Jean Paul Roy

Python

Apprentissage actif

pour l'étudiant et le futur enseignant

2e édition

ellipses

Première partie

L'essentiel du langage

1 L'environnement Python

*J'ai essayé de ne pas trop détailler l'**environnement intégré de programmation** ou IDE*[1]*, vous trouverez sur le Web beaucoup de tutoriaux à cet effet. Suivant la formation que vous suivrez, vous serez sous IDLE, Spyder, Jupyter, Emacs, Eclipse, PyCharm, XCode ou Visual Studio, que sais-je ? Il est impossible de les passer tous en revue. Pour les débutants,* **IDLE** *(par les créateurs de Python) et* **Jupyter** *(plus évolué, avec la plateforme Anaconda) sont de bons choix, avec* **Visual Studio Code** *pour les plus hardis. Nous verrons comment rédiger et exécuter avec eux un programme.*

1.1 Installation de la distribution Python standard avec IDLE

La maison-mère **www.python.org** distribue le langage Python avec un petit éditeur intégré **IDLE**, fournissant le minimum requis pour programmer en Python. Il s'agit sans doute de la solution la plus simple au niveau du lycée en pré-BAC. Ultérieurement, il faudra disposer d'un environnement élargi pour la sience des données, tel celui d'Anaconda qui reste utilisable par des débutants.

La distribution standard se trouve sur **www.python.org/downloads** qui vous proposera la version adaptée à votre machine. Une fois l'installation terminée, fabriquez un raccourci vers le programme IDLE, sur le bureau ou dans un menu de démarrage. C'est ce programme qui va lancer Python dans une console avec un caractère d'invite (*prompt*) `>>>`. Par exemple, aux couleurs près :

```
1 Python 3.14.0, Oct 7 2025, on darwin          # ← darwin = Apple
2 >>> 2 + 3 * 4
3 14
4 >>> from math import sqrt                     # la racine carrée
5 >>> sqrt(2) ** 2                             # un calcul approché :-)
6 2.0000000000000004
```

1. IDE = *Integrated Development Environment*, comprenant les outils essentiels pour programmer dont en premier lieu un éditeur de texte et une console d'exécution.

La documentation de Python est un fichier `.html` situé dans le répertoire de l'application Python, à côté de IDLE. Une aide en ligne sommaire est disponible à la console avec la fonction `help`.

```
>>> help(sqrt)                                          # aide en ligne
Help on built-in function sqrt in module math:
sqrt(x, /)
    Return the square root of x.
```

Le caractère dièse `#` débute un *commentaire* sur la ligne, ignoré par Python. On travaille assez peu avec cette ligne de commande (*shell* ou *toplevel*) assez spartiate, plutôt pour de petits tests de programmes situés dans une fenêtre d'édition, que vous obtiendrez soit par le menu *File/New File...*, soit en ouvrant un fichier Python d'extension `.py` par *File/Open...*. Ouvrons une nouvelle fenêtre d'édition et dans cette fenêtre définissons une fonction `calcul` à deux paramètres `a` et `b`, fonction dont le résultat sera simplement le nombre $a + 2b$.

essai.py

```
# essai.py

def calcul(a,b):
    return a + 2*b

print('calcul(2,5.3) =',calcul(2,5.3))
```

Si maintenant vous appuyez sur la touche F5 de votre ordinateur, ou si vous invoquez le menu *Run/Run Module*, il vous sera demandé de sauvegarder le contenu de l'éditeur sur disque, ici sous le nom `essai.py`. Ce contenu sera ensuite analysé sommairement pour détecter d'éventuelles fautes de syntaxe, la mémoire sera remise à neuf (*Restart Shell*) et enfin le contenu sera évalué de haut en bas. Son évaluation produira au passage des affichages de résultats à la console toplevel.

```
========= RESTART: /Users/roy/Documents/prog-python/essai.py ======
calcul(2,5.3) = 12.6
```

Les **tests** du programme se font plutôt **dans l'éditeur** qu'au toplevel, de manière à ce qu'ils soient exécutés automatiquement sans avoir à les retaper à chaque modification du programme. Nous verrons au § 6.5 comment rédiger un *script*, petit programme exécutable au terminal en-dehors d'IDLE.

Sur Mac, le répertoire `"/Applications/Python 3.x/"` contient les logiciels pour l'utilisateur, dont IDLE. Mais les entrailles de la bête se situent à un niveau où seul le spécialiste ira mettre le nez, par exemple (sur Mac) :

`/Library/Frameworks/Python.framework/Versions/Current` .

dans le dossier `bin` duquel on trouve les commandes au Terminal `python3`, `idle3` et `pip3`. Cet environnement intégré minimal IDLE est rudimentaire mais plus que suffisant pour une première initiation à Python. Il est trop spartiate pour un développement avancé ou professionnel.

1.2 Installation de la distribution Anaconda

Pour les scientifiques, nous conseillons plutôt la **distribution Anaconda**.

ANACONDA

La société *Anaconda* (spécialisée dans l'analyse des données : *data science*) propose en effet une version gratuite mais très complète d'un environnement pour programmer en Python. Vous avez deux choix possibles pour l'installer :

- soit la version complète **Anaconda** (> 4 Go) contenant Python, de très nombreuses bibliothèques scientifiques (`numpy`, *etc.*), une interface graphique *Navigator*, et le gestionnaire `conda` pour les mises à jour.
- soit une version **Miniconda** (< 1 Go) contenant Python et quelques bibliothèques seulement, avec `conda` pour installer les autres.

  `https://docs.anaconda.com/distro-or-miniconda/`

Si vous avez suffisamment d'espace disque, téléchargez et installez *Anaconda Distribution*, cela vous simplifiera la vie. Prenez éventuellement un compte sur la plateforme *Anaconda Cloud*, vous pourrez profiter de nombreux contenus. Utilisez un mot de passe fort.

Vérifions si l'installation s'est bien passée. Je suis sur Mac, vous voudrez donc adapter à votre système d'exploitation Windows ou Linux. Ouvrons l'application *Terminal* située sur Mac dans le dossier `Applications/Utilitaires/`. Sur Windows, lancez l'application *Anaconda Prompt* qui vient d'être installée dans le menu `Démarrer`. Vous devez voir `(base)` au début de la ligne de commande. Je suppose hardiment que tout va bien : vous êtes dans l'environnement `base` d'Anaconda. Fermez le Terminal et poussez un soupir de soulagement.

Sur mon Mac, le Python installé par Anaconda vit dans le répertoire... euh, lequel ? Essayons au Terminal :

```
11 (base) $ which python3        # en Unix au Terminal sur Mac et Linux,
12 /Users/roy/opt/anaconda3/bin/python3      #  "where" sous Windows...
```

Donc les commandes `python3` et `idle3` concernent bien le Python d'Anaconda, celui de la maison-mère étant, lui, lancé à la souris par l'icône de l'application IDLE dans `"/Applications/Python 3.x"`. Donc deux Python co-existent paisiblement... ou pas. Il faut savoir avec lequel on travaille.

1.2.1 Le gestionnaire de paquets : `conda`

Nous serons amenés un jour ou l'autre à devoir utiliser une bibliothèque logicielle extérieure écrite en Python – on parle de **paquet** (*package*). Pour l'installer sous Anaconda, nous utiliserons au terminal un **gestionnaire de paquets et d'environnements** (*environment and package manager*) nommé `conda` [2]. Il existe un autre gestionnaire de paquets `pip` et d'environnements `venv`, mais nous veillerons à nous en tenir à `conda` pour éviter les soucis [3] ! Pour savoir si `conda` est bien installé, demandons son numéro de version. Il est recommandé de tenir `conda` à jour (`conda update conda`).

```
13 (base) $ conda update --all                    #  mise à jour globale !
14 Executing transaction: done
15 (base) $ conda --version
16 24.11.0
17 (base) $ python3 --version
18 3.12.8                          #  un poil en retard sur la maison-mère
```

Devant le prompt `$` de mon Terminal est apparu `(base)` qui me dit dans quel **environnement** de `conda` je me trouve. Je suis dans l'environnement de base, avec un grand nombre de paquets logiciels que je peux immédiatement utiliser. Pour savoir lesquels, je demande leur liste avec `conda list` (il y en a *beaucoup* !). En voici seulement quelques-uns :

```
19 (base) $ conda list
20 packages in environment at /opt/anaconda3:
21 # Name                    Version
22 anaconda-navigator        2.6.3
23 conda                     24.11.0
24 ipython                   8.30.0
25 jupyter                   1.1.1
26 matplotlib                3.9.3
27 notebook                  7.3.1
28 numpy                     1.26.4
```

Ne vous fiez pas aux numéros de version, ils évoluent au fil des mois. Vous pouvez en théorie travailler dans l'environnement `(base)` pour du Python pur, mais si vous commencer à importer des paquets [4] extérieurs, il se peut que tel paquet A

2. Documentation sur `https://conda.io/docs/`. L'application *Anaconda Navigator* offre une interface graphique pour `conda`, mais il est souvent plus rapide et sain de travailler au terminal.

3. Voir : `https://www.anaconda.com/blog/using-pip-in-a-conda-environment`.

4. Ces paquets peuvent être des modules ou des répertoires de modules et de paquets.

utilise un autre paquet P en version 7.2 alors qu'un paquet B demande le paquet P en version 6.5, vous voyez le problème qui surgit. C'est encore pire s'ils exigent des versions de Python différentes !

Imaginons que nous souhaitions programmer une simulation avec le *moteur de calcul physique* `Pymunk`. Il est sain de créer avec `conda create` un nouvel environnement – par exemple de nom `simul` – et d'y installer ce paquet (qui ne fait pas partie des paquets initiaux de notre distribution Anaconda).

```
29 (base) $ conda create --name simul                    # --name ou -n
30 ...environment location: /Users/roy/opt/anaconda3/envs/simul
```

Les environnements se trouvent donc dans le répertoire `anaconda3/envs/`, vérifiez-le en ouvrant ce répertoire ou mieux avec la commande `conda info --envs` où `--envs` peut s'abréger en `-e`, vous verrez que l'environnement courant est précédé d'une étoile `*`. Plaçons-nous dans l'environnement `simul` encore vide avec la commande `conda activate`.

```
31 (base) $ conda activate simul
32 (simul) $ conda list                                  # simul est vide !
33 (simul) $
```

Maintenant que nous sommes dans l'environnement `simul`, installons le moteur physique `Pymunk` dont la page Web est `https://www.pymunk.org`. Celle-ci me propose d'installer le paquet `pymunk` version 7.0.1 avec l'un des gestionnaires `pip` ou `conda`. Étant sous Anaconda, j'opte pour un `conda install` et seulement en désespoir de cause aurais-je eu recours à `pip` (qui fait partie des paquets d'Anaconda[5]). On me dit même d'utiliser le **canal** `conda-forge`. J'utilise ci-dessous la manière traditionnelle avec `--channel` au lieu de simplement `-c` :

```
34 (simul) $ conda install --channel conda-forge pymunk
35 ...
36 Executing transaction: done
```

Pour installer un paquet, Anaconda utilise en effet des **canaux de recherche** (*channels*) et de manière générale, si aucun d'eux ne trouve ce que je cherche, j'essaye le canal `conda-forge` qui contient de nombreux paquets scientifiques. Ceci dit, avec un navigateur Web, j'aurais pu mettre la main sur la page du canal `anaconda.org/conda-forge` et y trouver `pymunk`, pour obtenir une requête `conda install conda-forge::pymunk` avec une troisième syntaxe équivalente.

