Alain Busser

Prépas
CAPES
AGREG

Python pour les (futurs) professeurs

ellipses

Chapitre 1

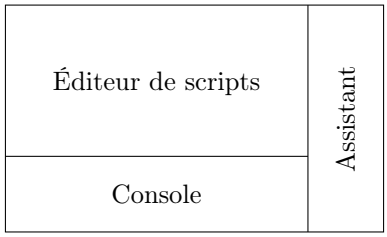
La console

Il y a basiquement deux façons d'utiliser l'application `python` :

- On peut créer un script (écrit en Python) qui est dans un fichier appelé par exemple ***module.py*** puis l'exécuter en entrant en ligne de commande `python module.py`. C'est de cette manière que fonctionnent les logiciels aidant à programmer en Python, comme Mu, Thonny, EduPython, etc., mais aussi ceux permettant d'installer des modules additionnels comme `pip3`.
- On peut aussi entrer directement la commande `python` (sans nom de fichier derrière) qui permet de programmer de façon interactive, ligne par ligne, et s'avère plus efficace, pédagogiquement, que se lancer directement dans le grand bain.

La console de Python n'existe pas seulement en ligne de commande, on la trouve également sur les calculatrices (parfois sous le nom de *shell*) mais aussi dans des logiciels de programmation Python (Anaconda, jupyter, etc.) ou pas ; par exemple le logiciel de topologie `regina-normal` qui permet de faire des calculs dans des espaces topologiques de dimensions 3 ou 4 dotés d'une structure de géométrie hyperbolique (difficile d'imaginer plus mathématique que ça !) est doté d'une console Python, ainsi que le logiciel de traitement d'images `Gimp` (sous la forme du filtre `Python-fu`) et le logiciel de modélisation 3D `Blender` (sous la forme d'une vue pouvant par exemple remplacer celle de la caméra). L'installation de Python depuis le site `https://www.python.org/` donne de toute façon une console appelée IDLE, elle-même programmée en Python.

Dans cet ouvrage, les extraits de dialogue avec la console proviendront en général de Thonny [Ann15], dont la fenêtre est organisée ainsi :



1.1 Fonctionnement

Dans une console Python, on voit régulièrement apparaître un ***prompt***, qui, dans le cas de Python, est représenté par trois caractères ***chevron*** (supérieur, en mathématiques) disposés ainsi :

```
>>>
```

Ce prompt est à considérer comme une parole du *djinn* du conte *al Ad'Din*. Dans le conte, le djinn demande sans cesse à celui qui possède la lampe merveilleuse, quel est le vœu à réaliser. La console Python, quant à elle, demande l'expression à évaluer.

1.1.1 Expressions

Une ***expression*** est une suite de caractères (lettres, chiffres, symboles opératoires, parenthèses) ayant une valeur. Par exemple `2+3` est une expression, tout comme `sin(pi)` ou `(x+1)*(3*x-1)`. Toute expression (bien formée tout au moins) a une valeur et le travail de la console python, c'est justement d'évaluer cette valeur. Pour cela, on va sur la console :

```
>>>
```

on entre l'expression :

```
>>> 2+3
```

puis, pour évaluer l'expression, on appuie sur le bouton Entrée :

```
>>> 2+3
5
```

Une expression (ou plutôt, sa valeur) a un type, qu'on peut connaître en lui appliquant la fonction `type` :

```
>>> type(2+3)
<class 'int'>
```

Le type d'une valeur (un objet Python) est en fait une fonction qui peut transformer un objet de type donné en un objet du nouveau type :

```
>>> int(3.14)
3
```

Ici un réel (le nombre 3,14) a été converti en un entier (celui qu'on obtient en supprimant tout ce qui est après la virgule). La conversion ne marche pas toujours :

```
>>> int(3.1.4)
  File "<stdin>", line 1
    int(3.1.4)
        ^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Le type `type` est lui-même un type :

```
>>> type(type)
<class 'type'>
```

Mais ce n'est pas le plus utile en mathématiques. Un des types les plus importants est celui qui permet de faire de la logique, c'est le type `bool` qu'on verra dans le chapitre sur les ensembles :

```
>>> 1 < 2
True
>>> type(True)
<class 'bool'>
```

Pour vérifier que $2 \leqslant 2$ on écrit :

```
>>> 2 <= 2
True
```

Mais pour vérifier que $2 + 2 = 4$ on n'écrit pas le signe égal :

```
>>> 2+2 = 4
  File "<stdin>", line 1
    2+2 = 4
    ^^^
SyntaxError: cannot assign to expression here.
Maybe you meant '==' instead of '='?
```

En effet, le signe d'égalité est utilisé pour une instruction (voir plus bas). Pour tester une égalité on utilise le double égal :

```
>>> 2+2 == 4
True
```

Les types les plus utiles en mathématiques sont `bool` (propositions logiques), `int` (entiers relatifs) et `float` (réels).
En ce qui concerne les expressions algébriques, Python connaît les priorités opératoires :

