Calcul Scientifique avec Python

Licence 2 Spéciale

Christophe Besse

Université Paul Sabatier Toulouse 3 – 2015

1 Le calcul scientifique

Le calcul scientifique consiste en l'utilisation d'un outil de calcul (calculatrice, ordinateur) et le développement de méthodes numériques pour la résolution de problèmes qu'il est difficile de résoudre analytiquement. Deux difficultés apparaissent dans cette définition : utilisation d'un outil de calcul et développement de méthodes numériques. La première nécessite de connaître le langage de programmation de l'outil. Nous avons fait le choix dans ce cours du langage Python qui sera largement développé par la suite. La seconde fait référence à la construction de schémas numériques. Donnons pour cela un exemple.

On considère une population d'individus initiale $P(t = 0)$ et on souhaite examiner son évolution. Malthus (fin du 18ème siècle) nous donne un premier modèle

$$P'(t) = rP(t), \quad t > 0.$$

$P(t)$ est la taille de la population et r son taux de croissance. Évidemment, la solution est

$$P(t) = e^{rt} P_0$$

et on en déduit donc que si $r = 0$, la population reste constante, si $r < 0$, la population tend à s'éteindre de manière exponentielle et si $r > 0$, la population croit de manière exponentielle. Les résultats sont peu réalistes. Verhulst (1838) propose alors un nouveau modèle où la population ne peut pas tendre vers l'infini. C'est le modèle à croissance logistique

$$P'(t) = rP\left(1 - \frac{P}{K}\right)$$

avec r le taux de croissance de la population et K la capacité d'accueil du milieu, c'est à dire le nombre maximal d'individus que le milieu peut accueillir en tenant compte de l'espace, des ressources, etc... Il est beaucoup plus difficile de trouver une solution analytique à ce modèle car l'équation est non linéaire. Ceci reste cependant possible.

$$\int \frac{dP}{P(r - rP/K)} = \int dt$$

$$\frac{1}{P(r - rP/K)} = \frac{A}{P} + \frac{B}{r - rP/K}$$

$$1 = Ar - ArP/K + PB$$

Ceci conduit immédiatement à

$$A = 1/r \quad \text{et} \quad B = Ar/K = 1/K$$

On a donc

$$\int \frac{dP}{P(r - rP/K)} = \int \frac{dP}{rP} + \frac{dP}{r(K - P)} = \int dt$$

$$\frac{1}{r}\ln|P| - \frac{1}{r}\ln|K - P| = t + d$$

$$\ln|P| - \ln|K - P| = rt + rd$$

$$\ln\frac{|P|}{|K - P|} = rt + d$$

Ainsi, on a

$$P(t) = \frac{KCe^{rt}}{1 + Ce^{rt}}, \quad C = e^d.$$

En règle général, l'obtention d'une formule analytique est impossible pour les équations différentielles non linéaires. Il faut alors passer par une résolution numérique : c'est l'objet du calcul scientifique. Cherchons à proposer un algorithme pour étudier son évolution temporelle de l'équation logistique. Elle se présente sous la forme

$$P'(t) = f(t, P(t))$$

qu'on peut réécrire

$$P(t+h) = P(t) + \int_t^{t+h} f(s, P(s))ds.$$

On est ramené au calcul d'une intégrale. On sait, si h n'est pas trop grand qu'on peut utiliser la formule des rectangles à gauche pour approcher cette intégrale. On obtient alors une approximation

$$P(t+h) \approx P(t) + hf(t, P(t)).$$

Si on utilise cette formule pour tous les temps, on voit qu'on peut fournir une approximation de la solution de l'équation différentielle ordinaire. Notons $t_n = nh$. On a ainsi la suite

$$\begin{array}{ll} t_0 & P_0 = P(t=0), \\ t_1 = h & P_1 = P_0 + hf(0, P_0), \\ t_2 = h & P_2 = P_1 + hf(t_1, P_1), \\ \vdots & \vdots \\ t_{n+1} = h & P_{n+1} = P_n + hf(t_n, P_n). \end{array}$$

On peut donc construire un algorithme

```
Donner P=P(t=0) et h
Pour j=1 a n+1:
  P=P+h*f((j-1)*h,P)

Afficher la valeur finale
```

On peut reproduire les exemples de ce type.

L'idée de ce cours est donc d'étudier un langage de programmation qui permette de mettre en œuvre des algorithmes pour résoudre des problèmes issus de la physique, de la chimie, de la biologie, des mathématiques, de la finance, etc ... dont la résolution analytique est difficile. Nous présentons donc le langage Python et donnerons quelques algorithmes standard pour

- calculer des intégrales numériques
- interpoler une fonction
- résoudre des systèmes linéaires
- résoudre des équations non linéaires
- résoudre des équations différentielles ordinaires
- ...

2 Documentation Python

Voir la bibliographie en fin de document. Pour une référence complète, on pourra consulter `http://docs.python.org/2/`.

3 Introduction

3.1 Caractéristiques de Python

Python est un langage récent (1989) très utilisé dans la programmation WEB (accès aux bases de données, programmation objet), comme langage de scripts (manipulation de fichiers, administration de systèmes, configuration de machines), le calcul scientifique (librairies mathématiques).

Python est un langage de programmation très polyvalent et modulaire, qui est utilisé aussi bien pour écrire des applications comme YouTube, que pour traiter des données scientifiques.

Python est un langage fourni avec peu de fonctions de base mais que l'on enrichit avec des bibliothèques.

C'est un langage interprété, puissant, compact, visuel

Il est multiparadigmes, supportant les principaux styles de programmation: impératif, procédural, orienté objet ...

Il est Multiplaterformes : GNU/Linux, MacOSX, Windows...

Il est très polyvalent, grâce à de nombreux modules couvrant des domaines très variés :

numpy algèbre linéaire, matrices, vecteurs, systèmes linéaires...

scipy probalilité/statistiques, FFT 1D, 2D..., filtrage numérique, images

matplotlib tracé de courbes (look & feel MatLab)

os manipulation des répertoires et des fichiers...

C'est un langage libre et gratuit : logiciel OpenSource (www.opensource.org) distribué sous la licence PSF (Python Software Foundation) compatible avec la GPL (Gnu Public Licence).

Il est simple à prendre en main, de plus en plus utilisé en recherche, enseignement, industrie ...

Enseigné au lycée (programme 2009, classe de seconde), au programme des classes prépas (rentrée 2013)

Utilisé par les acteurs majeurs du monde industriel : NASA, Google, CEA, Youtube, ...

À quoi peut servir Python?

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir.

Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

- de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie...
- des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- créer des interfaces graphiques ;
- faire circuler des informations au travers d'un réseau ;
- dialoguer d'une façon avancée avec votre système d'exploitation ;
- ... et j'en passe...

Un langage interprété

Python est un langage de programmation interprété, c'est-à-dire que les instructions que vous lui envoyez sont "transcrites" en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés "langages compilés" car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la "compilation". À chaque modification du code, il faut rappeler une étape de compilation.

Python est un langage interprété, ce qui peut vous faire gagner un temps considérable lors de l'élaboration du programme, car aucune compilation et édition de liens ne sont nécessaires. L'interprète peut être utilisé de manière interactive, ce qui le rend facile d'expérimenter avec des caractéristiques de la langue, d'écrire à jeter des programmes, ou de tester les fonctions pendant bottom-up élaboration du programme. Il est également une calculatrice de bureau à portée de main.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment. En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations.

Python permet aux programmes d'être écrite de façon compacte et lisible. Les programmes écrits en Python sont généralement beaucoup plus courte que équivalent C, C + +, ou des programmes Java, pour plusieurs raisons:

- les types de données de haut niveau vous permettent d'exprimer des opérations complexes en une seule instruction;
- le regroupement des instructions se fait par indentation au lieu de début et de fin parenthèses;
- aucune déclaration de variable n'est nécessaire.

Les principales caractéristiques du langage sont les suivantes:

- langage impératif: un programme est une suite d'instructions (conditionnelles, répétées) réalisant des accès mémoire (lecture, écriture de variables), fichier, ...

```
x = 0
s = 0
while x < 10:
  s = s + x
  x = x + 1

print s
```

- langage objet: un objet a un état et offre des opérations de consultation, modification de l'état (un point a une position, peut être déplacé)

```
p.x = 1
p.y = 2
p.avancer (3,4)
```

- langage de classes: une classe décrit les opérations dont disposent un ensemble d'objets similaires (ex la classe des points, des vecteurs, des matrices, ...).

```
class Point:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def avancer(self,dx,dy):
    self.x += dx
    self.y += dy

p = Point(1,2)
p.avancer(3,4)
```

- langage d'ordre supérieur: passage de fonctions en paramètre, retour de fonctions

```
def composition(f,g):
  return lambda(x): f(g(x))

r = composition(lambda(x):x+1, lambda(x): 2 * x)
r(3)
# --> 7
```

- Python permet la programation fonctionnelle (définition de fonctions éventuellement récursives, composition de fonctions, pas d'effets de bord)

```
def factorielle(n):
  if n == 0:
    return 1
  else:
    return n*factorielle(n-1)

composition(factorielle, lambda x: 2*x)(5)
# --> 3628800
```

- langage à typage dynamique : le type d'une variable est celui de la valeur qui lui est affectée. Elle peut donc changer de type en cours d'exécution ou dépendre de la branche suivie:

```
x = 1
x = 'abc'

if ... :
  x = 1
else:
  x = 'abc'

x = x + 3   # erreur de type possible
```

  Remarque: il existe des outils d'analyse statique pouvant détecter certaines erreurs avant l'exécution (pylint).

- gestion automatique de la mémoire: ie système récupère automatiquement la mémoire inutilisée (malloc implicite, free inutile). Il n'y a pas de pointeurs.

```
x = 'abc'
x = 1  # la mémoire nécessaire pour stocker 'abc' est récupérée
```

- Python est un langage interpété : boucle d'interprétation qui lit une commande et l'exécute immédiatement. Il n'y a pas de phase de compilation.

3.2 Installation de Python

Tout dépend du système d'exploitation. Il existe cependant une distribution commune simple d'utilisation : Canopy d'Enthought.

Linux sudo apt-get install python2.7 python-numpy python-scipy python-matplotlib python-qt4 idle spyder

Windows Installer la distribution "Python scientifique" Python(x,y), téléchargeable sur : `http://code.google.com/p/pythonxy/wiki/Downloads#Current_release`

MacOS
- Installer la distribution "Python scientifique" Académique proposée par la société Enthought : `http://www.enthought.com`
- ajouter l'IDE spyder : `http://code.google.com/p/spyderlib/downloads/list` (disponible pour tous les environnements.)

3.3 L'interpréteur Python

Cet interpréteur est particulièrement utile pour comprendre les bases de Python et réaliser nos premiers petits programmes. Le principal inconvénient, c'est que le code que vous saisissez n'est pas sauvegardé.

