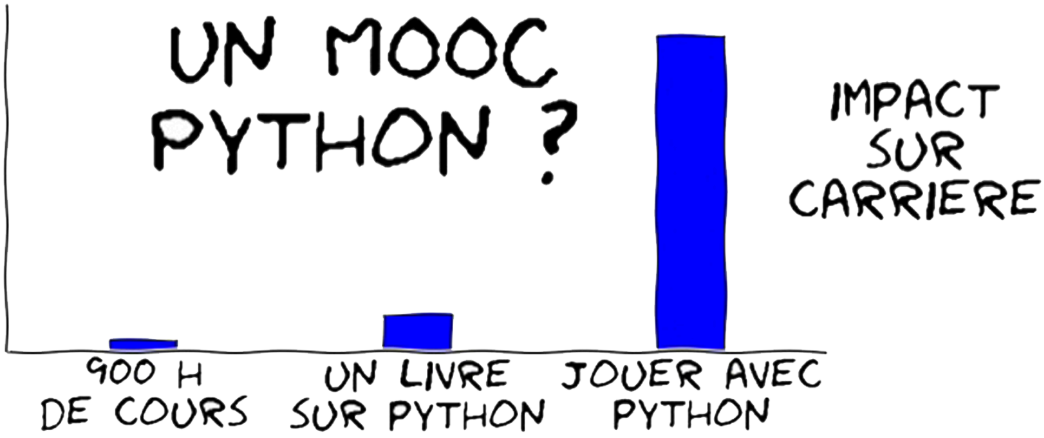
python™ 3

Des fondamentaux au concepts avancés du langage

Session 2 - 17 septembre 2018

Thierry Parmentelat Arnaud Legout



https ://www.fun-mooc.fr

Table des matières

Chapitre 1

Introduction au MOOC et aux outils Python

1.1 w1-s1-c1-versions-python

Versions de Python

Version de référence : Python-3.6

Comme on l'indique dans la vidéo, la version de Python qui a servi de référence pour le MOOC est la version 3.6, c'est notamment avec cette version que l'on a tourné les vidéos.

Versions plus anciennes

Certaines précautions sont à prendre si vous utilisez une version plus ancienne :

Python-3.5

Si vous préférez utiliser python-3.5, la différence la plus visible pour vous apparaitra avec les f-strings :

[1]:
```
age = 10

# un exemple de f-string
f"Jean a {age} ans"
```

[1]:
```
'Jean a 10 ans'
```

Cette construction - que nous utilisons très fréquemment - n'a été introduite qu'en Python-3.6, aussi si vous utilisez Python-3.5 vous verrez ceci :

```
>>> age = 10
>>> f"Jean a {age} ans"
  File "<stdin>", line 1
    f"Jean a {age} ans"
                       ^
SyntaxError: invalid syntax
```

Dans ce cas vous devrez remplacer ce code avec la méthode `format` - que nous verrons en Semaine 2 avec les chaines de caractères - et dans le cas présent il faudrait remplacer par ceci :

[2]:
```
age = 10

"Jean a {} ans".format(age)
```

[2]:
```
'Jean a 10 ans'
```

Comme ces f-strings sont très présents dans le cours, il est recommandé d'utiliser au moins python-3.6.

Python-3.4

La remarque vaut donc a fortiori pour python-3.4 qui, en outre, ne vous permettra pas de suivre la semaine 8 sur la programmation asynchrone, car les mots-clés `async` et `await` ont été introduits seulement dans Python-3.5.

Version utilisée dans les notebooks / versions plus récentes

Tout le cours doit pouvoir s'exécuter tel quel avec une version plus récente de Python.

Cela dit, certains compléments illustrent des nouveautés apparues après la 3.6, comme les dataclasses qui sont apparues avec python-3.7, et que nous verrons en semaine 6.

Dans tous les cas, nous signalons systématiquement les notebooks qui nécessitent une version plus récente que 3.6.

Voici enfin, à toutes fins utiles, un premier fragment de code Python qui affiche la version de Python utilisée dans tous les notebooks de ce cours.

Nous reviendrons en détail sur l'utilisation des notebooks dans une prochaine séquence, dans l'immédiat pour exécuter ce code vous pouvez :

— désigner avec la souris la cellule de code ; vous verrez alors apparaître une petite flèche à côté du mot `In`, en cliquant cette flèche vous exécutez le code ;
— une autre méthode consiste à sélectionner la cellule de code avec la souris ; une fois que c'est fait vous pouvez cliquer sur le bouton `>| Run` dans la barre de menu (bleue claire) du notebook.

[3]:
```
# ce premier fragment de code affiche des détails sur la
# version de python qui exécute tous les notebooks du cours
import sys
print(sys.version_info)
```

```
sys.version_info(major=3, minor=10, micro=8, releaselevel='final', serial=0
  )
```

Pas de panique si vous n'y arrivez pas, nous consacrerons très bientôt une séquence entière à l'utilisation des notebooks :)

1.2 w1-s2-c1-installer-python

Installer la distribution standard Python

1.2.1 Complément - niveau basique

Ce complément a pour but de vous donner quelques guides pour l'installation de la distribution standard Python 3.

Notez bien qu'il ne s'agit ici que d'indications, il existe de nombreuses façons de procéder.

En cas de souci, commencez par chercher par vous-même, sur Google ou autre, une solution à votre problème ; pensez également à utiliser le forum du cours.

Le point important est de bien vérifier le numéro de version de votre installation qui doit être au moins 3.6

1.2.2 Sachez à qui vous parlez

Mais avant qu'on n'avance sur l'installation proprement dite, il nous faut insister sur un point qui déroute parfois les débutants. On a parfois besoin de recourir à l'emploi d'un terminal, surtout justement pendant la phase d'installation.

Lorsque c'est le cas, il est important de bien distinguer :

— les cas où on s'adresse au terminal (en jargon, on dit le shell),
— et les cas où on s'adresse à l'interpréteur Python.

C'est très important car ces deux programmes ne parlent pas du tout le même langage ! Il peut arriver au début qu'on écrive une commande juste, mais au mauvais interlocuteur, et cela peut être source de frustration. Essayons de bien comprendre ce point.

Le terminal

Je peux dire que je parle à mon terminal quand l'invite de commande (en jargon on dit le prompt) se termine par un dollar `$` - ou un simple chevron `>` sur Windows

Par exemple sur un mac :

```
~/git/flotpython/w1 $
```

Ou sur Windows :

```
C:\Users>
```

L'interprète Python

À partir du terminal, je peux lancer un interpréteur Python, qui se reconnaît car son prompt est fait de 3 chevrons `>>>`

```
~/git/flotpython/w1 $ python3
Python 3.7.0 (default, Jun 29 2018, 20:14:27)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Pour sortir de l'interpréteur Python, et retourner au terminal, j'utilise la fonction Python `exit()` :

```
~/git/flotpython/w1 $ python3
>>> 20 * 60
1200
>>> exit()
~/git/flotpython/w1 $ python3
```

Les erreurs typiques

Gardez bien cette distinction présente à l'esprit, lorsque vous lisez la suite. Voici quelques symptômes habituels de ce qu'on obtient si on se trompe.

Par exemple, la commande `python3 -V` est une commande qui s'adresse au terminal ; c'est pourquoi nous la faisons précéder d'un dollar `$`.

Si vous essayez de la taper alors que vous êtes déjà dans un interpréteur python - ou sous IDLE d'ailleurs -, vous obtenez un message d'erreur de ce genre :

```
>>> python3 -V
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python3' is not defined
```

Réciproquement, si vous essayez de taper du Python directement dans un terminal, ça se passe mal aussi, forcément. Par exemple sur Mac, avec des fragments Python tout simples :

```
~/git/flotpython/w1 $ import math
-bash: import: command not found
~/git/flotpython/w1 $ 30 * 60
-bash: 30: command not found
~/git/flotpython/w1 $ foo = 30 * 60
-bash: foo: command not found
```

1.2.3 Digression - coexistence de Python2 et Python3

Avant l'arrivée de la version 3 de Python, les choses étaient simples, on exécutait un programme Python avec une seule commande `python`. Depuis 2014-2015, les deux versions de Python ont coexisté, il est nécessaire d'adopter une convention qui permette d'installer les deux langages sous des noms qui sont non-ambigus

C'est pourquoi actuellement, on trouve le plus souvent la convention suivante sous Linux et macOS :

- `python3` est pour exécuter les programmes en Python-3 ; du coup on trouve alors également les commandes comme `idle3` pour lancer IDLE, et par exemple `pip3` pour le gestionnaire de paquets (voir ci-dessous) ;
- `python2` est pour exécuter les programmes en Python-2 (il faut savoir par ailleurs que MacOS arrive avec un python2 préinstallé, car certaines parties de l'OS en ont besoin !; du coup sur cet OS on ne peut pas se débarrasser complètement de Python2)
- enfin selon les systèmes, la commande `python` tout court est un alias pour `python2` ou `python3`. De plus en plus souvent bien entendu, `python` désigne `python3`.

à titre d'illustration, voici ce que j'obtiens sur mon mac dans mon environnement de tous les jours (Déc 2022) :

```
$ python3 -V
Python 3.11.0
$ python2 -V
Python 2.7.16
$ python -V
Python 3.11.0
```

Sous Windows, vous avez un lanceur qui s'appelle `py`. Par défaut, il lance la version de Python la plus récente installée, mais vous pouvez spécifier une version spécifique de la manière suivante :

```
C:\> py -2.7
```

pour lancer, par exemple, Python en version 2.7. Vous trouverez toute la documentation nécessaire pour Windows sur cette page (en anglais)

Pour éviter d'éventuelles confusions, nous précisons toujours `python3` dans le cours.

1.2.4 Installation de base

Vous utilisez Windows

La méthode recommandée sur Windows est de partir de la page https://www.python.org/download où vous trouverez un programme d'installation qui contient tout ce dont vous aurez besoin pour suivre le cours.

Pour vérifier que vous êtes prêt, il vous faut lancer IDLE (quelque part dans le menu Démarrer) et vérifier le numéro de version.

Vous utilisez macOS

Ici encore, la méthode recommandée est de partir de la page https://www.python.org/download et d'utiliser le programme d'installation.

Sachez aussi, si vous utilisez déjà MacPorts https://www.macports.org, que vous pouvez également utiliser cet outil pour installer, par exemple Python 3.6, avec la commande