```
>>> 2+3*4
14
```

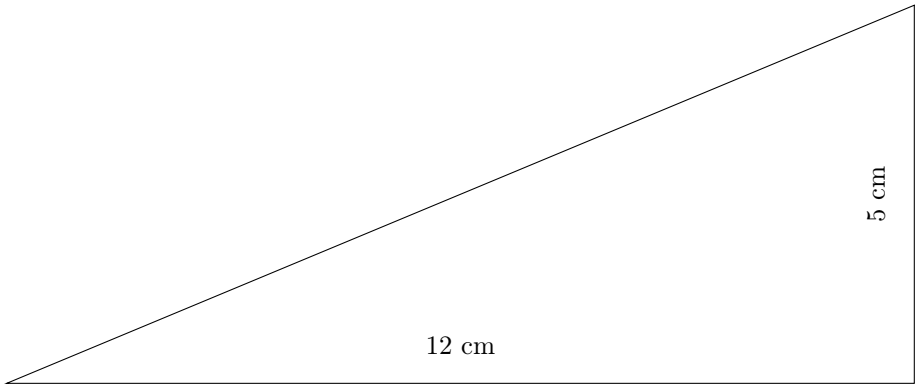
ainsi que la soustraction des relatifs :

```
>>> 2-(-3)
5
```

L'opération d'exponentiation se note par un double astérisque :

```
>>> 3**2
9
>>> 2**3
8
```

Par exemple, si on veut calculer l'hypoténuse de ce triangle rectangle :



On peut commencer par calculer le carré de cette hypoténuse, qui d'après Pythagore, est

```
>>> 12**2 + 5**2
169
```

puis chercher de qui 169 est le carré. Mais comment ***racine carrée*** se traduit en Python ? En anglais on dit *square root* qui s'abrège en `sqrt` :

```
>>> sqrt(169)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

En réalité, le mot `sqrt` n'est pas défini en Python. Il faut donc le définir (par exemple avec l'algorithme de Héron d'Alexandrie ; voir le chapitre sur l'analyse) ou, plus simple, l'importer (ici, depuis le module `math`) :

```
>>> from math import sqrt
>>> sqrt(169)
13.0
```

Remarque : il y a d'autres moyens de calculer l'hypoténuse, par exemple si on sait que $\sqrt{x} = x^{\frac{1}{2}}$:

```
>>> 169**0.5
13.0
```

ou, avec une meilleure connaissance du module `math` :

```
>>> from math import hypot
>>> hypot(12,5)
13.0
```

qui fait utiliser Pythagore par la pythie de Python plutôt que par l'utilisatrice humaine...
Les listes de Python peuvent aussi être considérées comme des expressions, ce qui permet parfois de se passer de boucles :

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

De fait, il est bon d'attendre au moins quelques heures avant de programmer en Python (c'est-à-dire écrire un script) parce que la console est un outil d'apprentissage hors pair, ne serait-ce que pour réviser les priorités opératoires (et aussi, se familiariser avec la syntaxe d'un langage qui est, comme tout langage de programmation, assez long à apprendre).
Mais dès qu'on a appris à programmer en Python (définir des fonctions dans un script) il reste possible de continuer à utiliser la console. Par exemple si on veut calculer (en kg/m^2) un IMC connaissant la masse corporelle et la taille d'un individu, on constate que la fonction `imc` n'existe pas, pas même dans le module `math` :

```
>>> from math import imc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'imc' from 'math' (unknown location)
```

Qu'à cela ne tienne, définissons-la :

```
>>> def imc(masse,taille): return masse/taille**2

>>> imc(80,1.75)
26.122448979591837
```

La définition de la fonction `imc` peut éventuellement être stockée dans un fichier nommé `nutri.py` et ensuite, utilisée dans la console sous peine d'avoir exécuté l'instruction (voir plus bas) suivante :

```
>>> from nutri import imc
```

Même lorsqu'on maîtrise la programmation Python, l'usage de la console pour tester des hypothèses en évaluant des expressions (en particulier avec la fonction `type`) peut s'avérer utile pour la programmation. Ceci dit, si on définit, avec indentation [1] correcte, la fonction `imc` dans un fichier Python, cela donne

```
def imc(masse,taille):
    return masse/taille**2
```

1. L'indentation d'une ligne est le décalage de cette ligne vers la droite.

où l'indentation est bien visible, alors que dans la console on a

```
>>> def imc(masse,taille):
    return masse/taille**2
```

où le prompt gêne la vue. Dans ce livre il faudra donc, à chaque fois que l'indentation apparaît (`def`, `for` etc.) faire abstraction de ce prompt et imaginer que la ligne avec le prompt est reculée pour avoir une indentation correcte sur les autres lignes.

