Partie 1

Qu'est-ce que Python ?

Vous avez décidé d'apprendre le Python et je ne peux que vous en féliciter. J'essayerai d'anticiper vos questions et de ne laisser personne en arrière.

Dans ce chapitre, je vais d'abord vous expliquer ce qu'est un langage de programmation. Nous verrons ensuite brièvement l'histoire de Python, afin que vous sachiez au moins d'où vient ce langage ! Ce chapitre est théorique mais je vous conseille vivement de le lire quand même.

La dernière section portera sur l'installation de Python, une étape essentielle pour continuer ce tutoriel. Que vous travailliez avec Windows, Linux ou Mac OS X, vous y trouverez des explications précises sur l'installation.

Allez, on attaque !

Un langage de programmation ? Qu'est-ce que c'est ?

La communication humaine

Non, ceci n'est pas une explication biologique ou philosophique, ne partez pas !
Très simplement, si vous arrivez à comprendre ces suites de symboles étranges et déconcertants que sont les lettres de l'alphabet, c'est parce que nous respectons certaines conventions, dans le langage et dans l'écriture. En français, il y a des règles de grammaire et d'orthographe, je ne vous apprends rien. Vous communiquez en connaissant plus ou moins consciemment ces règles et en les appliquant plus ou moins bien, selon les cas.
Cependant, ces règles peuvent être aisément contournées : personne ne peut prétendre connaître l'ensemble des règles de la grammaire et de l'orthographe françaises, et peu de gens s'en soucient. Après tout, même si vous faites des fautes, les personnes avec qui vous communiquez pourront facilement vous comprendre.
Quand on communique avec un ordinateur, cependant, c'est très différent.

Mon ordinateur communique aussi !

Eh oui, votre ordinateur communique sans cesse avec vous et vous communiquez sans cesse avec lui. D'accord, il vous dit très rarement qu'il a faim, que l'été s'annonce caniculaire et que le dernier disque de ce groupe très connu était à pleurer.
Il n'y a rien de magique si, quand vous cliquez sur la petite croix en haut à droite de l'application en cours, celle-ci comprend qu'elle doit se fermer.

Le langage machine

En fait, votre ordinateur se fonde aussi sur un langage pour communiquer avec vous ou avec lui-même. Les opérations qu'un ordinateur peut effectuer à la base sont des plus classiques et constituées de l'addition de deux nombres, leur soustraction, leur multiplication, leur division, entière ou non. Et pourtant, ces cinq opérations suffisent amplement à faire fonctionner les logiciels de simulation les plus complexes ou les jeux super-réalistes.
Tous ces logiciels fonctionnent en gros de la même façon :

- une suite d'instructions écrites en langage machine compose le programme ;
- lors de l'exécution du programme, ces instructions décrivent à l'ordinateur ce qu'il faut faire (l'ordinateur ne peut pas le deviner).

Une liste d'instructions ? Qu'est-ce que c'est encore que cela ?

En schématisant volontairement, une instruction pourrait demander au programme de se fermer si vous cliquez sur la croix en haut à droite de votre écran, ou de rester en tâche de fond si tel est son bon plaisir. Toutefois, en langage machine, une telle action demande à elle seule un nombre assez important d'instructions.
Mais bon, vous pouvez vous en douter, parler avec l'ordinateur en langage machine, qui ne comprend que le binaire, ce n'est ni très enrichissant, ni très pratique, et en tous cas pas très marrant.
On a donc inventé des langages de programmation pour faciliter la communication avec l'ordinateur.

Le langage binaire est uniquement constitué de 0 et de 1. « 01000010011011110110111001101010011011110111010101110010 », par exemple, signifie « Bonjour ». Bref, autant vous dire que discuter en binaire avec un ordinateur peut être long (surtout pour vous).

Les langages de programmation

Les langages de programmation sont des langages bien plus faciles à comprendre pour nous, pauvres êtres humains que nous sommes. Le mécanisme reste le même, mais le langage est bien plus compréhensible. Au lieu d'écrire les instructions dans une suite assez peu intelligible de 0 et de 1, les ordres donnés à l'ordinateur sont écrits dans un « langage », souvent en anglais, avec une syntaxe particulière qu'il est nécessaire de respecter. Mais avant que l'ordinateur puisse comprendre ce langage, celui-ci doit être traduit en langage machine (figure suivante).

En gros, le programmeur « n'a qu'à » écrire des **lignes de code** dans le langage qu'il a choisi, les étapes suivantes sont automatisées pour permettre à l'ordinateur de les décoder.

Il existe un grand nombre de langages de programmation et Python en fait partie. Il n'est pas nécessaire pour le moment de donner plus d'explications sur ces mécanismes très schématisés. Si vous n'avez pas réussi à comprendre les mots de vocabulaire et l'ensemble de ces explications, cela ne vous pénalisera pas pour la suite. Mais je trouvais intéressant de donner ces précisions quant aux façons de communiquer avec son ordinateur.

Pour la petite histoire

Python est un langage de programmation, dont la première version est sortie en 1991. Créé par **Guido van Rossum**, il a voyagé du Macintosh de son créateur, qui travaillait à cette époque au *Centrum voor Wiskunde en Informatica* aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la **Python Software Foundation**, créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les « Monty Python ».

À quoi peut servir Python ?

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir tout au long de ce livre.

Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

- de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie…
- des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- créer des interfaces graphiques ;
- faire circuler des informations au travers d'un réseau ;
- dialoguer d'une façon avancée avec votre système d'exploitation ;
- … et j'en passe…

Bien entendu, vous n'allez pas apprendre à faire tout cela en quelques minutes. Mais ce cours vous donnera des bases suffisamment larges pour développer des projets qui pourront devenir, par la suite, assez importants.

Un langage de programmation interprété

Eh oui, vous allez devoir patienter encore un peu car il me reste deux ou trois choses à vous expliquer, et je suis persuadé qu'il est important de connaître un minimum ces détails qui peuvent sembler peu pratiques de prime abord.
Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

Différentes versions de Python

Lors de la création de la Python Software Foundation, en 2001, et durant les années qui ont suivi, le langage Python est passé par une suite de versions que l'on a englobées dans l'appellation Python 2.x (2.3, 2.5, 2.6…). Depuis le 13 février 2009, la version 3.0.1 est disponible. Cette version casse la **compatibilité ascendante** qui prévalait lors des dernières versions.

Compatibilité quoi ?

Quand un langage de programmation est mis à jour, les développeurs se gardent bien de supprimer ou de trop modifier d'anciennes fonctionnalités. L'intérêt est qu'un programme qui fonctionne sous une certaine version marchera toujours avec la nouvelle version en date. Cependant, la Python Software Foundation, observant un bon nombre de fonctionnalités obsolètes, mises en œuvre plusieurs fois… a décidé de nettoyer tout le projet. Un programme qui tourne à la perfection sous Python 2.x devra donc être mis à jour un minimum pour fonctionner de nouveau sous Python 3. C'est pourquoi je vais vous conseiller ultérieurement de télécharger et d'installer la dernière version en date de Python. Je m'attarderai en effet sur les fonctionnalités de Python 3 et certaines d'entre elles ne seront pas accessibles (ou pas sous le même nom) dans les anciennes versions.

Ceci étant posé, tous à l'installation !

Installer Python

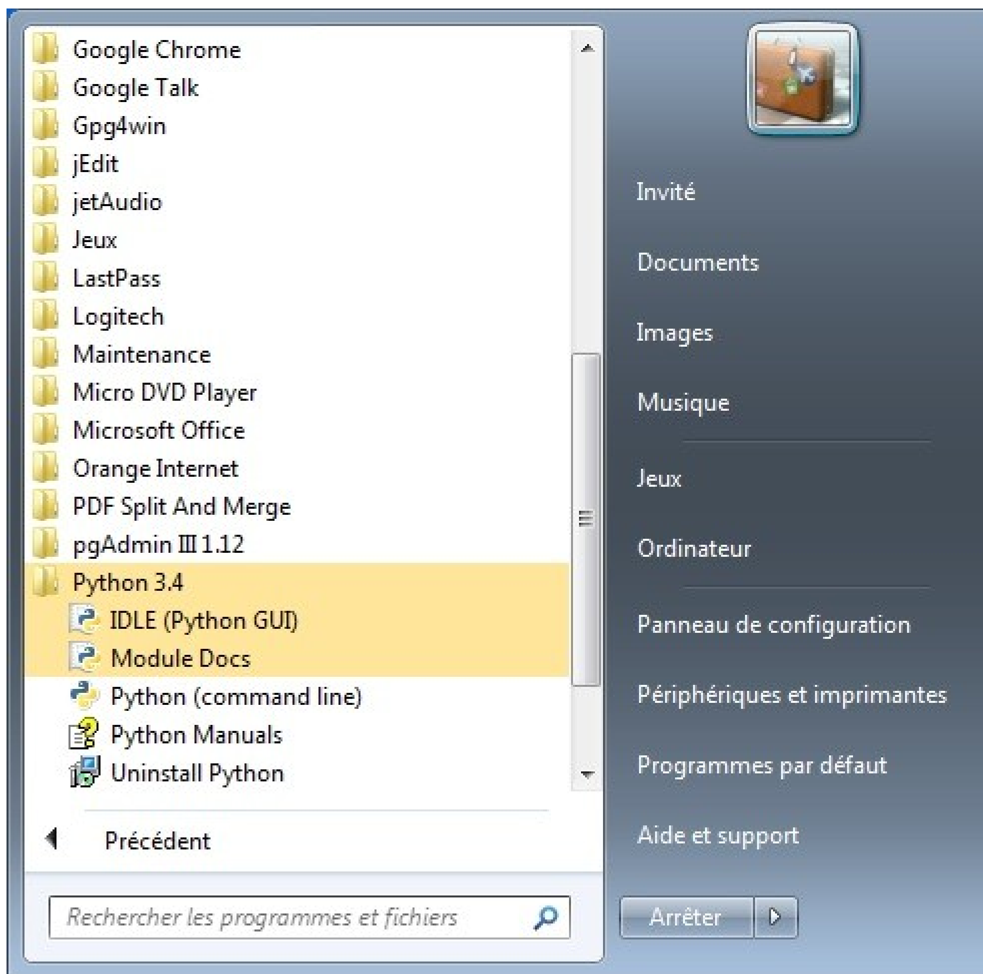
L'installation de Python est un jeu d'enfant, aussi bien sous Windows que sous les systèmes Unix. Quel que soit votre système d'exploitation, vous devez vous rendre sur le site officiel de Python.

Sous Windows

1. Cliquez sur le lien `Download` dans le menu principal de la page.
2. Sélectionnez la version de Python que vous souhaitez utiliser (je vous conseille la dernière en date).
3. On vous propose un (ou plusieurs) lien(s) vers une version Windows : sélectionnez celle qui conviendra à votre processeur. Si vous avez un doute, téléchargez une version « x86 ».

Si votre ordinateur vous signale qu'il ne peut exécuter le programme, essayez une autre version de Python.

1. Enregistrez puis exécutez le fichier d'installation et suivez les étapes. Ce n'est ni très long ni très difficile.
2. Une fois l'installation terminée, vous pouvez vous rendre dans le menu `Démarrer` > `Tous les programmes`. Python devrait apparaître dans cette liste (figure suivante). Nous verrons bientôt comment le lancer, pas d'impatience…



Sous Linux

Python est pré-installé sur la plupart des distributions Linux. Cependant, il est possible que vous n'ayez pas la dernière version en date. Pour le vérifier, tapez dans un terminal la commande `python -V`. Cette commande vous renvoie la version de Python actuellement installée sur votre système. Il est très probable que ce soit une version **2.x**, comme 2.6 ou 2.7, pour des raisons de compatibilité. Dans tous les cas, je vous conseille d'installer Python 3.x, la syntaxe est très proche de Python 2.x mais diffère quand même…

Cliquez sur `download` et téléchargez la dernière version de Python (actuellement « Python 3.4 gzipped source tarball »). Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive :

1. Décompressez l'archive en tapant : `tar -xzf Python-3.4.0.tar.bz2` (cette commande est bien entendu à adapter suivant la version et le type de compression).
2. Attendez quelques instants que la décompression se termine, puis rendez-vous dans le dossier qui vient d'être créé dans le répertoire courant (`Python-3.4.0` dans mon cas).
3. Exécutez le script `configure` en tapant `./configure` dans la console.

4. Une fois que la configuration s'est déroulée, il n'y a plus qu'à compiler en tapant `make` puis `make install` en tant que super-utilisateur.

Sous Mac OS X

Téléchargez la dernière version de Python. Ouvrez le fichier `.dmg` et faites un double-clic sur le paquet d'installation `Python.mpkg`

Un assistant d'installation s'ouvre, laissez-vous guider : Python est maintenant installé !

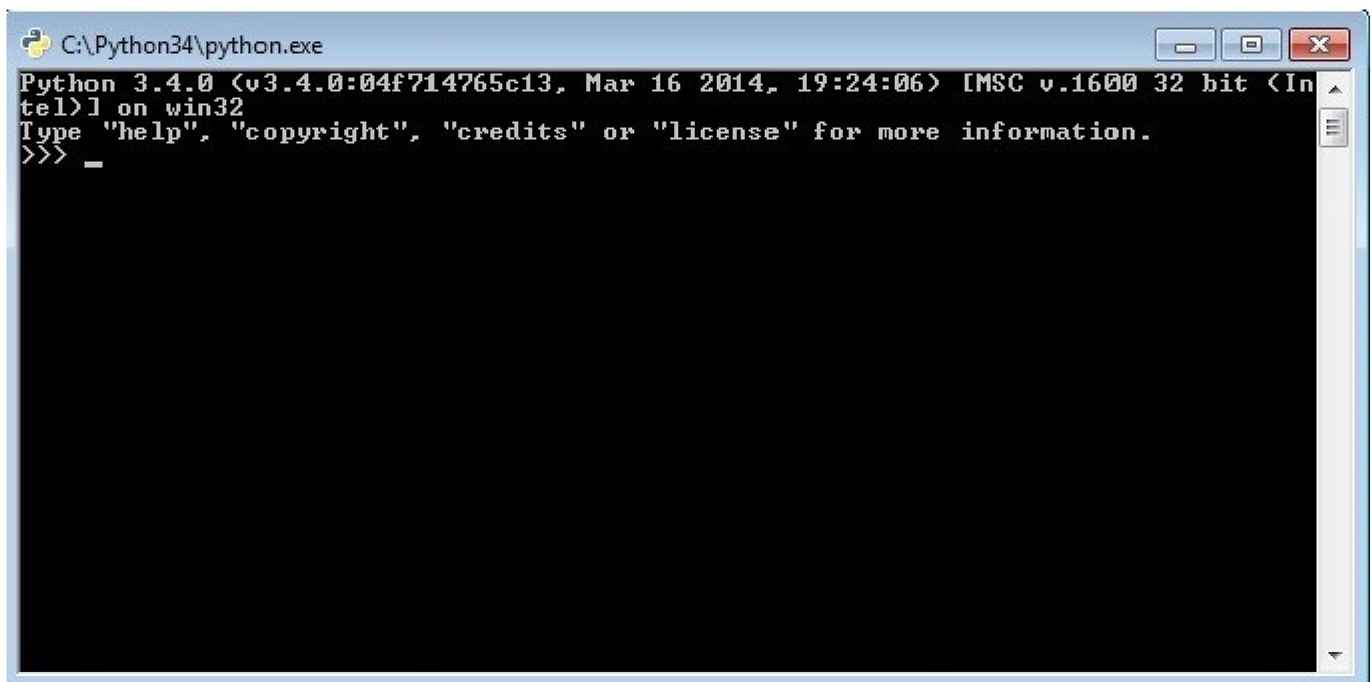
Lancer Python

Ouf ! Voilà qui est fait !