```
$ sudo port install python36
```

Vous utilisez Linux

Dans ce cas il est très probable que Python-3.x soit déjà disponible sur votre machine. Pour vous en assurer, essayez de lancer la commande `python3` dans un terminal.

RHEL / Fedora

Voici par exemple ce qu'on obtient depuis un terminal sur une machine installée en Fedora-20 :

```
$ python3
Python 3.6.2 (default, Jul 20 2017, 12:30:02)
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Vérifiez bien le numéro de version qui doit être en 3.x. Si vous obtenez un message du style `python3: command not found` utilisez `dnf` (anciennement connu sous le nom de `yum`) pour installer le rpm `python3` comme ceci :

```
$ sudo dnf install python3
```

S'agissant d'`idle`, l'éditeur que nous utilisons dans le cours (optionnel si vous êtes familier avec un éditeur de texte), vérifiez sa présence comme ceci :

```
$ type idle3
idle is hashed (/usr/bin/idle3)
```

Ici encore, si la commande n'est pas disponible vous pouvez l'installer avec :

```
$ sudo yum install python3-tools
```

Debian / Ubuntu

Ici encore, Python-2.7 est sans doute déjà disponible. Procédez comme ci-dessus, voici un exemple recueilli dans un terminal sur une machine installée en Ubuntu-14.04/trusty :

```
$ python3
Python 3.6.2 (default, Jul 20 2017, 12:30:02)
[GCC 6.3.1 20161221 (Red Hat 6.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Pour installer Python :

```
$ sudo apt-get install python3
```

Pour installer idle :

```
$ sudo apt-get install idle3
```

Installation de bibliothèques complémentaires

Il existe un outil très pratique pour installer des bibliothèques Python, il s'appelle `pip3`, qui est documenté ici

Sachez aussi, si par ailleurs vous utilisez un gestionnaire de paquets comme `rpm` sur RHEL, `apt-get` sur Debian, ou `port` sur macOS, que de nombreux paquets sont également disponibles au travers de ces outils.

Anaconda

Sachez qu'il existe beaucoup de distributions alternatives qui incluent Python ; parmi elles, la plus populaire est sans aucun doute Anaconda, qui contient un grand nombre de bibliothèques de calcul scientifique, et également d'ailleurs Jupyter pour travailler nativement sur des notebooks au format `.ipynb`.

Anaconda vient avec son propre gestionnaire de paquets pour l'installation de bibliothèques supplémentaires qui s'appelle `conda`.

1.3 w1-s2-c2-lecture

Un peu de lecture

1.3.1 Complément - niveau basique

Mise à jour de Juillet 2018

Le 12 Juillet 2018, Guido van Rossum a annoncé qu'il quittait la fonction de BDFL qu'il occupait depuis près de trois décennies. Il n'est pas tout à fait clair à ce stade comment va évoluer la gouvernance de Python.

Le Zen de Python

Vous pouvez lire le "Zen de Python", qui résume la philosophie du langage, en important le module `this` avec ce code : (pour exécuter ce code, cliquez dans la cellule de code, et faites au clavier "Majuscule/Entrée" ou "Shift/Enter")

[1]:
```
# le Zen de Python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Documentation

— On peut commencer par citer l'article de Wikipédia sur Python en français.

— La page sur le langage en français.

— La documentation originale de Python 3 - donc, en anglais - est un très bon point d'entrée lorsqu'on cherche un sujet particulier, mais (beaucoup) trop abondante pour être lue d'un seul trait. Pour chercher de la documentation sur un module particulier, le plus simple est encore d'utiliser Google - ou votre moteur de recherche favori - qui vous redirigera, dans la grande majorité des cas, vers la page qui va bien dans, précisément, la documentation de Python.

  À titre d'exercice, cherchez la documentation du module `pathlib` en cherchant sur Google les mots-clé `"python module pathlib"`.

— J'aimerais vous signaler également une initiative pour traduire la documentation officielle en français.

Historique et survol

— La FAQ officielle de Python (en anglais) sur les choix de conception et l'historique du langage.
— L'article de Wikipédia (en anglais) sur l'historique du langage.
— Sur Wikipédia, un article (en anglais) sur la syntaxe et la sémantique de Python.

Un peu de folklore

— Le discours de Guido van Rossum à PyCon 2016.
— Sur YouTube, le sketch des Monty Python, (malheureusement plus disponible sur YouTube) d'où proviennent les termes `spam`, `eggs` et autres `beans` que l'on utilise traditionnellement dans les exemples en Python plutôt que `foo` et `bar`.
— L'article Wikipédia correspondant, qui cite le langage Python.

1.3.2 Complément - niveau intermédiaire

Licence

— La licence d'utilisation est disponible ici.
— La page de la Python Software Foundation, qui est une entité légale similaire à nos associations de 1901, à but non lucratif ; elle possède les droits sur le langage.

Le processus de développement

— Comment les choix d'évolution sont proposés et discutés, au travers des PEP (Python Enhancement Proposals) - sur wikipedia
— Le premier PEP : PEP-001 donc décrit en détail le cycle de vie des PEPs
— Le PEP 008, qui préconise un style de présentation (style guide)
— L'index de tous les PEPs

1.4 w1-s4-c1-utiliser-les-notebooks

"Notebooks" Jupyter comme support de cours

Pour illustrer les vidéos du MOOC, nous avons choisi d'utiliser Jupyter pour vous rédiger les documents "mixtes" contenant du texte et du code Python, qu'on appelle des "notebooks", et dont le présent document est un exemple.

Nous allons, dans la suite, utiliser du code Python, pourtant nous n'avons pas encore abordé le langage. Pas d'inquiétude, ce code est uniquement destiné à valider le fonctionnement des notebooks, et nous n'utilisons que des choses très simples.

Avantages des notebooks

Comme vous le voyez, ce support permet un format plus lisible que des commentaires dans un fichier de code.

Nous attirons votre attention sur le fait que les fragments de code peuvent être évalués et modifiés. Ainsi vous pouvez facilement essayer des variantes autour du notebook original.

Notez bien également que le code Python est interprété sur une machine distante, ce qui vous permet de faire vos premiers pas avant même d'avoir procédé à l'installation de Python sur votre propre ordinateur.

Comment utiliser les notebooks

En haut du notebook, vous avez une barre de menu (sur fond bleu clair), contenant :

— un titre pour le notebook, avec un numéro de version ;
— une barre de menus avec les entrées `File`, `Insert`, `Cell`, `Kernel` ;
— et une barre de boutons qui sont des raccourcis vers certains menus fréquemment utilisés. Si vous laissez votre souris au dessus d'un bouton, un petit texte apparaît, indiquant à quelle fonction correspond ce bouton.

Nous avons vu dans la vidéo qu'un notebook est constitué d'une suite de cellules, soit textuelles, soit contenant du code. Les cellules de code sont facilement reconnaissables, elles sont précédées de `In [ ]:`. La cellule qui suit celle que vous êtes en train de lire est une cellule de code.

Pour commencer, sélectionnez cette cellule de code avec votre souris, et appuyez dans la barre de menu - en haut du notebook, donc - sur celui en forme de flèche triangulaire vers la droite (Play) :

```
[1]: 20 * 30
```

```
[1]: 600
```

Comme vous le voyez, la cellule est "exécutée" (on dira plus volontiers évaluée), et on passe à la cellule suivante.

Alternativement, vous pouvez simplement taper au clavier Shift+Enter, ou selon les claviers Maj-Entrée, pour obtenir le même effet. D'une manière générale, il est important d'apprendre et d'utiliser les raccourcis clavier, cela vous fera gagner beaucoup de temps par la suite.

La façon habituelle d'exécuter l'ensemble du notebook consiste :

— à sélectionner la première cellule,
— et à taper Shift+Enter jusqu'à atteindre la fin du notebook.

Lorsqu'une cellule de code a été évaluée, Jupyter ajoute sous la cellule `In` une cellule `Out` qui donne le résultat du fragment Python, soit ci-dessus 600.

Jupyter ajoute également un nombre entre les crochets pour afficher, par exemple ci-dessus, `In [1]:`. Ce nombre vous permet de retrouver l'ordre dans lequel les cellules ont été évaluées.

Vous pouvez naturellement modifier ces cellules de code pour faire des essais ; ainsi vous pouvez vous servir du modèle ci-dessous pour calculer la racine carrée de 3, ou essayer la fonction sur un nombre négatif et voir comment est signalée l'erreur.

[2]:
```
# math.sqrt (pour square root) calcule la racine carrée
import math
math.sqrt(2)
```

[2]:
```
1.4142135623730951
```

On peut également évaluer tout le notebook en une seule fois en utilisant le menu Cell -> Run All.

Attention à bien évaluer les cellules dans l'ordre

Il est important que les cellules de code soient évaluées dans le bon ordre. Si vous ne respectez pas l'ordre dans lequel les cellules de code sont présentées, le résultat peut être inattendu.

En fait, évaluer un programme sous forme de notebook revient à le découper en petits fragments, et si on exécute ces fragments dans le désordre, on obtient naturellement un programme différent.

On le voit sur cet exemple :

[3]:
```
message = "Faites attention à l'ordre dans lequel vous évaluez les notebooks"
```

[4]:
```
print(message)
```

```
Faites attention à l'ordre dans lequel vous évaluez les notebooks
```

Si un peu plus loin dans le notebook on fait par exemple :

[5]:
```
# ceci a pour effet d'effacer la variable 'message'
del message
```

qui rend le symbole `message` indéfini, alors bien sûr on ne peut plus évaluer la cellule qui fait `print` puisque la variable `message` n'est plus connue de l'interpréteur.