5. Les gestionnaires `pip` et `conda` sont parfois en conflit, surtout si l'on ne s'assure pas qu'on utilise les bons, au cas où l'on conserve plusieurs distributions de Python !

```
(simul) $ conda list
...
pymunk          7.0.1           py312hea69d52_0         conda-forge
python          3.12.8          hc22306f_1_cpython      conda-forge
...
```

Vérifions au Terminal que tout fonctionne bien en essayant d'importer `pymunk` au sein de l'environnement `simul`.

```
(simul) $ python3
>>> import pymunk
>>> pymunk.version                          # ok, l'importation est correcte
'7.0.1'
>>> quit()
(simul) $
```

Ce dépôt `conda-forge` me semble plus qu'intéressant, je l'ajoute donc en tête (`--add`) ou en queue (`--append`) à la liste des canaux qui seront essayés dans l'ordre lors d'une nouvelle installation.

```
(simul) $ conda config --add channels conda-forge   # ajouter en tête
(simul) $ conda config --get channels               # puis vérifier
--add channels 'defaults'      # lowest priority
--add channels 'conda-forge'   # highest priority
```

Une autre manière de le vérifier consiste à demander `conda config --show` montrant la configuration de `conda`. Les canaux de `defaults` sont ceux hébergés par la société Anaconda, tandis que `conda-forge` est une communauté libre sur *GitHub*. Nous n'aurons pas besoin d'autres canaux dans ce livre.

Vous pouvez **supprimer** un paquet `paq` dans l'environnement courant avec la commande `conda remove paq`, ou même l'environnement `simul` tout entier à partir de l'environnement `base` avec `conda env remove --name simul`. Pour remonter dans `base`, il suffit de désactiver l'environnement courant.

```
(simul) $ conda deactivate          # retour à l'environnement de base
(base) $ python3
>>> import pymunk
ModuleNotFoundError: No module named 'pymunk'      # pas dans base !
>>> quit()
(base) $
```

Oui, il faut mémoriser quelques commandes au terminal, mais on s'y fait vite. Au besoin, demandez à Google la dernière version de la « **conda cheat sheet** » avec votre navigateur et la vie pétillera ☺. Ou pas, si vous êtes allergique au Terminal, auquel cas vous voudrez passer par *Anaconda Navigator* (page suivante).

Remarque — S'il vous arrive de perdre le prompt `(base)` au Terminal, vérifiez que `conda` est bien accessible dans le PATH, et ré-initialisez-le avec `conda init`.

1.2.2 IDLE est aussi intégré dans Anaconda !

L'éditeur et toplevel IDLE (§ 1.1), conçu par la maison-mère `www.python.org`, a été intégré à Anaconda pour faire du travail simple, ce qui montre qu'il n'est pas utile de télécharger la distribution de la maison-mère pour profiter de IDLE qui est accessible en ligne de commande sur Anaconda.

```
58 (base) $ which idle3          #  où est l'exécutable IDLE maintenant ?
59 /Users/roy/opt/anaconda3/bin/idle3                #  dans Anaconda !
60 (base) $ idle3&               #  j'ouvre une fenêtre IDLE
```

L'esperluette `&` à la fin du mot `idle3` est une convention Unix pour signifier que l'exécution de IDLE doit se faire *en tâche de fond*, permettant d'entrer d'autres commandes au Terminal qui ne sera pas bloqué jusqu'à la sortie de IDLE.

Attention donc à **ne pas vous mélanger les pinceaux** si vous gardez la distribution de la maison-mère en même temps qu'Anaconda ! Il y aura deux exécutables `idle3` dans la machine, par exemple chez moi :

```
/Library/Frameworks/Python.framework/Versions/Current/bin/idle3 (*)
/Users/roy/opt/anaconda3/bin/idle3                              (**)
```

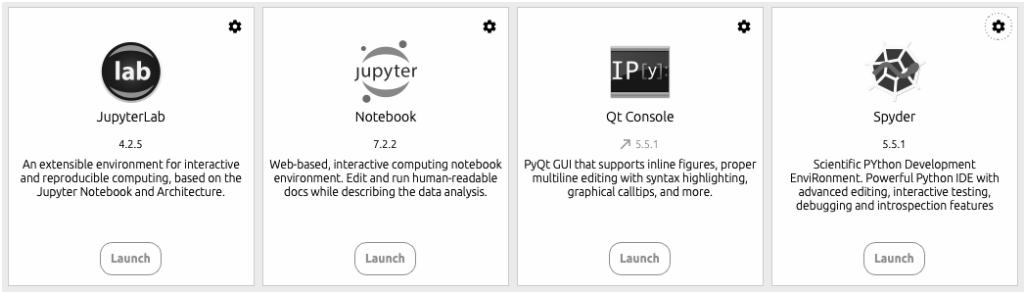
La différence inestimable entre les deux est que le premier (*) ne bénéfiera **pas** des paquets installés dans Anaconda, alors que le second (**) oui. Si je demande `import numpy` dans le premier, j'aurai une erreur *No module named 'numpy'* et je devrai alors l'installer avec `pip3` (celui de (*) !), alors que le second considère que `numpy` est déjà présent dans Anaconda – comme on le vérifie avec `conda list`. Il est possible de gérer des environnements virtuels dans (*) avec `venv` et `pip`, je n'en parlerai pas, vous pouvez demander à une IA de vous montrer les différences entre `conda` et `venv`. N'hésitez pas à utiliser une IA en mode gratuit : *ChatGPT*, *Claude*, *DeepSeek*, *Mistral*, *etc.*

Il peut être intéressant de réserver un environnement `idle` pour faire dans IDLE (d'Anaconda) des programmes ne nécessitant que les paquets initiaux d'Anaconda. Travailler directement dans `base` est possible mais assez déconseillé.

1.2.3 L'interface graphique : Anaconda Navigator

Ananconda propose une application **Anaconda Navigator** que vous mettrez dans vos raccourcis. Si vous avez installé *Miniconda*, utilisez `conda` pour installer le paquet `anaconda-navigator`.

Une fois Navigator lancé, sa fenêtre s'ouvre, dans laquelle vous trouverez différents logiciels (*JupyterLab*, *Notebook*, *Qt Console*, *Spyder*, *VS Code etc.*).



Parmi eux se trouvent **Jupyter Notebook** et **Spyder** qui vous serviront à rédiger des programmes si vous n'avez pas opté pour IDLE.

Vous pouvez gérer les environnements avec Navigator, plutôt qu'en ligne de commande avec `conda`. En cliquant sur le bouton *Environments* sur la gauche de la fenêtre du Navigator, vous allez voir apparaître nos trois environnements courants : `base`, `simul` et `idle` avec au-dessous cinq boutons : pour créer un nouvel environnement, dupliquer, importer[6], sauvegarder ou supprimer un autre. Sur la droite, vous visualisez les paquets de chaque environnement, avec la possibilité d'en rajouter ou d'en supprimer. Vous avez même le bouton *Channels* pour gérer les canaux de recherche.

Mieux vaut avoir lu la section 1.2.1 précédente pour bien saisir le sens des manipulations, et nous vous renvoyons aux tutoriels en ligne :

`https://docs.anaconda.com/navigator/tutorials/`

notamment aux chapitres *Managing environments*, *packages* et *channels*. Vous trouverez aussi des vidéos sur YouTube. Je n'utilise pas trop *Navigator*.

1.2.4 L'éditeur Spyder

Outre la gestion des environnements et des canaux, Navigator permet de lancer des logiciels, dont certains sont de véritables usines à traitement des données, complexes à utiliser sans un livre spécialisé sous la main. La tendance actuelle

6. Vous pouvez télécharger ou transmettre à autrui une description d'environnement sous la forme d'un fichier `.yml`, demandez à votre IA de vous parler des fichiers YAML. L'analogue pour `pip` est le fichier `requirements.txt`.

est à l'Intelligence Artificielle, qui a pénétré aussi le monde des codeurs, et il existe déjà un *Anaconda AI Navigator* collaboratif permettant d'installer les outils d'IA.

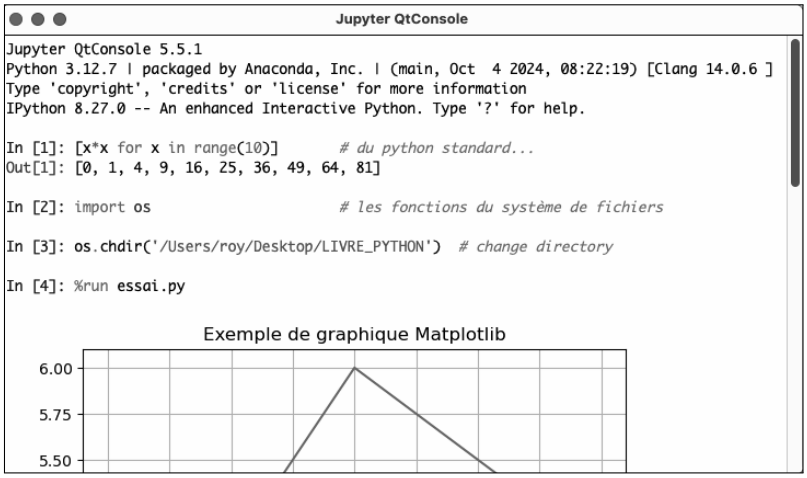
Parmi tous ces logiciels, l'éditeur **Spyder** a des aspects plaisants, il est maintenu par une équipe indépendante dévouée et ambitieuse, et peut se lancer *via* Navigator : c'est un paquet d'Anaconda présent dans l'environnement `base`.

Remarque — Sur `https://www.spyder-ide.org`, il est possible de télécharger Spyder comme **application autonome**. De nombreux paquets pour l'analyse des données sont déjà intégrés (`numpy`, `matplotlib`, `pandas`, *etc.*).

En ouvrant la fenêtre de Spyder, vous verrez sa console toplevel (à droite en bas) basée sur **IPython**. Ce toplevel est plus complexe que celui d'IDLE mais avec un peu de pratique, il s'avère productif. Il permet de voir les graphiques produits soit dans des fenêtres séparées, soit directement au toplevel.