Bon, en théorie, on commence à utiliser Python dès le prochain chapitre mais, pour que vous soyez un peu récompensés de votre installation exemplaire, voici les différents moyens d'accéder à la ligne de commande Python que nous allons tout particulièrement étudier dans les prochains chapitres.

Sous Windows

Vous avez plusieurs façons d'accéder à la ligne de commande Python, la plus évidente consistant à passer par les menus `Démarrer` > `Tous les programmes` > `Python 3.4` > `Python (Command Line)`. Si tout se passe bien, vous devriez obtenir une magnifique console (figure suivante). Il se peut que les informations affichées dans la vôtre ne soient pas les mêmes, mais ne vous en inquiétez pas.



Qu'est-ce que c'est que cela ?

On verra plus tard. L'important, c'est que vous ayez réussi à ouvrir la console d'interprétation de Python, le reste attendra le prochain chapitre.

Vous pouvez également passer par la ligne de commande Windows ; à cause des raccourcis, je privilégie en général cette méthode, mais c'est une question de goût.
Allez dans le menu `Démarrer`, puis cliquez sur `Exécuter`. Dans la fenêtre qui s'affiche, tapez simplement « python » et la ligne de commande Python devrait s'afficher de nouveau. Sachez que vous pouvez directement vous rendre dans `Exécuter` en tapant le raccourci `Windows` + `R`.

Pour fermer l'interpréteur de commandes Python, vous pouvez tapez « exit() » puis appuyer sur la touche `Entrée`.

Sous Linux

Lorsque vous l'avez installé sur votre système, Python a créé un lien vers l'interpréteur sous la forme **python3.X** (le X étant le numéro de la version installée).

Si, par exemple, vous avez installé Python 3.4, vous pouvez y accéder grâce à la commande :

```
$ python3.4
Python 3.4.0 (default, Apr 23 2014, 05:55:41)
[GCC 4.4.5] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pour fermer la ligne de commande Python, n'utilisez pas `CTRL` + `C` mais `CTRL` + `D` (nous verrons plus tard pourquoi).

Sous Mac OS X

Cherchez un dossier `Python` dans le dossier `Applications`. Pour lancer Python, ouvrez l'application `IDLE` de ce dossier. Vous êtes prêts à passer au concret !

En résumé

- Python est un langage de programmation interprété, à ne pas confondre avec un langage compilé.
- Il permet de créer toutes sortes de programmes, comme des jeux, des logiciels, des progiciels, etc.
- Il est possible d'associer des **bibliothèques** à Python afin d'étendre ses possibilités.
- Il est portable, c'est à dire qu'il peut fonctionner sous différents systèmes d'exploitation (Windows, Linux, Mac OS X,…).

Premiers pas avec l'interpréteur de commandes Python

Après les premières notions théoriques et l'installation de Python, il est temps de découvrir un peu l'interpréteur de commandes de ce langage. Même si ces petits tests vous semblent anodins, vous découvrirez dans ce chapitre les premiers rudiments de la syntaxe du langage et je vous conseille fortement de me suivre pas à pas, surtout si vous êtes face à votre premier langage de programmation.

Comme tout langage de programmation, Python a une syntaxe claire : on ne peut pas lui envoyer n'importe quelle information dans n'importe quel ordre. Nous allons voir ici ce que Python mange… et ce qu'il ne mange pas.

Où est-ce qu'on est, là ?

Pour commencer, je vais vous demander de retourner dans l'interpréteur de commandes Python (je vous ai montré, à la fin du chapitre précédent, comment y accéder en fonction de votre système d'exploitation).

Je vous rappelle les informations qui figurent dans cette fenêtre, même si elles peuvent être différentes chez vous en fonction de votre version et de votre système d'exploitation.

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

À sa façon, Python vous souhaite la bienvenue dans son interpréteur de commandes.

Attends, attends. C'est quoi cet interpréteur ?

Souvenez-vous, au chapitre précédent, je vous ai donné une brève explication sur la différence entre langages compilés et langages interprétés. Eh bien, cet interpréteur de commandes va nous permettre de tester directement du code. Je saisis une ligne d'instructions, j'appuie sur la touche `Entrée` de mon clavier, je regarde ce que me répond Python (s'il me dit quelque chose), puis j'en saisis une deuxième, une troisième… Cet interpréteur est particulièrement utile pour comprendre les bases de Python et réaliser nos premiers petits programmes. Le principal inconvénient, c'est que le code que vous saisissez n'est pas sauvegardé (sauf si vous l'enregistrez manuellement, mais chaque chose en son temps).

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre est la série de trois chevrons qui se trouve en bas à gauche des informations : `>>>`. Ces trois signes signifient : « je suis prêt à recevoir tes instructions ».

Comme je l'ai dit, les langages de programmation respectent une syntaxe claire. Vous ne pouvez pas espérer que l'ordinateur comprenne si, dans cette fenêtre, vous commencez par lui demander : « j'aimerais que tu me codes un jeu vidéo génial ». Et autant que vous le sachiez tout de suite (bien qu'à

mon avis, vous vous en doutiez), on est très loin d'obtenir des résultats aussi spectaculaires à notre niveau.

Tout cela pour dire que, si vous saisissez n'importe quoi dans cette fenêtre, la probabilité est grande que Python vous indique, clairement et fermement, qu'il n'a rien compris.

Si, par exemple, vous saisissez « premier test avec Python », vous obtenez le résultat suivant :

```
>>> premier test avec Python
File "<stdin>", line 1
premier test avec Python
^
SyntaxError: invalid syntax
>>>
```

Eh oui, l'interpréteur parle en anglais et les instructions que vous saisirez, comme pour l'écrasante majorité des langages de programmation, seront également en anglais. Mais pour l'instant, rien de bien compliqué : l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction. Il vous indique le numéro de la ligne (en l'occurence la première), qu'il vous répète obligeamment (ceci est très utile quand on travaille sur un programme de plusieurs centaines de lignes). Puis il vous dit ce qui l'arrête, ici : `SyntaxError: invalid syntax`. Limpide n'est-ce pas ? Ce que vous avez saisi est incompréhensible pour Python. Enfin, la preuve qu'il n'est pas rancunier, c'est qu'il vous affiche à nouveau une série de trois chevrons, montrant bien qu'il est prêt à retenter l'aventure.

Bon, c'est bien joli de recevoir un message d'erreur au premier test mais je me doute que vous aimeriez bien voir des trucs qui fonctionnent, maintenant. C'est parti donc.

Vos premières instructions : un peu de calcul mental pour l'ordinateur

C'est assez trivial, quand on y pense, mais je trouve qu'il s'agit d'une excellente manière d'aborder pas à pas la syntaxe de Python. Nous allons donc essayer d'obtenir les résultats de calculs plus ou moins compliqués. Je vous rappelle encore une fois qu'exécuter les tests en même temps que moi sur votre machine est une très bonne façon de vous rendre compte de la syntaxe et surtout, de la retenir.

Saisir un nombre

Vous avez pu voir sur notre premier (et à ce jour notre dernier) test que Python n'aimait pas particulièrement les suites de lettres qu'il ne comprend pas. Par contre, l'interpréteur adore les nombres. D'ailleurs, il les accepte sans sourciller, sans une seule erreur :

D'accord, ce n'est pas extraordinaire. On saisit un nombre et l'interpréteur le renvoie. Mais dans bien des cas, ce simple retour indique que l'interpréteur a bien compris et que votre saisie est en accord avec sa syntaxe. De même, vous pouvez saisir des nombres à virgule.

Attention : on utilise ici la notation anglo-saxonne, c'est-à-dire que le point remplace la virgule. La virgule a un tout autre sens pour Python, prenez donc cette habitude dès maintenant.

Il va de soi que l'on peut tout aussi bien saisir des nombres négatifs (vous pouvez d'ailleurs faire l'essai).

Opérations courantes

Bon, il est temps d'apprendre à utiliser les principaux opérateurs de Python, qui vont vous servir pour la grande majorité de vos programmes.

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 + 2
11.5
>>> 3.11 + 2.08
5.1899999999999995
>>>
```

Pourquoi ce dernier résultat approximatif ?

Python n'y est pas pour grand chose. En fait, le problème vient en grande partie de la façon dont les nombres à virgule sont écrits dans la mémoire de
votre ordinateur. C'est pourquoi, en programmation, on préfère travailler autant que possible avec des nombres entiers. Cependant, vous remarquerez que l'erreur
est infime et qu'elle n'aura pas de réel impact sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve essayent de pallier ces défauts par d'autres moyens mais ici, ce ne sera pas nécessaire.

Faites également des tests pour la soustraction, la multiplication et la division : il n'y a rien de difficile.

Division entière et modulo

Si vous avez pris le temps de tester la division, vous vous êtes rendu compte que le résultat est donné avec une virgule flottante.

```
>>> 10 / 5
2.0
>>> 10 / 3
3.3333333333333335
>>>
```

Il existe deux autres opérateurs qui permettent de connaître le résultat d'une division entière et le reste de cette division.

Le premier opérateur utilise le symbole « // ». Il permet d'obtenir la partie entière d'une division.

L'opérateur « % », que l'on appelle le « modulo », permet de connaître le reste de la division.

Ces notions de *partie entière* et de *reste de division* ne sont pas bien difficiles à comprendre et vous serviront très probablement par la suite.

Si vous avez du mal à en saisir le sens, sachez donc que :

- La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurence, 3).
- Pour obtenir le modulo d'une division, on « récupère » son reste. Dans notre exemple, 10/3 = 3 et il reste 1. Une fois que l'on a compris cela, ce n'est pas bien compliqué.

Souvenez-vous bien de ces deux opérateurs, et surtout du modulo « % », dont vous aurez besoin dans vos programmes futurs.

En résumé

- L'interpréteur de commandes Python permet de tester du code au fur et à mesure qu'on l'écrit.
- L'interpréteur Python accepte des nombres et est capable d'effectuer des calculs.
- Un nombre décimal s'écrit avec un point et non une virgule.
- Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs, c'est pourquoi on préfèrera, dans la mesure du possible, travailler avec des nombres entiers.

Le monde merveilleux des variables

Au chapitre précédent, vous avez saisi vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions saisies auraient fonctionné dans la plupart des langages. Ici, cependant, nous allons commencer à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de la programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire l'impasse dessus. Mais je vous rassure, il n'y a rien de compliqué, que de l'utile et de l'agréable.

C'est quoi, une variable ? Et à quoi cela sert-il ?

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

C'est quoi, une variable ?

Une variable est une donnée de votre programme, stockée dans votre ordinateur. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants avec. C'est bien joli de savoir faire des opérations mais, si on ne peut pas stocker le résultat quelque part, cela devient très vite ennuyeux.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

Comment cela fonctionne-t-il ?

Le plus simplement du monde. Vous allez dire à Python : « je veux que, dans une variable que je nomme `age`, tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est ».

Comme je vous l'ai dit, on ne peut pas passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire `nom_de_la_variable = valeur`.

Une variable doit respecter quelques règles de syntaxe incontournables :

1. Le nom de la variable ne peut être composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné « _ » (appelé *underscore* en anglais).
2. Le nom de la variable ne peut pas commencer par un chiffre.

3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable `AGE` est différente de `aGe`, elle-même différente de `age`).

Au-delà de ces règles de syntaxe incontournables, il existe des conventions définies par les programmeurs eux-mêmes. L'une d'elles, que j'ai tendance à utiliser assez souvent, consiste à écrire la variable en minuscules et à remplacer les espaces éventuels par un espace souligné « _ ». Si je dois créer une variable contenant mon âge, elle se nommera donc `mon_age`. Une autre convention utilisée consiste à passer en majuscule le premier caractère de chaque mot, à l'exception du premier mot constituant la variable. La variable contenant mon âge se nommerait alors `monAge`.

Vous pouvez utiliser la convention qui vous plaît, ou même en créer une bien à vous, mais essayez de rester cohérent et de n'utiliser qu'une seule convention d'écriture. En effet, il est essentiel de pouvoir vous repérer dans vos variables dès que vous commencez à travailler sur des programmes volumineux.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

L'interpréteur vous affiche aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape *l'affectation de valeur à une variable* (parfois raccourci en « affectation de variable »). On dit en effet qu'on a affecté la valeur 21 à la variable `mon_age`.

On peut afficher la valeur de cette variable en la saisissant simplement dans l'interpréteur de commandes.

Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité.

Bon, c'est bien joli tout cela, mais qu'est-ce qu'on fait avec cette variable ?

Eh bien, tout ce que vous avez déjà fait au chapitre précédent, mais cette fois en utilisant la variable comme un nombre à part entière. Vous pouvez même affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable `mon_age`.

```
>>> mon_age = mon_age + 2
>>> mon_age
23
>>>
```

Encore une fois, lors de l'affectation de la valeur, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une valeur à une autre variable d'après la valeur de `mon_age`.

```
>>> mon_age_x2 = mon_age * 2
>>> mon_age_x2
46
>>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est extrêmement simple. Si la variable n'est pas créée, Python s'en charge automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?

Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

En voici la liste pour Python 3 :

`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`	
`def`	`for`	`lambda`	`return`	

Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

Les types de données en Python

Là se trouve un concept très important, que l'on retrouve dans beaucoup de langages de programmation. Ouvrez grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer la lecture de ce livre. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de compliqué à comprendre.

Qu'entend-on par « type de donnée » ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et, s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. À terme, vous serez même capables de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec. Dans ce chapitre, vous allez apprendre à travailler avec des chaînes de caractères, et multiplier une chaîne de caractères ne se fait pas du tout comme la multiplication d'un nombre. Pour

certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres entiers

Et oui, Python différencie les entiers des nombres à virgule flottante !

Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme `int` en Python (qui correspond à l'anglais « integer », c'est-à-dire entier). La forme d'un entier est un nombre sans virgule.

Nous avons vu au chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données et, même si vous ne vous en souvenez pas, les deviner est assez élémentaire.

Les nombres flottants

Les flottants sont les nombres à virgule. Ils se nomment `float` en Python (ce qui signifie « flottant » en anglais). La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point). Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui ajouter une partie flottante de 0 (exemple **52.0**).

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Mais la précision est assez importante pour travailler sur des données très fines.

Les chaînes de caractères

Heureusement, les types de données disponibles en Python ne sont pas limités aux seuls nombres, bien loin de là. Le dernier type « simple » que nous verrons dans ce chapitre est la chaîne de caractères. Ce type de donnée permet de stocker une série de lettres, pourquoi pas une phrase.

On peut écrire une chaîne de caractères de différentes façons :

- entre guillemets (`"ceci est une chaîne de caractères"`) ;
- entre apostrophes (`'ceci est une chaîne de caractères'`) ;
- entre triples guillemets (`"""ceci est une chaîne de caractères"""`).

On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaine = "Bonjour, la foule !"`)

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez `chaine = 'J'aime le Python!'`, vous obtenez le message suivant :

```
File "<stdin>", line 1
chaine = 'J'aime le Python!'
^
SyntaxError: invalid syntax
```

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà.
Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi un caractère anti-slash « \ » avant les apostrophes contenues dans le message.

```
chaine = 'J\'aime le Python!'
```

On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs.

```
chaine2 = "\"Le seul individu formé, c'est celui qui a appris comment apprendre
(...)\" (Karl Rogers, 1976)"
```

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire un véritable anti-slash dans une chaîne, il faut l'échapper lui-même (et donc écrire « \\ »).

L'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ». Nous verrons dans la partie suivante comment afficher réellement ces chaînes de caractères et pourquoi l'interpréteur ne les affiche pas comme il le devrait.

Utiliser les triples guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets et apostrophes, et permet d'écrire plusieurs lignes sans symboliser les retours à la ligne au moyen de « \n ».