1.1.2 Instructions

Pour faire un peu de trigonométrie en Python, on peut tenter

```
>>> from math import sin,cos,pi
>>> sin(pi/2)
1.0
>>> cos(pi/2)
6.123233995736766e-17
```

qui montre que, pour Python, le cosinus d'un angle droit n'est pas tout-à-fait égal à zéro (voir le chapitre sur l'analyse pour en savoir plus) mais aussi que la première expression `from math import sin,cos,pi` n'a pas été évaluée. En fait ce n'est ***pas une expression***. C'est une instruction. Une instruction n'a pas de valeur (enfin, sa valeur est `None` pour être précis, voir plus bas à propos de `None`) mais par contre, elle a vocation à être exécutée. L'exécution d'une instruction a pour effet de modifier la mémoire de l'ordinateur (ou la position d'un robot). Par exemple

```
>>> from turtle import *
>>> left(45)
>>> forward(100)
```

ne donne aucune évaluation d'expression, mais a pour effet de

- créer un robot virtuel de type *tortue*,
- faire tourner ce robot d'un angle de 45° vers sa gauche,
- faire avancer le robot d'une longueur de 100 pixels,

ce qui produit un dessin (le robot est muni d'un crayon qui trace le parcours effectué, comme un escargot qui dépose sa bave) mais ne calcule rien dans la console.
Des instructions courantes dans la console sont `import`, `from ... import ...`, `assert ...`, `def ...: return ...` (`return` est en soi une instruction). La plus taoïste des instructions est `pass` : elle ne fait rien (*wu wei* en chinois).
Mais les instructions les plus fréquentes sont les affectations, qui servent à créer et modifier des variables.

1.2 Affectation de variables

L'expression `x+2` n'a de valeur que si `x` en a une, ce qui, au début d'une session Python, n'est pas le cas :

```
>>> x+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Si `x` est égal à 3, l'expression a une valeur, à savoir 3+2 :

```
>>> x = 3
>>> x+2
5
```

L'instruction `x = 3` a eu pour effet de créer une variable, ayant

- un nom : `x`,
- une valeur : 3.

Une fois qu'une variable est créée, on ne peut plus modifier son nom, mais on peut modifier sa valeur (c'est pour ça qu'on l'appelle une variable) en lui imposant le résultat d'une évaluation d'expression. Par exemple si on veut que x soit égal à 1 on écrit :

```
>>> x = 1
>>> x+2
3
```

1.2.1 Affectation directe

Souvent, on a besoin d'incrémenter une variable (par exemple pour compter), c'est-à-dire augmenter sa valeur d'une unité (si elle était à 5, la faire passer à 6). Or ceci ne marche pas :

```
>>> compteur = 5
>>> compteur + 1
6
```

En effet, si l'instruction `compteur = 5` a bien eu pour effet que le compteur soit égal à 5, la ligne suivante `compteur + 1` est une expression, et n'a aucun effet :

```
>>> compteur
5
```

Pour que le compteur soit effectivement incrémenté, il faut

- évaluer l'expression `compteur + 1`,
- puis placer le résultat de cette évaluation dans la variable `compteur` :

```
>>> compteur = compteur + 1
>>> compteur
6
```

C'est là qu'un se rend compte que l'utilisation du signe d'égalité (depuis Fortran au milieu des années 1950) pour une affectation, est malheureuse, car il est évident que l'équation $x = x + 1$ n'a pas de solution :

```
>>> compteur == compteur + 1
False
```

En Python, le signe d'égalité ne sert pas à exprimer une égalité, mais une affectation. C'est le signe d'égalité dédoublé qui sert à exprimer (ou tester) une égalité.
Au fait, est-il si évident que cela, que l'équation $x = x + 1$ n'a pas de solution ?

```
>>> x = float('inf')
>>> x == x+1
True
>>> x + 1
inf
```

En fait, $\infty + 1 = \infty$ et $-\infty + 1 = -\infty$ donc l'équation $x = x + 1$ a deux solutions dans $\overline{\mathbf{R}} = [-\infty, +\infty]$ et une solution sur la droite projective puisque sur la droite projective $\infty = -\infty$ (point invariant par les translations).
Mais un compteur a rarement le temps de devenir infini, et on a besoin d'une façon plus intuitive que `compteur = compteur + 1` pour augmenter d'une unité un compteur. Or, tout comme Sofus [Bus18] qui a été créé pour éviter ce problème, Python permet de faire des affectations en place.

1.2.2 Affectations en place

Pour modifier une variable, en général

- on lit son contenu actuel,
- on évalue une expression où intervient ce contenu actuel,
- puis on écrase l'ancien contenu, en le remplaçant par le résultat de l'évaluation.

C'est compliqué. Et pas seulement pour les élèves. Python permet également de modifier directement des variables, *in situ* c'est-à-dire sans passer par l'extraction et la réinjection de données. Pour cela on écrit le symbole d'une opération (addition, soustraction, multiplication, division euclidienne, division exacte, nim-addition, opérations sur les booléens, etc.) entre le nom de la variable et la quantité dont elle doit changer, suivi par un signe d'égalité. Par exemple pour incrémenter une variable `compteur`, il suffit de faire

```
>>> compteur += 1
```

Pour décrémenter une variable, ce sera

```
>>> decompte -= 1
```

Pour doubler une variable, on fait

```
>>> variable *= 2
```

1.2.3 Underscore

Dans la console Python, il y a une variable dont le nom est formé d'une espace soulignée (appelée *underscore* en anglais) que les élèves appellent *tiret du 8* et qui contient le résultat de la dernière évaluation d'expression :

```
>>> 6*7
42
>>> _
42
>>> _+1
43
>>> _
43
```

Quand on a besoin d'une variable (par exemple une variable de boucle) et qu'on ne sait pas comment la nommer, on peut utiliser cette variable presque anonyme :