Réinitialiser l'interpréteur

Si vous faites trop de modifications, ou perdez le fil de ce que vous avez évalué, il peut être utile de redémarrer votre interpréteur. Le menu Kernel → Restart vous permet de faire cela, un peu à la manière de IDLE qui repart d'un interpréteur vierge lorsque vous utilisez la fonction F5.

Le menu Kernel → Interrupt peut être quant à lui utilisé si votre fragment prend trop longtemps à s'exécuter (par exemple vous avez écrit une boucle dont la logique est cassée et qui ne termine pas).

Vous travaillez sur une copie

Un des avantages principaux des notebooks est de vous permettre de modifier le code que nous avons écrit, et de voir par vous-même comment se comporte le code modifié.

Pour cette raison, chaque élève dispose de sa propre copie de chaque notebook, vous pouvez bien sûr apporter toutes les modifications que vous souhaitez à vos notebooks sans affecter les autres étudiants.

Revenir à la version du cours

Vous pouvez toujours revenir à la version "du cours" grâce au menu File → Reset to original.

Attention, avec cette fonction vous restaurez tout le notebook et donc vous perdez vos modifications sur ce notebook.

Télécharger au format Python

Vous pouvez télécharger un notebook au format Python sur votre ordinateur grâce au menu File → Download as → Python

Les cellules de texte sont préservées dans le résultat sous forme de commentaires Python.

Partager un notebook en lecture seule

Enfin, avec le menu File → Share static version, vous pouvez publier une version en lecture seule de votre notebook ; vous obtenez une URL que vous pouvez publier, par exemple pour demander de l'aide sur le forum. Ainsi, les autres étudiants peuvent accéder en lecture seule à votre code.

Notez que lorsque vous utilisez cette fonction plusieurs fois, c'est toujours la dernière version publiée que verront vos camarades, l'URL utilisée reste toujours la même pour un étudiant et un notebook donné.

Ajouter des cellules

Vous pouvez ajouter une cellule n'importe où dans le document avec le bouton + de la barre de boutons.

Aussi, lorsque vous arrivez à la fin du document, une nouvelle cellule est créée chaque fois que vous évaluez la dernière cellule ; de cette façon vous disposez d'un brouillon pour vos propres essais.

À vous de jouer.

1.5 w1-s4-c2-interpreteur-et-notebooks

Modes d'exécution

Nous avons donc à notre disposition plusieurs façons d'exécuter un programme Python. Nous allons les étudier plus en détail :

Quoi	Avec quel outil
fichier complet	`python3 <fichier>.py`
ligne à ligne	`python3` en mode interactif
	ou sous `ipython3`
	ou avec IDLE
par fragments	dans un notebook

Pour cela nous allons voir le comportement d'un tout petit programme Python lorsqu'on l'exécute sous ces différents environnements.

On veut surtout expliquer une petite différence quant au niveau de détail de ce qui se trouve imprimé.

Essentiellement, lorsqu'on utilise l'interpréteur en mode interactif - ou sous IDLE - à chaque fois que l'on tape une ligne, le résultat est calculé (on dit aussi évalué) puis imprimé.

Par contre, lorsqu'on écrit tout un programme, on ne peut plus imprimer le résultat de toutes les lignes, cela produirait un flot d'impression beaucoup trop important. Par conséquent, si vous ne déclenchez pas une impression avec, par exemple, la fonction `print`, rien ne s'affichera.

Enfin, en ce qui concerne le notebook, le comportement est un peu hybride entre les deux, en ce sens que seul le dernier résultat de la cellule est imprimé.

L'interpréteur Python interactif

Le programme choisi est très simple, c'est le suivant :

```
10 * 10
20 * 20
30 * 30
```

Voici comment se comporte l'interpréteur interactif quand on lui soumet ces instructions :

```
$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10 * 10
100
>>> 20 * 20
400
>>> 30 * 30
900
>>> exit()
$
```

Notez que pour terminer la session, il nous faut "sortir" de l'interpréteur en tapant `exit()`.

On peut aussi taper `Control-D` sous Linux ou macOS.

Comme on le voit ici, l'interpréteur imprime le résultat de chaque ligne. On voit bien apparaître toutes les valeurs calculées, 100, 400, puis enfin 900.

Sous forme de programme constitué

Voyons à présent ce que donne cette même séquence de calculs dans un programme complet. Pour cela, il nous faut tout d'abord fabriquer un fichier avec un suffixe en `.py`, en utilisant par exemple un éditeur de fichier. Le résultat doit ressembler à ceci :

```
$ cat foo.py
10 * 10
20 * 20
30 * 30
$
```

Exécutons à présent ce programme :

```
$ python3 foo.py
$
```

On constate donc que ce programme ne fait rien ! En tout cas, selon toute apparence.

En réalité, les 3 valeurs 100, 400 et 900 sont bien calculées, mais comme aucune instruction **print** n'est présente, rien n'est imprimé et le programme se termine sans signe apparent d'avoir réellement fonctionné.

Ce comportement peut paraître un peu déroutant au début, mais comme nous l'avons mentionné c'est tout à fait délibéré. Un programme fonctionnel faisant facilement plusieurs milliers de lignes, voire beaucoup plus, il ne serait pas du tout réaliste que chaque ligne produise une impression, comme c'est le cas en mode interactif.

Dans un notebook

Voici à présent le même programme dans un notebook :

[1]:
```
10 * 10
20 * 20
30 * 30
```

[1]:
```
900
```

Lorsqu'on exécute cette cellule (rappel : sélectionner la cellule, et utiliser le bouton en forme de flèche vers la droite, ou entrer "Shift+Enter" au clavier), on obtient une seule valeur dans la rubrique `Out[]`, 900, qui correspond au résultat de la dernière ligne.

Utiliser **print**

Ainsi, pour afficher un résultat intermédiaire, on utilise l'instruction `print`. Nous verrons cette instruction en détail dans les semaines qui viennent, mais en guise d'introduction disons seulement que c'est une fonction comme les autres en Python 3.

[2]:
```
a = 10
b = 20

print(a, b)
```

```
10 20
```

On peut naturellement mélanger des objets de plusieurs types, et donc mélanger des chaînes de caractères et des nombres pour obtenir un résultat un peu plus lisible. En effet, lorsque le programme devient gros, il est important de savoir à quoi correspond une ligne dans le flot de toutes les impressions. Aussi on préfèrera quelque chose comme :

[3]:
```
print("a =", a, "et b =", b)
```

```
a = 10 et b = 20
```

[4]:
```
# ou encore, équivalente mais avec un f-string
print(f"a = {a} et b = {b}")
```

```
a = 10 et b = 20
```

Une pratique courante consiste d'ailleurs à utiliser les commentaires pour laisser dans le code les instructions `print` qui correspondent à du debug (c'est-à-dire qui ont pu être utiles lors de la mise au point et qu'on veut pouvoir réactiver rapidement).

Utiliser **print** pour “sous-titrer” une affectation

Remarquons enfin que l'affectation à une variable ne retourne aucun résultat.

C'est-à-dire, en pratique, que si on écrit :

[5]:
```
a = 100
```

même une fois l'expression évaluée par l'interpréteur, aucune ligne `Out[]` n'est ajoutée.

C'est pourquoi, il nous arrivera parfois d'écrire, notamment lorsque l'expression est complexe et pour rendre explicite la valeur qui vient d'être affectée :

[6]:
```
a = 100; print(a)
```

```
100
```

Notez bien que cette technique est uniquement pédagogique, et n'a absolument aucun autre intérêt dans la pratique ; il n'est pas recommandé de l'utiliser en dehors de ce contexte.

1.6 w1-s4-c3-fibonacci-prompt

La suite de Fibonacci

1.6.1 Complément - niveau basique

Voici un premier exemple de code qui tourne.

Nous allons commencer par le faire tourner dans ce notebook. Nous verrons en fin de séance comment le faire fonctionner localement sur votre ordinateur.

Le but de ce programme est de calculer la suite de Fibonacci, qui est définie comme ceci :

- $u_0 = 0$
- $u_1 = 1$
- $\forall n >= 2, u_n = u_{n-1} + u_{n-2}$

Ce qui donne pour les premières valeurs :

n	fibonacci(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13

On commence par définir la fonction `fibonacci` comme il suit. Naturellement vous n'avez pas encore tout le bagage pour lire ce code, ne vous inquiétez pas, nous allons vous expliquer tout ça dans les prochaines semaines. Le but est uniquement de vous montrer un fonctionnement de l'interpréteur Python et de IDLE.

[1]:
```
def fibonacci(n):
    "retourne le nombre de fibonacci pour l'entier n"
    # pour les deux petites valeurs de n, on peut retourner n
```

```
    if n <= 1:
        return n
    # sinon on initialise f2 pour n-2 et f1 pour n-1
    f2, f1 = 0, 1
    # et on itère n-1 fois pour additionner
    for i in range(2, n + 1):
        f2, f1 = f1, f1 + f2
#         print(i, f2, f1)
    # le résultat est dans f1
    return f1
```

Pour en faire un programme utilisable on va demander à l'utilisateur de rentrer un nombre ; il faut le convertir en entier car `input` renvoie une chaîne de caractères :

[2]:
```
entier = int(input("Entrer un entier "))

# NOTE:
# auto-exec-for-latex has used instead:
##########
entier = 12
##########
```

On imprime le résultat :

[3]:
```
print(f"fibonacci({entier}) = {fibonacci(entier)}")
```

```
fibonacci(12) = 144
```

Exercice

Vous pouvez donc à présent :

- exécuter le code dans ce notebook
- télécharger ce code sur votre disque comme un fichier `fibonacci_prompt.py`
  - utiliser pour cela le menu "File -> Download as -> Python"
  - et renommer le fichier obtenu au besoin
- l'exécuter sous IDLE
- le modifier, par exemple pour afficher les résultats intermédiaires
  - on a laissé exprès une fonction `print` en commentaire que vous pouvez réactiver simplement
- l'exécuter avec l'interpréteur Python comme ceci :

```
$ python3 fibonacci_prompt.py
```

Ce code est volontairement simple et peu robuste pour ne pas l'alourdir. Par exemple, ce programme se comporte mal si vous entrez un entier négatif.

