
Analyse scientifique avec Python

Version Février 2019

Yannick Copin

01/02/19, 11:11

Table des matières

1	Introduction	1
1.1	Pourquoi un module d'analyse scientifique ?	1
1.2	Pourquoi Python ?	1
1.3	Informations pratiques 2019	2
1.4	Index et recherche	3
2	Initiation à Python	5
2.1	Introduction	6
2.2	Types de base	7
2.3	Structures de programmation	8
2.4	Les chaînes de caractères	9
2.5	Objets itérables	11
2.6	Fonctions	13
2.7	Bibliothèques et scripts	14
2.8	Exceptions	16
2.9	Classes	18
2.10	Entrées-sorties	20
2.11	Fonctionnalités avancées	21
3	Bibliothèque standard	25
3.1	Gestion des arguments/options de la ligne de commande	25
3.2	Pickle : sérialisation des données	26
3.3	<i>Batteries included</i>	27
3.4	<i>Text/Graphical User Interfaces</i>	27
4	Bibliothèques numériques de base	29
4.1	Numpy	29
4.2	Scipy	39
4.3	Matplotlib	40
5	Bibliothèques scientifiques avancées	47
5.1	Pandas et xarray	47
5.2	Astropy	60
5.3	Autres bibliothèques scientifiques	61
6	Développer en Python	63
6.1	Le zen du Python	63
6.2	Développement piloté par les tests	65
6.3	Outils de développement	67
6.4	Python 2 vs. Python 3	71

7	Références supplémentaires	73
7.1	Documentation générale	73
7.2	Listes de liens	73
7.3	Livres libres	74
7.4	Cours en ligne	74
8	Exemples	77
8.1	Mean power (fonction, argparse)	77
8.2	Formes (POO)	78
8.3	Cercle circonscrit (POO, argparse)	80
8.4	Matplotlib	88
9	Exercices	93
9.1	Introduction	93
9.2	Manipulation de listes	94
9.3	Programmation	95
9.4	Manipulation de tableaux (<code>arrays</code>)	96
9.5	Méthodes numériques	97
9.6	Visualisation (matplotlib)	97
9.7	Mise en oeuvre de l'ensemble des connaissances acquises	99
9.8	Exercices en vrac	100
10	Annales d'examen	103
10.1	Simulation de chute libre (partiel nov. 2014)	103
10.2	Examen janvier 2015	103
11	Projets	105
11.1	Projets de physique	105
11.2	Projets astrophysiques	110
11.3	Projets divers	112
11.4	Projets statistiques	115
11.5	Projets de visualisation	115
12	Démonstration Astropy	117
12.1	Fichiers FITS	117
12.2	Tables	121
12.3	Quantités et unités	123
12.4	Calculs cosmologiques	124
13	Pokémon Go ! (démonstration Pandas/Seaborn)	127
13.1	Lecture et préparation des données	127
13.2	Accès aux données	129
13.3	Quelques statistiques	129
13.4	Visualisation	130
14	Méthode des rectangles	139
15	Fizz Buzz	141
16	Algorithme d'Euclide	143
17	Crible d'Ératosthène	145
18	Carré magique	147
19	Suite de Syracuse	149
20	Flocon de Koch	151
21	Jeu du plus ou moins	155

22 Animaux	157
23 Particules	161
24 Jeu de la vie	171
25 <i>Median Absolute Deviation</i>	175
26 Distribution du <i>pull</i>	177
27 Quadrature	179
28 Zéro d'une fonction	181
29 Quartet d'Anscombe	183
30 Suite logistique	187
31 Ensemble de Julia	189
32 Trajectoire d'un boulet de canon	191
33 Équation d'état de l'eau	193
34 Solutions aux exercices	197
35 Examen final, Janvier 2015	199
35.1 Exercice	199
35.2 Le problème du voyageur de commerce	200
35.3 Correction	202
Bibliographie	203

Version Master de Physique Fondamentale, Université Lyon 1 du 01/02/19, 11 :11

Auteur Yannick Copin <ipnl.in2p3.fr>

1.1 Pourquoi un module d'analyse scientifique ?

- Pour *générer* ses données, p.ex. simulations numériques, contrôle d'expériences.
- Pour *traiter* ses données, i.e. supprimer les artefacts observationnels.
- Pour *analyser* ses données, i.e. extraire les quantités physiques pertinentes, p.ex. en ajustant un modèle.
- Pour *visualiser* ses données, et appréhender leur richesse multi-dimensionnelle.
- Pour *présenter* ses données, p.ex. générer des figures prêtes à publier.

Ce module s'adresse donc avant tout aux futurs expérimentateurs, phénoménologues ou théoriciens voulant se frotter à la réalité des observations.

1.2 Pourquoi Python ?

Les principales caractéristiques du langage **Python** :

- syntaxe simple et lisible : langage pédagogique et facile à apprendre et à utiliser ;
- langage interprété : utilisation interactive ou script exécuté ligne à ligne, pas de processus de compilation ;
- haut niveau : typage dynamique, gestion active de la mémoire, pour une plus grande facilité d'emploi ;
- multi-paradigme : langage impératif et/ou orienté objet, selon les besoins et les capacités de chacun ;
- logiciel libre et ouvert, largement répandu (multi-plateforme) et utilisé (forte communauté) ;
- riche bibliothèque standard : *Batteries included* ;
- riche bibliothèque externe : de nombreuses bibliothèques de qualité, dans divers domaines (y compris scientifiques), sont déjà disponibles.

L'objectif est bien d'apprendre *un seul* langage de haut niveau, permettant tout aussi bien des analyses rapides dans la vie de tous les jours – quelques lignes de code en interactif – que des programmes les plus complexes (projets de plus de 100 000 lignes).

Liens :

- [Getting Started](#)
- [Python Advocacy](#)

1.3 Informations pratiques 2019

- Atelier *Analyse scientifique avec Python*
- [Cours en ligne](#)
- Responsable : Yannick Copin <ipn1.in2p3.fr>, Bureau 420 de l'IPNL (4 rue Fermi)

Calendrier

Toutes les séances ont lieu au bâtiment [Ariane](#).

Date	TD	Salle
Lun. 04/02/2019	10h-12h30 14h-16h30	05
Mar. 05/02	10h-12h30 14h-16h30	05
Jeu. 07/02	10h-12h30 14h-16h30	01 (projets)

Participants

Nom	Mail (prenom.nom+)	Statut
vivien.duboisdendien	etu.univ-lyon1.fr	M2 Astro
marion.farcy	etu.univ-lyon1.fr	M2 Astro
maxime.rey	etu.univ-lyon1.fr	M2 Astro
antoine.rocher	etu.univ-lyon1.fr	M2 Astro
deborah.polderman	etu.univ-lyon1.fr	M2 PAMMCO
fabien.rondepierre	etu.univ-lyon1.fr	M2 PAMMCO
antonin.pardon	univ-lyon1.fr	ILM
floriane.perrier	univ-lyon1.fr	ILM

Installations locales

Avertissement : le cours utilise dorénavant **Python 3**.

Si des programmes ou des bibliothèques Python (p.ex. `ipython`) manquent sur votre ordinateur (p.ex. en salle Ariane), il est relativement aisé de les installer localement à l'aide du gestionnaire d'installation `pip`.

- Compléter votre `~/.bashrc` :

```
export PATH=$PATH:$HOME/.local/bin/
export PYTHONPATH=$HOME/.local/lib/python3.6/site-packages/
```

- Installer p.ex. `ipython` :

```
pip3 install --user ipython
```

Si vous avez le contrôle de votre ordinateur, il peut être préférable d'utiliser le gestionnaire de paquets du système (p.ex. `synaptic` sur Ubuntu).

1.4 Index et recherche

- `genindex`
- `search`

Table des matières

- *Initiation à Python*
 - *Introduction*
 - *Installation*
 - *Notions d'Unix*
 - *L'invite de commande*
 - *Types de base*
 - *Structures de programmation*
 - *Les chaînes de caractères*
 - *Indexation*
 - *Sous-liste (slice)*
 - *Méthodes*
 - *Formatage*
 - *Objets itérables*
 - *Fonctions*
 - *Bibliothèques et scripts*
 - *Bibliothèques externes*
 - *Bibliothèques personnelles et scripts*
 - *Exceptions*
 - *Classes*
 - *Entrées-sorties*
 - *Interactif*
 - *Fichiers*
 - *Fonctionnalités avancées*
 - *Arguments anonymes*
 - *Dépaquetage des arguments*
 - *Dépaquetage des itérables*
 - *Décorateurs*
 - *Fonction anonyme*
 - *Éléments passés sous silence*
 - *Python 3.x*

2.1 Introduction

2.1.1 Installation

Cette introduction repose essentiellement sur les outils suivants :

- Python 3.5+ (inclus l'interpréteur de base et la bibliothèque standard) ;
- les bibliothèques scientifiques [Numpy](#) et [Scipy](#) ;
- la bibliothèque graphique [Matplotlib](#) ;
- un interpréteur évolué, p.ex. [ipython](#) ;
- un éditeur de texte évolué, p.ex. [emacs](#), [vi](#), [gedit](#) ou [Atom](#).

Ces logiciels peuvent être installés indépendamment, de préférence sous Linux (p.ex. [Ubuntu](#) ou votre distribution préférée), ou sous Windows ou MacOS. Il existe également des distributions « clés en main » :

- [Anaconda](#) (multi-plateforme) ;
- [Python\(x,y\)](#) (Windows) ;
- [Enthought Canopy](#) (Windows, MacOS, Linux, gratuite pour les étudiants du supérieur).

2.1.2 Notions d'Unix

Les concepts suivants sont supposés connus :

- ligne de commande : exécutables et options ;
- arborescence : chemin relatif (`[./]...`) et absolu (`/...`), navigation (`cd`) ;
- gestion des fichiers (`ls`, `rm`, `mv`) et répertoires (`mkdir`) ;
- gestion des exécutables : `$PATH`, `chmod +x` ;
- gestion des processus : `&`, `Control-c`, `Control-z` + `bg` ;
- variables d'environnement : `export`, `.bashrc`.

Liens :

- [Quelques notions et commandes d'UNIX](#) 
- [Introduction to Unix Study Guide](#)

2.1.3 L'invite de commande

Il existe principalement deux interpréteurs interactifs de commandes Python :

- **python** : interpréteur de base :

```
$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- `Control-d` pour sortir ;
- `help(commande)` pour obtenir l'aide d'une commande ;
- *a priori*, pas d'historique des commandes ni de complétion automatique.

L'interpréteur de base permet également d'interpréter un « script », c.-à-d. un ensemble de commandes regroupées dans un fichier texte (généralement avec une extension `.py`) : `python mon_script.py`

- **ipython** : interpréteur évolué (avec historique et complétion automatique des commandes) :

```
$ ipython
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
Type "copyright", "credits" or "license" for more information.
```

(suite sur la page suivante)

(suite de la page précédente)

```
IPython 5.3.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
```

In [1]:

- Control-d pour sortir ;
- Tab pour la complétion automatique ;
- Haut et Bas pour le rappel des commandes ;
- aide ipython : `object?` pour une aide sur un objet, `object??` pour une aide plus complète (au niveau source) ;
- commandes *magic* (voir `%magic`) :
 - `%run mon_script.py` pour exécuter un script dans l'interpréteur,
 - `%debug` pour lancer le mode débogage interactif *post-mortem*,
 - `%cpaste` pour coller et exécuter un code préformaté.

Liens :

- [Tutorial](#)
- [IPython Tips & Tricks](#)

2.2 Types de base

- None (rien)
- Chaînes de caractères : `str`
 - Entre (simples ou triples) apostrophes ' ou guillemets " : `'Calvin'`, `"Calvin'n'Hobbes"`, `'''Deux\nlignes'''`, `"""Pourquoi? demanda-t-il."""`
 - Conversion : `str(3.2)`
- Types numériques :
 - Booléens `bool` (vrai/faux) : `True`, `False`, `bool(3)`
 - Entiers `int` (pas de valeur limite explicite, correspond au moins au long du C) : `-2`, `int(2.1)`, `int("4")`
 - Réels `float` (entre $\pm 1.7e\pm 308$, correspond au double du C) : `2.`, `3.5e-6`, `float(3)`
 - Complexes `complex` : `1+2j` (sans espace), `5.1j`, `complex(-3.14)`, `complex('j')`

```
>>> 5 / 2      # Division réelle par défaut dans Python 3.x
2.5
>>> 6 // 2.5   # Division euclidienne explicite
2.0
>>> 6 % 2.5    # Reste de la division euclidienne
1.0
>>> (1 + 2j)**-0.5 # Puissance entière, réelle ou complexe
(0.5688644810057831-0.3515775842541429j)
```

- Objets itérables :
 - Listes `list` : `['a', 3, [1, 2], 'a']`
 - Listes immuables `tuple` : `(2, 3.1, 'a', [])` (selon les conditions d'utilisation, les parenthèses ne sont pas toujours nécessaires)
 - Listes à clés `dict` : `{'a':1, 'b':[1, 2], 3:'c'}`
 - Ensembles non ordonnés d'éléments uniques `set` : `{1, 2, 3, 2}`

```
>>> l = ['a', True] # Définition d'une liste
>>> x, y = 1, 2.5   # Affectations multiples via tuples (les parenthèses ne sont pas
↪ nécessaires)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> list(range(5))    # Liste de 5 entiers commençant par 0
[0, 1, 2, 3, 4]
>>> 1 + [x, y]        # Concaténation de listes
['a', True, 1, 2.5]
>>> {2, 1, 3} | {1, 2, 'a'} # Union d'ensembles (non-ordonnés)
{'a', 1, 2, 3}
```

Attention : en Python 3, `range()` n'est plus un constructeur de liste, mais un *itérateur*, qui doit être converti en liste explicitement (équivalent à `xrange` de Python 2) :

```
>>> range(3)          # Itérateur
range(0, 3)
>>> list(range(3))    # Liste
[0, 1, 2]
```

- `type(obj)` retourne le type de l'objet, `isinstance(obj, type)` teste le type de l'objet.

```
>>> type(1)
<type 'list'>
>>> isinstance(1, tuple)
False
```

Liens :

- [The Floating Point Guide](#)
- [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

2.3 Structures de programmation

- Les blocs sont définis par l'**indentation** (en général par pas de quatre espaces) ¹.

Avertissement : Évitez autant que possible les caractères de tabulation, source de confusion. Configurez votre éditeur de texte pour qu'il n'utilise que des espaces.

- Une instruction par ligne *en général* (ou instructions séparées par `;`).
- Les commentaires commencent par `#`, et s'étendent jusqu'à la fin de la ligne.
- **Expression booléenne** : une condition est une expression s'évaluant à `True` ou `False` :
 - **False** : test logique faux (p.ex. `3 == 4`), valeur nulle, chaîne vide (`' '`), liste vide (`[]`), etc.
 - **True** : test logique vrai (p.ex. `2 + 2 == 4`), toute valeur ou objet non nul (et donc s'évaluant par défaut à `True` *sauf exception*)
 - Tests logiques : `==`, `!=`, `>`, `>=`, etc.

Attention : Ne pas confondre « `=` » (affectation d'une variable) et « `==` » (test logique d'égalité).

- Opérateurs logiques : `and`, `or`, `not`

```
>>> (5 >= 6) or (not 3 > 4)
True
>>> x = 3; 0 < x <= 5    # Conditions chaînées
True
```

ou `from __future__ import braces :-)`

- Opérateur ternaire (**PEP 308**) : *value if condition else altvalue*, p.ex.

```
>>> y = x**0.5 if (x > 0) else 0 # Retourne sqrt(max(x, 0))
```

- **Expression conditionnelle** : *if condition: ... [elif condition2: ...] [else: ...]*, p.ex. :

```
if (i > 0):      # Condition principale
    print("positif")
elif (i < 0):   # Condition secondaire (si nécessaire)
    print("négatif")
else:          # Cas final (si nécessaire)
    print("nul")
```

- **Boucle for** : *for element in iterable:*, s'exécute sur chacun des éléments d'un objet itérable :

```
>>> for val in ['un', (2, 3), 4]: # Itération sur une liste de 3 éléments
...     print(val)
un
(2, 3)
4
```

- **continue** : interrompt l'itération courante, et reprend la boucle à l'itération suivante,
- **break** : interrompt complètement la boucle.

Note : la logique des boucles Python est assez différente des langages C[++] / fortran, pour lesquels l'itération porte sur les *indices* plutôt que sur les éléments eux-mêmes.

- **Boucle while** : *while condition:* se répète tant que la *condition* est vraie, ou jusqu'à une sortie explicite avec **break**.

Attention : aux boucles infinies, dont la condition d'exécution reste invariablement vraie (typiquement un critère de convergence qui n'est jamais atteint). On peut toujours s'en protéger en testant *en outre* sur un nombre maximal (raisonnable) d'itérations :

```
niter = 0
while (error > 1e-6) and (niter < 100):
    error = ... # A priori, error va décroître, et la boucle s'interrompre...
    niter += 1  # ... mais on n'est jamais assez prudent!
if niter == 100: # Ne pas oublier de tester l'absence de convergence!!!
    print("Erreur de convergence!")
```

Note : Il n'y a pas en Python d'équivalent natif à l'instruction **switch** du C.

Exercices :

*Intégration : méthode des rectangles *, Fizz Buzz *, PGCD : algorithme d'Euclide ***

2.4 Les chaînes de caractères

2.4.1 Indexation

Les chaînes de caractères sont des objets *itérables* – c.-à-d. constitués d'éléments (ici les caractères) sur lesquels il est possible de « boucler » (p.ex. avec **for**) – et *immuables* – c.-à-d. dont les éléments individuels ne peuvent pas être modifiés intrinsèquement.

Note : Comme en C[++], l'indexation en Python commence à 0 : le 1er élément d'une liste est l'élément n°0, le 2e est le n°1, etc. Les n éléments d'une liste sont donc indexés de 0 à $n-1$.

```
>>> alpha = 'abcdefghijklmnopqrstuvwxyz'
>>> len(alpha)
26
>>> alpha[0]      # 1er élément (l'indexation commence à 0)
'a'
>>> alpha[-1]     # = alpha[26-1=25], dernier élément (-2: avant-dernier, etc.)
'z'
```

2.4.2 Sous-liste (*slice*)

Des portions d'une chaîne peuvent être extraites en utilisant des `slice` (« tranches »), de notation générique `[start=0]:[stop=len][:step=1]`. P.ex.

```
>>> alpha[3:7]     # De l'élément n°3 (inclus) au n°7 (exclu), soit 7-3=4 éléments
'defg'
>>> alpha[:3]      # Du n°0 (défaut) au n°3 (exclu), soit 3 éléments
'abc'
>>> alpha[-3:]     # Du n°26-3=23 (inclus) au dernier inclus (défaut)
'xyz'
>>> alpha[3:9:2]   # Du n°3 (inclus) au n°9 (exclu), tous les 2 éléments
'dfh'
>>> alpha[::5]     # Du 1er au dernier élément (défauts), tous les 5 éléments
'afkpuz'
```

2.4.3 Méthodes

Comme la plupart des objets en Python, les chaînes de caractères disposent de nombreuses fonctionnalités – appelées « méthodes » en POO (Programmation Orientée Objet) – facilitant leur manipulation :

```
>>> enfant, peluche = "Calvin", 'Hobbes'      # Affectations multiples
>>> titre = enfant + ' et ' + peluche; titre    # += Concaténation de chaînes
'Calvin et Hobbes'
>>> titre.replace('et', '&')                  # Remplacement de sous-chaînes (→ nouvelle chaîne)
'Calvin & Hobbes'
>>> titre                                     # titre est immuable et reste inchangé
'Calvin et Hobbes'
>>> ' & '.join(titre.split(' et '))          # Découpage (split) et jonction (join)
'Calvin & Hobbes'
>>> 'Hobbes' in titre                         # in: Test d'inclusion
True
>>> titre.find("Hobbes")                     # str.find: Recherche de sous-chaîne
10
>>> titre.center(30, '-')
'-----Calvin et Hobbes-----'
>>> dir(str)                                # Liste toutes les méthodes des chaînes
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__
↳ format__', '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__
↳ ', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__
↳ new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__
↳ sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
↳ 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
↳ 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
↳ 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition
↳ ', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
↳ 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```


(suite de la page précédente)

2.4.4 Formatage

Le système de *formatage* permet un contrôle précis de la conversion de variables en chaînes de caractères. Il s'appuie essentiellement sur la méthode `str.format()` :

```
>>> "{0} a {1} ans".format('Calvin', 6) # args énumérés
'Calvin a 6 ans'
>>> "{} a {} ans".format('Calvin', 6)    # Raccourci
'Calvin a 6 ans'
>>> "{nom} a {age} ans".format(nom='Calvin', age=6) # kwargs
'Calvin a 6 ans'
>>> pi = 3.1415926535897931
>>> "{x:f} {x:.2f} {y:f} {y:g}".format(x=pi, y=pi*1e9) # Options de formatage
'3.141593 3.14 3141592653.589793 3.14159e+09'
```

`print()` affiche à l'écran (plus spécifiquement la sortie standard) la conversion d'une variable en chaîne de caractères :

```
>>> print("Calvin and Hobbes\nScientific progress goes 'boink'!")
Calvin and Hobbes
Scientific progress goes 'boink'!
>>> print("{0:2d} fois {1:2d} font {2}".format(3, 4, 3*4)) # Formatage et affichage
3 fois 4 font 12
```

Exercice :

*Tables de multiplication **

2.5 Objets itérables

Les chaînes de caractères, listes, tuples et dictionnaires sont les objets itérables de base en Python. Les listes et dictionnaires sont *modifiables* (« *mutables* ») – leurs éléments constitutifs peuvent être changés à la volée – tandis que chaînes de caractères et les tuples sont *immuables*.

— Accès indexé : conforme à celui des chaînes de caractères

```
>>> l = list(range(1, 10, 2)); l # De 1 (inclus) à 10 (exclu) par pas de 2
[1, 3, 5, 7, 9]
>>> len(l) # Nb d'éléments dans la liste (i varie de 0 à 4)
5
>>> l[0], l[-2] # 1er et avant-dernier élément (l'indexation commence à 0)
(1, 7)
>>> l[5] # Erreur: indice hors-bornes
IndexError: list index out of range
>>> d = dict(a=1, b=2) # Création du dictionnaire {'a':1, 'b':2}
>>> d['a'] # Accès à une entrée via sa clé
1
>>> d['c'] # Erreur: clé inexistante!
KeyError: 'c'
>>> d['c'] = 3; d # Ajout d'une clé et sa valeur
{'a': 1, 'c': 3, 'b': 2}
>>> # Noter qu'un dictionnaire N'est PAS ordonné!
```

— Sous-listes (*slices*) :

```
>>> l[1:-1]          # Du 2e ('1') *inclus* au dernier ('-1') *exclu*
[3, 5, 7]
>>> l[1:-1:2]        # Idem, tous les 2 éléments
[3, 7]
>>> l[::2]           # Tous les 2 éléments (*start=0* et *stop=len* par défaut)
[1, 5, 9]
```

— Modification d'éléments d'une liste (chaînes et tuples sont **immuables**) :

```
>>> l[0] = 'a'; l      # Remplacement du 1er élément
['a', 3, 5, 7, 9]
>>> l[1:2] = ['x', 'y']; l # Remplacement d'éléments par *slices*
['a', 'x', 5, 'y', 9]
>>> l + [1, 2]; l      # Concaténation (l reste inchangé)
['a', 'x', 5, 'y', 9, 1, 2]
>>> l += [1, 2]; l     # Concaténation sur place (l est modifié)
['a', 'x', 5, 'y', 9, 1, 2]
>>> l.append('z'); l   # Ajout d'un élément en fin de liste
['a', 'x', 5, 'y', 9, 1, 2, 'z']
>>> l.extend([-1, -2]); l # Extension par une liste
['a', 'x', 5, 'y', 9, 1, 2, 'z', -1, -2]
>>> del l[-6:]; l      # Efface les 6 derniers éléments de la liste
['a', 'x', 5, 'y']
```

Attention : à la modification des objets *mutables* :

```
>>> l = [0, 1, 2]
>>> m = l; m          # m est un *alias* de la liste l: c'est le même objet
[0, 1, 2]
>>> id(l); id(m); m is l
171573452             # id({obj}) retourne le n° d'identification en mémoire
171573452             # m et l ont le même id:
True                 # ils correspondent donc bien au même objet en mémoire
>>> l[0] = 'a'; m     # puisque l a été modifiée, il en est de même de m
['a', 1, 2]
>>> m = l[:]          # copie de tous les éléments de l dans une *nouvelle* liste m
↪ (clonage)
>>> id(l); id(m); m is l
171573452
171161228            # m a un id différent de l: il s'agit de 2 objets distincts
False                # (contenant éventuellement la même chose!)
>>> del l[-1]; m     # les éléments de m n'ont pas été modifiés
['a', 1, 2]
```

— Liste en compréhension : elle permet la construction d'une liste à la volée

```
>>> [ i**2 for i in range(5) ] # Carré de tous les éléments de [0, ..., 4]
[0, 1, 4, 9, 16]
>>> [ 2*i for i in range(10) if (i%3 != 0) ] # Compréhension conditionnelle
[2, 4, 8, 10, 14, 16]
>>> [ 10*i+j for i in range(3) for j in range(4) ] # Double compréhension
[0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23]
>>> [ [ 10*i+j for i in range(3) ] for j in range(4) ] # Compréhensions imbriquées
[[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]]
>>> { i: i**2 for i in range(1, 5) } # Dictionnaire en compréhension
{1: 1, 2: 4, 3: 9, 4: 16}
```

— Utilitaires sur les itérables :

```
>>> humans = ['Calvin', 'Wallace', 'Boule']
```

(suite sur la page suivante)

(suite de la page précédente)

```

>>> for i in range(len(humans)): # Boucle sur les indices de humans
...     print(i, humans[i])      # Accès explicite, pas pythonique :-D
0 Calvin
1 Wallace
2 Boule
>>> for i, name in enumerate(humans): # Boucle sur (indice, valeur) de humans
...     print(i, name)           # Pythonique :-D
0 Calvin
1 Wallace
2 Boule
>>> animals = ['Hobbes', 'Gromit', 'Bill']
>>> for boy, dog in zip(humans, animals): # Boucle simultanée sur 2 listes (ou +)
...     print(boy, 'et', dog)
Calvin et Hobbes
Wallace et Gromit
Boule et Bill
>>> sorted(zip(humans, animals)) # Tri, ici sur le 1er élément de chaque tuple de la
↳ liste
[('Boule', 'Bill'), ('Calvin', 'Hobbes'), ('Wallace', 'Gromit')]

```

Exercices :*Crible d'Ératosthène *, Carré magique ***

2.6 Fonctions

Une fonction est un regroupement d'instructions impératives – assignations, branchements, boucles, etc. – s'appliquant sur des arguments d'entrée. C'est le concept central de la programmation *impérative*.

`def` permet de définir une fonction : `def fonction(arg1, arg2, ..., option1=valeur1, option2=valeur2, ...)`. Les « *args* » sont des arguments nécessaires (c.-à-d. obligatoires), tandis que les « *kwargs* » – arguments de type `option=valeur` – sont optionnels, puisqu'ils possèdent une valeur par défaut. Si la fonction doit retourner une valeur, celle-ci est spécifiée par le mot-clé `return`.

Exemples :

```

1 def temp_f2c(tf):
2     """
3     Convertit une température en d° Fahrenheit `tf` en d° Celsius.
4
5     Exemple:
6     >>> temp_f2c(104)
7     40.0
8     """
9
10    tc = (tf - 32.)/1.8      # Fahrenheit → Celsius
11
12    return tc

```

Dans la définition d'une fonction, la première chaîne de caractères (appelé *docstring*) servira de documentation pour la fonction, accessible de l'interpréteur via p.ex. `help(temp_f2c)`, ou `temp_f2c?` sous `ipython`. Elle se doit d'être tout à la fois pertinente, concise et complète. Elle peut également inclure des exemples d'utilisation (*doctests*, voir *Développement piloté par les tests*).

```

1 def mean_power(alist, power=1):
2     r"""
3     Retourne la racine `power` de la moyenne des éléments de `alist` à
4     la puissance `power`:
5
6     .. math:: \mu = (\frac{1}{N} \sum_{i=0}^{N-1} x_i^p)^{1/p}
7
8     `power=1` correspond à la moyenne arithmétique, `power=2` au *Root
9     Mean Squared*, etc.
10
11     Exemples:
12     >>> mean_power([1, 2, 3])
13     2.0
14     >>> mean_power([1, 2, 3], power=2)
15     2.160246899469287
16     """
17
18     # *mean* = (somme valeurs**power / nb valeurs)**(1/power)
19     mean = (sum( val ** power for val in alist ) / len(alist)) ** (1 / power)
20
21     return mean

```

Il faut noter plusieurs choses importantes :

- Python est un langage à typage *dynamique*, p.ex., le type des arguments d'une fonction n'est pas fixé *a priori*. Dans l'exemple précédent, `alist` peut être une `list`, un `tuple` ou tout autre itérable contenant des éléments pour lesquels les opérations effectuées – somme, exponentiation, division par un entier – ont été préalablement définies (p.ex. des entiers, des complexes, des matrices, etc.) : c'est ce que l'on appelle le *duck-typing*², favorisant le polymorphisme des fonctions ;
- le typage est *fort*, c.-à-d. que le type d'une variable ne peut pas changer à la volée. Ainsi, `"abra" + "cadabra"` a un sens (concaténation de chaînes), mais pas `3 + "cochons"` (entier + chaîne) ;
- la définition d'une fonction se fait dans un « espace parallèle » où les variables ont une portée (*scope*) locale³. Ainsi, la variable `s` définie *dans* la fonction `mean_power` n'interfère pas avec le « monde extérieur » ; inversement, la définition de `mean_power` ne connaît *a priori* rien d'autre que les variables explicitement définies dans la liste des arguments ou localement.

Exercice :

*Suite de Syracuse (fonction) **

2.7 Bibliothèques et scripts

2.7.1 Bibliothèques externes

Une bibliothèque (ou module) est un code fournissant des fonctionnalités supplémentaires – p.ex. des fonctions prédéfinies – à Python. Ainsi, le module `math` définit les fonctions et constantes mathématiques usuelles (`sqrt()`, `pi`, etc.)

Une bibliothèque est « importée » avec la commande `import module`. Les fonctionnalités supplémentaires sont alors accessibles dans l'espace de noms *module* via `module.fonction` :

```

>>> sqrt(2)                # sqrt n'est pas une fonction standard de python
NameError: name 'sqrt' is not defined
>>> import math             # Importe tout le module 'math'
>>> dir(math)               # Liste les fonctionnalités de 'math'

```

(suite sur la page suivante)

If it looks like a duck and quacks like a duck, it must be a duck.
La notion de « portée » est plus complexe, je simplifie...

(suite de la page précédente)

```
[ '__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'trunc']

>>> math.sqrt(math.pi)           # Les fonctionnalités sont disponibles sous 'math'
1.7724538509055159
>>> import math as M             # Importe 'math' dans l'espace 'M'
>>> M.sqrt(M.pi)
1.7724538509055159
>>> from math import sqrt, pi    # Importe uniquement 'sqrt' et 'pi' dans l'espace courant
>>> sqrt(pi)
1.7724538509055159
```

Avertissement : Il est possible d'importer toutes les fonctionnalités d'une bibliothèque dans l'espace de noms courant :

```
>>> from math import *          # Argh! Pas pythonique :-()
>>> sqrt(pi)
1.7724538509055159
```

Cette pratique est cependant fortement *déconseillée* du fait des confusions dans les espaces de noms qu'elle peut entraîner :

```
>>> from cmath import *
>>> sqrt(-1)                   # Quel sqrt: le réel ou le complexe?
```

Nous verrons par la suite quelques exemples de modules de la *Bibliothèque standard*, ainsi que des *Bibliothèques numériques de base* orientées analyse numérique.

Exercice :

*Flocon de Koch (programmation récursive) ****

2.7.2 Bibliothèques personnelles et scripts

Vous pouvez définir vos propres bibliothèques en regroupant les fonctionnalités au sein d'un même fichier *monfichier.py*.

- Si ce fichier est importé (p.ex. `import monfichier`), il agira comme une bibliothèque.
- Si ce fichier est exécuté – p.ex. `python ./monfichier.py` – il agira comme un *script*.

Attention : Toutes les instructions d'un module qui ne sont pas encapsulées dans le `__main__` (voir plus bas) sont interprétées et exécutées lors de l'import du module. Elles doivent donc en général se limiter à la définition de variables, de fonctions et de classes (en particulier, éviter les affichages ou les calculs longs).

Un code Python peut donc être :

- un module – s'il n'inclut que des définitions mais pas d'instruction exécutable en dehors d'un éventuel `__main__`⁴;
- un exécutable – s'il inclut un `__main__` ou des instructions exécutables;
- ou les deux à la fois.

⁴ Parfois prononcé *dunder main* (*dunder* désigne le `__`).

Exemple :

Le code `mean_power.py` peut être importé comme une bibliothèque (p.ex. `import mean_power`) dans un autre code Python, ou bien être exécuté depuis la ligne de commande (p.ex. `python mean_power.py`), auquel cas la partie `__main__` sera exécutée.

- `#!` (Hash-bang) : la première ligne d'un script définit l'interpréteur à utiliser⁵ :

```
#!/usr/bin/env python3
```

- Un fichier incluant des caractères non-ASCII (p.ex. caractères accentués, ou symboles UTF tels que \pm) doit définir le système d'encodage, généralement `utf-8` :

```
# -*- coding: utf-8 -*-
```

Notez que les noms de variables, fonctions, etc. doivent être purement ASCII⁶ (a-zA-Z0-9_). De manière générale, favorisez la langue anglaise (variables, commentaires, affichages).

- `"""doc"""` : la chaîne de documentation de la bibliothèque (*docstring*, **PEP 257**), qui sera utilisée comme aide en ligne du module (`help(mean_power)`), doit être la 1^{re} instruction du script.
- `if __name__ == '__main__':` permet de séparer le `__main__` (c.-à-d. le corps du programme, à exécuter lors d'une utilisation en script) des définitions de fonctions et classes, permettant une utilisation en module.

2.8 Exceptions

Lorsqu'il rencontre une erreur dans l'exécution d'une instruction, l'interpréteur Python génère (`raise`) une erreur (*Exception*), de nature différente selon la nature de l'erreur : `KeyError`, `ValueError`, `AttributeError`, `NameError`, `TypeError`, `IOError`, `NotImplementedError`, `KeyboardInterrupt`, etc. La levée d'une erreur n'est cependant pas nécessairement fatale, puisque Python dispose d'un mécanisme de *gestion des erreurs*.

Il est d'usage en Python d'utiliser la philosophie EAFP (Easier to Ask for Forgiveness than Permission)⁷ : plutôt que de tester explicitement toutes les conditions de validité d'une instruction, on « tente sa chance » d'abord, quitte à gérer les erreurs *a posteriori*. Cette gestion des exceptions se fait par la construction `try ... except`.

```
1 def lireEntier():
2     while True:
3         chaine = input('Entrez un entier: ') # Lecture du clavier → str
4         try:
5             # La conversion en type entier génère `ValueError` si nécessaire
6             return int(chaine)
7         except ValueError: # Gestion de l'exception ValueError
8             print("{} n'est pas un entier".format(chaine))
```

```
>>> lireEntier()
Entrez un entier: toto
'toto' n'est pas un entier
Entrez un entier: 3,4
'3,4' n'est pas un entier
Entrez un entier: 4
4
```

Il s'agit d'une fonctionnalité des *shells* d'Unix, pas spécifique à Python.
En fait, Python 3 gère nativement les caractères `Unicode` :

```
>>>  $\alpha$ ,  $\beta$  = 3, 4
>>> print(" $\alpha^2 + \beta^2 =$ ",  $\alpha^2 + \beta^2$ )
 $\alpha^2 + \beta^2 = 25$ 
```

Par opposition au LBYL (Look Before You Leap) du C/C++, basé sur une série *exhaustive* de tests *a priori*.

Dans l'élaboration d'un programme, gérez explicitement les erreurs que vous auriez pu tester *a priori* et pour lesquels il existe une solution de repli, et laissez passer les autres (ce qui provoquera éventuellement l'interruption du programme).

Danger : Évitez à tout prix les `except nus`, c.-à-d. ne spécifiant pas la ou les exceptions à gérer, car ils intercepteraient alors *toutes* les exceptions, y compris celles que vous n'aviez pas prévues ! Trouvez l'erreur dans le code suivant :

```
y = 2
try:
    x = z          # Copie y dans x
    print("Tout va bien")
except:
    print("Rien ne va plus")
```

Vos procédures doivent également générer des exceptions (*documentées*) – avec l'instruction `raise Exception()` – si elles ne peuvent conclure leur action, à charge pour la procédure appelante de les gérer si besoin :

```
1 def diff_sqr(x, y):
2     """
3     Return x**2 - y**2 for x >= y, raise ValueError otherwise.
4
5     Exemples:
6     >>> diff_sqr(5, 3)
7     16
8     >>> diff_sqr(3, 5)
9     Traceback (most recent call last):
10    ...
11    ValueError: x=3 < y=5
12    """
13
14    if x < y:
15        raise ValueError("x={} < y={}".format(x, y))
16
17    return x**2 - y**2
```

Avant de se lancer dans un calcul long et complexe, on peut vouloir tester la validité de certaines hypothèses fondamentales, soit par une structure `if ... raise`, ou plus facilement à l'aide d'`assert` (qui, si l'hypothèse n'est pas vérifiée, génère une `AssertionError`) :

```
1 def diff_sqr(x, y):
2     """
3     Returns x**2 - y**2 for x >= y, AssertionError otherwise.
4     """
5
6     assert x >= y, "x={} < y={}".format(x, y) # Test et msg d'erreur
7     return x**2 - y**2
```

Note : La règle générale à retenir concernant la gestion des erreurs :

Fail early, fail often, fail better !

Exercice :

*Jeu du plus ou moins (exceptions) **

2.9 Classes

Un objet est une entité de programmation, disposant de ses propres états et fonctionnalités. C'est le concept central de la [Programmation Orientée Objet](#).

Au concept d'objet sont liées les notions de :

- **Classe** : il s'agit d'un *modèle* d'objet, dans lequel sont définis ses propriétés usuelles. P.ex. la classe `Forme` peut représenter une forme plane caractérisée par sa couleur, et disposant de fonctionnalités propres, p.ex. `change_couleur()` ;
- **Instanciation** : c'est le fait générer un objet concret (une *instance*) à partir d'un modèle (une classe). P.ex. `rosie = Forme('rose')` crée une instance `rosie` à partir de la classe `Forme` et d'une couleur (chaîne de caractères `'rose'`) ;
- **Attributs** : variables internes décrivant l'état de l'objet. P.ex., `rosie.couleur` donne la couleur de la `Forme rosie` ;
- **Méthodes** : fonctions internes, s'appliquant en premier lieu sur l'objet lui-même (`self`), décrivant les capacités de l'objet. P.ex. `rosie.change_couleur('bleu')` change la couleur de la `Forme rosie` ;

Attention : Toutes les méthodes d'une classe doivent au moins prendre `self` – représentant l'objet lui-même – comme premier argument.

- **Surcharge d'opérateurs** : cela permet de redéfinir les opérateurs et fonctions usuels (`+`, `abs()`, `str()`, etc.), pour simplifier l'écriture d'opérations sur les objets. Ainsi, on peut redéfinir les opérateurs de comparaison (`<`, `>=`, etc.) dans la classe `Forme` pour que les opérations du genre `forme1 < forme2` aient un sens (p.ex. en comparant les aires).
- **Héritage de classe** : il s'agit de définir une classe à partir d'une (ou plusieurs) classe(s) parente(s). La nouvelle classe *hérite* des attributs et méthodes de sa (ses) parente(s), que l'on peut alors modifier ou compléter. P.ex. la classe `Rectangle` hérite de la classe `Forme` (elle partage la notion de couleur et d'aire), et lui ajoute des méthodes propres à la notion de rectangle (p.ex. formule explicite de l'aire, étirement).

Exemple de définition de classe

```

1 class Forme:
2
3     """Une forme plane, avec éventuellement une couleur."""
4
5     def __init__(self, couleur=None):
6         """Initialisation d'une Forme, sans couleur par défaut."""
7
8         if couleur is None:
9             self.couleur = 'indéfinie'
10        else:
11            self.couleur = couleur
12
13    def __str__(self):
14        """
15        Surcharge de la fonction `str()` : l'affichage *informel* de
16        l'objet dans l'interpréteur, p.ex. `print(a)` sera résolu comme
17        `a.__str__()`
18
19        Retourne une chaîne de caractères.
20        """
21
22        return "forme encore indéfinie de couleur {}".format(self.couleur)
23
24    def change_couleur(self, newcolor):
25        """Change la couleur de la Forme."""

```

(suite sur la page suivante)

(suite de la page précédente)

```

26         self.couleur = newcolor
27
28
29     def aire(self):
30         """
31         Renvoie l'aire de la Forme.
32
33         L'aire ne peut pas être calculée dans le cas où la forme n'est
34         pas encore spécifiée: c'est ce que l'on appelle une méthode
35         'abstraite', qui pourra être précisée dans les classes filles.
36         """
37
38         raise NotImplementedError(
39             "ATTENTION: impossible de calculer l'aire d'une forme indéfinie.")

```

Exemple d'héritage de classe

```

1  class Rectangle(Forme):
2      """
3      Un Rectangle est une Forme particulière.
4
5      La classe-fille hérite des attributs et méthodes de la
6      classe-mère, mais peut les surcharger (i.e. en changer la
7      définition), ou en ajouter de nouveaux:
8
9      - la méthode `Rectangle.change_couleur()` dérive directement de
10     `Forme.change_couleur()`;
11     - `Rectangle.__str__()` surcharge `Forme.__str__()`;
12     - `Rectangle.aire()` définit la méthode jusqu'alors abstraite
13     `Forme.aire()`;
14     - `Rectangle.allonger()` est une nouvelle méthode propre à
15     `Rectangle`.
16     """
17
18     def __init__(self, longueur, largeur, couleur=None):
19         """
20         Initialisation d'un Rectangle longueur × largeur, sans couleur par
21         défaut.
22         """
23
24         # Initialisation de la classe parente (nécessaire pour assurer
25         # l'héritage)
26         Forme.__init__(self, couleur)
27
28         # Attributs propres à la classe Rectangle
29         self.longueur = longueur
30         self.largeur = largeur
31
32     def __str__(self):
33         """Surcharge de `Forme.__str__()`."""
34
35         return "rectangle {x}, de couleur {}".format(
36             self.longueur, self.largeur, self.couleur)
37
38     def aire(self):
39         """
40         Renvoie l'aire du Rectangle.
41
42         Cette méthode définit la méthode abstraite `Forme.area()`,

```

(suite sur la page suivante)

(suite de la page précédente)

```
43     pour les Rectangles uniquement.
44     """
45
46     return self.longueur * self.largeur
47
48     def allonger(self, facteur):
49         """Multiplie la *longueur* du Rectangle par un facteur"""
50
51         self.longueur *= facteur
```

Note : Il est traditionnel de commencer les noms de classes avec une majuscule (**Forme**), et les noms d'instances de classe (les variables) avec une minuscule (**rosie**).

Exemples

Formes (POO), Cercle circonscrit (POO, argparse)

Études de cas

- `turtle.Vec2D`
- `fractions.Fraction`

Exercices :

*Animaux (POO/TDD) *, Jeu de la vie (POO) ***

2.10 Entrées-sorties

2.10.1 Interactif

Comme nous avons pu le voir précédemment, l'affichage à l'écran se fait par `print`, la lecture du clavier par `input`.

2.10.2 Fichiers

La gestion des fichiers (lecture et écriture) se fait à partir de la fonction `open()` retournant un objet de type `file object` :

```
1  # ===== ÉCRITURE =====
2  outfile = open("carres.dat", 'w') # Ouverture du fichier "carres.dat" en écriture
3  for i in range(1, 10):
4      outfile.write("{} {} \n".format(i, i**2)) # Noter la présence du '\n' (non-automatique)
5  outfile.close()                               # Fermeture du fichier (nécessaire)
6
7  # ===== LECTURE =====
8  infile = open("carres.dat") # Ouverture du fichier "carres.dat" en lecture
9  for line in infile:         # Boucle sur les lignes du fichier
10     if line.strip().startswith('#'): # Ne pas considérer les lignes "commentées"
11         continue
12     try:                         # Essayons de lire 2 entiers sur cette ligne
```

(suite sur la page suivante)

(suite de la page précédente)

```

13     x, x2 = [ int(tok) for tok in line.split() ]
14 except ValueError:         # Gestion des erreurs
15     print("Cannot decipher line '{}'".format(line))
16     continue
17 print("{}**3 = {}".format(x, x**3))

```

2.11 Fonctionnalités avancées

Cette brève introduction à Python se limite à des fonctionnalités relativement simples du langage. De nombreuses fonctionnalités plus avancées n'ont pas encore été abordées⁸.

2.11.1 Arguments anonymes

Il est possible de laisser libre *a priori* le nombre et le nom des arguments d'une fonction, traditionnellement nommés `args` (arguments nécessaires) et `kwargs` (arguments optionnels). P.ex. :

```

>>> def f(*args, **kwargs):
...     print("args:", args)
...     print("kwargs:", kwargs)
>>> f()
args: ()
kwargs: {}
>>> f(1, 2, 3, x=4, y=5)
args: (1, 2, 3)
kwargs: {'y': 5, 'x': 4}

```

Attention : Cela laisse une grande flexibilité dans la signature de la fonction, mais au prix d'une d'une très mauvaise lisibilité de la signature de la fonction. *À utiliser avec parcimonie...*

2.11.2 Dépaquetage des arguments

Il est possible de dépaqueter les `[kw]args` d'une fonction à la volée à l'aide de l'opérateur `[*]*`. Ainsi, avec la même fonction `f` précédemment définie :

```

>>> my_args = (1, 2, 3)
>>> my_kwargs = dict(x=4, y=5)
>>> f(my_args, my_kwargs)      # 2 args (1 liste et 1 dict.) et 0 kwarg
args: ((1, 2, 3), {'x': 4, 'y': 5})
kwargs: {}
>>> f(*my_args, **my_kwargs)  # 3 args (1, 2 et 3) et 2 kwargs (x et y)
args: (1, 2, 3)
kwargs: {'x': 4, 'y': 5}

```

À partir de Python 3.5, il est encore plus facile d'utiliser un ou plusieurs de ces opérateurs conjointement aux `[kw]args` traditionnels ([PEP 448](#)), dans la limite où les `args` sont toujours situés *avant* les `kwargs` :

```

>>> f(0, *my_args, 9, **my_kwargs, z=6)
args: (0, 1, 2, 3, 9)
kwargs: {'x': 4, 'z': 6, 'y': 5}

```

Je ne parlerai pas ici des [variables globales](#)...

2.11.3 Dépaquetage des itérables

Il est également possible d'utiliser l'opérateur `*` pour les affectations multiples ([PEP 3132](#)) :

```
>>> a, b, c = 1, 2, 3, 4
ValueError: too many values to unpack (expected 3)
>>> a, *b, c = 1, 2, 3, 4
>>> a, b, c
(1, [2, 3], 4)
```

2.11.4 Décorateurs

Les fonctions (et méthodes) sont en Python des objets comme les autres, et peuvent donc être utilisées comme arguments d'une fonction, ou retournées comme résultat d'une fonction.

```
1 def compute_and_print(fn, *args, **kwargs):
2
3     print("Function: ", fn.__name__)
4     print("Arguments: ", args, kwargs)
5     result = fn(*args, **kwargs)
6     print("Result:   ", result)
7
8     return result
```

Les décorateurs sont des *fonctions* s'appliquant sur une fonction ou une méthode pour en modifier le comportement : elles retournent de façon transparente une version « *décorée* » (augmentée) de la fonction initiale.

```
1 def verbose(fn):           # fonction → fonction décorée
2
3     def decorated(*args, **kwargs):
4         print("Function: ", fn.__name__)
5         print("Arguments: ", args, kwargs)
6         result = fn(*args, **kwargs)
7         print("Result:   ", result)
8
9         return result
10
11     return decorated      # version décorée de la fonction initiale
```

```
>>> verbose_sum = verbose(sum)  # Décore la fonction standard 'sum'
>>> verbose_sum([1, 2, 3])
Function:    sum
Arguments:   ([1, 2, 3],) {}
Result:      6
```

Il est possible de décorer une fonction à la volée lors de sa définition avec la notation `@` :

```
@verbose
def null(*args, **kwargs):
    pass
```

qui n'est qu'une façon concise d'écrire `null = verbose(null)`.

```
>>> null(1, 2, x=3)
Function:    null
Arguments:   (1, 2) {'x': 3}
Result:      None
```

Noter qu'il est possible d'ajouter plusieurs décorateurs, et de passer des arguments supplémentaires aux décorateurs.