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre est la série de trois chevrons qui se trouve en bas à gauche des informations : `>>>`. Ces trois signes signifient : " je suis prêt à recevoir tes instructions".

Exemples : utilisation de Python comme une calculatrice

```
>>> 7
7
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 + 2
11.5
>>> 3.11 + 2.08
5.1899999999999995
>>>
```

3.4 Le langage Python

- Le langage Python est constitué :
  - de mots clefs, qui correspondent à des instructions élémentaires (`for`, `if`...)
  - de littéraux, qui expriment des valeurs constantes de types variés (25, 1.e4, 'abc'...)
  - de types intrinsèques (`int`, `float`, `list`, `str`,...)
  - d'opérateurs (`=, +, *, /, %...`)
  - de fonctions intrinsèques (Built-in Functions) qui complètent le langage.
- L'utilisateur peut créer :
  - des identificateurs qui référencent des objets,

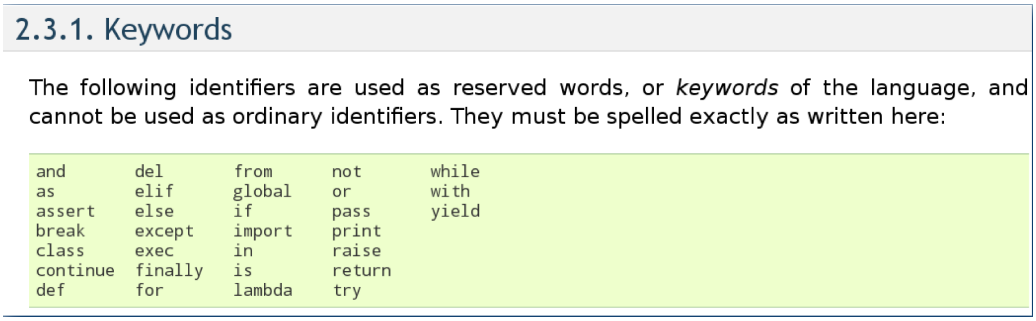
– des expressions combinant instructions, identificateurs, opérateurs et fonctions.

- Les expressions sont évaluées par l'interpréteur.
- L'évaluation fournit des objets, qui peuvent être référencés par aUectation.
- Un programme Python est une suite d'instructions combinant :

  instructions élémentaires,
  littéraux,
  identificateurs,
  opérateurs,
  fonctions intrinsèques
  expressions.

- Les instructions d'un programme sont exécutées par l'interpréteur Python.

Mots clefs du langage
Il n'y a que 31 mots clefs (*key words*) dans le langage Python (2.7).
Doc Python > Language Reference > Identifiers and keywords
`http://docs.python.org/2/reference/lexical_analysis.html`



2.3.1. Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Chacun des mots clefs est une instruction du langage.

Boucles	
`for`	boucle for
`while`	boucle while
`continue`	tour de boucle suivant
`break`	sortie de boucle

Définitions d'objets	
`def`	définition d'une fonction
`return`	fin fonction, renvoyer valeur
`class`	définition d'une classe
`global`	variable globale

Tests	
`if`	test
`else`	alternative
`elif`	test combiné

Opérateurs logiques	
`and`	ET logique
`or`	OU logique
`not`	négation

importation d'un Module	
`import`	module entier
`from`	objets d'un module

Objets	
`del`	détruire
`in`	parcourir
`is`	comparer
`print`	afficher

gestion des Exceptions	
`assert`	définir une assertion
`raise`	émettre une exception
`try`	ouvrir un bloc try
`except`	traitement des exceptions
`finally`	traitement des exceptions

Autre	
`exec`	interpréter une chaîne
`pass`	ne rien faire...
`with`	objet dans un context
`as`	utilisé avec `with`, `import`...
`yield`	fonction générateur
`lambda`	fonction *inline*

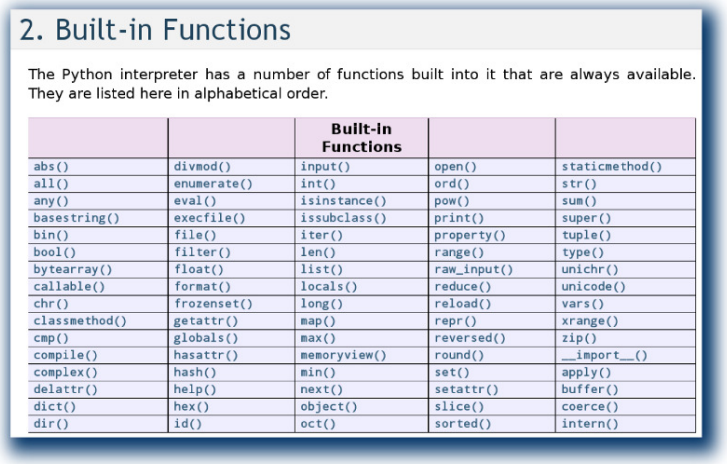
Les opérateurs

Opérateurs arithmétiques	
`+`	addition
`-`	soustraction
`*`	mutiplication
`/`	division (quotient de la division entière si les 2 opérandes sont entiers)
`//`	quotient de la division entière
`**`	exponentiation ($0^0 = 1$)
`%`	modulo
`<< n, >> n`	décalage à gauche, à droite de n bits
Opérateurs de comparaison	
`<, <=`	inférieur strict à, inférieur ou égal à
`>, >=`	supérieur strict à, supérieur ou égal à
`==`	égal à
`!=`	différent de

Opérateurs logiques	
`and`	ET
`not`	négation
`or`	OU
`&`	ET bit à bit
`^`	XOR bit à bit
`|`	OR bit à bit
`~`	complément à 2
Autres Opérateurs	
`is`	même identité ?
`is not`	identité différente ?
`in`	appartient à l'itérable ?
`not in`	n'appartient pas à l'itérable ?

Opérateurs avec affectation	
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`

Les Fonctions Intrinsèques http ://docs.python.org/2/library/functions.html
... il y en a beaucoup ☺



2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Fonctions intrinsèques >200(extraits)

- `abs` valeur absolue, module...
- `bin` convertir en base 2...
- `bool` convertir en booléen...
- `enumerate` parcourir un itérable (c'est à dire, de manière générale, les chaines, les listes, les tuples (n-uplets) et les dictionnaire) en numérotant les éléments... Renvoie un objet de type *enumerate*

```
>>> L=['y','yes','o','oui']
>>> enumerate(L)
<enumerate at 0x106cf6fa0>
>>> list(enumerate(L))
[(0, 'y'), (1, 'yes'), (2, 'o'), (3, 'oui')]
```

- `eval` interpréter en Python une chaîne de caractère...

```
>>> x = 1
>>> print eval('x+1')
2
```

- `float` convertir en float...
- `int` convertir en entier...
- `hex` convertir en base 16...
- `max` maximum d'un itérable ou de plusieurs objets...
- `map` appliquer une fonction aux éléments d'une liste...
- `min` minimum d'un itérable ou de plusieurs objets...

- `range` générer une suite de valeurs entières...

  `range(start, stop[, step])`

  Il s'agit d'une fonction polyvalente pour créer des listes contenant des progressions arithmétiques. Il est le plus souvent utilisé dans les boucles. Les arguments doivent être des nombres entiers. Si l'argument `step` est omis, sa valeur par défaut est 1. Si l'argument `start` est omis, sa valeur par défaut est 0. La forme complète renvoie une liste d'entiers simples `[start, start+step,start+2*step, ...]`. Si `step` est positif, le dernier élément est le plus grand `start + i*step` inférieur à `stop`, si step est négatif, le dernier élément est le plus petit `start + i*step` supérieur à `stop`. `step` ne doit pas être zéro.

  ```
  >>> range(10)
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  >>> range(1, 11)
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  >>> range(0, 30, 5)
  [0, 5, 10, 15, 20, 25]
  >>> range(0, 10, 3)
  [0, 3, 6, 9]
  >>> range(0, -10, -1)
  [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
  >>> range(0)
  []
  >>> range(1, 0)
  []
  ```

- `repr` créer la conversion d'un objet en chaîne de caractères...

  Retourne une chaîne contenant une représentation imprimable d'un objet. Il s'agit de la même valeur donné par les conversions (reversed quotes). Il est parfois utile de pouvoir accéder à cette opération comme une fonction ordinaire. Pour de nombreux types, cette fonction fait une tentative de retourner une chaîne qui donnerait un objet ayant la même valeur lorsqu'il est passé à `eval()`, sinon la représentation est une chaîne entre crochets qui contient le nom du type de l'objet ensemble avec des informations supplémentaires, y compris souvent le nom et l'adresse de l'objet. Une classe peut contrôler ce que cette fonction retourne à ses instances en définissant une méthode `__repr__()`.

- `reversed` parcourir un itérable en sens inverse...

- `str` convertir simplement en chaîne de caractères...

  Retourne une chaîne contenant une représentation joliment imprimable d'un objet. Pour les chaînes, cela renvoie la chaîne elle-même. La différence avec `repr(objet)` est que `str(objet)` n'essaie par toujours retourner une chaîne qui est acceptable à `eval()`; son but est de retourner une chaîne imprimable.

- `sum` somme d'un itérable ou de plusieurs objets...

- `zip` parcourir plusieurs itérables en même temps...

  `zip([iterable, ...])`

  Cette fonction retourne une liste de tuples (n-uplets), où le i-ème tuple contient le i-ème élément de chacune des séquences d'arguments ou iterables. La liste retournée est

tronquée en longueur à la longueur de la séquence de l'argument le plus court. Quand il y a plusieurs arguments qui sont tous de la même longueur, `zip()` est similaire à `map()` avec un argument initial de `None`. Avec un argument de séquence unique, il retourne une liste de 1-uplets (1-tuples). Sans argument, elle retourne une liste vide.

L'ordre d'évaluation de gauche à droite des iterables est garantie. Cela rend possible un idiome pour grouper une série de données en groupes de longueur n à l'aide de `zip(*[iter(s)]*n)`.

