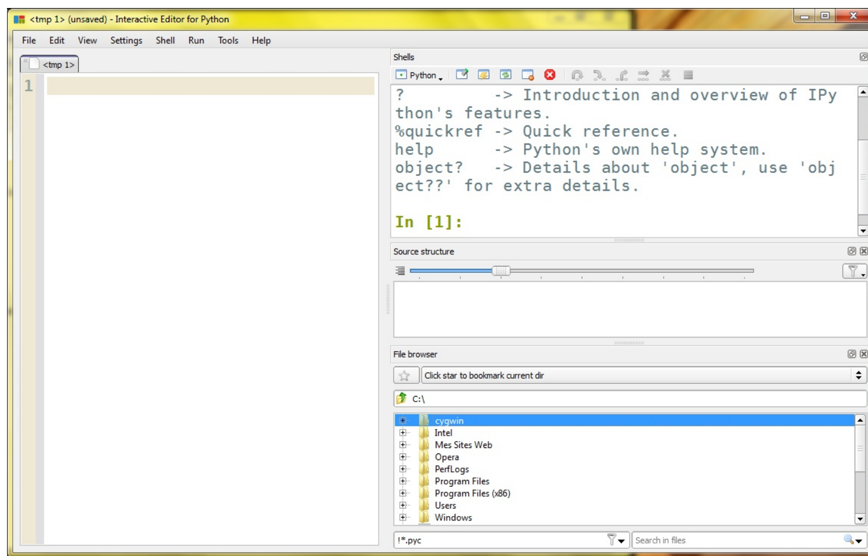
L'interpréteur interactif Python

L'informatique est, aujourd'hui, un outil précieux permettant d'automatiser des tâches ou bien de réaliser calculs et simulations. En guise d'introduction à ce domaine, et pour pouvoir faciliter l'étude de phénomènes scientifiques que vous rencontrerez dans les prochaines années, nous nous intéresserons au langage de programmation Python [1].

Pour dialoguer avec un interpréteur [2] Python, nous utiliserons l'interface de développement IEP (pour « Interactive Editor for Python »), fournie notamment avec la distribution *Pyzo*, qui est disponible pour la plupart des systèmes d'exploitation [3].

On la trouvera à l'adresse suivante : `http://www.pyzo.org`

Lorsque l'on ouvre l'interface de développement IEP, de nombreuses fenêtres sont présentes, comme on peut le voir sur la capture d'écran ci-après :



Dans un premier temps, on s'intéressera surtout à la « *console* » (également désignée sous le terme anglais « *shell* »), sise ici en haut à droite, laquelle permet de dialoguer interactivement avec un interpréteur Python.

La fenêtre de gauche est un éditeur, dont on parlera un peu plus tard. Les autres fenêtres, destinées à faciliter la vie au programmeur, peuvent être fermées pour le moment (en cliquant sur la croix située en haut à gauche). Les différentes fenêtres peuvent être réorganisées à la souris en fonction des besoins et préférences de l'utilisateur.

1. Ou plus précisément la version « Python 3k » du langage.
2. Le programme qui traduira les commandes que l'on écrira en actions réalisées par l'ordinateur.
3. C'est aussi la distribution présentement utilisée par le concours Centrale-Supélec.

1 Calculs avec l'interpréteur

La console interactive se comporte essentiellement comme une calculatrice un peu évoluée. Une invite (symbolisée par « `In` » avec l'interpréteur que l'on utilisera [4]) permet d'entrer une commande ou une expression (dans le cadre d'un programme, on parlera d'« *instruction* »). Une fois validée avec la touche *entrée*, la commande ou l'expression est évaluée par l'interpréteur, lequel fournit un résultat (précédé de l'indication « `Out` »).

Pour les nombres réels [5], le marqueur décimal utilisé sera le point « `.` » et, pour les nombres utilisant la notation scientifique, l'exposant (obtenu avec la touche EE ou $\times 10^x$ d'une calculatrice) est symbolisé par la lettre « `e` ». Ainsi, -2.5×10^{23} se note « `-2.5e23` ».

Les opérateurs arithmétiques habituels sont bien entendu disponibles, « `+` » pour l'addition, « `-` » pour la soustraction, « `*` » pour la multiplication et « `/` » pour la division. L'exponentiation (x^y) est notée « `**` ».