Lien

A guide to Python's function decorators

Python dispose de quelques décorateurs natifs d'intérêt pour les méthodes, notamment :

- `staticmethod()` transforme une méthode en une méthode statique, c.-à-d. ne s'appliquant pas à l'instance :

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): # L'instance self n'est pas le 1er argument!
        ...
```

La méthode statique peut alors être invoquée via la classe `C.f()` ou via une instance `C().f()`.

- `classmethod()` transforme une méthode en une méthode de classe, c.-à-d. s'appliquant à la classe plutôt qu'à l'instance :

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): # La classe cls est le 1er argument!
        ...
```

Comme une méthode statique, une méthode de classe peut être invoquée via la classe `C.f()` ou via une instance `C().f()`.

Les méthodes statiques sont souvent d'intérêt plus général, sans être spécifique à une instance particulière : c'est une fonction indépendante stockée à l'intérieur de la classe. Les méthodes de classe sont quant à elles souvent utilisées pour des initialisations alternatives (voir un [exemple d'utilisation et discussion associée](#)).

2.11.5 Fonction anonyme

Il est parfois nécessaire d'utiliser une fonction intermédiaire *simple* que l'on ne souhaite pas définir explicitement et nommément à l'aide de `def`. Cela est possible avec l'opérateur fonctionnel `lambda` *args* : *expression*. P.ex. :

```
>>> compute_and_print(sum, [1, 2]) # Fn nommée à 1 argument
Function:    sum
Arguments:   ([1, 2],), {}
Result:      3
>>> compute_and_print(lambda x, y: x + y, 1, 2) # Fn anonyme à 2 arguments
Function:    <lambda>
Arguments:   (1, 2) {}
Result:      3
```

La définition d'une fonction `lambda` ne peut inclure qu'une seule expression, et est donc contrainte *de facto* à être très simple, généralement pour être utilisée comme argument d'une autre fonction :

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Note : il est possible de « nommer » une fonction anonyme, p.ex. :

```
>>> adder = lambda x, y: x + y
```

Cependant, cela est considéré comme une faute de style, puisque ce n'est justement pas l'objectif d'une fonction anonyme ! Il n'y a p.ex. pas de *docstring* associée.

Voir également : [Functional Programming](#)

2.11.6 Éléments passés sous silence

Il existe encore beaucoup d'éléments passés sous silence :

- `iterator` (`next()`) et `generator` (`yield`);
- gestion de contexte : `with` ([PEP 343](#));
- héritages multiples et méthodes de résolution;
- annotations de fonctions ([PEP 484](#)) et de variables ([PEP 526](#));
- `f-strings` ([PEP 498](#));
- etc.

Ces fonctionnalités peuvent évidemment être très utiles, mais ne sont généralement pas strictement indispensables pour une première utilisation de Python dans un contexte scientifique.

2.11.7 Python 3.x

Pour des raisons historiques autant que pratiques⁹, ce cours présentait initialement le langage Python dans sa version 2. Cependant, puisque le développement actuel de Python (et de certaines de ses bibliothèques clés) se fait maintenant uniquement sur la branche 3.x, qui constitue une remise à plat *non rétrocompatible* du langage, et que la branche 2.x ne sera *a priori* plus supporté au-delà de 2020 ([PEP 466](#)), le cours a été porté sur Python 3 (voir *Python 2 vs. Python 3*).

Python 3 apporte [quelques changements fondamentaux](#), notamment :

- `print()` n'est plus un mot-clé mais une fonction : `print(...)`;
- l'opérateur `/` ne réalise plus la division euclidienne entre les entiers, mais toujours la division *réelle*;
- la plupart des fonctions qui retournaient des itérables en Python 2 (p.ex. `range()`) retournent maintenant des itérateurs, plus légers en mémoire;
- un support complet (mais encore complexe) des chaînes Unicode;
- un nouveau système de formatage des chaînes de caractères (`f-string` du [PEP 498](#) à partir de Python 3.6);
- la fonction de comparaison `cmp` (et la méthode spéciale associée `__cmp__`) n'existe plus¹⁰.

De nombreuses distributions Linux sont encore basées sur Python 2.7 par défaut.
Voir `functools.total_ordering()` pour une alternative.

Table des matières

- *Bibliothèque standard*
- *Gestion des arguments/options de la ligne de commande*
- *Pickle : sérialisation des données*
- Batteries included
- Text/Graphical User Interfaces

Python dispose d'une très riche bibliothèque de modules étendant les capacités du langage dans de nombreux domaines : nouveaux types de données, interactions avec le système, gestion des fichiers et des processus, protocoles de communication (internet, mail, FTP, etc.), multimédia, etc.

- The Python Standard Library (v3.x)
- Python Module of the Week (v3.x)

3.1 Gestion des arguments/options de la ligne de commande

Utilisation de `sys.argv`

Le module `sys` permet un accès direct aux arguments de la ligne de commande, via la liste `sys.argv` : `sys.argv[0]` contient le nom du script exécuté, `sys.argv[1]` le nom du 1er argument (s'il existe), etc. P.ex. :

```
1  # Gestion simplifiée d'un argument entier sur la ligne de commande
2  import sys
3
4  if sys.argv[1:]: # Présence d'au moins un argument sur la ligne de commande
5      try:
6          n = int(sys.argv[1]) # Essayer de lire le 1er argument comme un entier
7      except ValueError:
8          raise ValueError("L'argument '{}' n'est pas un entier"
9                          .format(sys.argv[1]))
10 else:
11     n = 101 # Pas d'argument sur la ligne de commande
              # Valeur par défaut
```

Module argparse

Pour une gestion avancée des arguments et/ou options de la ligne de commande, il est préférable d'utiliser le module `argparse`. P.ex. :

```

1  import argparse
2
3  parser = argparse.ArgumentParser(
4      usage="% (prog)s [-p/--plot] [-i/--input coordfile | x1,y1 x2,y2 x3,y3]",
5      description="Compute the circumscribed circle to 3 points in the plan.")
6  parser.add_argument('coords', nargs='*', type=str, metavar='x,y',
7                      help="Coordinates of point")
8  parser.add_argument('-i', '--input', nargs='?', type=argparse.FileType('r'),
9                      help="Coordinate file (one 'x,y' per line)")
10 parser.add_argument('-p', '--plot', action="store_true", default=False,
11                    help="Draw the circumscribed circle")
12 parser.add_argument('-T', '--tests', action="store_true", default=False,
13                    help="Run doc tests")
14 parser.add_argument('--version', action='version', version=__version__)
15
16 args = parser.parse_args()
```

Cette solution génère automatiquement une aide en ligne, p.ex. :

```

$ python3 circonscrip.py -h
usage: circonscrip.py [-p/--plot] [-i/--input coordfile | x1,y1 x2,y2 x3,y3]

Compute the circumscribed circle to 3 points in the plan.

positional arguments:
  x,y                Coordinates of point

optional arguments:
  -h, --help            show this help message and exit
  -i [INPUT], --input [INPUT]
                        Coordinate file (one 'x,y' per line)
  -p, --plot            Draw the circumscribed circle
  -T, --tests           Run doc tests
  --version             show program's version number and exit
```

3.2 Pickle : sérialisation des données

Le module `pickle` permet la sauvegarde pérenne d'objets python (« sérialisation »).

```

>>> d = dict(a=1, b=2, c=3)
>>> l = ["Calvin", 6, 1.20]
>>> import pickle
>>> pkl = open('archive.pkl', 'wb')          # Overture du fichier en écriture binaire
>>> pickle.dump((d, l), pkl, protocol=-1)    # Sérialisation du tuple (d, l)
>>> pkl.close()                             # *IMPORTANT!* Fermeture du fichier
>>> d2, l2 = pickle.load(open('archive.pkl', 'rb')) # Désérialisation (relecture)
>>> (d == d2) and (l == l2)
True
```

Attention : les pickles ne sont pas un format d'échange de données. Ils sont spécifiques à python, et peuvent dépendre de la machine utilisée. Ils peuvent en outre constituer une faille de sécurité.

3.3 Batteries included

Quelques modules de la bibliothèque standard qui peuvent être d'intérêt :

- `math` : accès aux fonctions mathématiques réelles

```
>>> math.asin(math.sqrt(2) / 2) / math.pi * 180
45.00000000000001
```

- `cmath` : accès aux fonctions mathématiques complexes

```
>>> cmath.exp(cmath.pi * 1j) + 1
1.2246467991473532e-16j
```

- `fractions` : définition des nombres rationnels

```
>>> print(fractions.Fraction(2, 3) + fractions.Fraction(5, 6))
3/2
>>> print(fractions.Fraction(*(3.5).as_integer_ratio()))
7/2
```

- `random` : générateurs de nombres aléatoires

```
>>> random.sample(range(10), 3) # Échantillon de 3 éléments sans remplacement
[9, 1, 6]
>>> random.gauss(0, 1)          # Distribution normale centrée réduite
0.1245612752121385
```

- autres modules numériques et mathématiques ;
- `collections` définit de nouveaux types spécialisés, p.ex. `collections.OrderedDict`, un dictionnaire *ordonné* ;
- `itertools` fournit des générateurs de boucle (*itérateurs*) et de combinatoire :

```
>>> [ ''.join(item) for item in itertools.combinations('ABCD', 2) ]
['AB', 'AC', 'AD', 'BC', 'BD', 'CD']
```

- interactions avec le système :
 - `sys`, `os` : interface système,
 - `shutil` : opérations sur les fichiers (*copy*, *move*, etc.),
 - `subprocess` : exécution de commandes système,
 - `glob` : métacaractères du *shell* (p.ex. `toto?.*`) ;
- expressions rationnelles (ou *regex*) : `re` ;
- `warnings` et `logging` : gestion des avertissements d'exécution et mécanismes de *logging*
- gestion du temps (`time`) et des dates (`datetime`, `calendar`) ;
- fichiers compressés et archives : `gzip`, `bz2`, `zipfile`, `tarfile` ;
- lecture et sauvegarde des données (outre `pickle`) :
 - `pprint` : affichage « amélioré » d'un objet,
 - `csv` : lecture/sauvegarde de fichiers CSV (Comma Separated Values),
 - `ConfigParser` : fichiers de configuration,
 - `json` : *lightweight data interchange format* ;
- lecture d'une URL (p.ex. page web) : `urllib2`.

3.4 Text/Graphical User Interfaces

- TUI (Text User Interface) : `curses`
- GUI (Graphical User Interface) : `Tkinter`,

Bibliothèques externes :

- TUI : `termcolor` (texte coloré ANSI), `blessed` (mise en page)
- GUI : `PyGI` (GTK3), `PyQt` / `pySide` (Qt), `wxPython` (wxWidgets)

Bibliothèques numériques de base

Table des matières

- *Bibliothèques numériques de base*
- *Numpy*
 - *Tableaux*
 - *Création de tableaux*
 - *Manipulations sur les tableaux*
 - *Opérations de base*
 - *Tableaux évolués*
 - *Entrées/sorties*
 - *Sous-modules*
 - *Performances*
- *Scipy*
 - *Tour d'horizon*
 - *Quelques exemples complets*
- *Matplotlib*
 - *pylab vs. pyplot*
 - *Figure et axes*
 - *Sauvegarde et affichage interactif*
 - *Anatomie d'une figure*
 - *Visualisation 3D*

4.1 Numpy

`numpy` est une bibliothèque *numérique* apportant le support efficace de larges tableaux multidimensionnels, et de routines mathématiques de haut niveau (fonctions spéciales, algèbre linéaire, statistiques, etc.).

Note : La convention d'import utilisé dans les exemples est « `import numpy as N` ».

Liens :

- [Numpy User Guide](#)
- [Numpy Reference](#)

4.1.1 Tableaux

Un `numpy.ndarray` (généralement appelé `array`) est un tableau multidimensionnel *homogène* : tous les éléments doivent avoir le même type, en général numérique. Les différentes dimensions sont appelées des *axes*, tandis que le nombre de dimensions – 0 pour un scalaire, 1 pour un vecteur, 2 pour une matrice, etc. – est appelé le *rang*.

```
>>> import numpy as N      # Import de la bibliothèque numpy avec le surnom N
>>> a = N.array([1, 2, 3]) # Création d'un array 1D à partir d'une liste d'entiers
>>> a.ndim                # Rang du tableau
1                        # Vecteur (1D)
>>> a.shape               # Format du tableau: par définition, len(shape)=ndim
(3,)                    # Vecteur 1D de longueur 3
>>> a.dtype               # Type des données du tableau
dtype('int32')          # Python 'int' = numpy 'int32' = C 'long'
>>> # Création d'un tableau 2D de float (de 0. à 12.) de shape 4x3
>>> b = N.arange(12, dtype=float).reshape(4, 3); b
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.],
       [ 9., 10., 11.]])
>>> b.shape               # Nb d'éléments le long de chacune des dimensions
(4, 3)                  # 4 lignes, 3 colonnes
>>> b.size                # Nb *total* d'éléments dans le tableau
12                       # Par définition, size = prod(shape)
>>> b.dtype               # Python 'float' = numpy 'float64' = C 'double'
dtype('float64')
```

Création de tableaux

- `numpy.array()` : convertit une liste d'éléments homogènes ou coercibles

```
>>> N.array([[1, 2],[3., 4.]]) # Liste de listes d'entiers et de réels
array([[ 1.,  2.],
       [ 3.,  4.]])           # Tableau 2D de réels
```

- `numpy.zeros()` (resp. `numpy.ones()` et `numpy.full()`) : crée un tableau de format donné rempli de zéros (resp. de uns et d'une valeur fixe)

```
>>> N.zeros((2, 1))      # Shape (2, 1): 2 lignes, 1 colonne, float par défaut
array([[ 0.],
       [ 0.]])
>>> N.ones((1, 2), dtype=bool) # Shape (1, 2): 1 ligne, 2 colonnes, type booléen
array([[True, True]], dtype=bool)
>>> N.full((2, 2), N.NaN)
array([[ nan,  nan],
       [ nan,  nan]])
```

- `numpy.arange()` : crée une séquence de nombres, en spécifiant éventuellement le *start*, le *end* et le *step* (similaire à `range()` pour les listes)

```
>>> N.arange(10, 30, 5)    # De 10 à 30 (exclu) par pas de 5, type entier par défaut
array([10, 15, 20, 25])
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> N.arange(0.5, 2.1, 0.3) # Accepte des réels en argument, DANGER!
array([ 0.5,  0.8,  1.1,  1.4,  1.7,  2. ])
```

- `numpy.linspace()` (resp. `numpy.logspace()`) : répartition uniforme (resp. logarithmique) d'un nombre fixe de points entre un *start* et un *end* (préférable à `numpy.arange()` sur des réels).

```
>>> N.linspace(0, 2*N.pi, 5) # 5 nb entre 0 et 2π *inclus*, type réel par défaut
array([ 0.,  1.57079633,  3.14159265,  4.71238898,  6.28318531])
>>> N.logspace(-1, 1, 5)      # 5 nb répartis log. entre 10**(±1)
array([ 0.1,  0.31622777,  1.,  3.16227766,  10. ])
```

- `numpy.meshgrid()` est similaire à `numpy.linspace()` en 2D ou plus :

```
>>> # 5 points entre 0 et 2 en "x", et 3 entre 0 et 1 en "y"
>>> x = N.linspace(0, 2, 5); x      # Tableau 1D des x, (5,)
array([ 0.,  0.5,  1.,  1.5,  2. ])
>>> y = N.linspace(0, 1, 3); y      # Tableau 1D des y, (3,)
array([ 0.,  0.5,  1. ])
>>> xx, yy = N.meshgrid(x, y)       # Tableaux 2D des x et des y
>>> xx                               # Tableau 2D des x, (3, 5)
array([[ 0.,  0.5,  1.,  1.5,  2. ],
       [ 0.,  0.5,  1.,  1.5,  2. ],
       [ 0.,  0.5,  1.,  1.5,  2. ]])
>>> yy                               # Tableau 2D des y, (3, 5)
array([[ 0.,  0.,  0.,  0.,  0. ],
       [ 0.5, 0.5, 0.5, 0.5, 0.5],
       [ 1.,  1.,  1.,  1.,  1. ]])
```

- `numpy.mgrid` permet de générer des rampes d'indices (entiers) ou de coordonnées (réels) de rang arbitraire avec une notation évoluée faisant appel aux *Index tricks*. Équivalent à `numpy.linspace()` en 1D et *similaire (mais différent)* à `numpy.meshgrid()` en 2D.

```
>>> N.mgrid[0:4, 1:6:2] # Grille 2D d'indices (entiers)
array([[0, 0, 0],      # 0:4 = [0, 1, 2, 3] selon l'axe 0
       [1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]],
       [[1, 3, 5],      # 1:6:2 = [1, 3, 5] selon l'axe 1
       [1, 3, 5],
       [1, 3, 5],
       [1, 3, 5]])
>>> N.mgrid[0:2*N.pi:5j] # Rampe de coordonnées (réels): 5 nb de 0 à 2π (inclus)
array([ 0.,  1.57079633,  3.14159265,  4.71238898,  6.28318531])
>>> # 3 points entre 0 et 1 selon l'axe 0, et 5 entre 0 et 2 selon l'axe 1
>>> z = N.mgrid[0:1:3j, 0:2:5j]; z
array([[[ 0.,  0.,  0.,  0.,  0. ], # Axe 0 variable, axe 1 constant
       [ 0.5, 0.5, 0.5, 0.5, 0.5],
       [ 1.,  1.,  1.,  1.,  1. ]],
       [[ 0.,  0.5,  1.,  1.5,  2. ], # Axe 0 constant, axe 1 variable
       [ 0.,  0.5,  1.,  1.5,  2. ],
       [ 0.,  0.5,  1.,  1.5,  2. ]]])
>>> z.shape
(2, 3, 5) # 2 plans 2D (x, y) de 3 lignes (y) × 5 colonnes (x)
>>> N.mgrid[0:1:5j, 0:2:7j, 0:3:9j].shape
(3, 5, 7, 9) # 3 volumes 3D (x, y, z) de 5 plans (z) × 7 lignes (y) × 9 colonnes (x)
```

Attention : à l'ordre de variation des indices dans les tableaux multidimensionnels, et aux différences entre `numpy.meshgrid()` et `numpy.mgrid`.

- `numpy.random.rand()` crée un tableau d'un format donné de réels aléatoires dans $[0, 1[$; `numpy.`

`random.randn()` génère un tableau d'un format donné de réels tirés aléatoirement d'une distribution gaussienne (normale) standard $\mathcal{N}(\mu = 0, \sigma^2 = 1)$.

Manipulations sur les tableaux

Les `array` 1D sont indexables comme les listes standard. En dimension supérieure, chaque axe est indexable indépendamment.

```
>>> x = N.arange(10); # Rampe 1D
>>> x[1::3] *= -1; x # Modification sur place ("in place")
array([ 0, -1,  2,  3, -4,  5,  6, -7,  8,  9])
```

Slicing

Les sous-tableaux de rang $< N$ d'un tableau de rang N sont appelées *slices* : le (ou les) axe(s) selon le(s)quel(s) la *slice* a été découpée, devenu(s) de longueur 1, est (sont) éliminé(s).

```
>>> y = N.arange(2*3*4).reshape(2, 3, 4); y # 2 plans, 3 lignes, 4 colonnes
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
>>> y[0, 1, 2] # 1er plan (axe 0), 2e ligne (axe 1), 3e colonne (axe 2)
6 # scalaire, shape *()**, ndim 0
>>> y[0, 1] # = y[0, 1, :] 1er plan (axe 0), 2e ligne (axe 1)
array([4, 5, 6, 7]) # Shape (4,)
>>> y[0] # = y[0, :, :] 1er plan (axe 0)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]]) # Shape (3, 4)
>>> y[0][1][2] # = y[0, 1, 2] en ~4x plus lent (slices successives)
6
>>> y[:, -1] # = y[:, 2, :] Dernière slice selon le 2e axe
array([[ 8,  9, 10, 11],
       [20, 21, 22, 23]]) # Shape (2, 4)
>>> y[:, :, 0] # = y[:, :, 0] 1re slice selon le dernier axe
array([[ 0,  4,  8],
       [12, 16, 20]]) # Shape (2, 3)
>>> # On peut vouloir garder explicitement la dimension "tranchée"
>>> y[:, :, 0:1] # 1re slice selon le dernier axe *en gardant le rang originel*
array([[[ 0],
        [ 4],
        [ 8]],
       [[12],
        [16],
        [20]]])
>>> y[:, :, 0:1].shape
(2, 3, 1) # Le dernier axe a été conservé, il ne contient pourtant qu'un seul élément
```

Modification de format

`numpy.ndarray.reshape()` modifie le format d'un tableau sans modifier le nombre total d'éléments :

```
>>> y = N.arange(6).reshape(2, 3); y # Shape (6,) → (2, 3) (*size* inchangé)
array([[0, 1, 2],
```

(suite sur la page suivante)

(suite de la page précédente)

```
[3, 4, 5]])
>>> y.reshape(2, 4)      # Format incompatible (*size* serait modifié)
ValueError: total size of new array must be unchanged
```

`numpy.ndarray.ravel()` « déroule » tous les axes et retourne un tableau de rang 1 :

```
>>> y.ravel()             # *1st axis slowest, last axis fastest*
array([ 0, 1, 2, 3, 4, 5]) # Shape (2, 3) → (6,) (*size* inchangé)
>>> y.ravel('F')         # *1st axis fastest, last axis slowest* (ordre Fortran)
array([0, 3, 1, 4, 2, 5])
```

`numpy.ndarray.transpose()` transpose deux axes, par défaut les derniers (raccourci : `numpy.ndarray.T`) :

```
>>> y.T                  # Transposition = y.transpose() (voir aussi *rollaxis*)
array([[0, 3],
       [1, 4],
       [2, 5]])
```

`numpy.ndarray.squeeze()` élimine tous les axes de dimension 1. `numpy.expand_dims()` ajoute un axe de dimension 1 en position arbitraire. Cela est également possible en utilisant notation *slice* avec `numpy.newaxis`.

```
>>> y[..., 0:1].squeeze() # Élimine *tous* les axes de dimension 1
array([0, 3])
>>> N.expand_dims(y[..., 0], -1).shape # Ajoute un axe de dim. 1 en dernière position
(2, 1)
>>> y[:, N.newaxis].shape   # Ajoute un axe de dim. 1 en 2de position
(2, 1, 3)
```

`numpy.resize()` modifie le format en modifiant le nombre total d'éléments :

```
>>> N.resize(N.arange(4), (2, 4)) # Complétion avec des copies du tableau
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
>>> N.resize(N.arange(4), (4, 2))
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])
```

Attention : `N.resize(arr, shape)` (complétion avec des copies de `arr`) est différent de `arr.resize(shape)` (complétion avec des 0).

Exercice :

*Inversion de matrice **

Stacking

```
>>> a = N.arange(5); a
array([0, 1, 2, 3, 4])
>>> N.hstack((a, a)) # Stack horizontal (le long des colonnes)
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
>>> N.vstack((a, a)) # Stack vertical (le long des lignes)
```

(suite sur la page suivante)

(suite de la page précédente)

```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
>>> N.dstack((a, a)) # Stack en profondeur (le long des plans)
array([[[0, 0],
        [1, 1],
        [2, 2],
        [3, 3],
        [4, 4]]])
```

Broadcasting

L'array `broadcasting` définit les règles selon lesquelles deux tableaux de formats *différents* peuvent éventuellement s'apparier.

1. Deux tableaux sont compatibles (*broadcastable*) si, pour chaque axe, soit les tailles sont égales, soit l'une d'elles est exactement égale à 1. P.ex. (5, 3) et (1, 3) sont des formats *broadcastable*, (5, 3) et (5, 1) également, mais (5, 3) et (3, 1) ne le sont pas.
2. Si un tableau a un axe de taille 1, le tableau sera dupliqué à la volée autant de fois que nécessaire selon cet axe pour atteindre la taille de l'autre tableau le long de cet axe. P.ex. un tableau (2, 1, 3) pourra être transformé en tableau (2, 5, 3) en le dupliquant 5 fois le long du 2e axe (`axis=1`).
3. La taille selon chaque axe après *broadcast* est égale au maximum de toutes les tailles d'entrée le long de cet axe. P.ex. $(5, 3, 1) \times (1, 3, 4) \rightarrow (5, 3, 4)$.
4. Si un des tableaux a un rang (`ndim`) inférieur à l'autre, alors son format (`shape`) est précédé d'autant de 1 que nécessaire pour atteindre le même rang. P.ex. $(5, 3, 1) \times (4,) = (5, 3, 1) \times (1, 1, 4) \rightarrow (5, 3, 4)$.

```
>>> a = N.arange(6).reshape(2, 3); a # Shape (2, 3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = N.array([10, 20, 30]); b # Shape (3,)
array([10, 20, 30])
>>> a + b # Shape (3,) ~ (1, 3) → (2, 3) = (1, 3) copié 2 fois
array([[10, 21, 32],
       [13, 24, 35]]) # Shape (2, 3)
>>> c = N.array([10, 20]); c # Shape (2,)
array([10, 20])
>>> a + c # Shape (2,) ~ (1, 2) incompatible avec (2, 3)
ValueError: shape mismatch: objects cannot be broadcast to a single shape
>>> c[:, N.newaxis] # = c.reshape(-1, 1) Shape (2, 1)
array([[10],
       [20]])
>>> a + c[:, N.newaxis] # Shape (2, 1) → (2, 3) = (2, 1) copié 3 fois
array([[10, 11, 12],
       [23, 24, 25]])
```

Voir également cette [présentation](#).

Indexation évoluée

```
>>> a = N.linspace(-1, 1, 5); a
array([-1. , -0.5,  0. ,  0.5,  1. ])
>>> a >= 0 # Test logique: tableau de booléens
array([False, False,  True,  True,  True], dtype=bool)
>>> (a >= 0).nonzero() # Indices des éléments ne s'évaluant pas à False
(array([2, 3, 4]),) # Indices des éléments >= 0
>>> a[(a >= 0).nonzero()] # Indexation par un tableau d'indices, pas pythonique :-(
```

(suite sur la page suivante)

(suite de la page précédente)

```
array([ 0. ,  0.5,  1. ])
>>> a[a >= 0]           # Indexation par un tableau de booléens, pythonique :-D
array([ 0. ,  0.5,  1. ])
>>> a[a < 0] -= 10; a    # = N.where(a < 0, a - 10, a)
array([-11. , -10.5,  0. ,  0.5,  1. ])
```

Opérations de base

```
>>> a = N.arange(3); a # Shape (3,), type *int*
array([0, 1, 2])
>>> b = 1.             # ~ Shape (), type *float*
>>> c = a + b; c        # *Broadcasting*: () → (1,) → (3,)
array([ 1.,  2.,  3.]) # *Upcasting*: int → float
>>> a += 1; a           # Modification *in place* (plus efficace si possible)
array([ 1.,  2.,  3.])
>>> a.mean()            # *ndarray* dispose de nombreuses méthodes numériques de base
2.0
```

Opérations sur les axes

```
>>> x = N.random.permutation(6).reshape(3, 2); x # 3 lignes, 2 colonnes
array([[3, 4],
       [5, 1],
       [0, 2]])
>>> x.min()             # Minimum global (comportement par défaut: `axis=None`)
0
>>> x.min(axis=0)        # Minima le long de l'axe 0 (i.e. l'axe des lignes)
array([0, 1])           # ce sont les minima colonne par colonne: (*3*, 2) → (2,)
>>> x.min(axis=1)        # Minima le long de l'axe 1 (i.e. l'axe des colonnes)
array([3, 1, 0])        # ce sont les minima ligne par ligne (3, *2*) → (3,)
>>> x.min(axis=1, keepdims=True) # Idem mais en *conservant* le format originel
array([[3],
       [1],
       [0]])
>>> x.min(axis=(0, 1)) # Minima le long des axes 0 *et* 1 (c.-à-d. ici tous les axes)
0
```

Opérations matricielles

Les opérations de base s'appliquent sur les *éléments* des tableaux, et n'ont pas une signification matricielle par défaut :

```
>>> m = N.arange(4).reshape(2, 2); m # Tableau de rang 2
array([[0, 1],
       [2, 3]])
>>> i = N.identity(2, dtype=int); i # Tableau "identité" de rang 2 (type entier)
array([[1, 0],
       [0, 1]])
>>> m * i               # Attention! opération * sur les éléments
array([[0, 0],
       [0, 3]])
>>> N.dot(m, i)         # Multiplication *matricielle* des tableaux:  $M \times I = M$ 
array([[0, 1],
       [2, 3]])
```

Il est possible d'utiliser systématiquement les opérations matricielles en manipulant des `numpy.matrix` plutôt que de `numpy.ndarray` :

```
>>> N.matrix(m) * N.matrix(i) # Opération * entre matrices
matrix([[0, 1],
        [2, 3]])
```

Le sous-module `numpy.linalg` fournit des outils spécifiques au calcul matriciel (inverse, déterminant, valeurs propres, etc.).

Ufuncs

`numpy` fournit de nombreuses fonctions mathématiques de base (`numpy.exp()`, `numpy.atan2()`, etc.) s'appliquant directement sur les éléments des tableaux d'entrée :

```
>>> x = N.linspace(0, 2*N.pi, 5) # [0, π/2, π, 3π/2, 2π]
>>> y = N.sin(x); y              # sin(x) = [0, 1, 0, -1, 0]
array([ 0.00000000e+00,  1.00000000e+00,  1.22460635e-16,
        -1.00000000e+00, -2.44921271e-16])
>>> y == [0, 1, 0, -1, 0]        # Test d'égalité stricte (élément par élément)
array([ True,  True, False,  True, False], dtype=bool) # Attention aux calculs en réels
↳(float)!
>>> N.all(N.sin(x) == [0, 1, 0, -1, 0]) # Test d'égalité stricte de tous les éléments
False
>>> N.allclose(y, [0, 1, 0, -1, 0])    # Test d'égalité numérique de tous les éléments
True
```

Exercices :

*Median Absolute Deviation *, Distribution du pull ****

4.1.2 Tableaux évolués

Types composés

Par définition, tous les éléments d'un tableau *homogène* doivent être du même type. Cependant, outre les types scalaires élémentaires – `bool`, `int`, `float`, `complex`, `str`, etc. – `numpy` supporte les types *composés*, c.-à-d. incluant plusieurs sous-éléments de types différents :

```
>>> dt = N.dtype([('nom', 'S10'),      # 1er élément: une chaîne de 10 caractères
...              ('age', 'i'),        # 2e élément: un entier
...              ('taille', 'd')])    # 3e élément: un réel (double)
>>> arr = N.array([('Calvin', 6, 1.20), ('Hobbes', 5, 1.80)], dtype=dt); arr
array([('Calvin', 6, 1.2), ('Hobbes', 6, 1.8)],
      dtype=[('nom', '<S10'), ('age', '<i4'), ('taille', '<f8')])
>>> arr[0]                            # Accès par élément
('Calvin', 6, 1.2)
>>> arr['nom']                        # Accès par sous-type
array(['Calvin', 'Hobbes'], dtype='<S10')
>>> rec = arr.view(N.recarray); arr    # Vue de type 'recarray'
rec.array([('Calvin', 6, 1.2), ('Hobbes', 5, 1.8)],
          dtype=[('nom', '<S10'), ('age', '<i4'), ('taille', '<f8')])
>>> rec.nom                          # Accès direct par attribut
chararray(['Calvin', 'Hobbes'], dtype='<S10')
```

Les tableaux structurés sont très puissants pour manipuler des données (semi-)hétérogènes, p.ex. les entrées du catalogue CSV des objets de Messier `Messier.csv` :

```

1 # Messier, NGC, Magnitude, Size [arcmin], Distance [pc], RA [h], Dec [deg], Constellation,
  ↳Season, Name
2 # Attention: les données n'ont pas vocation à être très précises!
3 # D'après http://astropixels.com/messier/messiercat.html
4 M,NGC,Type,Mag,Size,Distance,RA,Dec,Con,Season,Name
5 M1,1952,Sn,8.4,5.0,1930.0,5.575,22.017,Tau,winter,Crab Nebula
6 M2,7089,Gc,6.5,12.9,11600.0,21.558,0.817,Aqr,autumn,
7 M3,5272,Gc,6.2,16.2,10400.0,13.703,28.383,CVn,spring,
8 M4,6121,Gc,5.6,26.3,2210.0,16.393,-26.533,Sco,summer,

>>> dt = N.dtype([('M', 'S3'),          # N° catalogue Messier
...               ('NGC', 'i'),         # N° New General Catalogue
...               ('Type', 'S2'),       # Code type
...               ('Mag', 'f'),         # Magnitude
...               ('Size', 'f'),        # Taille [arcmin]
...               ('Distance', 'f'),    # Distance [pc]
...               ('RA', 'f'),          # Ascension droite [h]
...               ('Dec', 'f'),         # Déclinaison [deg]
...               ('Con', 'S3'),        # Code constellation
...               ('Season', 'S6'),     # Saison
...               ('Name', 'S30')])    # Nom alternatif
>>> messier = N.genfromtxt("Messier.csv", dtype=dt, delimiter=',', comments='#')
>>> messier[1]
('M1', 1952, 'Sn', 8.39999962, 5., 1930., 5.57499981, 22.0170002, 'Tau', 'winter', 'Crab_
↳Nebula')
>>> N.nanmean(messier['Mag'])
7.4927273

```

Tableaux masqués

Le sous-module `numpy.ma` ajoute le support des tableaux masqués (*Masked Arrays*). Imaginons un tableau (4, 5) de réels (positifs ou négatifs), sur lequel nous voulons calculer pour chaque colonne la moyenne des éléments *positifs* uniquement :

```

>>> x = N.random.randn(4, 5); x
array([[ -0.55867715,  1.58863893, -1.4449145 ,  1.93265481, -0.17127422],
       [ -0.86041806,  1.98317832, -0.32617721,  1.1358607 , -1.66150602],
       [ -0.88966893,  1.36185799, -1.54673735, -0.09606195,  2.23438981],
       [  0.35943269, -0.36134448, -0.82266202,  1.38143768, -1.3175115 ]])
>>> x[x >= 0]          # Donne les éléments >0 du tableau, sans leur indice
array([ 1.58863893,  1.93265481,  1.98317832,  1.1358607 ,  1.36185799,
        2.23438981,  0.35943269,  1.38143768])
>>> (x >= 0).nonzero() # Donne les indices ([i], [j]) des éléments positifs
(array([0, 0, 1, 1, 2, 2, 3, 3]), array([1, 3, 1, 3, 1, 4, 0, 3]))
>>> y = N.ma.masked_less(x, 0); y # Tableau où les éléments <0 sont masqués
masked_array(data =
  [[-- 1.58863892701 -- 1.93265481164 --] # Données
  [-- 1.98317832359 -- 1.13586070417 --]
  [-- 1.36185798574 -- -- 2.23438980788]
  [0.359432688656 -- -- 1.38143767743 --]],
  mask =
  [[ True False  True False  True]
  [ True False  True False  True]
  [ True False  True  True False]
  [False  True  True False  True]],
  fill_value = 1e+20)
>>> m0 = y.mean(axis=0); m0          # Moyenne sur les lignes (axe 0)
masked_array(data = [0.359432688656 1.64455841211 -- 1.48331773108 2.23438980788],
  mask = [False False  True False False],

```

(suite sur la page suivante)

(suite de la page précédente)

```

fill_value = 1e+20)                                # Le résultat est un *Masked Array*
>>> m0.filled(-1)                                   # Conversion en tableau normal
array([ 0.35943269,  1.64455841, -1.                  ,  1.48331773,  2.23438981])

```

Note : Les tableaux *évolués* de `numpy` sont parfois suffisants, mais pour une utilisation avancée, il peut être plus pertinent d'invoquer les bibliothèques dédiées *Pandas et xarray*.

4.1.3 Entrées/sorties

`numpy` peut lire – `numpy.loadtxt()` – ou sauvegarder – `numpy.savetxt()` – des tableaux dans un simple fichier ASCII :

```

>>> x = N.linspace(-1, 1, 100)
>>> N.savetxt('archive_x.dat', x) # Sauvegarde dans le fichier 'archive_x.dat'
>>> y = N.loadtxt("archive_x.dat") # Relecture à partir du fichier 'archive_x.dat'
>>> (x == y).all()                # Test d'égalité stricte
True

```

Attention : `numpy.loadtxt()` supporte les types composés, mais ne supporte pas les données manquantes ; utiliser alors la fonction `numpy.genfromtxt()`, plus robuste mais plus lente.

Le format texte n'est pas optimal pour de gros tableaux : il peut alors être avantageux d'utiliser le format binaire `.npy`, beaucoup plus compact (mais non *human readable*) :

```

>>> x = N.linspace(-1, 1, 1e6)
>>> N.save('archive_x.npy', x) # Sauvegarde dans le fichier 'archive_x.npy'
>>> y = N.load("archive_x.npy") # Relecture à partir du fichier 'archive_x.npy'
>>> (x == y).all()
True

```

Il est enfin possible de *sérialiser* les tableaux à l'aide de la bibliothèque standard *pickle*.

4.1.4 Sous-modules

`numpy` fournit en outre quelques fonctionnalités supplémentaires, parmi lesquelles les sous-modules suivants :

- `numpy.fft` : *Discrete Fourier Transform* ;
- `numpy.random` : valeurs aléatoires ;
- `numpy.polynomial` : manipulation des polynômes (racines, polynômes orthogonaux, etc.).

4.1.5 Performances

Avertissement : *Premature optimization is the root of all evil* – Donald Knuth

Même si `numpy` apporte un gain significatif en performance par rapport à du Python standard, il peut être possible d'améliorer la vitesse d'exécution par l'utilisation de bibliothèques externes dédiées, p.ex. :

- `NumExpr` est un évaluateur optimisé d'expressions numériques :

```

>>> a = N.arange(1e6)
>>> b = N.arange(1e6)
>>> %timeit a*b - 4.1*a > 2.5*b
100 loops, best of 3: 11.4 ms per loop
>>> %timeit numexpr.evaluate("a*b - 4.1*a > 2.5*b")
100 loops, best of 3: 1.97 ms per loop
>>> %timeit N.exp(-a)
10 loops, best of 3: 60.1 ms per loop
>>> timeit numexpr.evaluate("exp(-a)") # Multi-threaded
10 loops, best of 3: 19.3 ms per loop

```

- **Bottleneck** est une collection de fonctions accélérées, notamment pour des tableaux contenant des NaN ou pour des statistiques glissantes ;
- **theano**, pour optimiser l'évaluation des expressions mathématiques sur les tableaux **numpy**, notamment par l'utilisation des GPU (Graphics Processing Unit) et de code C généré à la volée.

Attention : **theano** n'est officiellement plus soutenu.

Voir également *Profilage et optimisation*.

4.2 Scipy

scipy est une bibliothèque *numérique*¹ d'algorithmes et de fonctions mathématiques, basée sur les tableaux **numpy.ndarray**, complétant ou améliorant (en termes de performances) les fonctionnalités de **numpy**.

Note : N'oubliez pas de citer **scipy** & co. dans vos publications et présentations utilisant ces outils.

4.2.1 Tour d'horizon

- **scipy.special** : fonctions spéciales (fonctions de Bessel, erf, gamma, etc.).
- **scipy.integrate** : intégration numérique (intégration numérique ou d'équations différentielles).
- **scipy.optimize** : méthodes d'optimisation (minimisation, moindres-carrés, zéros d'une fonction, etc.).
- **scipy.interpolate** : interpolation (interpolation, splines).
- **scipy.fftpack** : transformées de Fourier.
- **scipy.signal** : traitement du signal (convolution, corrélation, filtrage, ondelettes, etc.).
- **scipy.linalg** : algèbre linéaire.
- **scipy.stats** : statistiques (fonctions et distributions statistiques).
- **scipy.ndimage** : traitement d'images multi-dimensionnelles.
- **scipy.io** : entrées/sorties.

Liens :

- [Scipy Reference](#)
- [Scipy Cookbook](#)

Voir également :

- **Scikits** : modules plus spécifiques étroitement liés à **scipy**, parmi lesquels :

Python dispose également d'une bibliothèque de calcul *formel*, **sympy**, et d'un environnement de calcul mathématique, **sage**.

- scikit-learn : *machine learning*,
- scikit-image : *image processing*,
- statsmodel : modèles statistiques (tutorial),
- Scipy Topical softwares.

Exercices :

*Quadrature et zéro d'une fonction **, *Schéma de Romberg ***, *Méthode de Runge-Kutta ***

4.2.2 Quelques exemples complets

- Interpolation
- Integration (intégrales numériques, équations différentielles)
 - odeint notebook
 - Zombie Apocalypse
- Optimisation (moindres carrés, ajustements, recherche de zéros)
- Traitement du signal (splines, convolution, filtrage)
- Algèbre linéaire (systèmes linéaires, moindres carrés, décompositions)
- Statistiques (variables aléatoires, distributions, tests)

4.3 Matplotlib

Matplotlib est une bibliothèque graphique de visualisation 2D (avec un support pour la 3D, l'animation et l'interactivité), permettant des sorties de haute qualité « prêtes à publier ». C'est à la fois une bibliothèque de *haut niveau*, fournissant des fonctions de visualisation « clés en main » (échelle logarithmique, histogramme, courbes de niveau, etc., voir la [galerie](#)), et de *bas niveau*, permettant de modifier tous les éléments graphiques de la figure (titre, axes, couleurs et styles des lignes, etc., voir [Anatomie d'une figure](#)).

4.3.1 pylab vs. pyplot

Il existe deux interfaces pour deux types d'utilisation :

- `pylab` : interface procédurale, originellement très similaire à MATLAB™ et généralement réservée à l'analyse interactive :

```
>>> from pylab import *          # DÉCONSEILLÉ DANS UN SCRIPT!
>>> x = linspace(-pi, pi, 100)   # pylab importe numpy dans l'espace courant
>>> plot(x, sin(x), 'b-', label="Sinus")    # Trace la courbe y = sin(x)
>>> plot(x, cos(x), 'r:', label="Cosinus")   # Trace la courbe y = cos(x)
>>> xlabel("x [rad]")            # Ajoute le nom de l'axe des x
>>> ylabel("y")                  # Ajoute le nom de l'axe des y
>>> title("Sinus et Cosinus")     # Ajoute le titre de la figure
>>> legend()                     # Ajoute une légende
>>> savefig("simple.png")         # Enregistre la figure en PNG
```

- `matplotlib.pyplot` : interface orientée objet, préférable pour les scripts :

```
import numpy as N
import matplotlib.pyplot as P

x = N.linspace(-N.pi, N.pi, 100)

fig, ax = P.subplots() # Création d'une figure contenant un seul système d'axes
ax.plot(x, N.sin(x), c='b', ls='-', label="Sinus")    # Courbe y = sin(x)
ax.plot(x, N.cos(x), c='r', ls=':', label="Cosinus")  # Courbe y = cos(x)
```

(suite sur la page suivante)

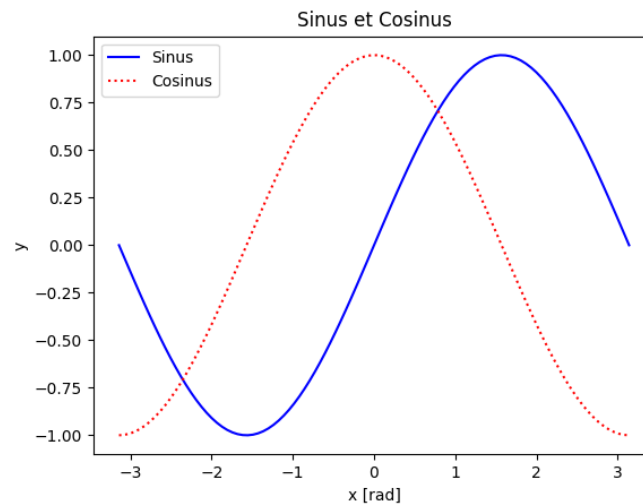
(suite de la page précédente)

```

ax.set_xlabel("x [rad]")      # Nom de l'axe des x
ax.set_ylabel("y")           # Nom de l'axe des y
ax.set_title("Sinus et Cosinus") # Titre de la figure
ax.legend()                  # Légende
fig.savefig("simple.png")     # Sauvegarde en PNG

```

Dans les deux cas, le résultat est le même :



Par la suite, nous nous concentrerons sur l'interface OO (Orientée Objet) `matplotlib.pyplot`, plus puissante et flexible.

4.3.2 Figure et axes

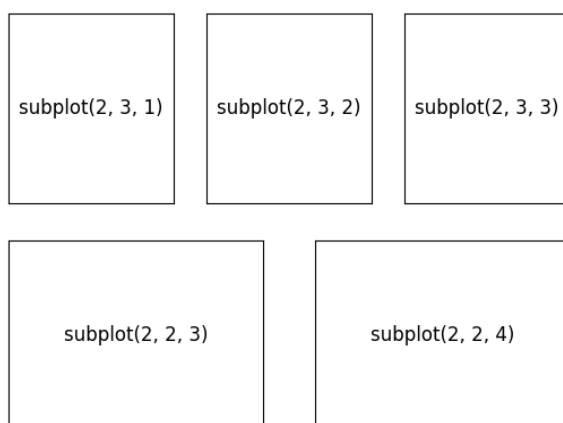
L'élément de base est le système d'axes `matplotlib.axes.Axes`, qui définit et réalise la plupart des éléments graphiques (tracé de courbes, définition des axes, annotations, etc.). Un ou plusieurs de ces systèmes d'axes sont regroupés au sein d'une `matplotlib.figure.Figure`.

Ainsi, pour générer une figure contenant 2 (vertical) \times 3 (horizontal) = 6 systèmes d'axes (numérotés de 1 à 6) :

```

fig = P.figure()
for i in range(1, 4):
    ax = fig.add_subplot(2, 3, i, xticks=[], yticks=[])
    ax.text(0.5, 0.5, "subplot(2, 3, {i})".format(i),
           ha='center', va='center', size='large')
for i in range(3, 5):
    ax = fig.add_subplot(2, 2, i, xticks=[], yticks=[])
    ax.text(0.5, 0.5, "subplot(2, 2, {i})".format(i),
           ha='center', va='center', size='large')

```



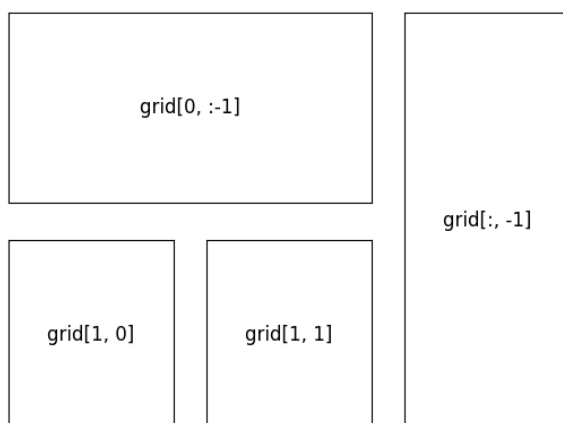
Pour des mises en page plus complexes, il est possible d'utiliser le kit `gridspec`, p.ex. :

```
from matplotlib.gridspec import GridSpec

fig = P.figure()
fig.suptitle("grid = GridSpec(2, 3)", fontsize='x-large')

grid = GridSpec(2, 3)
ax1 = fig.add_subplot(grid[0, :-1], xticks=[], yticks=[])
ax1.text(0.5, 0.5, "grid[0, :-1]", ha='center', va='center', size='large')
ax2 = fig.add_subplot(grid[:, -1], xticks=[], yticks=[])
ax3.text(0.5, 0.5, "grid[:, -1]", ha='center', va='center', size='large')
ax3 = fig.add_subplot(grid[1, 0], xticks=[], yticks=[])
ax3.text(0.5, 0.5, "grid[1, 0]", ha='center', va='center', size='large')
ax4 = fig.add_subplot(grid[1, 1], xticks=[], yticks=[])
ax4.text(0.5, 0.5, "grid[1, 1]", ha='center', va='center', size='large')
```

grid = GridSpec(2, 3)



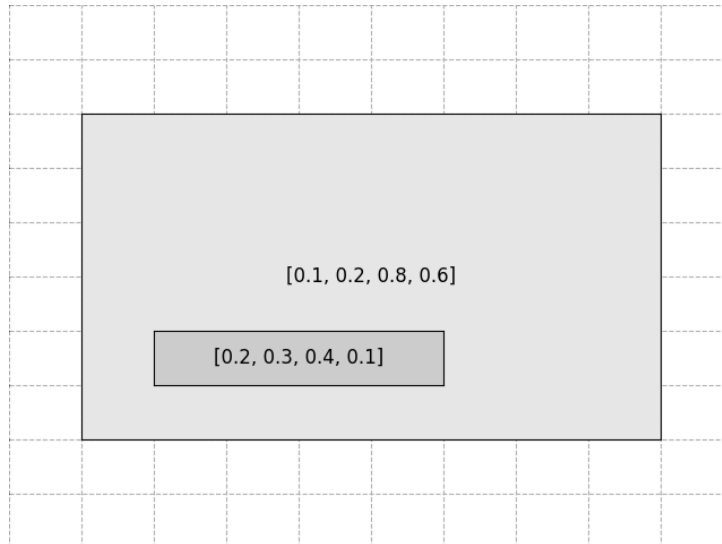
Enfin, il est toujours possible (mais peu pratique) de créer soi-même le système d'axes dans les coordonnées relatives à la figure :

```
fig = P.figure()
ax0 = fig.add_axes([0, 0, 1, 1], frameon=False,
                   xticks=N.linspace(0, 1, 11), yticks=N.linspace(0, 1, 11))
ax0.grid(True, ls='--')
```

(suite sur la page suivante)

(suite de la page précédente)

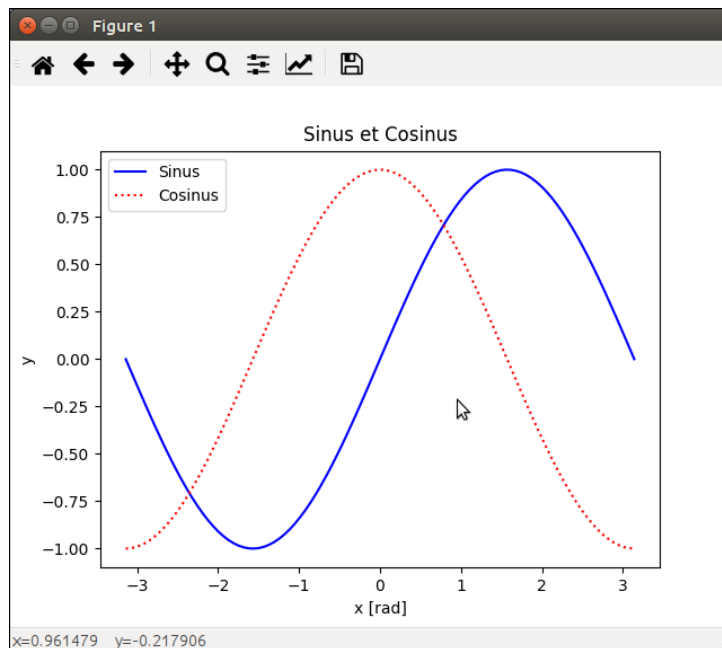
```
ax1 = fig.add_axes([0.1, 0.2, 0.8, 0.6], xticks=[], yticks=[], fc='0.9')
ax1.text(0.5, 0.5, "[0.1, 0.2, 0.8, 0.6]", ha='center', va='center', size='large')
ax2 = fig.add_axes([0.2, 0.3, 0.4, 0.1], xticks=[], yticks=[], fc='0.8')
ax2.text(0.5, 0.5, "[0.2, 0.3, 0.4, 0.1]", ha='center', va='center', size='large')
```



4.3.3 Sauvegarde et affichage interactif

La méthode `matplotlib.figure.Figure.savefig()` permet de sauvegarder la figure dans un fichier dont le format est automatiquement défini par son extension, `png` (*raster*), `[e]ps`, `pdf`, `svg` (*vector*), etc., via différents `backends`.