`zip()`, en collaboration avec l'opérateur `*` peut être utilisé pour décompresser une liste:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

Fonction intrinsèques (extraits)

- `enumerate` : parcourt un itérable en numérotant les éléments

```
>>> L = ['y', 'yes', 'o', 'oui']
>>> for i, rep in enumerate(L):
    print i, rep
(0, 'y'), (1, 'yes'), (2, 'o'), (3, 'oui')
```

- `eval` : renvoie l'évaluation d'une expression contenue dans une chaîne

```
>>> x=56.3
>>> rep = raw_input("Expression en x: ")
Expression en x: x**2-100
>>> eval(rep)
3069.6899999999996
>>> print eval(rep)
3069.69
```

- Les fonctions `min`, `max` et `sum` acceptent une liste (un itérable) en argument

```
>>> L1 = [10, 11, 12, 13, 14]
>>> sum(L1)
60
>>> min(L1), max(L1)
(10, 14)
```

- `map` : applique une fonction aux éléments d'une liste

```
>>> L = ["1e2", "2.3e-2", "1", "2.3"]
>>> map(float, L)
[100.0, 0.023, 1.0, 2.3]
```

- `range` : génére une suite d'entiers (progression arithmétique)

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, -5, -1)
[0, -1, -2, -3, -4]
```

très utilisée pour les boucles :

```
>>> for i in range(5):
    print i*i,
0 1 4 9 16
```

- `reversed` : parcourt un itérable en sens inverse

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in reversed(L):
      print i
5 4 3 2 1
```

- `repr` : convertit un objet en une expression `str`
  dans la plupart des cas, `eval(repr(•))` == • est vrai !

```
>>> 0.1 + 0.2
0.30000000000000004
>>> print 0.1 + 0.2
0.3
>>> str(0.1 + 0.2)
'0.3'
>>> repr(0.1 + 0.2)
'0.30000000000000004'
```

- `str` : convertit un objet en chaîne de caractères

```
>>> str(12.4)
'12.4'
>>> str([1,2])
'[1, 2]'
```

très utilisée pour les affichages écran :

```
>>> a = 12.7
>>> print 'valeur de a : ' + str(a)
valeur de a : 10
```

- `zip` : combine plusieurs itérables

```
>>> Z = zip([1,2,3,4,5], ["Red","Green", "Blue"], [200, 210, 220])
>>> Z
[(1, 'Red', 200), (2, 'Green', 210), (3, 'Blue', 220)]
>>> for x, y in zip([1,2,3], [1e2, 1.1e2, 1.3e2]):
      print x,y

1 100.0
2 110.0
3 130.0
```

3.5 Structuration d'une application Python

Comme nous l'avons vu, Python dispose d'un nombre limité de commandes de base. Comme ce langage est disponible pour de multiples applications (Web jusqu'au calcul scientifique), les développeurs ont choisi d'ajouter des fonctions sous la forme de modules. Tous les programmeurs peuvent également ajouter des fonctions propres dans de nouveaux modules.

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à importer le module et utiliser ensuite toutes les fonctions et variables prévues. Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module lui-même pour coder une calculatrice scientifique, nous verrons surtout les différentes méthodes d'importation.

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module math ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites "tiens, mon programme risque d'avoir besoin de fonctions mathématiques". Nous allons voir une première syntaxe d'importation.

```
>>> import math
>>>
```

La syntaxe est facile à retenir : le mot-clé import, qui signifie "importer" en anglais, suivi du nom du module, ici math.

Après l'exécution de cette instruction, rien ne se passe. . . en apparence. En réalité, Python vient d'importer le module math. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point "." puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math . sqrt (16)
4
>>>
```

En vérité, quand vous tapez import math, cela crée un espace de noms dénommé “math”, contenant les variables et fonctions du module math. Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms math. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction sqrt que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math`.

Dans certains cas, vous pourrez vouloir changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
mathematiques . sqrt (25)
```

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, on utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module math. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
>>> from math import fabs
>>> fabs ( -5)
5
>>> fabs (2)
2
>>>
```

Vous aurez remarqué qu'on ne met plus le préfixe `math` devant le nom de la fonction. En effet, nous l'avons importée avec la méthode `from` : celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme `print` par exemple. Si vous avez compris les explications sur les espaces de noms, vous voyez que `print` et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant “*” à la place du nom de la fonction à importer.

```
>>> from math import *
>>> sqrt (4)
2
>>> fabs (5)
5
```

Une application Python peut-être structrée hiérarchiquement en utilisant le système de fichiers de la machine.

- Un fichier définit un module. La commande “import” permet d'accéder au contenu d'une module.

```
m1.py:
def f1(x): return x + 3

def g1(x): return x - 1

m2.py:
import m1
from m1 import g1
def f2(x): return m1.f1(x) + g1(x)
```

- Un répertoire définit un package. Il contient donc des modules et des sous-packages. La commande "import" permet de naviguer dans la hiérarchie des packages.

4 Sous-ensemble impératif

4.1 Types de données et expressions

Python définit des types simples (entiers, booléens, ...) et des types composés dont les éléments peuvent contenir plusieurs objets (listes, tuples, ensembles, ...). Chaque type contient des valeurs et offre des opérations.

`http://docs.python.org/2/library/stdtypes.html`

Pour connaitre le type d'une donnée, on a accès à la commande `type()`.

```
>>> type(1)
int
>>> type(1L)
long
```

4.1.1 Types simples

- Booléens

type bool	
opérateur	type
True, False	bool
_ and _, _or_	bool, bool → bool
not _	bool → bool
_==_, _!=_, _<_,...	*,* → bool
bool(_)	* → int

Remarque: conversion implicite vers int

Tout objet peut être testé pour la valeur vraie. Tous les exemples suivants sont considérés comme faux

- None
- False
- le zéro de tout type numérique par exemple, `0, 0L, 0.0, 0j`.
- toute liste vide, par exemple, `'', (), []`.
- toute application (mapping) vide, par exemple, `{}`.

Toutes les autres valeurs sont considérées comme vraies. Les opértions booléennes renvoient `0` ou `False`, ou bien `1` ou `True`.

Les opérations booléennes

Opérations	Résultats
`x or y`	Si x est faux, alors y, sinon x
`x and y`	Si x est faux, alors x, sinon y
`not x`	Si x est faux, alors `True`, sinon, `False`.

Comparaisons

Opérations	Interprétation
`<`	inférieur strict
`<=`	inférieur ou égal
`>`	supérieur strict
`>=`	supérieur ou égal
`==`	égal
`!=`	différent
`is`	objet identifique
`is not`	objet non identique

- Entiers

Il y a deux types d'entiers en Python : les `int` et les `long`. Pour l'utilisation quotidienne, on ne se préoccupe pas de savoir si l'on travaille avec un type ou l'autre. Les `int` sont des entiers codés sur un nombre fixé de bits. Les `long` permettent de dépasser cette limite et de coder les entiers avec un nombre de bits arbitraires. Pour connaitre l'entier le plus grand disponible sur la machine sur laquelle on travaille, il suffit de taper les commandes suivantes

```
>>> import sys
>>> sys.maxint
9223372036854775807
```

Tous les nombres entiers dépassant ce nombre seront codés comme des `long`. Ici, on a

$$\mathrm{nb}_{\mathrm{bits}} = \frac{\ln\left(9223372036854775807+1\right)}{\ln\left(2\right)} + 1 = 64.$$

On a donc

```
>>> type(9223372036854775807)
int
>>> type(9223372036854775807+1)
long
```

type int	
opérateur	type
..., -1,0,1,...	int
_ + _, _ - _,_ * _, _ / _, _ % _, _ // _,_ ** _	int, int → int
abs(_)	int → int
int(_)	* → int
cmp(_,_)	*,* → int

Remarque: conversion implicite vers float

L'opération `/` entre deux entiers renvoie un entier sous la version 2.7, et un réel à partir de la version 3.0 (division entière ou flottante). L'opération `%` renvoie le reste de la division entière, alors que `//` renvoie la division entière arrondi vers $-\infty$.

La commande `cmp(x,y) --> integer` (x et y de n'importe quel type) renvoie "Return negative if x<y, zero if x==y, positive if x>y" (help(cmp)).

Afin de forcer l'utilisation des `long`, on les déclare avec un `l` ou `L` en suffixe `2L`.

- Flottants

type float	
opérateur	type
..., -1.5,0.3...	float
_ + _, _ - _,_ * _, _ / _,_**_	float, float → float
round(_,_)	float, int → float
float(_)	*→ float

L'instruction `round` renvoie l'entier le plus proche. Pour avoir accès aux parties entières inférieure et supérieure, on doit les importer depuis le module `math`.

```
>>> from math import floor,ceil
```

- Les complexes

Ils sont construits à partir des flottants : complex(re,im). On peut également utiliser le nombre imaginaire pur $1j$

```
>>> complex(3.,2.)
(3+2j)
>>> 4+5j
(4+5j)
```

Quelques exemples

```
>>> 2**31          # x**y  : x exposant y
2147483648
>>> c = 123        # int : entiers 32 bits, dans [-2147483648, 2147483647]
>>> d = 1.47e-7    # float : flottants IEEE754 64 bits,
                   # ~16 chiffres significatifs
                   # ~10**-308 < val. abs. < ~10**308
>>> e = 2.1 +3.45j # complex
>>> e.real, e.imag
(2.1, 3.45)
```

- Chaînes de caractères

type str		
opérateur	type	commentaire
. . . , 'abcd',"abcd". . .	str	constante
_ + _	str, str → str	concaténation
_ [_]	str, int → str	chaîne d'un caractère
_ [_:_]	str, int,int → str	sous-chaîne entre d et f-1
_ [:_]	str, int → str	sous-chaîne jusqu'à f-1
_ [_:]	str, int → str	sous-chaîne depuis d
len(_)	str → int	longueur
str(_)	→ string	conversion en chaîne

Les chaines n'incluent pas par défaut les caractères accentués. On utilise alors le codage unicode. Pour l'utiliser, on préfixe les chaines par `u`.

```
>>> s=u'é'
>>> type(s)
unicode
>>> s
u'\xe9'
>>> print s
é
```

- `len(•)` : renvoie le nombre d'éléments de la chaîne
- Indexation : $\bullet[i]$: renvoie l'élément de rang i de la chaîne; le premier élément commence à 0 et le dernier à $\text{len}(\bullet) - 1$.
- Sous-chaîne :
  $\bullet[i : j] \rightarrow$ sous-liste de i inclus à j exclu
  $\bullet[i : j : k] \rightarrow$ sous-liste de i inclus à j exclu par pas de k.
- Indexation positive, négative :
  $0 \leq i \leq \text{len}(\bullet) - 1 \rightarrow$ rang dans la liste
  $-\text{len}(\bullet) \leq i \leq -1 \rightarrow -1$: le dernier, -2 : l'avant dernier
  Il suffit d'imaginer une chaine comme un motif périodique

a	b	c	d	a	b	c	d
-4	-3	-2	-1	0	1	2	3

```
>>> 'abc'+'efg'
'abcefg'
>>> len('toto')
4
>>> len('abcdefgh')
8
>>> 'abcdefgh'[0:2]
'ab'
>>> 'abcdefgh'[0:7:2]
'aceg'
>>> 'abcdefgh'[-1]
'h'
```

La donnée `str` est un objet, et on a donc des méthodes ou fonctions spécifiques qui existent pour cet objet.