Spyder est la conjonction d'une console IPython, d'un éditeur de programmes pouvant utiliser des cellules (et destiné à évoluer à terme vers le concept de **notebook** comme dans Jupyter), et d'une vision sur le système de fichiers. Cet éditeur gratuit vaut néanmoins le détour même s'il a des soucis par moments, en général résolubles *via* un forum dédié et réactif. Vous trouverez de l'aide sur le site de Spyder et sur YouTube, avec des tutoriaux et des vidéos de prise en main.

1.2.5 La console Qt : juste le goût de IPython pur



Pour avoir une idée du fonctionnement de IPython, lançons avec Navigator la **Console Qt** : `https://qtconsole.readthedocs.io`. On peut aussi au Terminal entrer la commande : `jupyter qtconsole`. Une simple fenêtre toplevel surgit, de nom *Jupyter QtConsole*, sans éditeur.

Vous ne pouvez qu'entrer des demandes de calcul ou d'affichage de graphiques, ainsi qu'exécuter un fichier avec la **commande magique**[7] `%run`. Vous produisez et modifiez ce fichier avec un éditeur de programmes de votre choix (Spyder, Visual Studio, PyCharm, etc.), et vous exécutez dans la Qt-Console avec `%run`.

Le site plus général de IPython est `https://ipython.org`. La *console IPython* de Spyder est une variante de **QtConsole**. Il s'agit d'un lieu interactif où l'on peut avec la commande magique `load_ext` charger des extensions (nous utiliserons `load_ext cython`), échanger des données entre plusieurs langages dont majoritairement Python, et qui sera utilisé simplement comme une calculette évoluée ou bien pour visualiser les affichages et graphiques d'un programme lors de son exécution. Dans Spyder, la pression de l'icône *Exécuter* (**F5**) dans l'éditeur déclenche à la console IPython un appel à la commande magique `%runfile`.

Les *notebooks* de Jupyter sont aussi basés sur IPython, mais apportent des fonctionnalités supplémentaires : multi-langages (Python, R, Julia, Javascript, Matlab, Fortran...), édition en TEX, conversion vers PDF ou HTML, *etc.*).

Voir [ROSS], [VAU], [VAN] pour la prise en main de IPython et Jupyter en science des données. Ainsi que les fabuleuses ressources du Web et des IA.

1.2.6 Le projet Jupyter

Les travaux autour de IPython, lancé en 2001, s'élargiront pour aboutir vers 2014 au **projet Jupyter** `https://jupyter.org` du nom de trois langages de programmation scientifique : **Ju**lia, **Pyt**hon et **R**. Il a aussi été influencé par les *notebooks* de Mathematica, créé vers 1988 par Steven Wolfram. L'ambition de Mathematica, reprise par Jupyter, était de créer un environnement de production de documents électroniques contenant à la fois du texte bien mis en forme et du code exécutable produisant des résultats qui s'intègrent dynamiquement au texte.

La naissance de Jupyter visait à faciliter la rédaction de ces *carnets* électroniques ou **notebooks** contenant à la fois du texte enrichi (avec *Markdown* et LATEX), des graphiques résultats de calculs, et du code – Python ou autre – exécutable[8]. Un *notebook* Python – carnet[9] de notes – est un fichier d'extension `.ipynb`[10] organisé en **cellules** plus puissantes que celles de l'éditeur Spyder. Vous pourrez ainsi rédiger un texte scientifique (un exposé de maths, de physique ou

7. Les *commandes magiques* de IPython, préfixées par `%` pour une ligne et `%%` pour une cellule, permettent d'effectuer des actions. On y trouve `%run`, `%time`, `%timeit`, `%load`, `%load_ext`, `%gui`, `%matplotlib`, `%pwd`, `%ls`, `%%bash`, `%%latex`, `%%markdown`.... Demandez `%lsmagic` au toplevel.

8. Même si Python est le principal, une centaine d'autres langages sont disponibles. Vous les trouverez en cherchant « Jupyter kernels » dans votre moteur de recherche ou avec une IA.

9. Une application sur iPad se nomme *Carnets*.

10. Un fichier `.ipynb` est sauvegardé au format JSON et non comme un fichier usuel `.py`. Il faut au moins un *Notebook Viewer* pour le consulter et Jupyter pour le modifier.

de biologie par exemple) contenant des calculs interactifs. Pédagogiquement, cela permet de joindre la *rédaction* – trop souvent délaissée – au *codage*.

La force de Jupyter est de **mélanger l'éditeur de texte et la console**[11] pour former ce que l'on nommait déjà en LISP au début des années 1980 un *éditeur-toplevel*, initié par le celèbre éditeur Emacs.

Les *notebooks* (carnets) de Jupyter

Un carnet (notebook) contient les ***entrées et sorties*** *d'une session interactive ainsi que du* ***texte narratif*** *qui accompagne le code mais qui n'est pas destiné à être exécuté. La* ***sortie enrichie*** *générée par l'exécution du code, y compris avec du HTML, des images, vidéos et graphiques, est intégrée dans le carnet, ce qui en fait un document complet et autonome issu d'un calcul.*
(jupyter-notebook.readthedocs.io)

Une documentation des *carnets* se trouve en anglais sur `(*)` très complet, et en français sur `(**)` pour l'étudiant en sciences physiques :

`https://jupyter-notebook.readthedocs.io` `(*)`
`https://pyspc.readthedocs.io` `(**)`

Commençons par installer dans votre environnement un petit paquet trouvé dans `https://pypi.org/org/jupyter`, et permettant de **franciser** Jupyter.

```
61 (base) $ conda install -c conda-forge jupyterlab-language-pack-fr-FR
```

Ceci fait, vous disposez de deux manières pour ouvrir **Jupyter Notebook** :

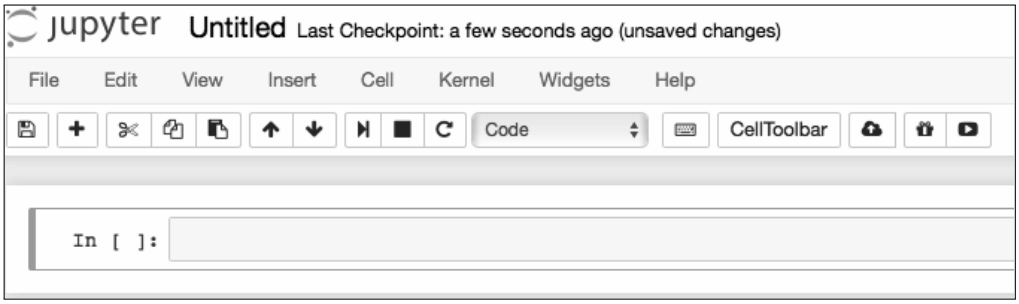
- Soit vous le lancez au Terminal dans l'environnement courant avec la commande `jupyter notebook` et au besoin en installant le paquet `jupyter`.
- Soit avec *Anaconda Navigator* : cliquez sur *Environments* pour choisir votre environnement (`base` ou bien par exemple notre `simul`). Ensuite revenez dans *Home* et cliquez sur *Launch* dans le cadre *Jupyter Notebook*, ou bien dans *Install* s'il n'est pas déjà installé dans cet environnement. Un terminal surgit qui va lui-même lancer une commande ouvrant un serveur Web dans votre navigateur à l'adresse `http://localhost:8888`. Les navigateurs conseillés à ce jour sont Chrome, Edge, Firefox et Safari.

Bref, dans les deux cas, vous vous retrouvez dans l'explorateur de fichiers de *Jupyter* qui affiche votre répertoire, dans lequel vous allez naviguer pour ouvrir un carnet `.ipynb` pour édition (il y a de nombreux exemples sur les sites web de [ROSS] et [VAN]). Attention, ce n'est pas un fichier textuel simple `xxx.py` en

11. Vous pouvez l'essayer sur `https://jupyter.org/try` avec Python, R, C++, Scheme, *etc.*

Python pur, mais un fichier structuré au format JSON que nous rencontrerons plus tard dans ce livre. Peu importe.

Nous allons créer un nouveau *carnet* avec le menu **Nouveau** situé en haut à droite, optez pour *Python 3 (ipykernel)*. Vous arrivez alors dans la fenêtre d'un carnet de nom *Untitled* en haut à gauche, sur lequel vous cliquez pour le changer en `essai` (l'extension `.ipynb` sera ajoutée automatiquement).



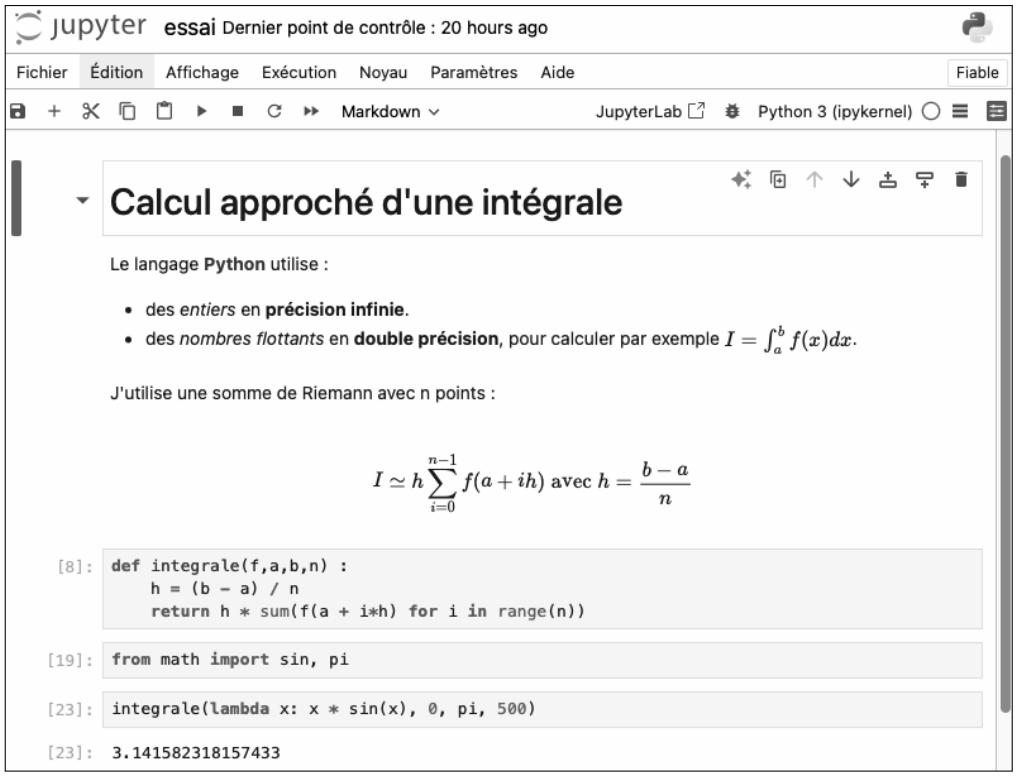
La première et unique cellule ci-dessus est en mode **Code** comme indiqué dans un menu déroulant : elle est censée contenir du code Python. Utilisons ce menu déroulant pour la mettre en mode **Markdown**, une sorte de HTML caché destiné à produire un **texte enrichi** avec balises et formules mathématiques en **TEX** (j'ai bien entendu installé le logiciel TEX sur ma machine, chez moi *TeXShop*). Le menu *Aide* contient une entrée pour apprendre *Markdown*[12]. Voici ce que j'entre dans les quatre premières cellules qui sont dans ce même mode ; j'utilise un '`------`' virtuel pour indiquer la séparation des cellules.

```
# Calcul approché d'une intégrale
-------
Le langage [**Python**](http://www.python.org) utilise :
-------
* des *entiers* en **précision infinie**.
* des *nombres flottants* en **double précision**, pour calculer
par exemple $I = \int_{a}^{b} f(x) dx$.
-------
J'utilise une somme de Riemann avec n points :
-------
$$I\simeq h\sum_{i=0}^{n-1}f(a+ih)\;{\rm avec}\;h=\frac{b-a}{n}$$
```

