Introduction à Python (pour les maths)

Mickaël Péchaud

2017

Table des matières

1 Démarrage **2**
1.1 Python . . . 2
1.2 Environnements de développement (IDE) . . . 2
1.2.1 Pyzo . . . 2
1.2.2 Idle . . . 3
1.3 Installation . . . 3

2 Syntaxe de base **4**
2.1 Types . . . 4
2.2 Fonctions mathématiques usuelles . . . 8
2.3 Variables . . . 9
2.4 Interactivité (minimale) . . . 11
2.4.1 Print . . . 11
2.4.2 Input . . . 12

3 Branchements conditionnels **13**

4 Fonctions **14**
4.1 Définir une fonction . . . 14
4.2 Quelques bonnes habitudes . . . 15
4.3 Portée des variables . . . 16

5 Boucles **16**
5.1 Boucle for . . . 17
5.1.1 Avec *range* . . . 17
5.1.2 À partir d'une liste . . . 18
5.2 Boucle while . . . 19

6 Lecture/Écriture dans un fichier texte (.txt) **22**
6.1 Lecture . . . 22
6.2 Écriture . . . 23

7 Graphiques **25**
7.1 Bases sur la commande plot . . . 25
7.2 Exemple : tracé d'une fonction avec *numpy* . . . 26
7.3 Exemple : tracé d'une famille de courbes . . . 28
7.4 Exemple : un graphique interactif . . . 28
7.5 Exemple : une animation . . . 29

8 Deux thèmes mathématiques **31**
8.1 Suites . . . 31
8.1.1 Une comparaison puissance/exponentielle . . . 31
8.1.2 Une série alternée . . . 31
8.1.3 Représentation graphique et animation de suites récurrentes . . . 32
8.1.4 Les sujets auxquels vous avez échappé . . . 33
8.2 Probabilités . . . 34
8.2.1 Le module *numpy.random* . . . 34
8.2.2 Simulation d'une loi binomiale . . . 34

8.2.3 Marche aléatoire 1D . . . 35
8.2.4 Tortue et marche aléatoire 2D . . . 35
8.2.5 Approximation de π par une méthode de Monte-Carlo . . . 35
8.2.6 Les sujets auxquels vous avez échappé . . . 36

A Très courte bibliographie **37**

Préambule

Ce document est distribué sous licence CC BY-NC-SA 3.0 FR.
https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

Vous pourrez retrouver ce poly, son corrigé, ainsi que les fichiers *Python* à l'adresse suivante :
http://mpechaud.fr/formationpython

Pour toutes remarques, questions, merci de vous adresser par mail à l'auteur de ce poly (prenomnom@gmail.com) !

1 Démarrage

1.1 Python

Python est un langage de programmation dont la première version est apparue en 1991. Il s'est fortement développé depuis, et est fréquemment cité dans le trio de tête des langages les plus utilisés. Il s'agit d'un langage **libre** et **gratuit**.
Il existe plusieurs versions de *Python*, avec 2 branches principales : *Python 2* et *Python 3*. Ce poly ne parle que de *Python 3*, qui est appelé à devenir la norme d'ici quelques années. Pour ce dont il sera question ici, les différences ne sont de toute façon pas énormes.

1.2 Environnements de développement (IDE)

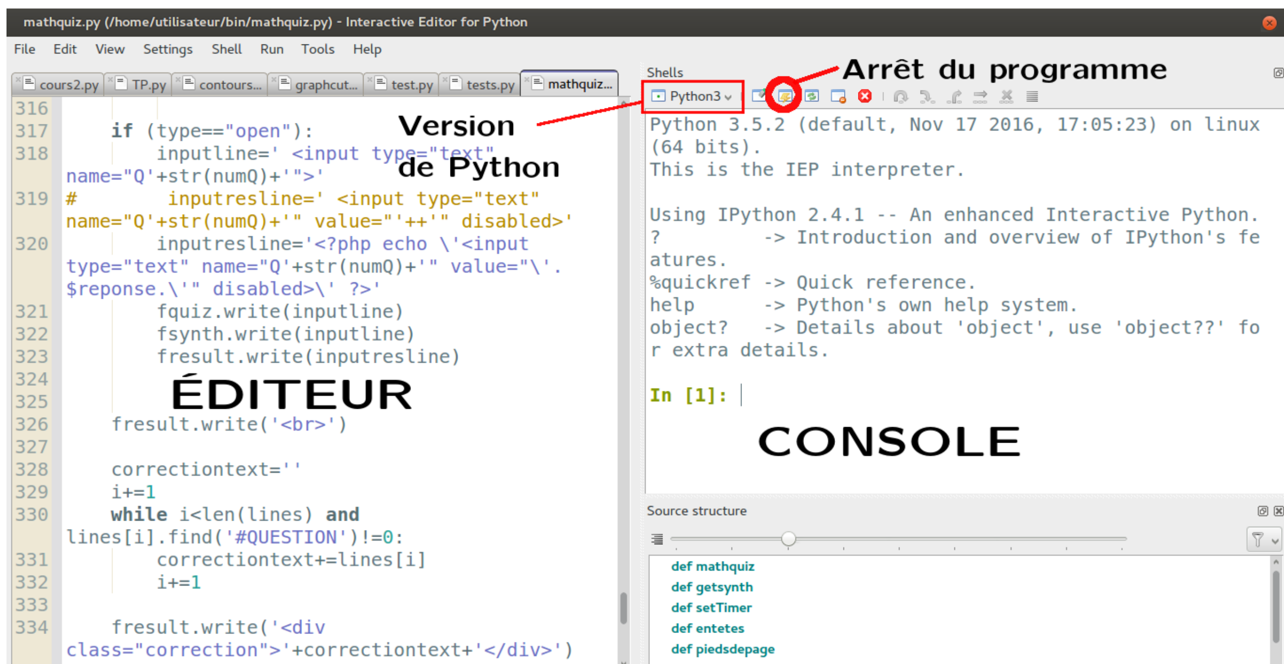
Il existe plusieurs *environnements de développement* libres et gratuits pour *Python*. Il s'agit d'interfaces graphiques permettant de taper du code *Python*, de l'exécuter, d'accéder à des pages d'aide...

1.2.1 Pyzo

Je propose de travailler dans l'environnement de développement *Pyzo* [1].
Pyzo, comme la plupart des environnements de développement, propose 2 manières de taper du code :

- On peut exécuter une ligne de code en utilisant la *console* (ou *shell*). Dès que l'on appuie sur la touche *entrée*, la ligne est exécutée, et le résultat affiché.
- On peut taper plusieurs lignes à la suite dans un fichier à l'aide de l'*éditeur de texte*. Il y a alors des raccourcis clavier permettant d'exécuter tout ou partie des instructions contenues dans le fichier.

1. Dans le cartable numérique, vous trouverez l'environnement *Pyscripter*, qui présente les mêmes fonctionnalités de base – il est cependant un peu moins pratique à mon goût.



Quelques raccourcis utiles dans *Pyzo*

- Dans la console, la *flêche vers le haut* permet de naviguer dans les commandes précédemment exécutées.
- Lorsque vous commencez à taper un nom de variable, de fonction, etc, *Pyzo* vous propose des complétions possibles. Utilisez les *flêches* et *tab* pour en sélectionner une.
- *Alt+entrée* permet d'exécuter la ligne courante, ou la sélection.
- *Ctrl+entrée* permet d'exécuter la cellule courante (délimitée par des lignes commençant par ##).

1.2.2 Idle

À noter : il existe d'autres environnements de développement. *Idle* est fourni par défaut lorsque vous installez *Python* sur votre machine, et se trouve également dans le cartable numérique. Je le trouve cependant moins agréable à manipuler que *Pyzo*.
Sous *Idle*, seule une console est affichée par défaut. Il faut taper `Ctrl-N` pour ouvrir un nouveau fichier.

1.3 Installation

Le cartable numérique contient *Python* 3, l'environnement de développement *Pyscripter*, ainsi que la plupart des modules utiles.
À défaut, vous pouvez télécharger (gratuitement) la dernière version stable de *Python* depuis le site de *Python* :
https://www.python.org/downloads/

Pyzo se trouve à l'adresse suivante :
http://www.pyzo.org/start.html

Pour les linuxiens sous Ubuntu, installez simplement les paquets *python3* et *iep*.

2 Syntaxe de base

2.1 Types

On peut travailler avec différents *types* de données en python. On peut obtenir le type d'un objet à l'aide de la commande `type`.

Les entiers (*int*). Munis de toutes les opérations usuelles. Virtuellement pas de limite de taille.