Nous allons voir tout de suite une version légèrement différente qui va vous permettre de donner la valeur d'entrée sur la ligne de commande.

1.7 w1-s4-c4-fibonacci

La suite de Fibonacci (version pour terminal)

1.7.1 Complément - niveau intermédiaire

Nous reprenons le cas de la fonction `fibonacci` que nous avons déjà vue, mais cette fois nous voulons que l'utilisateur puisse indiquer l'entier en entrée de l'algorithme, non plus en répondant à une question, mais sur la ligne de commande, c'est-à-dire en tapant :

```
$ python3 fibonacci.py 12
```

Avertissement :

Attention, cette version-ci ne fonctionne pas dans ce notebook, justement car on n'a pas de moyen dans un notebook d'invoquer un programme en lui passant des arguments de cette façon. Ce notebook est rédigé pour vous permettre de vous entraîner avec la fonction de téléchargement au format Python, qu'on a vue dans la vidéo, et de faire tourner ce programme sur votre propre ordinateur.

Le module **argparse**

Cette fois nous importons le module **argparse**, c'est lui qui va nous permettre d'interpréter les arguments passés sur la ligne de commande.

[1]:
```
from argparse import ArgumentParser
```

Puis nous répétons la fonction `fibonacci` :

[2]:
```
def fibonacci(n):
    "retourne le nombre de fibonacci pour l'entier n"
    # pour les deux premières valeurs de n, on peut renvoyer n
    if n <= 1:
        return n
    # sinon on initialise f2 pour n-2 et f1 pour n-1
    f2, f1 = 0, 1
    # et on itère n-1 fois pour additionner
    for i in range(2, n + 1):
        f2, f1 = f1, f1 + f2
#         print(i, f2, f1)
    # le résultat est dans f1
    return f1
```

Remarque :

Certains d'entre vous auront évidemment remarqué que l'on aurait pu éviter de copier-coller la fonction `fibonacci` comme cela ; c'est à ça que servent les modules, mais nous n'en sommes pas là.

Un objet **parser**

À présent, nous utilisons le module **argparse**, pour lui dire qu'on attend exactement un argument sur la ligne de commande, et qu'il doit être un entier. Ici encore, ne vous inquiétez pas si vous ne comprenez pas tout le code. L'objectif est de vous donner un morceau de code utilisable tout de suite, pour jouer avec votre interpréteur Python.

[3]:
```
# à nouveau : ceci n'est pas conçu pour être exécuté dans le notebook !
parser = ArgumentParser()
parser.add_argument(dest="entier", type=int,
```

```
                    help="entier d'entrée")
input_args = parser.parse_args()
entier = input_args.entier

# NOTE:
# auto-exec-for-latex has used instead:
##########
entier = 8
##########
```

Nous pouvons à présent afficher le résultat :

[4]:
```
print(f"fibonacci({entier}) = {fibonacci(entier)}")
```

```
fibonacci(8) = 21
```

Vous pouvez donc à présent :

— télécharger ce code sur votre disque comme un fichier `fibonacci.py` en utilisant le menu "File -> Download as -> Python"
— l'exécuter avec simplement l'interpréteur Python comme ceci :

```
$ python3 fibonacci.py 56
```

(sans taper le signe `$` qui indique simplement le prompt du terminal.)

1.8 w1-s4-c5-shebang

La ligne shebang

```
#!/usr/bin/env python3
```

1.8.1 Complément - niveau avancé

Ce complément est uniquement valable pour macOS et Linux.

Le besoin

Nous avons vu dans la vidéo que, pour lancer un programme Python, on fait depuis le terminal :

```
$ python3 mon_module.py
```

Lorsqu'il s'agit d'un programme que l'on utilise fréquemment, on n'est pas forcément dans le répertoire où se trouve le programme Python. Aussi, dans ce cas, on peut utiliser un chemin "absolu", c'est-à-dire à partir de la racine des noms de fichiers, comme par exemple :

```
$ python3 /le/chemin/jusqu/a/mon_module.py
```

Sauf que c'est assez malcommode, et cela devient vite pénible à la longue.

La solution

Sur Linux et macOS, il existe une astuce utile pour simplifier cela. Voyons comment s'y prendre, avec par exemple le programme `fibonacci.py` que vous pouvez télécharger ici (nous avons vu ce code en détail dans les deux compléments précédents). Commencez par sauver ce code sur votre ordinateur dans un fichier qui s'appelle, bien entendu, `fibonacci.py`.

On commence par éditer le tout début du fichier pour lui ajouter une première ligne :

```
#!/usr/bin/env python3

## La suite de Fibonacci (Suite)
...etc...
```

Cette première ligne s'appelle un Shebang dans le jargon Unix. Unix stipule que le Shebang doit être en première position dans le fichier.

Ensuite on rajoute au fichier, depuis le terminal, le caractère exécutable comme ceci :

```
$ pwd
/le/chemin/jusqu/a/
```

```
$ chmod +x fibonacci.py
```

À partir de là, vous pouvez utiliser le fichier `fibonacci.py` comme une commande, sans avoir à mentionner `python3`, qui sera invoqué au travers du shebang :

```
$ /le/chemin/jusqu/a/fibonacci.py 20
fibonacci(20) = 6765
```

Et donc vous pouvez aussi le déplacer dans un répertoire qui est dans votre variable `PATH` ; de cette façon vous les rendez ainsi accessible à partir de n'importe quel répertoire en faisant simplement :

```
$ export PATH=/le/chemin/jusqu/a:$PATH
```

```
$ cd /tmp
$ fibonacci.py 20
fibonacci(20) = 6765
```

Remarque : tout ceci fonctionne très bien tant que votre point d'entrée - ici `fibonacci.py` - n'utilise que des modules standards. Dans le cas où le point d'entrée vient avec au moins un module, il est également nécessaire d'installer ces modules quelque part, et d'indiquer au point d'entrée comment les trouver, nous y reviendrons dans la semaine où nous parlerons des modules.

1.9 w1-s4-x1-turtle

Dessiner un carré

1.9.1 Exercice - niveau intermédiaire

Voici un tout petit programme qui dessine un carré.

Il utilise le module `turtle`, conçu précisément à des fins pédagogiques. Pour des raisons techniques, le module `turtle` n'est pas disponible au travers de la plateforme FUN.

Il est donc inutile d'essayer d'exécuter ce programme depuis le notebook. L'objectif de cet exercice est plutôt de vous entraîner à télécharger ce programme en utilisant le menu "File -> Download as -> Python", puis à le charger dans votre IDLE pour l'exécuter sur votre machine.

Attention également à sauver le programme téléchargé sous un autre nom que `turtle.py`, car sinon vous allez empêcher Python de trouver le module standard `turtle` ; appelez-le par exemple `turtle_basic.py`.

[1]:
```
# on a besoin du module turtle
import turtle
```

On commence par définir une fonction qui dessine un carré de côté `length` :

[2]:
```
def square(length):
    "have the turtle draw a square of side <length>"
    for side in range(4):
        turtle.forward(length)
        turtle.left(90)
```

Maintenant on commence par initialiser la tortue :

[3]:
```
turtle.reset()
```

On peut alors dessiner notre carré :

[ ]:
```
square(200)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Et pour finir on attend que l'utilisateur clique dans la fenêtre de la tortue, et alors on termine :

[ ]:
```
turtle.exitonclick()

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

1.9.2 Exercice - niveau avancé

Naturellement vous pouvez vous amuser à modifier ce code pour dessiner des choses un peu plus amusantes.

Dans ce cas, commencez par chercher "module python turtle" dans votre moteur de recherche favori, pour localiser la documentation du module `turtle`.

Vous trouverez quelques exemples pour commencer ici :

— turtle_multi_squares.py pour dessiner des carrés à l'emplacement de la souris en utilisant plusieurs tortues ;
— turtle_fractal.py pour dessiner une fractale simple ;
— turtle_fractal_reglable.py une variation sur la fractale, plus paramétrable.

1.10 w1-s5-c1-noms-de-variables

Noms de variables

1.10.1 Complément - niveau basique

Revenons sur les noms de variables autorisés ou non.

Les noms les plus simples sont constitués de lettres. Par exemple :

[1]:
```
factoriel = 1
```

On peut utiliser aussi les majuscules, mais attention cela définit une variable différente. Ainsi :

[2]:
```
Factoriel = 100
factoriel == Factoriel
```

[2]:
```
False
```

Le signe `==` permet de tester si deux variables ont la même valeur. Si les variables ont la même valeur, le test retournera `True`, et `False` sinon. On y reviendra bien entendu.

Conventions habituelles

En règle générale, on utilise uniquement des minuscules pour désigner les variables simples (ainsi d'ailleurs que pour les noms de fonctions), les majuscules sont réservées en principe pour d'autres sortes de variables, comme les noms de classe, que nous verrons ultérieurement.

Notons qu'il s'agit uniquement d'une convention, ceci n'est pas imposé par le langage lui-même.

