiers, préférez des noms évocateurs et n'utilisez ni d'espaces ni de caractère spéciaux, accents compris.

Je dois vous informer ~~d'un petit mensonge~~ d'une omission de ma part : un package est en réalité un dossier comportant un fichier `__init__.py`. Nous nous contenterons d'un fichier vide, mais sachez que ce fichier peut contenir du code qui sera exécuté à l'initialisation du package. Sans ce fichier, Python ne reconnaitra pas votre dossier comme un package.

Ceci est d'autant plus vrai si vous utilisez une version de Python inférieure à la `3.3`. Les versions supérieures ne requièrent plus ce fichier, mais vous pouvez continuer à le mettre, même s'il reste vide, afin d'assurer une plus grande compatibilité.

6.2.1. Importer ses packages

Supposons que nous avons placé notre fichier `commun.py` dans un dossier nommé `conditions`. Je vous invite d'ailleurs à le faire pour tester les exemples suivants. Pour importer un module se trouvant dans un package, il faut utiliser `import package.module`. Les différentes syntaxes vues au-dessus sont également applicables dans ce contexte. En reprenant notre exemple, voici ce que cela donne, avec en commentaire l'appel associé :

```
1 import condition.commun # condition.commun.xor(a, b)
2 import condition.commun as com # com.xor(a, b)
3 from condition.commun import xor # xor(a, b)
```

Si vous souhaitez importer un sous-package, il faut alors séparer les noms des packages par un `.` et suivre la hiérarchie de vos dossiers, par exemple `from interieur.cuisine import frigo`.

6.2.1.1. Les importations relatives

Vos modules peuvent avoir besoin d'importer d'autres modules, se situant ou non dans le même package. Supposons que vous souhaitiez importer le module `placard` depuis le module `frigo`. On utilisera alors :

```
1 from . import placard # ex: placard.ingredients()
2 from .placard import ingredients # ex: ingredients()
```

Le `.` placé au début indique à Python qu'il doit chercher le module en partant du package *dans lequel il se trouve* et non à partir de l'emplacement de votre programme. En effet, votre programme, devra utiliser `from interieur.cuisine.placard import ingredients`, ce qui n'est pas le cas de `frigo` qui se trouve dans le même package. Vous pouvez appeler des sous-packages relativement, en séparant les différents niveaux par un point comme précédemment.

Essayons maintenant d'importer un package qui ne se situe pas sous le module actuel. Par exemple, essayons d'importer le module `tv` depuis `frigo` :

```
1 from ..salon.tv import volume # ex: volume()
```

Cette fois ci, le `..` au début informe Python qu'il doit chercher le module en se positionnant d'abord dans le package *au-dessus* de celui dans lequel il est actuellement. Dans notre exemple, les `..` font référence au package `interieur`. Si vous souhaitez remonter de plus d'un niveau, il faut alors rajouter un point supplémentaire par niveau à remonter. Par exemple, pour remonter de trois niveaux, utilisez `from ...`.

Vous ne pouvez néanmoins pas remonter d'un niveau si le dossier parent n'est pas lui-même un package. Par exemple, vous ne pouvez pas utiliser la syntaxe relative pour accéder au package `exterieur` depuis le package `interieur`. Vous devrez alors utiliser `from exterieur.jardin.piscine import temperature`. En effet, les packages `exterieur` et `interieur` ne font pas partie d'un plus grand package commun. Naturellement, si ces deux derniers faisaient partie d'un plus grand package, `propriete` par exemple, la syntaxe relative pourrait être utilisée. La syntaxe *absolue*, avec le chemin en entier, est toujours disponible : `frigo` pourrait importer `placard` comme suit : `from interieur.cuisine.placard import ingredients`.

Il est maintenant temps de vérifier vos connaissances avant de s'avancer plus profondément.

Contenu masqué

Contenu masqué n°5

```python
# commun.py
# N'oubliez pas l'encodage
def xor(a, b):
    return (a and not(b)) or (b and not(a))
```

Retourner au texte.

7. TP : Premier du nom

Vous ~~le craigniez~~ en rêviez, le voici : votre premier TP. Un entrainement opportun pour tester ce que nous avons appris jusqu'ici.

7.1. Enoncé

Pour notre premier TP, nous allons coder un convertisseur d'angle. Pour rappel, un angle peut, par exemple, être exprimé en *degrés* ou en *radians*. Nous ne considérons que ces deux unités dans un premier temps.

Notre convertisseur devra comporter un menu pour permettre à l'utilisateur de choisir l'action à réaliser. Par exemple :

```
Options disponibles :
1 : Convertir des degrés en radians
2 : Convertir des radians en degrés
q : Quitter

Votre choix :
```

Suite au choix de l'utilisateur, nous devrons alors, soit quitter, soit lui demander la valeur qu'il souhaite convertir. Nous utiliserons les formules suivantes pour les conversions :

$$\theta_{degrs} = \theta_{radians} \times \frac{180}{\pi}$$
$$\theta_{radians} = \theta_{degrs} \times \frac{\pi}{180}$$

> *i* Si vous ne savez pas ce qu'est un angle exprimé en radian, ne vous inquiétez pas, vous n'aurez pas besoin de le savoir pour la suite du TP, comprenez simplement qu'il s'agit d'une unité de mesure d'un angle, de la même façon que le mètre et le millimètre sont des unités de mesure d'une longueur. Pour passer d'une unité à une autre, nous utiliserons les formules données ci-dessus.

Nous afficherons ensuite le résultat de la conversion. Nous demanderons à l'utilisateur s'il souhaite quitter le programme. Si oui, on coupe tout, si non, on affiche de nouveau le menu et ainsi de suite. Cette question aura aussi l'avantage de ne pas afficher une fois de plus le menu tout de suite après la conversion, afin de mieux distinguer le résultat.