Ainsi, les expressions $2.5-1.7$, $42/12$ et 3^4 évalués dans l'interface donnent :

```
In [1]: 2.5-1.7
Out[1]: 0.8

In [2]: 42/12
Out[2]: 3.5

In [3]: 3**4
Out[3]: 81
```

Lorsqu'il y a plusieurs opérations identiques dans une même expression, l'évaluation se fait généralement de gauche à droite [6] :

```
In [4]: 8-4-2
Out[4]: 2

In [5]: 24/6/2
Out[5]: 2.0
```

4. Plusieurs interpréteurs Python sont en fait disponibles, et les invites de commandes peuvent être légèrement différentes, par exemple représentées par des chevrons >>>.
5. Précisions que pour ce qui est des entiers, bien qu'en mathématiques 17 et 017 représentent la même valeur, les 0 initiaux n'ayant pas de signification, un entier en Python ne doit pas commencer par un 0, car il ne serait pas reconnu par l'interpréteur.
6. On pourra remarquer, au passage, sur l'exemple précédent, que le langage semble faire une différence entre « `2` » et « `2.0` ». Cette différence est significative, le premier résultat est un entier, le second un nombre « flottant », ce qui se rapproche le plus des réels en informatique. Nous consacrerons un peu plus tard un chapitre aux représentations des nombres en informatique et aux limitations qu'elles impliquent.

Il existe toutefois une exception à cette règle, l'exponentiation qui, pour coller aux usages en mathématiques, est évaluée de droite à gauche :

```
In [6]: 2**1**2
Out[6]: 2
```

Si différents opérateurs sont utilisées dans une même expression, les règles usuelles de priorité s'appliquent. `**` est l'opérateur avec la plus grande priorité. Viennent ensuite, avec la même priorité, `*` et `/`, et enfin `+` et `-`.

```
In [7]: 3+4*5
Out[7]: 23

In [8]: 5*2**2/2
Out[8]: 10.0
```

Ainsi, dans le premier exemple, la multiplication `4*5` est effectuée avant l'addition. Dans le second exemple, on commence par calculer `2**2`, puis on multiplie le résultat par 5, et on le divise par deux.

Si l'on souhaite imposer un ordre d'évaluation différent, il convient d'utiliser des parenthèses dans l'expression :

```
In [9]: (3+4)*5
Out[9]: 35

In [10]: (5*2)**2/2
Out[10]: 50.0
```

Une autre opération très utile est également disponible, la *division entière*. Considérons deux nombres a et b, avec $b > 0$. La division entière consiste à trouver l'entier q et le réel r (respectivement appelés *quotient* et *reste*) vérifiant[7] $a = q \times b + r$ avec $0 \leqslant r < b$. On peut aisément vérifier que q et r existent et sont uniques.

l'opérateur « `//` » permet d'obtenir q, le quotient de la division entière de a par b :

```
In [11]: 7.5 // 2
Out[11]: 3.0
```

L'opérateur[8] « `%` » le reste r de cette même division entière.

```
In [12]: 7.5 % 2
Out[12]: 1.5
```

7. Attention, si ces opérations existent dans la plupart des langages, les règles lorsque a (ou b) est négatif peuvent différer.

8. Parfois appelé par commodité « modulo », dans ce qui est quelque peu un abus de langage.

On a bien $7.5 = 3 \times 2 + 1.5$ avec $0 \leqslant 1.5 < 2$.

Dans le cas où b est négatif, le quotient q et le reste r vérifient la même relation $a = q \times b + r$, mais cette fois avec $b < r \leqslant 0$. Comme pour une division classique, b ne doit pas être nul (on n'aurait plus une unique solution pour q). Lorsque l'on effectue une opération mathématique interdite, l'interpréteur retourne une erreur :

```
In [13]: 7.5 // 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-13-ae0fe8dd1b50> in <module>()
----> 1 7.5 // 0

ZeroDivisionError: float divmod()
```

On apprendra progressivement à décrypter ces messages que l'interpréteur renvoie lorsque quelque chose se passe mal. La dernière ligne est souvent la plus instructive. Ici, elle nous indique que l'on a tenté d'effectuer une division par zéro, une opération interdite.

Enfin, il est possible d'effectuer des calculs sur des nombres complexes. Le i des complexes (la racine de -1 dont la partie imaginaire est positive) est noté « `j` ».

Attention toutefois, ce `j` doit impérativement être précédé d'une valeur numérique, sans signe `*` entre les deux. Aussi écrira-t-on « `1j` » pour désigner le complexe i :

```
In [14]: (1+2j)**2
Out[14]: (-3+4j)

In [15]: (1+1j)*(1-1j)
Out[15]: (2+0j)
```

2 Fonctions mathématiques

On dispose par ailleurs de quelques fonctions mathématiques courantes pour les calculs. La première est `abs` qui permet d'obtenir la valeur absolue, ou bien la norme dans le cas d'un nombre complexe. Pour utiliser une fonction, on inscrit son nom, suivi, entre parenthèses, de son argument[9].

```
In [16]: abs(-10.23)
Out[16]: 10.23

In [17]: abs(3+4j)
Out[17]: 5.0
```

9. S'il y a plusieurs arguments, on les séparera par des virgules

On peut aussi utiliser la fonction `int` pour se débarasser de la partie décimale d'un nombre. Elle renvoie le nombre entier correspondant à une troncature à zéro décimales. C'est en fait une fonction un peu particulière sur laquelle nous reviendrons ultérieurement, mais qui peut d'ores-et-déjà se révéler utile.

```
In [18]: int(4.2)
Out[18]: 4

In [19]: int(-1.7)
Out[19]: -1
```

Ce sont, à peu de choses près, les seules fonctions qui peuvent être directement utilisées.