```
>>> [1 for _ in range(8)]
[1, 1, 1, 1, 1, 1, 1, 1]
```

Cette coutume vient du langage `Prolog` qui utilisait ce symbole pour une place vide c'est-à-dire une variable dont la valeur importe peu.

1.3 Quelques singletons de Python

Python est un langage de programmation objet, c'est-à-dire que les résultats des évaluations des expressions Python sont des objets, dotés de méthodes dépendant de leur type. On dit même que le type d'une variable est déterminé par le jeu de ses méthodes. Par exemple, on sait que 2+3 est un entier parce que si on demande quel est son type, Python répond que c'est un entier. Mais si on demande ses méthodes :

```
>>> type(2+3)
<class 'int'>
>>> dir(2+3)
['__abs__', '__add__', ..., 'real', 'to_bytes']
```

La liste des attributs (comme `denominator`) et des méthodes est très longue, cela veut dire que les entiers sont très riches et savent faire beaucoup de choses. Note : il n'est pas nécessaire d'appliquer la fonction `dir` à un entier particulier, on peut aussi l'appliquer au type correspondant :

```
>>> dir(int)
['__abs__', '__add__', ..., 'real', 'to_bytes']
```

En `Smalltalk` et en `Java`, un nombre est un objet doté de méthodes équivalant à ces méthodes de Python :

- `__lt__()` (*less than*) permettant la comparaison entre nombres,

- `__add__()` permettant d'additionner un nombre avec un autre nombre.

Mais comme il n'y a pas de relation d'ordre compatible avec l'addition sur les nombres complexes, ceux-ci ne sont pas des nombres en Java et en Smalltalk ! En revanche, les chaînes de caractères de Python peuvent être comparées (dans l'ordre alphabétique) entre elles :

```
>>> 'bon' < 'jour'
True
```

et être additionnées entre elles :

```
>>> 'bon' + 'jour'
'bonjour'
```

ce qui ferait d'elles des nombres. Cependant l'addition des chaînes de caractères n'est pas commutative :

```
>>> 'jour' + 'bon'
'jourbon'
```

et, de fait, on ne considère pas les chaînes de caractères comme des nombres, même en Python !

Revenons aux entiers. Un entier n'est pas tout à fait la même chose en Python et en mathématiques : en mathématiques, un entier est un objet ayant certaines propriétés qui font que, par exemple, on peut additionner les entiers entre eux. En Python, un entier est un objet qui possède une méthode `__add__()` qui ***lui permet*** de s'additionner avec un autre entier. Ce n'est pas la mathématicienne qui sait comment additionner des entiers, c'est l'entier qui sait comment s'additionner avec un autre entier.

Une autre différence est que, si on a besoin d'un grand entier (par exemple 2025) au cours d'un calcul, cet entier est créé (c'est-à-dire qu'on lui réserve une place dans la mémoire) pour le calcul, et ensuite, lorsqu'on n'a plus besoin de lui, il est effacé de la mémoire, par un algorithme appelé ***ramasse-miettes*** (`garbage collector` en anglais) et n'existe plus vraiment, jusqu'à ce qu'on ait besoin de lui à nouveau, auquel cas il sera recréé.

C'est comme la théorie de la subitisation, selon laquelle il n'existe pas de neurone reconnaissant le nombre 13, ce qui oblige à compter pour dénombrer 13 objets (et les neurologues mesurent que ça prend plus de temps pour dénombrer 13 objets, que pour en dénombrer 8). Par contre, le fait que les nombres 0, 1, 2 et 3 soient reconnus presque immédiatement laisse penser qu'il y a des neurones qui leurs sont dédiés. L'équivalent en Python est appelé singleton : c'est un objet qui existe en un seul exemplaire et qui a sa place réservée en mémoire, de façon à pouvoir s'y référer chaque fois qu'on en a besoin.

1.3.1 Les petits entiers

Les entiers (typiquement entre -255 et 255) les plus petits sont des singletons. Il ne serait pas utile (ni efficace) que Python recrée l'entier 1 chaque fois qu'il en a besoin,

aussi lui réserve-t-il une place en mémoire. La fonction `id` permet de connaître cette place en mémoire :

```
>>> id(2025)
1444589628368
>>> id(1)
1444540383472
>>> id(2025)
1444589628528
>>> id(1)
1444540383472
```

Chaque appel `id(2025)` a pour effet de réserver une nouvelle place en mémoire pour l'objet 2025 (qui a été recréé pour calculer son `id`) et le résultat de l'évaluation varie d'un appel à l'autre. Cela n'arrive pas pour l'entier 1 parce que, celui-ci étant un singleton, son `id` reste le même au cours d'une séance Python.

1.3.2 Les booléens

Deux autres singletons importants sont `False` et `True`, appelés ***booléens*** en l'honneur de George Boole qui a découvert une structure algébrique sur ces deux singletons. Cette structure sera abordée au chapitre suivant. Il y a aussi une relation d'ordre sur les booléens :

```
>>> False < True
True
```

Pour savoir si une variable `an` vaut 2025, on écrit `an==2025`, mais pour comparer avec un singleton, on utilise le mot-clé `is` qui évite d'avoir à évaluer deux expressions (il y a juste à regarder si l'une est au même endroit que l'autre) et donc plutôt qu'écrire `x == 1` on écrit de préférence `x is 1` qui est non seulement plus rapide à évaluer, mais aussi plus proche de l'anglais naturel.
Si on convertit un booléen en nombre, on a la correspondance notée par George Boole :