Il est également possible d'afficher la figure dans une *fenêtre interactive* avec la commande `matplotlib.pyplot.show()` :



Note : Utiliser `ipython --pylab` pour l'utilisation interactive des figures dans une session `ipython`.

4.3.4 Anatomie d'une figure

L'interface OO `matplotlib.pyplot` donne accès à tous les éléments d'une figure (titre, axes, légende, etc.), qui peuvent alors être ajustés (couleur, police, taille, etc.).

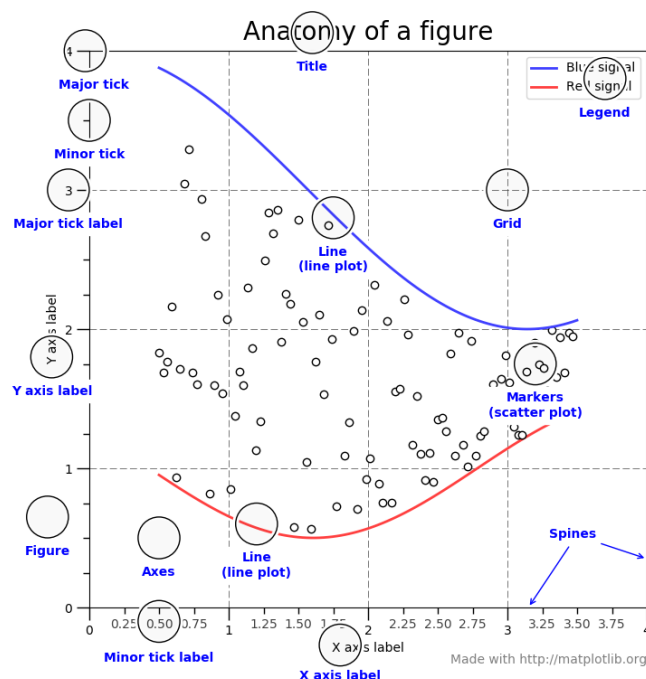



Fig. 4.1 – **Figure** : Anatomie d'une figure.

Note : N'oubliez pas de citer `matplotlib` (notamment *[Matplotlib07]*) dans vos publications et présentations utilisant cet outil.

Liens :

- User's Guide
- Gallery
- Tutorial matplotlib
- Tutoriel matplotlib 

Voir également :

- MPLD3, un *backend* matplotlib interactif basé sur la bibliothèque *web 3D.js* ;
- basemap et cartopy, bibliothèques de cartographie sphérique ;
- Seaborn, une surcouche de visualisation statistique à matplotlib et *Pandas et xarray* ;
- HoloViews, une surcouche de visualisation et d'analyse à matplotlib ;
- ggplot, une surcouche orientée *Grammar of Graphics* à matplotlib ;
- Bokeh, une bibliothèque graphique alternative à matplotlib plus orientée *web*/temps réel.

Exemples :

figure.py, *filtres2ndOrdre.py*

Exercices :

*Quartet d'Anscombe **, *Ensemble de Julia ***, *Diagramme de bifurcation : la suite logistique ***

4.3.5 Visualisation 3D

Matplotlib fournit d'emblée une interface `mplot3d` pour des figures 3D assez simples :

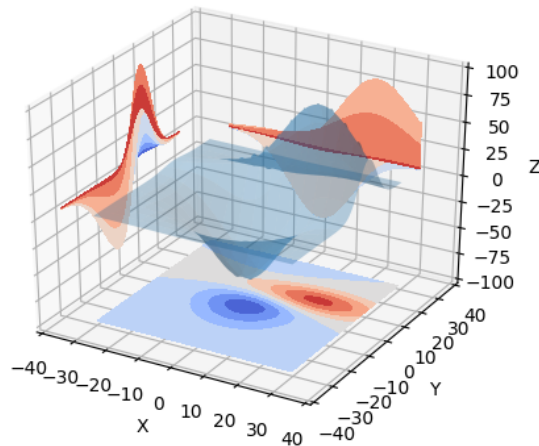


Fig. 4.2 – **Figure** : Exemple de figure matplotlib 3D.

Pour des visualisations plus complexes, `mayavi.mlab` est une bibliothèque graphique de visualisation 3D s'appuyant sur `Mayavi`.

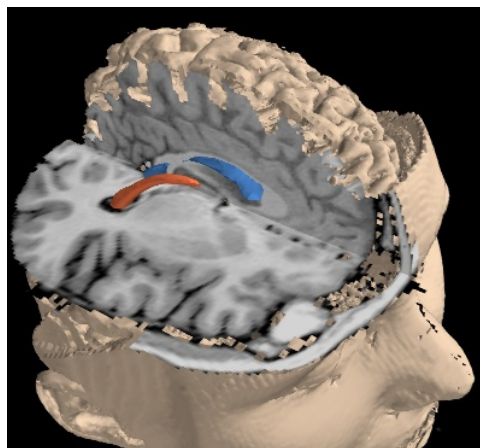


Fig. 4.3 – **Figure** : Imagerie par résonance magnétique.

Note : N'oubliez pas de citer `mayavi` dans vos publications et présentations utilisant cet outil.

Voir également :

— `VPython` : *3D Programming for Ordinary Mortals* ;

— [Glowscript](#) : VPython dans le navigateur.

Notes de bas de page et références bibliographiques

Table des matières

- *Bibliothèques scientifiques avancées*
- *Pandas et xarray*
 - *Structures*
 - *Accès aux données*
 - *Manipulation des données*
 - *Regroupement et agrégation de données*
 - *Visualisations*
 - *Xarray*
- *Astropy*
 - *Tour d'horizon*
 - *Démonstration*
- *Autres bibliothèques scientifiques*

5.1 Pandas et xarray

`pandas` est une bibliothèque pour la structuration et l'analyse avancée de données *hétérogènes* (PANel DATA). Elle fournit notamment :

- des structures de données relationnelles (« labellisées »), avec une indexation simple ou hiérarchique (c.-à-d. à plusieurs niveaux) ;
- des méthodes d'alignement et d'agrégation des données, avec gestion des données manquantes ;
- un support performant des labels temporels (p.ex. dates, de par son origine dans le domaine de l'économétrie), et des statistiques « glissantes » ;
- de nombreuses fonctions d'entrée/sortie, d'analyse statistiques et de visualisation.

Les fonctionnalités de `pandas` sont *très* riches et couvrent de nombreux aspects (données manquantes, dates, analyse statistiques, etc.) : il n'est pas question de toutes les aborder ici. Avant de vous lancer dans une manipulation qui vous semble complexe, bien inspecter la [documentation](#), très complète (p.ex. les recettes du [cookbook](#)), pour vérifier qu'elle n'est pas déjà implémentée ou documentée, ou pour identifier l'approche la plus efficace.

Note : La convention utilisée ici est « `import pandas as PD` ».

Attention : Les bibliothèques `pandas` et `xarray` sont encore en phase de développement assez intense, et vont probablement être amenées à évoluer significativement, et pas nécessairement de manière rétro-compatible. Nous travaillons ici sur les versions :

- `pandas 0.21.x`
- `xarray 0.10.x`

5.1.1 Structures

Références : [Introduction to Data Structures](#)

Pandas dispose de deux grandes structures de données¹ :

Nom de la structure	Rang	Description
<code>pandas.Series</code>	1	Vecteur de données <i>homogènes</i> labellisées
<code>pandas.DataFrame</code>	2	Tableau structuré de colonnes <i>homogènes</i>

```
>>> PD.Series(range(3)) # Série d'entiers sans indexation
0    0
1    1
2    2
dtype: int64
>>> PD.Series(N.random.randn(3), index=list('abc')) # Série de réels indexés
a   -0.553480
b    0.081297
c   -1.845835
dtype: float64
>>> PD.DataFrame(N.random.randn(3, 4))
      0         1         2         3
0  1.570977 -0.677845  0.094364 -0.362031
1 -0.136712  0.762300  0.068611  1.265816
2 -0.697760  0.791288  0.449645 -1.105062
>>> PD.DataFrame([(1, N.exp(1), 'un'), (2, N.exp(2), 'deux'), (3, N.exp(3), 'trois')],
...               index=list('abc'), columns='i val nom'.split())
   i      val      nom
a  1   2.718282      un
b  2   7.389056     deux
c  3  20.085537     trois
```

Pour mettre en évidence la puissance de Pandas, nous utiliserons le *catalogue des objets Messier* vu précédemment. Le fichier peut être importé à l'aide de la fonction `pandas.read_csv()`, et le *dataframe* résultant est labellisé à la volée par la colonne M (`pandas.DataFrame.set_index()`) :

```
>>> messier = PD.read_csv("Messier.csv", comment='#') # Lecture du fichier CSV
>>> messier.set_index('M', inplace=True) # Indexation sur la colonne "M"
>>> messier.info() # Informations générales
<class 'pandas.core.frame.DataFrame'>
Index: 110 entries, M1 to M110
Data columns (total 10 columns):
NGC      108 non-null object
Type     110 non-null object
Mag      110 non-null float64
```

(suite sur la page suivante)

¹ Les structures `pandas.Panel` (de rang 3), `pandas.Panel4D` (de rang 4) et `pandas.PanelND` (de rang arbitraire) sont considérées comme **dépréciées** et seront retirées dans une version ultérieure. Utiliser une indexation hiérarchique ou `xarray`.

(suite de la page précédente)

```

Size          110 non-null float64
Distance      110 non-null float64
RA            110 non-null float64
Dec           110 non-null float64
Con           110 non-null object
Season        110 non-null object
Name          31 non-null object
dtypes: float64(5), object(5)
memory usage: 9.5+ KB
>>> messier.head(3) # Par défaut les 5 premières lignes
   NGC Type  Mag  Size  Distance    RA    Dec  Con  Season      Name
M
M1  1952   Sn  8.4   5.0    1930.0  5.575  22.017  Tau  winter  Crab Nebula
M2  7089   Gc  6.5  12.9   11600.0  21.558   0.817  Aqr  autumn      NaN
M3  5272   Gc  6.2  16.2   10400.0  13.703  28.383  CVn   spring      NaN

```

Un *dataframe* est caractérisé par son indexation `pandas.DataFrame.index` et ses colonnes `pandas.DataFrame.columns` (de type `pandas.Index` ou `pandas.MultiIndex`), et les valeurs des données `pandas.DataFrame.values` :

```

>>> messier.index # Retourne un Index
Index([u'M1', u'M2', u'M3', ..., u'M108', u'M109', u'M110'],
      dtype='object', name=u'M', length=110)
>>> messier.columns # Retourne un Index
Index([u'NGC', u'Type', u'Mag', ..., u'Con', u'Season', u'Name'],
      dtype='object')
>>> messier.dtypes # Retourne une Series indexée sur le nom des colonnes
NGC          object
Type          object
Mag          float64
Size          float64
Distance      float64
RA            float64
Dec           float64
Con           object
Season        object
Name          object
dtype: object
>>> messier.values
array([[ '1952', 'Sn', 8.4, ..., 'Tau', 'winter', 'Crab Nebula'],
       [ '7089', 'Gc', 6.5, ..., 'Aqr', 'autumn', nan],
       ...,
       [ '3992', 'Ba', 9.8, ..., 'UMa', 'spring', nan],
       [ '205', 'El', 8.5, ..., 'And', 'autumn', nan]], dtype=object)
>>> messier.shape
(110, 10)

```

Une description statistique sommaire des colonnes numériques est obtenue par `pandas.DataFrame.describe()` :

```

>>> messier.drop(['RA', 'Dec'], axis=1).describe()
           Mag          Size          Distance
count  110.000000  110.000000  1.100000e+02
mean     7.492727   17.719091  4.574883e+06
std     1.755657   22.055100  7.141036e+06
min     1.600000    0.800000  1.170000e+02
25%     6.300000    6.425000  1.312500e+03
50%     7.650000    9.900000  8.390000e+03
75%     8.900000   17.300000  1.070000e+07
max    10.200000  120.000000  1.840000e+07

```

5.1.2 Accès aux données

Référence : Indexing and Selecting Data

L'accès par colonne retourne une `pandas.Series` (avec la même indexation) pour une colonne unique, ou un nouveau `pandas.DataFrame` pour plusieurs colonnes :

```
>>> messier.NGC # Équivalent à messier['NGC']
M
M1      1952
M2      7089
...
M109    3992
M110     205
Name: NGC, Length: 110, dtype: object
>>> messier[['RA', 'Dec']] # = messier.filter(items=('RA', 'Dec'))
      RA      Dec
M
M1    5.575  22.017
M2   21.558   0.817
...
M109  11.960  53.383
M110   0.673  41.683
[110 rows x 2 columns]
```

L'accès par `slice` retourne un nouveau `dataframe` :

```
>>> messier[:6:2] # Lignes 0 (inclus) à 6 (exclu) par pas de 2
      NGC Type  Mag  Size  Distance      RA      Dec  Con  Season      Name
M
M1  1952  Sn  8.4   5.0   1930.0   5.575  22.017  Tau  winter  Crab Nebula
M3  5272  Gc  6.2  16.2  10400.0  13.703  28.383  CVn   spring    NaN
M5  5904  Gc  5.6  17.4   7520.0  15.310   2.083  Ser   summer    NaN
```

L'accès peut également se faire par labels via `pandas.DataFrame.loc` :

```
>>> messier.loc['M31'] # Retourne une Series indexée par les noms de colonne
NGC      224
Type      Sp
...
Season      autumn
Name  Andromeda Galaxy
Name: M31, Length: 10, dtype: object
>>> messier.loc['M31', ['Type', 'Name']] # Retourne une Series
Type      Sp
Name  Andromeda Galaxy
Name: M31, dtype: object
>>> messier.loc[['M31', 'M51'], ['Type', 'Name']] # Retourne un DataFrame
      Type      Name
M
M31  Sp  Andromeda Galaxy
M51  Sp  Whirlpool Galaxy
>>> messier.loc['M31':'M33', ['Type', 'Name']] # De M31 à M33 inclu
      Type      Name
M
M31  Sp  Andromeda Galaxy
M32  El      NaN
M33  Sp  Triangulum Galaxy
```

De façon symétrique, l'accès peut se faire par position (n° de ligne/colonne) via `pandas.DataFrame.iloc`, p.ex. :


```
>>> messier.iloc[30:33, [1, -1]] # Ici, l'indice 33 n'est PAS inclu!
      Type      Name
M
M31  Sp    Andromeda Galaxy
M32  El              NaN
M33  Sp   Triangulum Galaxy
>>> messier.iloc[30:33, messier.columns.get_loc('Name')] # Mix des 2 approches
M
M31    Andromeda Galaxy
M32              NaN
M33    Triangulum Galaxy
Name: Name, dtype: object
```

Les fonctions `pandas.DataFrame.at` et `pandas.DataFrame.iat` permettent d'accéder *rapidement* aux données individuelles :

```
>>> messier.at['M31', 'NGC'] # 20× plus rapide que messier.loc['M31']['NGC']
'224'
>>> messier.iat[30, 0]      # 20× plus rapide que messier.iloc[0][0]
'224'
```

Noter qu'il existe une façon de filtrer les données sur les colonnes ou les labels :

```
>>> messier.filter(regex='M.7', axis='index').filter('RA Dec'.split())
      RA      Dec
M
M17  18.347 -16.183
M27  19.993  22.717
M37   5.873  32.550
M47   7.610 -14.500
M57  18.893  33.033
M67   8.840  11.817
M77   2.712   0.033
M87  12.513  12.400
M97  11.247  55.017
```

Comme pour `numpy`, il est possible d'opérer une sélection booléenne :

```
>>> messier.loc[messier['Con'] == 'UMa', ['NGC', 'Name']]
      NGC      Name
M
M40  Win4    Winnecke 4
M81  3031    Bode's Galaxy
M82  3034    Cigar Galaxy
M97  3587     Owl Nebula
M101 5457   Pinwheel Galaxy
M108 3556              NaN
M109 3992              NaN
>>> messier[messier['Season'].isin('winter spring'.split())].head(3)
      NGC Type  Mag  Size  Distance      RA      Dec  Con  Season      Name
M
M1  1952  Sn   8.4   5.0    1930.0   5.575  22.017  Tau  winter  Crab Nebula
M3  5272  Gc   6.2  16.2   10400.0  13.703  28.383  CVn  spring      NaN
M35 2168  Oc   5.3  28.0    859.0   6.148  24.333  Gem  winter      NaN
>>> messier.loc[lambdaf: N.radians(df.Size / 60) * df.Distance < 1].Name
M
M27    Dumbbell Nebula
M40    Winnecke 4
M57    Ring Nebula
M73              NaN
M76  Little Dumbbell Nebula
M78              NaN
```

(suite sur la page suivante)

(suite de la page précédente)

```

M97                Owl Nebula
Name: Name, dtype: object
>>> messier.query("(Mag < 5) & (Size > 60)").Name
M
M7                Ptolemy's Cluster
M24       Sagittarius Star Cloud
M31                Andromeda Galaxy
M42       Great Nebula in Orion
M44                Beehive Cluster
M45                Pleiades
Name: Name, dtype: object

```

Sélection	Syntaxe	Résultat
Colonne unique	<code>df.col</code> or <code>df[col]</code>	<code>pandas.Series</code>
Liste de colonnes	<code>df[[c1, ...]]</code>	<code>pandas.DataFrame</code>
Lignes par tranche	<code>df[slice]</code>	<code>pandas.DataFrame</code>
Label unique	<code>df.loc[label]</code>	<code>pandas.Series</code>
Liste de labels	<code>df.loc[[lab1, ...]]</code>	<code>pandas.DataFrame</code>
Labels par tranche	<code>df.loc[lab1:lab2]</code>	<code>pandas.DataFrame</code>
Ligne entière par n°	<code>df.iloc[i]</code>	<code>pandas.Series</code>
Ligne partielle par n°	<code>df.iloc[i, [j, ...]]</code>	<code>pandas.Series</code>
Valeur par labels	<code>df.at[lab, col]</code>	Scalaire
Valeur par n°	<code>df.iat[i, j]</code>	Scalaire
Ligne par sél. booléenne	<code>df.loc[sel]</code> or <code>df[sel]</code> or <code>df.query("sel")</code>	<code>pandas.DataFrame</code>

`pandas.DataFrame.drop()` permet d'éliminer une ou plusieurs colonnes d'un *dataframe* :

```

>>> messier.drop(['RA', 'Dec'], axis=1).head(3) # Élimination de colonnes
      NGC Type  Mag  Size  Distance  Con  Season  Name
M
M1  1952   Sn  8.4   5.0    1930.0  Tau  winter  Crab Nebula
M2  7089   Gc  6.5  12.9   11600.0  Aqr  autumn      NaN
M3  5272   Gc  6.2  16.2   10400.0  CVn  spring      NaN

```

`pandas.DataFrame.dropna()` et `pandas.DataFrame.fillna()` permettent de gérer les données manquantes (*NaN*) :

```

>>> messier.dropna(axis=0, how='any', subset=['NGC', 'Name']).head(3)
      NGC Type  Mag  Size  Distance      RA      Dec  Con  Season      Name
M
M1  1952   Sn  8.4   5.0    1930.0  5.575  22.017  Tau  winter  Crab Nebula
M6  6405   Oc  4.2  25.0     491.0  17.668 -32.217  Sco  summer  Butterfly Cluster
M7  6475   Oc  3.3  80.0     245.0  17.898 -34.817  Sco  summer  Ptolemy's Cluster
>>> messier.fillna('', inplace=True) # Remplace les NaN à la volée
>>> messier.head(3)
      NGC Type  Mag  Size  Distance      RA      Dec  Con  Season      Name
M
M1  1952   Sn  8.4   5.0    1930.0  5.575  22.017  Tau  winter  Crab Nebula
M2  7089   Gc  6.5  12.9   11600.0  21.558   0.817  Aqr  autumn
M3  5272   Gc  6.2  16.2   10400.0  13.703  28.383  CVn  spring

```

Référence : [Working with missing data](#)

Attention : par défaut, beaucoup d'opérations retournent une *copie* de la structure, sauf si l'opération se fait « sur place » (`inplace=True`). D'autres opérations d'accès retournent seulement une *nouvelle vue* des mêmes données.

```

>>> df = PD.DataFrame(N.arange(12).reshape(3, 4),
...                    index=list('abc'), columns=list('ABCD')); df
   A  B  C  D
a  0  1  2  3
b  4  5  6  7
c  8  9 10 11
>>> df.drop('a', axis=0)
   A  B  C  D
b  4  5  6  7
c  8  9 10 11
>>> colA = df['A'] # Nouvelle vue de la colonne 'A'
>>> colA += 1      # Opération sur place
>>> df             # la ligne 'a' est tjs là, la colonne 'A' a été modifiée
   A  B  C  D
a  1  1  2  3
b  5  5  6  7
c  9  9 10 11

```

Lien : [Returning a view versus a copy](#)

Indéxation hiérarchique

Références : Multi-index / Advanced Indexing

`pandas.MultiIndex` offre une indexation *hiérarchique*, permettant de stocker et manipuler des données avec un nombre arbitraire de dimensions dans des structures plus simples.

```

>>> saisons = messier.reset_index() # Élimine l'indexation actuelle
>>> saisons.set_index(['Season', 'Type'], inplace=True) # MultiIndex
>>> saisons.head(3)

```

		M	NGC	Mag	Size	Distance	RA	Dec	Con	Name
Season	Type									
winter	Sn	M1	1952	8.4	5.0	1930.0	5.575	22.017	Tau	Crab Nebula
autumn	Gc	M2	7089	6.5	12.9	11600.0	21.558	0.817	Aqr	
spring	Gc	M3	5272	6.2	16.2	10400.0	13.703	28.383	CVn	

Les informations contenues sont toujours les mêmes, mais structurées différemment :

```

>>> saisons.loc['spring'].head(3) # Utilisation du 1er label

```

	M	NGC	Mag	Size	Distance	RA	Dec	Con	Name
Type									
Gc	M3	5272	6.2	16.2	10400.0	13.703	28.383	CVn	
Ds	M40	Win4	8.4	0.8	156.0	12.373	58.083	UMa	Winnecke 4
El	M49	4472	8.4	8.2	18400000.0	12.497	8.000	Vir	

```

>>> saisons.loc['spring', 'El'].head(3) # Utilisation des 2 labels

```

	M	NGC	Mag	Size	Distance	RA	Dec	Con	Name
Season	Type								
spring	El	M49	4472	8.4	8.2	18400000.0	12.497	8.00	Vir
	El	M59	4621	9.6	4.2	18400000.0	12.700	11.65	Vir
	El	M60	4649	8.8	6.5	18400000.0	12.728	11.55	Vir

La fonction `pandas.DataFrame.xs()` permet des sélections sur les différents niveaux d'indexation :

```

>>> saisons.xs('spring', level='Season').head(3) # = saisons.loc['spring']

```

	M	NGC	Mag	Size	Distance	RA	Dec	Con	Name
Type									
Gc	M3	5272	6.2	16.2	10400.0	13.703	28.383	CVn	
Ds	M40	Win4	8.4	0.8	156.0	12.373	58.083	UMa	Winnecke 4
El	M49	4472	8.4	8.2	18400000.0	12.497	8.000	Vir	

(suite sur la page suivante)

(suite de la page précédente)

```
>>> saisons.xs('El', level='Type').head(3) # Sélection sur le 2e niveau
      M   NGC  Mag  Size  Distance    RA    Dec  Con Name
Season
autumn M32   221  8.1   7.0   920000.0  0.713  40.867  And
spring M49  4472  8.4   8.2  18400000.0 12.497   8.000  Vir
spring M59  4621  9.6   4.2  18400000.0 12.700  11.650  Vir
```

Le (multi-)index n'est pas nécessairement trié à sa création, `pandas.sort_index()` permet d'y remédier :

```
>>> saisons[['M', 'NGC', 'Name']].head()
      M   NGC      Name
Season Type
winter Sn   M1  1952  Crab Nebula
autumn Gc   M2  7089
spring Gc   M3  5272
summer Gc   M4  6121
Gc      M5  5904
>>> saisons[['M', 'NGC', 'Name']].sort_index().head()
      M   NGC      Name
Season Type
autumn El   M32   221
      El  M110   205
      Gc   M2  7089
      Gc  M15  7078  Great Pegasus Globular
      Gc  M30  7099
```

5.1.3 Manipulation des données

Références : [Essential Basic Functionality](#)

Comme dans `numpy`, il est possible de modifier les valeurs, ajouter/retirer des colonnes ou des lignes, tout en gérant les données manquantes.

Note : l'interopérabilité entre `pandas` et `numpy` est totale, toutes les fonctions Numpy peuvent prendre une structure Pandas en entrée, et s'appliquer aux colonnes appropriées :

```
>>> N.mean(messier, axis=0) # Moyenne sur les lignes → Series indexée sur les colonnes
Mag      7.492727e+00
Size     1.771909e+01
Distance 4.574883e+06
RA       1.303774e+01
Dec      9.281782e+00
dtype: float64
```

```
>>> N.random.seed(0)
>>> df = PD.DataFrame(
    {'one': PD.Series(N.random.randn(3), index=['a', 'b', 'c']),
     'two': PD.Series(N.random.randn(4), index=['a', 'b', 'c', 'd']),
     'three': PD.Series(N.random.randn(3), index=['b', 'c', 'd'])})
>>> df
      one      three      two
a  1.764052      NaN  2.240893
b  0.400157 -0.151357  1.867558
c  0.978738 -0.103219 -0.977278
d      NaN  0.410599  0.950088
>>> df['four'] = df['one'] + df['two']; df # Création d'une nouvelle colonne
      one      three      two      four
```

(suite sur la page suivante)

(suite de la page précédente)

```

a  1.764052      NaN  2.240893  4.004946
b  0.400157 -0.151357  1.867558  2.267715
c  0.978738 -0.103219 -0.977278  0.001460
d      NaN  0.410599  0.950088      NaN
>>> df.sub(df.loc['b'], axis='columns') # Soustraction d'une ligne à toutes les colonnes
↳ (axis=1)
      one      three      two      four
a  1.363895      NaN  0.373335  1.737230
b  0.000000  0.000000  0.000000  0.000000
c  0.578581  0.048138 -2.844836 -2.266255
d      NaN  0.561956 -0.917470      NaN
>>> df.sub(df['one'], axis='index') # Soustraction d'une colonne à toutes les lignes (axis=0
↳ ou 'rows')
      one      three      two      four
a  0.0      NaN  0.476841  2.240893
b  0.0 -0.551514  1.467401  1.867558
c  0.0 -1.081957 -1.956016 -0.977278
d  NaN      NaN      NaN      NaN

```

```

>>> df.sort_values(by='a', axis=1) # Tri des colonnes selon les valeurs de la ligne 'a'
      one      two      three
a  1.764052  2.240893      NaN
b  0.400157  1.867558 -0.151357
c  0.978738 -0.977278 -0.103219
d      NaN  0.950088  0.410599
>>> df.min(axis=1) # Valeur minimale le long des colonnes
a    1.764052
b   -0.151357
c   -0.977278
d    0.410599
dtype: float64
>>> df.idxmin(axis=1) # Colonne des valeurs minimales le long des colonnes
a      one
b     three
c       two
d     three
dtype: object

```

```

>>> df.mean(axis=0) # Moyenne sur toutes les lignes (gestion des données manquantes)
one      1.047649
three    0.052007
two      1.020315
dtype: float64

```

Note : Si les bibliothèques d'optimisation de performances `Bottleneck` et `NumExpr` sont installées, `pandas` en bénéficiera de façon transparente.

5.1.4 Regroupement et agrégation de données

Histogramme et discrétisation

Compter les objets Messier par constellation avec `pandas.value_counts()` :

```

>>> PD.value_counts(messier['Con']).head(3)
Sgr      15
Vir      11

```

(suite sur la page suivante)

(suite de la page précédente)

```
Com      8
Name: Con, dtype: int64
```

Partitionner les objets en 3 groupes de magnitude (par valeurs : `pandas.cut()`, par quantiles : `pandas.qcut()`), et les compter :

```
>>> PD.value_counts(PD.cut(messier['Mag'], 3)).sort_index() # Par valeurs
(1.591, 4.467]      6
(4.467, 7.333]     40
(7.333, 10.2]      64
Name: Mag, dtype: int64
>>> PD.value_counts(PD.qcut(messier['Mag'], [0, .3, .6, 1])).sort_index() # Par quantiles
(1.599, 6.697]      36
(6.697, 8.4]        38
(8.4, 10.2]         36
Name: Mag, dtype: int64
```

Group-by

Référence : Group By: split-apply-combine

Pandas offre la possibilité de regrouper les données selon divers critères (`pandas.DataFrame.groupby()`), de les agréger au sein de ces groupes et de stocker les résultats dans une structure avec indéxation hiérarchique (`pandas.DataFrame.agg()`).

```
>>> seasonGr = messier.groupby('Season') # Retourne un DataFrameGroupBy
>>> seasonGr.groups
{'autumn': Index([u'M2', u'M15', ..., u'M103', u'M110'],
                 dtype='object', name=u'M'),
 'spring': Index([u'M3', u'M40', ..., u'M108', u'M109'],
                 dtype='object', name=u'M'),
 'summer': Index([u'M4', u'M5', ..., u'M102', u'M107'],
                 dtype='object', name=u'M'),
 'winter': Index([u'M1', u'M35', ..., u'M79', u'M93'],
                 dtype='object', name=u'M')}
>>> seasonGr.size()
Season
autumn    14
spring    38
summer    40
winter    18
dtype: int64
>>> seasonGr.get_group('winter').head(3)
   Con  Dec  Distance  Mag  NGC      Name      RA  Size Type
M
M1   Tau  22.017   1930.0  8.4  1952  Crab Nebula  5.575   5.0  Sn
M35  Gem  24.333    859.0  5.3  2168             6.148  28.0  Oc
M36  Aur  34.133   1260.0  6.3  1960             5.602  12.0  Oc
>>> seasonGr['Size'].agg([N.mean, N.std]) # Taille moyenne et stddev par groupe
              mean      std
Season
autumn    24.307143   31.472588
spring     7.197368    4.183848
summer    17.965000   19.322400
winter    34.261111   29.894779
>>> seasonGr.agg({'Size': N.max, 'Mag': N.min})
      Mag  Size
Season
autumn   3.4  120.0
```

(suite sur la page suivante)

(suite de la page précédente)

```
spring 6.2 22.0
summer 3.3 90.0
winter 1.6 110.0
```

```
>>> magGr = messier.groupby(
...     [PD.qcut(messier['Mag'], [0, .3, .6, 1],
...         labels='Bright Medium Faint'.split()),
...     "Season"])
>>> magGr['Mag', 'Size'].agg({'Mag': ['count', 'mean'],
...     'Size': [N.mean, N.std]})
```

		Mag count	mean	Size mean	std
Mag	Season				
Bright	autumn	6	5.316667	45.200000	40.470878
	spring	1	6.200000	16.200000	NaN
	summer	15	5.673333	30.840000	26.225228
	winter	13	5.138462	42.923077	30.944740
Faint	autumn	4	9.225000	8.025000	4.768910
	spring	30	9.236667	5.756667	2.272578
	summer	7	8.971429	7.814286	9.135540
	winter	3	8.566667	9.666667	6.429101
Medium	autumn	4	7.500000	9.250000	3.304038
	spring	7	7.714286	12.085714	5.587316
	summer	18	7.366667	11.183333	4.825453
	winter	2	7.550000	14.850000	8.697413

Tableau croisé (*Pivot table*)

Référence : Reshaping and Pivot Tables

Calculer la magnitude et la taille moyennes des objets Messier selon leur type avec `pandas.DataFrame.pivot_table()` :

```
>>> messier['Mag Size Type'.split()].pivot_table(columns='Type')
Type  As  Ba      Di  ...  Pl  Sn      Sp
Mag   9.0  9.85  7.216667  ...  9.050  8.4  8.495652
Size  2.8  4.80  33.333333  ...  3.425  5.0  15.160870
```

5.1.5 Visualisations

Exemple :

Démonstration Pandas/Seaborn (`pokemon.ipynb`) sur le jeu de données `Pokemon.csv`.

Références :

- Visualization
- Seaborn: statistical data visualization

Autres exemples de visualisation de jeux de données complexes (utilisation de pandas et seaborn)

- Iris Dataset
- Histoire des sujets télévisés 

Liens

- [Pandas Tutorial](#)
- [Pandas Cookbook](#)
- [Pandas Lessons for New Users](#)
- [Practical Data Analysis](#)

Exercices :

- [Exercices for New Users](#)

5.1.6 Xarray

`xarray` est une bibliothèque pour la structuration de données *homogènes* de dimension arbitraire. Suivant la philosophie de la bibliothèque `Pandas` dont elle est issue (et dont elle dépend), elle permet notamment de nommer les différentes dimensions (*axes*) des tableaux (p.ex. `x.sum(axis='time')`), d'indexer les données (p.ex. `x.loc['M31']`), de naturellement gérer les opérations de *broadcasting*, des opérations de regroupement et d'agrégation (p.ex. `x.groupby(time.dayofyear).mean()`), une gestion plus facile des données manquantes et d'alignement des tableaux indexés (p.ex. `align(x, y, join='outer')`).

pandas excels at working with tabular data. That suffices for many statistical analyses, but physical scientists rely on N-dimensional arrays – which is where xarray comes in.

`xarray` fournit deux structures principales, héritées du format `netCDF` :

- `xarray.DataArray`, un tableau N-D indexé généralisant le `pandas.Series` ;
- `xarray.DataSet`, un dictionnaire regroupant plusieurs `DataArray` alignés selon une ou plusieurs dimensions, et similaire au `pandas.DataFrame`.

Note : La convention utilisée ici est « `import xarray as X` ».

```
>>> N.random.seed(0)
>>> data = X.DataArray(N.arange(3*4, dtype=float).reshape((3, 4)), # Tableau de données
...                    dims=('x', 'y'),                             # Nom des dimensions
...                    coords={'x': list('abc')},                  # Indexation des coordonnées en 'x'
...                    name='mesures',                             # Nom du tableau
...                    attrs=dict(author='Y. Copin'))               # Métadonnées
>>> data
<xarray.DataArray 'mesures' (x: 3, y: 4)>
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
Coordinates:
  * x          (x) |S1 'a' 'b' 'c'
Dimensions without coordinates: y
Attributes:
  author:      Y. Copin
>>> data.to_pandas()          # Conversion en DataFrame à indexation simple
y    0    1    2    3
x
a  0.0  1.0  2.0  3.0
b  4.0  5.0  6.0  7.0
c  8.0  9.0 10.0 11.0
>>> data.to_dataframe()      # Conversion en DataFrame multi-indexé (hiérarchique)
mesures
x y
a 0    0.0
  1    1.0
  2    2.0
```

(suite sur la page suivante)

(suite de la page précédente)

```

3      3.0
b 0      4.0
  1      5.0
  2      6.0
  3      7.0
c 0      8.0
  1      9.0
  2     10.0
  3     11.0
>>> data.dims
('x', 'y')
>>> data.coords
Coordinates:
  * x      (x) |S1 'a' 'b' 'c'
>>> data.values
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
>>> data.attrs
OrderedDict([('author', 'Y. Copin')])

```

```

>>> data[:, 1] # Accès par indices
<xarray.DataArray 'mesures' (x: 3)>
array([ 1.,  5.,  9.])
Coordinates:
  * x      (x) |S1 'a' 'b' 'c'
>>> data.loc['a':'b', -1] # Accès par labels
<xarray.DataArray 'mesures' (x: 2)>
array([ 3.,  7.])
Coordinates:
  * x      (x) |S1 'a' 'b'
>>> data.sel(x=['a', 'c'], y=2)
<xarray.DataArray 'mesures' (x: 2)>
array([ 2., 10.])
Coordinates:
  * x      (x) |S1 'a' 'c'

```

```

>>> data.mean(dim='x') # Moyenne le long de la dimension 'x' = data.mean(axis=0)
<xarray.DataArray 'mesures' (y: 4)>
array([ 4.,  5.,  6.,  7.])
Dimensions without coordinates: y

```

```

>>> data2 = X.DataArray(N.arange(6).reshape(2, 3) * 10,
...                      dims=('z', 'x'), coords={'x': list('abd')})
>>> data2.to_pandas()
x   a   b   d
z
0   0  10  20
1  30  40  50
>>> data.to_pandas() # REMINDER
y   0   1   2   3
x
a  0.0  1.0  2.0  3.0
b  4.0  5.0  6.0  7.0
c  8.0  9.0 10.0 11.0
>>> data2.values + data.values # Opération sur des tableaux numpy incompatibles
ValueError: operands could not be broadcast together with shapes (2,3) (3,4)
>>> data2 + data # Alignement automatique sur les coord. communes!
<xarray.DataArray (z: 2, x: 2, y: 4)>
array([[[ 0.,  1.,  2.,  3.],

```

(suite sur la page suivante)

(suite de la page précédente)

```

    [ 14., 15., 16., 17.]],
    [[ 30., 31., 32., 33.],
    [ 44., 45., 46., 47.]])

```

Coordinates:

```
* x      (x) object 'a' 'b'
```

Dimensions without coordinates: z, y

```
>>> data['isSmall'] = data.sum(dim='y') < 10; data # Booléen "Somme sur y < 10"
```

```
<xarray.DataArray 'mesures' (x: 3, y: 4)>
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

Coordinates:

```
* x      (x) |S1 'a' 'b' 'c'
```

```
isSmall  (x) bool True False False
```

Dimensions without coordinates: y

```
>>> data.groupby('isSmall').mean(dim='x') # Regroupement et agrégation
```

```
<xarray.DataArray 'mesures' (isSmall: 2, y: 4)>
```

```
array([[ 6.,  7.,  8.,  9.],
       [ 0.,  1.,  2.,  3.]])
```

Coordinates:

```
* isSmall (isSmall) object False True
```

Dimensions without coordinates: y

Exemples plus complexes :

— [Examples Data](#)

Note : N'oubliez pas de citer `xarray` dans vos publications et présentations.

5.2 Astropy

`Astropy` est une bibliothèque astronomique maintenue par la communauté et visant à fédérer les efforts jusque là disparates. Elle offre en outre une interface unifiée à des [bibliothèques affiliées](#) plus spécifiques.

5.2.1 Tour d'horizon

- Structures de base :
 - `astropy.constants` : constantes fondamentales (voir également `scipy.constants`);
 - `astropy.units` : unités et quantités dimensionnées;
 - `astropy.nddata` : extension des `numpy.ndarray` (incluant métadonnées, masque, unité, incertitude, etc.);
 - `astropy.table` : tableaux hétérogènes;
 - `astropy.time` : manipulation du temps et des dates;
 - `astropy.coordinates` : systèmes de coordonnées;
 - `astropy.wcs` : *World Coordinate System*;
 - `astropy.modeling` : modèles et ajustements;
 - `astropy.analytic_functions` : fonctions analytiques.
- Entrées/sorties :
 - `astropy.io.fits` : fichiers FITS;
 - `astropy.io.ascii` : tables ASCII;
 - `astropy.io.votable` : XML VO-tables;
 - `astropy.io.misc` : divers;

- `astropy.vo` : accès au *Virtual Observatory*.
- Calculs astronomiques :
 - `astropy.cosmology` : calculs cosmologiques ;
 - `astropy.convolution` : convolution et filtrage ;
 - `astropy.visualization` : visualisation de données ;
 - `astropy.stats` : méthodes astrostatistiques.

5.2.2 Démonstration

Démonstration Astropy (`astropy.ipynb`)

Voir également :

- AstroBetter tutorials

Note : N'oubliez pas de citer [*Astropy13*] ou de mentionner l'utilisation d'astropy dans vos publications et présentations.

5.3 Autres bibliothèques scientifiques

Python est maintenant très largement utilisé par la communauté scientifique, et dispose d'innombrables bibliothèques dédiées aux différents domaines de la physique, chimie, etc. :

- Astronomie : `Kapteyn`, `AstroML`, `HyperSpy` ;
- Mécanique quantique : `QuTiP` ;
- Électromagnétisme : `EMpy` ;
- Optique physique : `Physical Optics Propagation in PYthon`, `OpticsPy` ;
- *PDE solver* : `FiPy`, `SfePy` ;
- Analyse statistique bayésienne : `PyStan` ;
- *Markov Chain Monte-Carlo* : `emcee`, `PyMC3` ;
- *Machine Learning* : `mlpy`, `milk`, `MDP`, `Keras` et autres modules d'intelligence artificielle ;
- Calcul symbolique : `sympy` (voir également ce [tutoriel sympy](#)) et `sage` ;
- `PyROOT` ;
- `High Performance Computing in Python` ;
- Etc.

Notes de bas de page et références bibliographiques

Table des matières

- *Développer en Python*
- *Le zen du Python*
 - *Us et coutumes*
 - *Principes de conception logicielle*
- *Développement piloté par les tests*
- *Outils de développement*
 - Integrated Development Environment
 - *Vérification du code*
 - *Débogage*
 - *Profilage et optimisation*
 - *Documentation*
 - Python packages
 - *Système de gestion de versions*
 - *Intégration continue*
- *Python 2 vs. Python 3*

6.1 Le zen du Python

Le *zen du Python* (**PEP 20**) est une série de 20 aphorismes¹ donnant les grands principes du Python :

```
>>> import this
```

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.

Dont seulement 19 ont été écrits.

7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one – and preferably only one – obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea – let's do more of those !

Une traduction libre en français :

1. Préférer le beau au laid,
2. l'explicite à l'implicite,
3. le simple au complexe,
4. le complexe au compliqué,
5. le déroulé à l'imbriqué,
6. l'aéré au compact.
7. La lisibilité compte.
8. Les cas particuliers ne le sont jamais assez pour violer les règles,
9. même s'il faut privilégier l'aspect pratique à la pureté.
10. Ne jamais passer les erreurs sous silence,
11. ou les faire taire explicitement.
12. Face à l'ambiguïté, ne pas se laisser tenter à deviner.
13. Il doit y avoir une – et si possible une seule – façon évidente de procéder,
14. même si cette façon n'est pas évidente à première vue, à moins d'être Hollandais.
15. Mieux vaut maintenant que jamais,
16. même si jamais est souvent mieux qu'immédiatement.
17. Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
18. Si l'implémentation s'explique facilement, c'est peut-être une bonne idée.
19. Les espaces de nommage sont une sacrée bonne idée, utilisons-les plus souvent !

6.1.1 Us et coutumes

- *Fail early, fail often, fail better!* (**raise**)
- *Easier to Ask for Forgiveness than Permission* (**try ... except**)
- le *Style Guide for Python Code* (**PEP 8**)
- [Google Python Style Guide](#)
- [The Best of the Best Practices \(BOBP\) Guide for Python](#)
- [The hitchhiker's guide to Python Code Style](#)

Quelques conseils supplémentaires :

- « Ne réinventez pas la roue, sauf si vous souhaitez en savoir plus sur les roues » (Jeff Atwood³) : cherchez si ce que vous voulez faire n'a pas déjà été fait (éventuellement en mieux...) pour vous concentrer sur *votre* valeur ajoutée, réutilisez le code (en citant évidemment vos sources), améliorez le, et contribuez en retour si possible !
- Écrivez des programmes pour les humains, pas pour les ordinateurs : codez *proprement*, structurez vos algorithmes, commentez votre code, utilisez des noms de variable qui ont un sens, soignez le style et le formatage, etc.
- Codez proprement *dès le début* : ne croyez pas que vous ne relirez jamais votre code (ou même que personne n'aura jamais à le lire), ou que vous aurez le temps de le refaire mieux plus tard...
- « L'optimisation prématurée est la source de tous les maux » (Donald Knuth⁴) : mieux vaut un code lent mais juste et maintenable qu'un code rapide et faux ou incompréhensible. Dans l'ordre absolu des priorités :
 1. *Make it work.*
 2. *Make it right.*
 3. *Make it fast.*
- Respectez le zen du python, il vous le rendra.

6.1.2 Principes de conception logicielle

La bonne conception d'un programme va permettre de gérer efficacement la complexité des algorithmes, de faciliter la maintenance (p.ex. correction des erreurs) et d'accroître les possibilités d'extension.

Modularité Le code est structuré en répertoires, fichiers, classes, méthodes et fonctions. Les blocs ne font pas plus de quelques dizaines de lignes, les fonctions ne prennent que quelques arguments, la structure logique n'est pas trop complexe, etc.

En particulier, le code doit respecter le *principe de responsabilité unique* : chaque entité élémentaire (classe, méthode, fonction) ne doit avoir qu'une unique raison d'exister, et ne pas tenter d'effectuer plusieurs tâches sans rapport direct (p.ex. lecture d'un fichier de données *et* analyse des données).

Flexibilité Une modification du comportement du code (p.ex. l'ajout d'une nouvelle fonctionnalité) ne nécessite de changer le code qu'en un nombre restreint de points.

Un code *rigide* devient rapidement difficile à faire évoluer, puisque chaque changement requiert un grand nombre de modifications.

Robustesse La modification du code en un point ne change pas de façon inopinée le comportement dans une autre partie *a priori* non reliée.

Un code *fragile* est facile à modifier, mais chaque modification peut avoir des conséquences inattendues et le code tend à devenir instable.

Réutilisabilité La réutilisation d'une portion de code ne demande pas de changement majeur, n'introduit pas trop de dépendances, et ne conduit pas à une duplication du code.

L'application de ces principes de développement dépend évidemment de l'objectif final du code :

- une bibliothèque de bas niveau, utilisée par de nombreux programmes (p.ex. `numpy`), favorisera la robustesse et la réutilisabilité aux dépens de la flexibilité : elle devra être particulièrement bien pensée, et ne pourra être modifiée qu'avec parcimonie ;
- inversement, un script d'analyse de haut niveau, d'utilisation restreinte, pourra être plus flexible mais plus fragile et peu réutilisable.

6.2 Développement piloté par les tests

Le *Test Driven Development* (TDD, ou en français « développement piloté par les tests ») est une méthode de programmation qui permet d'éviter des bogues *a priori* plutôt que de les résoudre *a posteriori*. Ce n'est pas une méthode propre à Python, elle est utilisée très largement par les programmeurs professionnels.

³ « Don't reinvent the wheel, unless you plan on learning more about wheels » – Jeff Atwood

⁴ « Premature optimization is the root of all evil » – Donald Knuth

Le cycle préconisé par TDD comporte cinq étapes :

1. Écrire un premier test ;
2. Vérifier qu'il échoue (puisque le code qu'il teste n'existe pas encore), afin de s'assurer que le test est valide et exécuté ;
3. Écrire un code minimal pour passer le test ;
4. Vérifier que le test passe correctement ;
5. Éventuellement « réusiner » le code (*refactoring*), c'est-à-dire l'améliorer (rapidité, lisibilité) tout en gardant les mêmes fonctionnalités.

«~Diviser pour mieux régner~» : chaque fonction, classe ou méthode est testée indépendamment. Ainsi, lorsqu'un nouveau morceau de code ne passe pas les tests qui y sont associés, il est certain que l'erreur provient de cette nouvelle partie et non des fonctions ou objets que ce morceau de code utilise. On distingue ainsi hiérarchiquement :

1. Les tests unitaires vérifient individuellement chacune des fonctions, méthodes, etc. ;
2. Les tests d'intégration évaluent les interactions entre différentes unités du programmes ;
3. Les tests système assurent le bon fonctionnement du programme dans sa globalité.

Il est très utile de transformer toutes les vérifications réalisées au cours du développement et du débogage sous forme de tests, ce qui permet de les réutiliser lorsque l'on veut compléter ou améliorer une partie du code. Si le nouveau code passe toujours les anciens tests, on est alors sûr de ne pas avoir cassé les fonctionnalités précédentes (régressions).

Nous avons déjà vu aux TD précédents plusieurs façons de rédiger des tests unitaires.

- Les `doctest` sont des exemples (assez simples) d'exécution de code inclus dans les *docstring* des classes ou fonctions :