Code Python :

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

>>> 99**99  #puissance
369729637649726772657187905628805440595668764281741102
302599724235525704552775234214106500101282327279409788
954832654011942999676949435945162157019364401441807106
66765930138499977999915920049989 9

>>> 5//3  #quotient de la division euclidienne
1

>>> 5%3  #reste de la division euclidienne
2
```

Les booléens (*bool*)

Maths	Python
$=$	`==`
$\neq$	`!=`
$<$	`<`
$\leqslant$	`<=`
et	`and`
ou	`or`
non	`not`
vrai	`True`
faux	`False`

Code Python :

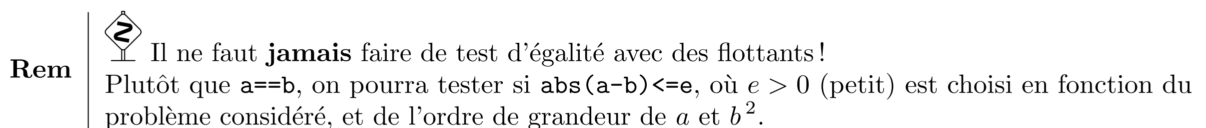
```
>>> not (2>=3 and 4==4) or 2>=3.4
True

>>> 2 !=3 and 4<3
False
```

Les «réels» = nombres à virgule flottante (*float*). Attention, ce sont des *valeurs approchées* de réels...

Code Python :

```
>>> type(2.4)
<class 'float'>

>>> 99**99.  #puissance
3.697296376497268e+197

>>> 999**999.
  File "<stdin>", line 1, in <module>
OverflowError : (34, 'Numerical result out of range')

>>> 10**-350
0.0

>>> 5/3
1.6666666666666667

>>> 1.4-0.4
0.9999999999999999

>>> 1.4-0.4 == 1
False

>>> 1-2**(-53)==1
False

>>> 1-2**(-54) == 1
True

>>> 1+2**(-54)-1 == 1-1+2**(-54)
False

>>> 99**99.==99**99.+1
True
```

Rem | En 64 bits, les flottants ont une précision relative d'environ 10^{-16}.



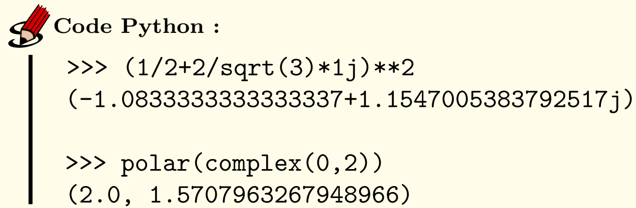
Rem | Il ne faut **jamais** faire de test d'égalité avec des flottants !
Plutôt que `a==b`, on pourra tester si `abs(a-b)<=e`, où $e > 0$ (petit) est choisi en fonction du problème considéré, et de l'ordre de grandeur de a et b [2].

Pour plus de précision sur la représentation des réels en machine :
https://fr.wikipedia.org/wiki/IEEE_754

2. Une solution plus robuste, qui évite d'avoir à connaitre a priori l'ordre de grandeur de a et b : utiliser la fonction `isclose(a,b)` du module *math* (cf ci-après), qui en gros regarde si $|a-b| \leqslant e \max(a,b)$, où $e = 10^{-9}$ par défaut.

Les complexes (*complex*).

Syntaxe assez peu agréable. i s'écrit $1j$, ou `complex(0,1)`

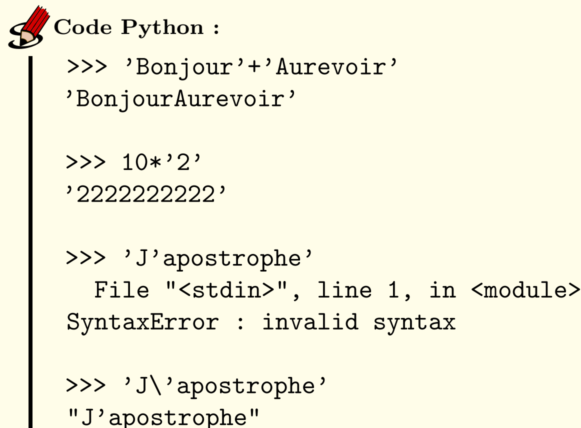


Code Python :

```
>>> (1/2+2/sqrt(3)*1j)**2
(-1.0833333333333337+1.1547005383792517j)

>>> polar(complex(0,2))
(2.0, 1.5707963267948966)
```

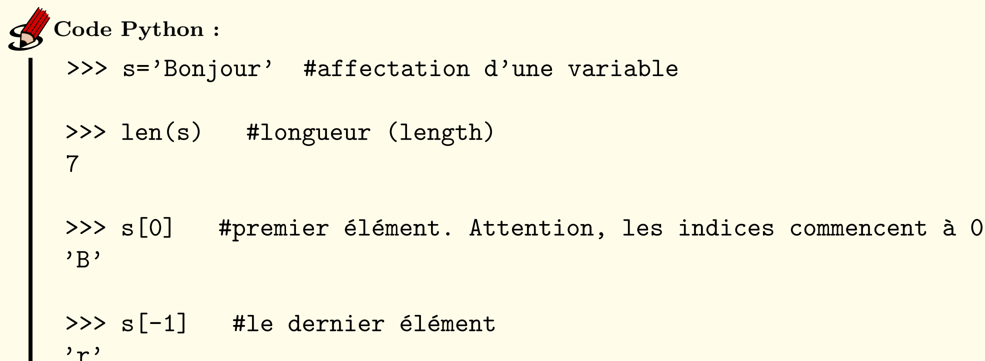
Les fonctions usuelles peuvent être appliquées aux complexes après importation du module *cmath* (`from cmath import *`).
https://fr.wikibooks.org/wiki/Math%C3%A9matiques_avec_Python_et_Ruby/Nombres_complexes_en_Python

Les chaînes de caractères (*str*)



Code Python :

```
>>> 'Bonjour'+'Aurevoir'
'BonjourAurevoir'

>>> 10*'2'
'2222222222'

>>> 'J'apostrophe'
  File "<stdin>", line 1, in <module>
SyntaxError : invalid syntax

>>> 'J\'apostrophe'
"J'apostrophe"
```



Code Python :

```
>>> s='Bonjour'  #affectation d'une variable

>>> len(s)   #longueur (length)
7

>>> s[0]   #premier élément. Attention, les indices commencent à 0
'B'

>>> s[-1]   #le dernier élément
'r'
```

Particularité de *Python* : les extractions de sous-chaînes sont facilitées par rapport à beaucoup d'autres langages par les mécanismes d'*extraction de tranche* (ou saucissonnage, ou encore *slices*).
La syntaxe est `s[debut :fin :pas]`. L'indice de début est inclus, celui de fin exclu. *debut*, *fin* et *pas* sont facultatifs. S'ils ne sont pas précisés, ils sont respectivement remplacés par 0, $len(s)$ et 1 [3].

3. Il est déconseillé d'utiliser ces 3 paramètres à la fois : cela donne rapidement du code difficile à lire. . .

Code Python :

```
>>> s='Bonjour'

>>> s[2 :4]  #2 inclus, 4 exclus.
'nj'

>>> s[2 :]   #de 2 jusqu'à la fin
'njour'

>>> s[ :-2]  #du début jusqu'à l'avant-avant dernier
'Bonjo'

>>> s='abcdefghijkl'

>>> s[0 : :2] #les indices pairs
'acegik'

>>> s[1 : :2] #les indices impairs
'bdfhjl'

>>> s[ : :-1] #retournement de la chaîne
'lkjihgfedcba'
```

La page suivante couvre toutes les fonctionalités des extractions de tranches :
https://zestedesavoir.com/tutoriels/582/les-slices-en-python/

Les tuples (*tuple*)

Sur lesquels les mécanismes d'extraction de tranches fonctionnent également.

Code Python :

```
>>> (1,2,3)+(4,5,6)
(1,2,3,4,5,6)

>>> 10*(0,)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

>>> t=(2,3,4,6,2,3)

>>> t[2 :4]
(4,6)

>>> t[ : :-1]
(3, 2, 6, 4, 3, 2)
```

Les listes (*list*)

Comme les tuples, avec des crochets au lieu des parenthèses. Contrairement aux tuples, il est possible d'aller modifier directement un élément d'une liste (on dit qu'il s'agit d'un objet *mutable*), ce qui pose tout un tas de difficultés sur lesquelles nous reviendrons plus tard.

À noter qu'il est possible de définir une liste à l'aide d'une syntaxe très proche de la notation en compréhension pour définir un ensemble :

Code Python :

```
>>> l=[i**2 for i in range(10)]

>>> l
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> len(l)  #longueur (length)
10

>>> l[3]=42

>>> l
[0, 1, 4, 42, 16, 25, 36, 49, 64, 81]

>>> l=l+[2]
[0, 1, 4, 42, 16, 25, 36, 49, 64, 81, 2]

>>> l=[1,2]+l
[1, 2, 0, 1, 4, 42, 16, 25, 36, 49, 64, 81, 2]
```

2.2 Fonctions mathématiques usuelles

Les fonctions usuelles ne sont pas présentes par défaut dans *Python*, mais sont disponibles dans le module `math`. Il faut les *importer* (une fois pour toute) à l'aide de la commande `from math import *`[4].
Les constantes e et π sont alors définies :

Code Python :