```
>>> dir(str)
[... 'capitalize', 'center','count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> help(str.__add__)
Help on wrapper_descriptor:

__add__(...)
    x.__add__(y) <==> x+y
>>> str.capitalize('toto')
'Toto'
>>> str.upper('abGc')
'ABGC'
>>> "this   is\na\ttest"
'this   is\na\ttest'
>>> print "this   is\na\ttest"
this   is
a test
>>> str.split("this   is\na\ttest")  # split without any arguments splits on
                                     # whitespace. So three spaces, a carriage
                                     # return, and a tab character are all the same
['this', 'is', 'a', 'test']
>>> print str.join(str.split("this   is\na\ttest"))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-2ed401e4bceb> in <module>()
----> 1 print str.join(str.split("this   is\na\ttest"))

TypeError: descriptor 'join' requires a 'str' object but received a 'list'
>>> print " ".join(str.split("this   is\na\ttest"))
this is a test
```

Explication du dernier test : You can normalize whitespace by splitting a string with split and then rejoining it with join, using a single space as a delimiter.

Exemple

```
2.0 + 3 --> 5.0
int('2') + 3 --> 5
float(2)/3 --> 0.666666
2 + 'abc' --->  erreur
True + 1 --> 2
'abc' < 1 --> False
```

4.1.2 Types composés

Ce sont des conteneurs pour les autres types.

Collections ordonnées : Types `list`, `tuple`, `str`, `unicode` : les séquences

```
>>> [1.e4, 2e4, 0, 1, e1] # Objet list entre [...]
>>> (1, 2, 3, e1, f)      # Objet tuple entre (...)
>>> 'l"impact'            # Objet str entre '...' ou "..."
>>> "l'impact"
>>> u'Chaîne accentuée'   # Objet unicode commence par un u
                          # (caractères accentués)
```

Collection non ordonnée de paires (clef, objet) : Type dict (dictionnaire)

```
>>> {'Lundi':1, 'Mardi':2}     # Objet dict entre {...}
>>> {'mean':1.2, 'stdDev':9.4}
```

Collection non ordonnée d'items uniques : Type set (ensemble)

```
set([1,5,12,6,'a'])    # Un set est cr\'e\'e avec : set(...)
```

Tous les conteneurs Python sont des objets itérables : on peut les parcourir avec une boucle for.

- Listes

  C'est une collection ordonnée d'objets quelconques (conteneur hétérogène). Une liste n'est pas un vecteur!

type list	
opérateur	type
[_,... _]	*,..., * → list
[]	list
_ + _	list, list → list
_ [_]	list, int → *
_ [_:_]	list, int,int → list
_ [:_]	list, int → list
_ [_:]	list, int → list
_ in _, _ not in _	*,list → bool
all(_)	list → bool
any(_)	list → bool
len(_)	list → int
str(_)	→ string

  Concatenation

```
>>> [10,12]+[13,14]
[10, 12, 13, 14]
>>> [3,4] + [1,2]
[3, 4, 1, 2]
```

  L'opération `list*n` ou `n*list` concatène n copies de la liste :

```
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [0,1]*2
[0, 1, 0, 1]
```

C'est un objet mutable

```
>>> L=[1,5,-2]
>>> L[1]=3
>>> L
[1, 3, -2]
```

- tuple

  Un tuple est une liste non mutable :

  - ses éléments ne supportent pas la ré-affectation
  - mais ses éléments mutables peuvent être modifiés

  Il en est de même pour les chaines qui ne sont pas mutables.

type tupe	
opérateur	type
..., _,... _,...	*,..., * → tuple
()	tuple
_ + _	tuple, tuple → tuple
_ [_]	tuple, int → *
_ [_:_]	tuple, int,int → tuple
_ [:_]	tuple, int → tuple
_ [_:]	tuple, int → tuple
_ in _, _ not in _	*,tuple → bool
len(_)	tuple → int
str(_)	→ string

```
>>> () # tuple vide
>>> (2,) # 1-tuple
>>> 2, # 1-tuple
>>> 2, 3, 4 # tuple implicite
>>> (1,'non',[1,2,3])
```

Il est possible de tester si un élément appartient à un tuple ou un liste

```
>>> L=[1,3,0]
>>> 2 in L
False
>>> 2 not in L
True
```

Opérateur de formatage pour l'affichage : Si l'on souhaite afficher par exemple un réel avec une certaine précision (nombre de chiffres avant et après la virgule) ou bien un entier

avec un certain nombre de caractères, il est nécessaire de procéder à un formatage. Cela met en jeu chaines et tuples.

`%: str, tuple → str`

permet de construire une chaîne à partir d'un format et d'un ou d'un tuple d'arguments.

```
"x=%d, y=%d" % (2,3) --> "x=2, y=3"
"%g" % 2.3 --> "2.3"
"%e" % 2.3 --> "2.300000e+00"
```

- Collection non-ordonnée d'items uniques : les ensembles
  Les objets set sont créés avec le constructeur de la classe set et un argument de type list, str ou dict, ou par la notation $\{e_1, \ldots, e_n\}$.

```
>>> {1,2,1}
{1,2}
>>> set([1,5,12,6])
{1, 5, 6, 12}
>>> set('Lettres ')
{' ', 'L', 'e', 'r', 's', 't'}
>>> {1,5,12,6} | {5,6,7}          # Union
{1, 5, 6, 7, 12}
>>> {1,5,12,6} & {5,6,7}          # intersection
{5, 6}
>>> {1,5,12,6} - {5,6,7}          # Dans la premi\'ere liste, mais pas
                                  # dans la seconde
{1, 12}
>>> {1,5,12,6} ^ {5,6,7}          # Dans la premi\'ere liste, ou
                                  # exclusif la seconde
{1, 7, 12}
```

type set	
opérateur	type
$\{e_1, \ldots, e_n\}$	*,...,* → set
set(_)	list → set
set(_)	table → set
set(_)	str → set
_ | _, _ & _, _ - _, _ ^ _	set, set → set
_ in _, _ not in _	*,list → bool
.issubset()	set,set → bool
len(_)	set → int
str(_)	→ string

- Table ou dictionnaire : une table définit une fonction en extension associant des clés à des valeurs.

type dict		
opérateur	type	
$\{x_1 : v_1, \ldots, x_n : v_n\}$	dict	associe v_i à x_i
_ [_]	dict, * → *	valeur associée à la clé
_ in _, _ not in _	*,dict → bool	contenance de la clé
len(_)	dict → int	
str(_)	dict → string	

Remarque Aucune opération de concaténation de tables

Exemple

```
>>> (('abc', 2) + (3,4)) [1:2]
(2,)
>>> (('abc', 2) + (3,4)) [1:]
(2,3,4)
>>> cyL1='L':1.2, 'D':0.5, 'unit':'n'
>>> cyL1['L']
1.2
>>> cyL1['unit']
'n'
>>> cyL1['d']=0.1
>>> cyL1
'd': 0.1, 'L': 1.2, 'unit': 'n', 'D': 0.5
>>> cyL1.keys()          # Recherche des mots clés
['d', 'L', 'unit', 'D']
>>> cyL1.values()        # Recherche des valeurs
[0.1, 1.2, 'n', 0.5]
```

4.2 Instructions

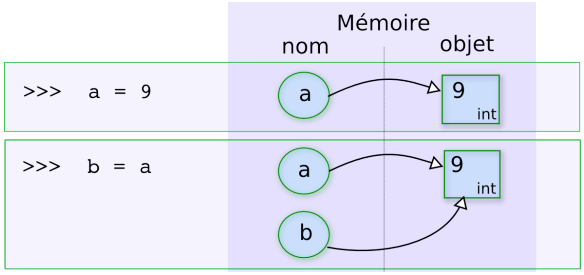
- Variables, affectation

```
x = 2 + 3
y = x + 1
l = [1,2,3]
f = lambda(x):x+1
r = f(1)
```

Une variable doit être initialisée avant d'être utilisée. Une variable ne contient pas un objet, elle est juste une référence faite à un objet qui existe en mémoire.

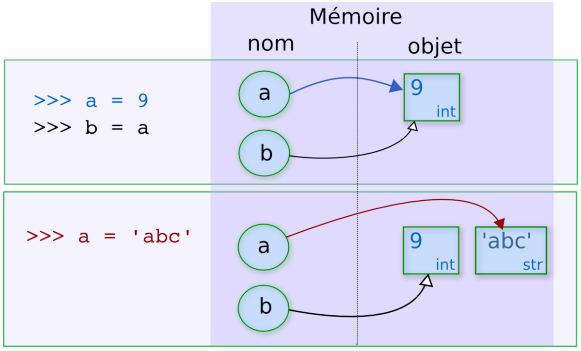
Affectation : référence → objet
L'affectation opère de droite à gauche :

- le terme de doite est une expression évaluée, qui crée ou référence un objet
- le terme de gauche est un identificateur (référence, nom, étiquette)



C'est l'objet qui porte le type et les données (la valeur, pour un objet numérique).

– Un objet ne peut pas changer d'identité (≈ adresse mémoire), ni de type.
– Un objet peut avoir plusieurs noms (alias).
– Un identificateur associé d'abord à un objet peut ensuite référencer un nouvel objet !



– Quand un objet n'a plus de nom (nombre de références nul), il est détruit (mécanisme automatique de "ramasse-miettes", garbage collector).

Cas 1

```
>>> x=[13,43,17]
>>> y=x
```

On a un seul objet liste en mémoire avec deux alias

Cas 2

```
>>> x=[13,43,17]
>>> y=[13,43,17]
```

On a deux objets liste différent en mémoire avec deux alias

On pose `y[0]=20` et `x[1]=12`.

Cas 1 : à la fois `x` et `y` sont changés

```
>>> x
[20, 12, 17]
>>> y
[20, 12, 17]
```

Cas 2 : `x` et `y` sont différents

```
>>> x
[13, 12, 17]
>>> y
[20, 43, 17]
```

Objet

– L'affectation = permet d'affecter un nom à un objet créé ou existant.
– **type**(•) renvoie le type de •.
– **id**(•) renvoie l'identité de •.
– L'opérateur == teste l'égalité des valeurs de 2 objets.
– L'opérateur `is` compare l'identité de 2 objets.

```
>>> a = 9   # Création objet "entier 9"
>>> type(a)
<type 'int'>
>>> b = a   # 2me nom pour "entier 9"
>>> id(a), id(b)
(33724152, 33724152)
>>> type(b)
<type 'int'>
```

```
>>> c = 9.
>>> id(a), id(b),id(c)
(33724152, 33724152, 38603664)
>>> a == c     # égalité des valeurs ?
True
>>> a is c     # même identité ?
False
>>> a is b     # même identité ?
True
```

– Le type d'un objet détermine :
  * les valeurs possibles de l'objet
  * les opérations que l'objet supporte
  * la signification des opérations supportées.