```
>>> chaine3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points : cela signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction. En effet, celle-ci ne s'achève qu'une fois la chaîne refermée avec trois nouveaux guillemets. Les sauts de lignes seront automatiquement remplacés, dans la chaîne, par des « \n ».

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui jouent exactement le même rôle. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Voilà, nous avons bouclé le rapide tour d'horizon des types simples. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai mais nous n'allons pas, dans ce chapitre, entrer dans le détail

des opérations que l'on peut effectuer sur ces chaînes. C'est inutile pour l'instant et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous mêmes quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez et, comme nous l'avons vu, il est peu rancunier).

Un petit bonus

Au chapitre précédent, nous avons vu les opérateurs « classiques » pour manipuler des nombres mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amenés par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ci-dessous pour augmenter une variable de 1 :

Cette syntaxe est claire et intuitive mais assez longue, et les programmeurs, tout le monde le sait, sont des fainéants nés. On a donc trouvé plus court.

L'opérateur `+=` revient à ajouter à la variable la valeur qui suit l'opérateur. Les opérateurs -=, *= et /= existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs… ici c'est bien plus simple :

```
>>> a = 5
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables `a` et `b` ont échangé leurs valeurs. On retrouvera cette distribution d'affectation bien plus loin.

On peut aussi affecter assez simplement une même valeur à plusieurs variables :

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
... -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole « \ » permet, avant un saut de ligne, d'indiquer à Python que « cette instruction se poursuit à la ligne suivante ». Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Première utilisation des fonctions

Eh bien, tout cela avance gentiment. Je me permets donc d'introduire ici, dans ce chapitre sur les variables, l'utilisation des fonctions. Il s'agit finalement bien davantage d'une application concrète de ce que vous avez appris à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction

À quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour faire une action précise et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Mais nous verrons tout cela plus en détail par la suite.

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée ; ces paramètres sont des informations que vous passez à la fonction afin qu'elle travaille dessus. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble mais rassurez-vous, les exemples devraient tout rendre limpide.

Les fonctions s'utilisent en respectant la syntaxe suivante :\\
`nom_de_la_fonction(parametre_1,parametre_2,…,parametre_n)`.

- Vous commencez par écrire le nom de la fonction.
- Vous placez entre parenthèses les paramètres de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction « type »

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation. Mais il est pratique de pouvoir savoir de quel type est une variable.

La syntaxe de cette fonction est simple :

La fonction renvoie le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous saisissez dans l'interpréteur les lignes suivantes :

Vous obtenez :

Python vous indique donc que la variable `a` appartient à la classe des entiers. Cette notion de classe ne sera pas approfondie avant bien des chapitres mais sachez qu'on peut la rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

```
>>> type(3.4)
<class 'float'>
>>> type("un essai")
<class 'str'>
>>>
```

`str` est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

La fonction `print`

La fonction `print` permet d'afficher la valeur d'une ou plusieurs variables.

Mais… on ne fait pas exactement la même chose en saisissant juste le nom de la variable dans l'interpréteur ?

Oui et non. L'interpréteur affiche bien la valeur de la variable car il affiche automatiquement tout ce qu'il peut, pour pouvoir suivre les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction `print` est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

Le premier *appel* à `print` se contente d'afficher la valeur de la variable `a`, c'est-à-dire « 3 ».
Le second appel à `print` affiche :

Ce deuxième appel à `print` est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à `print`, deux chaînes de caractères et les variables `a` et `b`. Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant indépendamment chaque paramètre.

Testez l'utilisation de `print` avec d'autres types de données et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit « Hello World ! » ?

Quand on fait un cours sur un langage, quel qu'il soit, il est d'usage de présenter le programme « Hello World ! », qui illustre assez rapidement la syntaxe superficielle d'un langage.

Le but du jeu est très simple : écrire un programme qui affiche « Hello World ! » à l'écran. Dans certains langages, notamment les langages compilés, vous pourrez nécessiter jusqu'à une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il suffit d'une seule ligne :

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

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée à « Hello World ! » ; vous avez même des codes rédigés en différents langages de programmation, cela peut être intéressant.

En résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : `int`, `float`, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.

Les structures conditionnelles

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Mais nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas.

Dans ce chapitre, je vais vous parler des structures conditionnelles, qui vont vous permettre de faire des tests et d'aller plus loin dans la programmation.

Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

Vous finirez ce chapitre en créant votre premier « vrai » programme : même si vous ne pensez pas encore pouvoir faire quelque chose de très consistant, à la fin de ce chapitre vous aurez assez de matière pour coder un petit programme dans un but très précis.

Vos premières conditions et blocs d'instructions

Forme minimale en `if`

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Elles vont vous permettre de faire une action précise si, par exemple, une variable est positive, une autre action si cette variable est négative, ou une troisième action si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.

Dès à présent dans mes exemples, j'utiliserai des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (« # ») et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le # en début de ligne) ou une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

```
>>> # Premier exemple de condition
>>> a = 5
>>> if a > 0: # Si a est supérieur à 0
...     print("a est supérieur à 0.")
...
a est supérieur à 0.
>>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter la valeur 5 à la variable `a`.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
    - du mot clé `if` qui signifie « si » en anglais ;
    - de la condition proprement dite, `a > 0`, qu'il est facile de lire (une liste des opérateurs autorisés pour la comparaison sera présentée plus bas) ;
    - du signe deux points, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.
4. Ici se trouve l'instruction à exécuter dans le cas où `a` est supérieur à `0`. Après que vous ayez appuyé sur `Entrée` à la fin de la ligne précédente, l'interpréteur vous présente la série de trois points qui signifie qu'il attend la saisie du **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée vers la droite. Des explications supplémentaires seront données un peu plus bas sur les indentations.
5. L'interpréteur vous affiche à nouveau la série de trois points et vous pouvez en profiter pour saisir une nouvelle instruction dans ce bloc d'instructions. Ce n'est pas le cas pour l'instant. Vous appuyez donc sur `Entrée` sans avoir rien écrit et l'interpréteur vous affiche le message « a est supérieur à 0 », ce qui est assez logique vu que `a` est effectivement supérieur à `0`.

Il y a deux notions importantes sur lesquelles je dois à présent revenir, elles sont complémentaires ne vous en faites pas.

La première est celle de bloc d'instructions. On entend par bloc d'instructions une série d'instructions qui s'exécutent dans un cas précis (par condition, comme on vient de le voir, par répétition, comme on le verra plus tard…). Ici, notre bloc n'est constitué que d'une seule instruction (la ligne 4 qui fait appel à `print`). Mais rien ne vous empêche de mettre plusieurs instructions dans ce bloc.

```
a = 5
b = 8
if a > 0:
    # On incrémente la valeur de b
    b += 1
    # On affiche les valeurs des variables
    print("a =",a,"et b =",b)
```

La seconde notion importante est celle d'indentation. On entend par indentation un certain décalage vers la droite, obtenu par un (ou plusieurs) espaces ou tabulations.

Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java, d'un confort de lecture mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Forme complète (`if`, `elif` et `else`)

Les limites de la condition simple en `if`

La première forme de condition que l'on vient de voir est pratique mais assez incomplète.

Considérons, par exemple, une variable `a` de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

```
>>> a = 5
>>> if a > 0: # Si a est positif
...    print("a est positif.")
... if a < 0: # a est négatif
...    print("a est négatif.")
```

Amusez-vous à changer la valeur de `a` et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si `a` est égal à 0. En effet, aucune action n'a été prévue si a vaut 0.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition `if` est donc bien pratique mais insuffisante.

L'instruction `else`:

Le mot-clé `else`, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction `if`.

```
>>> age = 21
>>> if age >= 18: # Si age est supérieur ou égal à 18
...    print("Vous êtes majeur.")
... else: # Sinon (age inférieur à 18)
...    print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de `else`. La seule subtilité est de bien se rendre compte que Python exécute soit l'un, soit l'autre, et jamais les deux. Notez que cette instruction `else` doit se trouver au même niveau d'indentation que l'instruction `if` qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple de tout à l'heure pourrait donc se présenter comme suit, avec l'utilisation de `else` :

```
>>> a = 5
>>> if a > 0:
```

```
...    print("a est supérieur à 0.")
... else:
...    print("a est inférieur ou égal à 0.")
```

Mais… le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que, cette fois, le cas où `a` vaut `0` est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si `a` est strictement supérieur à `0`. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs et nuls, il va falloir utiliser une condition intermédiaire.

L'instruction `elif`:

Le mot clé `elif` est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

- si `a` est strictement supérieur à `0`, on dit qu'il est positif ;
- sinon si `a` est strictement inférieur à `0`, on dit qu'il est négatif ;
- sinon, (`a` ne peut qu'être égal à `0`), on dit alors que `a` est nul.

Traduit en langage Python, cela donne :

```
>>> if a > 0: # Positif
...     print("a est positif.")
... elif a < 0: # Négatif
...     print("a est négatif.")
... else: # Nul
        print("a est nul.")
```

De même que le `else`, le `elif` est sur le même niveau d'indentation que le `if` initial. Il se termine aussi par deux points. Cependant, entre le `elif` et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si `a` est strictement supérieur à `0`. Si c'est le cas, on affiche « a est positif » et on s'arrête là.
2. Sinon, on regarde si `a` est strictement inférieur à `0`. Si c'est le cas, on affiche « a est négatif » et on s'arrête.
3. Sinon, on affiche « a est nul ».

Attention : quand je dis « on s'arrête », il va de soi que c'est uniquement pour cette condition. S'il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de `elif` que vous voulez après une condition en `if`. Tout comme le `else`, cette instruction est facultative et, quand bien même vous construiriez une instruction en `if`, `elif`, vous n'êtes pas du tout obligé de prévoir un `else` après. En revanche, l'instruction `else` ne peut

figurer qu'une fois, clôturant le bloc de la condition. Deux instructions `else` dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place non plus. :-)

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**. Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que les conditions qui se trouvent, par exemple, entre `if` et les deux points sont appelés des **prédicats**. Vous pouvez tester ces prédicats directement dans l'interpréteur pour comprendre les explications qui vont suivre.

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt `True` (c'est-à-dire « vrai »), tantôt `False` (c'est-à-dire « faux »).

`True` et `False` sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (bool).

N'oubliez pas que `True` et `False` sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire `True` sans un 'T' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et peuvent être pratiques, justement, pour stocker des prédicats, de la façon que nous avons vue ou d'une façon plus détournée.

```
>>> age = 21
>>> majeur = False
>>> if age >= 18:
>>>     majeur = True
>>>
```

À la fin de cet exemple, majeur vaut `True`, c'est-à-dire « vrai », si l'âge est supérieur ou égal à 18. Sinon, il continue de valoir `False`. Les booléens ne vous semblent peut-être pas très utiles pour l'instant mais vous verrez qu'ils rendent de grands services !

Les mots-clés `and`, `or` et `not`

Il arrive souvent que nos conditions doivent tester plusieurs prédicats, par exemple quand l'on cherche à vérifier si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait d'écrire :

```
# On fait un test pour savoir si a est comprise dans l'intervalle allant de 2 à 8
inclus
a = 5
if a >= 2:
    if a <= 8:
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Cela marche mais c'est assez lourd, d'autant que, pour être sûr qu'un message soit affiché à chaque fois, il faut fermer chacune des deux conditions à l'aide d'un `else` (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne, il n'y a rien de compliqué.

Il existe cependant le mot clé `and` (qui signifie « et » en anglais) qui va nous rendre ici un fier service. En effet, on cherche à tester à la fois si `a` est supérieur ou égal à 2 et inférieur ou égal à 8. On peut donc réduire ainsi les conditions imbriquées :

```
if a>=2 and a<=8:
    print("a est dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Simple et bien plus compréhensible, avouez-le.

Sur le même mode, il existe le mot clé `or` qui signifie cette fois « ou ». Nous allons prendre le même exemple, sauf que nous allons évaluer notre condition différemment.

Nous allons chercher à savoir si `a` n'est pas dans l'intervalle. La variable ne se trouve pas dans l'intervalle si elle est inférieure à 2 ou supérieure à 8. Voici donc le code :

```
if a<2 or a>8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Enfin, il existe le mot clé `not` qui « inverse » un prédicat. Le prédicat `not a==5` équivaut donc à `a!=5`.

`not` rend la syntaxe plus claire. Pour cet exemple, j'ajoute à la liste un nouveau mot clé, `is`, qui teste l'égalité non pas des valeurs de deux variables, mais de leurs références. Je ne vais pas rentrer dans le détail de ce mécanisme avant longtemps. Il vous suffit de savoir que pour les entiers, les flottants et les booléens, c'est strictement la même chose. Mais pour tester une égalité entre variables dont le type est plus complexe, préférez l'opérateur « == ». Revenons à cette démonstration :

```
>>> majeur = False
>>> if majeur is not True:
...     print("Vous n'êtes pas encore majeur.")
...
Vous n'êtes pas encore majeur.
>>>
```

Si vous parlez un minimum l'anglais, ce prédicat est limpide et d'une simplicité sans égale.

Vous pouvez tester des prédicats plus complexes de la même façon que les précédents, en les saisissant directement, sans le `if` ni les deux points, dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer suivant des priorités bien précises (nous verrons ce point plus loin, si vous n'en comprenez pas l'utilité).

Votre premier programme !

À quoi on joue ?

L'heure du premier TP est venue. Comme il s'agit du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous parveniez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette section écrire votre premier programme. Vous allez sûrement tester les syntaxes directement dans l'interpréteur de commandes.

Vous pourriez préférer écrire votre code directement dans un fichier que vous pouvez ensuite exécuter. Si c'est le cas, je vous renvoie au chapitre traitant de ce point dans ce cours.

Sujet

Le but de notre programme est de déterminer si une année saisie par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons, donc, à ceux qui ont déjà fait cet exercice dans un autre langage mais je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Je vous rappelle les règles qui déterminent si une année est bissextile ou non (vous allez peut-être même apprendre des choses que le commun des mortels ignore).

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
    - Si c'est le cas, on regarde si elle est multiple de 400.
        - Si c'est le cas, l'année est bissextile.
        - Sinon, elle n'est pas bissextile.
    - Sinon, elle est bissextile.

Solution ou résolution

Voilà. Le problème est posé clairement (sinon relisez attentivement l'énoncé autant de fois que nécessaire), il faut maintenant réfléchir à sa résolution en termes de programmation. C'est une phase de transition assez délicate de prime abord et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique, autrement dit, on réfléchit au programme sans réfléchir au code proprement dit.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début et, parfois il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction `input()`

Tout d'abord, j'ai mentionné une année saisie par l'utilisateur. En effet, depuis tout à l'heure, nous testons des variables que nous déclarons nous-mêmes, avec une valeur précise. La condition est donc assez ridicule.

`input()` est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre saisi par l'utilisateur.

`input()` accepte un paramètre facultatif : le message à afficher à l'utilisateur. Cette instruction interrompt le programme et attend que l'utilisateur saisisse ce qu'il veut puis appuie sur `Entrée`. À cet instant, la fonction renvoie ce que l'utilisateur a saisi. Il faut donc piéger cette valeur dans une variable.