```
>>> e
2.718281828459045
>>> pi
3.141592653589793
```

On a ensuite accès (entre autres) aux fonctions suivantes :

4. Cette commande permet d'importer d'un coup toutes les fonctions du module (`*` est un raccourci pour «tout»). La commande `import math` a presque le même effet, mais pour accéder à la commande `cos`, on devra la préfixer par le nom du module : `math.cos`. Cela peut se rélèver utile pour éviter des conflits entre plusieurs modules qui contiendraient des fonctions de noms identiques. . . On créé souvent dans ce cas un alias pour avoir moins de caractères à taper : par exemple, on tapera `import math as m`, puis `m.cos` pour accéder à la fonction `cos`. Enfin, pour éviter de surcharger la mémoire, on peut également importer une par une les fonctions d'un module – en utilisant par exemple `from math import cos`.

Code Python :

```
>>> floor(-3.4) #partie entière
-4

>>> sqrt(2)
0.7071067811865475
>>> exp(2)
7.38905609893065
>>> log(e) #ln
1.0

>>> cos(0)
1.0
>>> sin(pi/4)
0.7071067811865475
>>> tan(pi/4)
0.9999999999999999
>>> asin(1)
1.5707963267948966
>>> acos(0)
1.5707963267948966
>>> atan(1)
0.7853981633974483

>>> cosh(0)
1.0
>>> sinh(3)
10.017874927409903
```

`import math ;help(math)` donne une liste exhaustive des fonctions du module. On peut également demander de l'aide (en anglais) sur telle ou telle fonction (par exemple : `help(cos)`)
À noter également les commandes `degrees` et `radians`, qui permettent d'effectuer les conversions entre les 2 unités :

Code Python :

```
>>> degrees(pi/2)
90.0

>>> radians(180)
3.141592653589793
```

2.3 Variables

Une *variable* permet de stocker une valeur pour la réutiliser par la suite.
On utilise le symbole `=` pour *affecter* une *valeur* à une variable.

Code Python :

```
>>> a=3
>>> a
3
>>> b=a
>>> b
3
>>> a=-1.3
>>> b
3
```

Il est possible de faire des affectations multiples en utilisant les tuples :

Code Python :

```
>>> (a,b)=(2,4)
>>> a
2
>>> b
4
```

Utilisation classique de ce mécanisme : l'interversion des valeurs contenues dans deux variables.

Code Python :

```
...
>>> (a,b)=(b,a)
...
```

À comparer à la méthode moins lisible qui prévaudrait dans beaucoup d'autres langages de programmation :

Code Python :

```
tmp=a
a=b
b=tmp
```

Exercice 1 Qu'affiche l'exécution des lignes de code suivantes :

Code Python :

```
>>> a=1

>>> b=2

>>> c=a

>>> b=c

>>> a=b

>>> (a,b,c)
```

Code Python :

```
>>> a=1

>>> b=2

>>> c=a

>>> a=b

>>> b=c

>>> (a,b,c)
```

2.4 Interactivité (minimale)

(→ à partir d'ici, pour ne pas avoir à retaper tous les programmes, vous trouverez les sources dans le fichier `./sectionparsection.py`)

2.4.1 Print

La commande `print` permet d'afficher quelquechose lors de l'exécution d'un programme. Cela va être utile pour écrire des programmes de plusieurs lignes dans l'éditeur (dans la console, le quantité calculée est automatiquement affichée, mais ça n'est pas le cas dans l'éditeur !).

Code Python :

```
a=3
print(a)
print('a')
print('La valeur de a est',a)
```

```
3
a
La valeur de a est 3
```

À noter l'existence de la commande `format`, très puissante pour formater des chaînes de caractères, avec notamment de nombreuses possibilités pour gérer l'alignement, le format d'affichage des réels, etc. Un exemple de base :

Code Python :

```
>>> x=4
>>> print("Le carré de {} est {}".format(x,x**2))
Le carré de 4 est 16
```

Les `{}` sont remplacés respectivement par x et $x**2$.
L'intérêt de cette commande est que l'on peut préciser des options de formatage entre les accolades. Voici en anticipant un peu sur les boucles, l'affichage d'un début de table de ln :

Code Python :

```
print('{ :>4}{ :>14}'.format('x','ln(x)'))
for k in range(1,11) :
    print('{ :>5.1f}{ :>15.3e}'.format(k/10,log(k/10)))
```

```
   x          ln(x)
  0.1     -2.303e+00
  0.2     -1.609e+00
  0.3     -1.204e+00
  0.4     -9.163e-01
  0.5     -6.931e-01
  0.6     -5.108e-01
  0.7     -3.567e-01
  0.8     -2.231e-01
  0.9     -1.054e-01
  1.0      0.000e+00
```

`{ :>4}` signifie que l'on écrit sur 4 caractères, en alignant à droite. `.1f` signifie que l'on veut 1 décimale lors de l'affichage du flottant. `.3e` que l'on veut un affichage en notation scientifique, avec 3 décimales...
Plus d'informations sur le sujet aux adresses suivantes :
https://www.afpy.org/doc/python/2.7/tutorial/inputoutput.html
https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3

2.4.2 Input

La commande `input` permet de demander une valeur à l'utilisateur en cours d'exécution. Elle renvoie une chaîne de caractères, que l'on doit convertir en entier ou en flottant le cas échéant.
Cela ne fait pas vraiment sens dans la console. L'exemple qui suit devra donc plutôt être tapé dans l'éditeur, puis exécuté en bloc.

Code Python :

```
a=float(input('Entrez un réel  : '))
print('Le carré de votre réel',a,'est',a**2)
```

3 Branchements conditionnels

Les *tests* (ou *branchement conditionnels*) permettent d'effectuer une série d'instructions si et seulement si une certaine condition est vérifiée.
La syntaxe est la suivante :

if *condition* :
␣ *bloc d'instructions*

Rem | Le bloc d'instruction doit être *indenté* par rapport au `if`, c'est-à-dire décalé de 4 espaces (que l'on peut obtenir directement à l'aide de la touche *tabulation*). Un éditeur correct doit placer le curseur au bon endroit lorsque vous passez à la ligne après un `if`.

Rem | Il est vivement conseillé que la condition soit un booléen.

Code Python :

```
a=float(input('Entrez un réel positif : '))
if a<0 :
    print('Vous n'avez pas rentré un réel positif...')
```

Exercice 2 Quelle est la différence entre les 2 programmes suivants :

Code Python :

```
a=int(input('Entrez un entier : '))
if a==0 :
    print('Votre entier est nul.')
print('Au revoir')
```

Code Python :

```
a=int(input('Entrez un entier : '))
if a==0 :
    print('Votre entier est nul.')
    print('Au revoir')
```

Il est possible d'ajouter des instructions à effectuer lorsque la condition n'est pas vérifiée.

Code Python :

```
a=int(input('Entrez un entier : '))
if a>=0 :
    print('votre entier est positif')
else :
    print('votre entier est strictement négatif)
```

Exercice 3 En imbriquant 2 instructions conditionnelles, écrire un programme qui demande un entier à l'utilisateur, et affiche un message qui dit si cet entier est nul, strictement positif ou strictement négatif.

Le programme de l'exercice précédent peut être compacté de la façon suivante :

Code Python :

```
a=int(input('Entrez un entier : '))
if a>0 :
    print('a est strictement positif')
elif a==0 :   #else-if compacté
    print('a est nul')
else :
    print('a est strictement négatif')
```

4 Fonctions

4.1 Définir une fonction

Lorsque l'on a besoin d'utiliser plusieurs fois une même série d'instructions, il faut les mettre sous forme de *fonction*. Une fonction possède :

- un **nom** – que l'on choisira de façon à ce qu'il soit suffisamment explicite ;
- des **paramètres**, qui sont ses entrées ;
- d'un **résultat**, ou *valeur de retour*

La syntaxe de base pour déclarer une fonction est la suivante :

def nom(*paramètres*) :
␣ """"description de la fonction""""
␣ *bloc d'instruction*
␣ return *valeur de retour*

Rem
- La description de la fonction est en partie normalisée, et permet par exemple de générer automatiquement des pages d'aide, des jeux de test, etc.
- Il peut y avoir plusieurs `return` pas nécessairement placés à la fin de la fonction.

Voici une fonction qui calcule le minimum de deux réels :

Code Python :

```
def mon_min(a,b) :
    """renvoie le minimum des deux réels"""
    if a<=b :
        return a
    return b
```

Lorsque l'on exécute cette fonction, rien n'est affiché. Elle a simplement été enregistrée par *Python*. Il est alors possible de l'utiliser dans la console, ou dans une autre fonction :

Code Python :

```
>>> a=mon_min(1,2.4)
>>> a
1
```

Une fonction peut ne pas renvoyer de résultat :

Code Python :