```
>>> int(False)
0
>>> int(True)
1
```

Et tout entier non nul est converti par la fonction `bool` en `True` (ainsi que tout flottant non nul, toute liste non vide, tout ensemble non vide, toute chaîne de caractères de longueur non nulle).
`False` et `True` sont deux singletons, et la classe `bool` ne contient que ces deux objets. Mais il y a un autre singleton, qui en plus est seul dans sa classe.

1.3.3 None

En fait les instructions sont considérées comme des expressions mais la valeur de ces expressions est `None` c'est-à-dire rien :

```
>>> False is None
False
>>> None
>>> None is None
True
>>> bool(None)
False
```

`None` est utile pour coder la notion de domaine de définition. Par exemple on veut que 0 n'ait pas de logarithme :

```
>>> from math import log, e
>>> def ln(x): return None if x<=0 else log(x)

>>> ln(1)
0.0
>>> ln(e)
1.0
>>> ln(0)
>>> log(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

La fonction `log` du module `math` calcule le ***logarithme népérien*** (cité pour la première fois par Jakob Bernoulli ; John Neper ne le connaissait pas) et pas le logarithme décimal (qui était celui de Neper). Mais par défaut, l'essai de calcul du logarithme de 0 déclenche l'affichage d'un message d'erreur alors qu'on pouvait se contenter comme on l'a fait ici, de ne pas définir la fonction en 0. On reviendra à cette fonction dans le chapitre sur l'analyse.

1.4 Les modules

Dans ce livre, on essayera autant que possible d'utiliser des modules fournis avec Python, comme `math` (fonctions trigonométriques, exponentielle, etc.), `cmath` (nombres complexes), `fractions` et `decimal` (pour les fractions et les décimaux), `turtle` (dessin vectoriel), `collections` (pour le dénombrement) mais on aura parfois besoin d'autres modules comme `sympy` (calcul formel), `numpy` (calcul matriciel), `matplotlib` (graphiques), `PIL` (traitement d'image)...

Chapitre 2

Ensembles et probabilités

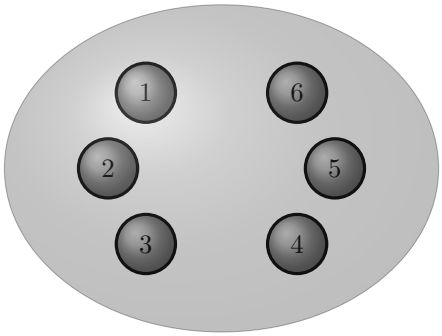
Vers la fin du XIX^{e} siècle, Gottlob Frege a démarré une construction de toutes les mathématiques, basée sur les relations entre ensembles. Dans la première moitié du XX^{e} siècle, c'est sur les ensembles qu'a été basée la construction de toutes les mathématiques, par Georg Cantor, David Hilbert, Bertrand Russel, Alfred Tarski et Nicolas Bourbaki.

2.1 Les ensembles

Les ensembles sont omniprésents en mathématiques : les courbes (droites, cercles, etc.) du plan sont des ensembles de points. Le plan lui-même est un ensemble de points. Résoudre une équation revient à donner l'ensemble de toutes ses solutions.

2.1.1 Ensembles

Supposons qu'on veuille jouer à un jeu de hasard comme le jeu de l'oie mais qu'on ne dispose pas d'un dé. Par contre on a des boules de billard numérotées, et on décide de simuler le jet d'un dé à 6 faces, par le tirage d'une boule au hasard, en l'extrayant d'un sac opaque appelé ***urne***, que voici (on l'a rendu un peu moins opaque pour y comprendre quelque chose) :



L'ensemble (le sac) contenant les 6 boules est noté Ω dans l'axiomatique de Kolmogorov (voir plus bas) mais on le notera `U` (première lettre du mot ***univers*** qui était utilisé par Lewis Carroll). Il existe un objet en Python qui fait varier une variable `x` de 1 à 6 et permet de construire `U` : c'est la fonction `range`. On lui fournit deux arguments qui sont

- La valeur initiale de `x` (ici, 1),
- la première valeur que ne prendra pas `x` (ici, 7)

Cette fonction ne donne pas directement un ensemble :

```
>>> range(1,7)
range(1, 7)
```

Mais la fonction (le type) `set` (ensemble, en anglais) fait la conversion :

```
>>> U = set(range(1,7))
>>> U
{1, 2, 3, 4, 5, 6}
```

Les nombres de 1 à 6 sont dans l'ensemble. On dit qu'ils lui ***appartiennent***. Par exemple $3 \in U$:

```
>>> 3 in U
True
```

mais $7 \notin U$:

```
>>> 7 in U
False
```

2.1.2 Inclusion

On verra dans le prochain chapitre qu'on peut définir des fonctions sur les ensembles. Par exemple la fonction `reste` (dans la division par 2) est définie sur $\mathbf{N}$ donc *a fortiori* sur U :

```
>>> def reste(x): return x%2

>>> {reste(x) for x in U}
{0, 1}
```