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

Il subsiste un problème : le type de la variable `annee` après l'appel à `input()` est… une chaîne de caractères. Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux : nous qui voulions travailler sur un entier, nous allons devoir convertir cette variable. Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction (c'est d'ailleurs exactement ce que c'est).

```
>>> type(annee)
<type 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que, si vous saisissez des lettres lors de l'appel à `input()`, la conversion renverra une erreur.

L'appel à la fonction `int()` en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractères issue de `input()`, pour tenter de la convertir. La fonction `int()` renvoie la valeur convertie en entier et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit, en fait, de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

```
>>> 5 % 2 # 5 n'est pas un multiple de 2
1
>>> 8 % 2 # 8 est un multiple de 2
0
>>>
```

À vous de jouer

Je pense vous avoir donné tous les éléments nécessaires pour réussir. À mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la section suivante.

Bonne chance !

Correction

C'est l'heure de comparer nos méthodes et, avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible. Vous pouvez très bien avoir trouvé quelque chose de différent mais qui fonctionne tout aussi bien.

Attention… la voiiiciiiiiii…

```
# Programme testant si une année, saisie par l'utilisateur,
# est bissextile ou non
annee = input("Saisissez une année : ") # On attend que l'utilisateur saisisse
l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre
bissextile = False # On crée un booléen qui vaut vrai ou faux
                   # selon que l'année est bissextile ou non
if annee % 400 == 0:
    bissextile = True
elif annee % 100 == 0:
    bissextile = False
elif annee % 4 == 0:
    bissextile = True
else:
    bissextile = False
if bissextile: # Si l'année est bissextile
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers afin de les exécuter. Je vous renvoie au chapitre sur l'écriture de code Python dans des fichiers pour plus d'informations.

Je pense que le code est assez clair, reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On teste en effet d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100, et enfin si c'est un multiple de 4. En effet, le `elif` garantit que, si `annee` est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité au-dessus). De cette façon, on s'assure que tous les cas sont gérés. Vous pouvez faire des essais avec plusieurs années et vous rendre compte si le programme a raison ou pas.

L'utilisation de `bissextile` comme d'un prédicat à part entière vous a peut-être déconcertés. C'est en fait tout à fait possible et logique, puisque `bissextile` est un booléen. Il est de ce fait vrai ou faux et donc on peut le tester simplement. On peut bien entendu aussi écrire `if bissextile==True:`, cela revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition, que je vais passer au crible afin d'en construire une plus courte et plus logique, si possible. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de diminuer ces ressources et d'améliorer la rapidité de l'application. Mais, pour une petite application comme celle-ci, je ne pense pas qu'on perdra du temps sur l'optimisation du temps d'exécution.

Le premier détail que vous auriez pu remarquer, c'est que le `else` de fin est inutile. En effet, la variable `bissextile` vaut par défaut `False` et conserve donc cette valeur si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition car les deux seuls cas correspondant à une année bissextile sont « si l'année est un multiple de 400 »ou « si l'année est un multiple de 4 mais pas de 100 ».

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Je ne m'attendais pas que vous le trouviez tout seuls mais je souhaite que vous le compreniez bien à présent.

```
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non
annee = input("Saisissez une année : ") # On attend que l'utilisateur saisisse
l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre
if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

Du coup, on n'a plus besoin de la variable `bissextile`, c'est déjà cela de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien.

En résumé

- Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
- Les conditions sont marquées par les mot-clés `if` (« si »), `elif` (« sinon si ») et `else` (« sinon »).
- Les mot-clés `if` et `elif` doivent être suivis d'un test (appelé aussi prédicat).
- Les booléens sont des données soit vraies (`True`) soit fausses (`False`).

Les boucles

Les boucles sont un concept nouveau pour vous. Elles vont vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique car les applications pratiques présentées dans ce chapitre ne vous paraîtront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle.

En outre, les boucles peuvent permettre de parcourir certaines séquences comme les chaînes de caractères pour, par exemple, en extraire chaque caractère.

Alors, on commence ?

En quoi cela consiste-t-il ?

Comme je l'ai dit juste au-dessus, les boucles constituent un moyen de répéter un certain nombre de fois des instructions de votre programme. Prenons un exemple simple, même s'il est assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

… bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains. Dans un premier temps, vous devriez arriver au programme suivant :

```
print(" 1 * 7 =", 1 * 7)
print(" 2 * 7 =", 2 * 7)
print(" 3 * 7 =", 3 * 7)
print(" 4 * 7 =", 4 * 7)
print(" 5 * 7 =", 5 * 7)
print(" 6 * 7 =", 6 * 7)
print(" 7 * 7 =", 7 * 7)
print(" 8 * 7 =", 8 * 7)
print(" 9 * 7 =", 9 * 7)
print("10 * 7 =", 10 * 7)
```

… et le résultat :

```
1 * 7 = 7
 2 * 7 = 14
 3 * 7 = 21
 4 * 7 = 28
 5 * 7 = 35
 6 * 7 = 42
 7 * 7 = 49
 8 * 7 = 56
 9 * 7 = 63
10 * 7 = 70
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers. Vous trouverez la marche à suivre sur le chapitre sur l'écriture de code Python dans des fichiers.

Bon, c'est sûrement la première idée qui vous est venue et cela fonctionne, très bien même. Seulement, vous reconnaîtrez qu'un programme comme cela n'est pas bien utile.
Essayons donc le même programme mais, cette fois-ci, en utilisant une variable ; ainsi, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable ! Pour cet exemple, on utilise une variable `nb` qui contiendra 7. Les instructions seront légèrement différentes mais vous devriez toujours pouvoir écrire ce programme :

```
nb = 7
print(" 1 *", nb, "=", 1 * nb)
print(" 2 *", nb, "=", 2 * nb)
print(" 3 *", nb, "=", 3 * nb)
print(" 4 *", nb, "=", 4 * nb)
print(" 5 *", nb, "=", 5 * nb)
print(" 6 *", nb, "=", 6 * nb)
print(" 7 *", nb, "=", 7 * nb)
print(" 8 *", nb, "=", 8 * nb)
print(" 9 *", nb, "=", 9 * nb)
print("10 *", nb, "=", 10 * nb)
```

Le résultat est le même, vous pouvez vérifier. Mais le code est quand-même un peu plus intéressant : on peut changer la table de multiplication à afficher en changeant la valeur de la variable `nb`.

Mais ce programme reste assez peu pratique et il accomplit une tâche bien répétitive. Les programmeurs étant très paresseux, ils préfèrent utiliser les boucles.

La boucle while

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation et porte le même nom. Elle permet de répéter un **bloc d'instructions** tant qu'une condition est vraie (`while` signifie « tant que » en anglais). J'espère que le concept de **bloc d'instructions** est clair pour vous, sinon je vous renvoie au chapitre précédent.

La syntaxe de `while` est :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaître la forme d'un bloc d'instructions, du moins je l'espère.

Quelle condition va-t-on utiliser ?

Eh bien, c'est là le point important. Dans cet exemple, on va créer une variable qui sera incrémentée dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez ce code, quelques commentaires suffiront pour le comprendre :

```
nb = 7 # On garde la variable contenant le nombre dont on veut la table de
multiplication
i = 0 # C'est notre variable compteur que nous allons incrémenter dans la boucle
while i < 10: # Tant que i est strictement inférieure à 10
    print(i + 1, "*", nb, "=", (i + 1) * nb)
    i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Analysons ce code ligne par ligne :

1. On instancie la variable `nb` qui accueille le nombre sur lequel nous allons travailler (en l'occurence, 7). Vous pouvez bien entendu faire saisir ce nombre par l'utilisateur, vous savez le faire à présent.
2. On instancie la variable `i` qui sera notre compteur durant la boucle. `i` est un standard utilisé quand il est question de boucles et de variables s'incrémentant mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 0.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction `while` qui se décode, comme je l'ai indiqué en commentaire, en **« tant que *i* est strictement inférieure à 10 »**. N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du `print`, vous devez la reconnaître. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques. Vous remarquez qu'à chaque fois qu'on utilise `i` dans cette ligne, pour l'affichage ou le calcul, on lui ajoute 1 : cela est dû au fait qu'en programmation, on a l'habitude (habitude que vous devrez prendre) de commencer à compter à partir de 0. Seulement ce n'est pas le cas de la table de multiplication, qui va de 1 à 10 et non de 0 à 9, comme c'est le cas pour les valeurs de `i`. Certes, j'aurais pu changer la condition et la valeur initiale de `i`, ou même placer l'incrémentation de `i` avant l'affichage, mais j'ai voulu prendre le cas le plus courant, le format de boucle que vous retrouverez le plus souvent. Rien ne vous empêche de faire les tests et je vous y encourage même.
6. Ici, on incrémente la variable `i` de 1. Si on est dans le premier tour de boucle, `i` passe donc de 0 à 1. Et alors, puisqu'il s'agit de la fin du bloc d'instructions, on revient à l'instruction `while`. `while` vérifie que la valeur de `i` est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on exécute à nouveau le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que `i` passe de 9 à 10. Alors, l'instruction `while` vérifie la condition, se rend compte qu'elle est à présent fausse (la valeur de `i` n'est pas inférieure à 10 puisqu'elle est maintenant égale à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.

N'oubliez pas d'incrémenter `i` ! Sinon, vous créez ce qu'on appelle une boucle infinie, puisque la valeur de `i` n'est jamais supérieure à 10 et la condition du `while`, par conséquent, toujours vraie… La boucle s'exécute donc à l'infini, du moins en théorie. Si votre ordinateur se lance dans une boucle infinie à

cause de votre programme, pour interrompre la boucle, vous devrez taper `CTRL` + `C` dans la fenêtre de l'interpréteur (sous Windows ou Linux). Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

La boucle for

Comme je l'ai dit précédemment, on retrouve l'instruction `while` dans la plupart des autres langages. Dans le C++ ou le Java, on retrouve également des instructions `for` mais qui n'ont pas le même sens. C'est assez particulier et c'est le point sur lequel je risque de manquer d'exemples dans l'immédiat, toute son utilité se révélant au chapitre sur les listes. Notez que, si vous avez fait du Perl ou du PHP, vous pouvez retrouver les boucles `for` sous un mot-clé assez proche : `foreach`.

L'instruction `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Nous n'avons pas vu (et nous ne verrons pas tout de suite) ces séquences assez particulières mais très répandues, même si elles peuvent se révéler complexes. Toutefois, il en existe un type que nous avons rencontré depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des séquences… de caractères ! Vous pouvez parcourir une chaîne de caractères (ce qui est également possible avec `while` mais nous verrons plus tard comment). Pour l'instant, intéressons-nous à `for`.

L'instruction `for` se construit ainsi :

`element` est une variable créée par le `for`, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code !

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    print(lettre)
```

Ce qui nous donne le résultat suivant :

```
B
o
n
j
o
u
r
l
e
s
Z
E
R
0
S
```

Est-ce plus clair ? En fait, la variable `lettre` prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n…). On affiche ces valeurs avec `print` et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie **« pour lettre dans chaine »**. Arrivé à cette ligne, l'interpréteur va créer une variable `lettre` qui contiendra le premier élément de la chaîne (autrement dit, la première lettre). Après l'exécution du bloc, la variable `lettre` contient la seconde lettre, et ainsi de suite tant qu'il y a une lettre dans la chaîne.

Notez bien que, du coup, il est inutile d'incrémenter la variable `lettre` (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction `for`.

À l'instar des conditions que nous avons vues jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle `for`.

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    if lettre in "AEIOUYaeiouy": # lettre est une voyelle
        print(lettre)
    else: # lettre est une consonne... ou plus exactement, lettre n'est pas une
voyelle
        print("*")
```

… ce qui donne :

```
*
o
*
*
o
u
*
*
*
e
*
*
*
E
*
*
*
```

Voilà ! L'interpréteur affiche les lettres si ce sont des voyelles et, sinon, il affiche des « * ». Notez bien que le 0 n'est pas affiché à la fin, Python ne se doute nullement qu'il s'agit d'un « o » stylisé.

Retenez bien cette utilisation de `in` dans une condition. On cherche à savoir si un élément quelconque est contenu dans une collection donnée (ici, si la lettre est contenue dans « AEIOUYaeiouy », c'est-à-dire si `lettre` est une voyelle). On retrouvera plus loin cette fonctionnalité.

Un petit bonus : les mots-clés break et continue

Je vais ici vous montrer deux nouveaux mots-clés, `break` et `continue`. Vous ne les utiliserez peut-être pas beaucoup mais vous devez au moins savoir qu'ils existent… et à quoi ils servent.

Le mot-clé `break`

Le mot-clé **`break`** permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

```
while 1: # 1 est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

La boucle `while` a pour condition `1`, c'est-à-dire une condition qui sera *toujours* vraie. Autrement dit, en regardant la ligne du `while`, on pense à une boucle infinie. En pratique, on demande à l'utilisateur de taper une lettre (un 'Q' pour quitter). Tant que l'utilisateur ne saisit pas cette lettre, le programme lui redemande de taper une lettre. Quand il tape 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé `break`.

Ce mot-clé permet d'arrêter une boucle quelle que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

C'est un exemple un peu simpliste mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas où `break` est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du `while`. Par exemple, pourquoi ne pas créer un booléen qui sera **vrai** tout au long de la boucle et **faux** quand la boucle doit s'arrêter ? Ou bien tester directement si `lettre != « Q »` dans le `while` ?

Parfois, `break` est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`, qui sera plus dur à appréhender d'un seul coup d'œil.

Le mot-clé `continue`

Le mot-clé `continue` permet de… continuer une boucle, en repartant directement à la ligne du `while` ou `for`. Un petit exemple s'impose, je pense :

```
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # On retourne au while sans exécuter les autres lignes
    print("La variable i =", i)
    i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

```
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, `i` s'incrémente de 4. Arrivé au mot-clé `continue`, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du `while`. Autrement dit, quand Python arrive à la ligne 6, il saute à la ligne 2 sans exécuter les lignes 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale de la boucle (`continue` n'ignore la fin du bloc que pour le tour de boucle courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de `continue`. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes. L'essentiel, pour l'instant, c'est que vous vous souveniez de ces deux mots-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction. Personnellement, je n'utilise pas très souvent ces mots-clés mais c'est aussi une question de goût.

En résumé

- Une boucle sert à répéter une portion de code en fonction d'un prédicat.
- On peut créer une boucle grâce au mot-clé `while` suivi d'un prédicat.
- On peut parcourir une séquence grâce à la syntaxe `for element in sequence:`.

Pas à pas vers la modularité (1/2)

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions dans un but très précis. Attention, je ne parle pas ici de boucles. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions ou modules. Je vais détailler tranquillement ces deux concepts.

Les fonctions permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom. D'ailleurs, vous avez déjà vu des fonctions : `print` et `input` en font partie par exemple.

Les modules permettent de regrouper plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié aux mathématiques.

Les fonctions : à vous de jouer

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire `print`, `type` et `input`, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que nous allons nous habituer à regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser. Au prochain chapitre, nous apprendrons à regrouper nos fonctions ayant un rapport entre elles dans un fichier, pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons, pour illustrer cet exemple, reprendre le code de la table de multiplication, que nous avons vu au chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons `table_par_7`.

On crée une fonction selon le schéma suivant :

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, parametreN):

    # Bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer. Si l'on décortique la ligne de définition de la fonction, on trouve dans l'ordre :

- `def`, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.

- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
def table_par_7():

    nb = 7

    i = 0 # Notre compteur ! L'auriez-vous oublié ?

    while i < 10: # Tant que i est strictement inférieure à 10,

        print(i + 1, "*", nb, "=", (i + 1) * nb)

        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
>>> table_par_7()

1 * 7 = 7

2 * 7 = 14

3 * 7 = 21

4 * 7 = 28

5 * 7 = 35

6 * 7 = 42

7 * 7 = 49

8 * 7 = 56

9 * 7 = 63

10 * 7 = 70

>>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire au chapitre précédent et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !

Mais, si on saisit des paramètres pour pouvoir afficher la table de 5 ou de 8… ?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

```
def table(nb):
    i = 0
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez passer en argument différents nombres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
def table(nb, max):
    i = 0
    while i < max: # Tant que i est strictement inférieure à la variable max,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Si vous tapez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ?

Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom: nous verrons cela plus loin.

Si vous fournissez en second paramètre un nombre négatif, vous avez toutes les chances de créer une magnifique boucle infinie… vous pouvez l'empêcher en rajoutant des vérifications avant la boucle : par exemple, si le nombre est négatif ou nul, je le mets à 10. En Python, on préférera mettre un commentaire en tête de fonction ou une `docstring`, comme on le verra ultérieurement, pour indiquer que `max` doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflétant la philosophie du langage et qui peut s'appliquer à ce type de situation est « *we're all consenting adults here* » (en français, « Nous sommes entre adultes consentants »). Sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code. On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut pour les paramètres de la fonction. Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

```
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb
    de 1*nb à max*nb

    (max >= 0)"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Il suffit de rajouter `=10` après `max`. À présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, `max` vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez sans doute remarquées. Nous avons placé une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on appelle une `docstring` que l'on pourrait traduire par une chaîne d'aide. Si vous tapez `help(table)`, c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

```
def fonc(a=1, b=2, c=3, d=4, e=5):
    print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Simple, n'est-ce pas ? Eh bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	Résultat
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5
`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications supplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur (qui peut être une variable bien entendu). Peu importent les paramètres que vous précisez (comme vous le voyez dans cet exemple où tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par « signature de fonction » les éléments qui permettent au langage d'identifier ladite fonction. En C++, par exemple, la signature d'une fonction est constituée de son nom et du type de chacun de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions portant le même nom mais dont les paramètres diffèrent. Au moment de l'appel de fonction, le compilateur recherche la fonction qui s'applique à cette signature.

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
def exemple():

    print("Un exemple d'une fonction sans paramètre")



exemple()



def exemple(): # On redéfinit la fonction exemple

    print("Un autre exemple de fonction sans paramètre")



exemple()
```

A la ligne 1 on définit la fonction `exemple`. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la fonction `exemple`. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction unique, on ne peut surcharger de fonctions en Python.

L'instruction `return`

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que `input` ou `type` qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
def carre(valeur):

    return valeur * valeur
```

L'instruction `return` signifie qu'on va **renvoyer** la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le `return` ne s'exécutera pas.

Certains d'entre vous ont peut-être l'habitude d'employer le mot « retourner » ; il s'agit d'un anglicisme et je lui préfère l'expression « renvoyer ».

```
variable = carre(5)
```

La variable `variable` contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction `return` et ce sera déjà bien.

Les fonctions lambda

Nous venons de voir comment créer une fonction grâce au mot-clé `def`. Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes car limitées à une seule instruction.

Pourquoi une autre façon de créer des fonctions ? La première suffit, non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte, pour lequel définir une fonction à l'aide de `def` serait plus long et moins pratique.

Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction `lambda`. Nous allons utiliser le mot-clé `lambda` comme ceci : `lambda arg1, arg2,… : instruction de retour`.

Je pense qu'un exemple vous semblera plus clair. On veut créer une fonction qui prend un paramètre et renvoie ce paramètre au carré.

```
>>> lambda x: x * x

<function <lambda> at 0x00BA1B70>

>>>
```

D'abord, on a le mot-clé `lambda` suivi de la liste des arguments, séparés par des virgules. Ici, il n'y a qu'un seul argument, c'est `x`. Ensuite figure un nouveau signe deux points « : » et l'instruction de la `lambda`. C'est le résultat de l'instruction que vous placez ici qui sera renvoyé par la fonction. Dans notre exemple, on renvoie donc `x * x`.

Comment fait-on pour appeler notre `lambda` ?

On a bien créé une fonction `lambda` mais on ne dispose ici d'aucun moyen pour l'appeler. Vous pouvez tout simplement stocker votre fonction `lambda` nouvellement définie dans une variable, par une simple affectation :

```
>>> f = lambda x: x * x

>>> f(5)

25

>>> f(-18)

324

>>>
```

Un autre exemple : si vous voulez créer une fonction `lambda` prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

```
lambda x, y: x + y
```

Utilisation

Il vous faudra cependant attendre un peu pour que je vous montre une réelle application des `lambda`. En attendant, n'oubliez pas ce mot-clé et la syntaxe qui va avec… on passe à la suite !

À la découverte des modules

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Il y en a déjà un certain nombre et nous pourrions continuer et finir cette première partie sans utiliser de module Python… ou presque. Mais il faut bien qu'à un moment, je vous montre cette possibilité des plus intéressantes !

Les modules, qu'est-ce que c'est ?

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à **importer** le module et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module lui-même pour coder une calculatrice scientifique, nous verrons surtout les différentes méthodes d'importation.

La méthode `import`

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module `math` ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
>>> import math

>>>
```

La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe… en apparence. En réalité, Python vient d'importer le module `math`. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un

point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4
>>>
```

Comme vous le voyez, la fonction `sqrt` du module `math` renvoie la racine carrée du nombre passé en paramètre.

Mais comment suis-je censé savoir quelles fonctions existent et ce que fait `math.sqrt` dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt car, oui, c'est une fonction qui va nous donner la solution. Il s'agit de `help`, qui prend en argument la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais mais c'est de l'anglais technique, c'est-à-dire une forme de l'anglais que vous devrez maîtriser pour programmer, si ce n'est pas déjà le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez maintenant en trouver une bonne part en français.

```
>>> help("math")
Help on built-in module math:


NAME
    math


FILE
    (built-in)


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


FUNCTIONS
    acos(...)
```

```
        acos(x)


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


    acosh(...)
        acosh(x)


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


    asin(...)
-- Suite  --
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module `math`. Vous voyez en haut de la page le nom du module, le fichier qui l'héberge, puis la description du module. Ensuite se trouve une liste des fonctions, chacune étant accompagnée d'une courte description.

Tapez `Q` pour revenir à la fenêtre d'interpréteur, `Espace` pour avancer d'une page, `Entrée` pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:


sqrt(...)
    sqrt(x)


    Return the square root of x.


>>>
```

Ne mettez pas les parenthèses habituelles après le nom de la fonction. C'est en réalité la référence de la fonction que vous envoyez à `help`. Si vous rajoutez les parenthèses ouvrantes et fermantes après le

nom de la fonction, vous devrez préciser une valeur. Dans ce cas, c'est la valeur renvoyée par `math.sqrt` qui sera analysée, soit un nombre (entier ou flottant).

Nous reviendrons plus tard sur le concept des références des fonctions. Si vous avez compris pourquoi il ne fallait pas mettre de parenthèses après le nom de la fonction dans `help`, tant mieux. Sinon, ce n'est pas grave, nous y reviendrons en temps voulu.

Utiliser un espace de noms spécifique

En vérité, quand vous tapez `import math`, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module `math`. Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction `sqrt` que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math` que vous appellerez grâce à l'instruction `math.sqrt`.

Mais, concrètement, un espace de noms, c'est quoi ?

Il s'agit de regrouper certaines fonctions et variables sous un préfixe spécifique. Prenons un exemple concret :

```
import math

a = 5

b = 33.2
```

Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début de ce tutoriel, on trouve :

- La variable `a`.
- La variable `b`.
- Le module `math`, qui se trouve dans un espace de noms s'appelant `math` également. Dans cet espace de noms, on trouve :
  - la fonction `sqrt` ;
  - la variable `pi` ;
  - et bien d'autres fonctions et variables…

C'est aussi l'intérêt des modules : des variables et fonctions sont stockées à part, bien à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Mais dans certains cas, vous pourrez vouloir changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
```

```
mathematiques.sqrt(25)
```

Qu'est-ce qu'on a fait là ?

On a simplement importé le module `math` en spécifiant à Python de l'héberger dans l'espace de noms dénommé « mathematiques » au lieu de `math`. Cela permet de mieux contrôler les espaces de noms des modules que vous importerez. Dans la plupart des cas, vous n'utiliserez pas cette fonctionnalité mais, au moins, vous savez qu'elle existe. Quand on se penchera sur les packages, vous vous souviendrez probablement de cette possibilité.

Une autre méthode d'importation : `from … import …`

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module `math`. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
>>> from math import fabs

>>> fabs(-5)

5

>>> fabs(2)

2

>>>
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative, et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe `math.` devant le nom de la fonction. En effet, nous l'avons importée avec la méthode `from` : celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme `print` par exemple. Si vous avez compris les explications sur les espaces de noms, vous voyez que `print` et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

```
>>> from math import *

>>> sqrt(4)

2
```

```
>>> fabs(5)
5
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module `math` et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms `math`.

Bilan

Quelle méthode faut-il utiliser ?

Vaste question ! Je dirais que c'est à vous de voir. La seconde méthode a l'avantage inestimable d'économiser la saisie systématique du nom du module en préfixe de chaque fonction. L'inconvénient de cette méthode apparaît si l'on utilise plusieurs modules de cette manière : si par hasard il existe dans deux modules différents deux fonctions portant le même nom, l'interpréteur ne conservera que la dernière fonction appelée (je vous rappelle qu'il ne peut y avoir deux variables ou fonctions portant le même nom). Conclusion… c'est à vous de voir en fonction de vos besoins !

En résumé

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonctions permet une meilleure organisation.
- Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé `return`.
- Les fonctions se définissent de la façon suivante : `def nom_fonction(parametre1, parametre2, parametreN):`

Pas à pas vers la modularité (2/2)

Nous allons commencer par voir comment mettre nos programmes en boîte… ou plutôt en fichier. Je vais faire d'une pierre deux coups : d'abord, c'est chouette d'avoir son programme dans un fichier modifiable à souhait, surtout qu'on commence à pouvoir faire des programmes assez sympas (même si vous n'en avez peut-être pas l'impression). Ensuite, c'est un prélude nécessaire à la création de modules.

Comme vous allez le voir, nos programmes Python peuvent être mis dans des fichiers pour être exécutés ultérieurement. De ce fait, vous avez déjà pratiquement toutes les clés pour créer un programme Python exécutable. Le même mécanisme est utilisé pour la création de modules. Les modules sont eux aussi des fichiers contenant du code Python.

Enfin, nous verrons à la fin de ce chapitre comment créer des **packages** pour regrouper nos modules ayant un rapport entre eux.

C'est parti !

Mettre en boîte notre code

Fini, l'interpréteur ?

Je le répète encore, l'interpréteur est véritablement très pratique pour un grand nombre de raisons. Et la meilleure d'entre elles est qu'il propose une manière interactive d'écrire un programme, qui permet de tester le résultat de chaque instruction. Toutefois, l'interpréteur a aussi un défaut : le code que vous saisissez est effacé à la fermeture de la fenêtre. Or, nous commençons à être capables de rédiger des programmes relativement complexes, même si vous ne vous en rendez pas encore compte. Dans ces conditions, devoir réécrire le code entier de son programme à chaque fois qu'on ouvre l'interpréteur de commandes est assez lourd.

La solution ? Mettre notre code dans un fichier que nous pourrons lancer à volonté, comme un véritable programme !

Comme je l'ai dit au début de ce chapitre, il est grand temps que je vous présente cette possibilité. Mais on ne dit pas adieu à l'interpréteur de commandes pour autant. On lui dit juste au revoir pour cette fois… on le retrouvera bien assez tôt, la possibilité de tester le code à la volée est vraiment un atout pour apprendre le langage.

Emprisonnons notre programme dans un fichier

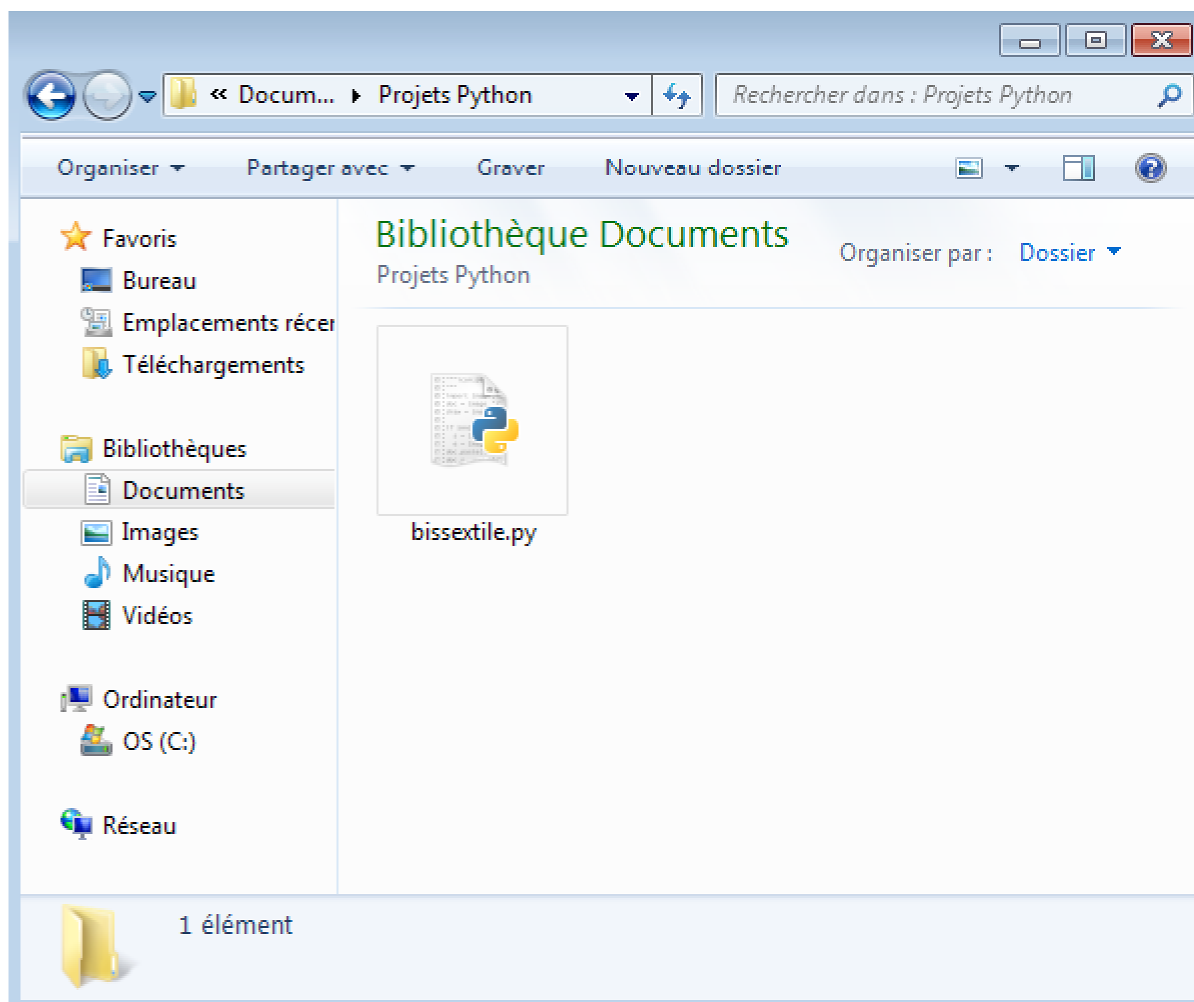
Pour cette démonstration, je reprendrai le code optimisé du programme calculant si une année est bissextile. C'est un petit programme dont l'utilité est certes discutable mais il remplit un but précis, en l'occurrence dire si l'année saisie par l'utilisateur est bissextile ou non : cela suffit pour un premier essai.

Je vous remets le code ici pour que nous travaillions tous sur les mêmes lignes, même si votre version fonctionnera également sans problème dans un fichier, si elle tournait sous l'interpréteur de commandes.

```
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non
annee = input("Saisissez une année : ") # On attend que l'utilisateur fournisse
l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre
if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

C'est à votre tour de travailler maintenant, je vais vous donner des pistes mais je ne vais pas me mettre à votre place, chacun prend ses habitudes en fonction de ses préférences.