```
def dis_bonjour(nom) :
    """affiche un message de bienvenue personalisé"""
    print('Bonjour',nom,' !')
```

Rem

Ne pas confondre :

- `return`
  - — Quitte immédiatement la fonction.
  - — *Renvoie* un résultat, que l'on peut stocker dans une variable, utiliser dans une autre fonction...
- `print`
  - — *Affiche* quelquechose.
  - — On ne peut rien faire dans le langage de ce quelquechose, qui est seulement destiné à être lu (et dont on ne pourra plus rien faire en *Python*).
  - — L'exécution de la fonction suit son cours.

Exercice 4 Sans utiliser `abs`, écrire une fonction `mon_abs(a)`, qui renvoie la valeur absolue du réel `a` passé en argument.

Exercice 5 Écrire un programme qui prend en arguments `a,b` et `c`$\in \mathbb{R}$, et affiche les solutions réelles de $ax^2 + bx + c = 0$.

4.2 Quelques bonnes habitudes

- Toute série d'opérations destinée à être effectuée plusieurs fois devrait faire l'objet d'une fonction.
- Mieux valent trois fonctions simples qu'une fonction compliquée.
- Tester une fonction avec différents paramètres immédiatement après l'avoir écrite : il peut s'avérer délicat de détecter un bug dans une fonction faisant appel à 4 autres fonctions qui n'ont pas été testées individuellement...
- Donner un nom suffisamment explicite aux fonctions (ne pas hésiter à avoir des noms un peu longs et à utiliser l'auto-complétion)
- Donner un nom raisonnable aux variables (`l` pour une liste, `c` pour un entier servant de **c**ompteur, `i`, `j` ou `k` pour une variable entière de boucle, etc.)
- Décrire une fonction (entrées, sorties) dans la ligne suivant sa déclaration.
- Dès que le contenu de la fonction est non-trivial, commenter le code à l'aide de `#`.

Code Python :

```
def gabuzomeu(l1) :
    bonjour=0
    while not(l1==0) :

        if l1%2==1 :
            bonjour+=1
        l1//=2
    return bonjour
```

Code Python :

```
def nombre_1_binaire(n) :
    """prend en argument un entier naturel n,
    et renvoie le nombre de 1 dans la
    représentation binaire de n"""
    c=0
    while n !=0 :
        if n%2==1 : #si n est impair
            c+=1
        n=n//2
    return c
```

4.3 Portée des variables

Les variables utilisées à l'intérieur d'une fonction sont appelées **variables locales**. Une variable locale n'est pas visible à l'extérieur de la fonction.

Code Python :

```
def test(a) :
    b=2
    a=a+1
    return a+b

____________________________________

>>> a=-1
>>> b=-1
>>> test(a)
2
>>> a
-1
>>> b
-1
```

On parle de **variable globale** pour une variable définie à l'extérieur de toute fonction. L'usage des variables globales doit être limité au strict nécessaire. Il est impossible de modifier une variable globale dans une fonction [5].

Code Python :

```
#vitesse de la lumière, en m.s^-1
c=299792458

def distance_km_min(m) :
    """calcule la distance en km parcouru par la lumière en m minutes"""
    return 60*m*c/1000

def temps_km_min(k) :
    """calcule le temps en min mis par la lumière pour parcourir k km"""
    return k*1000/(60*c)

def test() :
    c=c+1
__________________________________________________________________________________________
>>> distance_km_min(60)
1079252848.8
>>> temps_km_min(1)
5.5594015866358675e-08
>>> test()
.....
UnboundLocalError : local variable 'c' referenced before assignment
```

5 Boucles

Les boucles sont des instructions qui permettent de répéter plusieurs fois un bloc d'instructions donné.

5. À moins de la déclarer juste après la déclaration de la fonction à l'aide du mot clef `global`. C'est rarement une bonne idée

5.1 Boucle for

5.1.1 Avec *range*

Une *boucle for* permet de répéter un nombre de fois prédéterminé une séquence d'instructions.
Voici la syntaxe la plus proche de la plupart des autres langages de programmation :

> **for** *compteur* **in** **range**((*debut*),*fin*) :
> ␣ *bloc d'instructions*

Au premier passage dans la boucle, le compteur prend la valeur début (0 si elle est omise). Après le première passage, le compteur est augmenté de 1 (on dit qu'il est *incrémenté*). On sort de la boucle lorsque la valeur de fin est atteinte ou dépassée (la fin est donc exclue : le compteur ne vaut jamais *fin* lors de l'exécution de la boucle).

Code Python :

```
for k in range(10) : #10 passages dans la boucle
    print ('bonjour',end=',')
```

```
bonjour,bonjour,bonjour,bonjour,bonjour,bonjour,bonjour,bonjour,bonjour,bonjour,
```

La variable k n'étant pas utilisée dans le bloc d'instruction dans la boucle, on peut omettre son nom :

Code Python :

```
for _ in range(10) :
    print ('bonjour',end=',')
```

Un autre exemple, où la variable est utilisée :

Code Python :

```
for k in range(1,11) :
    print (k**2)
```

```
1
4
9
16
25
36
49
64
81
100
```

Code Python :

```
def somme_des_premiers_entiers(n) :
    """prend en argument un entier positif n, et renvoie la somme des entiers de 0 à n"""
    s=0
    for k in range(n+1) :
        s=s+k
    return s
```

Exercice 6 Écrire une fonction `fact` (non-récursive) qui prend un entier positif `n` en argument, et renvoie $n!$. On pourra tester à l'aide de la commande `[fact(n) for n in range(10)]`

Rem | `factorial(n)` existe dans le module `math` de *Python*.

Exercice 7 On considère la suite (u_n) définie par $u_0 = 1$, et, pour tout $n \in \mathbb{N}$, $u_{n+1} = \cos(u_n)$. Écrire un programme qui demande $n \in \mathbb{N}$ à l'utilisateur, puis calcule et affiche u_n.

Exercice 8 Même question avec la suite définie par $v_1 = 1$ et pour tout $n \in \mathbb{N}$, $v_{n+1} = exp(v_n)/n$.

Exercice 9 Soit (u_n) la suite définie par $u_0 = 0$, $u_1 = 1$, et $\forall n \in \mathbb{N}, \quad u_{n+2} = u_{n+1} + u_n$.
Écrire une fonction `fibo(n)` qui renvoie u_n.

Il est tout à fait possible d'imbriquer des boucles, toujours en respectant l'indentation. Par exemple, pour calculer une valeur approchée de $\sum_{i=0}^{100} \sum_{j=0}^{i} \cos(i + j)$:

Code Python :

```
from math import cos

s=0
for i in range(101) :
    for j in range(i+1) :
        s=s+cos(i+j)
print(s)
```

Rem | Ne pas exagérer avec les imbrications de boucles : souvent, créer une fonction auxiliaire pour remplacer la boucle intérieure rendra le code plus clair.

5.1.2 À partir d'une liste

Si l'on souhaite itérer sur les éléments d'une liste, *Python* fournit une syntaxe très pratique :

> **for** *variable* **in** l :
> ␣ *bloc d'instructions*

Code Python :

```
def contient_0(l) :
    """teste si la liste passée en argument contient 0"""
    for k in l :
        if k==0 :
            return True
    return False
```

En utilisant cette syntaxe, on n'a plus accès aux indices des éléments dans la liste. Si l'on a besoin de cet indice, il faut revenir à la syntaxe précédente [6] :

Code Python :

```
def indice_0(l) :
    """renvoie le premier indice où 0 apparait dans l, -1 si la liste ne contient pas 0"""
    for k in range(len(l)) :
        if l[k]==0 :
            return k
    return -1
```

5.2 Boucle while

On utilise une *boucle while* lorsqu'on ne connait pas a priori le nombre d'itérations d'une série d'instructions à effectuer. L'exécution s'arrêtera lorsqu'une certaine condition sera vérifiée.

while *condition* :
␣ *bloc d'instructions*

Code Python :

```
n=int(input('Entrez un entier strictement positif'))
p=1
while p<=n :
    print(p)          #(1)
    p=2*p             #(2)
```

Exercice 10 Que se passe-t-il si l'on intervertit les lignes (1) et (2) dans le programme précédent ?

6. il existe également une syntaxe très spécifique à *Python* : `for i,v in enumerate(l) :` qui itère à la fois sur les indices (i) et sur les valeurs (v). Il est théoriquement conseillé de l'utiliser lorsqu'on a besoin des indices pour traiter les éléments de la liste. Pédagogiquement, je préfère utiliser `range`, en remplaçant les v par $l[i]$.

Code Python :

```
def pgcd_euclide(a,b) :
    """calcule le pgcd de deux entiers naturels a et b, avec a>=b"""
    while b !=0 :
        (a,b)=(b,a%b)
    return a
```

Exercice 11 On considère la suite (u_n) définie par $u_0 = 1$ et $u_{n+1} = \sin(u_n)$. On peut montrer que (u_n) est de limite nulle.
Écrire un programme qui demande $e \in \mathbb{R}^{+*}$ à l'utilisateur, et qui affiche n et u_n, où n est le plus petit entier tel que $u_n \leqslant e$.

Exercice 12 Sans utiliser les fonctions du module `math` ni de conversion `float` en `int`, écrire une fonction qui prend en argument un réel positif, et renvoie sa partie entière.

Rem

Toute boucle *for* peut se réécrire comme une boucle *while* :

Code Python :

```
for k in range(n) :
    #instructions
```

Code Python :