```
>>> a = 2 + 3  # addition d'entiers
>>> a
5
>>> 2*a, 2.*a  # mult. dans N, R
(10, 10.0)
>>> 2/3, 2./3. # divis. dans N, R
(0, 0.6666666666666666)
```

```
>>>[1, 2]+[5, 6] # concat. des listes
[1, 2, 5, 6]
>>> 2*[5, 6]     # n concaténations
[5, 6, 5, 6]
>>> [5, 6]/2     # pas défini !
...unsupported operand type(s) for /
'list' and 'int'
```

– Un objet dont le contenu peut être changé est dit mutable (non mutable sinon).

```
>>> a = 'Un str est non-mutable !'
>>> a[0]
'U'
>>> a[0] = 'u'
TypeError: 'str' object does
  not support item assignment
```

```
>>> a = [1, 2, 3]  # liste: mutable
>>> a[1]
2
>>> a[1] = 2.3
>>> a
[1, 2.3, 3]
```

Séquences : la Classe `list`

- **Sous-liste** (*Slicing*) :
  - `•[i:j]` ⇝ sous-liste de `i` inclus à `j` **exclu**
  - `•[i:j:k]` ⇝ sous-liste de `i` inclus à `j` **exclu**, par pas de `k`
- Indexation positive, **négative** :
  - `0 ⩽ i ⩽ len(•)-1` ⇝ rang dans la liste
  - `-len(•) ⩽ i ⩽ -1` ⇝ `-1` : le dernier, `-2` : l'avant dernier...

```
>>> L1 = [10,11,12,13,14]
>>> L1[1:3]
[11, 12]
>>> L1[:3]
[10, 11, 12]
>>> L1[1:]
[11, 12, 13, 14]
>>> L1[:] # toutes les valeurs de L1
[10, 11, 12, 13, 14]
```

```
>>> L1 = [10,11,12,13,14]
>>> L1[::2]
[10, 12, 14]
>>> L1[::-1]
[14, 13, 12, 11, 10]
>>> L1[-2:1]
[]
>>> L1[-2:1:-1]
[13, 12]
```

Séquences : la Classe `list`

⚠ **Copie** d'un objet `list` : Référence, copie superficielle ou copie profonde

`L2 = L1` référence supplémentaire `L2` sur l'objet référencé par `L1`

```
>>> L1 = [10, 11, 12] ; L2 = L1      # L1 et L2 : 2 noms d'un même objet
>>> id(L1), id(L2), L2 is L1
(50660560, 50660560, True)
>>> L2[0] = 0
>>> L1, L2
([0, 11, 12], [0, 11, 12])
```

⚠ **Copie** d'un objet `list` : Référence, copie superficielle ou copie profonde

`L3 = L1[:]` nouvel objet `list` référencé par `L3`, initialisé par `L1` (*shallow copy*)

```
>>> L1 = [10, [11, 12]] ; L3 = L1[:]    # Shallow copy !
>>> id(L1), id(L3), L3 is L1
(50660560, 51420768, False)
>>> L3[0] = 0; L3[1][0] = -1
>>> L1, L3
([10, [-1, 13]], [0, [-1, 13]])
```

Séquences : la Classe `list`

⚠ **Copie** d'un objet `list` : Référence, copie superficielle ou copie profonde

```
from copy import deepcopy
L4 = deepcopy(L1)
```

nouvel objet `list` référencé par `L4`, copie complète de l'objet référencé par `L1` (*deep copy*)

```
>>> L1 = [10, [11, 12]]
>>> from copy import deepcopy
>>> L4 = deepcopy(L1)       # Deep copy !
>>> id(L1), id(L4), L4 is L1
(53980336, 53982200, False)
>>> L4[0] = 0
>>> L4[1][0] = -1
>>> L1, L4
([10, [12, 13]], [0, [-1, 13]])
```

Exemple de variables non mutables : les tuples

```
>>> a  = ()      # tuple vide
>>> b  = (2,)    # 1-tuple
>>> c  = 2,      # 1-tuple
>>> d  = 2, 3, 4 # tuple implicite
>>> d
(2, 3, 4)
>>> t=(1,'non',[1,2,3])
>>> t
(1, 'non', [1, 2, 3])
>>> t[0]=2 # ré-affectation élément
...Error: 'tuple' object does
not support item assignment
```

```
>>> t[1]='oui'  # ré-affection élément!
...Error: 'tuple' object does not
support item assignment
>>> t[1][0]='N' # élément str non mutable
...Error: 'str' object does not
support item assignment
>>> t[2]=[3,4,5] # ré-affection élément
...Error: 'tuple' object does not
support item assignment
>>> t[2][0]=-3 # élément list mutable
>>> t
(1, 'non', [-3, 2, 3])
```

- Branchement conditionnel.

  Il est possible de réaliser des instructions seulement si une condition est vérifiée. Attention aux indentations.

```
if condition :
  ...
elif condition :
  ...
else :
  ...
```

```
>>> x = 0
>>> if 17 % 2 == 1:
...     x = x + 1
...
>>> x
1
```

  Ici, une des particularités de Python entre en jeu. Pour indiquer à Python le bloc d'instruction à exécuter si la condition est vérifiée, ce bloc d'instruction doit être indenté, c'est-à-dire être mis en retrait par rapport à la marge gauche. Il suffit pour cela de mettre au début de chaque ligne de ce bloc des espaces ou une tabulation pour placer le nouveau bloc plus à droite. On peut utiliser au choix un nombre fixe d'espaces ou une tabulation mais il faut prendre garde à conserver ce choix tout au long du bloc. Cette indentation se substitue aux blocs `begin...end` de nombreux langages.

  Donc : Attention à l'indentation des branches et au saut de ligne final. Par exemple, ceci est valide :

```
>>> if 17 % 2 == 1:
...     x += 1
...     x *= x
```

  mais ceci ne l'est pas :

```
>>> if 17 % 2 == 1:
...     x += 1
...         x *= x
```

On peut effectuer des instructions si la condition n'est pas vérifiée grâce à l'instruction `else` qui doit se trouver au même niveau que le if :

```
>>> if 17 % 2 == 1:
...     x += 1
...else:
...     x += 2
```

Il est également possible et même recommander d'abréger les tests imbriqués à l'aide de `elif`. Ainsi, au lieu d'écrire :

```
>>> if n % 3 == 0:
...     x += 1
... else:
...     if n % 3 == 1:
...             x += 3
...     else:
...             x *= x
```

on utilisera :

```
>>> if n % 3 == 0:
...     x += 1
... elif n % 3 == 1:
...     x += 3
... else:
...     x *= x
```

La condition peut être de n'importe quel type. Chaque type contient une valeur représentant faux (résultat de la conversion vers bool). Les autres valeurs représentent vrai:

type	bool	int	float	str	tuple	list
faux	False	0	0.0	”	()	[]

- Répétition : Boucles conditionnelles. On réalise une boucle conditionnelle à l'aide du mot-clé while.

```
while expression:
  ...
  break   # sortie de boucle
```

```
>>> n = 42
>>> while n != 0:
...     n /= 2
...
```

- Itération : Boucles inconditionnelles; Les boucles bornées sont introduites par le mot-clé `for`.

```
for x in ... :
  ...
  continue # passe à l'élément suivant
  break   # sort de la boucle
```

Exemple: somme des éléments pairs entre 0 à N-1

```
N = 10
s = 0
for i in range(N):
  if i % 2 == 0:
    s += i

print s
```

Exemple bateau : le calcul de la factorielle.

```
>>> n = 100
>>> r = 1
>>> for i in range(1, n+1):
...   r *= i
...
>>> r
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000L
```

Remarquez la syntaxe : pour effectuer une itération bornée de la variable i de a inclus à b exclu, on dit que i doit parcourir la liste `range(a,b)`. Attention, `range(a,b)` est l'intervalle d'entiers $[a, b[$ ouvert à droite, d'où le $n + 1$ dans le code ci-dessus.

Mais on peut aussi itérer sur une liste arbitraire :

```
>>> premiers = [2, 3, 5, 7, 11, 13, 17]
>>> prod = 1
>>> for x in premiers :
...     prod *= x
>>> prod
510510
```

- Manipulation du flot de contrôle

  Pour manipuler le flot de contrôle on peut utiliser deux instructions spécifiques :

  - `break` interrompt la boucle ;
  - `continue` termine le tour de boucle en cours et retourne au début de la boucle.

```
>>> l = [ 2, 3, 5, 7 ]
>>> i = 0
>>> for n in l:
...     if p == n:
...             break
...     i += 1
>>> i
1
```

4.3 Liste en compréhension

Quelques goodies Pythoniques...

list comprehensions écriture *inline* d'une liste où les éléments sont le résultat d'un calcul :

```
[expression for element in liste]
[expression for element in liste if predicat]
```

Exemples

```
>>> [i*i for i in range(5)]
[0, 1, 4, 9, 16]
>>> [i*i for i in range(10) if i % 2 == 0]
[0, 4, 16, 36, 64]
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Listes

```
[ x ** 2 for x in [1,2,3] ] --> [1,4,9]
[ (x,y) for x in range(2) for y in ['a', 'b'] ] --> [(0,'a'), (0,'b'), (1,'a'), (1,'b')]
[ x for x in range(10) if x % 2 == 0 ] --> [0, 2, 4, 6, 8]
[ x for y in [[1],[2,3],[4]] for x in y ] --> [1,2,3,4]
[ x/2 for x in range(5) ] ---> [0, 0, 1, 1, 2]
```

Ensembles

```
{x/2 for x in range(4)} --> set([0,1])
```

Tables

```
{ x : 2*x+1 for x in range(4) } --> {0:1, 1:3, 2:5, 3:7 }
```

4.4 Les Entrées/Sorties clavier, écran (Input/Output)

Les Entrée/Sortie clavier/écran sont très utiles pour faire dialoguer simplement un programme avec un utilisateur.

- La fonction `raw_input('message')` est utilisée :
  - pour afficher un message à l'écran,
  - pour capturer la saisie clavier et la renvoyer comme un str.

```
>>> rep = raw_input("Entrer un nombre : ")
Entrer un nombre : 47
>>> rep
'47'
>>> rep == 47
False
>>> type(rep)
<type 'str'>
>>> x = float(rep) ; x
47.0
>>> type(x)
<type 'float'>
>>> x == 47
True
```

- Le formatage des sorties écran peut se faire avec l'opérateur `%`

```
formatString % v ou (v1, v2, ...)
```

```
>>> x = 1.2e-3
>>> print "la variable %s a pour valeur: %8.3E" % ("x", x)
la variable x a pour valeur: 1.200E-03
```

- La chaîne de formatage est identique au standard `printf` du C :

`%.df`	flottant, d décimales
`%.de`	scientifique, d décimales
`%n.de`	flottant scientifique (n caractères au total, dont d décimales)
`%s`	chaîne de caractères
`%d`	format entier
`%g`	choisit le format le plus approprié

```
>>> print "nom: %s, age:%4d, taille:%5.2f m" % ("Smith", 42, 1.73)
nom: Smith, age: 42, taille: 1.73 m
```

- Des fonctionnalités plus avancées sont possibles avec la méthode `format` de la classe `str`, ou avec la fonction intrinsèque `format`.