Pour des raisons de lisibilité, il est également possible d'utiliser le tiret bas `_` dans les noms de variables. On préfèrera ainsi :

[3]:
```
age_moyen = 75 # oui
```

plutôt que ceci (bien qu'autorisé par le langage) :

[4]:
```
AgeMoyen = 75 # autorisé, mais non
```

On peut également utiliser des chiffres dans les noms de variables comme par exemple :

[5]:
```
age_moyen_dept75 = 80
```

avec la restriction toutefois que le premier caractère ne peut pas être un chiffre, cette affectation est donc refusée :

[ ]:
```
75_age_moyen = 80 # erreur de syntaxe

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Le tiret bas comme premier caractère

Il est par contre, possible de faire commencer un nom de variable par un tiret bas comme premier caractère ; toutefois, à ce stade, nous vous déconseillons d'utiliser cette pratique qui est réservée à des conventions de nommage bien spécifiques.

[6]:
```
_autorise_mais_deconseille = 'Voir le PEP 008'
```

Et en tout cas, il est fortement déconseillé d'utiliser des noms de la forme `__variable__` qui sont réservés au langage. Nous reviendrons sur ce point dans le futur, mais regardez par exemple cette variable que nous n'avons définie nulle part mais qui pourtant existe bel et bien :

[7]:
```
__name__  # ne définissez pas vous-même de variables de ce genre
```

[7]:
```
'__main__'
```

Ponctuation

Dans la plage des caractères ASCII, il n'est pas possible d'utiliser d'autres caractères que les caractères alphanumériques et le tiret bas. Notamment le tiret haut `-` est interprété comme l'opération de soustraction. Attention donc à cette erreur fréquente :

[ ]:
```
age-moyen = 75  # erreur : en fait python l'interprète comme 'age - moyen = 75'

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Caractères exotiques

En Python 3, il est maintenant aussi possible d'utiliser des caractères Unicode dans les identificateurs :

[8]:
```
# les caractères accentués sont permis
nom_élève = "Jules Maigret"
```

[9]:
```
# ainsi que l'alphabet grec
from math import cos, pi as
θ =  / 4
cos(θ)
```

[9]:
```
0.7071067811865476
```

Tous les caractères Unicode ne sont pas permis - heureusement car cela serait source de confusion. Nous citons dans les références les documents qui précisent quels sont exactement les caractères autorisés.

Conseil Il est très vivement recommandé :

- tout d'abord de coder en anglais ;
- ensuite de ne pas définir des identificateurs avec des caractères non ASCII, dans toute la mesure du possible , voyez par exemple la confusion que peut créer le fait de nommer un identificateur ou II ou ;
- enfin si vous utilisez un encodage autre que UTF-8, vous devez bien spécifier l'encodage utilisé dans votre fichier source ; nous y reviendrons en deuxième semaine.

Pour en savoir plus

Pour les esprits curieux, Guido van Rossum, le fondateur de Python, est le co-auteur d'un document qui décrit les conventions de codage à utiliser dans la bibliothèque standard Python. Ces règles sont plus restrictives que ce que le langage permet de faire, mais constituent une lecture intéressante si vous projetez d'écrire beaucoup de Python.

Voir dans le PEP 008 la section consacrée aux règles de nommage - (en anglais)

Voir enfin, au sujet des caractères exotiques dans les identificateurs :

— le PEP 3131 qui définit les caractères exotiques autorisés, et qui repose à son tour sur
— http://www.unicode.org/reports/tr31/ (très technique !)

1.11 w1-s5-c2-mots-cles

Les mots-clés de Python

Mots réservés

Il existe en Python certains mots spéciaux, qu'on appelle des mots-clés, ou keywords en anglais, qui sont réservés et ne peuvent pas être utilisés comme identifiants, c'est-à-dire comme un nom de variable.

C'est le cas par exemple pour l'instruction `if`, que nous verrons prochainement, qui permet bien entendu d'exécuter tel ou tel code selon le résultat d'un test.

[1]:
```
variable = 15
if variable <= 10:
    print("en dessous de la moyenne")
else:
    print("au dessus")
```

```
au dessus
```

À cause de la présence de cette instruction dans le langage, il n'est pas autorisé d'appeler une variable `if`.

[ ]:
```
# interdit, if est un mot-clé
if = 1

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Liste complète

Voici la liste complète des mots-clés :

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

Nous avons indiqué en gras les nouveautés par rapport à Python 2 (sachant que réciproquement `exec` et `print` ont perdu leur statut de mot-clé depuis Python 2, ce sont maintenant des fonctions).

Il vous faudra donc y prêter attention, surtout au début, mais avec un tout petit peu d'habitude vous saurez rapidement les éviter.

Vous remarquerez aussi que tous les bons éditeurs de texte supportant du code Python vont colorer les mots-clés différemment des variables. Par exemple, IDLE colorie les mots-clés en orange, vous pouvez donc très facilement vous rendre compte que vous allez, par erreur, en utiliser un comme nom de variable.

Cette fonctionnalité, dite de coloration syntaxique, permet d'identifier d'un coup d'œil, grâce à un code de couleur, le rôle des différents éléments de votre code : variables, mots-clés, etc. D'une manière générale, nous vous déconseillons fortement d'utiliser un éditeur de texte qui n'offre pas cette fonctionnalité de coloration syntaxique.

Pour en savoir plus

On peut se reporter à cette page :

https://docs.python.org/3/reference/lexical_analysis.html#keywords

1.12 w1-s5-c3-introduction-types

Un peu de calcul sur les types

1.12.1 Complément - niveau basique

La fonction **type**

Nous avons vu dans la vidéo que chaque objet possède un type. On peut très simplement accéder au type d'un objet en appelant une fonction built-in, c'est-à-dire prédéfinie dans Python, qui s'appelle, eh bien oui, **type**.

On l'utilise tout simplement comme ceci :

[1]:
```
type(1)
```

[1]:
```
int
```

[2]:
```
type('spam')
```

[2]:
```
str
```

Cette fonction est assez peu utilisée par les programmeurs expérimentés, mais va nous être utile à bien comprendre le langage, notamment pour manipuler les valeurs numériques.

Types, variables et objets

On a vu également que le type est attaché à l'objet et non à la variable.

[3]:
```
x = 1
type(x)
```

[3]:
```
int
```

[4]:
```
# la variable x peut référencer un objet de n'importe quel type

x = [1, 2, 3]
type(x)
```

[4]:
```
list
```

1.12.2 Complément - niveau avancé

La fonction `isinstance`

Une autre fonction prédéfinie, voisine de `type` mais plus utile dans la pratique, est la fonction `isinstance` qui permet de savoir si un objet est d'un type donné. Par exemple :

[5]:
```
isinstance(23, int)
```

[5]:
```
True
```

À la vue de ce seul exemple, on pourrait penser que `isinstance` est presque identique à `type` ; en réalité elle est un peu plus élaborée, notamment pour la programmation objet et l'héritage, nous aurons l'occasion d'y revenir.

On remarque ici en passant que la variable `int` est connue de Python alors que nous ne l'avons pas définie. Il s'agit d'une variable prédéfinie, qui désigne le type des entiers, que nous étudierons très bientôt.

Pour conclure sur `isinstance`, cette fonction est utile en pratique précisément parce que Python est à typage dynamique. Aussi il est souvent utile de s'assurer qu'une variable passée à une fonction est du (ou des) type(s) attendu(s), puisque contrairement à un langage typé statiquement comme C++, on n'a aucune garantie de ce genre à l'exécution. À nouveau, nous aurons l'occasion de revenir sur ce point.

1.13 w1-s5-c4-garbage-collector

Gestion de la mémoire

1.13.1 Complément - niveau basique

L'objet de ce complément est de vous montrer qu'avec Python vous n'avez pas à vous préoccuper de la mémoire. Pour expliquer la notion de gestion de la mémoire, il nous faut donner un certain nombre de détails sur d'autres langages comme C et C++. Si vous souhaitez suivre ce cours à un niveau basique vous pouvez ignorer ce complément et seulement retenir que Python se charge de tout pour vous :)

1.13.2 Complément - niveau intermédiaire

Langages de bas niveau

Dans un langage traditionnel de bas niveau comme C ou C++, le programmeur est en charge de l'allocation - et donc de la libération - de la mémoire.

Ce qui signifie que, sauf pour les valeurs stockées dans la pile, le programmeur est amené :

— à réclamer de la mémoire au système d'exploitation en appelant explicitement `malloc` (C) ou `new` (C++) ;
— et réciproquement à rendre cette mémoire au système d'exploitation lorsqu'elle n'est plus utilisée, en appelant `free` (C) ou `delete` (C++).

Avec ce genre de langage, la gestion de la mémoire est un aspect important de la programmation. Ce modèle offre une grande flexibilité, mais au prix d'un coût élevé en matière de vitesse de développement.

En effet, il est assez facile d'oublier de libérer la mémoire après usage, ce qui peut conduire à épuiser les ressources disponibles. À l'inverse, utiliser une zone mémoire non allouée peut conduire à des bugs très difficiles à localiser et à des problèmes de sécurité majeurs. Notons qu'une grande partie des attaques en informatique reposent sur l'exploitation d'erreurs de gestion de la mémoire.

Langages de haut niveau

Pour toutes ces raisons, avec un langage de plus haut niveau comme Python, le programmeur est libéré de cet aspect de la programmation.

Pour anticiper un peu sur le cours des semaines suivantes, voici ce que vous pouvez garder en tête s'agissant de la gestion mémoire en Python :

- vous créez vos objets au fur et à mesure de vos besoins ;
- vous n'avez pas besoin de les libérer explicitement, le "Garbage Collector" de Python va s'en charger pour recycler la mémoire lorsque c'est possible ;
- Python a tendance à être assez gourmand en mémoire, comparé à un langage de bas niveau, car tout est objet et chaque objet est assorti de méta-informations qui occupent une place non négligeable. Par exemple, chaque objet possède au minimum :
  - une référence vers son type - c'est le prix du typage dynamique ;
  - un compteur de références - le nombre d'autres valeurs (variables ou objets) qui pointent vers l'objet, cette information est notamment utilisée, précisément, par le Garbage Collector pour déterminer si la mémoire utilisée par un objet peut être libérée ou non.
- un certain nombre de types prédéfinis et non mutables sont implémentés en Python comme des singletons, c'est-à-dire qu'un seul objet est créé et partagé, c'est le cas par exemple pour les petits entiers et les chaînes de caractères, on en reparlera ;
- lorsqu'on implémente une classe, il est possible de lui conférer cette caractéristique de singleton, de manière à optimiser la mémoire nécessaire pour exécuter un programme.

[1]:
```
# quand vous faites par exemple ceci

print("hello world")

# Python a alloué pour vous la mémoire nécessaire
# pour ranger l'objet chaine "Hello world"
# et vous n'avez pas besoin de libérer cette mémoire vous-même
# le garbage collector s'en occupera en temps utile
```