```
k=0
while k<n :
    #instructions
    k=k+1
```

C'est en général peu souhaitable. Lorsqu'une boucle *for* fait l'affaire, je conseille (pour des raisons de simplicité et de lisibilité) de toujours la préférer à une boucle *while*.

Dès que l'on utilise des boucles `while`, un problème de **terminaison** peut se poser.

Code Python :

```
t=0
while t>-1 :
    t=t+1
    print(t)
```

Et il n'est pas toujours trivial de savoir si une boucle termine ou pas [7]...

Rem

Code Python :

```
def teste_fermat(x,y,z,n) :
    """teste si y^n+z^n=x^n, où x,y,z et n sont quatre entiers naturels"""
    return y**n+z**n==x**n

def teste_tout_fermat(n) :
    """prend en argument un entier naturel n.
    Renvoie un triplet (x,y,z) d'entiers non-nuls tels
    que y^n+z^n=x^n si un tel triplet existe.
    Ne termine pas sinon."""
    x=1
    while True :
        for y in range(x) :
            for z in range(x) : #parcours des entiers y et z < x
                if teste_fermat(x,y,z,n) :
                    return (x,y,z)
        x+=1
```

Il faut dans ce cas interrompre de force l'exécution du programme, à l'aide du bouton prévu à cet effet (ou de Ctrl-C dans une console).

7. C'est même un exemple classique de propriété indécidable.

6 Lecture/Écriture dans un fichier texte (.txt)

(→ les sources dans le fichier ./lecture_ecriture.py)
Utiles lorsque l'on souhaite sauvegarder des valeurs d'une session de travail à l'autre, ou pour fournir des données textuelles lors d'un TP...

6.1 Lecture

On utilisera le fichier texte suivant dans les exemples de cette section.

fichier.txt

```
Ceci est un fichier test.
Numéro 1  : 1234 5678

Numéro 2  : 1342 5434 5960
```

Le programme suivant ouvre le fichier, et lit et affiche ses 3 premières lignes.

Code Python :

```
f=open('chemin/fichier.txt','r')  #ouverture en lecture (r pour read)

print (f.readline())   #lecture et affichage d'une ligne
print (f.readline())
print (f.readline())

f.close()   #fermeture du fichier
```

```
Ceci est un fichier test.

Numéro 1  : 1234 5678
```

Rem Par défaut, les fichiers sont recherchés dans le répertoire de travail courant (que l'on peut trouver à partir des commandes `import os ; print(os.getcwd())`). Si l'on veut accéder (en lecture ou en écriture) à un fichier qui est ailleurs, il faut préciser le chemin complet.

Le programme suivant ouvre le fichier, lit toutes ses lignes, et calcule son nombre de caractères.

Code Python :

```
f=open('fichier.txt','r')

s=0
for l in f :
    s=s+len(l)

print('Le fichier a',s,'caractères')

f.close()
```

```
Le fichier a 75 caractères
```

Profitons-en pour voir quelques commandes permettant de traiter des données textuelles. Le programme suivant ouvre le fichier, cherche toutes les lignes commençant par «Numéro», et affiche alors le nombre de nombres suivant «Numéro : » ainsi que leur somme.

Code Python :

```
f=open('fichier.txt','r')

for l in f :
    if l.find('Numéro')==0 :  #La ligne commence par 'Numéro'
        [a,b]=l.split(" :") # on découpe selon les deux points
        liste_num=b.split() # on découpe selon les espaces
        sum=0
        print('Il y a',len(liste_num),'nombres. ', end='')
              #end='' permet de ne pas revenir à la ligne.
              #La valeur par défaut est '\n',
              #qui est le caractère de retour à la ligne.
        for s in liste_num :
            sum+=int(s)
        print('Leur somme est',sum)

f.close()
```

```
Il y a 2 nombres. Leur somme est 6912
Il y a 3 nombres. Leur somme est 12736
```

6.2 Écriture

Code Python :

```
f=open('fichier2.txt','w')
# Attention, l'option 'w' («write») écrase le fichier s'il existe déjà...
# Pour rajouter des lignes à un fichier existant, utiliser 'a' («append»).

f.write('J\'écris une première ligne dans le fichier. ')
f.write('Nous sommes toujours dans la première ligne.\n')  #\n  : retour à la ligne
f.write('Maintenant, nous sommes passés à la 2ème...\n\n')
f.write('Entier\tCarré\n')  #\t  : tabulation
for k in range(10) :
    f.write(str(k)+'\t'+str(k**2)+'\n')

f.close()
```

fichier.txt

```
J'écris une première ligne dans le fichier. Nous sommes toujours dans la première ligne.
Maintenant, nous sommes passés à la 2ème...

Entier Carré
0      0
1      1
2      4
3      9
4      16
5      25
6      36
7      49
8      64
9      81
```

7 Graphiques

(→ les sources dans le fichier ./graphiques.py)
Le principal module permettant de faire des graphiques est *matplotlib.pyplot*. Il est très puissant, mais de prise en main peu aisée.
Je vais simplement donner quelques exemples dans cette section.
Vous pourrez trouver des tutoriaux plus complets, respectivement en français et an anglais, aux adresses suivantes :
http://www.courspython.com/introduction-courbes.html
https://www.labri.fr/perso/nrougier/teaching/matplotlib/

Deux conseils :

- ne pas hésiter à procéder par une approche «copier/coller des exemples trouvés sur internet (ou sur ce poly), les comprendre et les modifier selon ses besoins» plutôt que de tout recoder depuis le début ;
- pour les graphiques, animations, et graphiques interactifs, le langage de calcul formel *Sage* (basé sur *Python*) est une alternative sérieuse à matplotlib, de prise en main beaucoup plus simple.

7.1 Bases sur la commande plot

On importe le module *pyplot* à l'aide de la commande

```
import matplotlib.pyplot as plt
```

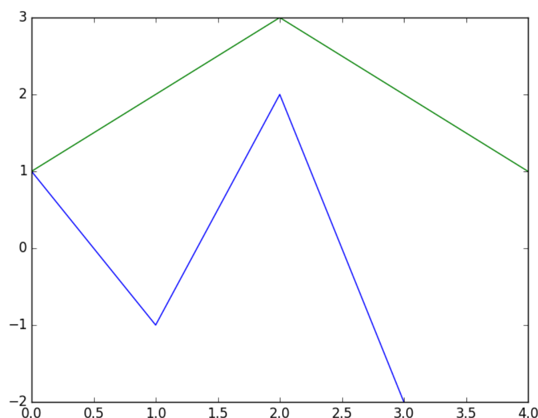
La commande à tout faire pour les tracés est `plt.plot(x,y,options)`
où

- x est la liste des abscisses des points à tracer.
- y est la liste des ordonnées des points à tracer.
- Nous reviendrons sur les (très nombreuses) options dans les exemples qui suivent.

Il faut la faire suivre de `plt.show()` pour effectivement ouvrir la fenêtre graphique. `plt.show()` est une commande bloquante : il faut fermer la fenêtre graphique avant de pouvoir reprendre la main sur la console.

Code Python :

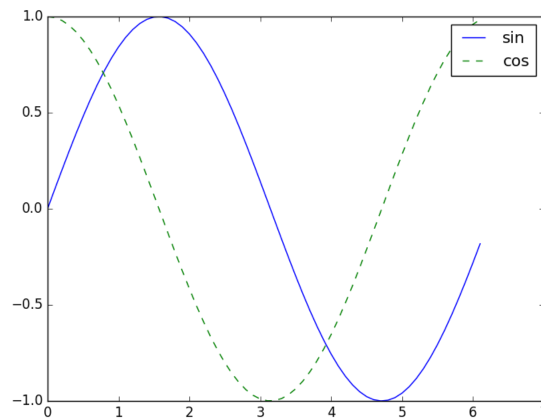
```
import matplotlib.pyplot as plt

plt.plot([0,1,2,3],[1,-1,2,-2])
plt.plot([0,2,4],[1,3,1])
plt.show()
```



Pour tracer une fonction, l'idée est de prendre des abscisses suffisamment rapprochées pour donner l'illusion d'une courbe.

Code Python :

```
# abscisses : un échantillonnage au 1/10 de [0,2pi]
x=[k/10 for k in range(0,int(2*pi*10))]
# ordonnées correspondantes pour sin et cos
y=[sin(k) for k in x]
z=[cos(k) for k in x]