Les **cellules** qui suivent sont en mode **Code**, exécutables par le noyau (*kernel*) Python 3. Elles contiennent une définition de fonction, une importation et un

12. *Markdown* a été choisi pour contrebalancer le M de HTML qui signifie *Markup*. Pour les commandes les plus utiles, consultez `https://commonmark.org/help`.

calcul. Au final, un *notebook* exécutable a été produit, exportable en PDF, HTML, LaTeX, *etc.* : outil précieux pour étudiants, enseignants et ingénieurs. Les menus *Édition*, *Affichage*, *Exécution*, *etc.* permettent de manipuler le style et l'évaluation individuelle ou collective des cellules, ou procéder à divers réglages.

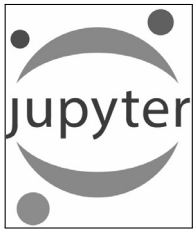


Quelques mots sur le codage en ***Markdown***. En ligne 62, la balise `#` signifie `<h1>` en HTML, elle indique un **titre** sur une seule ligne. En ligne 64, un **lien** en gras (balise `**`) est situé entre crochets, l'URL étant placée à la suite entre parenthèses, c'est l'analogue du `<a href="..."><b>...</b></a>` de HTML. Une ligne vide permet de changer de paragraphe en restant dans la même cellule, c'est pourquoi il faut valider une cellule entière par *Maj-Entrée*. Enfin, la balise solitaire `*` en début de ligne indique une puce de liste non ordonnée.

Les *carnets* de Jupyter sont sophistiqués. On peut y afficher des graphiques, intégrer des cellules audio ou vidéo, et les **diffuser** grâce au **Notebook Viewer**, consultez `https://nbviewer.org/faq`. L'ensemble reste quand même assez complexe, notamment au niveau de la sécurité dans le cadre d'un travail partagé ou dans le *cloud*. Sans laisser la forme prendre le pas sur le fond, qui est pour ce livre l'apprentissage de la programmation en Python, j'ai quand même

rédigé certaines solutions à la fois sous la forme de fichiers d'extension `.py` exécutables en principe par tous, et de fichiers `.ipynb` pour être ouverts dans Jupyter ou Visual Studio Code sous la forme de *notebooks*. À vous de faire votre choix, et d'en changer le moment venu.

Une introduction à Jupyter se trouve au chap. 18 du cours pour les biologistes décrit dans [FUC] et à un niveau plus avancé sur la page de Marc Buffat à l'Université Lyon-1 qui utilise `nbgrader` permettant à l'enseignant de concevoir des *carnets* Jupyter et préparer leur évaluation avec notation automatique.



Jupyter Lab

Le projet Jupyter est issu de IPython, avec la volonté de se détacher de Python et d'abstraire le concept de *notebook* qui est devenu indépendant du langage de programmation. Les pédagogues intéressés par le travail collaboratif avec Jupyter jetteront un œil sur le livre **Teaching and Learning with Jupyter** [JUP] qui explore des *tactiques* d'enseignement avec les carnets. Vous pourrez enfin suivre les nouveautés sur Jupyter dans les conférences annuelles des spécialistes disponibles sur la chaîne *JupyterCon* de YouTube.

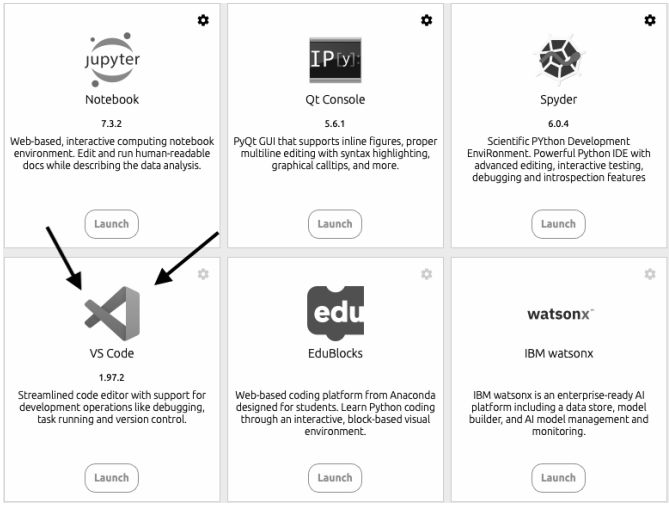
Le niveau suivant **Jupyter Lab** `https://jupyterlab.readthedocs.io` devient un véritable environnement interactif permettant d'utiliser conjointement des *carnets*, des éditeurs de texte, des Terminaux et des outils de visualisation pour ces fameuses *data* qui sont le point de mire de l'informatique d'aujourd'hui.

Afin de ne pas forcer le trait sur les IDE, je vous laisserai devenir acteur de cette aventure en explorant les ressources du Web, notamment celles des universités voire lycées qui utilisent Jupyter depuis plusieurs années. En cas de difficultés liées à l'environnement Jupyter, dirigez-vous vers `discourse.jupyter.org`.

1.3 L'éditeur professionnel : Visual Studio

Visual Studio (VS) est un IDE développé par Microsoft pour Windows ([VIS]) mais on trouve aussi sur `visualstudio.microsoft.com/fr/` une version allégée **Visual Studio Code** pour Windows, Mac et Linux. Il se trouve que **VS Code** se trouve dans les applications proposées par *Anaconda Navigator*. Je vous propose

un petit tour plus que minimal, il existe de nombreux tutoriels et vidéos de prise en main pour cet éditeur très complet. Lançons Navigator[13] et cherchons *VS Code* parmi les applications (installées ou à installer). Cliquons sur **Launch**.



Une fenêtre s'ouvre, je presse Maj-Cmd-P (sur Mac) puis j'écris *installer la commande* `code` *dans PATH* afin de pouvoir lancer VS Code à partir du Terminal.

Edition ou exécution d'un fichier Python `xxx.py` situé dans un dossier de programmes Python. Dans le menu *Fichier*, je clique sur *Ouvrir...* et je cherche mon fichier pour l'ouvrir. Il est alors chargé dans l'éditeur avec des lignes numérotées mais il faut **installer un interpréteur Python**, comme on le voit en bas à droite. Je peux l'exécuter dans l'environnement `(base)` par exemple. Je clique sur *Sélectionner un interpréteur* et j'opte pour *Python 3.12.9 ('base')*. Oui, chaque environnement a son propre Python, même le Python de la maison-mère *Python 3.13.0 64-bit* aurait convenu aussi. On peut lire maintenant en bas à droite *Python 3.12.9 ('base' : conda)*. J'exécute sans débogage avec le menu *Exécuter* ou Ctl-F5, et je vois les affichages dans le Terminal en bas.

> **Remarque** — L'éditeur connaît les cellules IPython, comme Jupyter. Si j'avais chargé `xxx.ipynb`, j'aurai eu droit à une présentation en cellules comme avec Jupyter, chacune pouvant être exécutée isolément avec son propre résultat juste en-dessous. Il existe un module `ipynb2py` pour obtenir un fichier Python à plat, non cellulaire, à partir d'un *notebook*.

Installation de Copilot. Microsoft (à-travers GitHub) livre une version de Copilot, une IA d'aide à la programmation. Dans menu *Affichage* de VS, cliquez sur *Extensions* (ou sur la 4ème icône verticale sur le bord gauche, ou Cmd-Maj-X).

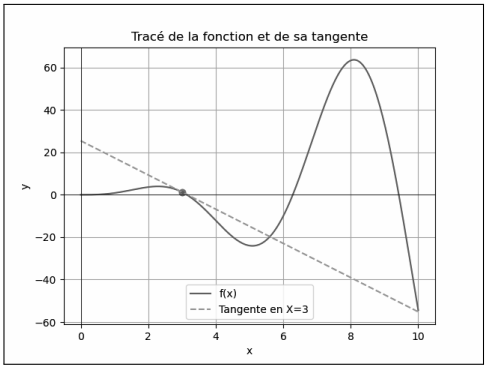
13. Si vous n'êtes pas sous Anaconda, téléchargez et lancez l'application *Visual Studio Code*.

Tapez `GitHub Copilot` puis cliquez sur *Installer*. Vous devrez avoir un compte GitHub, même si vous ne l'utilisez pas et il vous faudra peut-être accepter un accès sécurisé à ce compte GitHub, vous entrez dans un domaine professionnel.

Une fois installé, ouvrez un nouveau fichier. Au lieu d'entrer du code dans l'éditeur, tapez Cmd-I. Vous êtes alors invité à demander à Copilot de préparer votre travail. Exemple :

> **Demander à Copilot** : *Prépare-moi un code pour tracer une fonction f(x) sur un intervalle [a,b]. Tu traceras la tangente à la courbe de f au point d'abscisse X. Je définirai moi-même f,a,b,X. Le repère est orthonormé.*

Un programme Python utilisant `numpy` et `matplotlib` s'affiche alors dans l'éditeur, que je complète en choisissant la fonction $f(x) = x^2 \sin(x)$, $X = 3$ et $[a, b] = [0, 10]$. Je l'accepte puis j'exécute le fichier, et voilà. Il a oublié (heureusement) que le repère était orthonormé. C'est de la triche ? Oui, mais il faut accepter que nous sommes en 2025 et que l'IA monte, monte... Profitons-en mais restons vigilant. Parfois les codes sont un peu faux et ce peut être la galère.