```

1 def mean_power(alist, power=1):
2     r"""
3     Retourne la racine `power` de la moyenne des éléments de `alist` à
4     la puissance `power`:
5
6     .. math:: \mu = (\frac{1}{N} \sum_{i=0}^{N-1} x_i^p)^{1/p}
7
8     `power=1` correspond à la moyenne arithmétique, `power=2` au *Root
9     Mean Squared*, etc.
10
11     Exemples:
12     >>> mean_power([1, 2, 3])
13     2.0
14     >>> mean_power([1, 2, 3], power=2)
15     2.160246899469287
16     """
17
18     # *mean* = (somme valeurs**power / nb valeurs)**(1/power)
19     mean = (sum( val ** power for val in alist ) / len(alist)) ** (1 / power)
20
21     return mean

```

Les *doctests* peuvent être exécutés de différentes façons (voir ci-dessous) :

- avec le module standard `doctest` : `python -m doctest -v mean_power.py`
- avec `pytest` : `py.test --doctest-modules -v mean_power.py`
- avec `nose` : `nosetests --with-doctest -v mean_power.py`
- Les fonctions dont le nom commence par `test_` et contenant des `assert` sont automatiquement détectées par `pytest`². Cette méthode permet d'effectuer des tests plus poussés que les *doctests*, éventuellement dans un fichier séparé du code à tester. P.ex. :

```

1 def test_empty_init():
2     with pytest.raises(TypeError):

```

(suite sur la page suivante)

² `pytest` ne fait pas partie de la bibliothèque standard. Il vous faudra donc l'installer indépendamment si vous voulez l'utiliser.

(suite de la page précédente)

```

3     Animal()
4
5
6 def test_wrong_init():
7     with pytest.raises(ValueError):
8         Animal('Youki', 'lalala')
9
10
11 def test_init():
12     youki = Animal('Youki', 600)
13     assert youki.masse == 600
14     assert youki.vivant
15     assert youki.estVivant()
16     assert not youki.empoisonne

```

Les tests sont exécutés via `py.test programme.py`.

- Le module `unittest` de la bibliothèque standard permet à peu près la même chose que `pytest`, mais avec une syntaxe souvent plus lourde. `unittest` est étendu par le module non-standard `nose`.

6.3 Outils de développement

Je fournis ici essentiellement des liens vers des outils pouvant être utiles pour développer en Python.

6.3.1 Integrated Development Environment

- `idle`, l'IDE intégré à Python
- `emacs` + `python-mode` pour l'édition, et `ipython` pour l'exécution de code (voir [Python Programming In Emacs](#))
- `spyder`
- `PythonToolKit`
- `pyCharm` (la version `community` est gratuite)
- Etc.

6.3.2 Vérification du code

Il s'agit d'outils permettant de vérifier *a priori* la validité stylistique et syntaxique du code, de mettre en évidence des constructions dangereuses, les variables non-définies, etc. Ces outils ne testent pas nécessairement la validité des algorithmes et de leur mise en oeuvre...

- `pycodestyle` (ex-`pep8`) et `autopep8`
- `pyflakes`
- `pychecker`
- `pylint`

6.3.3 Débogage

Les débogueurs permettent de se « plonger » dans un code en cours d'exécution ou juste après une erreur (analyse post-mortem).

- Module de la bibliothèque standard : `pdb`
Pour déboguer un script, il est possible de l'exécuter sous le contrôle du débogueur `pdb` en s'interrompant dès la 1re instruction :

```
python -m pdb script.py
(Pdb)
```

Commandes (très similaires à `gdb`) :

- `h[elp]` [*command*] : aide en ligne ;
- `q[uit]` : quitter ;
- `r[un]` [*args*] : exécuter le programme avec les arguments ;
- `d[own]/u[p]` : monter/descendre dans le stack (empilement des appels de fonction) ;
- `p expression` : afficher le résultat de l'expression (`pp` : *pretty-print*) ;
- `l[ist]` [*first[, last]*] : afficher le code source autour de l'instruction courante (`ll` : *long list*) ;
- `n[ext]/s[tep]` : exécuter l'instruction suivante (sans y entrer/en y entrant) ;
- `unt[il]` : continuer l'exécution jusqu'à la ligne suivante (utile pour les boucles) ;
- `c[ont[inue]]` : continuer l'exécution (jusqu'à la prochaine interruption ou la fin du programme) ;
- `r[eturn]` : continuer l'exécution jusqu'à la sortie de la fonction ;
- `b[reak]` [*[[filename:]lineno | function[, condition]]*] : mettre en place un point d'arrêt (`tbreak` pour un point d'arrêt *temporaire*). Sans argument, afficher les points d'arrêts déjà définis ;
- `disable/enable` [*bpnumber*] : désactiver/réactiver tous ou un point d'arrêt ;
- `cl[ear]` [*bpnumber*] : éliminer tous ou un point d'arrêt ;
- `ignore` *bpnumber* [*count*] : ignorer un point d'arrêt une ou plusieurs fois ;
- `condition` *bpnumber* : ajouter une condition à un point d'arrêt ;
- `commands` [*bpnumber*] : ajouter des instructions à un point d'arrêt.
- Commandes `ipython` : `%run monScript.py, %debug, %pdb`
 Si un script exécuté sous `ipython` (commande `%run`) génère une exception, il est possible d'inspecter l'état de la mémoire au moment de l'erreur avec la commande `%debug`, qui lance une session `pdb` au point d'arrêt. `%pdb` lance systématiquement le débogueur à chaque exception.

L'activité de débogage s'intègre naturellement à la nécessité d'écrire des tests unitaires :

1. trouver un bogue ;
2. écrire un test qui aurait du être validé en l'absence du bogue ;
3. corriger le code jusqu'à validation du test.

Vous aurez alors au final corrigé le bug, *et* écrit un test s'assurant que ce bogue ne réapparaîtra pas inopinément.

6.3.4 Profilage et optimisation

Avertissement : *Premature optimization is the root of all evil* – Donald Knuth

Avant toute optimisation, s'assurer extensivement que le code fonctionne et produit les bons résultats dans tous les cas. S'il reste trop lent ou gourmand en mémoire *pour vos besoins*, il peut être nécessaire de l'optimiser.

Le **profilage** permet de déterminer le temps passé dans chacune des sous-fonctions d'un code (ou ligne par ligne : **line profiler**, ou selon l'utilisation de la mémoire : **memory profiler**), afin d'y identifier les parties qui gagneront à être optimisées.

- `python -O, __debug__, assert`
 Il existe un mode « optimisé » de python (option `-O`), qui pour l'instant ne fait pas grand chose (et n'est donc guère utilisé...) :
 - la variable interne `__debug__` passe de `True` à `False` ;
 - les instructions `assert` ne sont pas évaluées.
- `timeit` et `%timeit statement` sous `ipython` :

```
In [1]: def t1(n):
...:     l = []
...:     for i in range(n):
...:         l.append(i**2)
...:     return l
...:
```

(suite sur la page suivante)

(suite de la page précédente)

```

...: def t2(n):
...:     return [ i**2 for i in range(n) ]
...:
...: def t3(n):
...:     return N.arange(n)**2
...:
In [2]: %timeit t1(10000)
2.7 ms ± 12.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [3]: %timeit t2(10000)
2.29 ms ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [4]: %timeit t3(10000)
15.9 µs ± 120 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

```

- `cProfile` et `pstats`, et `%prun statement` sous `ipython` :

```

$ python -m cProfile calc_pi.py
3.1415925580959025
    10000005 function calls in 4.594 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    1.612    1.612    4.594    4.594 calc_pi.py:10(approx_pi)
      1    0.000    0.000    4.594    4.594 calc_pi.py:5(<module>)
10000000    2.982    0.000    2.982    0.000 calc_pi.py:5(recip_square)
      1    0.000    0.000    4.594    4.594 {built-in method builtins.exec}
      1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
↳objects}

```

- [Tutoriel de profilage](#)

Une fois identifiée la partie du code à optimiser, quelques conseils généraux :

- en cas de doute, favoriser la lisibilité aux performances ;
- utiliser des opérations sur les tableaux, plutôt que sur des éléments individuels (*vectorization*) : listes en compréhension, tableaux `numpy` (qui ont eux-mêmes été optimisés) ;
- `cython` est un langage de programmation **compilé** très similaire à python. Il permet d'écrire des extensions en C avec la facilité de python (voir notamment [Working with Numpy](#)) ;
- `numba` permet *automatiquement* de compiler *à la volée* (JIT (Just In Time)) du pur code python via le compilateur `LLVM`, avec une optimisation selon le CPU (éventuellement le GPU) utilisé, p.ex. :

```

from numba import guvectorize

@guvectorize(['void(float64[:,], intp[:,], float64[:,])'], '(n),()->(n)')
def move_mean(a, window_arr, out):
    ...

```

- à l'avenir, l'interpréteur CPython actuel sera éventuellement remplacé par `pypy`, basé sur une compilation JIT.

Lien :

[Performance tips](#)

6.3.5 Documentation

- Outils de documentation, ou comment transformer *automatiquement* un code-source bien documenté en une documentation fonctionnelle.
 - [Sphinx](#) ;

- [reStructuredText](#) for Sphinx;
- [Awesome Sphinx](#);
- [apidoc](#) (documentation automatique).
- Conventions de documentation :
 - *Docstring convention* : [PEP 257](#);
 - [Documenting Your Project Using Sphinx](#);
 - [A Guide to NumPy/SciPy Documentation](#);
 - [Sample doc](#) (matplotlib).

Lien :

[Documentation Tools](#)

6.3.6 Python packages

Comment installer/créer des modules externes :

- [pip](#);
- [Hitchhiker's Guide to Packaging](#);
- [Packaging a python library](#);
- [Cookiecutter](#) est un générateur de squelettes de projet via des *templates* (pas uniquement pour Python);
- [cx-freeze](http://cx-freeze.readthedocs.io/) <<http://cx-freeze.readthedocs.io/>>, pour générer un exécutable à partir d'un script.

6.3.7 Système de gestion de versions

La gestion des versions du code permet de suivre avec précision l'historique des modifications du code (ou de tout autre projet), de retrouver les changements critiques, de développer des branches alternatives, de faciliter le travail collaboratif, etc.

[Git](#) est un VCS (Version Controlling System) particulièrement performant (p.ex. utilisé pour le développement du [noyau Linux](#) ⁵). Il est souvent couplé à un dépôt en ligne faisant office de dépôt de référence et de solution de sauvegarde, et offrant généralement des solutions d'intégration continue, p.ex. :

- les très célèbres [GitHub](#) et [GitLab](#), gratuits pour les projets libres;
- pour des projets liés à votre travail, je conseille plutôt des dépôts directement gérés par votre institution, p.ex. [GitLab-IN2P3](#).

[Git](#) mérite un cours en soi, et devrait être utilisé très largement pour l'ensemble de vos projets (p.ex. rédaction d'articles, de thèse de cours, fichiers de configuration, tests numériques, etc.).

Quelques liens d'introduction :

- [Pro-Git book](#), le livre « officiel »;
- [Git Immersion](#);
- [Git Magic](#).

6.3.8 Intégration continue

L'intégration continue est un ensemble de pratiques de développement logiciel visant à s'assurer de façon systématique que chaque modification du code n'induit aucune *régression*, et passe l'ensemble des tests. Cela passe généralement par la mise en place d'un système de gestion des sources, auquel est accolé un mécanisme automatique de compilation (*build*), de déploiement sur les différentes infrastructures, d'exécution des tests (unitaires, intégration, fonctionnels, etc.) et de mise à disposition des résultats, de mise en ligne de la documentation, etc.

La plupart des développements des logiciels *open source* majeurs se fait maintenant sous intégration continue en utilisant des services en ligne directement connectés au dépôt source. Exemple sur [Astropy](#) :

Et maintenant du [code Windows](#)!

- [Travis CI](#) intégration continue ;
- [Coveralls](#) taux de couverture des tests unitaires ;
- [Readthedocs](#) documentation en ligne ;
- [Depsy](#) mise en valeur du développement logiciel dans le monde académique (*measure the value of software that powers science*).

6.4 Python 2 vs. Python 3

Si votre code est encore sous python-2.x, il existe de nombreux outils permettant de faciliter la transition vers 3.x :

- L'interpréteur Python 2.7 dispose d'une option `-3` mettant en évidence dans un code les parties qui devront être modifiées pour un passage à Python 3.
- Le script `2to3` permet également d'automatiser la conversion du code 2.x en 3.x.
- La bibliothèque standard `__future__` permet d'introduire des constructions 3.x dans un code 2.x, p.ex. :

```
from __future__ import print_function # Fonction print()
from __future__ import division      # Division non-euclidienne

print(1/2)                           # Affichera '0.5'
```


- La bibliothèque *non* standard `six` fournit une couche de compatibilité 2.x-3.x, permettant de produire de façon transparente un code compatible simultanément avec les deux versions.

Avertissement : Si vous vous lancez dans un nouveau développement, il est dorénavant indispensable d'utiliser exclusivement Python 3.

Liens

- [Py3 Readiness](#) : liste (réduite) des bibliothèques encore non-compatibles avec Python 3
- [Porting Python 2 Code to Python 3](#)
- [The Conservative Python 3 Porting Guide](#)
- [Python 2/3 compatibility](#)


Références supplémentaires

Voici une liste très partielle de documents Python disponibles en ligne. La majorité des liens sont en anglais, quelques-uns  sont en français.

7.1 Documentation générale

- Python
- Python Wiki
- Python Frequently Asked Questions
- The Python Package Index

7.2 Listes de liens

- Python facile (2005) 
- Python: quelques références, trucs et astuces: (2014) 
- Improving your programming style in Python (2014)
- Starter Kit (py4science) (2010)
- Learning Python For Data Science (2016)
- Awesome Python
- A curated list of courses on Python
- Real Python tutorials

ipython

- IPython tutorial
- IPython cookbook
- IPython en ligne
- IPython quick refsheets



Expressions rationnelles

- [regex tester](#)

Python 3.x




- [10 awesome features of Python that you can't use because you refuse to upgrade to Python 3](#)

7.3 Livres libres

- [How to Think Like a Computer Scientist](#)
 - [Wikibook](#)
 - [Interactive edition](#)
- [Dive into Python](#)
- [10 Free Python Programming Books](#)
- [A Python Book](#)
- [Start Programming with Python](#)
- [Learn Python the Hard Way](#)
- [Python for Informatics: Exploring Information](#)
- [Intermediate Python](#)
- [The Best Python Books](#)
- [Apprendre à programmer avec Python](#) 
- [Programmation Python](#) 

7.4 Cours en ligne

7.4.1 Python

- [Python Tutorial \(v3.7\)](#)
- [Apprenez à programmer en Python](#)
- [Débuter avec Python au lycée](#) 
- [Présentation de Python](#) 
- [Introduction à Python pour la programmation scientifique](#) 
- [Begginer's Guide](#)
- [DIY python workshop](#)
- [Google's Python Class](#)
- [CheckIO, pour apprendre la programmation Python en s'amusant !](#)
- [Python Testing Tools](#)
- [Python Programming \(Code Academy\)](#)

7.4.2 Scientifique

- [Python Scientific Lecture Notes](#)
- [Handbook of the Physics Computing Course \(2002\)](#)
- [Practical Scientific Computing in Python](#)
- [Computational Physics with Python \(avec exercices\)](#)
- [SciPy tutorials \(numpy, scipy, matplotlib, ipython\)](#) : 2011, 2012, 2013,
- [Advance Scientific Programming in Python](#)
- [Lectures on Computational Economics \(avec exercices\)](#)
- [Intro to Python for Data Science \(DataCamp avec vidéos et exercices\)](#)
- [Python for Data Science](#)
- [Learning Python For Data Science](#)

- Computational Statistics in Python
- Python Data Science Handbook

En français

- Formation à Python scientifique 
- NumPy et SciPy 
- La programmation scientifique avec Python 

Astrophysique et physique des hautes énergies

- Practical Python for Astronomers
- Astropy tutorials
- Python for Euclid 2016
- Advanced software programming for astrophysics and astroparticle physics, *ASTERICS-OBELICS International School*, 2017, 2018

7.4.3 Snippets

- Python cheatsheets

8.1 Mean power (fonction, argparse)

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  Exemple de script (shebang, docstring, etc.) permettant une
6  utilisation en module (`import mean_power`) et en exécutable (`python
7  mean_power.py -h`);
8  """
9
10
11 def mean_power(alist, power=1):
12     r"""
13     Retourne la racine `power` de la moyenne des éléments de `alist` à
14     la puissance `power`:
15
16     .. math:: \mu = (\frac{1}{N} \sum_{i=0}^{N-1} x_i^p)^{1/p}
17
18     `power=1` correspond à la moyenne arithmétique, `power=2` au *Root
19     Mean Squared*, etc.
20
21     Exemples:
22     >>> mean_power([1, 2, 3])
23     2.0
24     >>> mean_power([1, 2, 3], power=2)
25     2.160246899469287
26     """
27
28     # *mean* = (somme valeurs**power / nb valeurs)**(1/power)
29     mean = (sum( val ** power for val in alist ) / len(alist)) ** (1 / power)
30
31     return mean
32
33
34 if __name__ == '__main__':
35

```

(suite sur la page suivante)

(suite de la page précédente)

```

36  # start-argparse
37  import argparse
38
39  parser = argparse.ArgumentParser()
40  parser.add_argument('list', nargs='*', type=float, metavar='nombres',
41                      help="Liste de nombres à moyenner")
42  parser.add_argument('-i', '--input', nargs='?', type=argparse.FileType('r'),
43                      help="Fichier contenant les nombres à moyenner")
44  parser.add_argument('-p', '--power', type=float, default=1.,
45                      help="Puissance de la moyenne (%default)")
46
47  args = parser.parse_args()
48  # end-argparse
49
50  if args.input:      # Lecture des coordonnées du fichier d'entrée
51      # Le fichier a déjà été ouvert en lecture par argparse (type=file)
52      try:
53          args.list = [float(x) for x in args.input
54                       if not x.strip().startswith('#')]
55      except ValueError:
56          parser.error(
57              "Impossible de déchiffrer la ligne '{}' du fichier '{}'".
58              .format(x, args.input))
59
60  # Vérifie qu'il y a au moins un nombre dans la liste
61  if not args.list:
62      parser.error("La liste doit contenir au moins un nombre")
63
64  # Calcul
65  moyenne = mean_power(args.list, args.power)
66
67  # Affichage du résultat
68  print("La moyenne puissance 1/{} des {} nombres à la puissance {} est {}".format(
69      args.power, len(args.list), args.power, moyenne))

```

Source : mean_power.py

8.2 Formes (POO)

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  Exemple de Programmation Orientée Objet.
6  """
7
8
9  # Définition d'une classe =====
10
11  class Forme:
12
13      """Une forme plane, avec éventuellement une couleur."""
14
15      def __init__(self, couleur=None):
16          """Initialisation d'une Forme, sans couleur par défaut."""
17
18          if couleur is None:
19              self.couleur = 'indéfinie'
20          else:

```

(suite sur la page suivante)

(suite de la page précédente)

```

21         self.couleur = couleur
22
23     def __str__(self):
24         """
25         Surcharge de la fonction `str()`: l'affichage *informel* de
26         l'objet dans l'interpréteur, p.ex. `print(a)` sera résolu comme
27         `a.__str__()`
28
29         Retourne une chaîne de caractères.
30         """
31
32         return "forme encore indéfinie de couleur {}".format(self.couleur)
33
34     def change_couleur(self, newcolor):
35         """Change la couleur de la Forme."""
36
37         self.couleur = newcolor
38
39     def aire(self):
40         """
41         Renvoie l'aire de la Forme.
42
43         L'aire ne peut pas être calculée dans le cas où la forme n'est
44         pas encore spécifiée: c'est ce que l'on appelle une méthode
45         'abstraite', qui pourra être précisée dans les classes filles.
46         """
47
48         raise NotImplementedError(
49             "ATTENTION: impossible de calculer l'aire d'une forme indéfinie.")
50
51 class Rectangle(Forme):
52     """
53     Un Rectangle est une Forme particulière.
54
55     La classe-fille hérite des attributs et méthodes de la
56     classe-mère, mais peut les surcharger (i.e. en changer la
57     définition), ou en ajouter de nouveaux:
58
59     - la méthode `Rectangle.change_couleur()` dérive directement de
60       `Forme.change_couleur()`;
61     - `Rectangle.__str__()` surcharge `Forme.__str__()`;
62     - `Rectangle.aire()` définit la méthode jusqu'alors abstraite
63       `Forme.aire()`;
64     - `Rectangle.allonger()` est une nouvelle méthode propre à
65       `Rectangle`.
66     """
67
68     def __init__(self, longueur, largeur, couleur=None):
69         """
70         Initialisation d'un Rectangle longueur × largeur, sans couleur par
71         défaut.
72         """
73
74         # Initialisation de la classe parente (nécessaire pour assurer
75         # l'héritage)
76         Forme.__init__(self, couleur)
77
78         # Attributs propres à la classe Rectangle
79         self.longueur = longueur
80         self.largeur = largeur

```

(suite sur la page suivante)

(suite de la page précédente)

```

82
83     def __str__(self):
84         """Surcharge de `Forme.__str__()``."""
85
86         return "rectangle {}x{}, de couleur {}".format(
87             self.longueur, self.largeur, self.couleur)
88
89     def aire(self):
90         """
91         Renvoie l'aire du Rectangle.
92
93         Cette méthode définit la méthode abstraite `Forme.area()``,
94         pour les Rectangles uniquement.
95         """
96
97         return self.longueur * self.largeur
98
99     def allonger(self, facteur):
100         """Multiplie la *longueur* du Rectangle par un facteur"""
101
102         self.longueur *= facteur
103
104
105 if __name__ == '__main__':
106
107     s = Forme()                                # Forme indéfinie et sans couleur
108     print("s:", str(s))                        # Interprété comme `s.__str__()``
109     s.change_couleur('rouge')                  # On change la couleur
110     print("s après change_couleur:", str(s))
111     try:
112         print("aire de s:", s.aire())           # La méthode abstraite lève une exception
113     except NotImplementedError as err:
114         print(err)
115
116     q = Rectangle(1, 4, 'vert')                # Rectangle 1x4 vert
117     print("q:", str(q))
118     print("aire de q:", q.aire())
119
120     r = Rectangle(2, 1, 'bleu')                # Rectangle 2x1 bleu
121     print("r:", str(r))
122     print("aire de r:", r.aire())
123
124     print("allongement de r d'un facteur 2")
125     r.allonger(2)                             # r devient un rectangle 4x1
126     print("r:", str(r))

```

Source : formes.py

8.3 Cercle circonscrit (POO, argparse)

```

1  #!/usr/bin/env python3
2  #-*- coding: utf-8 -*-
3
4  """
5  Calcule le cercle circonscrit à 3 points du plan.
6
7  Ce script sert d'illustration à plusieurs concepts indépendants:
8
9  - un exemple de script (shebang, docstring, etc.) permettant une

```

(suite sur la page suivante)

(suite de la page précédente)

```

10 utilisation en module (`import circonscrit`) et en exécutable
11 (`python circonscrit.py -h`);
12 - des exemples de Programmation Orientée Objet: classe `Point` et la
13 classe héritière `Vector`;
14 - un exemple d'utilisation du module `argparse` de la bibliothèque
15 standard, permettant la gestion des arguments de la ligne de
16 commande;
17 - l'utilisation de tests unitaires sous la forme de `doctest` (tests
18 inclus dans les *docstrings* des éléments à tester).
19
20 Pour exécuter les tests unitaires du module:
21
22 - avec doctest: `python -m doctest -v circonscrit.py`;
23 - avec pytests: `py.test --doctest-modules -v circonscrit.py`;
24 - avec nose:    `nosetests --with-doctest -v circonscrit.py`.
25 """
26
27 __author__ = "Yannick Copin <y.copin@ipnl.in2p3.fr>"
28 __version__ = "Time-stamp: <2014-01-12 22:19 ycopin@lyonovae03.in2p3.fr>"
29
30 # Définition d'une classe =====
31
32
33 class Point:
34
35     """
36     Classe définissant un `Point` du plan, caractérisé par ses
37     coordonnées `x`, `y`.
38     """
39
40     def __init__(self, x, y):
41         """
42         Méthode d'instanciation à partir de deux coordonnées réelles.
43
44         >>> Point(0, 1)           # doctest: +ELLIPSIS
45         <...Point object at 0x...>
46         >>> Point(1 + 3j)
47         Traceback (most recent call last):
48         ...
49         TypeError: __init__() missing 1 required positional argument: 'y'
50         """
51
52         try: # Convertit les coords en `float`
53             self.x = float(x)
54             self.y = float(y)
55         except (ValueError, TypeError):
56             raise TypeError("Invalid input coordinates ({}, {})".format(x, y))
57
58     def __str__(self):
59         """
60         Surcharge de la fonction `str()`: l'affichage *informel* de l'objet
61         dans l'interpréteur, p.ex. `str(self)` sera résolu comme
62         `self.__str__()`
63
64         Retourne une chaîne de caractères.
65
66         >>> str(Point(1,2))
67         'Point (x=1.0, y=2.0)'
68         """
69
70         return "Point (x={p.x}, y={p.y})".format(p=self)

```

(suite sur la page suivante)

(suite de la page précédente)

```

71
72 def isOrigin(self):
73     """
74     Teste si le point est à l'origine en testant la nullité des deux
75     coordonnées.
76
77     Attention aux éventuelles erreurs d'arrondis: il faut tester
78     la nullité à la précision numérique près.
79
80     >>> Point(1,2).isOrigin()
81     False
82     >>> Point(0,0).isOrigin()
83     True
84     """
85
86     import sys
87
88     eps = sys.float_info.epsilon # Le plus petit float non nul
89
90     return ((abs(self.x) <= eps) and (abs(self.y) <= eps))
91
92 def distance(self, other):
93     """
94     Méthode de calcul de la distance du point (`self`) à un autre point
95     (`other`).
96
97     >>> A = Point(1,0); B = Point(1,1); A.distance(B)
98     1.0
99     """
100
101     # hypot(dx, dy) = sqrt(dx**2 + dy**2)
102     return ((self.x - other.x)**2 + (self.y - other.y)**2)**0.5
103
104
105 # Définition du point origine 0
106 0 = Point(0, 0)
107
108
109 # Héritage de classe =====
110
111
112 class Vector(Point):
113
114     """
115     Un `Vector` hérite de `Point` avec des méthodes additionnelles
116     (p.ex. la négation d'un vecteur, l'addition de deux vecteurs, ou
117     la rotation d'un vecteur).
118     """
119
120     def __init__(self, A, B):
121         """
122         Définit le vecteur `AB` à partir des deux points `A` et `B`.
123
124         >>> Vector(Point(1,0), Point(1,1)) # doctest: +ELLIPSIS
125         <...Vector object at 0x...>
126         >>> Vector(0, 1)
127         Traceback (most recent call last):
128         ...
129         AttributeError: 'int' object has no attribute 'x'
130         """
131

```

(suite sur la page suivante)

(suite de la page précédente)

```

132     # Initialisation de la classe parente
133     Point.__init__(self, B.x - A.x, B.y - A.y)
134
135     # Attribut propre à la classe dérivée
136     self.sqnorm = self.x ** 2 + self.y ** 2 # Norme du vecteur au carré
137
138     def __str__(self):
139         """
140         Surcharge de la fonction `str()`: `str(self)` sera résolu comme
141         `Vector.__str__(self)` (et non pas comme `Point.__str__(self)`)
142
143         >>> A = Point(1, 0); B = Point(1, 1); str(Vector(A, B))
144         'Vector (x=0.0, y=1.0)'
145         """
146
147         return "Vector (x={v.x}, y={v.y})".format(v=self)
148
149     def __add__(self, other):
150         """
151         Surcharge de l'opérateur binaire `{self} + {other}`: l'instruction
152         sera résolue comme `self.__add__(other)`.
153
154         On construit une nouvelle instance de `Vector` à partir des
155         coordonnées propres à l'objet `self` et à l'autre opérande
156         `other`.
157
158         >>> A = Point(1, 0); B = Point(1, 1)
159         >>> str(Vector(A, B) + Vector(B, 0)) # = Vector(A, 0)
160         'Vector (x=-1.0, y=0.0)'
161         """
162
163         return Vector(0, Point(self.x + other.x, self.y + other.y))
164
165     def __sub__(self, other):
166         """
167         Surcharge de l'opérateur binaire `{self} - {other}`: l'instruction
168         sera résolue comme `self.__sub__(other)`.
169
170         Attention: ne surcharge pas l'opérateur unaire `-{self}`, géré
171         par `__neg__`.
172
173         >>> A = Point(1, 0); B = Point(1, 1)
174         >>> str(Vector(A, B) - Vector(A, B)) # Différence
175         'Vector (x=0.0, y=0.0)'
176         >>> -Vector(A, B) # Négation
177         Traceback (most recent call last):
178         ...
179         TypeError: bad operand type for unary -: 'Vector'
180         """
181
182         return Vector(0, Point(self.x - other.x, self.y - other.y))
183
184     def __eq__(self, other):
185         """
186         Surcharge du test d'égalité `{self}=={other}`: l'instruction sera
187         résolue comme `self.__eq__(other)`.
188
189         >>> Vector(0, Point(0, 1)) == Vector(Point(1, 0), Point(1, 1))
190         True
191         """

```

(suite sur la page suivante)

(suite de la page précédente)

```

193     # On teste ici la nullité de la différence des 2
194     # vecteurs. D'autres tests auraient été possibles -- égalité
195     # des coordonnées, nullité de la norme de la différence,
196     # etc. -- mais on tire profit de la méthode héritée
197     # `Point.isOrigin()` testant la nullité des coordonnées (à la
198     # précision numérique près).
199     return (self - other).isOrigin()
200
201     def __abs__(self):
202         """
203         Surcharge la fonction `abs()` pour retourner la norme du vecteur.
204
205         >>> abs(Vector(Point(1, 0), Point(1, 1)))
206         1.0
207         """
208
209         # On pourrait utiliser sqrt(self.sqnorm), mais c'est pour
210         # illustrer l'utilisation de la méthode héritée
211         # `Point.distance`...
212         return Point.distance(self, 0)
213
214     def rotate(self, angle, deg=False):
215         """
216         Rotation (dans le sens trigonométrique) du vecteur par un `angle`,
217         exprimé en radians ou en degrés.
218
219         >>> Vector(Point(1, 0), Point(1, 1)).rotate(90, deg=True) == Vector(0, Point(-1, 0))
220         True
221         """
222
223         from cmath import rect # Bibliothèque de fonctions complexes
224
225         # On calcule la rotation en passant dans le plan complexe
226         z = complex(self.x, self.y)
227         phase = angle if not deg else angle / 57.29577951308232 # [rad]
228         u = rect(1., phase) # exp(i*phase)
229         zu = z * u # Rotation complexe
230
231         return Vector(0, Point(zu.real, zu.imag))
232
233
234     def circumscribedCircle(M, N, P):
235         """
236         Calcule le centre et le rayon du cercle circonscrit aux points
237         M, N, P.
238
239         Retourne: (centre [Point], rayon [float])
240
241         Lève une exception `ValueError` si le rayon ou le centre du cercle
242         circonscrit n'est pas défini.
243
244         >>> M = Point(-1, 0); N = Point(1, 0); P = Point(0, 1)
245         >>> C, r = circumscribedCircle(M, N, P) # Centre 0, rayon 1
246         >>> C.distance(0), round(r, 6)
247         (0.0, 1.0)
248         >>> circumscribedCircle(M, 0, N) # Indéfini
249         Traceback (most recent call last):
250         ...
251         ValueError: Undefined circumscribed circle radius.
252         """
253

```

(suite sur la page suivante)

(suite de la page précédente)

```

254 MN = Vector(M, N)
255 NP = Vector(N, P)
256 PM = Vector(P, M)
257
258 # Rayon du cercle circonscrit
259 m = abs(NP) # |NP|
260 n = abs(PM) # |PM|
261 p = abs(MN) # |MN|
262
263 d = (m + n + p) * (-m + n + p) * (m - n + p) * (m + n - p)
264 if d > 0:
265     rad = m * n * p / d**0.5
266 else:
267     raise ValueError("Undefined circumscribed circle radius.")
268
269 # Centre du cercle circonscrit
270 d = -2 * (M.x * NP.y + N.x * PM.y + P.x * MN.y)
271 if d == 0:
272     raise ValueError("Undefined circumscribed circle center.")
273
274 om2 = Vector(0, M).sqnorm # |OM|**2
275 on2 = Vector(0, N).sqnorm # |ON|**2
276 op2 = Vector(0, P).sqnorm # |OP|**2
277
278 x0 = -(om2 * NP.y + on2 * PM.y + op2 * MN.y) / d
279 y0 = (om2 * NP.x + on2 * PM.x + op2 * MN.x) / d
280
281 return (Point(x0, y0), rad) # (centre [Point], R [float])
282
283
284 if __name__ == '__main__':
285
286     # start-argparse
287     import argparse
288
289     parser = argparse.ArgumentParser(
290         usage="%(prog)s [-p/--plot] [-i/--input coordfile | x1,y1 x2,y2 x3,y3]",
291         description="Compute the circumscribed circle to 3 points in the plan.")
292     parser.add_argument('coords', nargs='*', type=str, metavar='x,y',
293                         help="Coordinates of point")
294     parser.add_argument('-i', '--input', nargs='?', type=argparse.FileType('r'),
295                         help="Coordinate file (one 'x,y' per line)")
296     parser.add_argument('-P', '--plot', action="store_true", default=False,
297                         help="Draw the circumscribed circle")
298     parser.add_argument('-T', '--tests', action="store_true", default=False,
299                         help="Run doc tests")
300     parser.add_argument('--version', action='version', version=__version__)
301
302     args = parser.parse_args()
303     # end-argparse
304
305     if args.tests: # Auto-test mode
306         import sys, doctest
307
308         fails, tests = doctest.testmod(verbose=True) # Run doc tests
309         sys.exit(fails > 0)
310
311     if args.input: # Lecture des coordonnées du fichier d'entrée
312         # Le fichier a déjà été ouvert en lecture par argparse (type=file)
313         args.coords = [coords for coords in args.input
314                        if not coords.strip().startswith('#')]

```

(suite sur la page suivante)

(suite de la page précédente)

```

315
316     if len(args.coords) != 3: # Vérifie le nb de points
317         parser.error("Specify 3 points by their coordinates 'x,y' (got {})"
318                     .format(len(args.coords)))
319
320     points = [] # Liste des points
321     for i, arg in enumerate(args.coords, start=1):
322         try: # Déchiffre de l'argument 'x,y'
323             x, y = (float(t) for t in arg.split(','))
324         except ValueError:
325             parser.error(
326                 "Cannot decipher coordinates #{}: '{}'.format(i, arg))
327
328         points.append(Point(x, y)) # Création du point et ajout à la liste
329         print("#{:d}: {}".format(i, points[-1])) # Affichage du dernier point
330
331     # Calcul du cercle circonscrit (lève une ValueError en cas de problème)
332     center, radius = circumscribedCircle(*points) # Délistage
333     print("Circumscribed circle: {}, radius: {}".format(center, radius))
334
335     if args.plot: # Figure
336         import matplotlib.pyplot as P
337
338         fig = P.figure()
339         ax = fig.add_subplot(1, 1, 1, aspect='equal')
340         # Points
341         ax.plot([p.x for p in points], [p.y for p in points], 'ko')
342         for i, p in enumerate(points, start=1):
343             ax.annotate("#{}".format(i), (p.x, p.y),
344                       xytext=(5, 5), textcoords='offset points')
345         # Cercle circonscrit
346         c = P.matplotlib.patches.Circle((center.x, center.y), radius=radius,
347                                         fc='none', ec='k')
348         ax.add_patch(c) # Cercle
349         ax.plot(center.x, center.y, 'r+') # Centre
350
351     P.show()

```

```

$ ./circonscrip.py -h
usage: circonscrip.py [-p/--plot] [-i/--input coordfile | x1,y1 x2,y2 x3,y3]

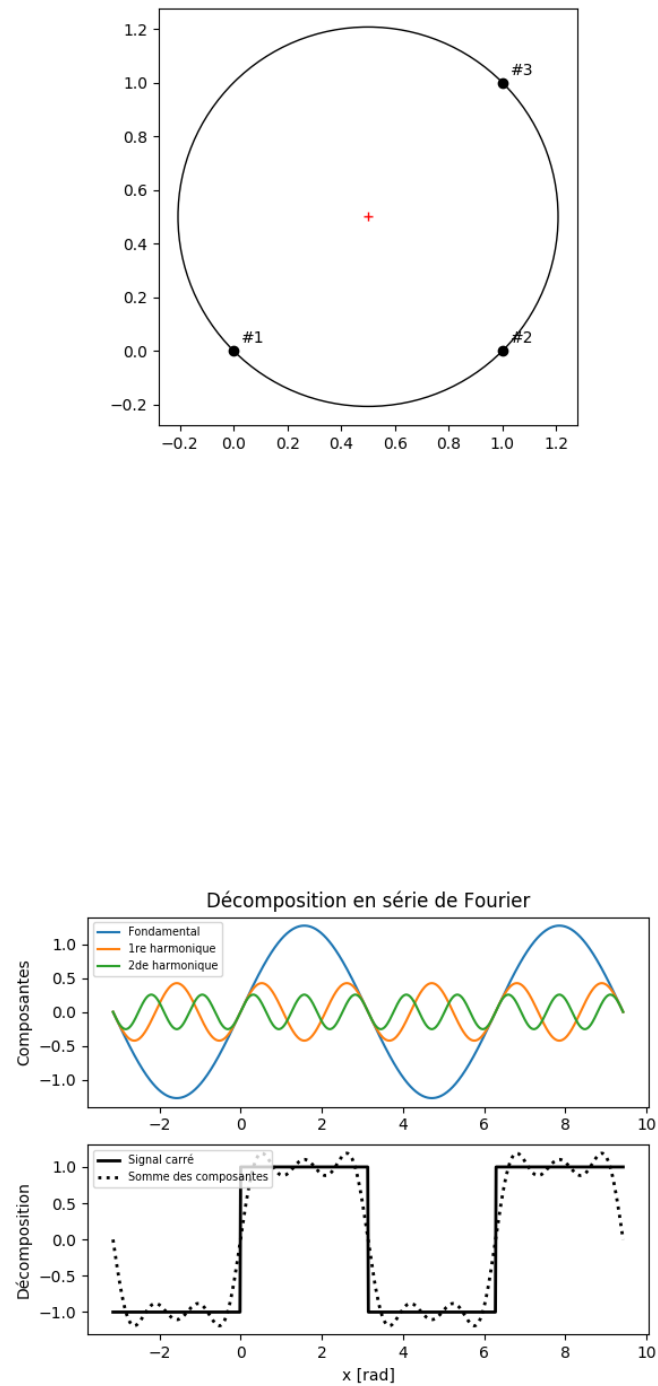
positional arguments:
  x,y                Coordinates of point

optional arguments:
  -h, --help          show this help message and exit
  -i [INPUT], --input [INPUT]
                        Coordinate file (one 'x,y' per line)
  -p, --plot           Draw the circumscribed circle
  --version            show program's version number and exit

$ ./circonscrip.py -p 0,0 1,0 1,1

```

Source : circonscrip.py

Fig. 8.1 – **Figure** : exemple de figure (charte graphique : `seaborn`)

8.4 Matplotlib

8.4.1 Figure (relativement) simple

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-01-09 15:03:52 ycopin>
4
5  """
6  Exemple un peu plus complexe de figure, incluant 2 axes, légendes, axes, etc.
7  """
8
9  import numpy as N
10 import matplotlib.pyplot as P
11
12 x = N.linspace(-N.pi, 3*N.pi, 2*360)
13
14 # Signal carré
15 y = N.sign(N.sin(x))          # = ± 1
16
17 # 3 premiers termes de la décomposition en série de Fourier
18 y1 = 4/N.pi * N.sin(x)      # Fondamentale
19 y2 = 4/N.pi * N.sin(3*x) / 3 # 1re harmonique
20 y3 = 4/N.pi * N.sin(5*x) / 5 # 2de harmonique
21 # Somme des 3 premières composantes
22 ytot = y1 + y2 + y3
23
24 # Figure
25 fig = P.figure()            # Création de la Figure
26
27 # 1er axe: composantes
28 ax1 = fig.add_subplot(2, 1, 1, # 1er axe d'une série de 2 × 1
29                        ylabel="Composantes",
30                        title="Décomposition en série de Fourier")
31 ax1.plot(x, y1, label="Fondamental")
32 ax1.plot(x, y2, label=u"1re harmonique")
33 ax1.plot(x, y3, label=u"2de harmonique")
34 ax1.legend(loc="upper left", fontsize="x-small")
35
36 # 2nd axe: décomposition
37 ax2 = fig.add_subplot(2, 1, 2, # 2d axe d'une série de 2 × 1
38                        ylabel=u"Décomposition",
39                        xlabel="x [rad]")
40 ax2.plot(x, y, lw=2, color='k', label="Signal carré")
41 ax2.plot(x, ytot, lw=2, ls=':', color='k', label="Somme des composantes")
42 ax2.legend(loc="upper left", fontsize="x-small")
43
44 # Sauvegarde de la figure (pas d'affichage interactif)
45 fig.savefig("figure.png")

```

Source : figure.py

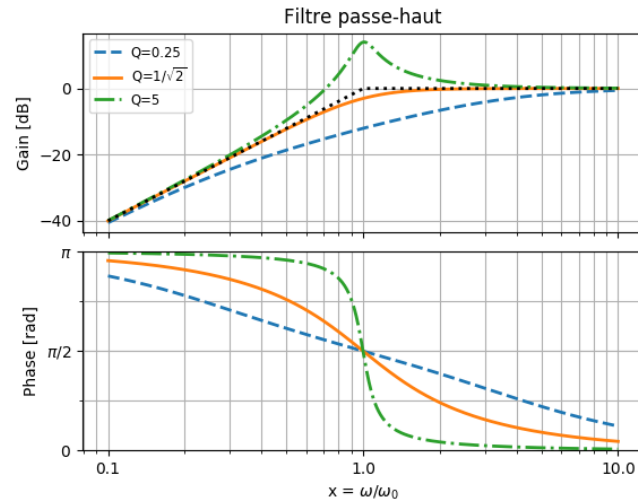
8.4.2 Filtrés du 2nd ordre

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import numpy as N
5  import matplotlib.pyplot as P

```

(suite sur la page suivante)

Fig. 8.2 – **Figure** : Filtre passe-haut du 2nd ordre.

(suite de la page précédente)

```

6
7
8 def passeBas(x, Q=1):
9     """
10    Filtre passe-bas en pulsation réduite *x* = omega/omega0, facteur de
11    qualité *Q*.
12    """
13
14    return 1 / (1 - x ** 2 + x / Q * 1j)
15
16
17 def passeHaut(x, Q=1):
18
19    return -x ** 2 / (1 - x ** 2 + x / Q * 1j)
20
21
22 def passeBande(x, Q=1):
23
24    return 1 / (1 + Q * (x - 1 / x) * 1j)
25
26
27 def coupeBande(x, Q=1):
28
29    return (1 - x ** 2) / (1 - x ** 2 + x / Q * 1j)
30
31
32 def gainNphase(f, dB=True):
33     """
34     Retourne le gain (éventuellement en dB) et la phase [rad] d'un
35     filtre de fonction de transfert complexe *f*.
36     """
37
38     g = N.abs(f)                # Gain
39     if dB:                      # [dB]
40         g = 20 * N.log10(g)
41     p = N.angle(f)             # [rad]
42
43     return g, p

```

(suite sur la page suivante)

(suite de la page précédente)

```

44
45
46 def asympGain(x, pentes=(0, -40)):
47
48     lx = N.log10(x)
49     return N.where(lx < 0, pentes[0] * lx, pentes[1] * lx)
50
51
52 def asympPhase(x, phases=(0, -N.pi)):
53
54     return N.where(x < 1, phases[0], phases[1])
55
56
57 def diagBode(x, filtros, labels,
58             title='', plim=None, gAsymp=None, pAsymp=None):
59     """
60     Trace le diagramme de Bode -- gain [dB] et phase [rad] -- des filtros
61     de fonction de transfert complexe *filtros* en fonction de la pulsation
62     réduite *x*.
63     """
64
65     fig = P.figure()
66     axg = fig.add_subplot(2, 1, 1,          # Axe des gains
67                          xscale='log',
68                          ylabel='Gain [dB]')
69     axp = fig.add_subplot(2, 1, 2,          # Axe des phases
70                          sharex=axg,
71                          xlabel=r' $x = \omega / \omega_0$ ', xscale='log',
72                          ylabel='Phase [rad]')
73
74     lstyles = ['--', '-', '-.', ':']
75     for f, label, ls in zip(filtros, labels, lstyles): # Tracé des courbes
76         g, p = gainNphase(f, dB=True)                # Calcul du gain et de la phase
77         axg.plot(x, g, lw=2, ls=ls, label="Q=" + str(label)) # Gain
78         axp.plot(x, p, lw=2, ls=ls)                    # Phase
79
80     # Asymptotes
81     if gAsymp is not None: # Gain
82         axg.plot(x, asympGain(x, gAsymp), 'k:', lw=2, label='_')
83     if pAsymp is not None: # Phase
84         # axp.plot(x, asympPhase(x, pAsymp), 'k:')
85         pass
86
87     axg.legend(loc='best', prop=dict(size='small'))
88
89     # Labels des phases
90     axp.set_yticks(N.arange(-2, 2.1) * N.pi / 2)
91     axp.set_yticks(N.arange(-4, 4.1) * N.pi / 4, minor=True)
92     axp.set_yticklabels([r' $-\pi$ ', r' $-\pi/2$ ', r' $0$ ', r' $\pi/2$ ', r' $\pi$ '])
93     # Domaine des phases
94     if plim is not None:
95         axp.set_ylim(plim)
96
97     # Ajouter les grilles
98     for ax in (axg, axp):
99         ax.grid() # x et y, majors
100        ax.grid(which='minor') # x et y, minors
101
102     # Ajustements fins
103     gmin, gmax = axg.get_ylim()
104     axg.set_ylim(gmin, max(gmax, 3))

```

(suite sur la page suivante)

(suite de la page précédente)

```

105
106 fig.subplots_adjust(hspace=0.1)
107 axg.xaxis.set_major_formatter(P.matplotlib.ticker.ScalarFormatter())
108 P.setp(axg.get_xticklabels(), visible=False)
109
110 if title:
111     axg.set_title(title)
112
113 return fig
114
115
116 if __name__ == '__main__':
117
118     x = N.logspace(-1, 1, 1000)           # de 0.1 à 10 en 1000 pas
119
120     # Facteurs de qualité
121     qs = [0.25, 1 / N.sqrt(2), 5]         # Valeurs numériques
122     labels = [0.25, r'$1/\sqrt{2}$', 5]    # Labels
123
124     # Calcul des fonctions de transfert complexes
125     pbs = [ passeBas(x, Q=q) for q in qs ]
126     phs = [ passeHaut(x, Q=q) for q in qs ]
127     pcs = [ passeBande(x, Q=q) for q in qs ]
128     cbs = [ coupeBande(x, Q=q) for q in qs ]
129
130     # Création des 4 diagrammes de Bode
131     figPB = diagBode(x, pbs, labels, title='Filtre passe-bas',
132                     plim=(-N.pi, 0),
133                     gAsymp=(0, -40), pAsymp=(0, -N.pi))
134     figPH = diagBode(x, phs, labels, title='Filtre passe-haut',
135                     plim=(0, N.pi),
136                     gAsymp=(40, 0), pAsymp=(N.pi, 0))
137     figPC = diagBode(x, pcs, labels, title='Filtre passe-bande',
138                     plim=(-N.pi / 2, N.pi / 2),
139                     gAsymp=(20, -20), pAsymp=(N.pi / 2, -N.pi / 2))
140     figCB = diagBode(x, cbs, labels, title='Filtre coupe-bande',
141                     plim=(-N.pi / 2, N.pi / 2),
142                     gAsymp=(0, 0), pAsymp=(0, 0))
143
144     P.show()

```

Source : `filtres2ndOrdre.py`

Note : Les exercices sont de difficulté variable, de * (simple) à *** (complexe).

9.1 Introduction

9.1.1 Intégration : méthode des rectangles *

La [méthode des rectangles](#) permet d'approximer numériquement l'intégrale d'une fonction f :

$$\int_a^b f(x) dx \approx h \sum_{i=0}^{n-1} f(x_i) \quad \text{avec} \quad h = (b-a)/n \quad \text{et} \quad x_i = a + (i+1/2)h.$$

On définit la fonction `sq` renvoyant le carré d'un nombre par (cf. [Fonctions](#)) :