Fort heureusement, des milliers d'autres fonctions sont disponibles, rangées dans des « modules » et disponibles à la demande, à condition de les « importer » préalablement. Par défaut, l'interpréteur Python ne les connaît pas :

```
In [20]: sin(1.0)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-dacc23301901> in <module>()
----> 1 sin(1.0)

NameError: name 'sin' is not defined
```

Le message d'erreur nous indique ici que l'interpréteur Python ne sait pas ce que signifie « `sin` ».

Les modules peuvent, chacun, contenir un grand nombre de fonctions et plus encore. Celui qui nous intéressera le plus pour le moment est le module `math` qui contient la plupart des fonctions mathématiques usuelles [10].

Pour importer la fonction mathématique *sinus*, appelée `sin` en Python, depuis le module `math`, on utilise la commande « `from nom_de_module import nom_de_fonction` » :

```
In [21]: from math import sin
```

L'interpréteur ne répond rien si tout se passe bien, c'est-à-dire s'il a pu trouver la fonction requise dans le module spécifié, mais ladite fonction est alors disponible pour les calculs ultérieurs :

```
In [22]: sin(1.0)
Out[22]: 0.8414709848078965
```

Le module `math` contient, entre autres, la liste non-exhaustive de fonctions suivante :

- la fonction `sqrt` qui correspond à $\sqrt{x}$;
- la fonction `exp` qui correspond à e^x ;
- la fonction `log` qui correspond au logarithme népérien $\ln(x)$ [11] ;
- les fonctions trigonométriques `sin`, `cos` et `tan` ;
- leurs inverses `asin` (pour $\arcsin(x)$), `acos` et `atan` ;
- les fonctions trigonométriques hyperboliques `sinh`, `cosh` et `tanh` ;
- leurs inverses `asinh`, `acosh` et `atanh` ;
- les fonctions `floor` correspondant à la fonction *partie entière* $\lfloor x \rfloor$ et `ceil` pour $\lceil x \rceil$;
- les fonctions `hypot` et `atan2` qui, lorsqu'on leur fournit les composantes d'un vecteur du plan [12], retournent respectivement sa norme et l'angle qu'il forme avec l'axe des abscisses.

On pourra trouver l'intégralité des fonctions contenues dans le module `math`, de même que des détails sur leur fonctionnement, dans la documentation en ligne du module :

`https://docs.python.org/3.2/library/math.html`

Profitons cependant de l'occasion pour signaler un mécanisme bien pratique permettant d'obtenir des informations sur une fonction via l'interpréteur :

```
In [23]: help(sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

Il est possible d'importer plusieurs fonctions d'un module d'un seul coup, en les séparant par des virgules :

```
In [24]: from math import cos, exp

In [25]: exp(cos(1.0))
Out[25]: 1.7165256995489035
```

On notera au passage que la composition de fonctions est possible, avec une syntaxe tout à fait similaire à celle des mathématiques.

Vous verrez sans doute parfois la commande « `from math import *` » dans divers ou-

10. On peut trouver ces mêmes fonctions dans un autre module appelé `numpy` dont nous nous servirons plus tard, même s'il existe de subtiles différences entre les fonctions des deux modules. Les fonctions du module `math` sont par ailleurs destinées aux réels, on trouve des fonctions similaires agissants sur les complexes dans le module `cmath`.

11. Attention, en mathématiques, « $\log(x)$ » désigne généralement le logarithme décimal $\log(x) = \ln(x)/\ln(10)$, fréquemment utilisé en sciences de l'ingénieur et en sciences physiques, lequel correspond à la fonction Python `log10` du même module `math`.

12. Attention, pour la fonction `atan2`, c'est la composante verticale qui est fournie comme premier argument, pour des raisons historiques.

vrages, programmes ou sur certains sites. Cette commande importe, d'un seul coup, la *totalité* des fonctions d'un module (ici, le module math). Généralement, mieux vaut éviter cette solution de facilité : parfois, on importe de la sorte, sans le savoir, une fonction portant le même nom qu'une autre que l'on souhaite utiliser, causant des conflits aux conséquences potentiellement très fâcheuses.

Si l'on a besoin d'un grand nombre de fonctions d'un même module, on préférera plutôt importer le module lui même :