Voilà pour les consignes. Je vous recommande d'essayer de distinguer les portions de code qui peuvent former une fonction et de les séparer.

Maintenant, un petit point technique que nous verrons plus en détail bientôt, mais qui nous sera utile dans ce TP. Pour placer une variable à la suite d'une chaîne de caractère, notamment dans un `print`, on utilise le `+` :

```
1 nom = "Clem"
2 print("Bonjour " + nom)
```

La variable **doit** être une chaîne de caractères. S'il s'agit d'un nombre, il faudra la convertir, sous peine d'erreur.

C'est à vous de jouer maintenant. Prenez le temps qu'il faut, faites-le en plusieurs fois si besoin, écrivez un petit brouillon sur papier du fonctionnement (ça peut vous aider, ne riez pas). Je vais, pour ma part, coder une proposition de correction dans mon coin. Ne copiez pas !

7.2. Correction

Voici une proposition de correction, à regarder une fois que vous avez cherché bien sûr. Ce n'est pas parce que vous n'avez pas le même code que celui proposé ci-dessous que le vôtre est faux ; il existe plusieurs façons de procéder. Le code proposé est commenté afin de « parler de lui-même ».

Contenu masqué n°6

Je me répète, mais ce n'est pas parce que votre code souffre de quelques problèmes, ou se présente différemment, que vous devez tout effacer pour utiliser cette correction. Au contraire, gardez votre code et améliorez-le. Vous comprendrez mieux vos erreurs en les corrigeant qu'en utilisant un autre code tout prêt. Vous aurez ensuite le plaisir de faire évoluer et présenter fièrement votre création.

D'ailleurs, voici quelques idées de modifications, mais n'hésitez pas à apporter les vôtres en plus :

- Placer les fonctions de conversion dans un module séparé
- Ajouter la conversion en grades (je suis généreux, $\theta_{grades} = \theta_{radians} \times \frac{200}{\pi}$)
- Ajouter un symbole après l'affichage du résultat reflétant l'unité
- Faire un convertisseur d'unité de mesure anglaises/internationales (et ainsi sauvez de pauvres sondes spatiales)

Maintenant que les bases sont posées, nous allons nous intéresser à un concept tenant une place importante en Python : la POO.

Maintenant que les bases sont posées, passons à une utilisation un peu plus riche de Python.

Ce cours n'est actuellement pas terminé.

Ce cours avait pour but de vous initier et de vous donner les principales bases de Python. Et avec un peu de chance, il y est parvenu. Mais le monde de Python, et celui de la programmation en général, est vaste.

Vous pouvez chercher d'autres cours pour approfondir certaines notions, visiter des forums ... Mais surtout : faites des programmes. D'abord simples, vous verrez que vous commencerez à faire des progrès, aussi bien au niveau de votre maitrise de Python, que dans votre façon de penser vos programmes.

Puissiez-vous longtemps arpenter la voix de la programmation.

Remerciements :

— Dr@zielux, Kje et Smokiev pour leur relecture attentive
— Karnaj pour son exemple de fonction anonyme
— Vayel pour son aide

Contenu masqué

Contenu masqué n°6

```python
# N'oubliez pas l'encodage
from math import pi

def deg_en_rad(deg):
    # Converti des degrés en radians
    return deg * pi / 180

def rad_en_deg(rad):
    # Converti des radians en degrés
    return rad * 180 / pi

def menu():
    # Affiche le menu et renvoie le choix de l'utilisateur
    print("\nOptions disponibles : ") # On saute une ligne au début
    print("1 : Convertir des degrés en radians")
    print("2 : Convertir des radians en degrés")
    print("q : Quitter\n") # On saute une ligne

```

```
19      return input("Votre choix : ")
20
21  def demande_continuer():
22      # On demande si l'utilisateur veut quitter avant de réafficher
            le menu
23      reponse = input("Effectuer une autre conversion ? (O/N) :\n ")
24
25      if reponse == "N":
26          return False
27
28      return True
29
30  # On définit une variable qui fait office de drapeau
31  continuer = True
32
33  print("Super Convertisseur Degré-Radian Réversible")
34
35  # Notre boucle principale permet de proposer
36  # et d'effectuer plusieurs conversions à la suite
37  while continuer:
38      # On affiche le menu et on récupère le choix de l'utilisateur
39      choix = menu()
40
41      # On traite le choix de l'utilisateur en se souvenant
42      # qu'il s'agit d'une chaine de caractères
43      if choix == "1":
44          # On récupère et converti l'angle donné par l'utilisateur
45          deg = float(input("Mesure en degré : "))
46          # On affiche le résultat en pensant à convertir en chaine
                de caractères
47          print("Mesure en radian : " + str(deg_en_rad(deg)))
48          # On demande si l'on souhaite continuer (cela permet aussi
                de mieux
49          # voir le résultat avant d'être submergé par le menu)
50          continuer = demande_continuer()
51      elif choix == "2":
52          # De même dans l'autre sens de conversion
53          rad = float(input("Mesure en radian : "))
54          print("Mesure en degré : " + str(rad_en_deg(rad)))
55          continuer = demande_continuer()
56      elif choix == "q":
57          # On veut quitter
58          continuer = False
59      else:
60          print("Option inconnue")
61
62  print("\nCe convertisseur vous est juteusement offert par la Clem's Foundation"
```

Retourner au texte.

Liste des abréviations

EDI Environnement de Développement Intégré. 11

exponentiation élévation à la puissance. 10

IDE Integrated Development Environment. 11

PGCD Plus Grand Commun Diviseur. 27, 28

POO Programmation Orientée Objet. 46

TP Travaux Pratiques. 2, 45, 46