```
def sq(x) :
    return x**2
```

Écrire un programme calculant l'intégrale de cette fonction entre $a=0$ et $b=1$, en utilisant une subdivision en $n=100$ pas dans un premier temps. Quelle est la précision de la méthode, et comment dépend-elle du nombre de pas ?

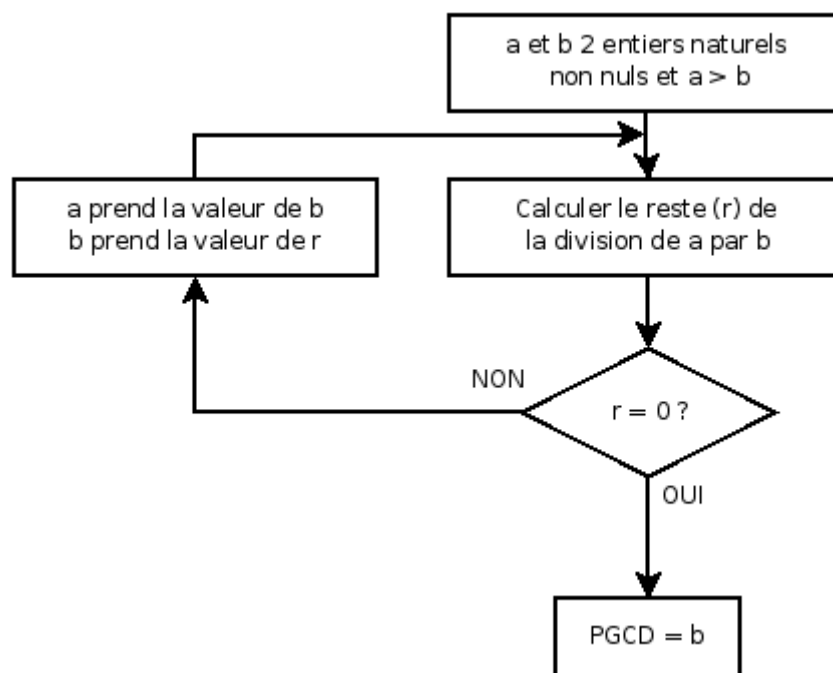
9.1.2 Fizz Buzz *

Écrire un programme jouant au [Fizz Buzz](#) jusqu'à 99 :

```
1 2 Fizz! 4 Buzz! Fizz! 7 8 Fizz! Buzz! 11 Fizz! 13 14 Fizz Buzz! 16...
```

9.1.3 PGCD : algorithme d'Euclide **

Écrire un programme calculant le PGCD (Plus Grand Commun Dénominateur) de deux nombres (p.ex. 306 et 756) par l'[algorithme d'Euclide](#).



9.1.4 Tables de multiplication *

Écrire un programme affichant les tables de multiplication :

```

1 x 1 = 1
1 x 2 = 2
...
9 x 9 = 81

```

9.2 Manipulation de listes

9.2.1 Crible d'Ératosthène *

Implémenter le [crible d'Ératosthène](#) pour afficher les nombres premiers compris entre 1 et un entier fixe, p.ex. :

```

Liste des entiers premiers <= 41
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]

```

9.2.2 Carré magique **

Un carré magique d'ordre n est un tableau carré $n \times n$ dans lequel on écrit une et une seule fois les nombres entiers de 1 à n^2 , de sorte que la somme des n nombres de chaque ligne, colonne ou diagonale principale soit constante. P.ex. le carré magique d'ordre 5, où toutes les sommes sont égales à 65 :

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Pour les carrés magiques d'ordre impair, on dispose de l'algorithme suivant – (i,j) désignant la case de la ligne i , colonne j du carré; on se place en outre dans une indexation « naturelle » commençant à 1 :

1. la case $(n, (n+1)/2)$ contient 1;
2. si la case (i,j) contient la valeur k , alors on place la valeur $k+1$ dans la case $(i+1, j+1)$ si cette case est vide, ou dans la case $(i-1, j)$ sinon. On respecte la règle selon laquelle un indice supérieur à n est ramené à 1.

Programmer cet algorithme pour pouvoir construire un carré magique d'ordre impair quelconque.

9.3 Programmation

9.3.1 Suite de Syracuse (fonction) *

Écrire une fonction `suite_syracuse(n)` retournant la (partie non-triviale de la) *suite de Syracuse* pour un entier n . Écrire une fonction `temps_syracuse(n, altitude=False)` retournant le temps de vol (éventuellement en altitude) correspondant à l'entier n . Tester ces fonctions sur $n=15$:

```
>>> suite_syracuse(15)
[15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
>>> temps_syracuse(15)
17
>>> temps_syracuse(15, altitude=True)
10
```

9.3.2 Flocon de Koch (programmation récursive) ***

En utilisant les commandes `left`, `right` et `forward` de la bibliothèque graphique standard `turtle` dans une fonction *récursive*, générer à l'écran un *flocon de Koch* d'ordre arbitraire.

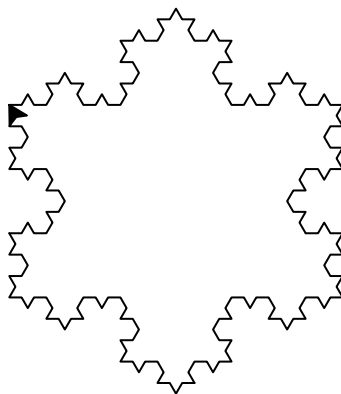


Fig. 9.1 – **Figure** : Flocon de Koch d'ordre 3.

9.3.3 Jeu du plus ou moins (exceptions) *

Écrire un jeu de « plus ou moins » :

```
Vous devez deviner un nombre entre 1 et 100.
Votre proposition: 27
C'est plus.
[...]
Vous avez trouvé en 6 coups!
```

La solution sera générée aléatoirement par la fonction `random.randint()`. Le programme devra être robuste aux entrées invalides (« toto », 120, etc.), et aux lâches abandons par interruption (`KeyboardInterrupt`).

9.3.4 Animaux (POO/TDD) *

Téléchargez `animaux.py` et complétez les classes `Animal` et `Chien` pour qu'elles passent avec succès tous les tests (voir *Développement piloté par les tests*). On appellera les tests via la ligne de commande :

```
py.test animaux.py
```

9.3.5 Jeu de la vie (POO) **

On se propose de programmer l'automate cellulaire le plus célèbre, le *Jeu de la vie*.

Pour cela, vous créerez une classe `Life` qui contiendra la grille du jeu ainsi que les méthodes qui permettront son évolution. Vous initialiserez la grille aléatoirement à l'aide de la fonction `random.choice()`, et vous afficherez l'évolution de l'automate dans la sortie standard du terminal, p.ex. :

```
...#.#.....##.....
.....###.....
#.....#.....
.....#.#.....
.....###.....
.....#.#.....##.##.
.....#.#.....##.##.
.....###.##.
.....###.##.
.....#.....
```

Astuce : Pour que l'affichage soit agréable à l'oeil, vous marquerez des pauses entre l'affichage de chaque itération grâce à la fonction `time.sleep()`.

9.4 Manipulation de tableaux (arrays)

9.4.1 Inversion de matrice *

Créer un tableau carré réel `r` aléatoire (`numpy.random.randn()`), calculer la matrice hermitienne $\mathbf{m} = \mathbf{r} \cdot \mathbf{r}^T$ (`numpy.dot()`), l'inverser (`numpy.linalg.inv()`), et vérifier que $\mathbf{m} \cdot \mathbf{m}^{-1} = \mathbf{m}^{-1} \cdot \mathbf{m} = \mathbf{1}$ (`numpy.eye()`) à la précision numérique près (`numpy.allclose()`).

9.4.2 Median Absolute Deviation *

En statistique, le *Median Absolute Deviation* (MAD) est un estimateur robuste de la dispersion d'un échantillon 1D : `MAD = median(| x - median(x) |)`.

À l'aide des fonctions `numpy.median()` et `numpy.abs()`, écrire une fonction `mad(x, axis=None)` calculant le MAD d'un tableau, éventuellement le long d'un ou plusieurs de ses axes.

9.4.3 Distribution du *pull* ***

Le *pull* est une quantité statistique permettant d'évaluer la conformité des erreurs par rapport à une distribution de valeurs (typiquement les résidus d'un ajustement). Pour un échantillon $\mathbf{x} = [x_i]$ et les erreurs associées $d\mathbf{x} = [\sigma_i]$, le *pull* est défini par :

- moyenne optimale (pondérée par la variance) : $E = (\sum_i x_i/\sigma_i^2)/(\sum_i 1/\sigma_i^2)$;
- erreur sur la moyenne pondérée : $\sigma_E^2 = 1/\sum_i (1/\sigma_i^2)$;
- définition du *pull* : $p_i = (x_i - E_i)/(\sigma_{E_i}^2 + \sigma_i^2)^{1/2}$, où E_i et σ_{E_i} sont calculées *sans* le point i .

Si les erreurs σ_i sont correctes, la distribution du *pull* est centrée sur 0 avec une déviation standard de 1.

Écrire une fonction `pull(x, dx)` calculant le *pull* de tableaux 1D.

9.5 Méthodes numériques

9.5.1 Quadrature et zéro d'une fonction *

À l'aide des algorithmes disponibles dans `scipy` :

- calculer numériquement l'intégrale $\int_0^\infty \frac{x^3}{e^x-1} dx = \pi^4/15$;
- résoudre numériquement l'équation $x e^x = 5(e^x - 1)$.

9.5.2 Schéma de Romberg **

Écrire une fonction `integ_romberg(f, a, b, epsilon=1e-6)` permettant de calculer l'intégrale numérique de la fonction f entre les bornes a et b avec une précision *epsilon* selon la [méthode de Romberg](#).

Tester sur des solutions analytiques et en comparant à `scipy.integrate.romberg()`.

9.5.3 Méthode de Runge-Kutta **

Développer un algorithme permettant d'intégrer numériquement une équation différentielle du 1er ordre en utilisant la [méthode de Runge-Kutta](#) d'ordre quatre.

Tester sur des solutions analytiques et en comparant à `scipy.integrate.odeint()`.

9.6 Visualisation (matplotlib)

9.6.1 Quartet d'Anscombe *

Après chargement des données, calculer et afficher les propriétés statistiques des quatres jeux de données du *Quartet d'Anscombe* :

- moyenne et variance des x et des y (`numpy.mean()` et `numpy.var()`) ;
- corrélation entre les x et les y (`scipy.stats.pearsonr()`) ;
- équation de la droite de régression linéaire $y = ax + b$ (`scipy.stats.linregress()`).

Tableau 9.1 – Quartet d'Anscombe

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Pour chacun des jeux de données, tracer y en fonction de x , ainsi que la droite de régression linéaire.

9.6.2 Diagramme de bifurcation : la suite logistique **

Écrivez une fonction qui calcule la valeur d'équilibre de la [suite logistique](#) pour un x_0 (nécessairement compris entre 0 et 1) et un paramètre r (parfois noté μ) donné.

Générez l'ensemble de ces points d'équilibre pour des valeurs de r comprises entre 0 et 4 :

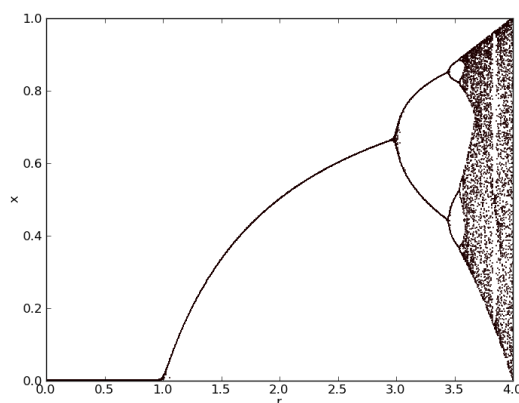


Fig. 9.2 – **Figure** : Diagramme de bifurcation.

N.B. Vous utiliserez la bibliothèque [Matplotlib](#) pour tracer vos résultats.

9.6.3 Ensemble de Julia **

Représentez l'ensemble de Julia pour la constante complexe $c = 0.284 + 0.0122j$:

On utilisera la fonction `numpy.meshgrid()` pour construire le plan complexe, et l'on affichera le résultat grâce à la fonction `matplotlib.pyplot.imshow()`.

Voir également : [Superposition d'ensembles de Julia](#)

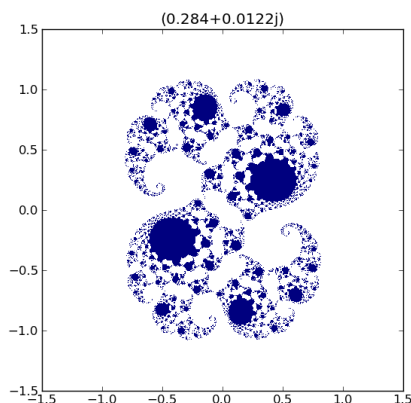


Fig. 9.3 – **Figure** : Ensemble de Julia pour $c = 0.284 + 0.0122j$.

9.7 Mise en oeuvre de l'ensemble des connaissances acquises

9.7.1 Équation différentielle *

À l'aide de la fonction `scipy.integrate.odeint()`, intégrer les équations du mouvement d'un boulet de canon soumis à des forces de frottement « turbulentes » (en v^2) :

$$\ddot{\mathbf{r}} = \mathbf{g} - \frac{\alpha}{m} v \times \mathbf{v}.$$

Utiliser les valeurs numériques pour un boulet de canon de 36 livres :

```
g = 9.81          # Pesanteur [m/s2]
cx = 0.45         # Coefficient de frottement d'une sphère
rhoAir = 1.2      # Masse volumique de l'air [kg/m3]
rad = 0.1748/2    # Rayon du boulet [m]
rho = 6.23e3      # Masse volumique du boulet [kg/m3]
mass = 4./3.*N.pi*rad**3 * rho          # Masse du boulet [kg]
alpha = 0.5*cx*rhoAir*N.pi*rad**2 / mass # Coeff. de frottement / masse
v0 = 450.         # Vitesse initiale [m/s]
alt = 45.         # Inclinaison du canon [deg]
```

Voir également : Équations de prédation de Lotka-Volterra

9.7.2 Équation d'état de l'eau à partir de la dynamique moléculaire ***

Afin de modéliser les planètes de type Jupiter, Saturne, ou même des exo-planètes très massives (dites « super-Jupiters »), la connaissance de l'équation d'état des composants est nécessaire. Ces équations d'état doivent être valables jusqu'à plusieurs centaines de méga-bar ; autrement dit, celles-ci ne sont en aucun cas accessibles expérimentalement. On peut cependant obtenir une équation d'état numériquement à partir d'une dynamique moléculaire.

Le principe est le suivant : on place dans une boîte un certain nombre de particules régies par les équations microscopiques (Newton par exemple, ou même par des équations prenant en considération la mécanique quantique) puis on laisse celles-ci évoluer dans la boîte ; on calcule à chaque pas de temps l'énergie interne à partir des interactions électrostatiques et la pression à partir du tenseur des contraintes. On obtient en sortie l'évolution du système pour une densité fixée (par le choix de taille de la boîte) et une température fixée (par un algorithme de thermostat que nous ne détaillerons pas ici).

On se propose d'analyser quelques fichiers de sortie de tels calculs pour l'équation d'état de l'eau à très haute pression. Les fichiers de sortie sont disponibles [ici](#) ; leur nom indique les conditions thermodynamiques correspondant au fichier, p.ex. 6000K_30gcc.out pour $T = 6000$ K et $\rho = 30$ gcc. Le but est,

pour chaque condition température-densité, d'extraire l'évolution de l'énergie et de la pression au cours du temps, puis d'en extraire la valeur moyenne ainsi que les fluctuations. Il arrive souvent que l'état initial choisi pour le système ne corresponde pas à son état d'équilibre, et qu'il faille donc « jeter » les quelques pas de temps en début de simulation qui correspondent à cette relaxation du système. Pour savoir combien de temps prend cette relaxation, il sera utile de tracer l'évolution au cours du temps de la pression et l'énergie pour quelques simulations. Une fois l'équation d'état $P(\rho, T)$ et $E(\rho, T)$ extraite, on pourra tracer le réseau d'isothermes.

Indication : Vous écrirez une classe `Simulation` qui permet de charger un fichier de dynamique moléculaire, puis de tracer l'évolution de la température et de la densité, et enfin d'en extraire la valeur moyenne et les fluctuations. À partir de cette classe, vous construirez les tableaux contenant l'équation d'état.

9.8 Exercices en vrac

- Exercices de base 
- Entraînez-vous ! 
- Learn Python The Hard Way
- Google Code Jam
- CheckIO

9.8.1 Points matériels et ions (POO/TDD)

Pour une simulation d'un problème physique, on peut construire des classes qui connaissent elles-mêmes leurs propriétés physiques et leurs lois d'évolution.

La structure des classes est proposée dans ce `squelette`. Vous devrez *compléter* les définitions des classes `Vector`, `Particle` et `Ion` afin qu'elles passent toutes les tests lancés automatiquement par le programme principal `main`. À l'exécution, la sortie du terminal doit être :

```
***** Test functions *****
Testing Vector class... ok
Testing Particle class... ok
Testing Ion class... ok
***** Test end *****

***** Physical computations *****
** Gravitational computation of central-force motion for a Particle with mass 1.00, position
↪ (1.00,0.00,0.00) and speed (0.00,1.00,0.00)
=> Final system : Particle with mass 1.00, position (-1.00,-0.00,0.00) and speed (0.00,-1.00,0.
↪ 00)
** Electrostatic computation of central-force motion for a Ion with mass 1.00, charge 4,
↪ position (0.00,0.00,1.00) and speed (0.00,0.00,-1.00)
=> Final system : Ion with mass 1.00, charge 4, position (0.00,0.00,7.69) and speed (0.00,0.00,
↪ 2.82)
***** Physical computations end *****
```

9.8.2 Protein Data Bank

On cherche à réaliser un script qui analyse un fichier de données de type [Protein Data Bank](#).

La banque de données [Worldwide Protein Data Bank](#) regroupe les structures obtenues par diffraction aux rayons X ou par RMN. Le format est parfaitement défini et conventionnel ([documentation](#)).

On propose d'assurer une lecture de ce fichier pour calculer notamment :

- le barycentre de la biomolécule
- le nombre d'acides aminés ou nucléobases
- le nombre d'atomes
- la masse moléculaire
- les dimensions maximales de la protéine
- etc.

On propose de considérer par exemple la structure résolue pour la GFP (*Green Fluorescent Protein*, Prix Nobel 2008) ([Fichier PDB](#))

10.1 Simulation de chute libre (partiel nov. 2014)

- Énoncé (PDF) et fichier d'entrée
- Corrigé

10.2 Examen janvier 2015

- Énoncé (PDF) ou *Examen final, Janvier 2015*.
 - Exercice : `velocimetrie.dat`
 - Problème : `exam_1501.py`, `ville.dat`
- Corrigé, figure

Table des matières

- *Projets*
 - *Projets de physique*
 - *Formation d'agrégats*
 - *Modèle d'Ising*
 - *Modèle de Potts 3D*
 - *Méthode de Hückel*
 - *Densité d'états d'un nanotube*
 - *Solitons*
 - *Diagramme de phase du potentiel de Lennard-Jones*
 - *États de diffusion pour l'équation de Schrödinger 1D stationnaire*
 - *Percolation*
 - *Autres possibilités*
 - *Projets astrophysiques*
 - *Relation masse/rayon d'une naine blanche*
 - *Section de Poincaré*
 - *Projets divers*
 - *Formation de pistes de fourmis sur un pont à 2 branches*
 - *Auto-organisation d'un banc de poisson*
 - *Évacuation d'une salle & déplacement d'une foule dans une rue*
 - *Suivi de particule(s)*
 - *Projets statistiques*
 - *Projets de visualisation*

11.1 Projets de physique

11.1.1 Formation d'agrégats

La formation d'agrégats est par essence un sujet interdisciplinaire, où la modélisation joue un rôle certain comme « microscope computationnel ». Pour un projet en ce sens, un soin particulier sera donné à la

contextualisation. P.ex., on pourra tester les limites de la règle de Wade pour la structure de clusters métalliques, ou bien dans un contexte plus biologique.

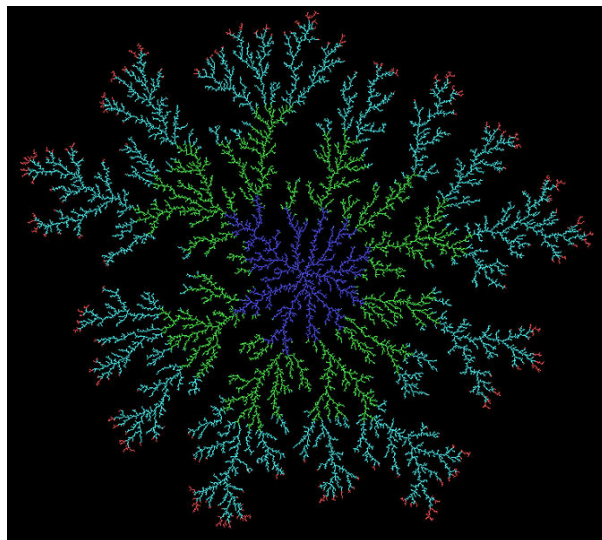


Fig. 11.1 – **Figure** : Résultat d’une agrégation limitée par la diffusion d’environ 33 000 particules obtenue en permettant à des marcheurs aléatoires d’adhérer à une semence centrale. Les couleurs indiquent le temps d’arrivée des marcheurs. Source : [WingkLEE](#) (Own work) [Public domain], via Wikimedia Commons.

11.1.2 Modèle d’Ising

Le modèle d’Ising est le modèle le plus simple du magnétisme. Le modèle 1D est exactement soluble par la méthode de la matrice de transfert. La généralisation à 2 dimensions a été faite par Lars Onsager en 1944, mais la solution est assez compliquée. Il n’existe pas de solution analytique en 3D. On va ici considérer un système de spins sur réseau. Chaque spin σ_i peut prendre 2 valeurs (« up » et « down »). L’hamiltonien du système,

$$H = -J \sum_{i,j} \sigma_i \sigma_j - h \sum_i \sigma_i$$

contient deux contributions : l’interaction entre premiers voisins et le couplage à un champ magnétique. On va considérer un réseau carré avec une interaction ferromagnétique ($J > 0$). L’objectif du projet sera d’étudier le diagramme de phase du système en fonction de la température et du champ magnétique par simulation de Monte-Carlo.

11.1.3 Modèle de Potts 3D

Modèle de Potts en 3D dans un univers carré à condition périodique. Le but est la mise en évidence de la transition de phase pour plusieurs jeux de paramètres avec 3 types de spins différents.

1. Reproduire des résultats connus du modèle d’Ising en 2D pour valider le code.
2. Passer à un algorithme en *cluster* pour évaluer la différence avec un algorithme classique.
3. Passer en 3D
4. Changer le nombre de type de spins (de 2 à 3).

Jeux de paramètres à tester :

- **Ising** en 2D (2 types de spins, algorithme de Glauber) : Transition de phase attendue à $T \sim 227K$ pour un couplage $J=100$ et un champ externe nul
- Toujours Ising, mais avec l’algorithme de Wolff
- Ising en 3D avec Wolff
- Potts (changer $q=2$ par $q=3$) en 3D avec Wolff

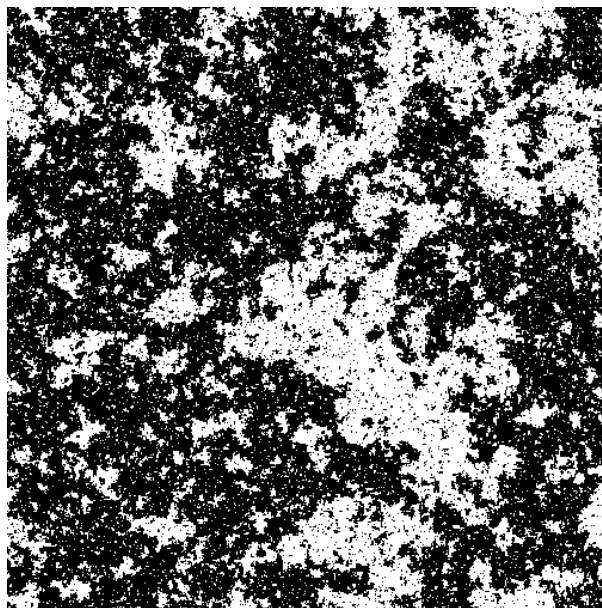


Fig. 11.2 – **Figure** : Modèle d'Ising au point critique. Source : [Paul Coddington](#).

Références :

[Computational Studies of Pure and Dilute Spin Models](#)

11.1.4 Méthode de Hückel

La spectroscopie et la réactivité des électrons π est centrale en chimie. Un outil efficace pour les appréhender est l'approche développée par Hückel. Il vous est demandé ici de mettre en œuvre cette méthode pour l'analyse des orbitales et de l'énergie d'une famille de molécules répondant aux hypothèses sous-jacentes. On discutera notamment du choix de la paramétrisation du système.

11.1.5 Densité d'états d'un nanotube

Les nanotubes de carbone ont été découverts bien avant celle du graphène. Ce sont des matériaux très résistants et durs qui possèdent une conductivité électrique et thermique élevées. Un nanotube de carbone monofeuillet consiste d'une couche de graphène enroulé selon un certain axe. L'axe d'enroulement détermine la chiralité du nanotube et, par la suite, les propriétés électroniques : selon la chiralité, le nanotube peut être soit semi-conducteur, soit métallique. L'objectif du projet sera de calculer la densité d'états de nanotubes de carbone de différentes chiralités et d'établir le lien entre la chiralité et le fait que le nanotube soit semiconducteur ou métallique.

11.1.6 Solitons

On considère un câble sous tension auquel sont rigidement et régulièrement attachés des pendules. Les pendules sont couplés grâce au câble à travers sa constante de torsion. Dans un tel système on peut observer une large gamme de phénomènes ondulatoires. Le but de cet projet est d'étudier une solution très particulière : le *soliton*.

Imaginons qu'une des extrémités du câble est attachée à une manivelle qui peut tourner librement. Il est alors possible de donner une impulsion au système en faisant un tour rapide ce qui déclenche la propagation d'un soliton. Dans ce projet, on considérera les pendules individuellement. Il n'est pas demandé de passer au modèle continu et de résoudre l'équation obtenue.

Pour chaque pendule n dont la position est décrite par θ_n , l'équation d'évolution s'écrit :

$$\frac{d^2\theta_n}{dt^2} = \alpha \sin \theta_n + \beta(\theta_{n-1} + \theta_{n+1} - 2\theta_n)$$

où α, β sont des paramètres physiques. On résoudra numériquement cette équation pour chaque pendule. En donnant un « tour de manivelle numérique », on essaiera d'obtenir la solution soliton. On cherchera en particulier à ajuster la solution par une équation du type $\theta_n = a \tan^{-1}(\exp(b(n - n_0)))$ où a, b, n_0 sont des paramètres à déterminer.

De très nombreuses questions se posent (il ne vous est pas demandé de répondre à chacune d'entre elle) :

- Est-il toujours possible d'obtenir un soliton ?
- Sa vitesse est-elle constante ?
- Le soliton conserve-t-il sa forme ?
- Que se passe-t-il avec des pendules plus lourdes ? ou plus rapprochés ? avec un câble plus rigide ? avec un frottement ?
- Comment le soliton se réfléchit-il si l'extrémité du câble est rigidement fixée ? et si elle tourne librement ?
- Dans ce système, le soliton est chiral. En effet, on peut tourner la manivelle à gauche ou à droite. Un anti-soliton a-t-il les mêmes propriétés (taille, vitesse, énergie) qu'un soliton ?
- Si on place une manivelle à chaque extrémité, on peut faire se collisionner des solitons. Cette étude est très intéressante et pleine de surprises. Que se passe-t-il lors de la collision de deux solitons ? Entre un soliton et un anti-soliton ?



Fig. 11.3 – **Figure** : Un mascaret, une vague soliton, dans un estuaire de Grande Bretagne. *Source* : Arnold Price [CC-BY-SA-2.0], via Wikimedia Commons.

11.1.7 Diagramme de phase du potentiel de Lennard-Jones

Auteur de la section : Mathieu Leocmach <mathieu.leocmach@ens-lyon.fr>

Le potentiel de Lennard-Jones est souvent utilisé pour décrire les interactions entre deux atomes au sein d'un système monoatomique de type gaz rare. Son expression en fonction de la distance r entre les deux noyaux atomiques est :

$$E_p(r) = 4E_0 \left[\left(\frac{r_0}{r} \right)^{12} - \left(\frac{r_0}{r} \right)^6 \right]$$

avec r_0 la distance pour laquelle $E_p(r_0) = 0$.

On programmera un simulateur de dynamique moléculaire pour N particules identiques dans un cube périodique de taille fixe L et à une température T . On prendra soin d'alimenter toutes les grandeurs et d'imposer des conditions aux limites périodiques. On se renseignera sur les façons possibles de déterminer les conditions initiales et d'imposer la température.

Les positions et vitesses des particules seront exportées de façon régulières pour visualisation (par exemple dans [Paraview](#)).

- On pourra observer les collisions de 2 ou 3 particules à différentes températures avant de passer à des N plus grands (100 particules?).
- On fera varier $V = L^3$ et T pour déterminer les frontières des différentes phases.
- On pourra aussi essayer d'aller vers de plus grands N pour tester l'influence de la taille finie de l'échantillon. Des optimisations seront alors sûrement nécessaires pour accélérer le programme.
- On pourra aussi tester d'autres types de potentiels comme celui de Weeks-Chandler-Anderson et discuter des différences observées.

11.1.8 États de diffusion pour l'équation de Schrödinger 1D stationnaire

On s'intéresse à la diffusion d'une particule de masse m à travers un potentiel carré défini par $V(x) = V_0$ pour $0 \leq x \leq a$, et 0 sinon.

Les solutions de cette équation en dehors de la région où règne le potentiel sont connues. Les paramètres d'intégration de ces fonctions d'onde peuvent se déterminer par les relations de continuité aux frontières avec la région où règne le potentiel. En résolvant l'équation différentielle dans la région du potentiel pour x allant de a à 0 on peut obtenir une autre valeur pour ces paramètres d'intégration. Il faut ensuite appliquer un algorithme de minimisation pour déterminer les constantes d'intégration.

Les objectifs de ce projet sont :

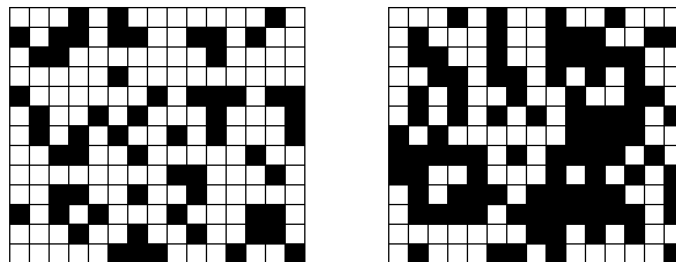
- Écrire un programme qui résolve l'équation de Schrödinger.
- En déduire les coefficients de transmission et de réflexion.

Références :

- *A numerical method for quantum tunnelling*, Pang T., Computers in Physics, 9, p 602-605.
- [Équation de Schrödinger 1D](#)
- [Quantum Python: Animating the Schrodinger Equation](#)

11.1.9 Percolation

Ce sujet propose d'étudier le phénomène de percolation. La percolation est un processus physique qui décrit pour un système, une transition d'un état vers un autre. Le système que nous étudierons est composé ici d'une grille carrée dont les cases sont soit vides, soit pleines. Initialement, la matrice est vide et l'on tire aléatoirement une case que l'on remplit. On définit la concentration comme le rapport du nombre de cases noires sur le nombre total de cases. À partir d'une certaine concentration critique un chemin continu de cases noires s'établit entre deux bords opposés du système (haut et bas, ou gauche et droite) et on dit alors que le système percole. Le but du sujet est d'étudier la transition d'un système qui ne percole pas (à gauche sur la figure) vers un système qui percole (à droite). Pour ce faire, on établira un algorithme qui pour une configuration donnée détermine si le réseau de cases noires percole ou non. On étudiera également la taille et le nombre des amas de cases noires en fonction de la concentration. On étudiera aussi les effets de la taille du système.



Cette étude repose sur un tirage pseudo aléatoire et pas conséquent nécessite un traitement statistique. On ne pourra pas se contenter d'étudier un cas particulier mais on prendra soin au contraire d'effectuer des moyennes sur un très grand nombre de tirages (plusieurs centaines).

Références :

- Percolation theory
- Concepts fondamentaux de la percolation
- Percolation exercises : 2006, 2012

11.1.10 Autres possibilités

- Reaction-Diffusion by the Gray-Scott Model
- Équation de Cahn–Hilliard (voir l'exemple NIST sous FiPy: A Finite Volume PDE Solver Using Python)
- Computational Methods for Nonlinear Systems

11.2 Projets astrophysiques

Auteur de la section : Méthodes numériques pour la physique et les SPI

11.2.1 Relation masse/rayon d'une naine blanche

D'après la théorie de l'évolution stellaire, les naines blanches sont l'un des états possibles d'une étoile (peu massive) à la fin de sa vie, lorsque les réactions de fusion thermonucléaire s'arrêtent.

En première approximation un corps astrophysique est essentiellement soumis à la force de gravitation (qui tend à le contracter) et une force interne de pression qui vient équilibrer la première. Ainsi on peut approcher le problème par un équilibre hydrostatique caractérisé par :

$$\nabla P(r) = -\rho(r) \frac{GM(r)}{r^2} \mathbf{e}_r$$

où G est la constante de gravitation, $P(r)$, $\rho(r)$ et $M(r)$ respectivement la pression, la densité à la distance r du centre et la masse dans une sphère de rayon r .

Il s'agit d'étudier ici quelle force peut équilibrer la gravitation pour une naine blanche et mettre en évidence une masse limite en étudiant la relation rayon/masse.

Modélisation

La masse et le rayon d'équilibre de ce système sont entièrement déterminés par l'équation d'état thermodynamique $P = P(\rho)$ et la densité centrale. En effet on montre facilement que :

$$\begin{aligned} \frac{d\rho}{dr} &= -\left(\frac{dP}{d\rho}\right)^{-1} \frac{GM}{r^2} \rho \\ \frac{dM}{dr} &= 4\pi r^2 \rho \end{aligned}$$

Une fois que les réactions thermonucléaires s'arrêtent, la première des forces empêchant l'étoile de s'effondrer vient de la pression due aux électrons. Le modèle que nous utiliserons sera donc un simple gaz d'électrons (masse m_e et de nombre par unité de volume n) plongé dans un gaz de noyaux (on note Y_e

le nombre d'électrons par nucléon et M_n la masse d'un nucléon) d'équation d'état :

$$\begin{aligned}\frac{E}{V} &= n_0 m_e c^2 x^3 \varepsilon(x), \\ \text{avec } x &= \left(\frac{\rho}{\rho_0} \right)^{\frac{1}{3}}, \\ n_0 &= \frac{m_e^3 c^3}{3 \hbar^3 \pi^2}, \\ \rho_0 &= \frac{M_n n_0}{Y_e} \\ \text{et } \varepsilon(x) &= \frac{3}{8x^3} \left[x(1+2x^2) \sqrt{1+x^2} - \ln \left(x + \sqrt{1+x^2} \right) \right]\end{aligned}$$

Si tous les noyaux sont du ^{12}C , alors $Y_e = 1/2$.

1. Montrer que le système d'équations à résoudre est

$$\begin{aligned}\frac{d\rho}{dr} &= - \left(\frac{3M_n G}{Y_e m_e c^2} \frac{\sqrt{1+x^2}}{x^2} \right) \frac{M}{r^2} \rho \\ \frac{dM}{dr} &= 4\pi r^2 \rho\end{aligned}$$

2. En fixant la densité centrale $\rho(r=0) = \rho_c$ tracer $\rho(r)$ et en déduire une méthode pour calculer le rayon R de l'étoile et sa masse M .
3. En faisant varier la densité centrale tracer la relation $M(R)$.
4. Discuter la validité numérique et physique des résultats par exemple en changeant la composition de l'étoile, la définition du rayon de l'étoile, etc.

11.2.2 Section de Poincaré

Les équations du mouvement ¹ $\mathbf{r}(t) = (x(t), y(t))$ d'une particule de masse m plongée dans un potentiel $\Phi(x, y)$ s'écrivent :

$$m\ddot{\mathbf{r}} = -\nabla\Phi.$$

En coordonnées polaires :

$$\begin{aligned}a_r &= \ddot{r} - r\dot{\theta}^2 = -\frac{1}{m} \frac{\partial\Phi}{\partial r} \\ a_\theta &= 2\dot{r}\dot{\theta} + r\ddot{\theta} = -\frac{1}{mr} \frac{\partial\Phi}{\partial\theta}\end{aligned}$$

Le système peut donc s'écrire :

$$\begin{aligned}\ddot{r} &= r\dot{\theta}^2 - \frac{1}{m} \frac{\partial\Phi}{\partial r} \\ \ddot{\theta} &= -\frac{2}{r} \dot{r}\dot{\theta} - \frac{1}{mr^2} \frac{\partial\Phi}{\partial\theta}\end{aligned}$$

ou en posant $r_p = \dot{r}$ et $\theta_p = \dot{\theta}$:

$$\begin{aligned}\dot{r} &= r_p \\ \dot{\theta} &= \theta_p \\ \dot{r}_p &= r\theta_p^2 - \frac{1}{m} \frac{\partial\Phi}{\partial r} \\ \dot{\theta}_p &= -\frac{2}{r} r_p \theta_p - \frac{1}{mr^2} \frac{\partial\Phi}{\partial\theta}\end{aligned}$$

¹ On se place dans toute la suite du problème dans un espace à deux dimensions.

L'intégration – analytique ou numérique – de ces équations pour des conditions initiales ($\mathbf{r}(t=0), \dot{\mathbf{r}}(t=0)$) particulières caractérise une *orbite*. Le tracé de l'ensemble des points d'intersection de différentes orbites de même énergie avec le plan, p.ex., (x, \dot{x}) (avec $y = 0$ et $\dot{y} > 0$) constitue une *section de Poincaré*.

Nous étudierons plus particulièrement le cas particulier $m = 1$ et les deux potentiels :

1. le potentiel intégrable de Sridhar & Touma (1997; MNRAS, **287**, L1)², qui s'exprime naturellement dans les coordonnées polaires (r, θ) :

$$\Phi(r, \theta) = r^\alpha \left[(1 + \cos \theta)^{1+\alpha} + (1 - \cos \theta)^{1+\alpha} \right].$$

avec p.ex. $\alpha = 1/2$;

2. le potentiel de Hénon-Heiles :

$$\Phi(r, \theta) = \frac{1}{2}r^2 + \frac{1}{3}r^3 \sin(3\theta).$$

Objectif

1. Écrire un intégrateur numérique permettant de résoudre les équations du mouvement pour un potentiel et des conditions initiales données.
2. Les performances de cet intégrateur seront testées sur des potentiels intégrables (p.ex. potentiel képlerien $\Phi \propto 1/r$), ou en vérifiant la stabilité des constantes du mouvement (l'énergie $E = \frac{1}{2}\dot{\mathbf{r}}^2 + \Phi$).
3. Pour chacun des potentiels, intégrer et stocker une grande variété d'orbites de même énergie, en prenant soin de bien résoudre la zone d'intérêt autour de $(y = 0, \dot{y} > 0)$.
4. À l'aide de fonctions d'interpolation et de recherche de zéro, déterminer pour chacune des orbites les coordonnées (x, \dot{x}) de l'intersection avec le plan $(y = 0, \dot{y} > 0)$.
5. Pour chacun des potentiels, regrouper ces points par orbite pour construire la section de Poincaré de ce potentiel.

11.3 Projets divers

11.3.1 Formation de pistes de fourmis sur un pont à 2 branches

Si on propose à une colonie de fourmis de choisir entre 2 branches pour rejoindre une source de nourriture la branche finalement choisie est toujours la plus courte. Le projet consiste à modéliser et caractériser ce comportement.

Indication : on peut étudier ce système avec des EDOs. Cela peut aussi donner lieu à une simulation individu centré et éventuellement une comparaison entre les deux types de modèle.

11.3.2 Auto-organisation d'un banc de poisson

Auteur de la section : Hanna Julienne <hanna.julienne@gmail.com>

La coordination d'un **banc de poissons** ou d'un **vol d'oiseaux** est tout à fait frappante : les milliers d'individus qui composent ces structures se meuvent comme un seul. On observe aussi, dans les bancs de poisson, d'impressionnants comportements d'évitement des prédateurs (**flash expansion**, **fountain effect**).

Pourtant ces mouvements harmonieusement coordonnés ne peuvent pas s'expliquer par l'existence d'un poisson leader. Comment pourrait-il être visible par tous ou diriger les *flash expansion* qui ont lieu à un endroit précis du banc de poisson ? De la même manière on ne voit pas quelle contrainte extérieure pourrait expliquer le phénomène.

² Nous utiliserons toutefois les notations de l'appendice de Copin, Zhao & de Zeeuw (2000; MNRAS, **318**, 781).

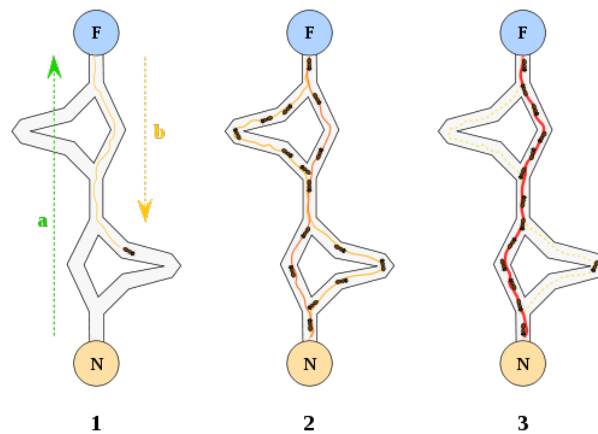


Fig. 11.4 – **Figure :** 1) la première fourmi trouve la source de nourriture (F), via un chemin quelconque (a), puis revient au nid (N) en laissant derrière elle une piste de phéromone (b). 2) les fourmis empruntent indifféremment les 4 chemins possibles, mais le renforcement de la piste rend plus attractif le chemin le plus court. 3) les fourmis empruntent le chemin le plus court, les portions longues des autres chemins voient la piste de phéromones s'évaporer. Source : [Johann Dréo](#) via Wikimedia Commons.

Une hypothèse plus vraisemblable pour rendre compte de ces phénomènes est que la cohérence de l'ensemble est due à la somme de comportements individuels. Chaque individu adapte son comportement par rapport à son environnement proche. C'est ce qu'on appelle *auto-organisation*. En effet, on a établi expérimentalement que les poissons se positionnent par rapport à leurs k plus proches voisins de la manière suivante :

- ils s'éloignent de leurs voisins très proches (zone de répulsion en rouge sur la figure ci-dessous)
- ils s'alignent avec des voisins qui sont à distance modérée (zone jaune)
- ils s'approchent de leur voisins s'ils sont à la fois suffisamment proches et distants (zone verte)

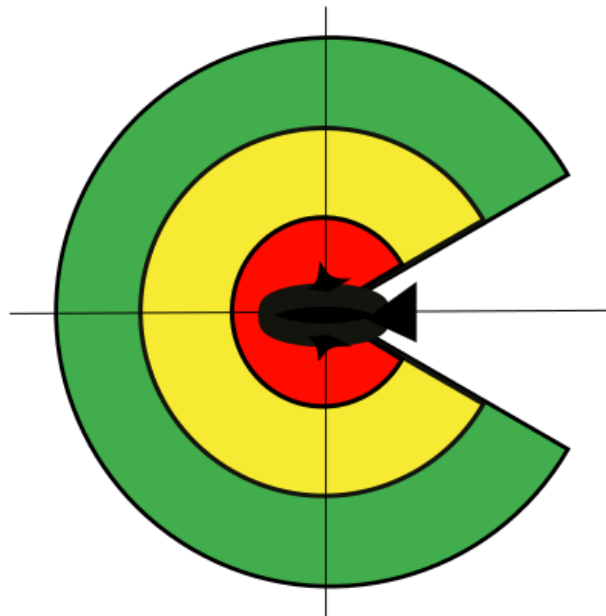


Fig. 11.5 – **Figure :** Environnement proche du poisson : zones dans lesquelles le positionnement d'un voisin provoque une réponse de la part de l'individu au centre

Dans notre modèle, nous allons prendre en compte l'influence des k plus proches voisins. On calculera la contribution de chaque voisin selon la zone dans laquelle il se situe. Le déplacement du poisson sera la moyenne de ces contributions. Il est à noter qu'un voisin en dehors des trois zones d'influence n'a pas d'effet.

L'environnement proche d'un poisson est modélisé par des sphères imbriquées qui présentent une zone aveugle (voir figure).

Par ailleurs, si un individu n'a pas de voisins dans son environnement proche il adopte un comportement de recherche. Il explore aléatoirement les alentours jusqu'à ce qu'il repère le banc de poissons et finalement s'en rapproche.

Ce projet vise à :

- Coder le comportement des poissons et à les faire évoluer dans un environnement **2D**.
- On essaiera d'obtenir un comportement collectif cohérent (similaire à un banc de poisson) et d'établir les conditions nécessaires à ce comportement.
- On étudiera notamment l'influence du nombre d'individus pris en compte. Est-ce que le positionnement par rapport au plus proche voisin ($k = 1$) est suffisant ?
- On pourra se servir de la visualisation pour rendre compte de la cohérence du comportement et éventuellement inventer des mesures pour rendre compte de manière quantifier de cette cohérence.

Liens :

- [Craig Reynolds Boids](#)
- [Décrypter les interactions entre poissons au sein d'un banc](#)

11.3.3 Évacuation d'une salle & déplacement d'une foule dans une rue

Le comportement d'une foule est un problème aux applications multiples : évacuation d'une salle, couloir du métro aux heures de pointes, manifestations... On peut en imaginer des modèles simples. P. ex., on peut décrire chaque individu par sa position, sa vitesse, et comme étant soumis à des « forces » :

- Une force qui spécifie la direction dans laquelle l'individu *veut* se déplacer, $\mathbf{f}_{dir} = (\mathbf{v}_0 - \mathbf{v}(t))/\tau$, où \mathbf{v}_0 est la direction et la vitesse que la personne veut atteindre, \mathbf{v} sa vitesse actuelle, et τ un temps caractéristique d'ajustement.
- Une force qui l'oblige à éviter des obstacles qui peuvent être fixes (un mur, un massif de fleurs, ...), ou qui peuvent être les autres individus eux-mêmes. On pourra essayer $f_{obs}(d) = a \exp(-d/d_0)$, où d est la distance entre le piéton et l'obstacle, d_0 la « portée » de la force, et a son amplitude.

On pourra varier les différents paramètres apparaissant ci-dessus, tout en leur donnant une interprétation physique réelle, et étudier leur influence dans des situations concrètes. P. ex., à quelle vitesse, en fonction de \mathbf{v}_0 et de la densité de piétons, se déplace une foule contrainte à avancer dans un couloir si chaque individu veut maintenir une vitesse \mathbf{v}_0 ? Comment s'organise l'évacuation d'une salle initialement uniformément peuplée, avec une ou plusieurs sorties, et en la présence éventuels d'obstacles ?

Il est également possible d'essayer d'autres expressions pour les forces.

Il existe une littérature conséquente sur le sujet, que l'on pourra explorer si besoin (p. ex : [Décrypter le mouvement des piétons dans une foule](#)).

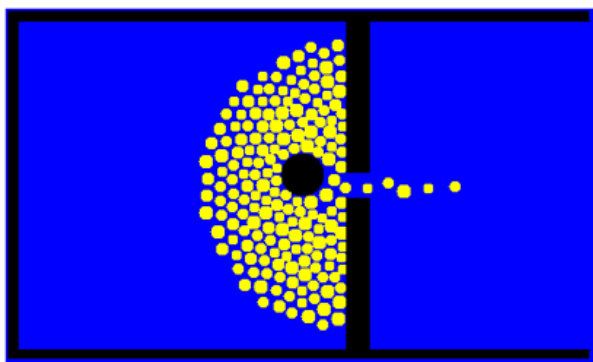


Fig. 11.6 – **Figure** : Un obstacle aide à l'évacuation. Source : [Crowd Behavior](#).

11.3.4 Suivi de particule(s)

Auteur de la section : Mathieu Leocmach <mathieu.leocmach@ens-lyon.fr>

Dans de nombreux domaines de recherche expérimentale, on a besoin de localiser des particules dans une image ou de reconstituer leurs trajectoires à partir d'une vidéo. Il peut s'agir de virus envahissant une cellule, de traceurs dans un écoulement, d'objets célestes fonçant vers la terre pour la détruire, etc.

Dans ce projet, on essaiera d'abord de localiser une particule unique dans une image à 2 dimensions (niveaux de gris) en utilisant l'algorithme de Crocker & Grier décrit [ici](#). On utilisera sans retenue les fonctions de la bibliothèque `scipy.ndimage`.

On essaiera d'obtenir une localisation plus fine que la taille du pixel. On essaiera ensuite de détecter plusieurs particules dans une image.

Afin de pouvoir traiter efficacement une séquence d'images de même taille, on privilégiera une implémentation orientée objet. L'objet de la classe `Finder` sera construit une seule fois en début de séquence et il contiendra les images intermédiaires nécessaire au traitement. On nourrira ensuite cet objet avec chaque image de la séquence pour obtenir les coordonnées des particules.

Enfin, on pourra essayer de relier les coordonnées dans des images successives pour constituer des trajectoires.

On contactera le créateur du sujet pour obtenir des séquences d'images expérimentales de particules Browniennes.

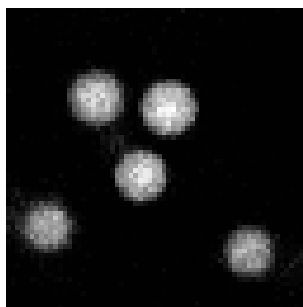


Fig. 11.7 – **Figure** : Exemple d'image test où on voudra localiser les particules.

11.4 Projets statistiques

- Tests statistiques du [NIST/SEMATECH e-Handbook of Statistical Methods](#), p.ex. [Comparisons based on data from two processes](#)
- Statistiques robustes, p.ex. [Beers et al. \(1990\)](#)

11.5 Projets de visualisation

L'objectif premier de ces projets est de développer des outils de visualisation sous Python/Matplotlib.

- [Coordonnées parallèles](#)
 - Sources éventuelles d'inspiration : [Parallel Coordinates plot in Matplotlib](#), [XDAT](#)
 - Exemples de jeu de données multi-variables : [Iris flower data set](#), [Cars](#) (source)
- [Andrew Curves](#) (voir également [Rip's Applied Mathematics Blog](#))
 - À appliquer sur les mêmes jeux de données que pour les coordonnées parallèles.
- [Stacked graphs](#)
 - Source éventuelle d'inspiration : [Python recipe](#)
- [Diagramme de Hertzsprung-Russell](#)

L'objectif est de développer une classe permettant de tracer des diagrammes HR à partir de diverses quantités observationnelles (magnitudes apparentes ou absolues, couleurs) ou théoriques (luminosité, températures effectives), ainsi que des isochrones.

- Source éventuelle d'inspiration : [Stellar evolutionary tracks](#)
- Données photométriques : p.ex. [M55](#) (source : [BVI photometry in M55](#))
- Données théoriques : p.ex. [Padova database of stellar evolutionary tracks and isochrones](#)
- [Treemaps](#)
 - Source éventuelle d'inspiration : [Treemaps under pylab](#)
- De façon plus générale, l'ensemble des visualisations proposées sous :
 - [Flare](#)
 - [D3](#)
 - [Periodic Table of Visualization Methods](#)

Démonstration Astropy

Nous présentons ici quelques possibilités de la bibliothèque [Astropy](#).

Référence : cette démonstration est très largement inspirée de la partie [Astropy](#) du cours [Python Euclid 2016](#).