```
hello world
```

1.14 w1-s5-c5-type-checking

Typages statique et dynamique

1.14.1 Complément - niveau intermédiaire

Parmi les langages typés, on distingue les langages à typage statique et ceux à typage dynamique. Ce notebook tente d'éclaircir ces notions pour ceux qui n'y sont pas familiers.

Typage statique

À une extrémité du spectre, on trouve les langages compilés, dits à typage statique, comme par exemple C ou C++.

En C on écrira, par exemple, une version simpliste de la fonction factoriel comme ceci :

```
int factoriel(int n) {
    int result = 1;
    for (int loop = 1; loop <= n; loop++)
        result *= loop;
    return result;
}
```

Comme vous pouvez le voir - ou le deviner - toutes les variables utilisées ici (comme par exemple `n`, `result` et `loop`) sont typées :

— on doit appeler `factoriel` avec un argument `n` qui doit être un entier (`int` est le nom du type entier) ;
— les variables internes `result` et `loop` sont de type entier ;
— `factoriel` retourne une valeur de type entier.

Ces informations de type ont essentiellement trois fonctions :

— en premier lieu, elles sont nécessaires au compilateur. En C si le programmeur ne précisait pas que `result` est de type entier, le compilateur n'aurait pas suffisamment d'éléments pour générer le code assembleur correspondant ;
— en contrepartie, le programmeur a un contrôle très fin de l'usage qu'il fait de la mémoire et du matériel. Il peut choisir d'utiliser un entier sur 32 ou 64 bits, signé ou pas, ou construire avec `struct` et `union` un arrangement de ses données ;
— enfin, et surtout, ces informations de type permettent de faire un contrôle a priori de la validité du programme, par exemple, si à un autre endroit dans le code on trouve :

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    /* le premier argument de la ligne de commande est argv[1] */
    char *input = argv[1];
    /* calculer son factoriel et afficher le résultat */
    printf("Factoriel (%s) = %d\n", input, factoriel(input));
    /*                                          ^^^^^

     * ici on appelle factoriel avec une entrée de type 'chaîne de caractères' */
}
```

alors le compilateur va remarquer qu'on essaie d'appeler `factoriel` avec comme argument `input` qui, pour faire simple, est une chaîne de caractères et comme `factoriel` s'attend à recevoir un entier, ce programme n'a aucune chance de compiler.

On parle alors de typage statique, en ce sens que chaque variable a exactement un type qui est défini par le programmeur une bonne fois pour toutes.

C'est ce qu'on appelle le contrôle de type, ou type-checking en anglais. Si on ignore le point sur le contrôle fin de la mémoire, qui n'est pas crucial à notre sujet, ce modèle de contrôle de type présente :

— l'inconvénient de demander davantage au programmeur (je fais abstraction, à ce stade et pour simplifier, de langages à inférence de types comme ML et Haskell) ;
— et l'avantage de permettre un contrôle étendu, et surtout précoce (avant même de l'exécuter), de la bonne correction du programme.

Cela étant dit, le typage statique en C n'empêche pas le programmeur débutant d'essayer d'écrire dans la mémoire à partir d'un pointeur `NULL` - et le programme de s'interrompre brutalement. Il faut être conscient des limites du typage statique.

Typage dynamique

À l'autre bout du spectre, on trouve des langages comme, eh bien, Python.

Pour comprendre cette notion de typage dynamique, regardons la fonction suivante `somme`.

[1]:
```python
def somme(*largs):
    "retourne la somme de tous ses arguments"
    if not largs:
        return 0
```

```
    result = largs[0]
    for i in range(1, len(largs)):
        result += largs[i]
    return result
```

Naturellement, vous n'êtes pas à ce stade en mesure de comprendre le fonctionnement intime de la fonction. Mais vous pouvez tout de même l'utiliser :

[2]:
```
somme(12, 14, 300)
```

[2]:
```
326
```

[3]:
```
liste1 = ['a', 'b', 'c']
liste2 = [0, 20, 30]
liste3 = ['spam', 'eggs']
somme(liste1, liste2, liste3)
```

[3]:
```
['a', 'b', 'c', 0, 20, 30, 'spam', 'eggs']
```

Vous pouvez donc constater que `somme` peut fonctionner avec des objets de types différents : on peut lui passer des nombres, ou lui passer des listes. En fait, telle qu'elle est écrite, elle va fonctionner s'il est possible de faire `+` entre ses arguments. Ainsi, par exemple, on pourrait même faire :

[4]:
```
# Python sait faire + entre deux chaînes de caractères
somme('abc', 'def')
```

[4]:
```
'abcdef'
```

Mais par contre on ne pourrait pas faire

[5]:
```
# ceci va déclencher une exception à l'exécution
somme(12, [1, 2, 3])

# NOTE:
# auto-exec-for-latex has used instead:
##########
try:
    somme(12, [1, 2, 3])
except:
    print('OOPS')
##########
```

```
OOPS
```

Il est utile de remarquer que le typage de Python, qui existe bel et bien comme on le verra, est qualifié de dynamique parce que le type est attaché à un objet et non à la variable qui le référence. On aura bien entendu l'occasion d'approfondir tout ça dans le cours.

En Python, on fait souvent référence au typage sous l'appellation duck typing, de manière imagée :

> If it looks like a duck and quacks like a duck, it's a duck.

On voit qu'on se trouve dans une situation très différente de celle du programmeur C/C++, en ce sens que :

— à l'écriture du programme, il n'y aucun des surcoûts qu'on trouve avec C ou C++ en matière de définition de type ;
— aucun contrôle de type n'est effectué a priori par le langage au moment de la définition de la fonction `somme` ;
— par contre au moment de l'exécution, s'il s'avère qu'on tente de faire une somme entre deux types qui ne peuvent pas être additionnés, comme ci-dessus avec un entier et une liste, le programme ne pourra pas se dérouler correctement.

Il y a deux points de vue vis-à-vis de la question du typage.

Les gens habitués au typage statique se plaignent du typage dynamique en disant qu'on peut écrire des programmes faux et qu'on s'en rend compte trop tard - à l'exécution.

À l'inverse les gens habitués au typage dynamique font valoir que le typage statique est très partiel, par exemple, en C si on essaie d'écrire dans un pointeur `NULL`, le système d'exploitation ne le permet pas et le programme sort tout aussi brutalement.

Bref, selon le point de vue, le typage dynamique est vécu comme un inconvénient (pas assez de bonnes propriétés détectées par le langage) ou comme un avantage (pas besoin de passer du temps à déclarer le type des variables, ni à faire des conversions pour satisfaire le compilateur).

Vous remarquerez cependant à l'usage, qu'en matière de vitesse de développement, les inconvénients du typage dynamique sont très largement compensés par ses avantages.

Type hints

Signalons enfin que depuis python-3.5, il est possible d'ajouter des annotations de type, pour expliciter les suppositions qui sont faites par le programmeur pour le bon fonctionnement du code.

Nous aurons là encore l'occasion de détailler ce point dans le cours, signalons simplement que ces annotations sont totalement optionnelles, et que même lorsqu'elles sont présentes elles ne sont pas utilisées à l'exécution par l'interpréteur. L'idée est plutôt de permettre à des outils externes, comme par exemple `mypy`, d'effectuer des contrôles plus poussés concernant la correction du programme.

1.15 w1-s6-c1-calculette

Utiliser Python comme une calculette

Lorsque vous démarrez l'interprète Python, vous disposez en fait d'une calculette, par exemple, vous pouvez taper :

[1]:
```
20 * 60
```

[1]:
```
1200
```

Les règles de priorité entre les opérateurs sont habituelles, les produits et divisions sont évalués en premier, ensuite les sommes et soustractions :

[2]:
```
2 * 30 + 10 * 5
```

[2]:
```
110
```

De manière générale, il est recommandé de bien parenthéser ses expressions. De plus, les parenthèses facilitent la lecture d'expressions complexes.

Par exemple, il vaut mieux écrire ce qui suit, qui est équivalent mais plus lisible :

[3]:
```
(2 * 30) + (10 * 5)
```

[3]:
```
110
```

Attention, en Python3 la division `/` est la division usuelle, qui renvoie un flottant :

[4]:
```
48 / 5
```

[4]:
```
9.6
```

Rappelez-vous des opérateurs suivants qui sont très pratiques :

code	opération
`//`	quotient
`%`	modulo
`**`	puissance

[5]:
```
# calculer un quotient
48 // 5
```

[5]:
```
9
```

[6]:
```
# modulo (le reste de la division par)
48 % 5
```

[6]:
```
3
```

[7]:
```
# puissance
2 ** 10
```

[7]:
```
1024
```

Vous pouvez facilement faire aussi des calculs sur les complexes. Souvenez-vous seulement que la constante complexe que nous notons `i` en français se note `j` en Python, ce choix a été fait par le BDFL - alias Guido van Rossum - pour des raisons de lisibilité :

[8]:
```
# multiplication de deux nombres complexes
(2 + 3j) * 2.5j
```

[8]:
```
(-7.5+5j)
```

Aussi, pour entrer ce nombre complexe `j`, il faut toujours le faire précéder d'un nombre, donc ne pas entrer simplement `j` (qui serait compris comme un nom de variable, nous allons voir ça tout de suite) mais plutôt `1j` ou encore `1.j`, comme ceci :

[9]:
```
1j * 1.j
```

[9]:
```
(-1+0j)
```

Utiliser des variables

Il peut être utile de stocker un résultat qui sera utilisé plus tard, ou de définir une valeur constante. Pour cela on utilise tout simplement une affectation comme ceci :

[10]:
```
# pour définir une variable il suffit de lui assigner une valeur
largeur = 5
```

[11]:
```
# une fois la variable définie, on peut l'utiliser, ici comme un nombre
largeur * 20
```

[11]:
```
100
```

[12]:
```
# après quoi bien sûr la variable reste inchangée
largeur * 10
```

[12]:
```
50
```

Pour les symboles mathématiques, on peut utiliser la même technique :

[13]:
```
# pour définir un réel, on utilise le point au lieu d'une virgule en français
pi = 3.14159
2 * pi * 10
```

[13]:
```
62.8318
```

Pour les valeurs spéciales comme π, on peut utiliser les valeurs prédéfinies par la bibliothèque mathématique de Python. En anticipant un peu sur la notion d'importation que nous approfondirons plus tard, on peut écrire :

[14]:
```
from math import e, pi
```

Et ainsi imprimer les racines troisièmes de l'unité par la formule :

$r_n = e^{2i\pi\frac{n}{3}}$, pour $n \in \{0,1,2\}$

[15]:
```
n = 0
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
n = 1
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
n = 2
print("n=", n, "racine = ", e**((2.j*pi*n)/3))
```

```
n= 0 racine =  (1+0j)
n= 1 racine =  (-0.4999999999999998+0.8660254037844388j)
n= 2 racine =  (-0.5000000000000004-0.8660254037844384j)
```

Remarque : bien entendu il sera possible de faire ceci plus simplement lorsque nous aurons vu les boucles `for`.