Ouvrez un éditeur basique : sous Windows, le bloc-notes est candidat, Wordpad ou Word sont exclus ; sous Linux, vous pouvez utiliser Vim ou Emacs. Insérez le code dans ce fichier et enregistrez-le avec l'extension `.py` (exemple `bissextile.py`), comme à la figure suivante. Cela permettra au système d'exploitation de savoir qu'il doit utiliser Python pour exécuter ce programme (cela est nécessaire sous Windows uniquement).



Sous Linux, vous devrez ajouter dans votre fichier une ligne, tout au début, spécifiant le chemin de l'interpréteur Python (si vous avez déjà rédigé des scripts, en bash par exemple, cette méthode ne vous surprendra pas). La première ligne de votre programme sera :

Remplacez alors le terme `chemin` par le chemin donnant accès à l'interpréteur, par exemple : `/usr/bin/python3.4`. Vous devrez changer le droit d'exécution du fichier avant de l'exécuter comme un script.

Sous Windows, rendez-vous dans le dossier où vous avez enregistré votre fichier `.py`. Vous pouvez faire un double-clic dessus, Windows saura qu'il doit appeler Python grâce à l'extension `.py` et Python reprend la main. Attendez toutefois car il reste quelques petites choses à régler avant de pouvoir exécuter votre programme.

Quelques ajustements

Quand on exécute un programme directement dans un fichier et que le programme contient des accents (et c'est le cas ici), il est nécessaire de préciser à Python l'encodage de ces accents. Je ne vais pas rentrer dans les détails, je vais simplement vous donner une ligne de code qu'il faudra placer tout en haut de votre programme (sous Linux, cette ligne doit figurer juste en-dessous du chemin de l'interpréteur Python).

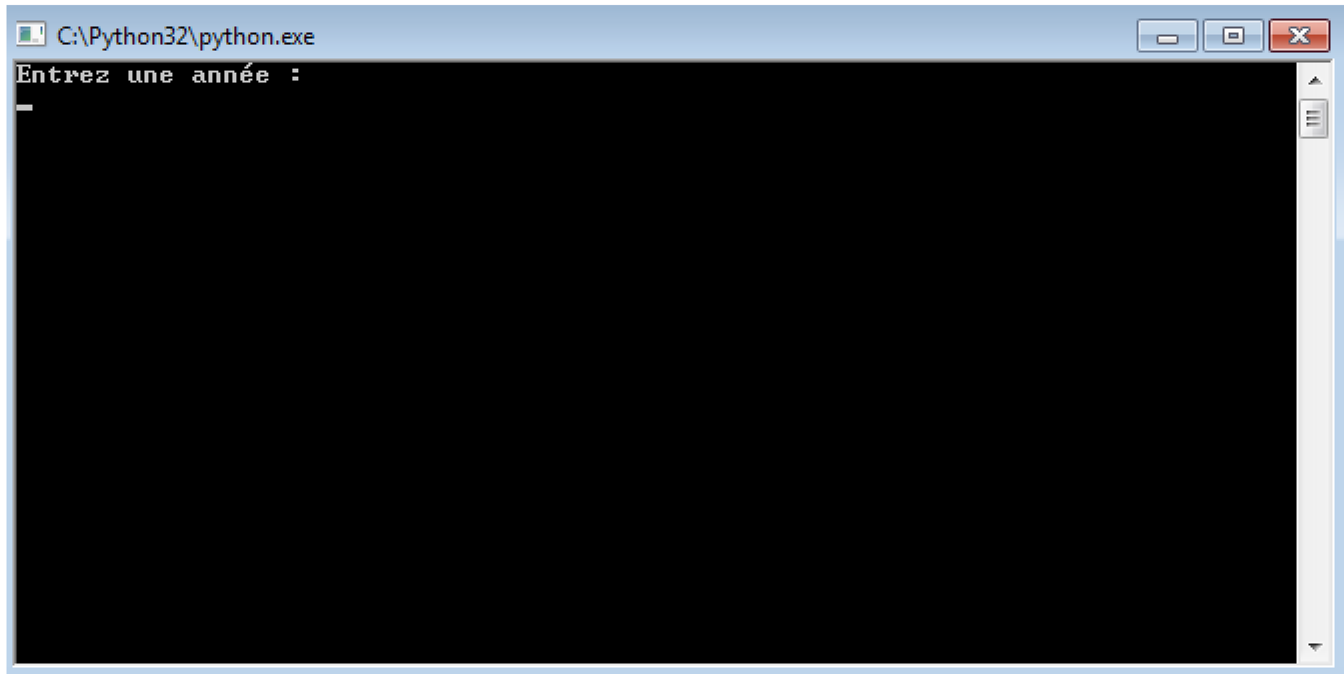
Sous Windows, vous devrez probablement remplacer `ENCODAGE` par « Latin-1 ». Sous Linux, ce sera plus vraissemblablement « utf-8 ». Ce n'est pas le lieu, ni le moment, pour un cours sur les encodages. Utilisez simplement la ligne qui marche chez vous et tout ira bien.

Il est probable, si vous exécutez votre application d'un double-clic, que votre programme se referme immédiatement après vous avoir demandé l'année. En réalité, il fait bel et bien le calcul mais il arrive à la fin du programme en une fraction de seconde et referme l'application, puisqu'elle est finie. Pour pallier cette difficulté, il faut demander à votre programme de se mettre en pause à la fin de son exécution. Vous devrez rajouter une instruction un peu spéciale, un appel système qui marche sous Windows (pas sous Linux). Il faut tout d'abord importer le module `os`. Ensuite, on rajoute l'appel à la fonction `os.system` en lui passant en paramètre la chaîne de caractères « pause » (cela, à la fin de votre programme). Sous Linux, vous pouvez simplement exécuter votre programme dans la console ou, si vous tenez à faire une pause, utilisez par exemple `input` avant la fin de votre programme (pas bien élégant toutefois).

```
# -*-coding:Latin-1 -*
import os # On importe le module os qui dispose de variables
          # et de fonctions utiles pour dialoguer avec votre
          # système d'exploitation
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non
annee = input("Saisissez une année : ") # On attend que l'utilisateur fournisse
l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre
if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

```
# On met le programme en pause pour éviter qu'il ne se referme (Windows)
os.system("pause")
```

Vous pouvez désormais ouvrir votre fichier `bissextile.py`, le programme devrait fonctionner parfaitement (figure suivante).



Quand vous exécutez ce script, que ce soit sous Windows ou Linux, vous faites toujours appel à l'interpréteur Python ! Votre programme n'est pas compilé mais chaque ligne d'instruction est exécutée à la volée par l'interpréteur, le même qui exécutait vos premiers programmes dans l'interpréteur de commandes. La grande différence ici est que Python exécute votre programme depuis le fichier et que donc, si vous souhaitez modifier le programme, il faudra modifier le fichier.

Sachez qu'il existe des éditeurs spécialisés pour Python, notamment Idle qui est installé en même temps que Python (personnellement je ne l'utilise pas). Vous pouvez l'ouvrir avec un clic droit sur votre fichier `.py` et regarder comment il fonctionne, ce n'est pas bien compliqué et vous pouvez même exécuter votre programme depuis ce logiciel. Mais, étant donné que je ne l'utilise pas, je ne vous ferai pas un cours dessus. Si vous avez du mal à utiliser une des fonctionnalités du logiciel, recherchez sur Internet : d'autres tutoriels doivent exister, en anglais dans le pire des cas.

Je viens pour conquérir le monde… et créer mes propres modules

Mes modules à moi

Bon, nous avons vu le plus dur… ça va ? Rassurez-vous, nous n'allons rien faire de compliqué dans cette dernière section. Le plus dur est derrière nous.

Commencez par vous créer un espace de test pour les petits programmes Python que nous allons être amenés à créer, un joli dossier à l'écart de vos photos et musiques. Nous allons créer deux fichiers `.py` dans ce dossier :

- un fichier `multipli.py`, qui contiendra la fonction `table` que nous avons codée au chapitre précédent ;
- un fichier `test.py`, qui contiendra le test d'exécution de notre module.

Vous devriez vous en tirer sans problème. N'oubliez pas de spécifier la ligne précisant l'encodage en tête de vos deux fichiers. Maintenant, voyons le code du fichier `multipli.py`.

```
"""module multipli contenant la fonction table"""
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

On se contente de définir une seule fonction, `table`, qui affiche la table de multiplication choisie. Rien de nouveau jusqu'ici. Si vous vous souvenez des `docstrings`, dont nous avons parlé au chapitre précédent, vous voyez que nous en avons inséré une nouvelle ici, non pas pour commenter une fonction mais bien un module entier. C'est une bonne habitude à prendre quand nos projets deviennent importants.

Voici le code du fichier `test.py`, n'oubliez pas la ligne précisant votre encodage, en tête du fichier.

```
import os
from multipli import *
# test de la fonction table
table(3, 20)
os.system("pause")
```

En le lançant directement, voilà ce qu'on obtient :

```
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
11 * 3 = 33
12 * 3 = 36
13 * 3 = 39
14 * 3 = 42
15 * 3 = 45
16 * 3 = 48
17 * 3 = 51
```

```
18 * 3 = 54
19 * 3 = 57
20 * 3 = 60
Appuyez sur une touche pour continuer...
```

Je ne pense pas avoir grand chose à ajouter. Nous avons vu comment créer un module, il suffit de le mettre dans un fichier. On peut alors l'importer depuis un autre fichier *contenu dans le même répertoire* en précisant le nom du fichier (sans l'extension `.py`). Notre code, encore une fois, n'est pas très utile mais vous pouvez le modifier pour le rendre plus intéressant, vous en avez parfaitement les compétences à présent.

Au moment d'importer votre module, Python va lire (ou créer si il n'existe pas) un fichier `.pyc`. À partir de la version 3.2, ce fichier se trouve dans un dossier `__pycache__`.

Ce fichier est généré par Python et contient le code compilé (ou presque) de votre module. Il ne s'agit pas réellement de langage machine mais d'un format que Python décode un peu plus vite que le code que vous pouvez écrire. Python se charge lui-même de générer ce fichier et vous n'avez pas vraiment besoin de vous en soucier quand vous codez, simplement ne soyez pas surpris.

Faire un test de module dans le module-même

Dans l'exemple que nous venons de voir, nous avons créé deux fichiers, le premier contenant un module, le second testant ledit module. Mais on peut très facilement tester le code d'un module dans le module même. Cela veut dire que vous pourriez exécuter votre module comme un programme à lui tout seul, un programme qui testerait le module écrit dans le même fichier. Voyons voir cela.

Reprenons le code du module `multipli` :

```
"""module multipli contenant la fonction table"""
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Ce module définit une seule fonction, `table`, qu'il pourrait être bon de tester. Oui mais… si nous rajoutons juste en dessous une ligne, par exemple `table(8)`, cette ligne sera exécutée lors de l'importation et donc, dans le programme appelant le module. Quand vous ferez `import multipli`, vous verrez la table de multiplication par 8 s'afficher… hum, il y a mieux.

Heureusement, il y a un moyen très rapide de séparer les éléments du code qui doivent être exécutés lorsqu'on lance le module directement en tant que programme ou lorsqu'on cherche à l'importer. Voici le code de la solution, les explications suivent :

```
"""module multipli contenant la fonction table"""
import os
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
```

```
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
# test de la fonction table
if __name__ == "__main__":
    table(4)
    os.system("pause")
```

N'oubliez pas la ligne indiquant l'encodage !

Voilà. À présent, si vous faites un double-clic directement sur le fichier `multipli.py`, vous allez voir la table de multiplication par 4. En revanche, si vous l'importez, le code de test ne s'exécutera pas. Tout repose en fait sur la variable `__name__`, c'est une variable qui existe dès le lancement de l'interpréteur. Si elle vaut `__main__`, cela veut dire que le fichier appelé est le fichier exécuté. Autrement dit, si `__name__` vaut `__main__`, vous pouvez mettre un code qui sera exécuté si le fichier est lancé directement comme un exécutable.

Prenez le temps de comprendre ce mécanisme, faites des tests si nécessaire, cela pourra vous être utile par la suite.

Les packages

Les modules sont un des moyens de regrouper plusieurs fonctions (et, comme on le verra plus tard, certaines classes également). On peut aller encore au-delà en regroupant des modules dans ce qu'on va appeler des **packages**.

En théorie

Comme je l'ai dit, un package sert à regrouper plusieurs modules. Cela permet de ranger plus proprement vos modules, classes et fonctions dans des emplacements séparés. Si vous voulez y accéder, vous allez devoir fournir un chemin vers le module que vous visez. De ce fait, les risques de conflits de noms sont moins importants et surtout, tout est bien plus ordonné.

Par exemple, imaginons que vous installiez un jour une bibliothèque tierce pour écrire une interface graphique. En s'installant, la bibliothèque ne va pas créer ses dizaines (voire ses centaines) de modules au même endroit. Ce serait un peu désordonné… surtout quand on pense qu'on peut ranger tout cela d'une façon plus claire : d'un côté, on peut avoir les différents objets graphiques de la fenêtre, de l'autres les différents évènements (clavier, souris,…), ailleurs encore les effets graphiques…

Dans ce cas, on va sûrement se retrouver face à un package portant le nom de la bibliothèque. Dans ce package se trouveront probablement d'autres packages, un nommé `evenements`, un autre `objets`, un autre encore `effets`. Dans chacun de ces packages, on pourra trouver soit d'autres packages, soit des modules et dans chacun de ces modules, des fonctions.

Ouf ! Cela nous fait une hiérarchie assez complexe non ? D'un autre côté, c'est tout l'intérêt. Concrètement, pour utiliser cette bibliothèque, on n'est pas obligé de connaître tous ses packages, modules et fonctions (heureusement d'ailleurs !) mais juste ceux dont on a réellement besoin.

En pratique

En pratique, les packages sont… des répertoires ! Dedans peuvent se trouver d'autres répertoires (d'autres packages) ou des fichiers (des modules).

Exemple de hiérarchie

Pour notre bibliothèque imaginaire, la hiérarchie des répertoires et fichiers ressemblerait à cela :

- Un répertoire du nom de la bibliothèque contenant :
    - un répertoire `evenements` contenant :
        - un module `clavier` ;
        - un module `souris` ;
        - …
    - un répertoire `effets` contenant différents effets graphiques ;
    - un répertoire `objets` contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menus…).

Importer des packages

Si vous voulez utiliser, dans votre programme, la bibliothèque fictive que nous venons de voir, vous avez plusieurs moyens qui tournent tous autour des mots clés `from` et `import` :

Cette ligne importe le package contenant la bibliothèque. Pour accéder aux sous-packages, vous utiliserez un point « . » afin de modéliser le chemin menant au module ou à la fonction que vous voulez utiliser :

```
nom_bibliotheque.evenements # Pointe vers le sous-package evenements
nom_bibliotheque.evenements.clavier # Pointe vers le module clavier
```

Si vous ne voulez importer qu'un seul module (ou qu'une seule fonction) d'un package, vous utiliserez une syntaxe similaire, assez intuitive :