```
[1]: import numpy as N
import matplotlib.pyplot as P
try:
    import seaborn
    seaborn.set_color_codes() # Override default matplotlib colors 'b', 'r', 'g', etc.
except ImportError:
    pass

# Interactive figures
# %matplotlib notebook
# Static figures
%matplotlib inline
```

12.1 Fichiers FITS

Le format [FITS](#) (*Flexible Image Transport System*) constitue le format de données historique (et encore très utilisé) de la communauté astronomique. Il permet le stockage simultané de données – sous forme de tableaux numériques multidimensionnels (spectre 1D, image 2D, cube 3D, etc.) ou de tables de données structurées (texte ou binaires) – et des métadonnées associées – sous la forme d’un entête ASCII nommé *header*. Il autorise en outre de combiner au sein d’un même fichier différents segments de données (*extensions*, p.ex. le signal et la variance associée) sous la forme de HDU (*Header-Data Units*).

Le fichier FITS de test est disponible ici : [image.fits](#) (données *Herschel Space Observatory*)

12.1.1 Lire un fichier FITS

```
[2]: from astropy.io import fits as F

filename = "image.fits"
hdulist = F.open(filename)
```

hdulist est un objet `HDUList` de type liste regroupant les différents HDU du fichier :

```
[3]: hdulist.info()

Filename: image.fits
No.    Name      Ver    Type      Cards  Dimensions  Format
  0 PRIMARY      1 PrimaryHDU    151      ()
  1 image        1 ImageHDU     52 (273, 296) float64
  2 error        1 ImageHDU     20 (273, 296) float64
  3 coverage     1 ImageHDU     20 (273, 296) float64
  4 History      1 ImageHDU     23      ()
  5 HistoryScript 1 BinTableHDU   39 105R x 1C [300A]
  6 HistoryTasks 1 BinTableHDU   46 77R x 4C [1K, 27A, 1K, 9A]
  7 HistoryParameters 1 BinTableHDU 74 614R x 10C [1K, 20A, 7A, 46A, 1L, 1K, 1L, 74A, 11A, 41A]
```

Chaque HDU contient :

- un attribut `data` pour la partie *données* sous la forme d'un `numpy.array` ou d'une structure équivalente à un tableau à type structuré;
- un attribut `header` pour la partie *métadonnées* sous la forme « KEY = value / comment ».

```
[4]: imhdu = hdulist['image']
print(type(imhdu.data), type(imhdu.header))

<class 'numpy.ndarray'> <class 'astropy.io.fits.header.Header'>
```

Il est également possible de lire *directement* les données et les métadonnées de l'extension `image` :

```
[5]: ima, hdr = F.getdata(filename, 'image', header=True)
print(type(ima), type(hdr))

<class 'numpy.ndarray'> <class 'astropy.io.fits.header.Header'>
```

`data` contient donc les données numériques, ici un tableau 2D :

```
[6]: N.info(ima)

class: ndarray
shape: (296, 273)
strides: (2184, 8)
itemsize: 8
aligned: True
contiguous: True
fortran: False
data pointer: 0x7fc7b6551ec0
byteorder: big
byteswap: True
type: >f8
```

L'entête `hdr` est un objet de type `Header` similaire à un `OrderedDict` (dictionnaire ordonné).

```
[7]: hdr[:5] # Les 5 premières clés de l'entête

[7]: XTENSION= 'IMAGE'           / Java FITS: Wed Aug 14 11:37:21 CEST 2013
    BITPIX   =                -64
    NAXIS    =                  2 / Dimensionality
    NAXIS1   =                 273
    NAXIS2   =                 296
```

Attention : les axes des tableaux FITS et NumPy arrays sont inversés !

```
[8]: print("FITS: ", (hdr['naxis1'], hdr['naxis2'])) # format de l'image FITS
    print("Numpy:", ima.shape)                    # format du tableau numpy
```

```
FITS: (273, 296)
Numpy: (296, 273)
```

12.1.2 World Coordinate System

L'entête d'un fichier FITS peut notamment inclure une description détaillée du système de coordonnées lié aux données, le [World Coordinate System](#).

```
[9]: from astropy import wcs as WCS
```

```
wcs = WCS.WCS(hdr) # Décrypte le WCS à partir de l'entête
print(wcs)
```

```
WCS Keywords
```

```
Number of WCS axes: 2
CTYPE : 'RA--TAN' 'DEC-TAN'
CRVAL : 30.07379502155236 -24.903630299920962
CRPIX : 134.0 153.0
NAXIS : 273 296
```

```
[10]: ra, dec = wcs.wcs_pix2world(0, 0, 0) # Coordonnées réelles du px (0, 0)
print("World:", ra, dec)
x, y = wcs.wcs_world2pix(ra, dec, 0) # Coordonnées dans l'image de la position (ra, dec)
print("Image:", x, y)
```

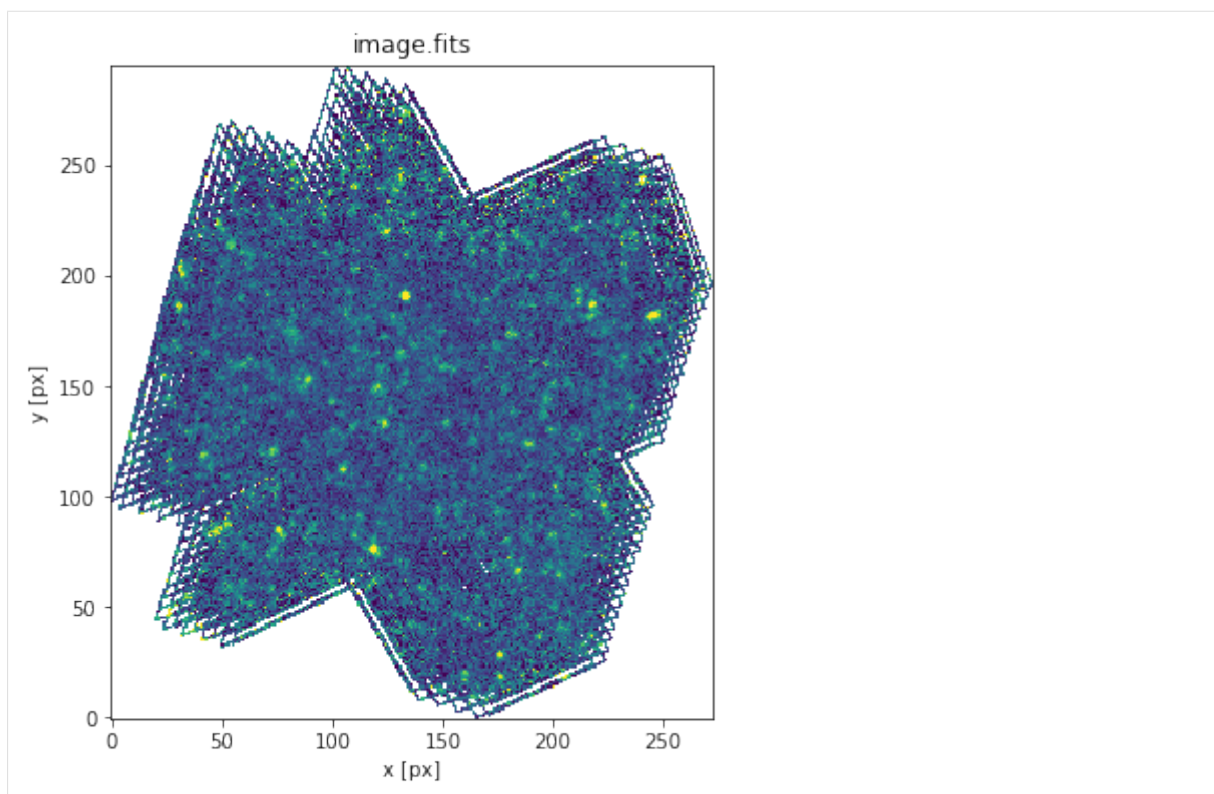
```
World: 30.31868700299246 -25.156760607162152
Image: -3.211653165635653e-12 -1.0516032489249483e-12
```

12.1.3 Visualisation dans matplotlib

Les tableaux 2D (*image*) se visualisent à l'aide de la commande `imshow` :

- `cmap` : table des couleurs;
- `vmin`, `vmax` : valeurs minimale et maximale des données visualisées;
- `origin` : position du pixel (0, 0) ("lower" = en bas à gauche).

```
[11]: fig, ax = P.subplots(figsize=(6, 6))
ax.imshow(ima, cmap='viridis', origin='lower', interpolation='None', vmin=-2e-2, vmax=5e-2)
ax.set_xlabel("x [px]")
ax.set_ylabel("y [px]")
ax.set_title(filename);
```



Il est possible d'ajouter d'autres systèmes de coordonnées via WCS.

```
[12]: import astropy.visualization as VIZ
from astropy.visualization.mpl_normalize import ImageNormalize

fig = P.figure(figsize=(8, 8))
ax = fig.add_subplot(1, 1, 1, projection=wcs)

# 10th and 99th percentiles
vmin, vmax = VIZ.AsymmetricPercentileInterval(10, 99).get_limits(ima)
# Linear normalization
norm = ImageNormalize(vmin=vmin, vmax=vmax, stretch=VIZ.LinearStretch())

ax.imshow(ima, cmap='gray', origin='lower', interpolation='None', norm=norm)

# Coordonnées équatoriales en rouge
ax.coords['ra'].set_axislabel(u' $\alpha$  [J2000]')
ax.coords['dec'].set_axislabel(u' $\delta$  [J2000]')

ax.coords['ra'].set_ticks(color='r')
ax.coords['dec'].set_ticks(color='r')

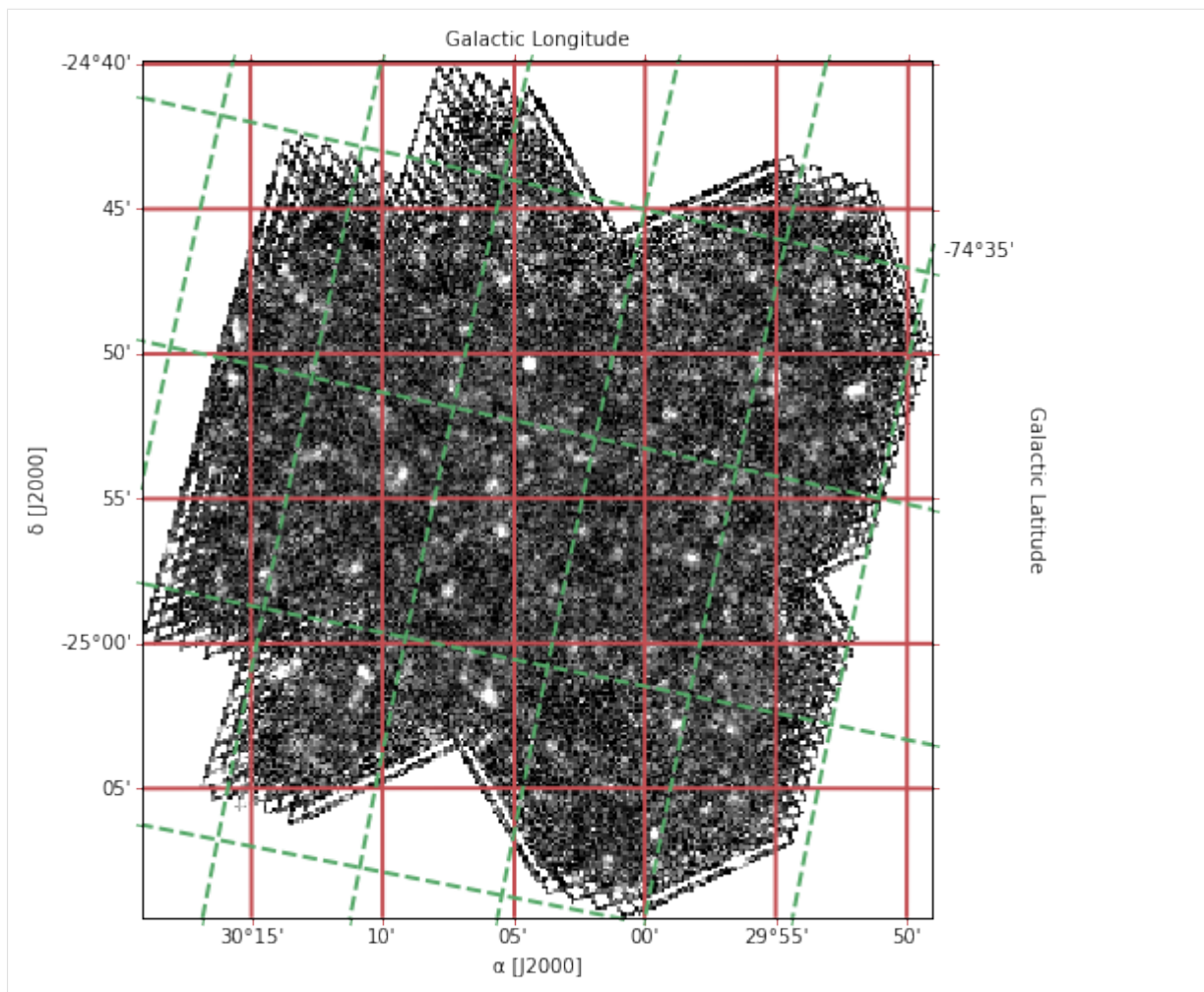
ax.coords.grid(color='r', ls='--', lw=2)

# Coordonnées galactiques en vert
overlay = ax.get_coords_overlay('galactic')

overlay['l'].set_axislabel('Galactic Longitude')
overlay['b'].set_axislabel('Galactic Latitude')

overlay['l'].set_ticks(color='g')
overlay['b'].set_ticks(color='g')

overlay.grid(color='g', ls='--', lw=2)
```



12.2 Tables

Outre les tables FITS, `astropy` permet lire et écrire des Tables dans de nombreux formats ASCII usuels en astronomie (LaTeX, HTML, CDS, SExtractor, etc.).

Le fichier ASCII de test est disponible ici : `sources.dat`

```
[13]: from astropy.io import ascii

catalog = ascii.read('sources.dat')
catalog.info()
```

```
<Table length=167>
  name      dtype
-----
      ra float64
      dec float64
      x float64
      y float64
  raPlusErr float64
  decPlusErr float64
  raMinusErr float64
  decMinusErr float64
  xPlusErr float64
  yPlusErr float64
```

(suite sur la page suivante)

(suite de la page précédente)

```

xMinusErr float64
yMinusErr float64
    flux float64
fluxPlusErr float64
fluxMinusErr float64
background float64
bgPlusErr float64
bgMinusErr float64
quality float64

```

```

[14]: catalog.sort('flux') # Ordonne les sources par 'flux' croissant
      catalog.reverse()   # Ordonne les sources par 'flux' décroissant

#catalog.show_in_notebook(display_length=5)
catalog[:5]               # Les cinq sources les plus brillantes du catalogue

```

```

[14]: <Table length=5>
      ra          dec          x      ...  bgMinusErr  quality
      float64      float64      float64  ...  float64      float64
-----
30.0736543481 -24.8389847181 133.076596062 ... 0.00280563968109 24.0841967062
30.0997563127 -25.030193106 118.886083699 ... 0.00310958937187 16.5084425251
30.2726211379 -25.0771584874 24.9485847511 ... 0.00603800958334 6.67900541976
29.8763509609 -24.7518860739 240.583543382 ... 0.00466051306854 9.08251222505
29.8668948822 -24.8539846811 245.642862323 ... 0.00330155226713 8.43689223988

```

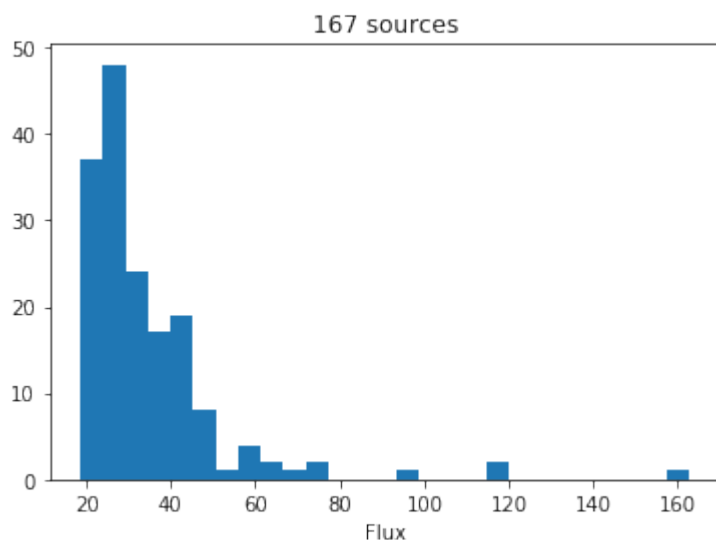
Histogramme des flux, avec choix automatique du nombre de *bins* :

```

[15]: import astropy.visualization as VIZ

fig, ax = P.subplots()
VIZ.hist(catalog['flux'], bins='freedman', ax=ax, histtype='stepfilled')
ax.set(xlabel="Flux", title="%d sources" % len(catalog));

```



Après conversion des coordonnées RA-Dec de la table en coordonnées, on peut superposer la position des 5 sources les plus brillantes du catalogue à l'image précédente :

```

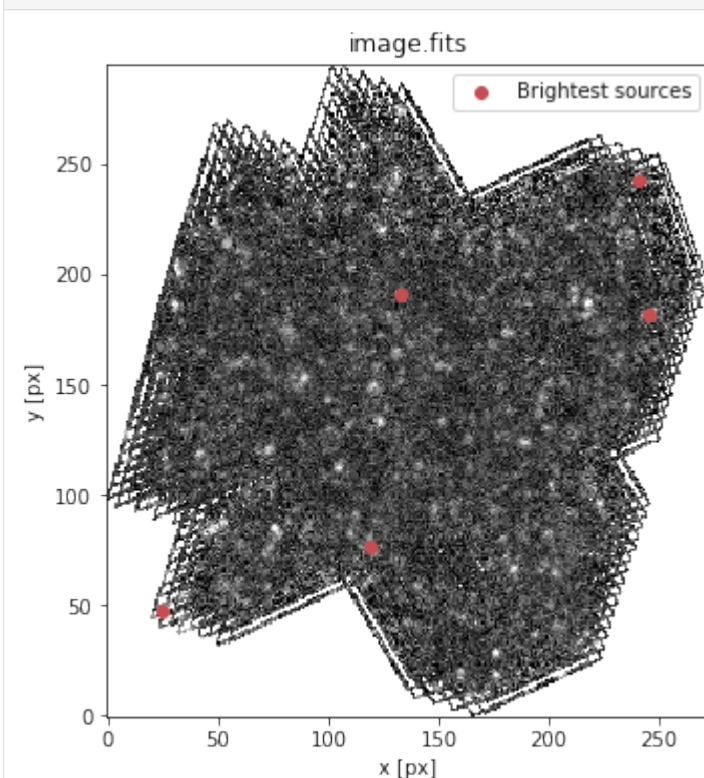
[16]: fig, ax = P.subplots(figsize=(6, 6))
      ax.imshow(ima, cmap='gray', origin='lower', interpolation='None', vmin=-2e-2, vmax=5e-2)
      ax.set(xlabel="x [px]", ylabel="y [px]", title=filename)

```

(suite sur la page suivante)

(suite de la page précédente)

```
x, y = wcs.wcs_world2pix(catalog['ra'][:5], catalog['dec'][:5], 0)
ax.scatter(x, y, color='r', label="Brightest sources")
ax.legend();
```



12.3 Quantités et unités

Astropy permet de définir des *Quantités dimensionnées* et de gérer les conversions d'unités.

```
[17]: from astropy import units as U
from astropy import constants as K

print("Vitesse de la lumière: {:.9g} = {:.9g}".format(K.c, K.c.to("Mpc/Ga")))
```

Vitesse de la lumière: 299792458 m / s = 306.601394 Mpc / Ga

```
[18]: H0 = 70 * U.km / U.s / U.Mpc
print ("H0 = {:.1f} = {:.1f} = {:.1f} = {:.3g}".format(H0, H0.to("nm/(a*km)"), H0.to("Mpc/
↪(Ga*Gpc)"), H0.decompose()))
```

H0 = 70.0 km / (Mpc s) = 71.6 nm / (a km) = 71.6 Mpc / (Ga Gpc) = 2.27e-18 1 / s

```
[19]: l = 100 * U.micron
print("{} = {}".format(l, l.to(U.GHz, equivalencies=U.spectral()))

s = 1 * U.mJy
print ("{} = {} à {}".format(s, s.to('erg/(cm**2 * s * angstrom)',
↪equivalencies=U.spectral_density(1 * U.micron)), 1 * U.
↪micron))
```

100.0 micron = 2997.9245800000003 GHz
 1.0 mJy = 2.9979245800000001e-16 erg / (Angstrom cm2 s) à 1.0 micron

12.4 Calculs cosmologiques

Astropy permet des calculs de base de cosmologie.

```
[20]: from astropy.cosmology import Planck15 as cosmo
print(cosmo)

FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.307, Tcmb0=2.725 K, Neff=3.05,
↳ m_nu=[ 0.      0.      0.06] eV, Ob0=0.0486)
```

```
[21]: z = N.logspace(-2, 1, 100)

fig = P.figure(figsize=(14, 6))

# Ax1: lookback time
ax1 = fig.add_subplot(1, 2, 1,
                      xlabel="Redshift", xscale='log')
ax1.plot(z, cosmo.lookback_time(z), 'b-')
ax1.set_ylabel("Lookback time [Ga]", color='b')
ax1.set_yscale('log')

ax1.xaxis.set_minor_locator(P.matplotlib.ticker.LogLocator(subs=range(2,10)))
ax1.yaxis.set_minor_locator(P.matplotlib.ticker.LogLocator(subs=range(2,10)))
ax1.grid(which='minor', color='w', linewidth=0.5)

# En parallèle: facteur d'échelle
ax1b = ax1.twinx()
ax1b.plot(z, cosmo.scale_factor(z), 'r-')
ax1b.set_ylabel("Scale factor", color='r')
ax1b.set_yscale('log')
ax1b.grid(False)

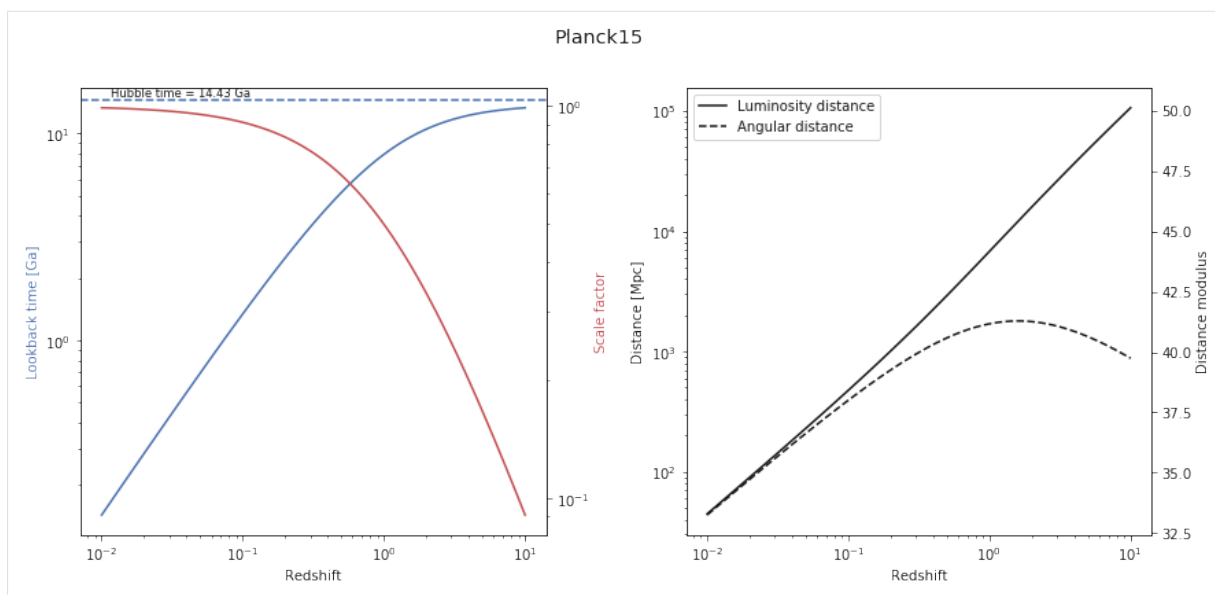
ht = (1/cosmo.H0).to('Ga') # Hubble time
ax1.axhline(ht.value, c='b', ls='--')
ax1.annotate("Hubble time = {:.2f}".format(ht), (z[1], ht.value), (3,3),
            textcoords='offset points', size='small');

# Ax2: distances de luminosité et angulaire
ax2 = fig.add_subplot(1, 2, 2,
                      xlabel="Redshift", xscale='log')
ax2.plot(z, cosmo.luminosity_distance(z), 'k-', label="Luminosity distance")
ax2.plot(z, cosmo.angular_diameter_distance(z), 'k--', label="Angular distance")
ax2.set_ylabel("Distance [Mpc]")
ax2.set_yscale('log')
ax2.legend()

ax2.xaxis.set_minor_locator(P.matplotlib.ticker.LogLocator(subs=range(2,10)))
ax2.yaxis.set_minor_locator(P.matplotlib.ticker.LogLocator(subs=range(2,10)))
ax2.grid(which='minor', color='w', linewidth=0.5)

# En parallèle, module de distance
ax2b = ax2.twinx()
ax2b.plot(z, cosmo.distmod(z), 'k-', visible=False) # Just to get the scale
ax2b.set_ylabel("Distance modulus")

fig.subplots_adjust(wspace=0.3)
fig.suptitle(cosmo.name, fontsize='x-large');
```



The following section was generated from `Cours/astropy.ipynb`

Pokémon Go ! (démonstration Pandas/Seaborn)

Voici un exemple d'utilisation des librairies [Pandas](#) (manipulation de données hétérogène) et [Seaborn](#) (visualisations statistiques), sur le [Pokémon dataset](#) d'Alberto Barradas.

Références :

- [Visualizing Pokémon Stats with Seaborn](#)
- [Pokemon Stats Analysis And Visualizations](#)

```
[1]: import pandas as PD
import seaborn as SNS
import matplotlib.pyplot as P

%matplotlib inline
```

13.1 Lecture et préparation des données

Pandas fournit des méthodes de lecture des données à partir de nombreux formats, dont les données *Comma Separated Values* :

```
[2]: df = PD.read_csv('./Pokemon.csv', index_col='Name') # Indexation sur le nom (unique)
df.info() # Informations générales
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 800 entries, Bulbasaur to Volcanion
Data columns (total 12 columns):
#                800 non-null int64
Type 1          800 non-null object
Type 2          414 non-null object
Total           800 non-null int64
HP              800 non-null int64
Attack          800 non-null int64
Defense         800 non-null int64
Sp. Atk         800 non-null int64
Sp. Def         800 non-null int64
Speed           800 non-null int64
Generation      800 non-null int64
Legendary       800 non-null bool
```

(suite sur la page suivante)

(suite de la page précédente)

```
dtypes: bool(1), int64(9), object(2)
memory usage: 75.8+ KB
```

Les premières lignes du `DataFrame` (tableau 2D) qui en résulte :

```
[3]: df.head(10) # Les 10 premières lignes
```

```
[3]:
```

	#	Type 1	Type 2	Total	HP	Attack	Defense	\
Name								
Bulbasaur	1	Grass	Poison	318	45	49	49	
Ivysaur	2	Grass	Poison	405	60	62	63	
Venusaur	3	Grass	Poison	525	80	82	83	
VenusaurMega Venusaur	3	Grass	Poison	625	80	100	123	
Charmander	4	Fire	NaN	309	39	52	43	
Charmeleon	5	Fire	NaN	405	58	64	58	
Charizard	6	Fire	Flying	534	78	84	78	
CharizardMega Charizard X	6	Fire	Dragon	634	78	130	111	
CharizardMega Charizard Y	6	Fire	Flying	634	78	104	78	
Squirtle	7	Water	NaN	314	44	48	65	

	Sp.	Atk	Sp. Def	Speed	Generation	Legendary
Name						
Bulbasaur	65		65	45	1	False
Ivysaur	80		80	60	1	False
Venusaur	100		100	80	1	False
VenusaurMega Venusaur	122		120	80	1	False
Charmander	60		50	65	1	False
Charmeleon	80		65	80	1	False
Charizard	109		85	100	1	False
CharizardMega Charizard X	130		85	100	1	False
CharizardMega Charizard Y	159		115	100	1	False
Squirtle	50		64	43	1	False

Le format est ici simple :

- nom du Pokémon (utilisé comme indice) et son n° (notons que le n° n'est pas unique)
- type primaire et éventuellement secondaire *str*
- différentes caractéristiques *int* (p.ex. points de vie, niveaux d'attaque et défense, vitesse, génération)
- type légendaire *bool*

Nous appliquons les filtres suivants directement sur le dataframe (`inplace=True`) :

- simplifier le nom des *mega* pokémons
- remplacer les NaN de la colonne « Type 2 »
- éliminer les colonnes « # » et « Sp. »

```
[4]: df.set_index(df.index.str.replace(".*(?=Mega)", ''), inplace=True) # Supprime la chaîne avant le
↳ Mega
df['Type 2'].fillna('', inplace=True) # Remplace NaN par ''
df.drop(['#'] + [col for col in df.columns if col.startswith('Sp.']],
axis=1, inplace=True) # "Laisse tomber" les colonnes commençant par 'Sp.'
df.head() # Les 5 premières lignes
```

```
[4]:
```

	Type 1	Type 2	Total	HP	Attack	Defense	Speed	Generation	\
Name									
Bulbasaur	Grass	Poison	318	45	49	49	45	1	
Ivysaur	Grass	Poison	405	60	62	63	60	1	
Venusaur	Grass	Poison	525	80	82	83	80	1	
Mega Venusaur	Grass	Poison	625	80	100	123	80	1	
Charmander	Fire		309	39	52	43	65	1	

	Legendary
Name	
Bulbasaur	False

(suite sur la page suivante)

(suite de la page précédente)

Ivysaur	False
Venusaur	False
Mega Venusaur	False
Charmander	False

13.2 Accès aux données

Pandas propose de multiples façons d'accéder aux données d'un DataFrame, ici :

— via le nom (indexé) :

```
[5]: df.loc['Bulbasaur', ['Type 1', 'Type 2']] # Seulement 2 colonnes
```

```
[5]: Type 1    Grass
     Type 2    Poison
     Name: Bulbasaur, dtype: object
```

— par sa position dans la liste :

```
[6]: df.iloc[-5:, :2] # Les 5 dernières lignes, et les 2 premières colonnes
```

```
[6]:      Type 1 Type 2
     Name
Diancie      Rock  Fairy
Mega Diancie      Rock  Fairy
HoopaHoopa Confined  Psychic  Ghost
HoopaHoopa Unbound  Psychic  Dark
Volcanion      Fire  Water
```

— par une sélection booléenne, p.ex. tous les pokémons légendaires de type herbe :

```
[7]: df[df['Legendary'] & (df['Type 1'] == 'Grass')]
```

```
[7]:      Type 1  Type 2  Total  HP  Attack  Defense  Speed  \
     Name
ShayminLand Forme  Grass      600  100    100    100    100
ShayminSky Forme  Grass  Flying    600  100    103    75    127
Virizion          Grass  Fighting    580   91    90    72    108

      Generation  Legendary
     Name
ShayminLand Forme      4     True
ShayminSky Forme      4     True
Virizion            5     True
```

13.3 Quelques statistiques

```
[8]: df[['Total', 'HP', 'Attack', 'Defense']].describe() # Description statistique des différentes
     ↪ colonnes
```

```
[8]:      Total      HP      Attack      Defense
count  800.00000  800.000000  800.000000  800.000000
mean    435.10250   69.258750   79.001250   73.842500
std     119.96304   25.534669   32.457366   31.183501
min     180.00000    1.000000    5.000000    5.000000
25%     330.00000   50.000000   55.000000   50.000000
50%     450.00000   65.000000   75.000000   70.000000
75%     515.00000   80.000000  100.000000   90.000000
max     780.00000  255.000000  190.000000  230.000000
```

```
[9]: df.loc[df['HP'].idxmax()] # Pokémon ayant le plus de points de vie
```

```
[9]: Type 1      Normal
     Type 2
     Total      540
     HP         255
     Attack      10
     Defense     10
     Speed       55
     Generation   2
     Legendary    False
     Name: Blissey, dtype: object
```

```
[10]: df.sort_values('Speed', ascending=False).head(3) # Les 3 pokémons plus rapides
```

```
[10]:
```

	Type 1	Type 2	Total	HP	Attack	Defense	Speed \
Name							
DeoxysSpeed Forme	Psychic		600	50	95	90	180
Ninjask	Bug	Flying	456	61	90	45	160
DeoxysNormal Forme	Psychic		600	50	150	50	150

	Generation	Legendary
Name		
DeoxysSpeed Forme	3	True
Ninjask	3	False
DeoxysNormal Forme	3	True

Statistiques selon le statut « légendaire » :

```
[11]: legendary = df.groupby('Legendary')
     legendary.size()
```

```
[11]: Legendary
     False    735
     True     65
     dtype: int64
```

```
[12]: legendary['Total', 'HP', 'Attack', 'Defense', 'Speed'].mean()
```

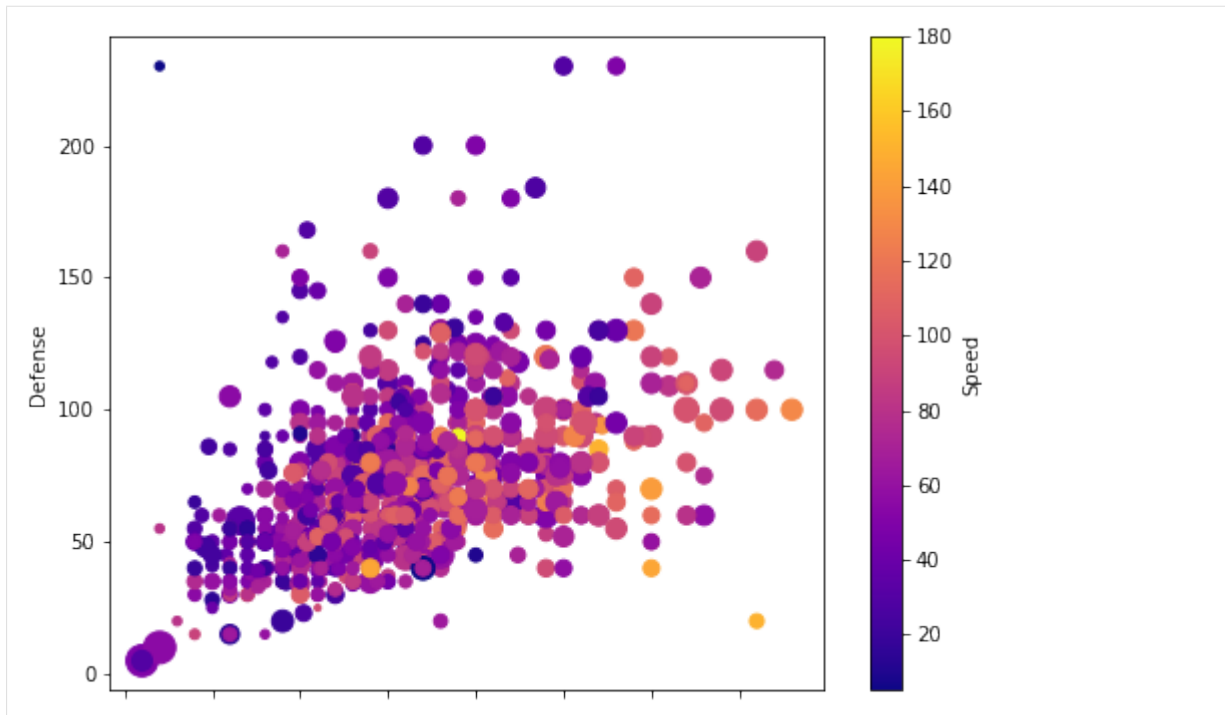
```
[12]:
```

	Total	HP	Attack	Defense	Speed
Legendary					
False	417.213605	67.182313	75.669388	71.559184	65.455782
True	637.384615	92.738462	116.676923	99.661538	100.184615

13.4 Visualisation

Pandas intègre de nombreuses fonctions de visualisation interfacées à matplotlib.

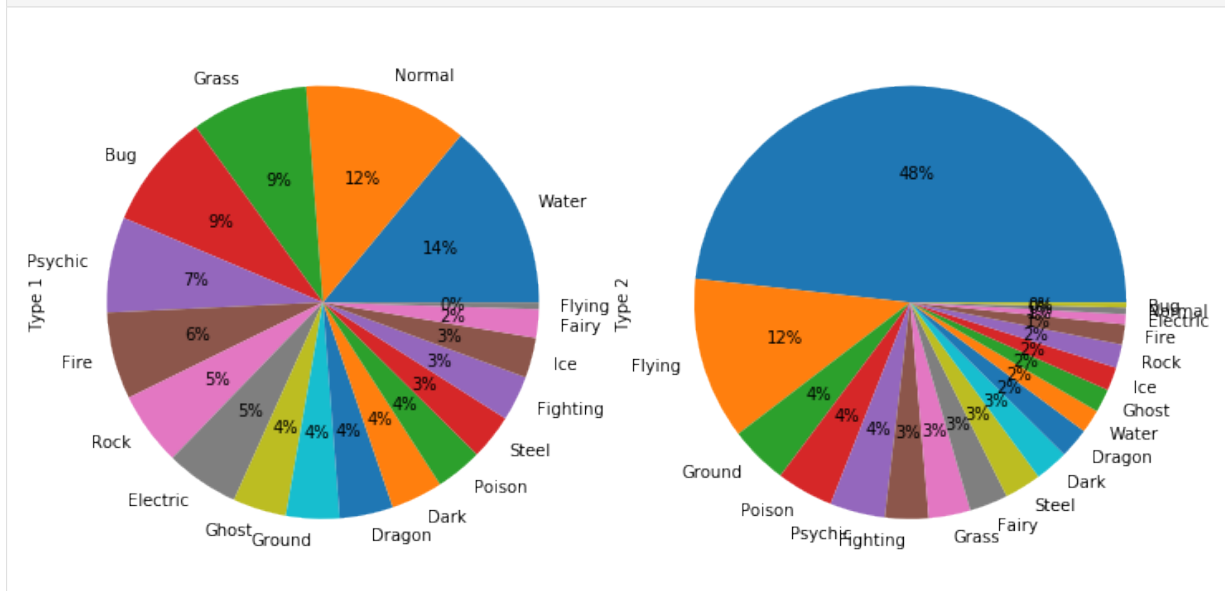
```
[13]: ax = df.plot.scatter(x='Attack', y='Defense', s=df['HP'], c='Speed', cmap='plasma')
     ax.figure.set_size_inches((8, 6))
```

```
[14]: fig, (ax1, ax2) = P.subplots(1, 2, subplot_kw={"aspect": 'equal'}, figsize=(10, 6))

df['Type 1'].value_counts().plot.pie(ax=ax1, autopct='%0f%%')
df['Type 2'].value_counts().plot.pie(ax=ax2, autopct='%0f%%')

fig.tight_layout()
```



Il est également possible d'utiliser la librairie `seaborn`, qui s'interface naturellement avec `Pandas`.