L'ensemble $\mathbf{Z}/2\mathbf{Z} = \{0, 1\}$ est l'***image*** de U par la fonction $reste$. Les ***antécédents*** de 0 forment aussi un ensemble, et trouver cet ensemble, c'est ***résoudre l'équation*** $reste(x) = 0$. Ceci permet de définir une nouvelle fonction sur U :

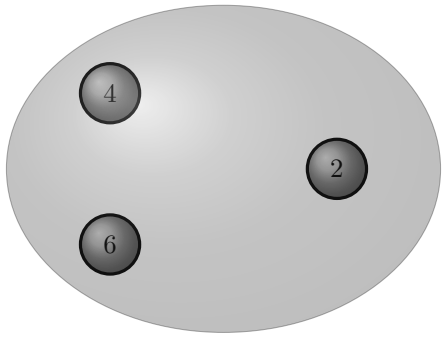
```
>>> def est_pair(x): return reste(x)==0

>>> est_pair(1)
False
>>> est_pair(2)
True
```

Une telle fonction, qui à chaque élément de U, associe `False` ou `True`, est appelée un ***prédicat***. On dit aussi une ***propriété***. Par exemple l'image de 2 par le prédicat `est_pair` est `True` ce qui veut dire que 2 a la propriété d'être pair.
Or il y a une correspondance entre ensembles et prédicats :

- l'appartenance à un ensemble E est un prédicat : ou bien l'élément est dans E, ou il n'y est pas,
- pour un prédicat p, on peut définir l'ensemble des éléments ayant la propriété p. Par exemple l'ensemble des nombres pairs est noté $2\mathbf{Z}$.

L'ensemble A des résultats pairs d'un lancer de dé peut ainsi être défini par ***compréhension*** :

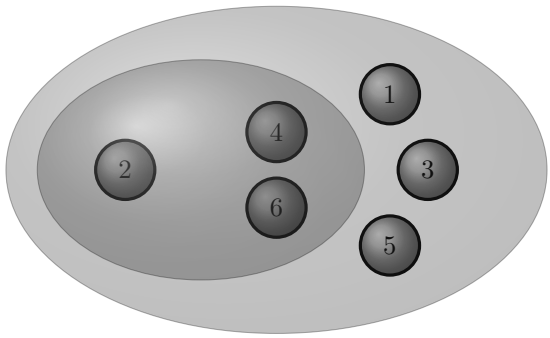
```
>>> A = {x for x in U if est_pair(x)}
>>> A
{2, 4, 6}
```



Remarque : on peut aussi utiliser `range` pour construire l'ensemble A :

```
>>> A = set(range(2,7,2))
>>> A
{2, 4, 6}
```

Bien entendu, par construction, A est entièrement à l'intérieur de U :

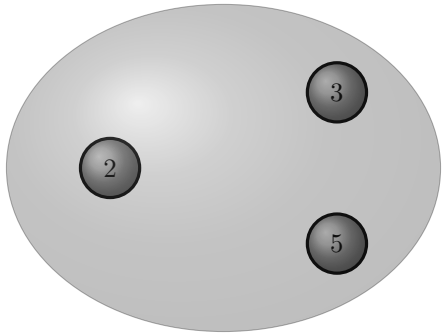


On note $A \subset U$ (A est ***inclus*** dans U) et la relation d'inclusion est connue de Python :

```
>>> A.issubset(U)
True
```

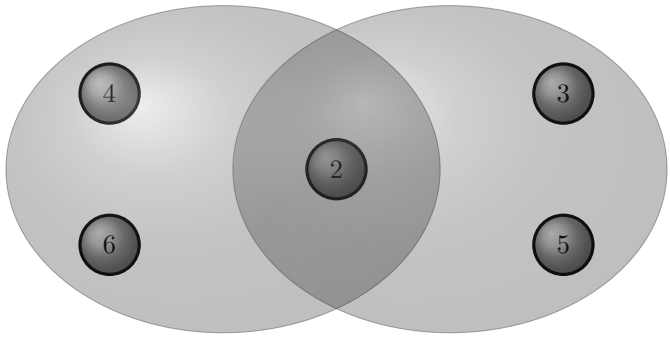
Attention à distinguer l'appartenance qui est une relation entre un élément et un ensemble, de l'inclusion qui est une relation entre deux ensembles. C'est Euler [Eul75] qui a, le premier, pensé à dessiner des ensembles et représenté ainsi leur inclusion. Euler dessinait les sacs avec un compas et on parle de ***cercles d'Euler***.
Pour la suite, on aura besoin d'un autre ensemble, celui des nombres entre 1 et 6 qui sont premiers. On le définit par ***extension*** plutôt que par compréhension :

```
>>> B = {2,3,5}
>>> B
{2, 3, 5}
```



2.1.3 Intersection

L'intersection de A et B est formée de tous les éléments qui sont à la fois dans A et dans B (2, en foncé ci-dessous puisqu'il est dans deux sacs) :



L'intersection de deux ensembles A et B se note $A \cap B$ et se calcule ainsi en Python :

```
>>> A.intersection(B)
{2}
>>> B.intersection(A)
{2}
```