# tracés, avec une légende
plt.plot(x,y,label='sin')
plt.plot(x,z,'--',label='cos')
plt.legend()

plt.show()
```



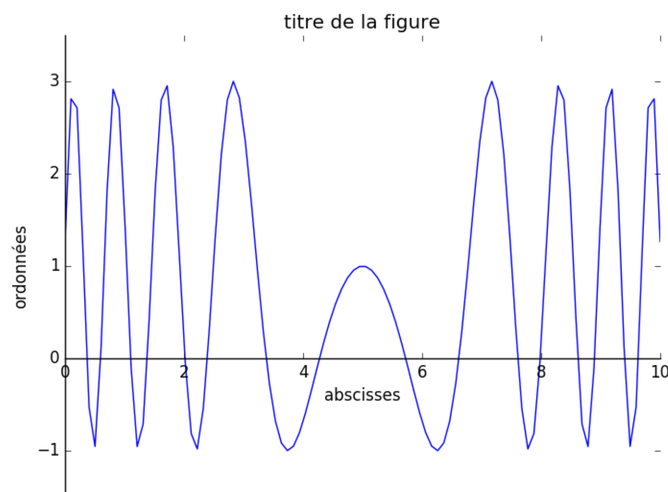
C'est assez peu agréable. Comme nous allons le voir dans l'exemple suivant, le module *numpy* permet de simplifier les choses.

7.2 Exemple : tracé d'une fonction avec *numpy*

Tracé du graphe de la fonction $f : \left\{ \begin{array}{ccc} [0,10] & \to & \mathbb{R} \\ x & \mapsto & 1-2\sin((x-5)^2) \end{array} \right.$.

Code Python :

```
import matplotlib.pyplot as plt
import numpy as np

#tableau numpy (array) constitué de 100 réels échantillonant [0,10]
x=np.linspace(0,10,100)

#image des points de x par x->1-2sin((x-5)**2)
#à noter que les tableaux numpy supportent les opérations mathématiques
y=1-2*np.sin((x-5)**2)

#tracé
plt.plot(x,y)

#limites horizontales et verticales de la figure
plt.xlim(0, 10)
plt.ylim(-1.5, 3.5)

#affichage plus "classique" des axes
axes=plt.gca() #récupération des axes de la figure
axes.spines['top'].set_color('none') #suppression du côté supérieur de la boite
axes.spines['right'].set_color('none') #suppression du côté droit
axes.spines['bottom'].set_position(('data',0)) #translation du côté inférieur

#légende des axes et de la figure
plt.xlabel('abscisses')
plt.ylabel('ordonnées')
plt.title('titre de la figure')

#sauvegarde dans un fichier
plt.savefig('/home/utilisateur/AP/stagePython/plot1.png')

#affichage
plt.show()
```



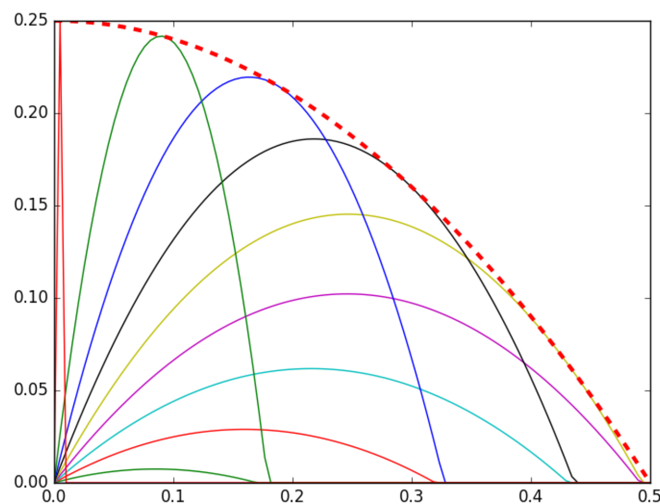
Noter les points anguleux vers les extrémités de l'intervalle (et les maxima et minima incorrects), liés à l'échantillonnage insuffisant.

7.3 Exemple : tracé d'une famille de courbes

Tracé de la famille des trajectoires paraboliques d'un objet lancé avec un angle variable depuis l'origine (la norme de la vitesse initiale étant constante), et «courbe de sûreté» correspondante.
Les trajectoires sont d'équations $y = -(1 + \tan^2(a))x^2 + \tan(a)x$, où $a \in [0, \pi/2[$ est l'angle de départ, et la courbe de sûreté d'équation $y = 1/4 - x^2$.

Code Python :

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi

#tableau numpy (array) constitué de 100 réels échantillonnant [0,10]
x=np.linspace(0,.5,100)
tan_angles=np.tan(np.linspace(0,pi/2-.01,10))

#tracé des différentes trajectoires selon l'angle de départ
for a in tan_angles :
    y=-x**2*(1+a**2)+x*a
    #mise à 0 des altitudes négatives
    for k in range(len(y)) :
        if y[k]<0 :
            y[k]=0
    plt.plot(x,y)

#tracé de la parabole de sureté - rouge, épais, et pointillé
y=0.25-x**2
plt.plot(x,y, color ='red', linewidth=3, linestyle="--")

#sauvegarde dans un fichier
plt.savefig('/home/utilisateur/AP/stagePython/parabolesurete.png')

#affichage
plt.show()
```



7.4 Exemple : un graphique interactif

Le même exemple, avec en haut un curseur permettant de faire varier l'angle.

Code Python :

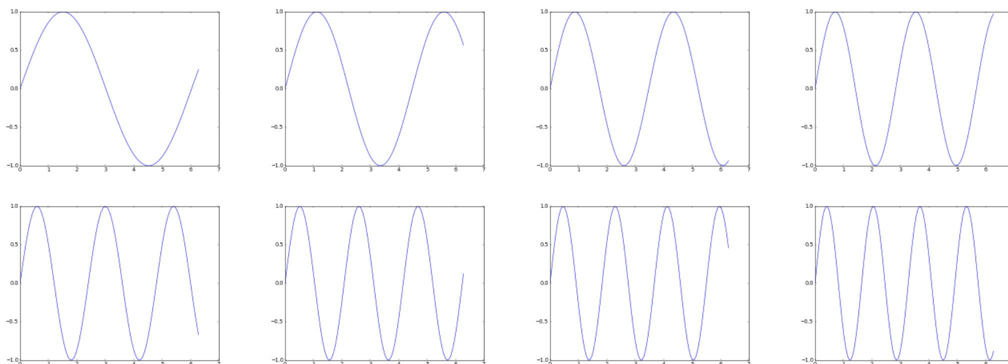
```
import matplotlib.pyplot as plt
import numpy as np
from math import pi
import pylab

#récupération des objets «figure» et «axes» de pyplot
fig, ax = plt.subplots()

#min et max sur les 2 axes
ax.set_xlim(0, .5) ;
ax.set_ylim(0, .25) ;

x=np.linspace(0,.5,100)

#tracé de la fonction nulle, et «affectation du tracé» dans la variable courbe
courbe, = ax.plot(x, 0*x)

#tracé de la parabole de sureté
y=0.25-x**2
ax.plot(x,y, color ='red', linewidth=3, linestyle="--")


#tracé d'une trajectoire selon l'angle de départ
def update(val) :
    angle = slider_angle.val
    y=-x**2*(1+tan(angle)**2)+x*tan(angle)
    #mise à 0 des altitudes négatives
    for k in range(len(y)) :
        if y[k]<0 :
            y[k]=0
    #mise à jour de la courbe
    courbe.set_ydata(y)

#mise en place du «slider» - curseur permettant de modifier le paramètre d'angle
position_slider = pylab.axes([0, 0.94, 0.6, 0.025])
slider_angle = pylab.Slider(position_slider, 'angle', 0, pi/2, valinit=0)
slider_angle.on_changed(update)

plt.show()
```

7.5 Exemple : une animation

Tracé animé des graphes de $x \mapsto \sin(\theta x)$, pour θ variant entre 1 et 5.

Code Python :

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation
from math import pi

#récupération des objets «figure» et «axes» de pyplot
fig, ax = plt.subplots()

x = np.linspace(0, 2*pi, 100)
#tracé, et «affectation du tracé» dans la variable line
line, = ax.plot(x, np.sin(x))

#fonction d'animation  : indique comment modifier le tracé en fonction de
#theta
def animation_step(theta) :
    line.set_ydata(np.sin(x*theta))
    return line,

#mise en place de l'animation
  #le np.linspace(1,5,100) décrit les valeurs prises par le theta de animate
  #interval=55 donne l'intervalle de temps (minimal, en ms) pour passer d'un theta
  #au suivant
anim = animation.FuncAnimation(fig, animation_step, np.linspace(1,5,100),
                              interval=55)

# sauvegarde de l'animation dans un gif animé, que l'on pourra
# par exemple mettre sur une page web.
anim.save('/home/utilisateur/AP/stagePython/animation.gif',
           writer='imagemagick', fps=30, dpi=40)