```
from nom_bibliotheque.objets import bouton
```

En fonction des besoins, vous pouvez décider d'importer tout un package, un sous-package, un sous-sous-package… ou bien juste un module ou même une seule fonction. Cela dépendra de vos besoins.

Créer ses propres packages

Si vous voulez créer vos propres packages, commencez par créer, dans le même dossier que votre programme Python, un répertoire portant le nom du package.

Dans ce répertoire, vous pouvez soit :

- mettre vos modules, vos fichiers à l'extension `.py` ;
- créer des sous-packages de la même façon, en créant un répertoire dans votre package.

Ne mettez pas d'espaces dans vos noms de packages et évitez aussi les caractères spéciaux. Quand vous les utilisez dans vos programmes, ces noms sont traités comme des noms de variables et ils doivent donc obéir aux mêmes règles de nommage.

Le fichier d'initialisation

En Python, vous trouverez souvent le fichier d'initialisation de package `__init__.py` dans un répertoire destiné à devenir un package. Ce fichier est optionnel depuis la version 3.3 de Python. Vous n'êtes pas obligé de le créer mais vous pouvez y mettre du code d'initialisation pour votre package. Je ne vais pas rentrer dans le détail ici (vous avez déjà beaucoup de choses à retenir), mais sachez que ce code d'initialisation est appelé quand vous importez votre package.

Un dernier exemple

Voici un dernier exemple, que vous pouvez cette fois faire en même temps que moi pour vous assurer que cela fonctionne.

Dans votre répertoire de code, là où vous mettez vos exemples Python, créez un fichier `.py` que vous appelerez `test_package.py`.

Créez dans le même répertoire un dossier `package`. Dedans, créez un fichier `fonctions.py` dans lequel vous recopierez votre fonction `table`.

Dans votre fichier `test_package.py`, si vous voulez importer votre fonction `table`, vous avez plusieurs solutions :

```
from package.fonctions import table
table(5) # Appel de la fonction table
# Ou ...
import package.fonctions
package.fonctions.table(5)  # Appel de la fonction table
```

Voilà. Il reste bien des choses à dire sur les packages mais je crois que vous avez vu l'essentiel. Cette petite explication révélera son importance quand vous aurez à construire des programmes assez volumineux. Évitez de tout mettre dans un seul module sans chercher à hiérarchiser, profitez de cette possibilité offerte par Python.

En résumé

- On peut écrire les programmes Python dans des fichiers portant l'extension `.py`.
- On peut créer des fichiers contenant des **modules** pour séparer le code.
- On peut créer des répertoires contenant des **packages** pour hiérarchiser un programme.

Les exceptions

Dans ce chapitre, nous aborderons le dernier concept que je considère comme indispensable avant d'attaquer la partie sur la Programmation Orientée Objet, j'ai nommé « les exceptions ».

Comme vous allez le voir, il s'agit des erreurs que peut rencontrer Python en exécutant votre programme. Ces erreurs peuvent être interceptées très facilement et c'est même, dans certains cas, indispensable.

Cependant, il ne faut pas tout intercepter non plus : si Python envoie une erreur, c'est qu'il y a une raison. Si vous ignorez une erreur, vous risquez d'avoir des résultats très étranges dans votre programme.

À quoi cela sert-il ?

Nous avons déjà été confrontés à des erreurs dans nos programmes, certaines que j'ai volontairement provoquées, mais la plupart que vous avez dû rencontrer si vous avez testé un minimum des instructions dans l'interpréteur. Quand Python rencontre une erreur dans votre code, il **lève une exception**. Sans le savoir, vous avez donc déjà vu des exceptions levées par Python :

```
>>> # Exemple classique : test d'une division par zéro
>>> variable = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Attardons-nous sur la dernière ligne. Nous y trouvons deux informations :

- `ZeroDivisionError` : le type de l'exception ;
- `int division or modulo by zero` : le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Python lève donc des exceptions quand il trouve une erreur, soit dans le code (une erreur de syntaxe, par exemple), soit dans l'opération que vous lui demandez de faire.

Notez qu'à l'instar des variables, on trouve différents types d'exceptions que Python va utiliser en fonction de la situation. Le type d'exception `ValueError`, notamment, pourra être levé par Python face à diverses erreurs de « valeurs ». Dans ce cas, c'est donc le message qui vous indique plus clairement le problème. Nous verrons dans la prochaine partie, consacrée à la Programmation Orientée Objet, ce que sont réellement ces types d'exceptions.

Bon, c'est bien joli d'avoir cette exception. On voit le fichier et la ligne à laquelle s'est produite l'erreur (très pratique quand on commence à travailler sur un projet) et on a une indication sur le problème qui suffit en général à le régler. Mais Python permet quelque chose de bien plus pratique.

Admettons que certaines erreurs puissent être provoquées par l'utilisateur. Par exemple, on demande à l'utilisateur de saisir au clavier un entier et il tape une chaîne de caractères… problème. Nous avons déjà rencontré cette situation : souvenez-vous du programme `bissextile`.

```
annee = input() # On demande à l'utilisateur de saisir l'année
annee = int(annee) # On essaye de convertir l'année en un entier
```

Je vous avais dit que si l'utilisateur fournissait ici une valeur impossible à convertir en entier (une lettre par exemple), le programme plantait. En fait, il lève une exception et Python arrête l'exécution du programme. Si vous testez le programme en faisant un double-clic directement dans l'explorateur, il va se fermer tout de suite (en fait, il affiche bel et bien l'erreur mais se referme aussitôt).

Dans ce cas, et dans d'autres cas similaires, Python permet de tester un extrait de code. S'il ne renvoie aucune erreur, Python continue. Sinon, on peut lui demander d'exécuter une autre action (par exemple, redemander à l'utilisateur de saisir l'année). C'est ce que nous allons voir ici.

Forme minimale du bloc try

On va parler ici de bloc `try`. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc. Sans plus attendre, voici la syntaxe :

```
try:
    # Bloc à essayer
except:
    # Bloc qui sera exécuté en cas d'erreur
```

Dans l'ordre, nous trouvons :

- Le mot-clé `try` suivi des deux points « : » (*try* signifie « essayer » en anglais).
- Le bloc d'instructions à essayer.
- Le mot-clé `except` suivi, une fois encore, des deux points « : ». Il se trouve au même niveau d'indentation que le `try`.
- Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Reprenons notre test de conversion en enfermant dans un bloc `try` l'instruction susceptible de lever une exception.

```
annee = input()
try: # On essaye de convertir l'année en entier
    annee = int(annee)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes pour la variable `annee`, comme « 2010 » ou « annee2010 ».

Dans le titre de cette section, j'ai parlé de *forme minimale* et ce n'est pas pour rien. D'abord, il va de soi que vous ne pouvez intégrer cette solution directement dans votre code. En effet, si l'utilisateur saisit une année impossible à convertir, le système affiche certes une erreur mais finit par planter (puisque l'année, au final, n'a pas été convertie). Une des solutions envisageables est d'attribuer une valeur par défaut à l'année, en cas d'erreur, ou de redemander à l'utilisateur de saisir l'année.

Ensuite et surtout, cette méthode est assez grossière. Elle essaye une instruction et intercepte *n'importe quelle* exception liée à cette instruction. Ici, c'est acceptable car nous n'avons pas énormément d'erreurs possibles sur cette instruction. Mais c'est une mauvaise habitude à prendre. Voici une manière plus élégante et moins dangereuse.

Forme plus complète

Nous allons apprendre à compléter notre bloc `try`. Comme je l'ai indiqué plus haut, la forme minimale est à éviter pour plusieurs raisons.

D'abord, elle ne différencie pas les exceptions qui pourront être levées dans le bloc `try`. Ensuite, Python peut lever des exceptions qui ne signifient pas nécessairement qu'il y a eu une erreur.

Exécuter le bloc except pour un type d'exception précis

Dans l'exemple que nous avons vu plus haut, on ne pense qu'à un type d'exceptions susceptible d'être levé : le type `ValueError`, qui trahirait une erreur de conversion. Voyons un autre exemple :

```
try:
    resultat = numerateur / denominateur
except:
    print("Une erreur est survenue... laquelle ?")
```

Ici, plusieurs erreurs sont susceptibles d'intervenir, chacune levant une exception différente.

- `NameError` : l'une des variables `numerateur` ou `denominateur` n'a pas été définie (elle n'existe pas). Si vous essayez dans l'interpréteur l'instruction `print(numerateur)` alors que vous n'avez pas défini la variable `numerateur`, vous aurez la même erreur.
- `TypeError` : l'une des variables `numerateur` ou `denominateur` ne peut diviser ou être divisée (les chaînes de caractères ne peuvent être divisées, ni diviser d'autres types, par exemple). Cette exception est levée car vous utilisez l'opérateur de division « / » sur des types qui ne savent pas quoi en faire.
- `ZeroDivisionError` : encore elle ! Si `denominateur` vaut 0, cette exception sera levée.

Cette énumération n'est pas une liste exhaustive de toutes les exceptions qui peuvent être levées à l'exécution de ce code. Elle est surtout là pour vous montrer que plusieurs erreurs peuvent se produire sur une instruction (c'est encore plus flagrant sur un bloc constitué de plusieurs instructions) et que la forme minimale intercepte toutes ces erreurs sans les distinguer, ce qui peut être problématique dans certains cas.

Tout se joue sur la ligne du `except`. Entre ce mot-clé et les deux points, vous pouvez préciser le type de l'exception que vous souhaitez traiter.

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
```

Ce code ne traite que le cas où une exception `NameError` est levée. On peut intercepter les autres types d'exceptions en créant d'autres blocs `except` à la suite :

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

C'est mieux non ?

Allez un petit dernier !

On peut capturer l'exception et afficher son message grâce au mot-clé `as` que vous avez déjà vu dans un autre contexte (si si, rappelez-vous de l'importation de modules).

```
try:
    # Bloc de test
except type_de_l_exception as exception_retournee:
    print("Voici l'erreur :", exception_retournee)
```

Dans ce cas, une variable `exception_retournee` est créée par Python si une exception du type précisé est levée dans le bloc `try`.

Je vous conseille de *toujours* préciser un type d'exceptions après `except` (sans nécessairement capturer l'exception dans une variable, bien entendu). D'abord, vous ne devez pas utiliser `try` comme une méthode miracle pour tester n'importe quel bout de code. Il est important que vous gardiez le maximum de contrôle sur votre code. Cela signifie que, si une erreur se produit, vous devez être capable de l'anticiper. En pratique, vous n'irez pas jusqu'à tester si une variable quelconque existe bel et bien, il faut faire un minimum confiance à son code. Mais si vous êtes en face d'une division et que le dénominateur pourrait avoir une valeur de 0, placez la division dans un bloc `try` et précisez, après le `except`, le type de l'exception qui risque de se produire (`ZeroDivisionError` dans cet exemple).

Si vous adoptez la forme minimale (à savoir `except` sans préciser un type d'exception qui pourrait se produire sur le bloc `try`), toutes les exceptions seront traitées de la même façon. Et même si `exception = erreur` la plupart du temps, ce n'est pas toujours le cas. Par exemple, Python lève une exception quand vous voulez fermer votre programme avec le raccourci `CTRL` + `C`. Ici vous ne

voyez peut-être pas le problème mais si votre bloc `try` est dans une boucle, vous ne pourrez pas arrêter votre programme avec `CTRL` + `C`, puisque l'exception sera traitée par votre `except`.

Je vous conseille donc de toujours préciser un type d'exception possible après votre `except`. Vous pouvez bien entendu faire des tests dans l'interpréteur de commandes Python pour reproduire l'exception que vous voulez traiter et ainsi connaître son type.

Les mots-clés `else` et `finally`

Ce sont deux mots-clés qui vont nous permettre de construire un bloc `try` plus complet.

Le mot-clé `else`

Vous avez déjà vu ce mot-clé et j'espère que vous vous en rappelez. Dans un bloc `try`, `else` va permettre d'exécuter une action si aucune erreur ne survient dans le bloc. Voici un petit exemple :

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

Dans les faits, on utilise assez peu `else`. La plupart des codeurs préfère mettre la ligne contenant le `print` directement dans le bloc `try`. Pour ma part, je trouve que c'est important de distinguer entre le bloc `try` et ce qui s'effectue ensuite. La ligne du `print` ne produira vraisemblablement aucune erreur, inutile de la placer dans le bloc `try`.

Le mot-clé `finally`

`finally` permet d'exécuter du code après un bloc `try`, *quelle que soit le résultat de l'exécution dudit bloc*. La syntaxe est des plus simples :

```
try:
    # Test d'instruction(s)
except TypeDInstruction:
    # Traitement en cas d'erreur
finally:
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

Est-ce que cela ne revient pas au même si on met du code juste après le bloc ?

Pas tout à fait. Le bloc `finally` est exécuté dans tous les cas de figures. Quand bien même Python trouverait une instruction `return` dans votre bloc `except` par exemple, il exécutera le bloc `finally`.

Un petit bonus : le mot-clé `pass`

Il peut arriver, dans certains cas, que l'on souhaite tester un bloc d'instructions… mais ne rien faire en cas d'erreur. Toutefois, un bloc `try` ne peut être seul.

```
>>> try:
...     1/0
...
  File "<stdin>", line 3
    ^
SyntaxError: invalid syntax
```

Il existe un mot-clé que l'on peut utiliser dans ce cas. Son nom est `pass` et sa syntaxe est très simple d'utilisation :

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

Je ne vous encourage pas particulièrement à utiliser ce mot-clé mais il existe, et vous le savez à présent.

`pass` n'est pas un mot-clé propre aux exceptions : on peut également le trouver dans des conditions ou dans des fonctions que l'on souhaite laisser vides.

Voilà, nous avons vu l'essentiel. Il nous reste à faire un petit point sur les assertions et à voir comment lever une exception (ce sera très rapide).

Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs `try … except`.

Voyons comment cela fonctionne : nous allons pour l'occasion découvrir un nouveau mot-clé (encore un), `assert`. Sa syntaxe est la suivante :

Si le test renvoie `True`, l'exécution se poursuit normalement. Sinon, une exception `AssertionError` est levée.

Voyons un exemple :

```
>>> var = 5
>>> assert var == 5
>>> assert var == 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>>
```

Comme vous le voyez, la ligne 2 s'exécute sans problème et ne lève aucune exception. On teste en effet si `var == 5`. C'est le cas, le test est donc vrai, aucune exception n'est levée.

À la ligne suivante, cependant, le test est `var == 8`. Cette fois, le test est faux et une exception du type `AssertionError` est levée.

À quoi cela sert-il, concrètement ?

Dans le programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur ne saisit pas une année inférieure ou égale à 0 par exemple. Avec les assertions, c'est très facile à faire :

```
annee = input("Saisissez une année supérieure à 0 :")
try:
    annee = int(annee) # Conversion de l'année
    assert annee > 0
except ValueError:
    print("Vous n'avez pas saisi un nombre.")
except AssertionError:
    print("L'année saisie est inférieure ou égale à 0.")
```

Lever une exception

Hmmm… je vois d'ici les mines sceptiques (non non, ne vous cachez pas !). Vous vous demandez probablement pourquoi vous feriez le boulot de Python en levant des exceptions. Après tout, votre travail, c'est en théorie d'éviter que votre programme plante.

Parfois, cependant, il pourra être utile de lever des exceptions. Vous verrez tout l'intérêt du concept quand vous créerez vos propres classes… mais ce n'est pas pour tout de suite. En attendant, je vais vous donner la syntaxe et vous pourrez faire quelques tests, vous verrez de toute façon qu'il n'y a rien de compliqué.

On utilise un nouveau mot-clé pour lever une exception… le mot-clé `raise`.

```
raise TypeDeLException("message à afficher")
```

Prenons un petit exemple, toujours autour de notre programme `bissextile`. Nous allons lever une exception de type `ValueError` si l'utilisateur saisit une année négative ou nulle.