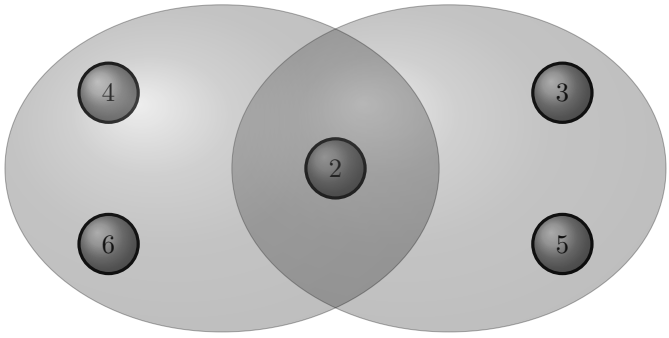
On constate que $A \cap B = B \cap A$. L'intersection admet U comme élément neutre :

```
>>> A.intersection(U)
{2, 4, 6}
>>> B.intersection(U)
{2, 3, 5}
```

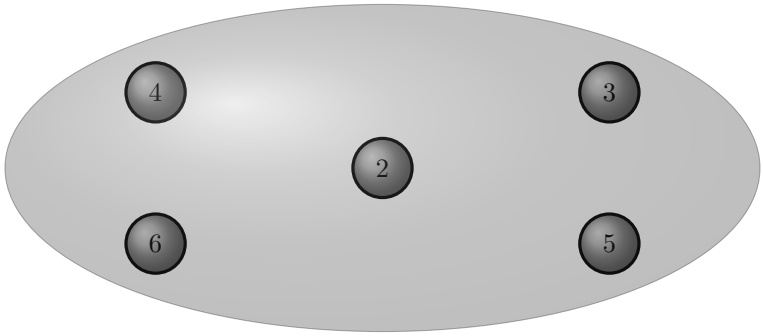
L'intersection a donc des propriétés similaires à la multiplication des entiers, aussi Boole [Boo54] la notait-il par un symbole de multiplication.
En partant du fait que l'inclusion est une relation d'ordre, l'intersection de deux ensembles est leur borne inférieure au sens de l'inclusion.
Lorsque l'intersection de deux ensembles E et F est vide, on dit que E et F sont ***disjoints***.

2.1.4 Réunion

Une fois dessinés les ensembles A et B :



on peut imaginer qu'on dissout les parois internes, et regrouper A et B en un seul sac :



C'est la ***réunion*** de A et B, on la note $A \cup B$ et Python sait aussi la calculer :

```
>>> A.union(B)
{2, 3, 4, 5, 6}
>>> B.union(A)
{2, 3, 4, 5, 6}
```

En partant du fait que l'inclusion est une relation d'ordre, la réunion de deux ensembles est leur borne supérieure au sens de l'inclusion.
L'ensemble `set()` qui ne contient aucun élément est appelé ***ensemble vide*** et noté $\emptyset$. Il est élément neutre pour la réunion ($E \cup \emptyset = E$), laquelle est par ailleurs associative et commutative. Boole la notait donc par le symbole d'addition ($E + F$ plutôt que $E \cup F$ utilisé aujourd'hui). En plus l'intersection est distributive par rapport à la réunion ce qui confère aux ensembles, munis des opérations $\cap$ et $\cup$, une structure d'***algèbre de Boole*** (voir plus bas).

2.1.5 Complément

En théorie des probabilités (voir la fin de ce chapitre), la notion de contraire d'un événement est modélisée par le complément d'un ensemble dans un autre (ici, `U`). En Python, on écrit `difference` :

```
>>> U.difference(A)
{1, 3, 5}
>>> U.difference(B)
{1, 4, 6}
```

Ces ensembles se notent respectivement $U \setminus A = \overline{A}$ et $U \setminus B = \overline{B}$. On peut vérifier les lois de De Morgan $\overline{A \cap B} = \overline{A} \cup \overline{B}$ et $\overline{A \cup B} = \overline{A} \cap \overline{B}$ mais aussi que $\overline{\overline{A}} = A$:

```
>>> U.difference(A.intersection(B))
{1, 3, 4, 5, 6}
>>> (U.difference(A)).union(U.difference(B))
{1, 3, 4, 5, 6}
>>> (U.difference(A)).intersection(U.difference(B))
{1}
>>> U.difference(A.union(B))
{1}
>>> U.difference(U.difference(A))
{2, 4, 6}
>>> U.difference(U.difference(B))
{2, 3, 5}
>>> U.difference(U)
set()
```

Boole utilisait le signe de soustraction pour noter le contraire d'une proposition logique.

2.1.6 Quantificateurs

Frege a introduit les quantificateurs suivants dans la logique des prédicats :

```
>>> all(est_pair(n) for n in range(10))
False
>>> any(est_pair(n) for n in range(10))
True
```

La première proposition se note $\forall n \in \mathbf{N}, 0 \leqslant n < 10 \implies n \in 2\mathbf{Z}$ (elle est évidemment fausse, elle signifie que tous les entiers entre 0 et 9 sont pairs) et la seconde se note $\exists n \in \mathbf{N}, 0 \leqslant n < 10, n \in 2\mathbf{Z}$ (elle par contre est vraie, elle signifie qu'au moins un entier entre 0 et 9 est pair).