#affichage
plt.show()
```



8 Deux thèmes mathématiques

8.1 Suites

→ les sources dans le fichier `./suites.py`
→ les corrections dans le fichier `./suites_corrections.py`

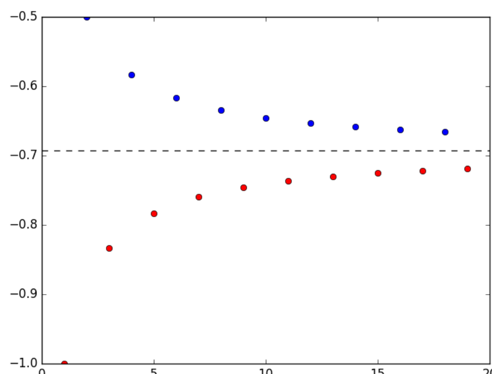
8.1.1 Une comparaison puissance/exponentielle

On considère la suite (u_n) définies par
$\forall n \in \mathbb{N},\ u_n = \dfrac{n^3}{3^n}$.

1. Écrire une fonction `u(n)` qui prend en argument un entier naturel `n`, et renvoie une valeur approchée de u_n.
2. On souhaite conjecturer le comportement de cette suite.
   (a) Afficher les 20 premiers termes de cette suite.
   (b) On souhaite maintenant obtenir un tracé des premiers termes.
   On va pour cela utiliser la commande `plt.plot(x,y,'o',color='r')` suivie de `plt.show()`, où
   - `x` est la liste des abscisses des points à tracer.
   - `y` est la liste des ordonnées des points à tracer.
   - `'o'` est une option de plot qui signifie que l'on veut tracer des points (et non pas une courbe).
   - `color='r'` indique la couleur des points (ici rouge) [8] [9].
3. Écrire une fonction `premier_terme_inferieur(e)` qui prend en argument un réel $e > 0$, et renvoie le plus petit indice $n \in \mathbb{N}^*$ tel que $u_n < e$.

8.1.2 Une série alternée

1. Écrire une fonction `v(n)` qui prend en argument un entier $n \in \mathbb{N}^*$, et renvoie (à l'aide d'une boucle [10]) une valeur approchée de v_n.
2. On souhaite conjecturer le comportement de cette suite.
   (a) Conjecturer sa limite en regardant $\exp(v_n)$.
   (b) En vous inspirant de l'exercice sur (u_n), tracer les premiers termes de cette suite. On souhaiterait cette fois-ci afficher les termes d'indice pair en bleu, et ceux d'indice impair en rouge (on rappelle que les extractions de tranches permettent d'extraires les termes pairs ou impairs d'une liste). Y rajouter la droite d'équation $y = -\ln(2)$, en noir et en pointillés (option `'--'` au lieu de `o` dans le plot).



8. On aurait pu compacter les deux dernières options en `'ro'`
9. Les couleurs de base de *matplotlib* sont en général codées par la première lettre de leur nom anglais : 'b' : bleu, 'g' : vert (green), 'r' : rouge, 'c' : bleu ciel (cyan), 'm' : magenta 'y' : jaune (yellow), 'k' : noir (blacK), 'w' : blanc (white)
10. On pourrait également écrire cela en une ligne : `sum([(-1)**k/k for k in range(1,n+1)])`. Pédagogiquement, je préfère faire travailler avec des boucles plutôt que d'utiliser des «spécificités trivialisantes» de *Python*. . .

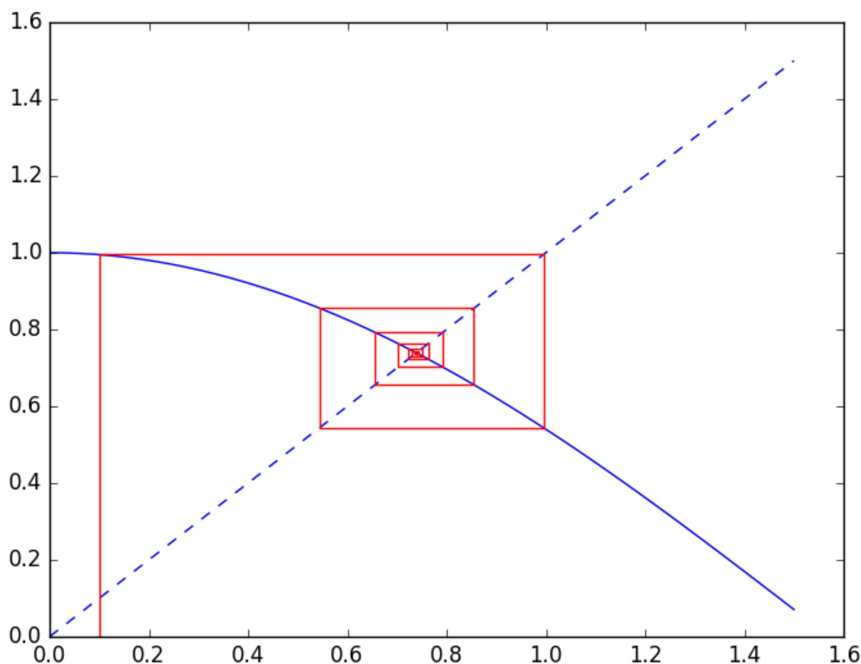
8.1.3 Représentation graphique et animation de suites récurrentes

On souhaite représenter graphiquement des suites récurrentes.
Voici un exemple de résultat attendu pour la suite (u_n) définie par

$$\begin{cases} u_0 = 1/10 \\ \forall n \in \mathbb{N},\ u_{n+1} = f(u_n) \end{cases}$$

où

$$f = \cos$$



On va s'intéresser plus particulièrement au tracé de la ligne brisée. Celle-ci part de $(u_0, 0)$, se projette verticalement sur le graphe $\mathcal{C}_f$ de f, puis horizontalement sur la droite $\mathcal{D}$ d'équation $y = x$, puis verticalement sur $\mathcal{C}_f$, etc.

1. Écrire une fonction `ligne_brisee(f,u0,n)` qui prend en argument :
   - une fonction `f`
   - le premier terme de la suite `u0`
   - un entier `n`

   et qui renvoie un couple de listes `(lx,ly)` où `lx` et `ly` sont les listes des abscisses et des ordonnées des points de la ligne brisée, après n pas de «projection verticale sur $\mathcal{C}_f$/projection horizontale sur $\mathcal{D}$». (Ce seront donc des listes de longueurs $2n+1$, qui commenceront respectivement par `lx=[u,u,f(u),...` et `ly=[0,f(u),f(u),...`).
2. Regarder et utiliser la fonction `trace` fournie pour voir quelques exemples (on pourra prendre pour f des fonctions du module mathématique, on des fonctions que l'on définira soi-même).
3. On s'intéresse maintenant au cas particulier où $f : x \mapsto ax(1-x)$, où a est un paramètre appartenant à $[1, 4]$. La suite correspondante est appelée *suite logistique*[11], et montre des comportement asymptotiques très différents selon la valeur de a.

   Exécuter en bloc la cellule intitulée «`##suite logistique, animation`» pour voir une animation, où le paramètre a varie.

   Se plonger rapidement dans le code, et le modifier pour «zoomer» sur la plage de paramètre $a \in [3, 3.7]$ où se fait la transition entre un point attracteur et un comportement chaotique (en passant par des cycles attracteurs).

11. https://fr.wikipedia.org/wiki/Suite_logistique

8.1.4 Les sujets auxquels vous avez échappé

Méfiez-vous des machines...

On considère, pour tout $n \in \mathbb{N}$, $I_n = \displaystyle\int_0^1 t^n e^{t^2} dt$.

On calcule $I_1 = \dfrac{1}{2}(e-1)$, et une intégration par parties montre que pour tout $n \geqslant 1$, $I_{n+2} = \dfrac{1}{2}e - \dfrac{n+1}{2} I_n$.
On peut alors utiliser *Python* pour calculer les premiers termes de la suite. Pousser jusqu'à plus de 50 pour voir apparaitre un comportement étrange.

La suite de Syracuse

C'est la suite définie par $u_0 \in \mathbb{N}$ et pour tout $n \in \mathbb{N}$,

$$u_{n+1} = \begin{cases} n/2 \text{ si } n \text{ est pair} \\ 3n+1 \text{ si } n \text{ est pair} \end{cases}$$

https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse
Savoir si cette suite atteint 1 quelquesoit la valeur de u_0 est une conjecture ouverte à l'heure actuelle.
Beaucoup de choses à faire avec des boucles, des graphiques, des listes : on peut vérifier la conjecture de Syracuse pour de petites valeurs initiales, tracer les points de la suite, calculer l'altitude, le temps de vol, etc.

Graphique interactif

En vous inspirant de la section sur les graphiques interactif, créer un graphique interactif représentant une suite géométrique (ou arithmétique, arithmético-géométrique), avec un curseur pour faire varier le premier terme, et un curseur pour faire varier la raison.

8.2 Probabilités

→ les sources dans le fichier ./probas.py.
→ les corrections dans le fichier ./probas_corrections.py.

8.2.1 Le module *numpy.random*

On l'importe une fois pour toute à l'aide de la commande `from numpy.random import *`.

Voici les 2 fonctions que nous allons utiliser dans la suite (les tester !) :

- `random()` : renvoie un réel dans $[0,1[$ selon la loi uniforme (où – on l'espère – une bonne approximation de cette loi sur les flottants).
- `randint(a,b)`, où a et b sont 2 entiers tels que $a \leqslant b$: renvoie un entier de $[\![a,b[\![$, selon la loi uniforme.

Le module permet par ailleurs de simuler les lois usuelles :

- `geometric(p)` : loi geometrique.
- `binomial(n,p)` : loi binomiale.
- `normal(mu, sigma)` : loi normale.
- `uniform(a, b)` : loi uniforme sur $[a,b]$.
- `expovariate(lambda)` : loi exponentielle.

On peut facilement l'utiliser pour simuler des expérience aléatoires.
Considérons par exemple une urne qui contient initialement 1 boule noire et 1 boule rouge. On effectue des tirages successifs avec sur-remise : lorsqu'une boule est tirée, on la remet dans l'urne et l'on rajoute une boule de même couleur. Voici un programme simulant cette expérience :

Code Python :