Dans le menu *Affichage*, vous trouverez *Conversation* qui est une sorte de ChatGPT adapté à la programmation, ainsi que *Editions Copilot* pour lui demander de travailler sur votre code : le comprendre, l'analyser, le transformer.

1.4 Conclusion. Quel éditeur choisir ?

• Si vous êtes **débutant** (lycée, université, futur enseignant), je vous conseille de démarrer très simplement en installant la distribution CPython de la maison-mère (`www.python.org`) avec son éditeur IDLE et son toplevel `>>>`. Elle est solide, vous n'aurez sans doute pas besoin d'environnements et s'il vous faut installer des modules, vous le ferez avec `pip` (ou mieux : `pip3`).

• Si vous êtes en période **intermédiaire**, disons que vous avez déjà un peu touché à la programmation, vous savez ce qu'est un Terminal et sa ligne de commande,

ou vous n'avez pas peur de l'apprendre (juste ce qu'il faut pour travailler), je vous conseille d'installer la distribution Anaconda. Vous pouvez conserver la distribution CPython ci-dessus si vous acceptez de prendre des risques car il y aura plusieurs Python sur votre Machine, un vrai nœud de serpents, avec plusieurs `pip` par exemple – en l'occurence `pip3`. Il sera essentiel de bien connaître les emplacements de ces Python (voir § 1.2.2). Bref vous avez assez de mémoire libre et vous avez installé Anaconda ou Miniconda, bravo. Vous pouvez commencer avec l'éditeur IDLE d'Anaconda, avec la commande `idle3` si vous voulez rester quelque temps en pays connu, puis essayez Spyder avec la commande `spyder` dont une vidéo du projet *Mathscope* de l'APMEP est en français, obtenue en cherchant « *Mathscope Spyder* ». N'oubliez pas qu'il existe Anaconda Navigator pour éviter la maudite ligne commande chérie des *geeks*.

Quand vous commencez à avoir tout cela en main, passez à Jupyter qui est un peu déroutant au début mais vous ne le regretterez pas. Il vous faudra être bien encadré pédagogiquement, ou mettre le nez dans les pages de doc sur le Web mais c'est le lot des programmeurs, non ?

• Enfin, vous êtes bien **avancé** et vous aimez lire les docs, les environnements complexes ne vous effraient pas. Approfondissez Jupyter ou passez à Visual Studio Code, et bonne route ! Dans les deux cas, vous pourrez être accompagné d'une IA pour coder plus vite (voir [POR]).

En attendant, nous allons nous concentrer sur le langage Python et son apprentissage depuis le début. Comme le disait un homme politique, *ne vous inquiétez pas, tout va bien se passer !*

Dans ce livre, je vais oublier Jupyter et VS Code pour des raisons typographiques et non scientifiques. Quant à IDLE et Spyder, je vais couper la poire en deux. Avec Spyder, je déplore – en tant qu'auteur seulement – la place perdue avec les `'In␣[14]:␣'`, les `'Out[14]:␣'` et les lignes vides de séparation, qui gaspillent un espace précieux sur le papier [14]. Je conserve donc le *prompt* `>>>` de IDLE, et remplacerai `Out[i]` par le symbole ▶. En effet, il est parfois désagréable avec IDLE de ne pouvoir distinguer le simple **effet** de l'évaluation d'une expression et la sortie de sa **valeur** affichée par la console.

```
>>> print(2 + 1)
3                       # effet
>>> 2 + 1
▶ 3                     # résultat
```

```
In[1] : print(2 + 1)
3                       # effet
In[2] : 2 + 1
Out[2]: 3               # résultat
```

14. Bien qu'il soit possible de personnaliser les *prompts* avec IPython.

********* **Interrogez votre IA** **********

Née vers 1956, l'intelligence artificielle (IA) a pris une ampleur considérable ces dernières années, transformant peu à peu divers secteurs, de l'industrie aux services et à l'éducation. La dernière avancée majeure qui nous intéresse ici est celle de l'**IA générative** issue de la rencontre entre les *grammaires génératives* du linguiste Noam Chomsky dans les années 1950 avec les réseaux de neurones formels (p. 309) qui ont pris leur envol dans les années 1980. Vous avez sans doute entendu parler de cette IA générative, avec la popularité de **GPT** (*Generative Pre-trained Transformer*, 2018) qui subit actuellement une forte concurrence avec **Claude**, **DeepSeek**, **Gemini**, **Grok**, **Le Chat**, **Llama** et autres.

Je vous invite, si vous l'acceptez, d'adopter un de ces grands modèles de langage (LLM : *Large Language Model*) et de vous en servir pour de la simple documentation de fonctions Python ou comme assistant scientifique, en programmation pour ce qui nous concerne. Vous l'utiliserez en mode conversationnel (*chat*), au sein d'un logiciel externe ou bien intégré (comme **Copilot**) dans votre environnement de programmation.

Perdez vos illusions, une IA n'est pas (encore) vraiment intelligente, ce sera peut-être pour plus tard, avec le domaine spéculatif de l'**IA générale**. Pour l'instant, vous lui envoyez une demande sous la forme d'un texte (le *prompt*), et elle fera de son mieux, avec ses neurones et ses connaissances issues d'un long et coûteux apprentissage, pour aller dans votre sens, en complétant le texte envoyé. Chaque année, les progrès sont fulgurants mais elle fera parfois des erreurs, il en suffit d'une petite pour envoyer un programme dans les choux. Il vous faudra donc être attentif et vérifier soigneusement la réponse, dont vous êtes propriétaire et dont elle vous délègue en principe les droits d'auteur, rejetant au passage sur vous la responsabilité de son utilisation.

Si vous ne comprenez pas tout ce qu'elle dit, n'hésitez pas à le lui faire savoir. Lorsque vous décelez une erreur, elle se confondra en plates excuses et tâchera d'améliorer sa production, pour le meilleur ou pour le pire, et le pire c'est la tentation de coller une rustine dans chaque cas où cela ne fonctionne pas. Il peut être pertinent alors de lui faire tout oublier pour repartir de zéro en reformulant et précisant votre *prompt*. Ce dernier est rédigé en langage naturel propre, pas un style SMS lapidaire. Des phrases bien articulées, avec des données précises et un objectif clair. Votre demande peut contenir du code Python à analyser ou à mettre au point, avec les réserves précédentes. Les IA codent assez bien en général, leur code est documenté mais vous pouvez leur demander des justifications, même mathématiques ou physiques. Voici une première occasion d'utiliser votre IA.

PROMPT : *Pour un débutant, est-il trop tôt pour utiliser "Visual Studio Code" ou "Jupyter" ? Embarquent-ils un assistant IA pour le codage en Python ?*

2 Variables, nombres et fonctions

Vous avez sans doute installé Anaconda, et choisi un éditeur de texte : IDLE, Spyder , Jupyter ou autre. Dans tout le livre nous ferons abstraction de l'environnement de programmation, sauf mention explicite du contraire (cf. § 1.4).

2.1 Qu'est-ce qu'une variable ?

Un **identificateur** en Python est un mot dont les seuls caractères sont les lettres minuscules de `a` à `z`, les lettres majuscules de `A` à `Z`, le souligné `_` ainsi que, sauf pour le premier caractère, les chiffres de `0` à `9`. Le trait d'union `-` est interdit. Par exemple `x`, `L`, `somme`, `fiche45`, `parent_de`, `L` et `Cercle` sont des identificateurs. Bien que cela soit autorisé, il est déconseillé d'utiliser des lettres accentuées dans les identificateurs [1].

Une **variable** est un identificateur faisant référence à un emplacement dans la mémoire de l'ordinateur contenant un **objet**. On dit que cet objet est la **valeur** de la variable, ou encore que la variable est **liée** à cet objet. Les variables débutant par une majuscule sont réservées en principe au nom des classes (§ 7.1).

Ci-dessous, la variable `longueur` est liée par le signe `=` au résultat de l'opération $2 + 3$, c'est-à-dire à l'objet 5, tandis que la variable `calcul` est liée à la fonction mathématique $x \mapsto 2x - 1$. Entrons deux lignes de code Python dans l'éditeur (IDLE, Spyder, Jupyter) :

```
81 longueur = 2+3                          # variable à valeur numérique
82 calcul = lambda x: 2*x-1                # variable à valeur fonctionnelle
```

Puis demandons leur exécution (par un menu ou par la touche F5 sous IDLE et Spyder), pour effectuer des tests à la console.

```
83 >>> longueur                            # sa valeur est un nombre
84 ▶ 5
```

1. La norme Python 3 précise que l'on peut utiliser des caractères Unicode dans les identificateurs, au risque de réduire la circulation des sources de programmes...

```
>>> calcul                                  # sa valeur est une fonction
▶ <function <lambda> at 0x1051d0680>
>>> calcul(longueur)                        # un appel à la fonction calcul
▶ 9
```

Pour illustrer du code Python, nous utiliserons des cadres numérotés contenant des lignes de code rédigées dans un **éditeur de texte**. *Vous obtiendrez une feuille d'édition en demandant « Nouveau fichier… » dans le menu « Fichier ». Remplacez « Fichier » par « File » si votre éditeur est en anglais.*

Une fois ce texte exécuté, les ordres d'affichage (aucun ici) ont lieu à la **console** *(ou* **toplevel***), où l'on peut prolonger l'exécution par des calculs interactifs. Un caractère* `#` *signale un* **commentaire** *qui court jusqu'au bout de la ligne et ne sert qu'à des fins de documentation, il est ignoré par Python. Enfin, les numéros de ligne ne sont là que pour y faire référence dans ce livre.*

Le signe `=` ci-dessus n'est pas une comparaison d'égalité, mais plutôt une égalité forcée nommée **affectation**. Cet opérateur d'affectation `=` lie la variable `longueur` sur sa gauche à la valeur `5`, et la variable `calcul` à la fonction mathématique $x \mapsto 2x - 1$. Nous verrons au § 2.16 le mot-clé `def` permettant de construire des fonctions contenant plusieurs instructions. On dit que `longueur` et `calcul` sont deux **variables**, dont la valeur est numérique pour l'une et fonctionnelle pour l'autre. Il est possible à tout moment de modifier la valeur d'une variable par une nouvelle affectation. Ajoutons deux nouvelles instructions à la fin du texte en cours d'édition.

```
...
longueur = longueur + 1                   # longueur "devient égal à"...
calcul = lambda x,y: x + 2 * y            # calcul "devient égal à"...
```

```
>>> longueur + calcul(3, longueur)        # 6 + calcul(3, 6)
▶ 21                                      # ⇝ 6 + 15 ⇝ 21
```

Programmer en modifiant des variables conduit au style dit **impératif** en programmation. Éviter de modifier les variables – ce qui peut paraître paradoxal – conduit à un autre style, dit **fonctionnel**, que nous rencontrerons parfois.