```
In [26]: import math
```

Une fois un module importé, on peut utiliser toutes les fonctions du module sans les charger une à une, sous réserve de faire précéder au nom de la fonction le nom du module, séparé par un point :

```
In [27]: math.floor( math.sqrt(10) )
Out[27]: 3
```

C'est en fait souvent la façon la plus claire d'utiliser les fonctions d'un module.

Une fois un module chargé de cette manière, on peut notamment obtenir des informations sur ce module (notamment la liste des fonctions mises à disposition) en utilisant la commande « help(math) ».

Cependant, cette solution conduit à utiliser des noms de fonctions un peu longs, de par la présence du nom du module. Pour abréger l'écriture des commandes, surtout lorsque le nom du module est long, on peut attribuer un « surnom » au module, grâce au mot-clé « **as** » :

```
In [28]: import math as m
```

Dès lors, m est un raccourci désignant le module math, et l'on peut écrire

```
In [29]: m.floor( m.sqrt(10) )
Out[29]: 3
```

On peut en fait utiliser ce même mécanisme de « surnom » pour importer une fonction d'un module en lui donnant un autre nom (par exemple pour éviter un conflit avec une fonction déjà existante) :

```
In [30]: from math import sin as sinus

In [31]: sinus(1.0)
Out[31]: 0.8414709848078965
```

3 Noms et affectations

3.1 Principe de l'affectation

Il est rapidement indispensable, lorsque l'on fait autre chose que des calculs élémentaires, de pouvoir désigner les valeurs importantes ou les résultats intermédiaires par un nom. Un *nom* en Python est possiblement constitué de lettres (majuscules ou minuscules), de chiffres et du symbole « _ » et ne peut pas débuter par un chiffre [13].

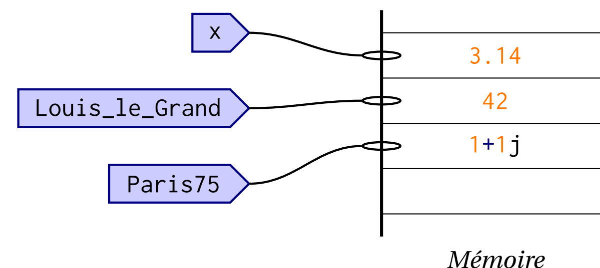
Pour associer un nom et une valeur, on utilise le symbole « = » en plaçant le nom à gauche du signe. Cette opération est appelée *affectation*.

```
In [32]: x = 3.14

In [33]: Louis_le_Grand = 42

In [34]: Paris75 = 1+1j
```

Lors d'une affectation, Python range en mémoire ce qui se trouve à droite du signe égal, et associe à l'emplacement en mémoire le nom placé à gauche de ce même signe égal [14]. On peut se représenter la mémoire de l'ordinateur comme un grand entrepôt : les valeurs (à droite du signe égal) sont « rangées » dans une case de la mémoire, et on accroche une étiquette, portant le nom situé à gauche du signe égal, à la case en question.



Mémoire

Dans l'interface interactive, entrer le nom seul permet de rappeler la valeur associée :

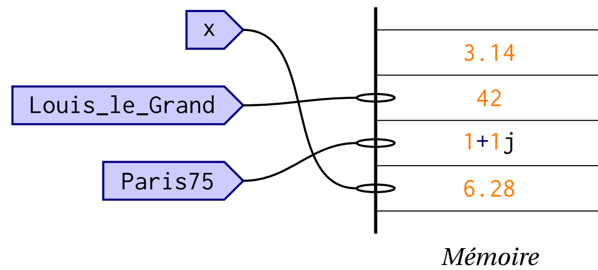
```
In [35]: x
Out[35]: 3.14
```

Lorsque l'on effectue une affectation avec un nom qui existe déjà, l'« étiquette » est simplement déplacée [15]. Ainsi, lorsque l'on écrit

```
In [36]: x = 6.28
```

13. C'est pour éviter la confusion entre le nom « j » et le complexe i que l'on écrit 1j et non simplement j.
14. Attention, l'instruction « 1 = x » n'a pas de sens en Python, car ce qui se trouve à gauche du signe = doit impérativement être l'étiquette !
15. On peut *supprimer* une étiquette via la commande « **del** » : « **del** z » supprime le nom z.

le résultat en mémoire est le suivant [16] :



Une fois un nom défini, il est possible de l'utiliser dans des calculs :

```
In [37]: sin(x)
Out[37]: -0.0031853017931379904

In [38]: x * Louis_le_Grand + Paris75 ** 2
Out[38]: (263.76+2j)
```

Dans une affectation, le membre de droite peut d'ailleurs être une expression :