```
def tirage_surremise(n) :
    """prend en argument un entier n>0, et simule n tirages avec sur-remise"""
    nb_n=1 #nombre de boules noires
    nb_r=1 #nombre de boules rouges
    for k in range(n) :
        if rand()<nb_n/(nb_n+nb_r) : #on a tiré une boule noire
            print('Tirage',k,' :boule noire.')
            nb_n+=1
        else :
            print('Tirage',k,' :boule rouge.')
            nb_r+=1
        print(' Il y a maintenant',nb_r,'boule(s) rouge(s) et',nb_n,'boule(s) noire(s)')
```

1. Le modifier pour qu'il s'arrête lorsque l'on a atteint 5 boules rouges dans l'urne (la fonction n'affichera plus rien, mais renverra alors le nombre de tirages effectués). En faisant 100000 expériences, évaluer empiriquement l'espérance du nombre de tirages nécessaires pour atteindre ces 5 boules rouges.
2. Même question, sauf que l'on souhaite s'arrêter lorsque l'on a tiré 3 boules rouges d'affilée ! Lancer quelques tests.

8.2.2 Simulation d'une loi binomiale

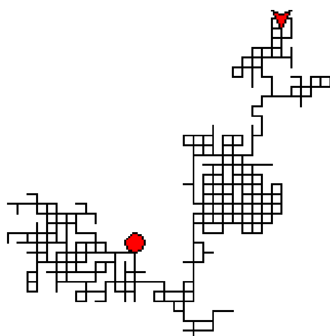
1. En utilisant simplement `random()`, écrire une fonction `binom(n,p)` qui simule une loi binomiale de paramètres n et p [12].
2. Exécuter des commandes du type `tracebinom(30,.5,10000)` pour afficher l'histogramme de répartition obtenu pour 10000 expériences suivant une loi binomiale de paramètres 30 et 0,5. Sur le graphique s'affiche la gaussienne «théorique» correspondante.

12. Indication : `random()<p` renvoie `True` avec probabilité p.

8.2.3 Marche aléatoire 1D

1. Écrire une fonction `marche1d(n)` qui prend en argument un entier n, et effectue une marche aléatoire de la façon suivante : on part de n, et à chaque pas, on augmente ou diminue de 1 avec probabilité 1/2, jusqu'à ce que l'on atteigne 0. La fonction renverra la *liste* des valeurs obtenues [13].
2. Écrire une fonction `tracemarche1d(l)`, qui prend en argument une sortie de `marche1d`, puis trace le résultat obtenu.

8.2.4 Tortue et marche aléatoire 2D



Le module *turtle* implémente une «tortue» : il s'agit d'un module graphique basé sur un curseur qui peut avancer, tourner, tout en traçant sa trajectoire (*Scratch* est essentiellement une tortue).
On l'importe à l'aide de `import turtle ;`. Les commandes principales de la tortue sont alors :

- `turtle.right(d)` : tourne la tortue vers la droite d'un angle de d degrés.
- `turtle.left(d)` : tourne la tortue vers la gauche d'un angle de d degrés.
- `turtle.forward(l)` : avance la tortue de l pixels.
- `turtle.goto(x,y)` : la tortue se translate jusqu'à la position (x, y) en ligne droite.

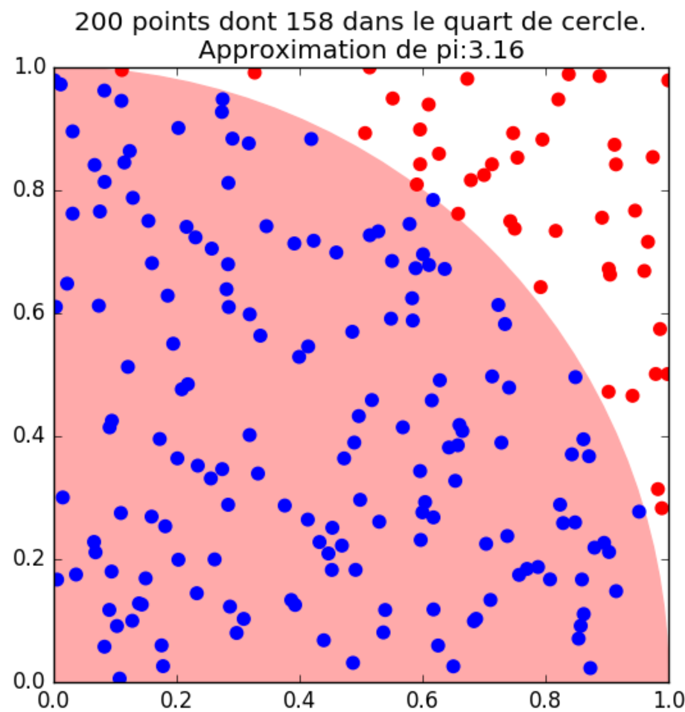
Vous trouverez d'autre commandes expliquées au début de la fonction `marche2d`.

1. Compléter la fonction `marche2d(n)` qui effectue une marche aléatoire en 2D de n pas. À chaque pas, la tortue sera déplacée de façon équiprobable dans l'une des 4 directions haut/bas/gauche/droite.
2. Même question, mais l'on veut à chaque pas une direction aléatoire quelconque, non-contrainte aux 2 axes.

8.2.5 Approximation de π par une méthode de Monte-Carlo

On souhaite approcher π par la méthode suivante : on tire au hasard et de façon uniforme des points dans $[0, 1]^2$. La proportion de points qui tombent dans le quart de cercle centré en 0 et de rayon 1 est une approximation de $\frac{\pi}{4}$.

13. Pour ajouter un élément à une liste `l`, on pourra utiliser `l=l+[2]`, ou `l.append(2)`.



1. Écrire une fonction `pi_montecarlo(n)` qui prend en argument un entier $n \in \mathbb{N}^*$, tire successivement n points. Après chaque tirage, elle calculera une approximation de π avec les points déjà tirés, et renverra la liste des approximations ainsi obtenues.
2. Tester la fonction `trace_pi_montecarlo`.

NB : la fonction `schema_pi_montecarlo` présente dans le fichier est celle qui a permis d'obtenir la figure ci-dessus.

8.2.6 Les sujets auxquels vous avez échappé

Approximation de pi par la méthode de l'aiguille de Buffon
https://fr.wikipedia.org/wiki/Aiguille_de_Buffon
On lance une aiguille de longueur l sur un parquet dont les lattes sont de largeur l. On peut montrer que la probabilité que l'aiguille tombe à cheval sur 2 lattes est $2/\pi$. On peut ainsi calculer une approximation de π en lançant un grand nombre d'aiguilles.
On peut en profiter pour s'entraîner à faire le dessin correspondant à l'aide de *matplotlib*.

Une expérience en 2 étapes
On lance un premier dé à 6 faces, puis autant de dés que le résultat du premier dé. On s'intéresse à la somme des dés ainsi lancés.
Simuler cette expérience, puis tracer un histogramme empirique de la loi du résultat obtenu.

Étude empirique des probabilités de figure au Yams
https://fr.wikipedia.org/wiki/Yahtzee
On lance 5 dés à 6 faces.

1. Écrire une fonction qui simule cette expérience, et renvoie le résultat sous forme de liste d'entiers.
2. Écrire des fonctions `est_carre(l)`, `est_full(l)`, etc, qui testent si une liste correspond à telle ou telle figure au Yams.
3. En déduire des estimations empiriques des probabilités d'obtenir (du premir coup) les différentes figures.

A Très courte bibliographie

Beaucoup d'informations et tutoriaux sur les langages se trouvent sur le web. Il n'est plus indispensable à l'heure actuelle d'avoir chez soi un ouvrage de référence sur tel ou tel langage ou tel ou tel module.
Voici cependant deux livres qui peuvent particulièrement vous intéresser sur *Python* et les maths, que vous trouverez au CDI :

- Casamayou-Boucau, Chauvin, Connan, *Programmation en* Python *pour les mathématiques*, Dunod, 2016
  (Reprend les bases du langages, et propose de nombreux exemples de thèmes mathématiques)
- Wack et al, *Informatique pour tous en classes préparatoires au grandes écoles*, Eyrolles, 2013
  (Le programme que nous traitons en CPGE – dans le cadre de l'*informatique pour tous* (IPT) [14] : les bases du langage, la représentation des nombres en machine, et des thèmes d'ingénierie numérique.)

14. Un «tronc commun» d'informatique, enseigné en MPSI, PCSI, PTST, et dans les classes de seconde année correspondantes. Il y a également une *option informatique*, exclusivement en MPSI/MP, qui est beaucoup plus orientée vers l'informatique théorique. La partie programmation est faite en *CamL*, un langage fonctionnel plus abstrait que *Python.*