2.2 Qu'est-ce qu'un type ?

Les valeurs 2 et -5 sont des nombres entiers, on dit que ces valeurs sont **typées** et que leur **type** se nomme `int`, qui dénote l'ensemble potentiel (on dit en Python

la *classe*) des entiers autorisés. Un entier est donc un objet de **type** `int`, ou de **classe** `int` ce qui revient au même (*type* et *classe* sont des synonymes). La valeur 3.05 sera en revanche un nombre approché, de **type** `float`. La fonction `type` retourne le type d'un objet Python, mais l'affichage d'un type peut dépendre du logiciel Python utilisé.

```
>>> type(-5)            # IDLE
<class 'int'>
>>> type(3.05)
<class 'float'>
>>> type(calcul)
<class 'function'>
```

```
In [1]: type(-5)      # Spyder
Out[1]: int
In [2]: type(3.05)
Out[2]: float
In [3]: type(calcul)
Out[3]: function
```

Ces différences cosmétiques dans l'affichage d'un type ne sont perceptibles qu'à la console. Celui d'IDLE est plus puriste, celui de IPython plus lisible. De toutes façons, on ne travaille que rarement à la console : on rédige dans l'éditeur le texte d'un programme comprenant des instructions `print` qui, elles, afficheront un type sous la forme `<class 'int'>`. Il faut comprendre que l'affichage du résultat d'un calcul à la console n'est pas un objet Python, mais seulement sa **représentation externe** pour un œil humain. En d'autres termes, si vous demandez l'évaluation du résultat d'un calcul, il se peut que ce résultat ne soit pas lisible par Python.

```
>>> <class 'function'>  # IDLE
SyntaxError: invalid syntax
```

```
In [1]: function      # Spyder
Error: 'function' not defined
```

Python reconnaît les mots `int`, `float`, `complex` et d'autres noms de types, mais pas le mot `function`. Pour tester le type d'une variable, on utilise l'**égalité** entre deux objets notée `==` en Python, à ne pas confondre avec l'**affectation** `=`.

```
>>> type(5) == int
▶ True
```

```
>>> type(5.0) == int
▶ False
```

Donc chaque objet Python a un type bien défini. Mais une variable a-t-elle un type ? Dans l'absolu non, mais *à un instant donné* nous pouvons convenir que le type d'une variable n'est autre que celui de sa valeur, qui est un objet Python. On dit qu'une **variable** Python est **dynamiquement typée**, rien n'impose qu'elle ait toujours le même type, elle a le droit de changer de type en cours d'exécution.

```
>>> longueur = 5
>>> type(longueur)
▶ int
```

```
>>> longueur = longueur + 0.5
>>> type(longueur)
▶ float
```

Ce n'est pas forcément conseillé car il est sain de pouvoir garantir dans un programme qu'une variable a toujours le même type, pour avoir un meilleur contrôle. Pour cela il suffirait plus haut d'initialiser `longueur` à 5.0 au lieu de 5. Le contrôle

du type des variables est un défi constant pour le programmeur. Mais rien ne l'y oblige, au contraire des langages *fortement typés* comme C ou Java dans lequel une variable de type `int` ne pourra jamais devenir de type `float`.

2.3 Le type `int` des nombres entiers

En abordant un langage de programmation, il est usuel de commencer son étude par les nombres et leurs opérations. Les **nombres entiers** sont des objets Python de type `int`, en **précision infinie**[2], et l'arithmétique y est **exacte**[3]. Les opérateurs sont `+` (addition), `-` (soustraction), `*` (multiplication), `//` (quotient entier), `%` (modulo : reste de la division) et `**` (puissance).

```
120 >>> 1234567890987654321**2          # un calcul de carré à la console
121 ▶ 1524157877457704723228166437789971041
122 >>> 13//5                           # quotient de la division de 13 par 5
123 ▶ 2
124 >>> 13%5                  # "13 modulo 5" : reste de la division de 13 par 5
125 ▶ 3
```

Vous pouvez si le besoin s'en fait sentir *importer* en plus les fonctions `gcd` (le PGCD) et `lcm` (le PPCM) à partir du module `math` (*cf.* § 2.6).

```
126 >>> from math import gcd           # le mot gcd devient disponible
127 >>> gcd(12,15)                    # Greatest Common Divisor
128 ▶ 3
```

Le test d'**égalité** de deux nombres se note `==` à ne pas confondre avec l'affectation `=`. L'inégalité $a \neq b$ de deux nombres entiers se note `a != b` ou avec l'opérateur de négation `not(a == b)`. On peut comparer des nombres (sauf les complexes) avec les opérateurs `<`, `<=`, `>` et `>=`. Le résultat d'une comparaison est l'une des deux constantes `True` (qui signifie *vrai*) et `False` (*faux*) qui sont des **valeurs booléennes** : de type `bool` (*cf.* § 2.13).

```
129 >>> (4**5) > (5**4)                 # est-ce que 4^5 > 5^4 ?
130 ▶ True
```

2.4 Les nombres flottants

Viennent ensuite les **nombres approchés** où l'arithmétique est **inexacte** à cause de la précision limitée. Par défaut ces nombres – dits *flottants* – sont en **double**

2. Le nombre de chiffres ne dépend que de la capacité mémoire de votre ordinateur.

3. Dans certains langages de programmation, le calcul de 30! pourrait donner un résultat faux, voire négatif !

précision sur 64 bits[4] : *grosso modo* vous disposez d'environ 15 chiffres après la virgule. Si vous approchez de zéro la situation se complique. Le plus petit nombre flottant strictement positif est $4.9406564584124654 \times 10^{-324}$ noté en Python `4.9406564584124654e-324`, et en-dessous de lui se trouve immédiatement `0`.

```
>>> x = 4.9406564584124654e-324
>>> x == 0
▶ False
>>> x / 2                                  # division dans les nombres flottants
▶ 0.0
```

Vous sentez au dernier exemple que **l'arithmétique des nombres flottants est dangereuse**. La division approchée se note `/`, elle pousse les calculs après la virgule (virgule qui est en fait un *point décimal* en programmation) ce que ne fait pas le quotient entier `//`.

```
>>> 5 / 3                                  # mais 5 // 3 == 1
▶ 1.6666666666666667                       # résultat approché, inexact !
>>> 3 * (5 / 3)
▶ 5.0                                      # et non 5 car 5/3 est approché
```

La fonction *valeur absolue* $x \mapsto |x|$ existe dans le noyau de Python, où elle se nomme `abs`. En ce qui concerne les opérations flottantes $\sqrt{x}$, $\cos(x)$, $\sin(x)$, e^x, $\ln(x)$ *etc*, il faudra demander explicitement leur *importation* depuis le module `math`. Les fonctions e^x et $\ln(x)$ se notent en Python `exp(x)` et `log(x)`.

```
>>> import math                            # cf. § 2.6
>>> dir(math)                              # les mots du module math
▶ ['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos', 'cosh',
'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan',
'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

```
>>> from math import pi, atan              # pi est le nom de π
>>> pi
▶ 3.141592653589793                        # une approximation de π
>>> atan(0.5) * 180 / pi                   # l'arc-tangente de 0.5 vaut
▶ 26.56505117707799                        # environ 26.6°
```

4. Avec 64 chiffres en binaire. L'arithmétique binaire sera abordée dans les exercices.

Remarque — Les nombres *réels* sont donc une illusion en programmation puisque l'on travaille avec des flottants dont le nombre de chiffres est limité : nous ne manipulons que des nombres rationnels approchés. Python ne connaît qu'une approximation de π, et $\sin(\pi)$ sera *presque* nul. Vérifiez sur `sin(pi)` et `cos(pi)`.

Les entiers sont de type `int` et les flottants de type `float`. Si `x` est un flottant, alors `int(x)` est le nombre entier obtenu en gommant la partie fractionnaire de `x`. Si `n` est un entier, `float(n)` retourne le même nombre mais approché.

```
>>> type(3)
▶ int
>>> type(3.14)
▶ float
```

```
>>> int(3.14)
▶ 3
>>> int(-3.14)
▶ -3
```

```
>>> float(3)
▶ 3.0
>>> float(3.14)
▶ 3.14
```

Remarque — Attention, `int` et `float` sont bien des noms de classes et pas de fonctions ! En écrivant `int(x)`, on fait appel au *constructeur* de la classe `int`.

2.5 Les nombres complexes

Enfin, Python propose les **nombres complexes** notés $2+3i$ en mathématiques, et `2+3j` en Python, le nombre $i = \sqrt{-1}$ se notant `1j` pour éviter toute confusion.

```
>>> z = 2+3j                          # construction du nombre complexe z
>>> z = complex(2,3)                  # une autre manière de le définir
>>> (z.real,z.imag)                   # parties réelle et imaginaire de z
▶ (2.0,3.0)                           # ce sont des nombres flottants !
>>> from cmath import phase           # l'argument d'un nombre complexe
>>> (abs(z),phase(z))                 # module et argument en radians
▶ (3.605551275463989, 0.982793723247329)
>>> z*z                               # multiplication complexe
▶ (-5+12j)                            # l'affichage est parenthésé
```

Les opérations complexes $\sqrt{z}, \sin(z), e^z, \ln(z)$ *etc.* sont disponibles dans le module `cmath` qu'il vous faudra importer. Ci-dessous, j'importe la fonction exponentielle sur $\mathbb{C}$ en la nommant `cexp` pour ne pas entrer en collision avec la fonction `exp` du module `math` qui est l'exponentielle sur les nombres flottants.

```
>>> from cmath import exp as cexp     # l'exponentielle complexe
>>> cexp(3+2j)                        # calcul de e^{3+2i}
▶ (-8.358532650935372+18.263727040666765j)
>>> cexp(1j*pi)                       # calcul de e^{iπ}
▶ (-1+1.2246467991473532e-16j)        # approximativement -1
```

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

Le **noyau** Python disponible au lancement du système contient les constructions syntaxiques et les classes de base permettant de programmer avec des nombres, des textes (chaînes de caractères Unicode) et des données composées : tuples, listes, ensembles, dictionnaires et fichiers. On peut aussi y construire des classes.

Un programme Python est constitué d'un ou plusieurs **modules** qui sont des fichiers écrits en Python, d'extension `.py` (ou `.ipynb` si ce sont des *carnets*), chaque module participant à une fonctionnalité du programme. Le langage Python propose de son côté ses propres modules spécialisés (pour les mathématiques, le graphisme, l'accès à Internet, l'audio, *etc.*). Le mot-clé `import` permet d'importer tout ou partie d'un module.

Un module contient des instructions mais aussi des définitions de fonctions et de classes d'objets. Il définit ainsi des noms de variables liées à des valeurs (nombres, fonctions, classes, *etc.*). Toutes ces *liaisons* variable/valeur forment l'**espace des noms** (en anglais *namespace*) du module. Au départ, dans la console Python, vous êtes dans un *espace de noms* ne contenant pas la fonction *racine carrée* qui se nomme `sqrt` en Python, et qui se trouve dans un module prédéfini (en anglais *built-in*) nommé `math`.