```
In [39]: z = sin(x) * x**2

In [40]: z
Out[40]: 0.015702920695393533
```

En fait, lorsque l'on soumet à l'interpréteur Python une telle affectation, il tente dans un premier temps d'évaluer ce qui se trouve à droite du signe égal, puis dans un second temps mémorise le résultat et lui associe le nom situé à gauche du signe égal.

Citons au passage la fonction `dir()`, qui permet d'obtenir la liste de tous les noms actuellement connus de l'interpréteur. La plupart des interpréteurs interactifs Python gèrent de façon particulière le nom « _ » qui, de façon automatique, désignera toujours le dernier résultat retourné par l'interpréteur (comme le ferait la touche « ANS » d'une calculatrice).

Signalons enfin que les modules tels `math` ne contiennent pas uniquement des fonctions. On y trouve également des noms associés à des valeurs, qui s'importent de façon similaire aux fonctions. Par exemple, `math.pi` et `math.e` sont des noms qui désignent des valeurs approchées de π et e, dont on peut se servir dans les calculs.

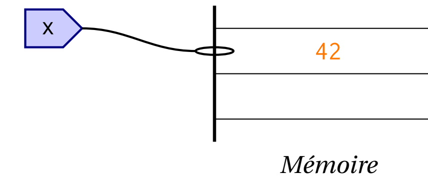
```
In [41]: math.pi
Out[41]: 3.141592653589793
```

16. Si, suite au déplacement ou à la suppression d'une étiquette, un élément en mémoire n'est plus associé à aucune étiquette, comme c'est le cas ici, il sera « supprimé » de la mémoire grâce à un mécanisme automatique appelé *ramasse-miettes* (ou *garbage collector*), de sorte que l'emplacement sera libéré pour mémoriser d'autres résultats.

3.2 Noms, variables

D'un langage de programmation à l'autre, la façon dont est gérée la mémoire de l'ordinateur peut différer quelque peu, aussi tâchons d'en avoir une image la plus claire possible. Supposons par exemple que l'on affecte à un nom `x` la valeur `42` :

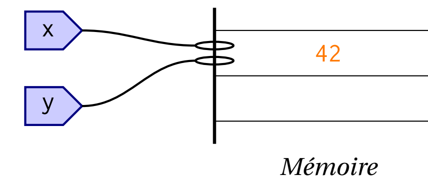
```
In [42]: x = 42
```



Considérons à présent l'affectation :

```
In [43]: y = x
```

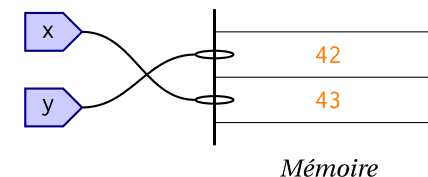
Suite à cette affectation, le nom `y` désignera le *même* emplacement dans la mémoire que le nom `x`. Ainsi, on parvient à cette situation en mémoire :



Si l'on exécute ensuite l'instruction suivante :

```
In [44]: x = x + 1
```

l'interpréteur Python caldule dans un premier temps l'expression « `x + 1` » (dont le résultat est `43`), puis range ce résultat en mémoire, et déplace l'étiquette « `x` », donnant :

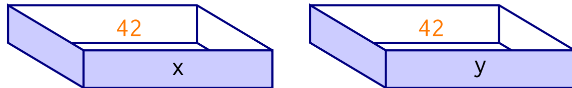


À l'arrivée, l'instruction `x = x + 1` aura eu pour effet d'augmenter la valeur désignée par le nom `x` de 1 (on parle d'*incrémentation*). En revanche, on remarquera cependant que la valeur associée au nom `y` n'a pas changé : l'affectation `y = x` a eu pour effet d'associer le nom `y` à ce que désigne le nom `x` *au moment de l'affectation* [17].

17. L'importance de ce qui se passe ici sera encore plus évidente lorsque l'on manipulera des objets plus complexes, notamment des listes Python.

En informatique, on utilise fréquemment le terme de *variable* pour désigner des associations entre noms et données. Si vous le trouvez dans d'autres ouvrages Python, vous pouvez le considérer comme un synonyme.

Toutefois, le terme de *nom* sera préféré dans ce cours, de préférence à *variable*, car, dans d'autres langages, les « variables » désignent souvent un emplacement fixe en mémoire, plutôt que des étiquettes qui sont déplacées. On peut voir ces variables comme des « boites » dans lesquelles on vient glisser des valeurs. De sorte que deux variables ne désignent jamais la *même* chose, deux boites contenant tout au plus deux *copies* d'une même donnée [18].



Nous verrons que cette représentation n'est pas adaptée au langage Python, même si pour l'instant la différence n'est pas visible.

3.3 Que désigne un nom ?

Un nom n'est pas nécessairement associé à une valeur numérique, comme dans les exemples précédents. Il peut être associé à à peu près n'importe quel objet que Python peut manipuler. Ainsi, les identifiants des fonctions dont on a parlé précédemment sont des noms ! On peut voir ce qu'ils désignent :