Les types

Ce qui change par rapport à une calculatrice standard est le fait que les valeurs sont typées. Pour illustrer les trois types de nombres que nous avons vus jusqu'ici :

[16]:
```
# le type entier s'appelle 'int'
type(3)
```

[16]:
```
int
```

[17]:
```
# le type flottant s'appelle 'float'
type(3.5)
```

[17]:
```
float
```

[18]:
```
# le type complexe s'appelle 'complex'
type(1j)
```

[18]:
```
complex
```

Chaînes de caractères

On a également rapidement besoin de chaînes de caractères, on les étudiera bientôt en détail, mais en guise d'avant-goût :

[19]:
```
chaine = "Bonjour le monde !"
print(chaine)
```

```
Bonjour le monde !
```

Conversions

Il est parfois nécessaire de convertir une donnée d'un type dans un autre. Par exemple on peut demander à l'utilisateur d'entrer une valeur au clavier grâce à la fonction `input`, comme ceci :

[20]:
```
reponse = input("quel est votre âge ? ")

# NOTE:
# auto-exec-for-latex has used instead:
##########
reponse = '25'
##########
```

[21]:
```
# vous avez entré la chaîne suivante
print(reponse)
```

```
25
```

[22]:
```
# ici reponse est une variable, et son contenu est de type chaîne de caractères
type(reponse)
```

[22]:
```
str
```

Maintenant je veux faire des calculs sur votre âge, par exemple le multiplier par 2. Si je m'y prends naïvement, ça donne ceci :

[23]:
```
# multiplier une chaîne de caractères par deux ne fait pas ce que l'on veut,
# nous verrons plus tard que ça fait une concaténation
```

```
2 * reponse
```

[23]: '2525'

C'est pourquoi il me faut ici d'abord convertir la (valeur de la) variable `reponse` en un entier, que je peux ensuite doubler (assurez-vous d'avoir bien entré ci-dessus une valeur qui correspond à un nombre entier)

[24]:
```
# reponse est une chaine
# je la convertis en entier en appelant la fonction int()
age = int(reponse)
type(age)
```

[24]: int

[25]:
```
# que je peux maintenant multiplier par 2
2 * age
```

[25]: 50

Ou si on préfère, en une seule fois :

[26]:
```
print("le double de votre age est", 2*int(reponse))
```

```
le double de votre age est 50
```

Conversions - suite

De manière plus générale, pour convertir un objet en un entier, un flottant, ou une chaîne de caractères, on peut simplement appeler une fonction built-in qui porte le même nom que le type cible :

Type	Fonction
Entier	`int`
Flottant	`float`
Complexe	`complex`
Chaîne	`str`

Ainsi dans l'exemple précédent, `int(reponse)` représente la conversion de `reponse` en entier.

On a illustré cette même technique dans les exemples suivants :

[27]:
```
# dans l'autre sens, si j'ai un entier
a = 2345
```

[28]:
```
# je peux facilement le traduire en chaîne de caractères
str(2345)
```

[28]: '2345'

[29]:
```
# ou en complexe
complex(2345)
```

[29]:
```
(2345+0j)
```

Nous verrons plus tard que ceci se généralise à tous les types de Python, pour convertir un objet `x` en un type `bidule`, on appelle `bidule(x)`. On y reviendra, bien entendu.

Grands nombres

Comme les entiers sont de précision illimitée, on peut améliorer leur lisibilité en insérant des caractères `_` qui sont simplement ignorés à l'exécution.

[30]:
```
tres_grand_nombre = 23_456_789_012_345

tres_grand_nombre
```

[30]:
```
23456789012345
```

[31]:
```
# ça marche aussi avec les flottants
123_456.789_012
```

[31]:
```
123456.789012
```

Entiers et bases

Les calculettes scientifiques permettent habituellement d'entrer les entiers dans d'autres bases que la base 10.

En Python, on peut aussi entrer un entier sous forme binaire comme ceci :

[32]:
```
deux_cents = 0b11001000
print(deux_cents)
```

```
200
```

Ou encore sous forme octale (en base 8) comme ceci :

[33]:
```
deux_cents = 0o310
print(deux_cents)
```

```
200
```

Ou enfin encore en hexadécimal (base 16) comme ceci :

[34]:
```
deux_cents = 0xc8
print(deux_cents)
```

```
200
```

Pour d'autres bases, on peut utiliser la fonction de conversion `int` en lui passant un argument supplémentaire :

[35]:
```
deux_cents = int('3020', 4)
print(deux_cents)
```

```
200
```

Fonctions mathématiques

Python fournit naturellement un ensemble très complet d'opérateurs mathématiques pour les fonctions exponentielles, trigonométriques et autres, mais leur utilisation ne nous est pas encore accessible à ce stade et nous les verrons ultérieurement.

1.16 w1-s6-c2-affectation-operateurs

Affectations et Opérations (à la +=)

1.16.1 Complément - niveau intermédiaire

Il existe en Python toute une famille d'opérateurs dérivés de l'affectation qui permettent de faire en une fois une opération et une affectation. En voici quelques exemples.

Incrémentation

On peut facilement augmenter la valeur d'une variable numérique comme ceci :

[1]:
```
entier = 10

entier += 2
print('entier', entier)
```

```
entier 12
```

Comme on le devine peut-être, ceci est équivalent à :

[2]:
```
entier = 10

entier = entier + 2
print('entier', entier)
```

```
entier 12
```

Autres opérateurs courants

Cette forme, qui combine opération sur une variable et réaffectation du résultat à la même variable, est disponible avec tous les opérateurs courants :

[3]:
```
entier -= 4
print('après décrément', entier)
entier *= 2
print('après doublement', entier)
entier /= 2
print('mis à moitié', entier)
```

```
après décrément 8
après doublement 16
mis à moitié 8.0
```

Types non numériques

En réalité cette construction est disponible sur tous les types qui supportent l'opérateur en question. Par exemple, les listes (que nous verrons bientôt) peuvent être additionnées entre elles :

[4]:
```
liste = [0, 3, 5]
print('liste', liste)
```

```
liste += ['a', 'b']
print('après ajout', liste)
```

```
liste [0, 3, 5]
après ajout [0, 3, 5, 'a', 'b']
```

Beaucoup de types supportent l'opérateur `+`, qui est sans doute de loin celui qui est le plus utilisé avec cette construction.

Opérateurs plus abscons

Signalons enfin que l'on trouve aussi cette construction avec d'autres opérateurs moins fréquents, par exemple :

[5]:
```
entier = 2
print('entier:', entier)
entier **= 10
print('à la puissance dix:', entier)
entier %= 5
print('modulo 5:', entier)
```

```
entier: 2
à la puissance dix: 1024
modulo 5: 4
```

Et pour ceux qui connaissent déjà un peu Python, on peut même le faire avec des opérateurs de décalage, que nous verrons très bientôt :

[6]:
```
entier <<= 2
print('double décalage gauche:', entier)
```

```
double décalage gauche: 16
```

1.17 w1-s6-c3-precision-flottants

Notions sur la précision des calculs flottants

1.17.1 Complément - niveau avancé

Le problème

Comme pour les entiers, les calculs sur les flottants sont, naturellement, réalisés par le processeur. Cependant contrairement au cas des entiers où les calculs sont toujours exacts, les flottants posent un problème de précision. Cela n'est pas propre au langage Python, mais est dû à la technique de codage des nombres flottants sous forme binaire.

Voyons tout d'abord comment se matérialise le problème :

[1]:
```
0.2 + 0.4
```

[1]:
```
0.6000000000000001
```

Il faut retenir que lorsqu'on écrit un nombre flottant sous forme décimale, la valeur utilisée en mémoire pour représenter ce nombre, parce que cette valeur est codée en binaire, ne représente pas toujours exactement le nombre entré.

[2]:
```
# du coup cette expression est fausse, à cause de l'erreur d'arrondi
0.3 - 0.1 == 0.2
```

[2]:
```
False
```

Aussi, comme on le voit, les différentes erreurs d'arrondi qui se produisent à chaque étape du calcul s'accumulent et produisent un résultat parfois surprenant. De nouveau, ce problème n'est pas spécifique à Python, il existe pour tous les langages, et il est bien connu des numériciens.

Dans une grande majorité des cas, ces erreurs d'arrondi ne sont pas pénalisantes. Il faut toutefois en être conscient car cela peut expliquer des comportements curieux.

Une solution : penser en termes de nombres rationnels

Tout d'abord si votre problème se pose bien en termes de nombres rationnels, il est alors tout à fait possible de le résoudre avec exactitude.

Alors qu'il n'est pas possible d'écrire exactement $3/10$ en base 2, ni d'ailleurs $1/3$ en base 10, on peut représenter exactement ces nombres dès lors qu'on les considère comme des fractions et qu'on les encode avec deux nombres entiers.