```
182 >>> sqrt                                  # elle n'est pas dans le noyau !
183 NameError: name 'sqrt' is not defined
184 >>> import math                           # j'importe le mot math
185 >>> math                                  # un module est un objet Python
186 ▶ <module math from '/Users/roy/...'>
187 >>> math.sqrt                             # le "nom qualifié" de sqrt
188 ▶ <built-in function sqrt>
189 >>> math.sqrt(2)                         # calcul de √2
190 ▶ 1.4142135623730951                     # une valeur approchée
191 >>> sqrt(2)                              # mais sqrt n'a pas été importée !
192 NameError: name 'sqrt' is not defined
```

La fonction `dir` en ligne 141 permettait d'obtenir la liste des noms contenus dans un module. Attention, la directive `import math` n'importe que le mot `math`, il faut utiliser ce point d'entrée dans le module pour obtenir la fonction *racine carrée* par son *nom qualifié* `math.sqrt`, car le mot `sqrt` tout seul n'a pas été importé ! Si l'on souhaite importer le seul mot `sqrt`, il faut demander `from math import sqrt` comme en ligne 151 mais alors c'est le mot `math` qui n'est pas importé. Dans le cas où le module contient beaucoup de fonctions à importer, la directive `from math import *` fait l'affaire, mais elle est dangereuse car le module peut contenir des fonctions ayant le même nom que les vôtres !

Si vous avez déjà programmé vous-même une fonction `sqrt` et souhaitez importer la vraie fonction `sqrt` du module `math`, nommez cette dernière `math.sqrt` ou importez-la sous un autre nom `rac2` par exemple en demandant :

```
from math import sqrt as rac2
```

Remarque — Deux modules `sys` et `imp` permettent de travailler – en Python – sur le système de modules. La variable `sys.modules` contient la liste des tous les modules importés (y compris les vôtres) depuis l'ouverture de Python. La variable `sys.path` contient la liste des chemins menant aux modules importables.

2.7 Donc pas de nombres rationnels en Python ?

Pas dans le noyau Python. Mais il est possible d'*importer* la classe `Fraction` contenue dans le module `fractions` pour construire des objets de type `Fraction`.

```
193 >>> from fractions import Fraction          # les fractions exactes
194 >>> r1 = Fraction(1,3)                     # ou bien Fraction('1/3')
195 >>> r2 = Fraction(10,12)                   # == Fraction(5,6)
196 >>> r3 = r1-r2
197 >>> r3
198 ▶ Fraction(-1,2)                          # 1/3 - 5/6 = -1/2
199 >>> print(r3)                             # print sait bien afficher !
200 ▶ -1/2
201 >>> (r3.numerator,r3.denominator)         # un couple
202 ▶ (-1,2)
```

La notation `r3.numerator` se lit l'*attribut* (ou *propriété*) `numerator` de l'objet `r3` de la classe `Fraction`. Il s'agit de la **notation pointée** adaptée à la programmation par objets (voir § 7.1). En Python, *le chapeau de papa* s'écrit `papa.chapeau`.

2.8 L'égalité est-elle fiable sur les nombres flottants ?

Pas vraiment si l'on considère les exemples suivants :

```
203 >>> 0.1+0.1+0.1+0.1 == 0.4
204 ▶ True
```

```
205 >>> 0.1+0.1+0.1 == 0.3
206 ▶ False
```

En fait, le nombre flottant 0.1 qui est simple en décimal, est compliqué en binaire car son écriture demande une infinité de chiffres après la virgule ! Mais le nombre étant approché, il y aura une troncature, donc une perte de précision et . . . boum ! Il est donc **fortement déconseillé d'utiliser l'opérateur == sur les nombres flottants**.

Mais alors, comment faire pour savoir si deux nombres flottants x et y sont *quasiment égaux* ? Le mieux consiste à se fixer une précision $h > 0$ assez petite (par exemple 0.001) et à demander si $|x - y| < h$. Un nombre x sera alors *quasiment nul* lorsque $|x| < h$.

Remarques — i) Le module `math` propose une fonction `isclose(a,b)` permettant de tester si `a` et `b` sont égaux à 10^{-9} près, la tolérance est réglable.
ii) Pour une grande précision dans les flottants, jetez un œil au module `decimal` de Python qui offre une arithmétique avec des flottants à précision fixe, pour laquelle `0.1+0.1+0.1` vaudra exactement `0.3` en écrivant `Decimal('0.1')` au lieu de `0.1`. Voir la page `https://pymotw.com/3/decimal/`.

2.9 L'infini existe-t-il en programmation ?

Il existe un nombre flottant bizarre construit par l'expression `float('inf')` qui répond peu ou prou aux règles de calcul avec $+\infty$. Sa valeur est affichée `inf`.

```
>>> inf = float('inf')                    # le mot inf n'est pas défini
>>> 0.5*inf - 1000000000000000000000000000000000000000000000000000
▶ inf
>>> -2*inf
▶ -inf
>>> 1/inf                                 # et 1/-inf ?
▶ 0.0
>>> 1/0
ZeroDivisionError: division by zero
```

Pour répondre aux impossibilités de calcul, un nombre flottant `float('nan')` est aussi disponible, le `'nan'` signifiant *Not A Number*. Sa valeur est affichée `nan`.

```
>>> inf - inf          # une "forme indéterminée" en maths, comme 0*inf
▶ nan
```

2.10 Qu'est-ce que la priorité d'un opérateur ?

L'utilisation des opérateurs arithmétiques est ambiguë : comment interpréter $3 + 2 \times 5$? Comme $(3 + 2) \times 5$ ou $3 + (2 \times 5)$? Simple convention prise au XVI^{e} siècle (*cf.* [SER]) où il a fallu passer de la phrase *Ajoute 3 au produit de 2 et de 5* au codage symbolique $3+2\times5$. Certains langages de programmation comme Scheme écrivent encore `(+ 3 (* 2 5))` avec l'opérateur en tête, sans ambiguïté. Mais en maths ou en Python, il faudra lever l'ambiguïté pour éviter que $3 + 2 \times 5$ signifie $(3 + 2) \times 5$. Il faut donc connaître les conventions de **priorité** des opérateurs.

`+` `-` `*` `/` `//` `%` `**`

Les opérations `+ -` ont une *même* priorité **basse** : ce sont les moins prioritaires, elles seront effectuées *après* les opérations `* / // %` qui sont de *même* priorité **plus haute**, et sont effectuées d'abord. Entre deux opérateurs de même priorité, le calcul se fait de gauche à droite.

Dans le doute, mettez certains calculs entre parenthèses. Par exemple, l'expression $E = 3 \times x + 2/y \times z$ s'écrira sans aucune ambiguïté $(3 \times x) + ((2/y) \times z)$. L'ordre des calculs serait donc :

$$E_1 = 3 \times x \quad \text{puis} \quad E_2 = 2/y \quad \text{puis} \quad E_3 = E_2 \times z \quad \text{puis} \quad E = E_1 + E_3$$

Enfin, le calcul d'une **puissance** a^b passe par l'opérateur `**`. Par exemple 2^{10} s'écrit `2 ** 10` en Python et vaut 1024 (le *kilo informatique*). L'opérateur `**` possède **la plus haute priorité** des sept opérateurs : les puissances sont calculées en premier ! Par exemple, `3*2+10*3**2/5-1` se comprendra :

```
((3 * 2) + ((10 * (3 ** 2)) / 5)) - 1
```

L'opérateur `**` est *associatif à droite* : `a**b**c == a**(b**c)`.

```
218 >>> 3*2+10*3**2/5-1 == ((3*2)+((10*(3**2))/5))-1
219 ▶ True
```

Les opérateurs arithmétiques de comparaison `<`, `<=`, `>`, `>=` ainsi que `==` et `!=` ont une priorité inférieure aux opérateurs de calcul `+`, `*`, *etc.* L'opérateur `==` ci-dessus est donc effectué en dernier, après calcul de ses membres gauche et droit.

2.11 Comment construire couples, triplets, *etc.* ?

Un couple, un triplet ou plus généralement un *n-uplet* d'expressions se note comme en mathématiques `(`α`,` β`)`, `(`α`,` β`,` γ`)`, *etc.* On parle en Python d'un **tuple**. Le tuple vide se note `()` mais le tuple à un seul élément α se note `(`α`,)` pour ne pas le confondre avec `(`α`)` qui est lu comme α.

Si l'on numérote les composantes d'un tuple à partir de 0, alors l'élément ou composante numéro `k` d'un tuple `t` se note `t[k]`, son accès est immédiat même si `k` est grand. Mathématiquement, on peut voir le tuple `t` de longueur n comme une suite de variables indexées $t_0, t_1, \ldots, t_{n-1}$, à ceci près que ces variables n'acceptent pas d'être ré-affectées.

```
220 >>> p = (1, 12, 7)                    # un point de l'espace 3D
221 >>> len(p)                            # le nombre de composantes
222 ▶ 3
223 >>> p[0] + p[1] + p[2]                # 1 + 12 + 7
224 ▶ 20
225 >>> (x, y, z) = p                     # déstructuration d'un tuple
```

```
226 >>> x+y+z
227 ▶ 20
```

En ligne 225, nous utilisons une **affectation déstructurante** basée sur le fait que l'on connaît la structure de `p` : il s'agit d'un tuple de longueur 3. S'il avait été de longueur 4, cette ligne aurait provoqué une erreur. Cela évite souvent les notations `p[k]` plus lourdes. Pour obtenir le nombre d'éléments d'un tuple, on utilise la fonction `len`, abréviation de l'anglais *length*.

```
228 >>> p[1] = 0                                    # affectation interdite !
229 TypeError: 'tuple' object does not support item assignment
```

La dernière ligne montre que les tuples sont des **objets non mutables** : on peut ré-affecter la variable `p` avec `p = (3,4)`, mais pas individuellement une composante `p[k]`. Nous verrons plus loin les *listes*, analogues aux tuples mais cette fois mutables. Si la mutabilité n'intervient pas, les tuples sont plus efficaces que les listes. Mais les tuples à un seul élément comme `(12,)` sont moins lisibles.

2.12 Quelle différence entre expression et instruction ?

Une **expression** est une combinaison de valeurs, de variables, d'opérateurs et d'appels de fonctions. Par exemple :

`2*x + sqrt(x-1)` et `print(2*x + sqrt(x-1))`

sont deux expressions que l'on peut évaluer, à condition bien entendu que les variables `x` et `sqrt` soient définies dans l'espace de noms courant.