```
In [45]: sin
Out[45]: <function math.sin>
```

On voit ici que le nom `sin` désigne la fonction sinus provenant du module `math`. On peut donc « coller une autre étiquette » sur cette fonction :

```
In [46]: f = sin

In [47]: f
Out[47]: <function math.sin>
```

Puisque `f` désigne à présent la fonction sin du module `math`, on peut l'utiliser comme telle :

```
In [48]: f(1.0)
Out[48]: 0.8414709848078965
```

Il convient de faire *très* attention, car il est tout à fait possible de réaffecter les noms associés aux fonctions, Python ne vous empêchera pas de le faire. Ainsi, on peut écrire

```
In [49]: sin = 3.14
```

18. pour de tels langages, le fonctionnement des noms Python s'apparente plutôt à la notion de *référence*.

À présent, le nom `sin` désigne un réel et non plus la fonction sinus...

```
In [50]: sin
Out[50]: 3.14
```

Par conséquent, il n'est plus possible de l'utiliser comme une fonction [19] !

```
In [51]: sin(1.0)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-dacc23301901> in <module>()
----> 1 sin(1.0)

TypeError: 'float' object is not callable
```

Soyez donc prudents ! Et efforcez-vous de ne jamais choisir comme nom ceux de fonctions Python, tout particulièrement si vous êtes susceptible de les utiliser.

3.4 Affectations multiples

Python propose par ailleurs un mécanisme intéressant et assez inhabituel, celui d'*affectations multiples*. Si l'on place à gauche un ensemble de noms séparés par des virgules, et à droite un même nombre d'expressions également séparées par des virgules, il est possible d'effectuer toutes les affectations d'un seul coup. Par exemple :

```
In [52]: a, b = 2, 3.5

In [53]: a
Out[53]: 2

In [54]: b
Out[54]: 3.5

In [55]: c, d = b**a, int(b*3)+1

In [56]: c
Out[56]: 12.25

In [57]: d
Out[57]: 11
```

Les choses se passent dans cet ordre : l'interpréteur Python évalue d'abord les expres-

19. Le message d'erreur obtenu, sans doute quelque peu cryptique pour le moment, précise justement que le nom `sin` désigne un réel (« `float` ») et ne peut être utilisé comme une fonction (« `not callable` »). Nous y reviendrons.

sions à droite du signe =, range les résultats en mémoire, puis effectue les affectations [20], comme on peut le constater sur l'exemple suivant :

```
In [58]: a, b = 2, 3

In [59]: a, b = a+b, a-b

In [60]: a
Out[60]: 5

In [61]: b
Out[61]: -1
```

C'est ce qui fait l'intérêt de cette construction. En effet, dans l'exemple précédent, si l'on avait écrit simplement `a = a+b`, il n'aurait plus été possible, simplement, de faire le second calcul, car la valeur désignée par a a changé !

Cette construction se révèle très utile lorsque l'on souhaite permuter ce que désignent deux noms, comme dans l'exemple ci-dessous :

```
In [62]: x, y = 1, 2

In [63]: x
Out[63]: 1

In [64]: y
Out[64]: 2

In [65]: x, y = y, x

In [66]: x
Out[66]: 2

In [67]: y
Out[67]: 1
```

Dans la plupart des autres langages, qui ne permettent pas ces affectations multiples, il faudrait utiliser un nom « temporaire » :

```
tmp = x
x = y
y = tmp
```

20. Les affectations se font de la gauche vers la droite dans les cas où cela peut avoir une importance, par exemple lorsque l'on écrit `a, a = 2, 3`, après quoi a désigne la valeur 3. Il en est de même pour l'évaluation des expressions à droite du signe =. Il est très rare que cela ait de l'importance, cela dit.

En effet, la solution suivante ne donne pas le résultat attendu, car y ne contiendrait pas la valeur attendue, mais la même que x (soit sa valeur initiale).