4.5 Fonctions et appel de fonctions

Il existe plusieurs manière de définir une fonction. Nous présentons ici la première.

L'expression

```
lambda x1,....,xn: ....
```

définit une fonction anonyme prenant n paramètres. Elle peut-être appliquée à n arguments.

```
  (lambda x,y: x+y) (2,3) --> 5
 (lambda f,x: f(x+2)) (lambda x:2*x, 5) --> 14

  (lambda (x,y): x+y) ((2,3)) --> 5    # fonction à 1 argument de type
tuple

>>> (lambda s: " ".join(s.split())) ("toto     aime maman")
'toto aime maman'
```

Deuxième méthode : **Définition et appel des fonctions**

- Définition : mot clef `def` ... terminé par le caractère ':'
  - le corps de la fonction est indenté d'un niveau (bloc indenté),
  - les objets définis dans le corps de la fonction sont locaux à la fonction,
  - une fonction peut renvoyer des objets avec le mot clef `return`.
- Appel : nom de la fonction suivi des parenthèses (...)

```
>>> q = 0
>>> def divide(a,b):
     q = a // b
     r = a - q*b
     return q,r

>>> x, y = divide(5,2)
>>> q, x, y
(0, 2, 1)
```

  - La variable q est définie en dehors de la fonction (valeur = 0)
  - L'opérateur `//` effectue une division entière
  - La variable q définie dans la fonction est locale !
- Passage des arguments par référence
  - à l'appel de la fonction, les arguments (objets) sont transmis par références
  - un argument donné comme un objet non mutable ne peut pas être modifié par la fonction
  - un argument donné comme un objet mutable peut être modifié par la fonction

```
>>> def f0(a, b):
     a = 2
     b = 3
     print "f0 : a,b=%d,%d" % (a,b)
     return

>>> a = 0; b = 1
>>> print a, b
(0, 1)
>>> f0(a, b)
f0 : a,b=2,3
>>> print a, b
(0, 1)

>>> def f1(a):
     for i,e in enumerate(a):
       a[i] = 2.*e
     return

>>> a = [1,2,3]
>>> id(a)
34280296
>>> f1(a)
>>> id(a)
34280296
>>> print a
[2.0, 4.0, 6.0]
```

- Arguments positionnels
  - à l'appel de la fonction, les arguments passés sont des objets
  - chaque objet correspond au paramètre de même position (même rang).

```
>>> def f2(a, b, c):
     print "a : %d, b : %d, c : %d" % (a,b,c)
     return

>>> f2(0) ...TypeError: f2() takes exactly 3 arguments (1 given)
>>> f2(0, 1) ...TypeError: f2() takes exactly 3 arguments (2 given)
>>> f2(0, 1, 2)
a : 0, b : 1, c : 2
```

- Arguments nommés
  - à l'appel, les arguments passés sont des affectations des noms des paramètres
  - l'ordre de passage des arguments nommés est indifférent
  - très souvent utilisé avec des paramètres ayant des valeurs par défaut.

```
>>> def f3(a, b=2, c=2):
      print "a : %d, b : %d, c : %d" % (a,b,c)
      return

>>> f3(0)
a : 0, b : 1, c : 2
>>> f3(0,c=5)
a : 0, b : 2, c : 5
>>> f3(0, c=5, b=4)
a : 0, b : 4, c : 5
```

- Arguments multiples avec le caractère `*`
  - utilisé quand on ne connaît pas à l'avance le nombre d'arguments qui seront passés à une fonction

```
>> def f4 (*x):
      print u" arg. ", x
      return
>> print "% -10s" % "f4 (1 ,2 ,3):", f4 (1 ,2 ,3)
>> L = [1 ,2 ,3]
>> print "% -10s" % "f4(L):", f4(L)
>> print "% -10s" % "f4 (*L):", f4 (*L)

f4(1,2,3): arg. (1, 2, 3)
f4(L): arg. ([1, 2, 3],)
f4(*L): arg. (1, 2, 3)
```

- Arguments multiples : caractères `**`
  - utilisé pour transmettre un nombre quelconque d'arguments nommés
  - la fonction reçoit un objet de type dict
  - utilise le caractère spécial `**`

```
>> def f5 (** x):
      print u" argument reçu par f5 : ", x
      return
>> print "% -20s" % "f5(a=1, b=2, c=3) : ", f5(a=1, b=2, c =3)

f5(a=1, b=2, c=3): argument reçu par f5 : {'a': 1, 'c': 3, 'b': 2}
```

La seconde méthode utilise la définition suivante

```
def f(x1,...xn):
    ''description de ''
    x = 1
    y = x + x1
    return x + y

r = f(e1,...,en) # appel de f, résultat dans r
```

- Une fonction peut être récursive

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

- Par défaut, une fonction retourne la valeur `None` de type none.
- Une variable affectée dans une fonction est considée comme locale à la fonction. Les variables locales sont réallouées à chaque appel

```
>>> x=1
>>> def f():
...     y=x  # y est locale, x est ...
...     x=2  # ... locale, donc x est lue avant son initialisation
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'x' referenced before assignment

>>> x=1
>>> def g():
...     y=x      # x est globale, y est locale
...     print y
...
>>> g()
1
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined

>>> x=1
>>> def h():
...     x=2
...     y=x
...
>>> h()
>>> x=1
>>> def h():
...     x=2
...     y=x
...     print y
...
```

```
>>> h()
2
>>> print x
1
```

- Les variables ou fonctions globales référencées dans une fonction prennent leur plus récente valeur, déterminée à l'exécution (liaison dynamique)

```
>>> x=1
>>> def f(u):
...    return u+x
...
>>> f(2)
3
>>> x=10
>>> f(2)
12
>>> def g(x):
...   return x*2
...
>>> def f(x):
...    return g(x)+1
...
>>> f(1)
3
>>> def g(x):
...    return x*3
...
>>> f(1)
4
```

- Un paramètre peut avoir une valeur par défaut. Les paramètres formels peuvent être nommés (passage par nom ou par position)

```
def incr(x,delta=1): return x+delta

y = incr(2)
z = incr(2,10)
t = incr(delta=3, x=5)
```

4.6 Représentation en compréhension

Deux fonctionnelles (fonctions d'ordre supérieur) sont principalement utilisées pour construire des listes:

- Map.

  `map(function, sequence)` appelle la `function(item)` pour chacun des membres de la sequence et renvoie une liste des valeurs de retours. Par exemple, pour calculer des cubes

```
>>> map(lambda x: x*x*x, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
>>>
```

Plus d'une sequence peut être passée; la fonction doit alors avoir autant d'arguments que le nombres de sequences et est appellée avec le nombre correspondant d'item de chaque sequence (ou `None` si une sequence est plus courte qu'une autre). Si `None` est passé pour la fonction, une fonction retournant son (ses) argument(s) est substituée.

En combinant ces deux cas spéciaux, on voit que `map(None, list1, list2)` est un moyen simple de transformer une paire de listes en liste de paires. Par exemple :

```
>>> seq = range(8)
>>> map(None, seq, map(lambda x: x*x, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
>>>
```

- Filter.

  `filter(function, sequence)` renvoie une sequence (du même type, si possible) consistant en les éléments de la sequence pour lesquels `function(item)` est vrai. Par exemple, pour calculer quelques nombres premiers

```
>>> filter(lambda x: x%2 != 0 and x%3 != 0, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
>>>
```

Une liste (ou plus généralement une collection) se construit en composant des appels à map et filter, donnant lieu à des expressions peu lisibles. La notation proche de la définition d'ensembles en compréhension existe pour créer de tels objets.

5 Classes et objets

5.1 Introduction

En Python, tous les objets manipulés sont des objets au sens de la *Programmation orientée objet*, c'est à dire des entités regroupant :

- des données, appelés *attributs* ou *variables* d'*instance* de l'objet
- des fonctions, appelées *méthodes* de l'objet

Par exemple, les entiers, les flottants, les listes, les tuples, les chaînes de caractères sont des objets. Parmi les attributs d'un objet de la classe liste, on peut citer ses éléments; parmi les méthodes, on peut citer les fonctions `append()`, `extend()`, ...

Seules les méthodes sont habilitées à manipuler les données d'un objet, ce que l'on traduit en disant que les données de l'objet sont *encapsulées* dans l'objet.

- Le type d'un objet détermine :
  - les valeurs possibles de l'objet
  - les opérations que l'objet supporte

– la signification des opérations supportées.

```
>>> a = 2 + 3 # addition d'entiers
>>> a
5
>>> 2*a, 2.*a # mult. dans N, R
(10, 10.0)
>>> 2/3, 2./3. # divis. dans N, R
(0, 0.6666666666666666)
>>>[1, 2]+[5, 6] # concat. des listes
[1, 2, 5, 6]
>>> 2*[5, 6] # n concat\'enations
[5, 6, 5, 6]
>>> [5, 6]/2 # pas d\'efini !
...unsupported operand type(s) for /
'list' and 'int'
```

- Un objet dont le contenu peut être changé est dit mutable (non mutable sinon).

```
>>> a = 'Un str est non-mutable !'
>>> a[0]
'U'
>>> a[0] = 'u'
TypeError: 'str' object does
not support item assignment
>>> a = [1, 2, 3] # liste: mutable
>>> a[1]
2
>>> a[1] = 2.3
>>> a
[1, 2.3, 3]
```

- Une classe rassemble dans un ensemble un collection de données et de fonctions. Une **classe** décrit la structure commune (attributs et opérations) d'une famille d'objets et permet de créer des objets ayant cette structure (appelés instances de la classe). À côté des nombreuses classes définies par défaut par Python (par exemple `int`, `float`, `list`, ...), il est possible de définir de nouvelles classes.

  Une classe introduit un nouveau type décrivant les opérations disponbiles sur une famille d'objets. Une opération prend toujours un argument (nommé par convention `self`) désignant l'objet sur lequelle elle s'applique. Une classe permet de créer des objets (ses instances).

- Pour connaître la liste des attributs et méthodes d'un objet, on invoque la primitive `dir` avec comme argument le nom de l'objet. `dir(•)` permet d'afficher "ce qu'il y a dans" • (classe, objet, module...)