```
230 >>> x = 5                                      # x est définie
231 >>> from math import sqrt                      # sqrt est définie
232 >>> 2*x+sqrt(x-1)                             # une expression,
233 ▶ 12.0                                        # et sa valeur
234 >>> print(2*x+sqrt(x-1))
235 12.0                                          # un effet
```

L'expression en ligne 232 est soumise à évaluation, produisant comme résultat la valeur `12.0` qui est automatiquement affichée par la console (valeur précédée ici d'un ▶). En ligne 234, la même expression est à nouveau évaluée et son résultat `12.0` est passé à la fonction `print` qui le transforme en texte puis affiche ce texte, sans retourner de résultat (absence de ▶). Ne confondons pas l'affichage de `12.0` en ligne 235 qui est un **effet** de `print` (*cf.* § 2.19), avec son résultat qui n'existe pas (on dit plutôt en Python que ce résultat est l'objet vide noté `None` qui ne s'affiche pas) ! Pour prouver qu'il ne s'agit pas d'une vue de l'esprit, vite un test.

```
>>> print(2*x+sqrt(x-1)) == None          # print renvoie None
12.0                                      # l'effet de print
▶ True                                    # et le résultat de ==
```

Une **instruction** est une expression en général sans résultat, **seul son effet importe**. L'instruction la plus simple est l'instruction vide `pass` qui ne fait rien ! Outre `print`, les plus célèbres sont sans doute l'instruction d'affectation `x = 2+3` qui nomme le résultat d'un calcul, et la définition d'une fonction avec `def` qui s'étend sur plusieurs lignes (*cf.* § 2.16). L'existence de la valeur `None` est importante, une fonction avec résultat pouvant la renvoyer pour indiquer qu'elle n'a pas réussi à calculer son résultat.

Une *expression* est **évaluée**, une *instruction* est **exécutée**.

Un **bloc** d'instructions est une suite d'instructions disposées à la verticale, correspondant à une même **indentation** (distance à la marge). La distance à la marge [5] doit être un multiple de la largeur de la tabulation de votre système (le plus souvent une tabulation correspond à 4 espaces). Dans un tel bloc, chaque instruction n'est exécutée que lorsque l'instruction précédente a terminé. Donc **exécution séquentielle** (les unes après les autres) de haut en bas !

Lorsque les instructions sont courtes, il est équivalent de les écrire à l'horizontale, séparées par un point-virgule. Mais les programmeurs Python n'apprécient que très modérément les points-virgules à l'horizontale, ils ont une nette préférence pour la station verticale ...

```
x = 1
y = x+1          ⟺      x = 1 ; y = x+1 ; z = (x,y)
z = (x,y)
```

Remarques — i) La **PEP 8** de Python (*Python Enhanced Proposal n° 8 : Style Guide for Python Code*) préfère les espaces aux tabulations. Elle a *horreur* du mélange entre espaces et tabulations, et des lignes ayant plus de 79 caractères.
ii) Une affectation `x = v` n'a pas de résultat mais un effet. Néanmoins, il existe une *expression nommée* `(x := v)` réalisant l'affectation mais renvoyant la valeur de `v` en bonus. Les parenthèses sont obligatoires dans ce que les anglo-saxons nomment un *walrus* (morse marin). Le lecteur avancé appréciera... ou pas.

2.13 Les expressions booléennes

Une **expression booléenne** est une expression Python dont la valeur est `True` ou `False`, par exemple `x > 0`. Mais dans un test, Python considérera comme

5. Pour indenter une ligne, n'utilisez jamais d'espaces, uniquement des tabulations !

`fausse` toute valeur égale à `False`, `None`, le zéro numérique, ou une collection vide (chaîne, tuple, liste, *etc.*). Toute valeur non réputée *fausse* est *vraie*. Donc 3 n'est pas un booléen mais sera considéré comme vrai dans un test. Ces conventions sont critiquables mais pratiques et utilisées, il faut donc vivre avec.

```
239 >>> if 1789: print('À la Bastille !')          # 1789 est « vrai »
240 À la Bastille !
```

Les opérateurs logiques `and`, `or` et `not` permettent de construire des expressions booléennes composées comme `(x > 0) and (x - y <= 2)`. Les parenthèses ne sont pas strictement obligatoires mais nous conseillons de les mettre pour éviter les problèmes liés aux priorités de tous ces opérateurs (voir § 2.14).

À partir de deux expressions booléennes α et β, on peut produire d'autres expressions booléennes composées à l'aide de ces mots-clés `and`, `or` et `not` :

- α `and` β est la **conjonction** de α et β. Elle vaut α si α est fausse, et vaut β sinon.
- α `or` β est la **disjonction** de α et β. Elle vaut α si α est vraie, et vaut β sinon.
- `not` α est la **négation** de α. Elle vaut `True` si α est fausse, et `False` sinon.

Ces définitions de `and` et `or` sont opérationnelles et non purement logiques. Elles expriment en langage courant que α `and` β n'est vraie que si les deux sont vraies, et que α `or` β n'est fausse que si les deux sont fausses. Mais elles précisent en plus que ces opérateurs `and` et `or` sont *court-circuités* et évaluent α avant β. Par exemple `False and` β est fausse et `True or` β est vraie, sans avoir besoin de calculer β. Il est donc important d'être bien conscient que les opérateurs `and` et `or` ne sont **pas commutatifs** et ne sont **pas des fonctions** ! Dans l'exemple suivant, la division `1/0` devrait provoquer en ligne 243 une erreur à l'exécution.

```
241 >>> 5 > 1 / 0                                # erreur !
242 ZeroDivisionError: division by zero
243 >>> (8 < 2) and (5 > 1 / 0)                  # erreur ?
244 ▶ False                                      # non : court-circuit !
```

Remarques — i) La définition précise donnée plus haut de `and` et `or` ne dit rien sur le type du résultat de α `and` β, qui peut être `bool` (`True` ou `False`) mais aussi du type de β dans le cas où α est « vraie ». Par exemple `1 and 2` vaut `2`.
ii) Idem pour α `or` β si α est faux.
iii) Dans une opération *arithmétique*, la valeur `True` (resp. `False`) est autorisée et prise pour `1` (resp. `0`). Par exemple `True + 2` vaut 3. Nous déconseillons d'utiliser ces conventions en-dehors de la logique numérique.

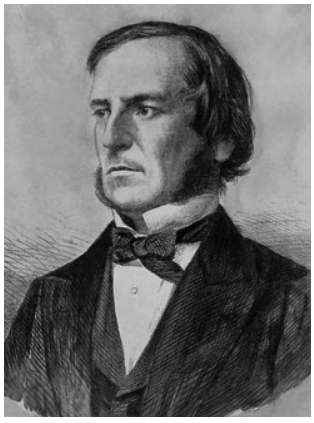
iv) En cas de doute, ou pour rendre vraiment booléenne une expression Python, il suffit de la passer au constructeur `bool` (le nom de la classe des booléens) qui forcera `True` ou `False` : `bool(3)` vaudra `True` et `bool([])` vaudra `False`.

Il faut savoir que, pour être compatible avec la logique mathématique, l'opérateur `or` est **distributif** par rapport à `and`.

$$\boxed{\alpha \texttt{ or } (\beta \texttt{ and } \gamma)} \iff \boxed{(\alpha \texttt{ or } \beta) \texttt{ and } (\alpha \texttt{ or } \gamma)}$$

et cette propriété est vraie si l'on inverse les opérateurs :

$$\boxed{\alpha \texttt{ and } (\beta \texttt{ or } \gamma)} \iff \boxed{(\alpha \texttt{ and } \beta) \texttt{ or } (\alpha \texttt{ and } \gamma)}$$



George Boole (1815-1864)
Photo du musée Powerhouse
de Sydney (source Wikicommons)

2.14 Les instructions conditionnelles

```
if x >= 0:
    r = math.sqrt(x)
    print('r vaut',r)
else:
    print('x est négatif !')
```

L'important mot-clé `if` permet de prendre une décision suivant l'état des données. Dans sa forme la plus simple, l'**instruction conditionnelle** `if:...else:...` prend la forme de gauche dans le tableau suivant, pour laquelle, si le `<test>` vaut *vrai*, le bloc indenté `<instruction> ...` est exécuté. La variante du milieu avec le mot-clé `else` situé à l'exacte verticale de `if` permet d'envisager un autre bloc d'instructions qui sera exécuté lorsque le `<test>` produit un résultat faux.

Enfin, s'il y a plus d'une alternative à considérer, le mot `elif` (contraction de `else if`) permet de les envisager de haut en bas avec un `else` (sinon) optionnel

à la fin. Il s'agit dans les trois cas d'une *instruction*, donc sans résultat, avec seulement un effet. Cette instruction `if` se développe sur plusieurs lignes.

`if <test>:`
`  <instruction>`
`  ...`
	`if <test>:`
	`  <instruction>`
	`  ...`
	`else:`
	`  <instruction>`
	`  ...`
		`if <test>:`
		`  <instruction>`
		`  ...`
		`elif <test>:`
		`  <instruction>`
		`  ...`
		`...`
		`else:  # optionnel`
		`  <instruction>`
		`  ...`

Remarques — i) Lorsqu'un bloc d'instructions est réduit à une instruction simple unique, on peut l'écrire immédiatement à côté des `':'`.
ii) La forme `if p(x) == True` peut se simplifier en `if p(x)`.
iii) La forme `if p(x) == False` s'écrira `if not p(x)` (rappel : `0`, `[]` et `None` sont « fausses »).

```
In [4]: x = -5                                  # au toplevel de Spyder
In [5]: if x >= 0:
   ...:     r = math.sqrt(x)
   ...:     print('r vaut',r)
   ...: else:
   ...:     print('x est négatif !')           # Maj-Entrée...
   ...:                                        # ... ou ligne vide
x est négatif !
```

À côté de cette **instruction** `if`, il existe une **expression** *if-else* dont l'évaluation produit cette fois un résultat. Sa syntaxe est `<expr> if <test> else <expr>` où `<test>` est une expression booléenne, et les `<expr>` sont des expressions évaluables, avec un `else` obligatoire. Rappelons que `if` n'est pas une fonction, mais un **mot-clé** (en anglais *keyword*), à l'instar de `lambda`, `def`, `and`, `while`, `for` *etc.*

```
>>> y = -1 if 2025 < 0 else 1          # y vaut -1 si 2025 < 0, sinon 1
>>> y
▶ 1
```

Résumons les **priorités des opérateurs** vus jusqu'à présent, de la priorité la plus forte à la plus basse (cette dernière est effectuée en dernier).

`**` | `* / // %` | `+ -` | `< <= > >= == !=` | `=` | `not` | `and` | `or` | `if-else`

priorité haute — *priorité basse*