```
[15]: pok_type_colors = {      # http://bulbapedia.bulbagarden.net/wiki/Category:Type_color_templates
    'Grass': '#78C850',
    'Fire': '#F08030',
    'Water': '#6890F0',
    'Bug': '#A8B820',
    'Normal': '#A8A878',
    'Poison': '#A040A0',
```

(suite sur la page suivante)

(suite de la page précédente)

```

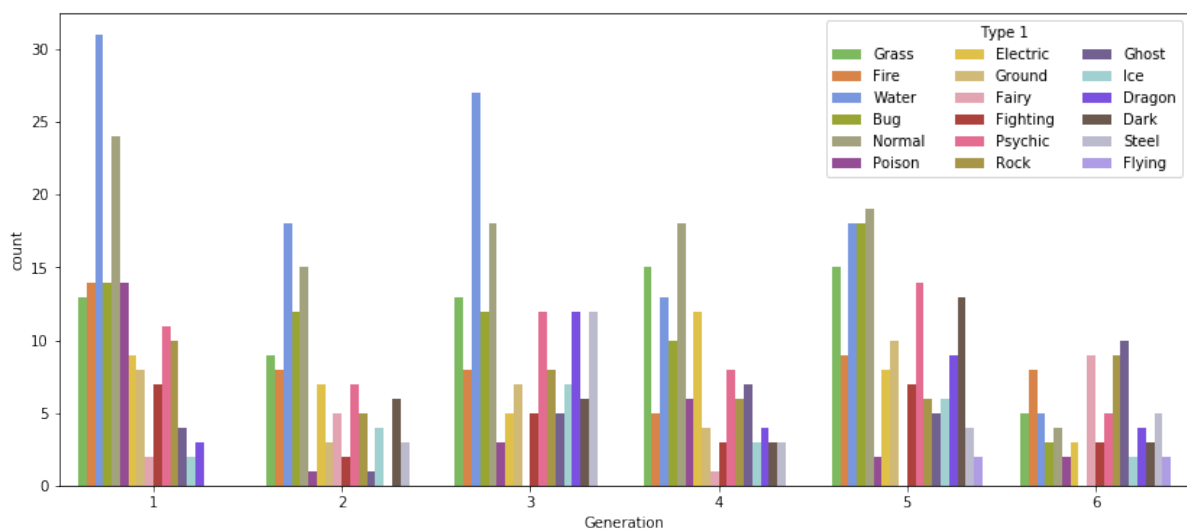
'Electric': '#F8D030',
'Ground': '#E0C068',
'Fairy': '#EE99AC',
'Fighting': '#C03028',
'Psychic': '#F85888',
'Rock': '#B8A038',
'Ghost': '#705898',
'Ice': '#98D8D8',
'Dragon': '#7038F8',
'Dark': '#705848',
'Steel': '#B8B8D0',
'Flying': '#A890F0',
}

```

```

[16]: ax = sns.countplot(x='Generation', hue='Type 1', palette=pok_type_colors, data=df)
ax.figure.set_size_inches((14, 6))
ax.legend(ncol=3, title='Type 1');

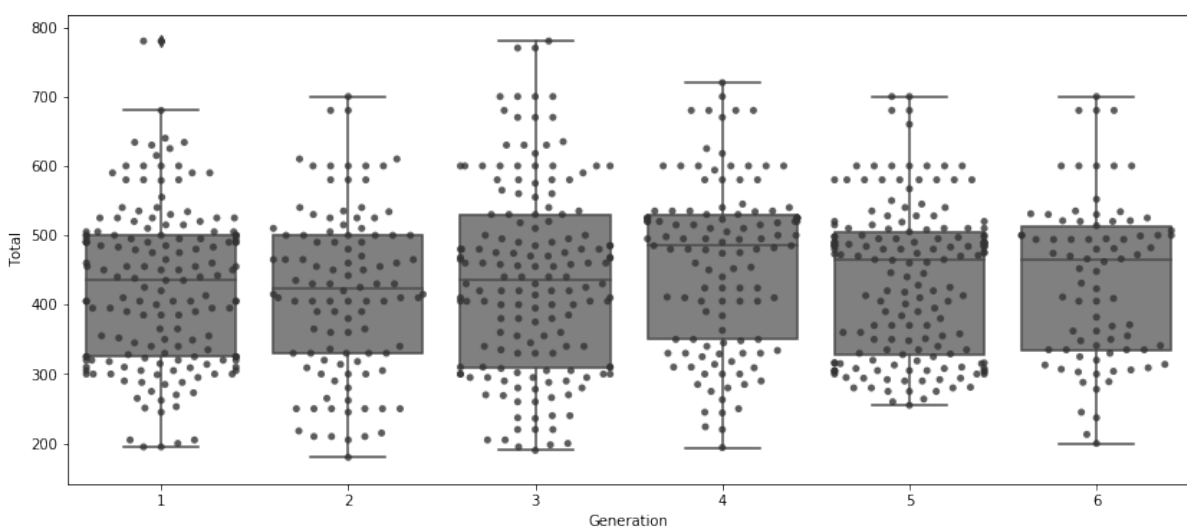
```



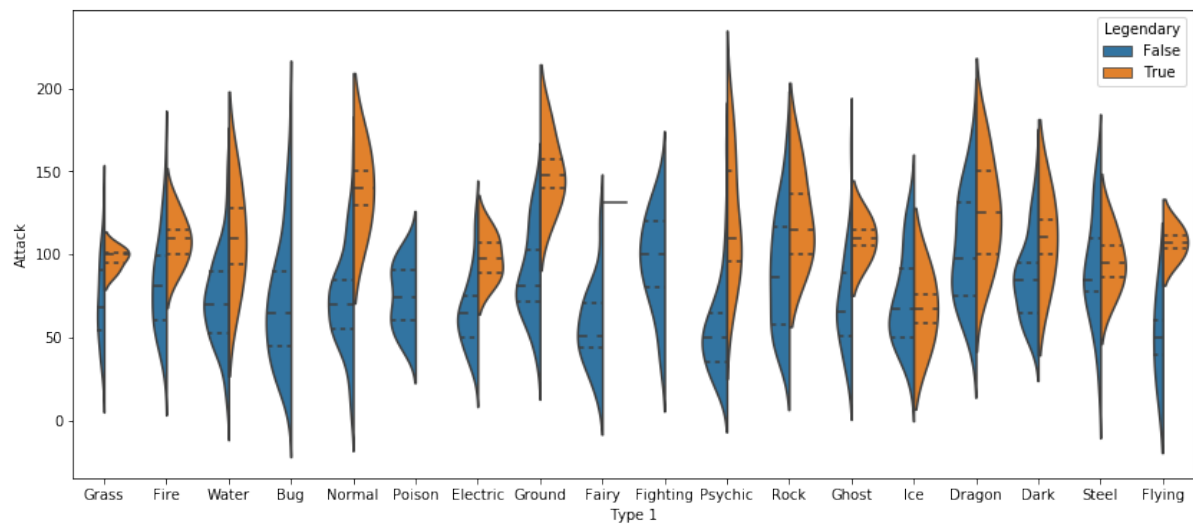
```

[17]: ax = sns.boxplot(x='Generation', y='Total', data=df, color='0.5');
sns.swarmplot(x='Generation', y='Total', data=df, color='0.2', alpha=0.8)
ax.figure.set_size_inches((14, 6))

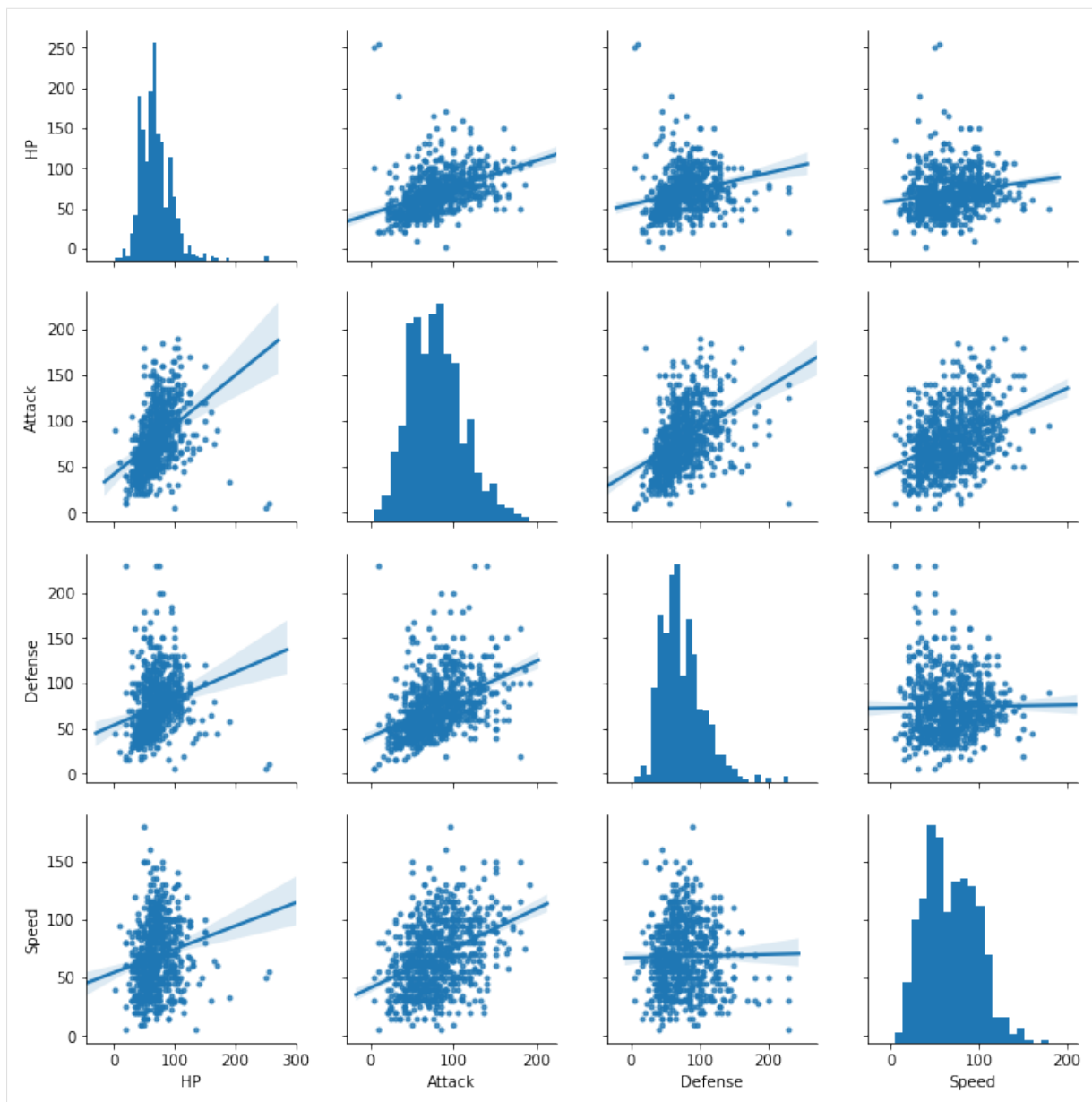
```



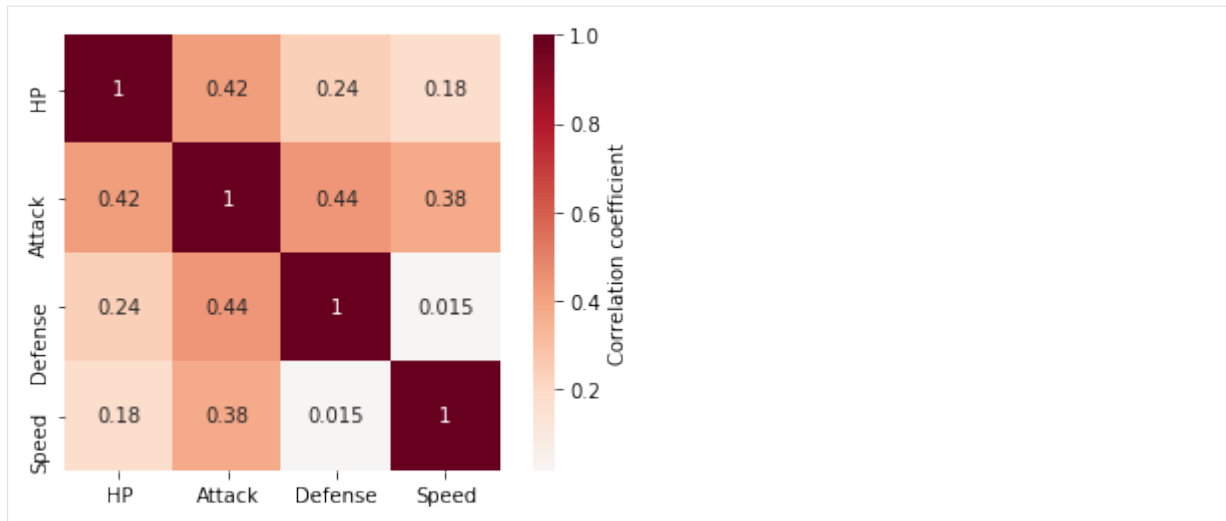
```
[18]: ax = sns.violinplot(x="Type 1", y="Attack", data=df, hue="Legendary", split=True, inner='quart
↪')
ax.figure.set_size_inches((14, 6))
```



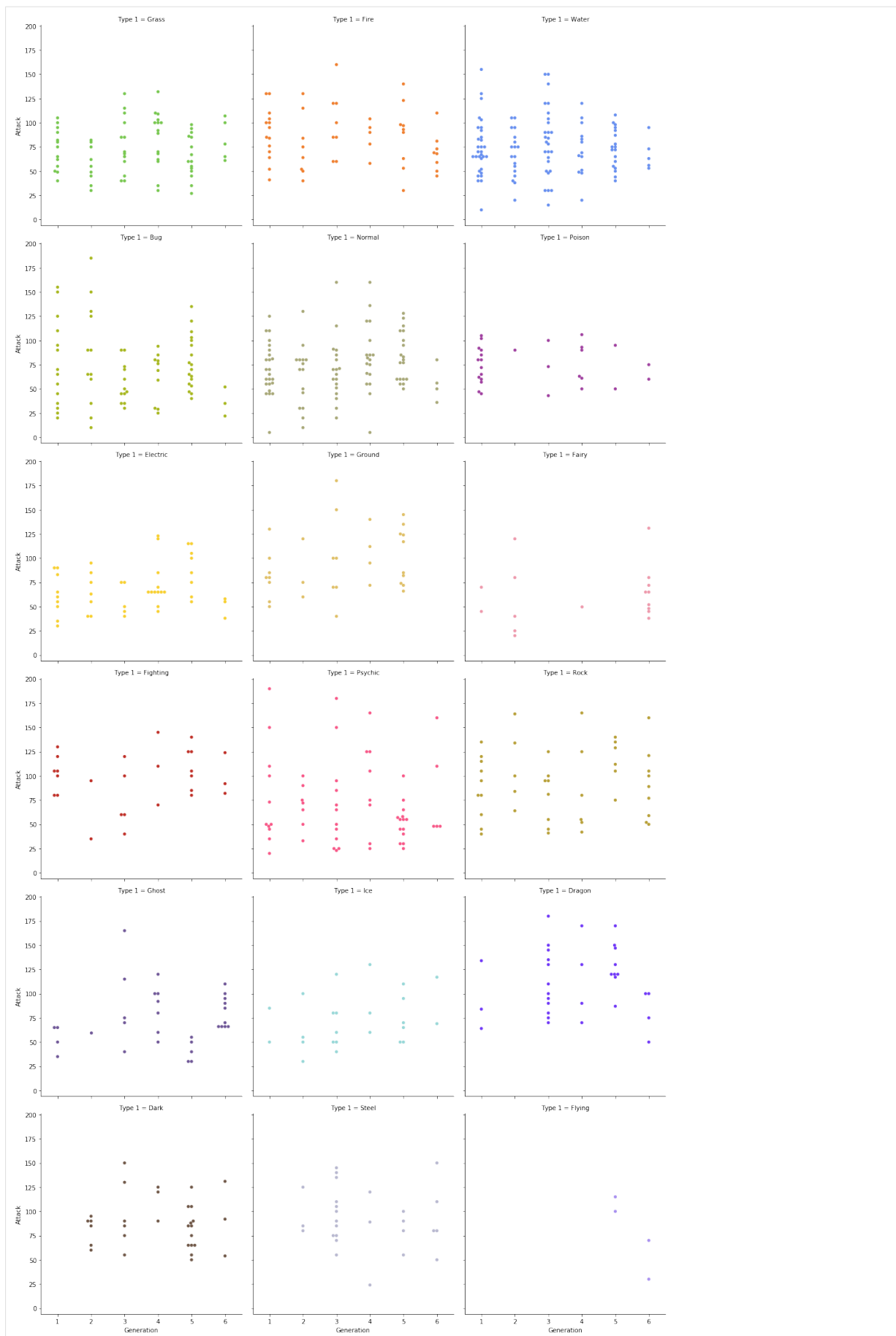
```
[19]: df2 = df.drop(['Total', 'Generation', 'Legendary'], axis=1)
sns.pairplot(df2, markers='.', kind='reg');
```



```
[20]: ax = sns.heatmap(df2.corr(), annot=True,
                      cmap='RdBu_r', center=0, cbar_kws={'label': 'Correlation coefficient'})
ax.set_aspect('equal')
```



```
[21]: sns.catplot(x='Generation', y='Attack', data=df,  
                hue='Type 1', palette=pok_type_colors, col='Type 1', col_wrap=3, kind='swarm');
```



The following section was generated from `Cours/pokemon.ipynb`

Méthode des rectangles

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-16 18:07:22 ycopin>
4
5  """
6  Calcul de l'intégrale de x**2 entre 0 et 1 par la méthode des rectangles
7  (subdivision en 100 pas)
8  """
9
10 def sq(x):
11     "Définition de la fonction sq: x → x**2."
12
13     return x**2
14
15 a, b = 0, 1                # Bornes d'intégration
16 n = 100                   # Nombre de pas
17
18 h = (b - a) / n           # Largeur des rectangles
19
20 total = 0                  # Cette variable accumulera les aires des rectangles
21 for i in range(n):         # Boucle de 0 à n - 1
22     x = a + (i + 0.5) * h  # Abscisse du rectangle
23     total += sq(x) * h     # On ajoute l'aire du rectangle au total
24
25 print("Intégrale de x**2 entre a =", a, "et b =", b, "avec n =", n, "rectangles")
26 # On affiche les résultats numérique et analytique, ainsi que l'erreur relative
27 print("Résultat numérique: ", total)
28 theorie = (b ** 3 - a ** 3) / 3
29 print("Résultat analytique:", theorie)
30 print("Erreur relative:", (total / theorie - 1))
```

```
$ python3 integ.py
```

```
Intégrale de x**2 entre a = 0 et b = 1 avec n = 100 rectangles
Résultat numérique:  0.33332500000000004
Résultat analytique: 0.3333333333333333
Erreur relative: -2.4999999999830713e-05
```

Source : integ.py

Fizz Buzz

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-26 16:46 ycopin@lyonovae03.in2p3.fr>
4
5  """
6  Jeu du Fizz Buzz
7  """
8
9  __author__ = "Yannick Copin <y.copin@ipnl.in2p3.fr>"
10
11
12 for i in range(1, 100):                # Entiers de 1 à 99
13     if ((i % 3) == 0) and ((i % 5) == 0): # Multiple de 3 *et* de 5
14         print('FIZZ BUZZ!', end=' ')      # Affichage sans retour à la ligne
15     elif (i % 3) == 0:                # Multiple de 3 uniquement
16         print('Fizz!', end=' ')
17     elif (i % 5) == 0:                # Multiple de 5 uniquement
18         print('Buzz!', end=' ')
19     else:
20         print(i, end=' ')
21 print()                               # Retour à la ligne final
```

```
$ python3 fizz.py
```

```
1 2 Fizz! 4 Buzz! Fizz! 7 8 Fizz! Buzz! 11 Fizz! 13 14 FIZZ BUZZ! 16...
```

Source : fizz.py

Algorithme d'Euclide

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-26 16:56 ycopin@lyonovae03.in2p3.fr>
4
5  """
6  Calcul du PGCD de deux entiers.
7  """
8
9  # Entiers dont on calcule le PGCD
10 a = 306
11 b = 756
12
13 # Test usuels : les nombres sont bien positifs et a > b
14 assert (a > 0) and (b > 0)
15 if a < b:
16     a, b = b, a                # Interversion: on suppose a >= b
17
18 a0, b0 = a, b                 # On garde une copie des valeurs originales
19
20 # On boucle jusqu'à ce que le reste soit nul, d'où la boucle while. Il faut
21 # être sûr que l'algorithme converge dans tous les cas!
22 while b != 0:
23     # On remplace a par b et b par le reste de la division euclidienne
24     # de a par b
25     a, b = b, a % b
26
27 print('Le PGCD de', a0, 'et', b0, 'vaut', a) # On affiche le résultat
28 # Vérifications
29 print(a0 // a, 'x', a, '=', (a0 // a * a)) # a//b: division euclidienne
30 print(b0 // a, 'x', a, '=', (b0 // a * a))

```

```
$ python3 pgcd.py
```

```

Le PGCD de 756 et 306 vaut 18
42 x 18 = 756
17 x 18 = 306

```

Source : pgcd.py

Crible d'Ératosthène

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  Crible d'Ératosthène.
6
7  Source: http://fr.wikibooks.org/wiki/Exemples\_de\_scripts\_Python#Impl%C3%A9mentation\_du\_crible\_d'
8  ↪ 'Ératosthène
9  """
10
11  __author__ = "Yannick Copin <y.copin@ipnl.in2p3.fr>"
12
13  # start-sys
14  # Gestion simplifiée d'un argument entier sur la ligne de commande
15  import sys
16
17  if sys.argv[1:]: # Présence d'au moins un argument sur la ligne de commande
18      try:
19          n = int(sys.argv[1]) # Essayer de lire le 1er argument comme un entier
20      except ValueError:
21          raise ValueError("L'argument '{}' n'est pas un entier".format(sys.argv[1]))
22  else:
23      n = 101 # Pas d'argument sur la ligne de commande
24      # Valeur par défaut
25
26  # end-sys
27
28  # Liste des entiers *potentiellement* premiers. Les nb non premiers
29  # seront étiquetés par 0 au fur et à mesure.
30  l = list(range(n + 1)) # <0,...,n>, 0 n'est pas premier
31  l[1] = 0 # 1 n'est pas premier
32
33  i = 2 # Entier à tester
34  while i**2 <= n: # Inutile de tester jusqu'à n
35      if l[i] != 0: # Si i n'est pas étiqueté (=0)...
36          # ...étiqueter tous les multiples de i
37          l[2 * i::i] = [0] * len(l[2 * i::i])
38          i += 1 # Passer à l'entier à tester suivant
```

(suite sur la page suivante)

(suite de la page précédente)

```
38 # Afficher la liste des entiers premiers (c-à-d non étiquetés)
39 print("Liste des entiers premiers <= {} :".format(n))
40 print([i for i in l if i != 0])
```

```
$ python3 crible.py
```

```
Liste des entiers premiers <= 101
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, ↵
↵97, 101]
```

Source : crible.py

CHAPITRE 18

Carré magique

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-16 18:13:01 ycopin>
4
5  """
6  Création et affichage d'un carré magique d'ordre impair.
7  """
8
9  __author__ = "Yannick Copin <y.copin@ipnl.in2p3.fr>"
10
11  n = 5                                # Ordre du carré magique
12
13  # On vérifie que l'ordre est bien impair
14  assert n % 2 == 1, "L'ordre {} n'est pas impair".format(n)
15
16  # Le carré sera stocké dans une liste de n listes de n entiers
17  # Initialisation du carré: liste de n listes de n zéros.
18  array = [[0 for j in range(n)] for i in range(n)]
19
20  # Initialisation de l'algorithme
21  i, j = n, (n + 1) // 2              # Indices de l'algo (1-indexation)
22  array[i - 1][j - 1] = 1             # Attention: python utilise une 0-indexation
23
24  # Boucle sur valeurs restant à inclure dans le carré (de 2 à n**2)
25  for k in range(2, n**2 + 1):
26      # Test de la case i+1, j+1 (modulo n)
27      i2 = (i + 1) % n
28      j2 = (j + 1) % n
29      if array[i2 - 1][j2 - 1] == 0: # La case est vide: l'utiliser
30          i, j = i2, j2
31      # La case est déjà remplie: utiliser la case i-1, j
32      else:
33          i = (i - 1) % n
34          array[i - 1][j - 1] = k      # Remplissage de la case
35
36  # Affichage, avec vérification des sommes par ligne et par colonne
37  print("Carré magique d'ordre {}:".format(n))
38  for row in array:
```

(suite sur la page suivante)

(suite de la page précédente)

```
39     print(' '.join('{:2d}'.format(k) for k in row), '=', sum(row))
40 print(' '.join('==' for k in row))
41 print(' '.join(str(sum(array[i][j] for i in range(n))) for j in range(n)))
```

```
$ python3 carre.py
```

```
Carré magique d'ordre 5:
```

```
11  18  25   2   9 = 65
10  12  19  21   3 = 65
  4   6  13  20  22 = 65
23   5   7  14  16 = 65
17  24   1   8  15 = 65
== == == == ==
65 65 65 65 65
```

Source : `carre.py`

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-26 16:57 ycopin@lyonovae03.in2p3.fr>
4
5  __author__ = "Adrien Licari <adrien.licari@ens-lyon.fr>; Yannick Copin <y.copin@ipnl.in2p3.fr>"
6
7
8  def suite_syracuse(n):
9      """
10     Retourne la suite de Syracuse pour l'entier n.
11
12     >>> suite_syracuse(15)
13     [15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
14     """
15
16     seq = [n]                                # La suite de Syracuse sera complétée...
17     while seq[-1] != 1:                      # ...jusqu'à tomber sur 1
18         if seq[-1] % 2 == 0:                 # u_n est pair
19             seq.append(seq[-1] // 2)        # Division euclidienne par 2
20         else:                                # u_n est impair
21             seq.append(seq[-1] * 3 + 1)
22
23     return seq
24
25
26  def temps_syracuse(n, altitude=False):
27      """
28     Calcule le temps de vol (éventuellement en altitude) de la suite
29     de Syracuse pour l'entier n.
30
31     >>> temps_syracuse(15)
32     17
33     >>> temps_syracuse(15, altitude=True)
34     10
35     """
36
37     seq = suite_syracuse(n)
38     if not altitude:                         # Temps de vol total

```

(suite sur la page suivante)

(suite de la page précédente)

```
39     return len(seq) - 1
40 else:                                     # Temps de vol en altitude
41     # Construction de la séquence en altitude
42     alt = []
43     for i in seq:
44         if i >= n:
45             alt.append(i)
46         else:
47             break
48     return len(alt) - 1
49
50 if __name__ == '__main__':
51
52     n = 15
53     print("Suite de Syracuse pour n =", n)
54     print(suite_syracuse(n))
55     print("Temps de vol total:      ", temps_syracuse(n))
56     print("Temps de vol en altitude:", temps_syracuse(n, altitude=True))
```

```
Suite de Syracuse pour n = 15
[15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
Temps de vol total:      17
Temps de vol en altitude: 10
```

Source : syracuse.py

CHAPITRE 20

Flocon de Koch

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  from __future__ import division # Pas de division euclidienne par défaut
5
6  """
7  Tracé (via 'turtle') d'un flocon de Koch d'ordre arbitraire.
8
9  Dans le même genre:
10
11  - courbe de Peano (http://fr.wikipedia.org/wiki/Courbe\_de\_Peano)
12  - courbe de Hilbert (http://fr.wikipedia.org/wiki/Courbe\_de\_Hilbert)
13  - île de Gosper (http://fr.wikipedia.org/wiki/Île\_de\_Gosper)
14
15  Voir également:
16
17  - L-système: http://fr.wikipedia.org/wiki/L-système
18  - Autres exemples: http://natesoares.com/tutorials/python-fractals/
19  """
20
21  import turtle as T
22
23  __version__ = "Time-stamp: <2013-01-14 00:49 ycopin@lyopc469>"
24  __author__ = "Yannick Copin <y.copin@ipnl.in2p3.fr>"
25
26
27  def koch(ordre=3, niveau=0, taille=100, delta=0):
28      """
29      Tracé du flocon de Koch d'ordre 'ordre', de taille 'taille'
30      (px).
31
32      Cette fonction récursive permet d'initialiser le flocon (niveau=0,
33      par défaut), de tracer les branches fractales (0 < niveau <= ordre) ou
34      bien juste de tracer un segment (niveau > ordre).
35      """
36
37      if niveau == 0:                                # Dessine le triangle de niveau 0
38          T.title("Flocon de Koch - ordre {}".format(ordre))
```

(suite sur la page suivante)

(suite de la page précédente)

```

39     koch(niveau=1, ordre=ordre, taille=taille, delta=delta)
40     T.right(120)
41     koch(niveau=1, ordre=ordre, taille=taille, delta=delta)
42     T.right(120)
43     koch(niveau=1, ordre=ordre, taille=taille, delta=delta)
44     elif niveau <= ordre:                # Trace une section _/\_ du flocon
45         koch(niveau=niveau + 1, ordre=ordre, taille=taille, delta=delta)
46         T.left(60 + delta)
47         koch(niveau=niveau + 1, ordre=ordre, taille=taille, delta=delta)
48         T.right(120 + 2 * delta)
49         koch(niveau=niveau + 1, ordre=ordre, taille=taille, delta=delta)
50         T.left(60 + delta)
51         koch(niveau=niveau + 1, ordre=ordre, taille=taille, delta=delta)
52     else:                                # Trace le segment de dernier niveau
53         T.forward(taille / 3 ** (ordre + 1))
54
55 if __name__ == '__main__':
56
57     # start-argparse
58     # Exemple d'utilisation de la bibliothèque de gestion d'arguments 'argparse'
59     import argparse
60
61     desc = "Tracé (via 'turtle') d'un flocon de Koch d'ordre arbitraire."
62
63     # Définition des options
64     parser = argparse.ArgumentParser(description=desc)
65     parser.add_argument('ordre', nargs='?', type=int,
66                         help="Ordre du flocon, >0 [%s]" % (default)s]",
67                         default=3)
68     parser.add_argument('-t', '--taille', type=int,
69                         help="Taille de la figure, >0 [%s px]" % (default)s]",
70                         default=500)
71     parser.add_argument('-d', '--delta', type=float,
72                         help="Delta [%s deg]" % (default)s]",
73                         default=0.)
74     parser.add_argument('-f', '--figure', type=str,
75                         help="Nom de la figure de sortie (format EPS)")
76     parser.add_argument('-T', '--turbo',
77                         action="store_true", default=False,
78                         help="Mode Turbo")
79
80     # Déchiffrage des options et arguments
81     args = parser.parse_args()
82
83     # Quelques tests sur les args et options
84     if not args.ordre > 0:
85         parser.error("Ordre requis '{0}' invalide".format(args.ordre))
86
87     if not args.taille > 0:
88         parser.error("La taille de la figure doit être positive")
89     # end-argparse
90
91     if args.turbo:
92         T.hideturtle()
93         T.speed(0)
94
95     # Tracé du flocon de Koch d'ordre 3
96     koch(ordre=args.ordre, taille=args.taille, delta=args.delta)
97     if args.figure:
98         # Sauvegarde de l'image
99         print("Sauvegarde de la figure dans '{0}'".format(args.figure))

```

(suite sur la page suivante)

(suite de la page précédente)

```
100     T.getscreen().getcanvas().postscript(file=args.figure)
101
102     T.exitonclick()
```

Source : koch.py

Jeu du plus ou moins

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  nmin, nmax = 1, 100
7  nsol = random.randint(nmin, nmax)
8
9  print("Vous devez deviner un nombre entre {} et {}".format(nmin, nmax))
10 ncoups = 0
11 try:
12     while True:
13         proposition = input("Votre proposition: ")
14         try:
15             ntest = int(proposition)
16             if not nmin <= ntest <= nmax:
17                 raise ValueError("Proposition invalide")
18         except ValueError:
19             print("Votre proposition {!r} n'est pas un entier compris entre {} et {}."
20                   .format(proposition, nmin, nmax))
21             continue
22         ncoups += 1
23         if nsol > ntest:
24             print("C'est plus.")
25         elif nsol < ntest:
26             print("C'est moins.")
27         else:
28             print("Vous avez trouvé en {} coup{}".format(
29                   ncoups, 's' if ncoups > 1 else ''))
30             break # Interrompt la boucle while
31 except (KeyboardInterrupt, EOFError): # Interception de Ctrl-C et Ctrl-D
32     print("\nVous abandonnez après seulement {} coup{}".format(
33         ncoups, 's' if ncoups > 1 else ''))
```

Source : pm.py


```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  """
5  Exercice: programmation orientée objet, développement dirigé par les tests.
6  """
7
8  import pytest                      # Module (non standard) de tests
9
10
11 class Animal:
12     """
13     Classe définissant un `Animal`, caractérisé par son nom et son
14     poids.
15     """
16
17     def __init__(self, nom, masse):
18         """
19         Méthode d'instanciation à partir d'un nom (str) et d'un poids (float).
20         """
21
22         # Ici, convertir les paramètres pour être sûr qu'il ont le bon
23         # type. On utilisera `str` et `float`
24         self.nom = str(nom)
25         self.masse = float(masse)
26
27         self.vivant = True          # Les animaux sont vivants à l'instanciation
28         self.empoisonne = False    # Animal empoisonné ?
29
30     def __str__(self):
31         """
32         Surcharge de l'opérateur `str`: l'affichage *informel* de l'objet
33         dans l'interpréteur, p.ex. `print self` sera résolu comme
34         `self.__str__()`
35
36         Retourne une chaîne de caractères.
37         """
38
```

(suite sur la page suivante)

(suite de la page précédente)

```

39     return "{a.nom} ({a.masse:.1f} kg)".format(a=self)
40
41     def estVivant(self):
42         """Méthode booléenne, vraie si l'animal est vivant."""
43
44         return self.vivant
45
46     def mourir(self):
47         """Change l'état interne de l'objet (ne retourne rien)."""
48
49         self.vivant = False
50
51     def __lt__(self, other):
52         """
53         Surcharge l'opérateur de comparaison '<' uniquement, sur la
54         base de la masse des animaux.
55
56         Note: Py3 impose de surcharger *explicitement* tous les opérateurs
57         de comparaison: '__lt__' pour '<', '__le__' pour '<=', '__eq__'
58         pour '==', etc.
59         """
60
61         return self.masse < other.masse
62
63     def __call__(self, other):
64         """
65         Surcharge de l'opérateur '()' pour manger un autre animal (qui
66         meurt s'il est vivant) et prendre du poids (mais pas plus que
67         la masse de l'autre ou 10% de son propre poids). Attention aux
68         animaux empoisonnés!
69
70         L'instruction `self(other)` sera résolue comme
71         `self.__call__(other)`.
72         """
73
74         other.mourir()
75         poids = min(other.masse, self.masse * 0.1)
76         self.masse += poids
77         other.masse -= poids
78         if other.empoisonne:
79             self.mourir()
80
81
82     class Chien(Animal):
83         """
84         Un `Chien` hérite de `Animal` avec des méthodes additionnelles
85         (p.ex. l'aboieement et l'odorat).
86         """
87
88         def __init__(self, nom, masse=20, odorat=0.5):
89             """Définit un chien plus ou moins fin limier."""
90
91             # Initialisation de la classe parente
92             Animal.__init__(self, nom, masse)
93
94             # Attribut propre à la classe dérivée
95             self.odorat = float(odorat)
96
97         def __str__(self):
98
99             return "{a.nom} (Chien, {a.masse:.1f} kg)".format(a=self)

```

(suite sur la page suivante)

(suite de la page précédente)

```

100
101     def aboyer(self):
102         """Une méthode bien spécifique aux chiens."""
103
104         print("Ouaf ! Ouaf !")
105
106     def estVivant(self):
107         """Quand on vérifie qu'un chien est vivant, il aboie."""
108
109         vivant = Animal.estVivant(self)
110
111         if vivant:
112             self.aboyer()
113
114         return vivant
115
116     #####
117     # Il est *INTERDIT* de modifier les tests ci-dessous!!! #
118     #####
119
120     # start-tests
121     def test_empty_init():
122         with pytest.raises(TypeError):
123             Animal()
124
125
126     def test_wrong_init():
127         with pytest.raises(ValueError):
128             Animal('Youki', 'lalala')
129
130
131     def test_init():
132         youki = Animal('Youki', 600)
133         assert youki.masse == 600
134         assert youki.vivant
135         assert youki.estVivant()
136         assert not youki.empoisonne
137     # end-tests
138
139
140     def test_str():
141         youki = Animal('Youki', 600)
142         assert str(youki) == 'Youki (600.0 kg)'
143
144
145     def test_mort():
146         youki = Animal('Youki', 600)
147         assert youki.estVivant()
148         youki.mourir()
149         assert not youki.estVivant()
150
151
152     def test_lt():
153         medor = Animal('Medor', 600)
154         kiki = Animal('Kiki', 20)
155         assert kiki < medor
156         with pytest.raises(AttributeError):
157             medor < 1
158
159
160     def test_mange():

```

(suite sur la page suivante)

(suite de la page précédente)

```

161 medor = Animal('Medor', 600)
162 kiki = Animal('Kiki', 20)
163 medor(kiki)           # Médor mange Kiki
164 assert medor.estVivant()
165 assert not kiki.estVivant()
166 assert kiki.masse == 0
167 assert medor.masse == 620
168 kiki = Animal("Kiki Jr.", 20)
169 kiki(medor)           # Kiki Jr. mange Médor
170 assert not medor.estVivant()
171 assert kiki.estVivant()
172 assert kiki.masse == 22
173 assert medor.masse == 618  # Médor a perdu du poids en se faisant manger!
174
175
176 def test_init_chien():
177     medor = Chien('Medor', 600)
178     assert isinstance(medor, Animal)
179     assert isinstance(medor, Chien)
180     assert medor.odorat == 0.5
181     assert str(medor) == 'Medor (Chien, 600.0 kg)'
182     assert medor.estVivant()

```

```

===== test session starts =====
platform linux2 -- Python 2.7.6 -- py-1.4.24 -- pytest-2.6.2
collected 8 items

animauxSol.py .....

===== 8 passed in 0.04 seconds =====

```

Source : animauxSol.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-19 10:34 ycopin@lyonovae03.in2p3.fr>
4
5
6  import pytest                # pytest importé pour les tests unitaires
7  import math
8
9  """
10 Définition d'une classe point matériel, avec sa masse, sa position et sa
11 vitesse, et des méthodes pour le déplacer. Le main test applique cela à un
12 problème à force centrale gravitationnel ou électrostatique.
13
14 Remarque : Toutes les unités ont été choisies adimensionnées.
15 """
16
17 __author__ = "Adrien Licari <adrien.licari@ens-lyon.fr>"
18
19
20 # Un critère pour déterminer l'égalité entre réels
21 tolerance = 1e-8
22
23
24 #####
25 ### Définition de la classe Vector, utile pour la position et la vitesse. ###
26 #####
27
28 class Vector:
29     """
30     Une classe-structure simple contenant 3 coordonnées.
31     Une méthode est disponible pour en calculer la norme et
32     une surcharge des opérateurs ==, !=, +, - et * est proposée.
33     """
34
35     def __init__(self, x=0, y=0, z=0):
36         """
37         Constructeur de la classe vector.
38         Par défaut, construit le vecteur nul.

```

(suite sur la page suivante)

```

39
40     Args:
41         x,y,z(float): Les composantes du vecteur à construire.
42
43     Raises:
44         TypeError en cas de composantes non réelles
45     """
46     try:
47         self.x = float(x)
48         self.y = float(y)
49         self.z = float(z)
50     except (ValueError, TypeError, AttributeError):
51         raise TypeError("The given coordinates must be numbers")
52
53     def __str__(self):
54         """
55         Surcharge de l'opérateur `str`.
56
57         Returns :
58             "(x,y,z)" avec 2 décimales
59         """
60         return "({:.2f},{:.2f},{:.2f})".format(self.x, self.y, self.z)
61
62     def __eq__(self, other):
63         """
64         Surcharge de l'opérateur `==` pour tester l'égalité
65         entre deux vecteurs.
66
67         Args :
68             other(Vector): Un autre vecteur
69
70         Raises :
71             TypeError si other n'est pas un objet Vector
72         """
73         try:
74             return abs(self.x - other.x) < tolerance and \
75                 abs(self.y - other.y) < tolerance and \
76                 abs(self.z - other.z) < tolerance
77         except (ValueError, TypeError, AttributeError):
78             raise TypeError("Tried to compare Vector and non-Vector objects")
79
80     def __ne__(self, other):
81         """
82         Surcharge de l'opérateur `!=` pour tester l'inégalité
83         entre deux vecteurs.
84
85         Args :
86             other(Vector): Un autre vecteur
87
88         Raises :
89             TypeError si other n'est pas un objet Vector
90         """
91         return not self == other
92
93     def __add__(self, other):
94         """
95         Surcharge de l'opérateur `+` pour les vecteurs.
96
97         Args :
98             other(Vector): Un autre vecteur
99

```


(suite de la page précédente)

```

100     Raises :
101         TypeError si other n'est pas un objet Vector
102     """
103     try:
104         return Vector(self.x + other.x, self.y + other.y, self.z + other.z)
105     except (ValueError, TypeError, AttributeError):
106         raise TypeError("Tried to add Vector and non-Vector objects")
107
108     def __sub__(self, other):
109         """
110         Surcharge de l'opérateur '-' pour les vecteurs.
111
112         Args :
113             other(Vector): Un autre vecteur
114
115         Raises :
116             TypeError si other n'est pas un objet Vector
117         """
118         try:
119             return Vector(self.x - other.x, self.y - other.y, self.z - other.z)
120         except (ValueError, TypeError, AttributeError):
121             raise TypeError("Tried to subtract Vector and non-Vector objects")
122
123     def __mul__(self, number):
124         """
125         Surcharge de l'opérateur '*' pour la multiplication entre
126         un vecteur et un nombre.
127
128         Args :
129             number(float): Un nombre à multiplier par le Vector.
130
131         Raises :
132             TypeError si other n'est pas un nombre
133         """
134         try:
135             return Vector(number * self.x, number * self.y, number * self.z)
136         except (ValueError, TypeError, AttributeError):
137             raise TypeError("Tried to multiply Vector and non-number objects")
138
139     __rmul__ = __mul__ # Ligne pour autoriser la multiplication à droite
140
141     def norm(self):
142         """
143         Calcul de la norme 2 d'un vecteur.
144
145         Returns :
146             sqrt(x**2 + y**2 + z**2)
147         """
148         return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** (1 / 2)
149
150     def clone(self):
151         """
152         Méthode pour construire un nouveau Vecteur, copie de self.
153         """
154         return Vector(self.x, self.y, self.z)
155
156
157     #####
158     ##### Quelques test pour la classe Vector #####
159     #####
160

```

(suite sur la page suivante)

(suite de la page précédente)

```

161 def test_VectorInit():
162     with pytest.raises(TypeError):
163         vec = Vector('Test', 'avec', 'strings')
164         vec = Vector(Vector())
165         vec = Vector([])
166     vec = Vector(0, -53.76, math.pi)
167     assert vec.x == 0
168     assert vec.y == -53.76
169     assert vec.z == math.pi
170
171
172 def test_VectorStr():
173     vec = Vector(0, 600, -2)
174     assert str(vec) == '(0.00,600.00,-2.00)'
175
176
177 def test_VectorEq(): # teste aussi l'opérateur !=
178     vec = Vector(2, 3, -5)
179     vec2 = Vector(2, 3, -4)
180     assert vec != vec2
181     assert vec != Vector(0, 3, -5)
182     with pytest.raises(TypeError):
183         Vector(2, 1, 4) == "Testing strings"
184         Vector(2, 1, 4) == 42
185         Vector(2, 1, 4) == ['list']
186
187
188 def test_VectorAdd():
189     vec = Vector(2, 3, -5)
190     vec2 = Vector(2, -50, 5)
191     assert (vec + vec2) == Vector(4, -47, 0)
192
193
194 def test_VectorSub():
195     vec = Vector(1, -7, 9)
196     vec2 = Vector()
197     assert (vec - vec) == Vector()
198     assert (vec - vec2) == vec
199
200
201 def test_VectorMul():
202     vec = Vector(1, -7, 9) * 2
203     vec2 = 6 * Vector(1, -1, 2)
204     assert vec == Vector(2, -14, 18)
205     assert vec2 == Vector(6, -6, 12)
206
207
208 def test_VectorNorm():
209     assert Vector().norm() == 0
210     assert Vector(1, 0, 0).norm() == 1
211     assert Vector(2, -5, -4).norm() == 45 ** (1 / 2)
212
213
214 def test_VectorClone():
215     vec = Vector(3, 2, 9)
216     vec2 = vec.clone()
217     assert vec == vec2
218     vec2.x = 1
219     assert vec != vec2
220
221

```

(suite sur la page suivante)

(suite de la page précédente)

```

222 #####
223 ##### Une classe point matériel qui se gère en interne #####
224 #####
225
226 class Particle:
227
228     """
229     La classe Particle représente un point matériel doté d'une masse,
230     d'une position et d'une vitesse. Elle possède également une méthode
231     pour calculer la force gravitationnelle exercée par une autre particule.
232     Enfin, la méthode update lui permet de mettre à jour sa position et
233     sa vitesse en fonction des forces subies.
234     """
235
236     def __init__(self, mass=1, position=Vector(), speed=Vector()):
237         """
238         Le constructeur de la classe Particle.
239         Définit un point matériel avec une position et une vitesse initiales.
240
241         Args :
242             mass(float): La masse de la particule (doit être
243                         strictement positive)
244             position(Vector): La position initiale de la particule
245             speed(Vector): La vitesse initiale de la particule
246
247         Raises :
248             TypeError si la masse n'est pas un nombre, ou si la position ou
249                 la vitesse ne sont pas des Vector
250             ValueError si la masse est négative ou nulle
251         """
252         try:
253             self.mass = float(mass)
254             self.position = position.clone()
255             self.speed = speed.clone()
256         except (ValueError, TypeError, AttributeError):
257             raise TypeError("The mass must be a positive float number. "
258                             "The position and speed must Vector objects.")
259         try:
260             assert mass > 0 # une masse négative ou nulle pose des problèmes
261         except AssertionError:
262             raise ValueError("The mass must be strictly positive")
263         self.force = Vector()
264
265     def __str__(self):
266         """
267         Surcharge de l'opérateur `str`.
268
269         Returns :
270             "Particle with mass m, position (x,y,z) and speed (vx,vy,vz)"
271             with 2 decimals
272         """
273         return "Particle with mass {:.2f}, position {} " \
274             "and speed {}".format(self.mass, self.position, self.speed)
275
276     def computeForce(self, other):
277         """
278         Calcule la force gravitationnelle exercée par une Particule
279         other sur self.
280
281         Args :
282             other(Particle): Une autre particule, source de l'interaction

```

(suite sur la page suivante)

```

283
284     Raises :
285         TypeError si other n'est pas un objet Vector
286     """
287     try:
288         r = self.position - other.position
289         self.force = -self.mass * other.mass / r.norm() ** 3 * r
290     except AttributeError:
291         raise TypeError("Tried to compute the force created by "
292                        "a non-Particle object")
293
294     def update(self, dt):
295         """
296         Mise à jour de la position et la vitesse au cours du temps.
297
298         Args :
299             dt(float): Pas de temps d'intégration.
300         """
301         try:
302             d = float(dt)
303         except (ValueError, TypeError, AttributeError):
304             raise TypeError("The integration timestep must be a number")
305         self.speed += self.force * dt * (1 / self.mass)
306         self.position += self.speed * dt
307
308 #####
309 ##### Des tests pour la classe Particle #####
310 #####
311 #####
312
313 def test_ParticleInit():
314     with pytest.raises(TypeError):
315         p = Particle("blabla")
316         p = Particle(2, position='hum') # on vérifie les erreurs sur Vector
317         p = Particle([])
318         p = Particle(3, Vector(2, 1, 4), Vector(-1, -1, -1))
319         assert p.mass == 3
320         assert p.position == Vector(2, 1, 4)
321         assert p.speed == Vector(-1, -1, -1)
322         assert p.force == Vector()
323
324
325 def test_ParticleStr():
326     p = Particle(3, Vector(1, 2, 3), Vector(-1, -2, -3))
327     assert str(p) == "Particle with mass 3.00, position (1.00,2.00,3.00) " \
328                    "and speed (-1.00,-2.00,-3.00)"
329
330
331 def test_ParticleForce():
332     p = Particle(1, Vector(1, 0, 0))
333     p2 = Particle()
334     p.computeForce(p2)
335     assert p.force == Vector(-1, 0, 0)
336     p.position = Vector(2, -3, 6)
337     p.mass = 49
338     p.computeForce(p2)
339     assert p.force == Vector(-2 / 7, 3 / 7, -6 / 7)
340
341
342 def test_ParticleUpdate():
343     dt = 0.1

```

(suite de la page précédente)

```

344 p = Particle(1, Vector(1, 0, 0), Vector())
345 p.computeForce(Particle())
346 p.update(dt)
347 assert p.speed == Vector(-0.1, 0, 0)
348 assert p.position == Vector(0.99, 0, 0)
349
350
351 #####
352 ##### Une classe Ion qui hérite de point matériel #####
353 #####
354
355 class Ion(Particle):
356     """
357     Un Ion est une particule ayant une charge en plus de sa masse et
358     interagissant électrostatiquement plutôt que gravitationnellement.
359     La méthode computeForce remplace donc le calcul de la force
360     gravitationnelle de Newton par celui de la force électrostatique de
361     Coulomb.
362     """
363
364     def __init__(self, mass=1, charge=1, position=Vector(), speed=Vector()):
365         """
366         Le constructeur de la classe Ion.
367
368         Args :
369             mass(float): La masse de l'ion (doit être strictement positive)
370             charge(float): La charge de l'ion (doit être entière et
371                          strictement positive)
372             position(Vector): La position initiale de la particule
373             speed(Vector): La vitesse initiale de la particule
374
375         Raises :
376             ValueError si charge < 0
377             TypeError si la masse n'est pas un réel,
378                     si la charge n'est pas un entier,
379                     si position ou speed ne sont pas des Vector
380         """
381         Particle.__init__(self, mass, position, speed)
382         try:
383             self.charge = int(charge)
384         except (ValueError, AttributeError, TypeError):
385             raise TypeError("The charge must be an integer.")
386         try:
387             assert self.charge > 0
388         except AssertionError:
389             raise ValueError("The charge must be positive.")
390
391     def __str__(self):
392         """
393         Surcharge de l'opérateur `str`.
394
395         Returns :
396             "Ion with mass m, charge q, position (x,y,z)
397             and speed (vx,vy,vz)" avec q entier et le reste à 2 décimales
398         """
399         return "Ion with mass {:.2f}, charge {:d}, position {} " \
400                "and speed {}".format(self.mass, self.charge,
401                                       self.position, self.speed)
402
403     def computeForce(self, other):
404         """

```

(suite sur la page suivante)

(suite de la page précédente)

```

405     Calcule la force électrostatique de Coulomb exercée par un Ion other
406     sur self. Masque la méthode de Particle.
407
408     Args :
409         other(Ion): Un autre Ion, source de l'interaction.
410     Raises :
411         TypeError si other n'est pas un objet Ion
412     """
413     try:
414         r = self.position - other.position
415         self.force = self.charge * other.charge / r.norm() ** 3 * r
416     except (AttributeError, TypeError, ValueError):
417         raise TypeError("Tried to compute the force created by "
418                        "a non-Ion object")
419
420
421 #####
422 ##### Des test pour la classe Ion #####
423 #####
424
425 def test_IonInit():
426     with pytest.raises(TypeError):
427         ion = Ion("blabla")
428         ion = Ion(2, position='hum') # on vérifie une erreur sur Vector
429         ion = Ion(2, 'hum')          # on vérifie une erreur sur la charge
430     ion = Ion(2, 3, Vector(2, 1, 4), Vector(-1, -1, -1))
431     assert ion.mass == 2
432     assert ion.charge == 3
433     assert ion.position == Vector(2, 1, 4)
434     assert ion.speed == Vector(-1, -1, -1)
435     assert ion.force == Vector()
436
437
438 def test_IonStr():
439     ion = Ion(3, 2, Vector(1, 2, 3), Vector(-1, -2, -3))
440     assert str(ion) == "Ion with mass 3.00, charge 2, " \
441                      "position (1.00,2.00,3.00) and speed (-1.00,-2.00,-3.00)"
442
443
444 def test_IonForce():
445     ion = Ion(mass=1, charge=1, position=Vector(1, 0, 0))
446     ion2 = Ion(charge=3)
447     ion.computeForce(ion2)
448     assert ion.force == Vector(3, 0, 0)
449     ion = Ion(charge=49, position=Vector(2, -3, 6))
450     ion.computeForce(ion2)
451     assert ion.force == Vector(6 / 7, -9 / 7, 18 / 7)
452
453
454 #####
455 ##### Un main de test #####
456 #####
457
458 if __name__ == '__main__':
459
460     # On lance tous les tests en bloc pour commencer
461     print(" Test functions ".center(50, "*"))
462     print("Testing Vector class...", end=' ')
463     test_VectorInit()
464     test_VectorStr()
465     test_VectorEq()

```

(suite sur la page suivante)

(suite de la page précédente)

```

466     test_VectorAdd()
467     test_VectorSub()
468     test_VectorMul()
469     test_VectorNorm()
470     test_VectorClone()
471     print("ok")
472     print("Testing Particle class...", end=' ')
473     test_ParticleInit()
474     test_ParticleStr()
475     test_ParticleForce()
476     test_ParticleUpdate()
477     print("ok")
478     print("Testing Ion class...", end=' ')
479     test_IonInit()
480     test_IonStr()
481     test_IonForce()
482     print("ok")
483     print(" Test end ".center(50, "*"), "\n")
484
485     # Un petit calcul physique
486     print(" Physical computations ".center(50, "*"))
487     dt = 0.0001
488
489     # Problème à force centrale gravitationnelle, cas circulaire
490     ntimesteps = int(10000 * math.pi) # durée pour parcourir pi
491     center = Particle()
492     M = Particle(mass=1, position=Vector(1, 0, 0), speed=Vector(0, 1, 0))
493     print("** Gravitationnal computation of central-force motion for a {}" \
494           .format(str(M)))
495     for i in range(ntimesteps):
496         M.computeForce(center)
497         M.update(dt)
498     print("\t => Final system : {}".format(str(M)))
499
500     # problème à force centrale électrostatique, cas rectiligne
501     center = Ion()
502     M = Ion(charge=4, position=Vector(0, 0, 1), speed=Vector(0, 0, -1))
503     print("** Electrostatic computation of central-force motion for a {}" \
504           .format(str(M)))
505     for i in range(ntimesteps):
506         M.computeForce(center)
507         M.update(dt)
508     print("\t => Final system : {}".format(str(M)))
509
510     print(" Physical computations end ".center(50, "*"))

```

Source : particleSol.py

CHAPITRE 24

Jeu de la vie

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-26 16:54 ycopin@lyonovae03.in2p3.fr>
4
5  """
6  Jeu de la vie (programmation orientée objet).
7  """
8
9  import random
10
11
12 class Life:
13
14     cells = {False: ".", True: "#"} # Dead and living cell representations
15
16     def __init__(self, h, w, periodic=False):
17         """
18         Create a 2D-list (the game grid *G*) with the wanted size (*h*
19         rows, *w* columns) and initialize it with random booleans
20         (dead/alive). The world is periodic if *periodic* is True.
21         """
22
23         self.h = int(h)
24         self.w = int(w)
25         assert self.h > 0 and self.w > 0
26         # Random initialization of a h*w world
27         self.world = [[random.choice([True, False])
28                        for j in range(self.w)]
29                       for i in range(self.h)] # h rows of w elements
30         self.periodic = periodic
31
32     def get(self, i, j):
33         """
34         This method returns the state of cell (*i*,*j*) safely, even
35         if the (*i*,*j*) is outside the grid.
36         """
37
38         if self.periodic:
```

(suite sur la page suivante)

(suite de la page précédente)