- On peut utiliser toutes les méthodes d'une classe sur un objet instance de la classe : `objet.methode(...)`.

```
>>> L1 = [4, 2, 1, 3] # L1 est un objet liste (instance de la classe liste)
>>> type(L1)
<type 'list'>
>>> dir(L1)
['__add__',..., 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> dir(list)
['__add__',..., 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> L1.sort() ; L1 # objet.m\'ethode_de_la_classe
[1, 2, 3, 4]
>>> L1.append(5) ; L1 # objet.m\'ethode_de_la_classe
[1, 2, 3, 4, 5]
>>> L1.reverse() ; L1 # objet.m\'ethode_de_la_classe
[5, 4, 3, 2, 1]
```

5.2 Un exemple

Considérons le problème suivant. Soit la fonctions à paramètres suivante

$$y(t) = v_o t - \frac{1}{2} g t^2.$$

La manière la plus simple de coder cette fonction en Python est

```
def y(t,v0):
   g=9.81
   return v0*t-0.5*g*t**2
```

Nous souhaitons pouvoir faire un calcul numérique de la dérivée de cette fonction. Une approximation de la dérivée est fournie par

$$f'(x) \equiv \frac{f(x+h) - f(x)}{h}.$$

Un exemple d'utilisation de cette approximation est donné ci-dessous

```
>>> def diff(f,x,h=1E-5):
        return (f(x+h)-f(x))/h
>>> def h(t):
        return t**4+4*t
>>> dh=diff(h,0.1)
```

Malheureusement, avec une telle approche, nous ne pouvons pas appliquer `diff` à `y` car cette fonction a deux paramètres.

Solution 1

- modifier `diff` : ce n'est pas une bonne solution car on doit avoir une méthode générale
- modifier `y` pour qu'elle ne dépende pas de v_0. On a deux possibilités

- ```
  def y1(t):
      v0=9
      g=9.81
      return v0*t-0.5*g*t**2

  def y2(t):
      v0=10
      g=9.81
      return v0*t-0.5*g*t**2
  ```
- Utiliser des variables globales

  ```
  >>> v0=9
  >>> def y(t,v0):
       g=9.81
       return v0*t-0.5*g*t**2
  >>> v0=9
  >>> r1=y(t)
  >>> v0=10
  >>> r2=y(t)
  ```

Le défaut principal de cette solution est qu'il est nécessaire de définir une multitude de fonctions pour chaque valeur de v_0. Un remède est d'utiliser la notion de classes. Une classe contient des données et des fonctions pour les manipuler. Dans l'exemple précédent, les données sont les v_0 et g, et une fonction (`valeur`) pour les manipuler.

```
class Y:
   def __init__(self,v0):       # \
     self.v0=v0                 #  >  constructeur
     self.g=9.81                # /

   def valeur (self,t):
     return self.v0*t-0.5*self.g*t**2
```

Le . permet d'accèder aux éléments d'une classe.

Pour utiliser cette classe, on effectue

```
>>> y=Y(3)
>>> v=y.valeur(0.1)
>>> print y.v0
```

y est une variable utilisateur qui s'appelle une *instance* de la classe. Les fonctions définies dans des classes s'appellent des *méthodes*. Les variables définies à l'intérieur des classes s'appellent des *attributs*.

La variable *self* : `self` est une variable qui contient la nouvelle *instance* (variable) à construire. Ainsi, on a l'équivalence

$$\texttt{y=Y(3)} \iff \texttt{Y.\_\_init\_\_(y,3)}$$

De même, on a

$$\texttt{valeur=y.valeur(0.1)} \iff \texttt{valeur=Y.valeur(y,0.1)}$$
```

La variable `self` suit quelques règles

- chaque classe doit contenir un argument `self`
- `self` représente une instance arbitraire de la classe
- pour utiliser des attributs ou des méthodes de la classe, on utilise `self.-----`.

Extension des classes On peut compléter au fur et à mesure les classes. Par exemple, si l'on souhaite faire une fonction pour afficher la formule codée, on peut réaliser

```
class Y:
   def __init__(self,v0):       # \
     self.v0=v0                 #  >  constructeur
     self.g=9.81                # /

   def valeur (self,t):
     return self.v0*t-0.5*self.g*t**2

   def formule (self):
     return 'v0*t-0.5*g*t**2; v0=%g'%self.v0
```

```
>>> y=Y(5)
>>> print y.formule()
v0*t-0.5*g*t**2; v0=5
```

Maintenant que nous avons notre fonction encoder dans une classe, nous pouvons voir comment l'utiliser avec la fonction `diff`

```
>>> y1=Y(1)
>>> y2=Y(1.5)
>>> dy1dt=diff(y1.valeur,0.1)
>>> dy2dt=diff(y2.valeur,0.2)
>>> dy3dt=diff(Y(-3).valeur,0.1)
```

Classe équivalente sans constructeur

```
class Y2:
   def valeur (self t, v0=None):
     if v0 is not None:
        self.v0=0
     g=9.81
     return self.v0*t-0.5*g*t**2
```

Ce code peut générer des erreurs comme on le voit ci-dessous

```
>>> y=Y2()
>>> print y.v0
print y.v0
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-8d14b60d2916> in <module>()
----> 1 print y.v0
```

```
AttributeError: Y2 instance has no attribute 'v0'
>>> v=y.valeur(0.1,5)
>>> print y.v0
5
>>> print v
0.45095
```

Il faut alors faire de la *gestion des erreurs* avec la commande `hasattr` ou la commande `try`

1. Commande `hasattr`. On rajoute le code suivant

```
class Y2:
   def valeur(self,t,v0=None):
     if v0 is not None:
       self.v0=v0
     g=9.81
     if not hasattr(self,'v0'):
       print 'vous ne pouvez pas appeler valeur(t) sans faire appel a ...
              valeur(t,v0)'
       return None
     return self.v0*t-0.5*g*t**2

>>> y=Y2()
>>> v=y.valeur(0.1)
vous ne pouvez pas appeler valeur(t) sans faire appel a valeur(t,v0)
```

2. Commande `try`. Dans ce cas, on a

```
class Y2:
   def valeur(self,t,v0=None):
     if v0 is not None:
       self.v0=v0
     g=9.81
     try:
       valeur=self.v0*t-0.5*g*t**2
     except AttributeError:
        msg='vous ne pouvez pas appeler valeur(t) sans faire appel a ...
             valeur(t,v0)'
        raise TypeError(msg)
     return valeur

>>> y=Y2()
>>> v=y.valeur(0.1)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-e37bc49328b2> in <module>()
----> 1 v=y.valeur(0.1)

<ipython-input-7-37351e22016a> in valeur(self, t, v0)
```

```
        8         except AttributeError:
        9             msg='vous ne pouvez pas appeler valeur(t) sans faire
                          appel a valeur(t,v0)'
   ---> 10            raise TypeError(msg)
       11         return valeur
       12

   TypeError: vous ne pouvez pas appeler valeur(t) sans faire appel a
              valeur(t,v0)
```

5.3 Classes et opérations spéciales

Soient C une classe et c,c1,c2 des instances de C:

opérateur	déclaration	exemple d'appel	remarque
constructeur	`__init__(self, args)`	C(args)	
appel de fonction	`__call__(self,args)`	c(args)	objet vu comme une fonction
conversion en str	`__str__(self)`	str(c)	appel implicite par print
addition	`__add__(self, other)`	c1 + c2	idem sub, mul, div, pow
addition inverse	`__radd__(self, other)`	o + c	idem rsub, rmul, ...
incrémentation	`__iadd__(self, other)`	c1 += c2	idem isub, imul, idiv
lecture à un index	`__getitem__(self,index)`	c[i]	
écriture à un index	`__setitem__(self,index, val)`	c[i] = v	
longueur	`__len__(self)`	len(c)	
égalité	`__eq__(self, other)`	c1 == c2	idem gt, ge, lt, le, ne
conversion en bool	`__bool__(self)`	bool(c)	appel implicite par if, while
représentation	`__repr__(self)`	repr(c)	chaîne évaluable

Nous avons déjà vu l'utilisation de la commande `.__init__`. Lors de l'utilisation de la classe `Y`, on a vu qu'il était nécessaire de faire la suite de commandes

```
>>> y=Y(3)
>>> y.valeur(0.1)
```

Cela nécessite donc deux commandes. Il y a un autre moyen plus concis pour réaliser ces opérations. On utilise la fonction `.__call__`.

```
class Y:
   -----
   -----
   def __call__(self,t):
     return self.v0*t-0.5*self.g*t**2
```

Alors, lors de l'utilisation, il suffit de faire

```
>>> y=Y(5)
>>> y(2)  # <=> y.valeur(2)
>>> dydt=diff(y,0.1)
```

On a rendu l'objet "appelable". Pour savoir si cet état existe dans un objet, on peut utiliser

```
if callable(a):
```

La méthode `.__str__` permet à un objet d'être directement utiliser avec la commande `print`. Dans la version actuelle de la classe `Y`, un résultat de la commande `print` donne

```
>>> y=Y(3)
>>> print y
<__main__ Y instance at 0x35ef82>
```

Pour modifier cet état, on définit

```
def __str__(self):
   return 'v0*t-0.5*g*t**2, v0=%g' % self.v0

>>> print y
v0*t-0.5*g*t**2, v0=3
```

Toujours dans la version actuelle de la classe `Y`, il n'est pas possible d'ajouter, de soustraire, de multiplier ou de diviser deux objets de la classe. Il existe des méthodes pour définir ces opérations

$a+b$	`a.__add__(b)`	$a==b$	`a.__eq__(b)`
$a-b$	`a.__sub__(b)`	$a>b$	`a.__gt__(b)`
$a*b$	`a.__mul__(b)`	$a\geq b$	`a.__ge__(b)`
a/b	`a.__div__(b)`	$a<b$	`a.__lt__(b)`
$a**b$	`a.__pow__(b)`	$a\leq b$	`a.__le__(b)`
		$a!=b$	`a.__ne__(b)`

Exemple d'utilisation avec des vecteurs en dimension 2

```
class Vec2D:
  def __init__(self,x,y):
    self.x=x
    self.y=y
  def __add__(self,other):
    return Vec2D(self.x+other.x,self.y+other.y)
  def __mul__(self,other):
    return self.x*other.x+self.y*other.y
  def __str__(self):
    return '(%g,%g)'%(self.x,self.y)
```

Variables privées Considérons que nous souhaitions réaliser une classe qui gère un compte bancaire.