Python fournit en standard le module `fractions` qui permet de résoudre le problème. Voici comment on pourrait l'utiliser pour vérifier, cette fois avec succès, que $0.3 - 0.1$ vaut bien 0.2. Ce code anticipe sur l'utilisation des modules et des classes en Python, ici nous créons des objets de type `Fraction` :

[3]:
```
# on importe le module fractions, qui lui-même définit le symbole Fraction
from fractions import Fraction

# et cette fois, les calculs sont exacts, et l'expression retourne bien True
Fraction(3, 10) - Fraction(1, 10) == Fraction(2, 10)
```

[3]:
```
True
```

Ou encore d'ailleurs, équivalent et plus lisible :

[4]:
```
Fraction('0.3') - Fraction('0.1') == Fraction('2/10')
```

[4]:
```
True
```

Une autre solution : le module decimal

Si par contre vous ne manipulez pas des nombres rationnels et que du coup la représentation sous forme de fractions ne peut pas convenir dans votre cas, signalons l'existence du module standard `decimal` qui offre des fonctionnalités très voisines du type `float`, tout en éliminant la plupart des inconvénients, au prix naturellement d'une consommation mémoire supérieure.

Pour reprendre l'exemple de départ, mais en utilisant le module decimal, on écrirait alors :

[5]:
```
from decimal import Decimal

Decimal('0.3') - Decimal('0.1') == Decimal('0.2')
```

[5]:
```
True
```

Pour aller plus loin

Tous ces documents sont en anglais :

— un tutoriel sur les nombres flottants ;
— la documentation sur la classe Fraction ;
— la documentation sur la classe Decimal ;
— une présentation plus fouillée sur l'encodage des flottants (en anglais) ; ce dernier document, très bien fait, ne dépend pas du langage Python mais illustre le standard IEE-754 sur des exemples concrets.

1.18 w1-s6-c4-entiers-bit-a-bit

Opérations bit à bit (bitwise)

1.18.1 Compléments - niveau avancé

Les compléments ci-dessous expliquent des fonctions évoluées sur les entiers. Les débutants en programmation peuvent sans souci sauter cette partie en cas de difficultés.

Opérations logiques : ET `&`, OU `|` et OU exclusif `^`

Il est possible aussi de faire des opérations bit à bit sur les nombres entiers. Le plus simple est de penser à l'écriture du nombre en base 2.

Considérons par exemple deux entiers constants dans cet exercice

[1]:
```
x49 = 49
y81 = 81
```

Ce qui nous donne comme décomposition binaire :

$$\begin{array}{rcrcll} x49 & = & 49 & = & 32+16+1 & \rightarrow \quad (0,1,1,0,0,0,1) \\ y81 & = & 81 & = & 64+16+1 & \rightarrow \quad (1,0,1,0,0,0,1) \end{array}$$

Pour comprendre comment passer de $32+16+1$ à $(0,1,1,0,0,0,1)$ il suffit d'observer que :

$$32+16+1 = 0*2^6+1*2^5+1*2^4+0*2^3+0*2^2+0*2^1+1*2^0$$

ET logique : opérateur `&` L'opération logique `&` va faire un et logique bit à bit entre les opérandes, ainsi

[2]:
```
x49 & y81
```

[2]:
```
17
```

Et en effet :

$$\begin{array}{rcl} x49 & \rightarrow & (0,1,1,0,0,0,1) \\ y81 & \rightarrow & (1,0,1,0,0,0,1) \\ x49 \ \& \ y81 & \rightarrow & (0,0,1,0,0,0,1) \rightarrow 16+1 \rightarrow 17 \end{array}$$

OU logique : opérateur `|` De même, l'opérateur logique `|` fait simplement un ou logique, comme ceci :

[3]:
```
x49 | y81
```

[3]:
```
113
```

On s'y retrouve parce que :

$$\begin{array}{rcl} x49 & \rightarrow & (0,1,1,0,0,0,1) \\ y81 & \rightarrow & (1,0,1,0,0,0,1) \\ x49 \mid y81 & \rightarrow & (1,1,1,0,0,0,1) \rightarrow 64+32+16+1 \rightarrow 113 \end{array}$$

OU exclusif : opérateur `^` Enfin, on peut également faire la même opération à base de ou exclusif avec l'opérateur `^` :

[4]:
```
x49 ^ y81
```

[4]:
```
96
```

Je vous laisse le soin de décortiquer le calcul à titre d'exercice (le ou exclusif de deux bits est vrai si et seulement si exactement une des deux entrées est vraie).

Décalages

Un décalage à gauche de, par exemple, 4 positions, revient à décaler tout le champ de bits de 4 cases à gauche (les 4 nouveaux bits insérés sont toujours des 0). C'est donc équivalent à une multiplication par $2^4 = 16$:

[5]:
```
x49 << 4
```

[5]:
```
784
```

$$\begin{array}{rcl} x49 & \rightarrow & (0,1,1,0,0,0,1) \\ x49 << 4 & \rightarrow & (0,1,1,0,0,0,1,0,0,0,0) \rightarrow 512+256+16 \rightarrow 784 \end{array}$$

De la même manière, le décalage à droite de n revient à une division par 2^n (plus précisément, le quotient de la division) :

[6]:
```
x49 >> 4
```

[6]:
```
3
```

$$\begin{array}{rcl} x49 & \rightarrow & (0,1,1,0,0,0,1) \\ x49 >> 4 & \rightarrow & (0,0,0,0,0,1,1) \rightarrow 2+1 \rightarrow 3 \end{array}$$

Une astuce

On peut utiliser la fonction built-in `bin` pour calculer la représentation binaire d'un entier. Attention, la valeur de retour est une chaîne de caractères de type `str` :

[7]:
```
bin(x49)
```

[7]:
```
'0b110001'
```

Dans l'autre sens, on peut aussi entrer un entier directement en base 2 comme ceci :

[8]:
```
x49bis = 0b110001
x49bis == x49
```

[8]:
```
True
```

Ici, comme on le voit, `x49bis` est bien un entier.

Pour en savoir plus

Section de la documentation Python.

1.19 w1-s6-x1-flottants

Estimer le plus petit (grand) flottant

1.19.1 Exercice - niveau basique

Le plus petit flottant

En corollaire de la discussion sur la précision des flottants, il faut savoir que le système de codage en mémoire impose aussi une limite. Les réels très petits, ou très grands, ne peuvent plus être représentés de cette manière.

C'est notamment très gênant si vous implémentez un logiciel probabiliste, comme des graphes de Markov, où les probabilités d'occurrence de séquences très longues tendent très rapidement vers des valeurs extrêmement petites.

Le but de cet exercice est d'estimer la valeur du plus petit flottant qui peut être représenté comme un flottant. Pour vous aider, voici deux valeurs :

[1]:
```
10**-320
```

[1]:
```
1e-320
```

[2]:
```
10**-330
```

[2]:
```
0.0
```

Comme on le voit, 10^{-320} est correctement imprimé, alors que 10^{-330} est, de manière erronée, rapporté comme étant nul.

Notes :

- À ce stade du cours, pour estimer le plus petit flottant, procédez simplement par approximations successives.
- Sans utiliser de boucle, la précision que vous pourrez obtenir n'est que fonction de votre patience, ne dépassez pas 4 à 5 itérations successives :)
- Il est par contre pertinent d'utiliser une approche rationnelle pour déterminer l'itération suivante (par opposition à une approche "au petit bonheur"). Pour ceux qui ne connaissent pas, nous vous recommandons de vous documenter sur l'algorithme de dichotomie.

[3]:
```
10**-325
```

[3]:
```
0.0
```

Voici quelques cellules de code vides ; vous pouvez en créer d'autres si nécessaire, le plus simple étant de taper `Alt+Enter`, ou d'utiliser le menu "Insert -> Insert Cell Below"

[4]:
```
# vos essais successifs ici
```

[5]:
```
.24*10**-323
```

[5]:
```
0.0
```

Le plus grand flottant

La même limitation s'applique sur les grands nombres. Toutefois, cela est un peu moins évident, car comme toujours il faut faire attention aux types :

[6]:
```
10**450
```

[6]:
```
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000000
```

Ce qui passe très bien car j'ai utilisé un `int` pour l'exposant. Dans ce premier cas Python calcule le résultat comme un `int`, qui est un type qui n'a pas de limitation de précision (Python utilise intelligemment autant de bits que nécessaire pour ce genre de calculs).

Par contre, si j'essaie de faire le même calcul avec un exposant flottant, Python essaie cette fois de faire son calcul avec un flottant, et là on obtient une erreur :

[ ]:
```
10**450.0

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

On peut d'ailleurs remarquer que le comportement ici n'est pas extrêmement cohérent, car avec les petits nombres Python nous a silencieusement transformé 10^{-330} en 0, alors que pour les grands nombres, il lève une exception (nous verrons les exceptions plus tard, mais vous pouvez dès maintenant remarquer que le comportement est différent dans les deux cas).

Quoi qu'il en soit, la limite pour les grands nombres se situe entre les deux valeurs 10^{300} et 10^{310}. On vous demande à nouveau d'estimer comme ci-dessus une valeur approchée du plus grand nombre qu'il soit possible de représenter comme un flottant.

[7]:
```
10**300.
```

[7]:
```
1e+300
```

[ ]:
```
10**310.

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

[8]:
```
# vos essais successifs ici
```

1.19.2 Complément - niveau avancé

En fait, on peut accéder à ces valeurs minimales et maximales pour les flottants comme ceci

[9]:
```
import sys
print(sys.float_info)
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, m
  in=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant
  _dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Et notamment, comme expliqué ici.

[10]:
```
print("Flottant minimum", sys.float_info.min)
print("Flottant maximum", sys.float_info.max)
```

```
Flottant minimum 2.2250738585072014e-308
Flottant maximum 1.7976931348623157e+308
```

Sauf que vous devez avoir trouvé un maximum voisin de cette valeur, mais le minimum observé expérimentalement ne correspond pas bien à cette valeur.

Pour ceux que cela intéresse, l'explication à cette apparente contradiction réside dans l'utilisation de nombres dénormaux.