2.2 Algèbre de Boole

2.2.1 Définition

Une ***algèbre de Boole*** est un ensemble B ayant au moins deux éléments notés 0 et 1, muni d'une fonction notée $-$ et de deux lois $+$ et $\times$ et ayant les propriétés suivantes (pour tous x, y et z de B) :

- $x + y = y + x$ et $x \times y = y \times x$ (commutativité)
- $+$ et $\times$ sont associatives (pas besoin de parenthèses)
- $\times$ est distributive par rapport à $+$ ($x \times (y + z) = x \times y + x \times z$)
- $x + 0 = x$
- $x \times 1 = x$
- $+$ est distributive par rapport à $\times$ ($x + (y \times z) = (x + y) \times (x + z)$
- $x \times (-x) = 0$
- $x + (-x) = 1$
- $x + x = x$
- $x \times x = x$

et les lois de De Morgan $-(x + y) = -x \times -y$ et $-(x \times y) = -x - y$. Une algèbre de Boole peut être munie d'une relation d'ordre définie par $x \leqslant y \iff x \times y = x$. On a alors $0 \leqslant x \leqslant 1$ et $x \leqslant y \implies -y \leqslant -x$.
On a vu ci-dessus que sur les sous-ensembles de U (les 64 ensembles qui sont inclus dans U) il y a une structure d'algèbre de Boole avec $\cup$ pour l'addition et $\cap$ pour la multiplication, $\emptyset$ jouant le rôle de 0 et U jouant le rôle de 1. La relation d'ordre est donnée par l'inclusion. Mais il y a un exemple plus classique, c'est celui des booléens de Python.

2.2.2 Logique propositionnelle

On simplifie le jeu de dé simulé, en ne gardant qu'une seule boule de Boole :

En Python on définit :

```
>>> faux = set()
>>> vrai = {1}
```

`False` et `True` modélisent la plus petite algèbre de Boole $\mathbf{B} = \{0, 1\}$. La multiplication (ou intersection) s'écrit `and` en Python :

```
>>> faux.intersection(vrai)
set()
>>> False and True
False
>>> faux.intersection(faux)
set()
>>> False and False
False
>>> vrai.intersection(faux)
set()
>>> True and False
False
>>> vrai.intersection(vrai)
{1}
>>> True and True
True
```

L'addition (ou réunion) s'écrit `or` en Python :

```
>>> faux.union(vrai)
{1}
>>> False or True
True
>>> faux.union(faux)
set()
>>> False or False
False
>>> vrai.union(faux)
{1}
>>> True or False
True
>>> vrai.union(vrai)
{1}
>>> True or True
True
```

Le complément à `vrai` s'écrit `not` en Python :

```
>>> vrai.difference(faux)
{1}
>>> not False
True
>>> not True
False
>>> vrai.difference(vrai)
set()
```

2.2.3 Notion d'équation

Une équation est un problème mathématique dont l'énoncé s'écrit sous la forme d'une égalité. En fait, une équation est un prédicat (fonction à valeurs booléennes) ! Par exemple L'équation $x + 2 = 5$:

```
>>> x = 1
>>> x+2 == 5
False
```

On a vu plus haut que, conceptuellement (via la définition en compréhension), un ensemble et un prédicat, c'est la même chose. On peut dire qu'il y a deux sortes de nombres x : ceux pour lesquels on a $x + 2 = 5$, et ceux pour lesquels $x + 2 \neq 5$. Cela permet alors de définir l'ensemble des x de la première sorte (ce sont les solutions de l'équation) :

```
>>> {x for x in U if x+2 == 5}
{3}
```

Résoudre une (in)équation, c'est donner l'ensemble, en général noté S, de ses solutions. Si $S = \emptyset$ on dit que l'équation n'a pas de solution. Mais une (in)équation peut avoir plusieurs solutions, et même une infinité. Par exemple l'inéquation $x + 2 \geqslant 5$ a quatre solutions dans U mais une infinité de solutions dans $\mathbf{N}$:

```
>>> {x for x in U if x+2 >= 5}
{3, 4, 5, 6}
```

Une équation n'a pas forcément vocation à être résolue. Prenons l'exemple d'une équation de droite d. On a vu qu'il s'agit d'un prédicat qui, à un point P du plan, associe la valeur `vrai` si $P \in d$ et `faux` si $P \notin d$. Mais chaque point du plan est repéré par une abscisse x et une ordonnée y (voir le prochain chapitre) et donc une équation de droite est un prédicat sur les couples (x, y). Par exemple on voit ci-dessous que le point de coordonnées $(-2, 3)$ est sur la droite d'équation $2x + 3y = 5$ alors que le point de coordonnées $(2, 3)$ n'y est pas :

```
>>> (x,y) = (-2,3)
>>> 2*x+3*y == 5
True
>>> (x,y) = (2,3)
>>> 2*x+3*y == 5
False
```

Dans le programme de Seconde de 2018 il est proposé comme exemple d'algorithme (sous-entendu, à programmer en Python), de déterminer une équation de droite passant par deux points donnés (sous-entendu, par leurs coordonnées). C'est probablement l'écriture de l'équation (ses coefficients) qui est attendue, parce que l'équation elle-même ressemble plutôt à ceci :