```
x = y
y = x
```

Supposons, à présent, que l'on effectue les deux opérations suivantes :

```
In [68]: from math import sin, cos

In [69]: sin, cos = cos, sin
```

Saurez-vous prédire quel résultat quelque peu inattendu donnerait `sin(0.0)` ?

4 L'éditeur

4.1 Utilisation de l'éditeur

Se servir de l'interface interactive n'est pas toujours très pratique. Si l'on commet une erreur dans un calcul, on peut être amené à entrer à nouveau toutes les opérations une par une en repartant du tout début.

Par ailleurs, Python permet de faire un peu plus que des calculs ponctuels, et on veut souvent pouvoir en garder une trace, s'en resservir ultérieurement, ou le transmettre à quelqu'un.

Pour ce faire, on utilisera l'éditeur intégré à l'interface. On peut créer une nouvelle fenêtre d'édition grâce à la commande *file* > *new*. Le contenu de la fenêtre peut être sauvegardé dans un fichier grâce à la commande *file* > *save*. Il est d'usage, pour les fichiers contenant du code Python, d'utiliser l'extension `.py` (elle sera ajoutée automatiquement si l'on ne précise pas d'extension).

Dans la fenêtre de l'éditeur, on peut écrire librement les instructions que l'on souhaite. Elles ne sont pas interprétées dès que l'on presse la touche entrée, donc il est possible d'écrire tranquillement les différentes opérations que l'on veut effectuer sans se soucier de commettre des erreurs.

Lorsque l'on est satisfait, on peut exécuter d'un bloc l'ensemble des commandes dans la fenêtre d'édition en pressant par exemple la touche **F5** [21].

Un message dans l'interpréteur indique la tentative d'exécution du fichier (ici appelé « test.py », et qui contient 25 lignes) :

```
In [70]: (executing lines 1 to 25 of "test.py")
```

21. Les raccourcis claviers peuvent dépendre de l'OS que vous utilisez, on peut les retrouver dans le menu *Run* de l'application.

Si l'une des lignes contient une erreur, l'interpréteur s'arrête et indique le numéro de la ligne qui a causé l'erreur, la ligne proprement dite, et le message d'erreur :

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    x = sin(1.0)
NameError: name 'sin' is not defined
```

Attention, les instructions sont exécutées à la suite de ce qui a déjà été exécuté dans l'interpréteur. Ainsi, les noms déjà définis, par exemple calculés précédemment, seront pris en compte. Cela peut parfois causer quelques désagréments. Il peut parfois être préférable de remettre l'interpréteur à zéro auparavant, grâce à la commande *shell* > *restart* (de raccourci **Ctrl-K**).

Signalons que l'on peut demander l'exécution d'une petite partie du fichier en la sélectionnant et en choisissant *Run* > *Execute selection* (de raccourci *Alt* + *Entrée*).

4.2 Extraire les résultats

Un des inconvénients de cette méthode peut se remarquer très rapidement : si toutes les commandes sont bien envoyées à l'interpréteur et exécutées, les résultats (les lignes « `Out` ») sont perdus !

Bien sûr, il est possible de mémoriser les résultats des calculs en les affectant à des noms. Il suffira d'entrer ces noms dans l'interpréteur pour obtenir lesdits résultats. Cela n'a cependant rien de très pratique.

Il existe, heureusement, une autre façon de faire apparaître les résultats : la fonction `print`. Elle permet d'afficher à l'écran tout ce que l'on souhaite. Ainsi, si l'on demande l'exécution du programme suivant :

```
1+2
x = 2
y = 3
x
y
z = x*y
print(z)
```

On obtiendra dans l'éditeur :

```
In [71]: (executing lines 1 to 7 of "test.py")
6
```

On peut voir que les lignes 1, 4 et 5 ne servent à rien, car le résultat renvoyé par l'interpréteur est purement et simplement perdu. En revanche, la dernière ligne a provoqué l'affichage de 6 dans l'interpréteur.

Pour bien comprendre ce qui se passe, on peut comparer ces deux instructions dans l'interpréteur :

```
In [72]: z
Out[72]: 6

In [73]: print(z)
6
```

Dans le premier cas, l'interpréteur retourne 6 et n'effectue aucune action.

Dans le second cas, l'interpréteur *affiche* 6 et *ne retourne rien*, comme en témoigne l'absence de ligne `Out`.

Il est possible d'afficher plusieurs choses d'un seul coup en plaçant plusieurs éléments séparés par des virgules à l'intérieur des parenthèses :

```
x = 2
y = 3
z = x*y
print(x, y, z)
```

donne

```
In [74]: (executing lines 1 to 4 of "test.py")
2 3 6
```

Par défaut, on le voit ici, les éléments affichés sont automatiquement séparés par des espaces.

Le résultat n'est pas forcément très lisible en l'état, car il n'est pas aisé pour l'utilisateur de se souvenir à quoi correspondent chacune de ces valeurs, aussi est-il possible de faire également figurer du texte, que l'on encadre de guillemets pour indiquer qu'il doit être utilisé en l'état (pour ne pas confondre la lettre x et la valeur désignée par le nom `x` par exemple).