```
annee = input() # L'utilisateur saisit l'année
try:
    annee = int(annee) # On tente de convertir l'année
    if annee<=0:
        raise ValueError("l'année saisie est négative ou nulle")
except ValueError:
    print("La valeur saisie est invalide (l'année est peut-être négative).")
```

Ce que nous venons de faire est réalisable sans l'utilisation des exceptions mais c'était surtout pour vous montrer la syntaxe dans un véritable contexte. Ici, on lève une exception que l'on intercepte immédiatement ou presque, l'intérêt est donc limité. Bien entendu, la plupart du temps ce n'est pas le cas.

Il reste des choses à découvrir sur les exceptions, mais on en a assez fait pour ce chapitre et cette partie. Je ne vous demande pas de connaître toutes les exceptions que Python est amené à utiliser (certaines d'entre elles pourront d'ailleurs n'exister que dans certains modules). En revanche, vous devez être

capables de savoir, grâce à l'interpréteur de commandes, quelles exceptions peuvent être levées par Python dans une situation donnée.

En résumé

- On peut intercepter les erreurs (ou exceptions) levées par notre code grâce aux blocs `tryexcept`.
- La syntaxe d'une assertion est `assert test`.
- Les assertions lèvent une exception `AssertionError` si le test échoue.
- On peut lever une exception grâce au mot-clé `raise` suivi du type de l'exception.

TP : tous au ZCasino

L'heure de vérité a sonné ! C'est dans ce premier TP que je vais faire montre de ma cruauté sans limite en vous lâchant dans la nature… ou presque. Ce n'est pas tout à fait votre premier TP, dans le sens où le programme du chapitre 4, sur les conditions, constituait votre première expérience en la matière. Mais, à ce moment-là, nous n'avions pas fait un programme très… récréatif.

Cette fois, nous allons nous atteler au développement d'un petit jeu de casino. Vous trouverez le détail de l'énoncé plus bas, ainsi que quelques conseils pour la réalisation de ce TP.

Si, durant ce TP, vous sentez que certaines connaissances vous manquent, revenez en arrière ; prenez tout votre temps, on n'est pas pressé !

Notre sujet

Dans ce chapitre, nous allons essayer de faire un petit programme que nous appellerons ZCasino. Il s'agira d'un petit jeu de roulette très simplifié dans lequel vous pourrez miser une certaine somme et gagner ou perdre de l'argent (telle est la fortune, au casino !). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un premier TP. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

- Le joueur mise sur un numéro compris entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser.
- La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, nous ne reprendrons pas tous ces détails « matériels » mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant.
- Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le croupier lui remet 3 fois la somme misée.
- Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50 % de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement misée avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le

troisième cas qu'il perd la somme misée. On utilisera pour devise le dollar $ à la place de l'euro pour des raisons d'encodage sous la console Windows.

Organisons notre projet

Pour ce projet, nous n'allons pas écrire de module. Nous allons utiliser ceux de Python, qui sont bien suffisants pour l'instant, notamment celui permettant de générer de l'aléatoire, que je vais présenter plus bas. En attendant, ne vous privez quand même pas de créer un répertoire et d'y mettre le fichier `ZCasino.py`, tout va se jouer ici.

Vous êtes capables d'écrire le programme `ZCasino` tel qu'expliqué dans la première partie sans difficulté… sauf pour générer des nombres aléatoires. Python a dédié tout un module à la génération d'éléments pseudo-aléatoires, le module `random`.

Le module `random`

Dans ce module, nous allons nous intéresser particulièrement à la fonction `randrange` qui peut s'utiliser de deux manières :

- en ne précisant qu'un paramètre (`randrange(6)` renvoie un nombre aléatoire compris entre 0 et 5) ;
- en précisant deux paramètres (`randrange(1, 7)` : renvoie un nombre aléatoire compris entre 1 et 6, ce qui est utile, par exemple, pour reproduire une expérience avec un dé à six faces).

Pour tirer un nombre aléatoire compris entre 0 et 49 et simuler ainsi l'expérience du jeu de la roulette, nous allons donc utiliser l'instruction `randrange(50)`.

Il existe d'autres façons d'utiliser `randrange` mais nous n'en aurons pas besoin ici et je dirais même que, pour ce programme, seule la première utilisation vous sera utile.

N'hésitez pas à faire des tests dans l'interpréteur de commandes (vous n'avez pas oublié où c'est, hein ?) et essayez plusieurs syntaxes de la fonction `randrange`. Je vous rappelle qu'elle se trouve dans le module `random`, n'oubliez pas de l'importer.

Arrondir un nombre

Vous l'avez peut-être bien noté, dans l'explication des règles je spécifiais que si le joueur misait sur la bonne couleur, il obtenait 50% de sa mise. Oui mais… c'est quand même mieux de travailler avec des entiers. Si le joueur mise 3$, par exemple, on lui rend 1,5$. C'est encore acceptable mais, si cela se poursuit, on risque d'arriver à des nombres flottants avec beaucoup de chiffres après la virgule. Alors autant arrondir au nombre supérieur. Ainsi, si le joueur mise 3$, on lui rend 2$. Pour cela, on va utiliser une fonction du module `math` nommée `ceil`. Je vous laisse regarder ce qu'elle fait, il n'y a rien de compliqué.

À vous de jouer

Voilà, vous avez toutes les clés en main pour coder ce programme. Prenez le temps qu'il faut pour y arriver, ne vous ruez pas sur la correction, le but du TP est que vous appreniez à coder vous-mêmes un programme… et celui-ci n'est pas très difficile. Si vous avez du mal, morcelez le programme, ne codez pas tout d'un coup. Et n'hésitez pas à passer par l'interpréteur pour tester des fonctionnalités : c'est réellement une chance qui vous est donnée, ne la laissez pas passer.

À vous de jouer !

Correction !

C'est encore une fois l'heure de comparer nos versions. Et, une fois encore, il est très peu probable que vous ayez un code identique au mien. Donc si le vôtre fonctionne, je dirais que c'est l'essentiel. Si vous vous heurtez à des difficultés insurmontables, la correction est là pour vous aider.

… ATTENTION… voici… la solution !

```
# Ce fichier abrite le code du ZCasino, un jeu de roulette adapté
import os
from random import randrange
from math import ceil
# Déclaration des variables de départ
argent = 1000 # On a 1000 $ au début du jeu
continuer_partie = True # Booléen qui est vrai tant qu'on doit
                        # continuer la partie
print("Vous vous installez à la table de roulette avec", argent, "$.")
while continuer_partie: # Tant qu'on doit continuer la partie
    # on demande à l'utilisateur de saisir le nombre sur
    # lequel il va miser
    nombre_mise = -1
    while nombre_mise < 0 or nombre_mise > 49:
        nombre_mise = input("Tapez le nombre sur lequel vous voulez miser (entre 0
et 49) : ")
        # On convertit le nombre misé
        try:
            nombre_mise = int(nombre_mise)
        except ValueError:
            print("Vous n'avez pas saisi de nombre")
            nombre_mise = -1
            continue
        if nombre_mise < 0:
            print("Ce nombre est négatif")
        if nombre_mise > 49:
            print("Ce nombre est supérieur à 49")
    # À présent, on sélectionne la somme à miser sur le nombre
    mise = 0
    while mise <= 0 or mise > argent:
        mise = input("Tapez le montant de votre mise : ")
        # On convertit la mise
        try:
            mise = int(mise)
        except ValueError:
            print("Vous n'avez pas saisi de nombre")
            mise = -1
            continue
```

```
        if mise <= 0:
            print("La mise saisie est négative ou nulle.")
        if mise > argent:
            print("Vous ne pouvez miser autant, vous n'avez que", argent, "$")
    # Le nombre misé et la mise ont été sélectionnés par
    # l'utilisateur, on fait tourner la roulette
    numero_gagnant = randrange(50)
    print("La roulette tourne... ... et s'arrête sur le numéro", numero_gagnant)
    # On établit le gain du joueur
    if numero_gagnant == nombre_mise:
        print("Félicitations ! Vous obtenez", mise * 3, "$ !")
        argent += mise * 3
    elif numero_gagnant % 2 == nombre_mise % 2: # ils sont de la même couleur
        mise = ceil(mise * 0.5)
        print("Vous avez misé sur la bonne couleur. Vous obtenez", mise, "$")
        argent += mise
    else:
        print("Désolé l'ami, c'est pas pour cette fois. Vous perdez votre mise.")
        argent -= mise
    # On interrompt la partie si le joueur est ruiné
    if argent <= 0:
        print("Vous êtes ruiné ! C'est la fin de la partie.")
        continuer_partie = False
    else:
        # On affiche l'argent du joueur
        print("Vous avez à présent", argent, "$")
        quitter = input("Souhaitez-vous quitter le casino (o/n) ? ")
        if quitter == "o" or quitter == "O":
            print("Vous quittez le casino avec vos gains.")
            continuer_partie = False
# On met en pause le système (Windows)
os.system("pause")
```

Encore une fois, n'oubliez pas la ligne spécifiant l'encodage si vous voulez éviter les surprises.

Une petite chose qui pourrait vous surprendre est la construction des boucles pour tester si le joueur a saisi une valeur correcte (quand on demande à l'utilisateur de taper un nombre entre 0 et 49 par exemple, il faut s'assurer qu'il l'a bien fait). C'est assez simple en vérité : on attend que le joueur saisisse un nombre. Si le nombre n'est pas valide, on demande à nouveau au joueur de saisir ce nombre. J'en ai profité pour utiliser le concept des exceptions afin de vérifier que l'utilisateur saisit bien un nombre. Comme vous l'avez vu, si ce n'est pas le cas, on affiche un message d'erreur. La valeur de la variable qui contient le nombre est remise à `-1` (c'est-à-dire une valeur qui indique à la boucle que nous n'avons toujours pas obtenu de l'utilisateur une valeur valide) et on utilise le mot-clé `continue` pour passer les autres instructions du bloc (sans quoi vous verriez s'afficher un autre message indiquant que le nombre saisi est négatif… c'est plus pratique ainsi). De cette façon, si l'utilisateur fournit une donnée inconvertible, le jeu ne plante pas et lui redemande tout simplement de taper une valeur valide.

La boucle principale fonctionne autour d'un booléen. On utilise une variable nommée `continuer_partie` qui vaut « vrai » tant qu'on doit continuer la partie. Une fois que la partie doit s'interrompre, elle passe à « faux ». Notre boucle globale, qui gère le déroulement de la partie, travaille sur ce booléen ; par conséquent, dès qu'il passe à la valeur « faux », la boucle s'interrompt et le programme se met en pause. Tout le reste, vous devriez le comprendre sans aide, les commentaires sont

là pour vous expliquer. Si vous avez des doutes, vous pouvez tester les lignes d'instructions problématiques dans votre interpréteur de commandes Python : encore une fois, n'oubliez pas cet outil.

Et maintenant ?

Prenez bien le temps de lire ma version et surtout de modifier la vôtre, si vous êtes arrivés à une version qui fonctionne bien ou qui fonctionne presque. Ne mettez pas ce projet à la corbeille sous prétexte que nous avons fini de le coder et qu'il marche. On peut toujours améliorer un projet et celui-ci ne fait évidemment pas exception. Vous trouverez probablement de nouveaux concepts, dans la suite de ce livre, qui pourront être utilisés dans le programme de `ZCasino`.

Eh bien, c'en est fini des concepts de base. Dès la prochaine partie, on s'attaque à la POO, la Programmation Orientée Objet, un concept franchement fascinant et très puissant en Python. Vous allez surtout apprendre à manier de nouveaux types de données, notamment les listes, les dictionnaires, les fichiers... ça donne envie non ?

Quiz

Question 1

Après ces instructions, de quel type est la variable `c` ?

```
a = 8
b = 3
c = a / b
```

- `int` (entier)
- `float` (flottant)
- `str` (chaîne de caractères)

Question 2

Quelle est la variable de type `str` (chaîne de caractères) parmi les choix suivants ?

- `3`
- `3.1`
- `"3"`

Question 3

Quelle est la différence entre entrer une variable dans l'interpréteur interactif et utiliser la fonction `print` ?

- Aucune
- Dans l'interpréteur interactif, toutes les variables apparaissent entourées de guillemets.
- La fonction `print` est dédiée à l'affichage, l'interpréteur au débuggage.

Question 4

De quoi doit être composée une condition au minimum ?

- D'un bloc `if`
- D'un bloc `if` et `elif`
- D'un bloc `if` et `else`

Question 5

Considérant les instructions ci-dessous, si `variable` vaut `2.8`, quel va être le résultat obtenu ?

```
if variable >= 3:
```

```
    print("1")
elif variable < -1:
    print("2")
else:
    print("3")
```

- Afficher 1.
- Afficher 2.
- Afficher 3.
- N'afficher rien.

Question 6

Considérant le prédicat combiné ci-dessous, dans quel cas sera-t-il `True` (vrai) ?

```
predicat_a and predicat_b
```

- `predicat_a` est vrai, peu importe `predicat_b`.
- `predicat_b` est vrai, peu importe `predicat_a`.
- L'un d'eux est vrai, peu importe l'autre.
- `predicat_a` et `predicat_b` sont tous deux vrais.

Question 7

Comment Python identifie-t-il les instructions formant un bloc (par exemple à l'intérieur d'une condition) ?

- Grâce à l'indentation
- Grâce aux accolades (`{}`) entourant le bloc
- Grâce aux deux points en début de bloc

Question 8

Quel est l'avantage de la boucle `while` sur la boucle `for` ?

- Aucun.
- Elle est préférable pour parcourir des séquences.
- Elle fait la même chose en moins de lignes de code.
- Elle crée rarement de boucle infinie.
- Elle est plus utile pour vérifier une condition.

Question 9

Quel est l'avantage de la boucle `for` sur la boucle `while` ?

- Elle est préférable pour parcourir des séquences.
- Elle est plus utile pour vérifier une condition.
- C'est l'unique façon de parcourir des séquences.

Question 10

Quelle est la différence entre le mot-clé `break` et `continue` ?

- `break` interrompt la boucle immédiatement alors que `continue` passe au prochain tour de boucle.
- `continue` interrompt la boucle immédiatement alors que `break` passe au prochain tour de boucle.
- Les deux mot-clés font exactement la même chose en interrompant la boucle immédiatement.

Question 11

Sachant la définition de fonction ci-dessous, quel est l'appel **INVALIDE** parmi les choix suivants ?

```
def table(nombre, maximum=10):
```

- `table(5, 20)`
- `table(12, maximum=5)`
- `table(8)`
- `table(maximum=15, nombre=4)`
- `table(7, entier=30)`

Question 12

Quelle est l'utilité des fonctions **lambda** par rapport aux fonctions définies par `def` ?

- Elles s'exécutent plus rapidement.
- On peut en créer avec une syntaxe très légère.
- Elles permettent des fonctionnalités inaccessibles aux fonctions définies par `def`.

Question 13

Qu'est-ce qu'un module en Python ?

- Un fichier contenant du code Python sans extension particulière.
- Un fichier contenant du code Python avec l'extension *.py*
- Un répertoire contenant des fichiers Python

Question 14

Si vous entrez l'instruction `import azerty`, les fonctions du module `azerty` seront ... ?

- Directement accessibles en entrant simplement leur nom.
- Accessibles en préfixant leur nom par `azerty.`
- Accessibles en préfixant leur nom par un simple point ("`.`").

Question 15

En utilisant l'instruction `from ... import *`, que peut-on importer ?

- Seulement des fonctions d'un module
- Seulement des modules d'un package
- Tout ce que contient un module ou package

Question 16

Qu'est-ce qui caractérise, au minimum, un package Python ?

- Un répertoire simple
- Un répertoire avec, au minimum, un fichier *__init__.py* dedans
- Un répertoire avec un fichier *__init__.py* et au moins un autre module ou package

Question 17

Quels sont les mot-clés minimums pour capturer une exception ?

- `try` et `catch`
- `try` et `except`
- `try` et `else`

Question 18

Quel mot-clé est utilisé pour lever une exception ?

- `throw`
- `raise`

- `try`

Question 19

Dans quel cas le bloc `finally` est-il exécuté ?

- Quand aucune exception ne se produit dans le bloc `try`.
- Quand une exception se produit dans le bloc `try`.
- Dans tous les cas.

Question 20

Dans quel cas l'instruction ci-dessous lèvera une exception ?

```
annee = int(entree)
```

- La variable `annee` n'existe pas.
- La variable `entree` est une chaîne de caractères.
- La variable `entree` ne peut être convertie en nombre.