```
class compte:
    def __init__(self,nom,numero,montant_init):
        self.nom=nom
        self.numero=numero
        self.etat=montant_init
    def depot(self,montant):
        self.etat += montant
    def retrait(self,montant):
```

```
        self.etat -= montant
    def affiche(self):
        s='%s, %s, etat : %s'%(self.nom,self.numero,self.etat)
        print s
```

```
>>> a1=compte('Arthur','13589',2000)
>>> a1.depot(500)
>>> a1.retrait(1200)
>>> a1.affiche()
Arthur, 13589, etat : 1300
```

Cette définition d'un compte pose un problème. Les attributs de la classe compte ne sont pas protégés. N'importe quel utilisateur peut modifier l'état du compte

```
>>> a1.etat=0
>>> a1.affiche()
Arthur,13589,etat : 0
```

Il n'y a pas de vrai système de variables privées en Python. Pour protéger les attributs, on préfixe les noms de variables (dont on souhaite qu'elles soient privées) par _ et on rajoute des méthode propres à la classe pour modifier ces attributs. C'est une convention respectée par la majorité des codes Python : un nom préfixé par un tiret bas doit être vu comme une partie non publique. On a alors une nouvelle version "protégée"

```
class compte:
    def __init__(self,nom,numero,montant_init):
        self._nom=nom
        self._numero=numero
        self._etat=montant_init
    def depot(self,montant):
        self._etat += montant
    def retrait(self,montant):
        self._etat -= montant
    def obtenir_etat(self):
        return self._etat
    def affiche(self):
        s='%s, %s, etat : %s'%            (self.nom,self.numero,self.etat)
        print s
```

```
>>> a1=compte('Arthur','13589',2000)
>>> a1.depot(500)
>>> a1.retrait(1200)
>>> print a1.obtenir_etat()
1300
>>> a1.etat=0
>>> print a1.obtenir_etat()
1300
```

5.4 Programmation orientée objet

Nous parlons ici d'*héritage et hierarchie de classes*. Comme dans une famille biologique, il y a les classes parents et les classes enfants. Les classes enfants héritent des classes parents

- par exemple du patrimoine génétique
- `Vecteur2D` $\subset$ `Vecteur3D`
- $\mathbb{R} \subset \mathbb{C}$

Une classe parent s'appelle *classe de base* ou *superclasse*. Une classe enfant s'appelle une *sous classe* ou *classe dérivée*.

Supposons que l'on souhaite définir une classe `ligne` qui code les fonctions affine $y = c_0 + c_1 x$.

```
class ligne:
  def __init__(self,c0,c1):
    self.c0=c0
    self.c1=c1
  def __call__(self,x):
    return self.c0+self.c1*x
  def table(self,L,R,n):
    S=' '
    import numpy as np
    for x in np.linspace(L,R,n):
      y=self(x)
      S += '%12g %12g\n'%(x,y)
    return S

>>> l=ligne(-1,1)
>>> print l.table(0.,1.,11)
           0           -1
         0.1         -0.9
         0.2         -0.8
         0.3         -0.7
         0.4         -0.6
         0.5         -0.5
         0.6         -0.4
         0.7         -0.3
         0.8         -0.2
         0.9         -0.1
           1            0
>>> print l.table(0.,1.,9)
           0           -1
       0.125       -0.875
        0.25        -0.75
       0.375       -0.625
         0.5         -0.5
       0.625       -0.375
        0.75        -0.25
       0.875       -0.125
           1            0
```

On souhaite maintenant créer des paraboles de la forme $y = c_0 + c_1 x + c_2 x^2$. Une classe parable peut être construire comme classe dérivée de la classe `ligne`

```
class parabole(ligne):
  def __init__(self,c0,c1,c2):
    ligne.__init__(self,c0,c1)
    self.c2=c2
  def __call__(self,x):
    return ligne.__call__(self,x)+self.c2*x**2
```

La méthode `table` de la classe `ligne` est automatiquement incluse dans la classe `parabole`.

Test sur les types

```
>>> l=ligne(-1,1)
>>> isinstance(l,ligne)
True
>>> isinstance(l,parabole)
False
```

Une ligne n'est pas une parabole

```
>>> p=parabole(-1,0,10)
>>> isinstance(p,parabole)
True
>>> isinstance(p,ligne)
True
```

Une parabole est une ligne.

Exemple

```
class Complexe:
  def __init__(self,r,i):
    self.re = r
    self.im = i
  def __add__(self,c):
    return Complexe(self.re+c.re, self.im+c.im)
  def __iadd__(self, c):
    self.re += c.re
    self.im += c.im
    return self      # c1 += c2 === c1 = c1.__iadd__(c2)
  def __str__(self): return str(self.re)+"+"+str(self.im)+"j"
```

6 Structures de contrôle avancées

6.1 Exceptions

Une exception est levée pour indiquer une situation anormale, comme l'appel d'une fonction en dehors de son domaine de définition. Par exemple, 1/0 lève l'exception ZeroDivisionError. La levée d'une exception termine l'exécution du code à moins qu'elle ne soit récupérée:

```
try
  x = 1/y
  print x   # n'est jamais execute si y = 0
except ZeroDivisionError:
  print 'erreur'  # si y = 0
except (ValueError, TypeError):
  print 'autre erreur'
except:
  print 'erreur inconnue'
```

Une exception peut être levée manuellement par `raise <exc>`.

```
def racine(n):
  if n < 0:
    raise ValueError
  return n ** 0.5
```

Une exception est une instance d'une classe héritant de `Exception`. Elle peut donc comporter des attributs et opérations.

```
def racine(n):
  if n < 0:
    raise ValueError(n)
  return n ** 0.5

try:
  racine(-1)
except ValueError as e:
  print 'erreur:', e.args     # affiche erreur: (-1,)
```

Il est possible de définir de nouvelles exceptions: une exception est une instance d'une classe héritant de `Exception`.

```
class MyError(Exception):
  def __init__(self, value):
    self.value = value
  def __str__(self):
    return repr(self.value)

try:
  raise MyError(1)
except MyError as e:
  print 'erreur:', e.value    # affiche 'erreur: 1'
```

6.2 Itérateurs

La boucle `for i in o: ...` permet de balayer tout objet itérable, c'est-à-dire définissant l'opération `__iter__(self)`. Cette opération doit renvoyer un *itérateur* définissant l'opération `next(self)`. Cette opération renvoie un objet (la valeur prise par i) et modifie l'état de l'itérateur pour passer à l'élément suivant. L'exception `StopIteration` est levée si l'appel à next est impossible. La boucle `for` est donc équivalente à:

```
it = o.__iter__
```

```
try:
  while True:
    i = it.next()
    ...
except StopIteration:
  pass
```

Exemple 1

```
class upto:
  def __init__(self,n):
    self.n = n
    self.i = 0
  def __iter__(self): return self

  def next(self):
    if self.i == self.n:
      raise StopIteration
    else:
      self.i = self.i + 1
      return self.i

for i in upto(10):
  print i  # affiche 1,2,...,10

list(upto(10)) --> [1,2,...,10]

u = upto(10)
list(u) --> [1,2,...,10]
list(u) --> []  # u a été modifié => usage unique
```

Pour contourner le problème, il `__iter__` doit retourner un nouvel objet:

```
class upto:
  def __init__(self,n):
    self.n = n

  def __iter__(self):
    it = upto(self.n)
    it.i = 0
    return it

  def next(self):  # non utilisable sur upto(_)
    if self.i == self.n:
      raise StopIteration
    else:
      self.i = self.i + 1
      return self.i

for i in upto(10):
```

```
  print i  # affiche 1,2,...,10

list(upto(10)) --> [1,2,...,10]

u = upto(10)
list(u) --> [1,2,...,10]
list(u) --> [1,2,...,10]
```

6.3 Générateurs

Un générateur est une fonction (`def`) contenant le mot clé `yield`. Elle est itérable (on peut l'utiliser dans une boucle). Les valeurs énumérées sont celles des expressions suivant `yield`.

```
>>> def fibo(n):
      x, y = 0, 1
      for k in range(n+1):
         yield x
         x, y = y, x + y

>>> type(f)
<type 'generator'>
>>> f = fibo(10) # initialisation du g\'en\'erateur
>>> next(f) # valeur suivante
0
>>> next(f) # valeur suivante
1
>>> next(f)
1
>>> next(f)
2
>>> [u for u in fibo(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
a
```

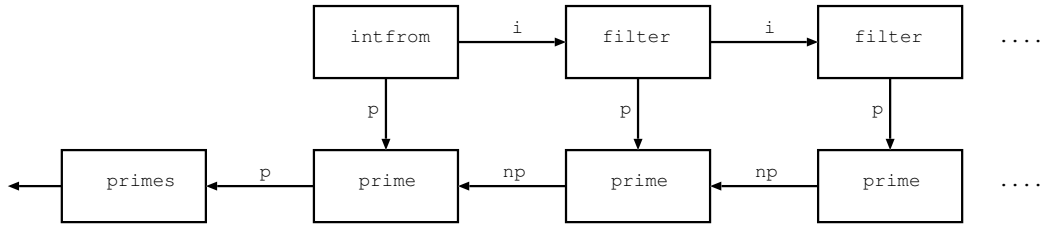
Générateur d'entiers Le générateur `xrange` ne crée pas de liste.

```
for i in xrange(10): # entiers de 0 à 9
  print i
for i in xrange(1,10,2):  # entiers impairs de 1 à 9
  print i
list(xrange(5)) --> [0,1,2,3,4]
```

Générateur en compréhension

```
for i in (u * u for u in xrange(5) if u > 2):  # 9, 16
  print i
```

Application : Crible d'Ératosthène. La séquence de nombres premiers est générée par l'application de filtres successifs ne retenant que les entiers non multiples du premier qu'ils recoivent. Ces filtres sont appliqués à la séquence des entiers à partir de 2. Dans le code ci-dessous, le générateur `prime` reçoit un filtre en paramètre. Il lit un un élément, nombre premier, construit le couple (filter,prime) suivant et transmet les nombres premiers produits par son successeur.




```
def filter(p,g):
  for i in g:
    if i % p != 0:
      yield i

def prime(f):
  p = f.next()
  yield p
  for np in prime(filter(p,f)):
    yield np

def intfrom(n):
  while True:
    yield n
    n = n + 1

def primes():
  for p in prime(intfrom (2)):
    yield p

for p in primes(): print p
```

Remarque En pratique, la longueur de la séquence générée est limitée par la taille de la pile.

References

[1] *Python scientifique - ENS Paris*, URL: `http://python-prepa.github.io`

[2] V. Le Goff, *Apprenez à programmer avec Python*, URL: `http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-python`

[3] J.-J. Charles, E. Ducasse, *Cours Python Bordeaux*, URL: `http://savoir.ensam.eu/moodle/mod/folder/view.php?id=10242`a

[4] H.-P. Langtangen, *A Primer on Scientific Programming with Python*, Texts in Computational Science and Engineering, Volume 6, (2014), Springer, ISBN: 978-3-642-54958-8.

[5] A. Casamayou-Boucau, P. Chauvin, G. Connan, *Programmation en Python pour les mathématiques*, Dunod, (2013)