Ainsi, les instructions

```
print("2 fois 3 égale", 2*3)
print("2 fois 3 égale", z)
print("2", "fois", "3", "égale", z)
print(2, "fois", 3, "égale", z)
print(x, "fois", y, "égale", z)
print(x, "fois", y, "égale", x*y)
```

permettront toutes d'obtenir

```
2 fois 3 égale 6
```

Tandis que

```
print("x", "fois", "y", "égale", "z")
```

conduit en revanche à

```
x fois y égale z
```

Par défaut, les différents arguments de la fonction `print` sont séparés par des espaces, et un retour à la ligne est ajouté à la fin. On peut changer ce comportement, comme dans l'exemple ci-dessous :

```
print(2, 3, 4, sep=";", end="\nFin\n")
```

ce qui donnera dans l'interpréteur

```
2;3;4
Fin
```

On remarquera plusieurs choses dans ce dernier exemple, sur lesquelles nous aurons l'occasion de revenir : certaines fonctions ont des arguments nommés (`sep` et `end` dans notre exemple), que l'on place toujours à la fin des arguments de la fonction (et pour lesquels l'ordre n'a pas d'importance).

Par ailleurs, on aperçoit également le code spécial `\n` qui désigne un retour à la ligne. Il existe bien des codes spéciaux, que l'on découvrira en fonction de nos besoins.

Exercices

Ex. 1 – Calculs et priorités

1. Effectuer les calculs suivants à l'aide de l'interface Python. On se posera systématiquement la question de l'utilité des parenthèses dans chaque cas :

$$2+(5\times 7)=\ldots\ldots\ldots\qquad (2+5)\times 7=\ldots\ldots\ldots\qquad 2-(5-7)=\ldots\ldots\ldots$$

$$(2\times 5)^3=\ldots\ldots\ldots\qquad 2\times\left(5^3\right)=\ldots\ldots\ldots\qquad 3^{\left(2^3\right)}=\ldots\ldots\ldots\qquad \left(3^2\right)^3=\ldots\ldots\ldots$$

$$\frac{24}{2\times 3}=\ldots\ldots\ldots\qquad \frac{24}{2}\times 3=\ldots\ldots\ldots\qquad \frac{24}{\frac{2}{3}}=\ldots\ldots\ldots\qquad \frac{\frac{24}{2}}{3}=\ldots\ldots\ldots$$

2. Déterminer (sans utiliser l'interpréteur Python) les résultats que donneront les expressions Python suivantes, puis vérifier que Python fournit bien le résultat attendu.

$$2+5*7=\ldots\ldots\ldots\qquad 2**2**3=\ldots\ldots\ldots\qquad 24/2*3=\ldots\ldots\ldots\qquad 24/2/3=\ldots\ldots\ldots$$

Ex. 2 – Calculs mathématiques et fonctions

Effectuer, à l'aide de Python, les calculs suivants (dans $\mathbb{C}$ au besoin) :

$$\sqrt{2+\sqrt{2+\sqrt{2}}}\qquad \sqrt[3]{7}\qquad \sqrt[2]{i}\qquad \sqrt[3]{-1}\qquad \left|(i+1)^3\right|$$

$$\arccos(0.5)\qquad \tan(\pi/6)\qquad e^{e^2}\qquad \ln\left(2^{30}\right)\qquad \log_{10}(1023)$$

Ex. 3 – Calculs scientifiques

Déterminer, entre la force d'attraction exercée par le Soleil, et la poussée d'Archimède exercée par l'air ambiant, quelle force est la plus grande en norme pour une personne de 70 kg dont le volume est 0,08 m^3. On effectuera les calculs avec l'interpréteur Python.

On fournit la masse du Soleil $\mathrm{M_{Sol}} \simeq 1{,}99\times 10^{30}$ kg, la distance moyenne séparant la Terre du Soleil $d_{\mathrm{S-T}} \simeq 150\times 10^{6}$ km, la constante de gravitation $\mathscr{G} \simeq 6{,}67\times 10^{-11}\ \mathrm{m^3.kg^{-1}.s^{-2}}$, la masse volumique de l'air $\rho_{\mathrm{air}} \simeq 1{,}0\ \mathrm{kg/m^3}$, ainsi que l'accélération de la pesanteur $\|\vec{g}\| \simeq 9{,}81\ \mathrm{m.s^{-2}}$.

Ex. 4 – Utilisation des noms

On donne la séquence d'instructions suivante :

```
a, b, c = 2, 3, 5

a, b, c = b, c, a

a, b, c = 7, a+2, a*a

a, a, a = a, b, c

a, b, c = a*c, b//a, c/a

a, b, c = b+c, a, a%c

a, b, c = (a+b)//(a-c), a**2+1, b//a-4*c
```

1. Déterminer, sans Python, les valeurs désignées par les noms `a`, `b` et `c` après chaque instruction.

2. Vérifier grâce à l'interpréteur Python (on pourra par exemple ajouter une instruction `print(a, b, c)` entre chaque ligne) que les valeurs précédentes sont correctes.