```

39         return self.world[i % self.h][j % self.w] # Periodic conditions
40     else:
41         if (0 <= i < self.h) and (0 <= j < self.w): # Inside grid
42             return self.world[i][j]
43         else: # Outside grid
44             return False # There's nobody out there...
45
46     def __str__(self):
47         """
48         Convert the grid to a visually handy string.
49         """
50
51         return '\n'.join([''.join([self.cells[val] for val in row])
52                             for row in self.world])
53
54     def evolve_cell(self, i, j):
55         """
56         Tells if cell (*i*,*j*) will survive during game iteration,
57         depending on the number of living neighboring cells.
58         """
59
60         alive = self.get(i, j) # Current cell status
61         # Count living cells *around* current one (excluding current one)
62         count = sum(self.get(i + ii, j + jj)
63                     for ii in [-1, 0, 1]
64                     for jj in [-1, 0, 1]
65                     if (ii, jj) != (0, 0))
66
67         if count == 3:
68             # A cell w/ 3 neighbors will either stay alive or resuscitate
69             future = True
70         elif count < 2 or count > 3:
71             # A cell w/ too few or too many neighbors will die
72             future = False
73         else:
74             # A cell w/ 2 or 3 neighbors will stay as it is (dead or alive)
75             future = alive # Current status
76
77         return future
78
79     def evolve(self):
80         """
81         Evolve the game grid by one step.
82         """
83
84         # Update the grid
85         self.world = [[self.evolve_cell(i, j)
86                         for j in range(self.w)]
87                        for i in range(self.h)]
88
89 if __name__ == "__main__":
90
91     import time
92
93     h, w = (20, 60) # (nrows,ncolumns)
94
95     # Instantiation (including initialization)
96     life = Life(h, w, periodic=True)
97
98     generation = 0
99     while True: # Infinite loop! (Ctrl-C to break)

```

(suite sur la page suivante)

(suite de la page précédente)

```
100     print(generation)
101     print(life)           # Print current world
102     generation += 1
103     time.sleep(0.1)       # Pause a bit
104     life.evolve()         # Evolve world
```

Source : life.py

Median Absolute Deviation

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import numpy as N
5
6
7  def mad(a, axis=None):
8      """
9      Compute *Median Absolute Deviation* of an array along given axis.
10     """
11
12     # Median along given axis, but *keeping* the reduced axis so that
13     # result can still broadcast against a.
14     med = N.median(a, axis=axis, keepdims=True)
15     mad = N.median(N.absolute(a - med), axis=axis) # MAD along given axis
16
17     return mad
18
19 if __name__ == '__main__':
20
21     x = N.arange(4 * 5, dtype=float).reshape(4, 5)
22
23     print("x =\n", x)
24     print("MAD(x, axis=None) =", mad(x))
25     print("MAD(x, axis=0) =", mad(x, axis=0))
26     print("MAD(x, axis=1) =", mad(x, axis=1))
27     print("MAD(x, axis=(0, 1)) =", mad(x, axis=(0, 1)))
```

Source : mad.py


```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import numpy as N
5
6
7  def pull(x, dx):
8      """
9      Compute the pull from x, dx.
10
11      * Input data: x = [x_i], error dx = [s_i] Optimal
12      * (variance-weighted) mean:  $E = \sum(x_i/s_i^2)/\sum(1/s_i^2)$  Variance
13      * on weighted mean:  $\text{var}(E) = 1/\sum(1/s_i^2)$  Pull:  $p_i = (x_i -$ 
14      *  $E_i)/\sqrt{\text{var}(E_i) + s_i^2}$  where  $E_i$  and  $\text{var}(E_i)$  are computed
15      * without point i.
16
17      If errors s_i are correct, the pull distribution is centered on 0
18      with standard deviation of 1.
19      """
20
21      assert x.ndim == dx.ndim == 1, "pull works on 1D-arrays only."
22      assert len(x) == len(dx), "dx should be the same length as x."
23
24      n = len(x)
25
26      i = N.resize(N.arange(n), n * n) # 0,...,n-1,0,...n-1,..., n times (n^2,)
27      i[:,n + 1] = -1 # Mark successively 0,1,2,...,n-1
28      # Remove marked indices & reshape (n,n-1)
29      j = i[i >= 0].reshape((n, n - 1))
30
31      v = dx ** 2 # Variance
32      w = 1 / v # Variance (optimal) weighting
33
34      Ei = N.average(x[j], weights=w[j], axis=1) # Weighted mean (n,)
35      vEi = 1 / N.sum(w[j], axis=1) # Variance of mean (n,)
36
37      p = (x - Ei) / N.sqrt(vEi + v) # Pull (n,)
38

```

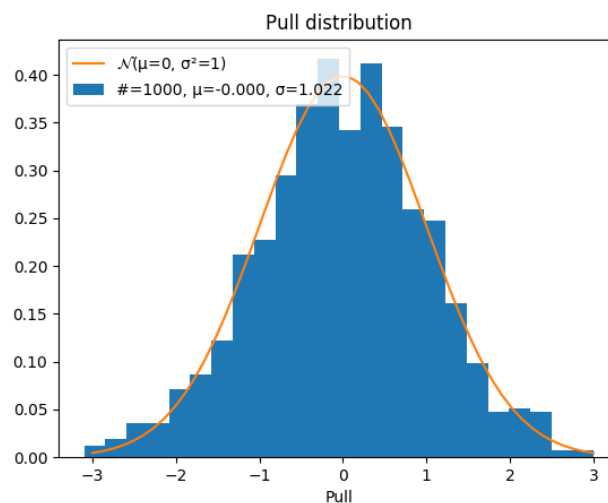
(suite sur la page suivante)

(suite de la page précédente)

```

39     return p
40
41 if __name__ == '__main__':
42
43     import matplotlib.pyplot as P
44     import scipy.stats as SS
45
46     n = 1000
47     mu = 1.
48     sig = 2.
49
50     # Normally distributed random sample of size n, with mean=mu and std=sig
51     x = N.random.normal(loc=mu, scale=sig, size=n)
52     dx = N.full_like(x, sig)          # Formal (true) errors
53
54     p = pull(x, dx)                  # Pull computation
55
56     m, s = p.mean(), p.std(ddof=1)
57     print("Pull ({} entries): mean={:.2f}, std={:.2f}".format(n, m, s))
58
59     fig, ax = P.subplots()
60     _, bins, _ = ax.hist(p, bins='auto', normed=True,
61                          histtype='stepfilled',
62                          label="#={}, p={:.3f}, sigma={:.3f}".format(n, m, s))
63     y = N.linspace(-3, 3)
64     ax.plot(y, SS.norm.pdf(y), label=r"$\mathcal{N}(\mu=0, \sigma^2=1)$")
65     ax.set(title='Pull distribution', xlabel='Pull')
66     ax.legend(loc='upper left')
67
68     P.show()

```



Source : pull.py

```

[1]: import numpy as N
import matplotlib.pyplot as P
# Insert figures within notebook
%matplotlib inline

```


Calcul de l'intégrale

$$\int_0^{\infty} \frac{x^3}{e^x - 1} dx = \frac{\pi^4}{15}$$

```
[2]: import scipy.integrate as SI
```

```
[3]: def integrand(x):  
      return x**3 / (N.exp(x) - 1)
```

```
[4]: q, dq = SI.quad(integrand, 0, N.inf)  
      print("Intégrale:", q)  
      print("Erreur estimée:", dq)  
      print("Erreur absolue:", (q - (N.pi ** 4 / 15)))
```

```
Intégrale: 6.49393940226683  
Erreur estimée: 2.628470028924825e-09  
Erreur absolue: 1.7763568394002505e-15
```

```
/home/ycopin/Softwares/VirtualEnvs/Python3/lib/python3.5/site-packages/ipykernel_launcher.  
↳ py:3: RuntimeWarning: overflow encountered in exp  
This is separate from the ipykernel package so we can avoid doing imports until
```


Zéro d'une fonction

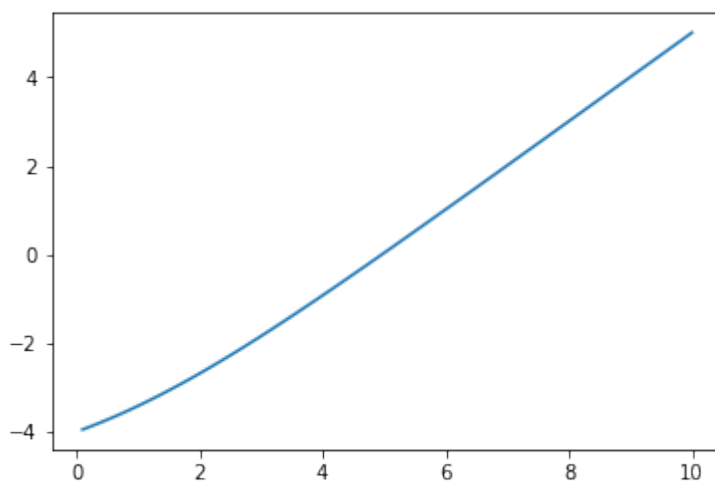
Résolution numérique de l'équation

$$f(x) = \frac{x e^x}{e^x - 1} - 5 = 0$$

```
[5]: def func(x):  
      return x * N.exp(x) / (N.exp(x) - 1) - 5
```

Il faut d'abord déterminer un intervalle contenant la solution, c.-à-d. le zéro de `func`. Puisque $f(0^+) = -4 < 0$ et $f(10) \simeq 5 > 0$, il est intéressant de tracer l'allure de la courbe sur ce domaine :

```
[6]: x = N.logspace(-1, 1) # 50 points logarithmiquement espacé de 10**-1 = 0.1 à 10**1 = 10  
      P.plot(x, func(x));
```



```
[7]: import scipy.optimize as SO
```

```
[8]: zero = SO.brentq(func, 1, 10)  
      print("Solution:", zero)
```

Solution: 4.965114231744277

The following section was generated from Exercices/numerique.ipynb

Quartet d'Anscombe

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import numpy as N
5  import scipy.stats as SS
6  import matplotlib.pyplot as P
7
8
9  def printStats(x, y):
10     """
11     Print out means and variances for x and y, as well as correlation
12     coeff. (Pearson) and linear regression for y vs. x.
13     """
14
15     assert N.shape(x) == N.shape(y), "Incompatible input arrays"
16
17     print("x: mean={:.2f}, variance={:.2f}".format(N.mean(x), N.var(x)))
18     print("y: mean={:.2f}, variance={:.2f}".format(N.mean(y), N.var(y)))
19     print("y vs. x: corrcoeff={:.2f}".format(SS.pearsonr(x, y)[0]))
20     # slope, intercept, r_value, p_value, std_err
21     a, b, _, _, _ = SS.linregress(x, y)
22     print("y vs. x: y = {:.2f} x + {:.2f}".format(a, b))
23
24
25  def plotStats(ax, x, y, title='', fancy=True):
26     """
27     Plot y vs. x, and linear regression.
28     """
29
30     assert N.shape(x) == N.shape(y), "Incompatible input arrays"
31
32     # slope, intercept, r_value, p_value, std_err
33     a, b, r, _, _ = SS.linregress(x, y)
34
35     # Data + corrcoeff
36     ax.plot(x, y, 'bo', label="r = {:.2f}".format(r))
37
38     # Linear regression
```

(suite sur la page suivante)

(suite de la page précédente)

```

39 xx = N.array([0, 20])
40 yy = a * xx + b
41 ax.plot(xx, yy, 'r-', label="y = {:.2f} x + {:.2f}".format(a, b))
42
43 leg = ax.legend(loc='upper left', fontsize='small')
44
45 if fancy:          # Additional stuff
46     # Add mean line ± stddev
47     m = N.mean(x)
48     s = N.std(x, ddof=1)
49     ax.axvline(m, color='g', ls='--', label='_')    # Mean
50     ax.axvspan(m - s, m + s, color='g', alpha=0.2) # Std-dev
51
52     m = N.mean(y)
53     s = N.std(y, ddof=1)
54     ax.axhline(m, color='g', ls='--', label='_')    # Mean
55     ax.axhspan(m - s, m + s, color='g', alpha=0.2) # Std-dev
56
57     # Title and labels
58     if title:
59         ax.set_title(title)
60     if ax.is_last_row():
61         ax.set_xlabel("x")
62     if ax.is_first_col():
63         ax.set_ylabel("y")
64
65
66 if __name__ == '__main__':
67
68     quartet = N.genfromtxt("anscombe.dat") # Read Anscombe's Quartet
69
70     fig = P.figure()
71
72     for i in range(4):          # Loop over quartet sets x,y
73         ax = fig.add_subplot(2, 2, i + 1)
74         print(" Dataset #{} ".format(i + 1).center(40, '='))
75         x, y = quartet[:, 2 * i:2 * i + 2].T
76         printStats(x, y)        # Print main statistics
77         plotStats(ax, x, y, title='#'+str(i + 1)) # Plots
78
79     fig.suptitle("Anscombe's Quartet", fontsize='x-large')
80     fig.tight_layout()
81
82     P.show()

```

```

$ python3 anscombe.py

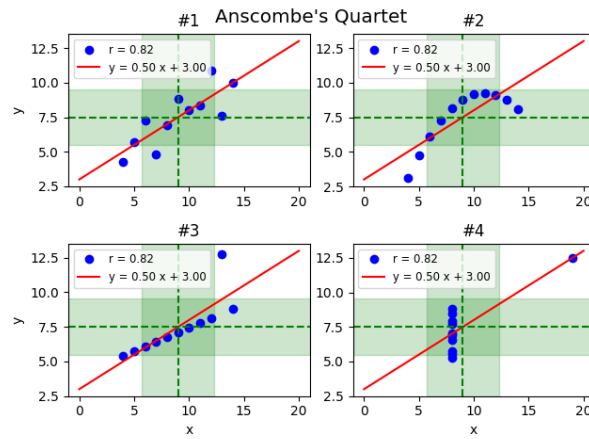
===== Dataset #1 =====
x: mean=9.00, variance=10.00
y: mean=7.50, variance=3.75
y vs. x: corrcoeff=0.82
y vs. x: y = 0.50 x + 3.00
===== Dataset #2 =====
x: mean=9.00, variance=10.00
y: mean=7.50, variance=3.75
y vs. x: corrcoeff=0.82
y vs. x: y = 0.50 x + 3.00
===== Dataset #3 =====
x: mean=9.00, variance=10.00
y: mean=7.50, variance=3.75
y vs. x: corrcoeff=0.82

```

(suite sur la page suivante)

(suite de la page précédente)

```
y vs. x: y = 0.50 x + 3.00
===== Dataset #4 =====
x: mean=9.00, variance=10.00
y: mean=7.50, variance=3.75
y vs. x: corrcoeff=0.82
y vs. x: y = 0.50 x + 3.00
```

Source : `anscombe.py`


```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-19 10:42 ycopin@lyonovae03.in2p3.fr>
4
5  import numpy as np
6  import random
7  import matplotlib.pyplot as plt
8
9
10 def iteration(r, niter=100):
11
12     x = random.uniform(0, 1)
13     i = 0
14     while i < niter and x < 1:
15         x = r * x * (1 - x)
16         i += 1
17
18     return x if x < 1 else -1
19
20
21 def generate_diagram(r, ntrials=50):
22     """
23     Cette fonction retourne (jusqu'à) *ntrials* valeurs d'équilibre
24     pour les *r* d'entrée. Elle renvoie un tuple:
25
26     + le premier élément est la liste des valeurs prises par le paramètre *r*
27     + le second est la liste des points d'équilibre correspondants
28     """
29
30     r_v = []
31     x_v = []
32     for rr in r:
33         j = 0
34         while j < ntrials:
35             xx = iteration(rr)
36             if xx > 0: # Convergence: il s'agit d'une valeur d'équilibre
37                 r_v.append(rr)
38                 x_v.append(xx)
```

(suite sur la page suivante)

(suite de la page précédente)

```
39         j += 1                                # Nouvel essai
40
41     return r_v, x_v
42
43 r = np.linspace(0, 4, 1000)
44 x, y = generate_diagram(r)
45
46 plt.plot(x, y, 'r,')
47 plt.xlabel('r')
48 plt.ylabel('x')
49 plt.show()
```

Source : logistique.py

CHAPITRE 31

Ensemble de Julia

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-19 10:43 ycopin@lyonovae03.in2p3.fr>
4
5  """
6  Visualisation de l'ensemble de julia
7  <http://fr.wikipedia.org/wiki/Ensemble_de_Julia>`_.
8
9  Exercice: proposer des solutions pour accélérer le calcul.
10 """
11
12 import numpy as np
13 import matplotlib.pyplot as plt
14
15 c = complex(0.284, 0.0122)          # Constante
16
17 xlim = 1.5                          # [-xlim,xlim] × i[-xlim,xlim]
18 nx = 1000                          # Nb de pixels
19 niter = 100                         # Nb d'itérations
20
21 x = np.linspace(-xlim, xlim, nx)    # nx valeurs de -xlim à +xlim
22 xx, yy = np.meshgrid(x, x)         # Tableaux 2D
23 z = xx + 1j * yy                   # Portion du plan complexe
24 for i in range(niter):              # Itération: z(n+1) = z(n)**2 + c
25     z = z ** 2 + c
26
27 # Visualisation
28 plt.imshow(np.abs(z), extent=[-xlim, xlim, -xlim, xlim], aspect='equal')
29 plt.title(c)
30 plt.show()
```

Source : julia.py

Trajectoire d'un boulet de canon

Nous allons intégrer les équations du mouvement pour un boulet de canon soumis à des forces de frottement « turbulentes » (non-linéaires) :

$$\ddot{\mathbf{r}} = \mathbf{g} - \frac{\alpha}{m} v \times \mathbf{v}.$$

Cette équation différentielle non linéaire du 2d ordre doit être réécrite sous la forme de deux équations différentielles couplées du 1er ordre :

$$\begin{cases} \dot{\mathbf{r}} &= \mathbf{v} \\ \dot{\mathbf{v}} &= \mathbf{g} - \frac{\alpha}{m} v \times \mathbf{v}. \end{cases}$$

Il s'agit donc de résoudre *une seule* équation différentielle du 1er ordre en $\mathbf{z} = (\mathbf{r}, \mathbf{v})$.

```
[1]: %matplotlib inline

import numpy as N
import scipy.integrate as SI
import matplotlib.pyplot as P
```

Valeurs numériques pour un boulet de canon de 36 livres :

```
[2]: g = 9.81          # Pesanteur [m/s2]
cx = 0.45             # Coefficient de frottement d'une sphère
rhoAir = 1.2          # Masse volumique de l'air [kg/m3] au niveau de la mer, T=20°C
rad = 0.1748/2        # Rayon du boulet [m]
rho = 6.23e3          # Masse volumique du boulet [kg/m3]
mass = 4./3.*N.pi*rad**3 * rho          # Masse du boulet [kg]
alpha = 0.5*cx*rhoAir*N.pi*rad**2 / mass # Coefficient de frottement par unité de masse
print("Masse du boulet: {:.2f} kg".format(mass))
print("Coefficient de frottement par unité de masse: {} S.I.".format(alpha))
```

Masse du boulet: 17.42 kg

Coefficient de frottement par unité de masse: 0.0003718994604243878 S.I.

Conditions initiales :

```
[3]: v0 = 450.         # Vitesse initiale [m/s]
alt = 45.              # Inclinaison du canon [deg]
alt *= N.pi / 180.    # Inclinaison [rad]
z0 = (0., 0., v0 * N.cos(alt), v0 * N.sin(alt)) # (x0, y0, vx0, vy0)
```

Temps caractéristique du système : $t = \sqrt{\frac{m}{g\alpha}}$ (durée du régime transitoire). L'intégration des équations se fera sur un temps caractéristique, avec des pas de temps significativement plus petits.

```
[4]: tc = N.sqrt(mass / (g * alpha))
print("Temps caractéristique: {:.1f} s".format(tc))
t = N.linspace(0, tc, 100)
```

Temps caractéristique: 69.1 s

Définition de la fonction $\dot{\mathbf{z}}$, avec $\mathbf{z} = (\mathbf{r}, \mathbf{v})$

```
[5]: def zdot(z, t):
    """Calcul de la dérivée de z=(x, y, vx, vy) à l'instant t."""

    x, y, vx, vy = z
    alphav = alpha * N.hypot(vx, vy)

    return (vx, vy, -alphav * vx, -g - alphav * vy) # dz/dt = (vx,vy,x..,y..)
```

Intégration numérique des équations du mouvement à l'aide de la fonction `scipy.integrate.odeint` :

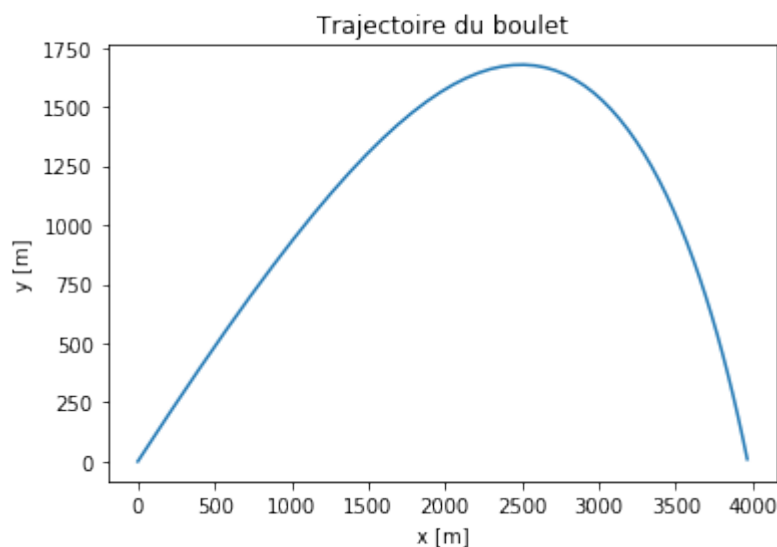
```
[6]: zs = SI.odeint(zdot, z0, t)
```

Le tableau `zs` contient les valeurs de \mathbf{z} à chaque instant t : il est donc de taille `(len(t),4)`.

```
[7]: ypos = zs[:,1]>=0 # y>0?
print("temps de coll. t(y~0) = {:.0f} s".format(t[ypos][-1])) # Dernier instant pour lequel y>0
print("portée x(y~0) = {:.0f} m".format(zs[ypos, 0][-1])) # Portée approximative du canon
#print "y(y~0) = {:.0f} m".format(zs[ypos, 1][-1]) # ~0
print("vitesse(y~0): {:.0f} m/s".format(N.hypot(zs[ypos, 2][-1], zs[ypos, 3][-1])))
```

temps de coll. t(y~0) = 36 s
portée x(y~0) = 3966 m
vitesse(y~0): 140 m/s

```
[8]: fig,ax = P.subplots()
ax.plot(zs[ypos, 0], zs[ypos, 1])
ax.set(xlabel="x [m]", ylabel="y [m]", title="Trajectoire du boulet");
```



The following section was generated from Exercices/canon.ipynb

Équation d'état de l'eau

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  # Time-stamp: <2018-07-19 10:38 ycopin@lyonovae03.in2p3.fr>
4
5
6  import numpy as N
7  import matplotlib.pyplot as P
8
9  import pytest                # pytest importe pour les tests unitaires
10
11  """
12  Construction d'un système d'extraction et d'analyse de fichiers de sortie de
13  dynamique moléculaire afin d'extraire les grandeurs thermodynamiques.
14  On affichera les ensuite isothermes.
15  """
16
17  __author__ = "Adrien Licari <adrien.licari@ens-lyon.fr>"
18
19
20  tolerance = 1e-8 # Un seuil de tolérance pour les égalités sur réels
21
22
23  #####
24  ##### A Simulation class #####
25  #####
26
27  class Simulation:
28      """
29      La classe Simulation représente une simulation de dynamique
30      moléculaire, donc un point de l'équation d'état. Son constructeur
31      doit impérativement être appelé avec le chemin du fichier output
32      correspondant. Elle possède des méthodes pour extraire les grandeurs
33      thermodynamiques et afficher la run, en pouvant enlever certains pas
34      de temps en début de simulation.
35      """
36
37      def __init__(self, temp, dens, path):
38          """

```

(suite sur la page suivante)

(suite de la page précédente)

```

39  Le constructeur doit impérativement être appelé avec le chemin du
40  fichier décrivant la simulation, ainsi que ses conditions
41  thermodynamiques.
42
43  Args :
44      temp,dens(float): La température et la densité de la simulation
45      path(string): Le chemin vers le fichier décrivant la simulation
46
47  Raises :
48      TypeError si temp ou dens ne sont pas des réels
49      IOError si le fichier n'existe pas
50
51  """
52  self.temp = float(temp)
53  self.dens = float(dens)
54  tmp = N.loadtxt(path, skiprows=1).T
55  self.pot = tmp[0]
56  self.kin = tmp[1]
57  self.tot = self.pot + self.kin
58  self.press = tmp[2]
59
60  def __str__(self):
61      """
62      Surcharge de l'opérateur str.
63      """
64      return "Simulation at {:.0f} g/cc and {:.0f} K ; {:d} timesteps". \
65             format(self.dens, self.temp, len(self.pot))
66
67  def thermo(self, skipSteps=0):
68      """
69      Calcule l'énergie et la pression moyenne au cours de la simulation.
70      Renvoie un dictionnaire.
71
72      Args:
73          skipSteps(int): Nb de pas à enlever en début de simulation.
74
75      Returns:
76          {'T':temperature, 'rho':density,
77           'E':energy, 'P':pressure,
78           'dE':dEnergy, 'dP':dPressure}
79      """
80      return {'T': self.temp,
81             'rho': self.dens,
82             'E': self.tot[skipSteps:].mean(),
83             'P': self.press[skipSteps:].mean(),
84             'dE': self.tot[skipSteps:].std(),
85             'dP': self.press[skipSteps:].std()}
86
87  def plot(self, skipSteps=0):
88      """
89      Affiche l'évolution de la Pression et l'énergie interne au cours de
90      la simulation.
91
92      Args:
93          skipSteps(int): Pas de temps à enlever en début de simulation.
94
95      Raises:
96          TypeError si skipSteps n'est pas un entier.
97      """
98      fig, (axen, axpress) = P.subplots(2, sharex=True)
99      axen.plot(list(range(skipSteps, len(self.tot))), self.tot[skipSteps:],
100               'rd--')
```

(suite sur la page suivante)

(suite de la page précédente)

```

100     axen.set_title("Internal energy (Ha)")
101     axpress.plot(list(range(skipSteps, len(self.press))), self.press[skipSteps:],
102                  'rd--')
103     axpress.set_title("Pressure (GPa)")
104     axpress.set_xlabel("Timesteps")
105
106     P.show()
107
108     ##### Tests pour Simulation #####
109
110
111     def mimic_simulation(filename):
112         with open(filename, 'w') as f:
113             f.write("""Potential energy (Ha)           Kinetic Energy (Ha)           Pressure (GPa)
114 -668.2463567264           0.7755612311           9287.7370229824
115 -668.2118514558           0.7755612311           9286.1395903265
116 -668.3119088218           0.7755612311           9247.6604398856
117 -668.4762735176           0.7755612311           9191.8574820856
118 -668.4762735176           0.7755612311           9191.8574820856
119 """)
120
121
122     def test_Simulation_init():
123         mimic_simulation("equationEtat_simuTest.out")
124         s = Simulation(10, 10, "equationEtat_simuTest.out")
125         assert len(s.kin) == 5
126         assert abs(s.kin[2] - 0.7755612311) < tolerance
127         assert abs(s.pot[1] + 668.2118514558) < tolerance
128
129
130     def test_Simulation_str():
131         mimic_simulation("equationEtat_simuTest.out")
132         s = Simulation(10, 20, "equationEtat_simuTest.out")
133         assert str(s) == "Simulation at 20 g/cc and 10 K ; 5 timesteps"
134
135
136     def test_Simulation_thermo():
137         mimic_simulation("equationEtat_simuTest.out")
138         s = Simulation(10, 20, "equationEtat_simuTest.out")
139         assert abs(s.thermo()['T'] - 10) < tolerance
140         assert abs(s.thermo()['rho'] - 20) < tolerance
141         assert abs(s.thermo()['E'] + 667.56897157674) < tolerance
142         assert abs(s.thermo()['P'] - 9241.0504034731) < tolerance
143         assert abs(s.thermo(3)['E'] + 667.7007122865) < tolerance
144         assert abs(s.thermo(3)['P'] - 9191.8574820856) < tolerance
145
146     #####
147     ### Main script ###
148     #####
149
150     if __name__ == '__main__':
151         """
152         On définit un certain nombre de pas de temps à sauter, puis on
153         charge chaque simulation et extrait les informations thermodynamiques
154         associées. On affiche enfin les isothermes normalisées (E/NkT et P/nkT).
155         """
156
157         ### Definitions ###
158         a0 = 0.52918          # Bohr radius in angstrom
159         amu = 1.6605          # atomic mass unit in e-24 g
160         k_B = 3.16681e-6      # Boltzmann's constant in Ha/K

```

(suite sur la page suivante)

```

161  # normalization factor for P/nkT
162  nk_GPa = a0 ** 3 * k_B * 2.942e4 / 6 / amu
163  nsteps = 200 # define skipped timesteps (should be done for
164  # each simulation...)
165  temps = [6000, 20000, 50000] # define temperatures
166  colors = {6000: 'r', 20000: 'b', 50000: 'k'}
167  denss = [7, 15, 25, 30] # define densities
168  keys = ['T', 'rho', 'E', 'dE', 'P', 'dP']
169  eos = dict.fromkeys(keys, N.zeros(0)) # {key:[]}
170
171  ### Extract the EOS out of the source files ###
172  for t, rho in [(t, rho) for t in temps for rho in denss]:
173      filenm = "outputs/{0}K_{1:0>2d}gcc.out".format(t, rho)
174      s = Simulation(t, rho, filenm)
175      for key in keys:
176          eos[key] = N.append(eos[key], s.thermo(nsteps)[key])
177
178  ### Plot isotherms ###
179  fig, (axen, axpress) = P.subplots(2, sharex=True)
180  fig.suptitle("High-pressure equation of state for water", size='x-large')
181  axen.set_title("Energy")
182  axen.set_ylabel("U / NkT")
183  axpress.set_title("Pressure")
184  axpress.set_ylabel("P / nkT")
185  axpress.set_xlabel("rho (g/cc)")
186  for t in temps:
187      sel = eos['T'] == t
188      axen.errorbar(x=eos['rho'][sel], y=eos['E'][sel] / k_B / t,
189                  yerr=eos['dE'][sel] / k_B / t, fmt=colors[t] + '-')
190      axpress.errorbar(x=eos['rho'][sel],
191                     y=eos['P'][sel] / eos['rho'][sel] / nk_GPa / t,
192                     yerr=eos['dP'][sel] / eos['rho'][sel] / nk_GPa / t,
193                     fmt=colors[t] + '-',
194                     label="{0} K".format(t))
195  axpress.legend(loc='best')
196  P.show()

```

Source : equationEtatSol.py

Solutions aux exercices

- *Méthode des rectangles*
- *Fizz Buzz*
- *Algorithme d'Euclide*
- *Crible d'Ératosthène*
- *Carré magique*
- *Suite de Syracuse*
- *Flocon de Koch*
- *Jeu du plus ou moins*
- *Animaux*
- *Particules*
- *Jeu de la vie*
- *Median Absolute Deviation*
- *Distribution du pull*
- *Calculs numériques* (`numerique.ipynb`)
- *Quartet d'Anscombe*
- *Suite logistique*
- *Ensemble de Julia*
- *Trajectoire d'un boulet de canon* (`canon.ipynb`)
- *Équation d'état de l'eau*

Consignes :

- Vous avez accès à tout l'internet « statique » (hors mail, tchat, forum, etc.), y compris donc au cours en ligne.
- Ne soumettez pas de codes non-fonctionnels (i.e. provoquant une exception à l'interprétation, avant même l'exécution) : les erreurs de syntaxe seront lourdement sanctionnées.
- Respectez scrupuleusement les directives de l'énoncé (nom des variables, des méthodes, des fichiers, etc.), en particulier concernant le nom des fichiers à renvoyer aux correcteurs.

35.1 Exercice

Un appareil de vélocimétrie a mesuré une vitesse à intervalle de temps régulier puis à sorti le fichier texte `velocimetrie.dat` (attention à l'entête). Vous écrirez un script python « `exo_nom_prénom.py` » (sans accent) utilisant `matplotlib` qui générera, affichera et sauvegardera sous le nom « `exo_nom_prénom.pdf` » une figure composée de trois sous-figures, l'une au dessus de l'autre :

1. la vitesse en mm/s mesurée en fonction du temps,
2. le déplacement en mètres en fonction du temps. On utilisera volontairement une intégration naïve à partir de zéro via la fonction `numpy.cumsum()`,
3. l'accélération en m/s² en fonction du temps. On utilisera volontairement une dérivation naïve à deux points :

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

via la fonction `numpy.diff()`. Attention, si l'entrée de cette fonction est un tableau de taille N , sa sortie est un tableau de taille $N-1$.

Le script doit lire le fichier `velocimetrie.dat` stocké dans le répertoire courant. On prendra soin des noms des axes et des unités physiques. Si les trois axes des abscisses sont identiques, seul celui de la troisième sous-figure peut être nommé.

35.2 Le problème du voyageur de commerce

35.2.1 Introduction

Le problème du voyageur de commerce est un problème d'optimisation consistant à déterminer le plus court chemin reliant un ensemble de destinations. Il n'existe pas d'algorithme donnant la solution optimale en un temps raisonnable (problème NP-complet), mais l'on peut chercher à déterminer des solutions approchées.

On va se placer ici dans le cas d'un livreur devant desservir une seule fois chacune des n destinations d'une ville américaine où les rues sont agencées en réseau carré (Figure). On utilise la « distance de Manhattan » (norme L1) entre deux points $A(x_A, y_A)$ et $B(x_B, y_B)$:

$$d(A, B) = |x_B - x_A| + |y_B - y_A|.$$

En outre, on se place dans le cas où les coordonnées des destinations sont *entières*, comprises entre 0 (inclus) et `TAILLE = 50` (exclus). Deux destinations peuvent éventuellement avoir les mêmes coordonnées.

Les instructions suivantes doivent permettre de définir les classes nécessaires (`Ville` et `Trajet`) et de développer deux algorithmes approchés (heuristiques) : l'algorithme du plus proche voisin, et l'optimisation 2-opt. Seules la librairie standard et la librairie `numpy` sont utilisables si nécessaire.

Un squelette du code, définissant l'interface de programmation et incluant des tests unitaires (à utiliser avec `py.test`), vous est fourni : `exam_1501.py`. Après l'avoir renommé « `pb_nom_prénom.py` » (sans accent), l'objectif est donc de compléter ce code progressivement, en suivant les instructions suivantes.

Une ville-test de 20 destinations est fournie : `ville.dat` (Fig.), sur laquelle des tests de lecture et d'optimisation seront réalisés.

35.2.2 Classe Ville

Les n coordonnées des destinations sont stockées dans l'attribut `destinations`, un tableau `numpy` d'entiers de format `(n, 2)`.

1. `__init__()` : initialisation d'une ville sans destination (déjà implémenté, ne pas modifier).
2. `aleatoire(n)` : création de n destinations aléatoires (utiliser `numpy.random.randint()`).
3. `lecture(nomfichier)` : lecture d'un fichier ASCII donnant les coordonnées des destinations.
4. `ecriture(nomfichier)` : écriture d'un fichier ASCII avec les coordonnées des destinations.
5. `nb_trajet()` : retourne le nombre total (entier) de trajets : $(n-1)!/2$ (utiliser `math.factorial()`).
6. `distance(i, j)` : retourne la distance (Manhattan-L1) entre les deux destinations de numéro i et j .

35.2.3 Classe Trajet

L'ordre des destinations suivi au cours du trajet est stocké dans l'attribut `etapes`, un tableau `numpy` d'entiers de format `(n,)`.

1. `__init__(ville, etapes=None)` : initialisation sur une `ville`. Si la liste `etapes` n'est pas spécifiée, le trajet par défaut est celui suivant les destinations de `ville`.
2. `longueur()` : retourne la longueur totale du trajet *bouclé* (i.e. revenant à son point de départ).

35.2.4 Heuristique *Plus proche voisin*

1. `Ville.plus_proche(i, exclus=[])` : retourne la destination la plus proche de la destination i (au sens de `Ville.distance()`), *hors* les destinations de la liste `exclus`.
2. `Ville.trajet_voisins(depart=0)` : retourne un `Trajet` déterminé selon l'heuristique des plus proches voisins (i.e. l'étape suivante est la destination la plus proche hors les destinations déjà visitées) en partant de l'étape initiale `depart`.

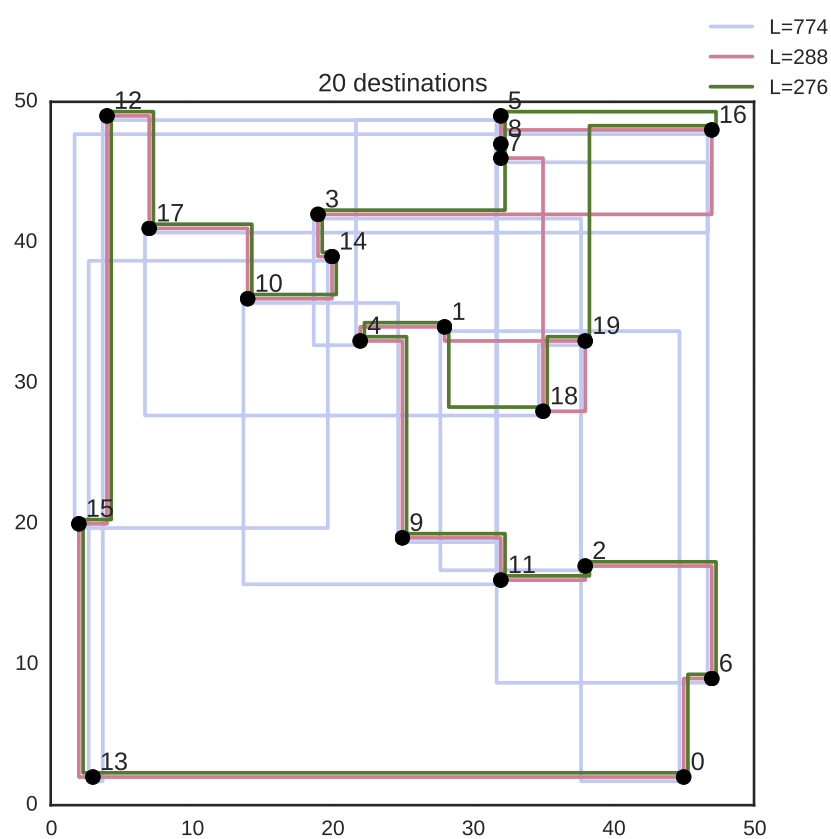


Fig. 35.1 – **Figure** : Ville-test, avec 20 destinations et trois trajets de longueurs différentes : un trajet aléatoire ($L=774$), un trajet *plus proche voisins* ($L=288$), et un trajet après optimisation *opt-2* ($L=276$).

35.2.5 Heuristique *Opt-2*

1. `Trajet.interversion(i, j)` : retourne un *nouveau Trajet* résultant de l'interversion des 2 étapes *i* et *j*.
2. `Ville.optimisation_trajet(trajet)` : retourne le *Trajet* le plus court de tous les trajets « voisins » à *trajet* (i.e. résultant d'une simple intervention de 2 étapes), ou *trajet* lui-même s'il est le plus court.
3. `Ville.trajet_opt2(trajet=None, maxiter=100)` : à partir d'un *trajet* initial (par défaut le trajet des plus proches voisins), retourne un *Trajet* optimisé de façon itérative par intervention successive de 2 étapes. Le nombre maximum d'itération est *maxiter*.

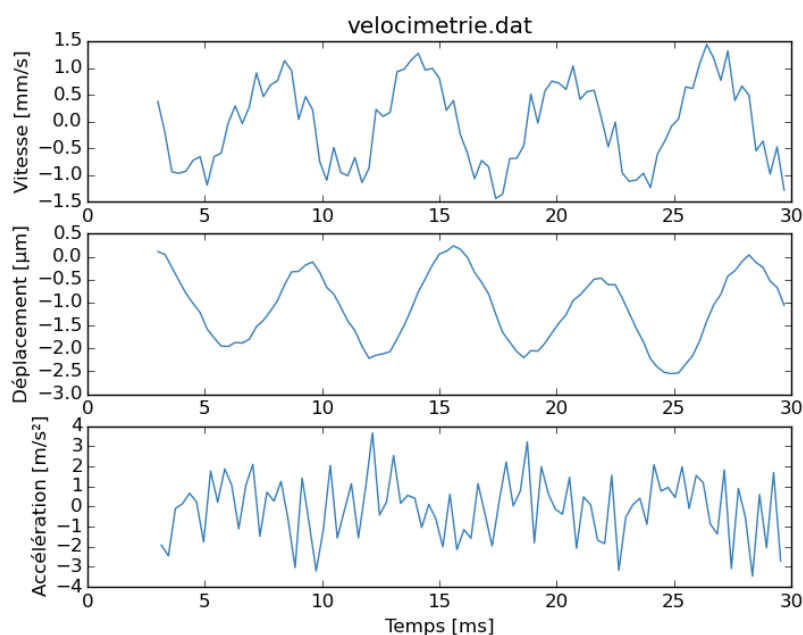
35.2.6 Questions hors-barème

À l'aide de la librairie `matplotlib` :

1. `Ville.figure()` : trace la figure représentant les destinations de la ville (similaire à la Figure).
2. `Ville.figure(trajet=None)` : compléter la méthode précédente pour ajouter un trajet au plan de la ville (utiliser `matplotlib.step()` pour des trajets de type « Manhattan »).

35.3 Correction

Corrigé



Bibliographie

[Matplotlib07] 2007CSE....9...90H
[Astropy13] 2013A&A...558A..33A

Symbols

*
 dépaquetage, 21, 22
 **
 dépaquetage, 21
 **kwargs, 21
 *args, 21
 @
 décorateur, 22
 \$PATH, 6

A

agg
 pandas, 56
 all
 numpy, 36
 allclose
 numpy, 36
 any
 numpy, 36
 arange
 numpy, 30
 argparse
 module, 26
 args, 13
 array
 numpy, 30
 assert
 exceptions, 17
 astropy
 module, 60
 at
 pandas, 50
 Axes
 matplotlib, 41
 axis
 numpy, 35

B

bool
 type numérique, 7, 8
 break, 9
 broadcasting
 numpy, 34

C

class, 18
 columns
 pandas, 49
 complex
 type numérique, 7
 continue, 9
 cut
 pandas, 55

D

décorateur
 @, 22
 dépaquetage
 *, 21, 22
 **, 21
 DataArray
 xarray, 58
 DataFrame
 pandas, 48
 DataSet
 xarray, 58
 def, 13
 dict
 itérables, 7
 dir, 10
 dot
 numpy, 35
 drop
 pandas, 52
 dropna
 pandas, 52
 dstack
 numpy, 33
 dtype
 numpy, 36

E

exceptions
 assert, 17
 raise, 17
 try ... except, 16
 expand_dims
 numpy, 33

F

Figure
 matplotlib, 41
file, 20
fillna
 pandas, 52
filter
 pandas, 50
float
 type numérique, 7
for ... in, 9
full
 numpy, 30

G

genfromtxt
 numpy, 36, 38
GridSpec
 matplotlib, 42
groupby
 pandas, 56

H

hstack
 numpy, 33

I

iat
 pandas, 50
identity
 numpy, 35
idxmin
 pandas, 55
if ... elif ... else, 9
iloc
 pandas, 50
import, 14
Index
 pandas, 49
index
 pandas, 49
input, 20
int
 type numérique, 7
interpréteur
 ipython, 6
 python, 6
isinstance, 7
itérables, 11
 dict, 7
 len, 9
 list, 7
 set, 7
 slice, 10
 str, 7, 9
 tuple, 7

K

kwargs, 13

L

lambda, 23
len
 itérables, 9
linspace
 numpy, 31
list
 itérables, 7
loc
 pandas, 50
 xarray, 58
logspace
 numpy, 31

M

matplotlib
 Axes, 41
 Figure, 41
 GridSpec, 42
 module, 40
 mplot3d, 45
 pylab, 40
 pyplot, 40
 savefig, 43
 show, 43
matrix
 numpy, 35
mayavi/mlab
 module, 45
meshgrid
 numpy, 31
mgrid
 numpy, 31
module
 argparse, 26
 astropy, 60
 matplotlib, 40
 mayavi/mlab, 45
 numpy, 29
 numpy.fft, 38
 numpy.linalg, 35
 numpy.ma, 37
 numpy.polynomial, 38
 numpy.random, 31
 pandas, 47
 pickle, 26
 scipy, 39
 seaborn, 57
 sys, 25
 turtle, 95
 xarray, 58
mplot3d
 matplotlib, 45

N

- ndarray
 - numpy, 30
- newaxis
 - numpy, 33, 34
- None, 7
- nonzero
 - numpy, 34
- numpy
 - all, 36
 - allclose, 36
 - any, 36
 - arange, 30
 - array, 30
 - axis, 35
 - broadcasting, 34
 - dot, 35
 - dstack, 33
 - dtype, 36
 - expand_dims, 33
 - full, 30
 - genfromtxt, 36, 38
 - hstack, 33
 - identity, 35
 - linspace, 31
 - logspace, 31
 - matrix, 35
 - meshgrid, 31
 - mgrid, 31
 - module, 29
 - ndarray, 30
 - newaxis, 33, 34
 - nonzero, 34
 - ones, 30
 - ravel, 32
 - recarray, 36
 - reshape, 32
 - resize, 33
 - rollaxis, 33
 - save/load, 38
 - savetxt/loadtxt, 38
 - slicing, 32
 - squeeze, 33
 - transpose, 33
 - ufuncs, 36
 - vstack, 33
 - where, 34
 - zeros, 30
- numpy.fft
 - module, 38
- numpy.linalg
 - module, 35
- numpy.ma
 - module, 37
- numpy.polynomial
 - module, 38
- numpy.random
 - module, 31

O

- ones
 - numpy, 30
- opérateur ternaire (... if ... else ...), 8
- open, 20

P

- pandas
 - agg, 56
 - at, 50
 - columns, 49
 - cut, 55
 - DataFrame, 48
 - drop, 52
 - dropna, 52
 - fillna, 52
 - filter, 50
 - groupby, 56
 - iat, 50
 - idxmin, 55
 - iloc, 50
 - Index, 49
 - index, 49
 - loc, 50
 - module, 47
 - pivot_table, 57
 - qcut, 55
 - query, 50
 - reset_index, 53
 - Series, 48
 - set_index, 53
 - sort_index, 53, 55
 - sort_values, 55
 - value_counts, 55
 - values, 49
 - xs, 53
- pickle
 - module, 26
- pivot_table
 - pandas, 57
- print, 11, 20
- pylab
 - matplotlib, 40
- pyplot
 - matplotlib, 40
- Python Enhancement Proposals
 - PEP 20, 63
 - PEP 257, 16, 70
 - PEP 308, 9
 - PEP 3132, 22
 - PEP 343, 24
 - PEP 448, 21
 - PEP 466, 24
 - PEP 484, 24
 - PEP 498, 24
 - PEP 526, 24
 - PEP 8, 64

Q

qcut
 pandas, 55
query
 pandas, 50

R

raise
 exceptions, 17
range, 7
ravel
 numpy, 32
recarray
 numpy, 36
reset_index
 pandas, 53
reshape
 numpy, 32
resize
 numpy, 33
rollaxis
 numpy, 33

S

save/load
 numpy, 38
savefig
 matplotlib, 43
savetxt/loadtxt
 numpy, 38
scipy
 module, 39
seaborn
 module, 57
sel
 xarray, 58
Series
 pandas, 48
set
 itérables, 7
set_index
 pandas, 53
show
 matplotlib, 43
slice
 itérables, 10
slicing
 numpy, 32
sort_index
 pandas, 53, 55
sort_values
 pandas, 55
squeeze
 numpy, 33
str
 itérables, 7, 9
sys
 module, 25

T

transpose
 numpy, 33
try ... except
 exceptions, 16
tuple
 itérables, 7
turtle
 module, 95
type, 7
type numérique
 bool, 7, 8
 complex, 7
 float, 7
 int, 7

U

ufuncs
 numpy, 36

V

value_counts
 pandas, 55
values
 pandas, 49
variable d'environnement
 \$PATH, 6
vstack
 numpy, 33

W

where
 numpy, 34
while, 9

X

xarray
 DataArray, 58
 DataSet, 58
 loc, 58
 module, 58
 sel, 58
xs
 pandas, 53

Z

Zen du Python, 63
zeros
 numpy, 30