

Logiciels scientifiques - HLMA310

Partie 1 : Python

Joseph Salmon

<http://josephsalmon.eu>

Université de Montpellier



Enseignant : cours magistral

- **Joseph Salmon :**

- ▶ Situation actuelle : Professeur à l'université de Montpellier
- ▶ Précédemment : Paris Diderot-Paris 7, Duke University, Télécom ParisTech, University of Washington
- ▶ Spécialités : statistiques en grande dimension, optimisation, agrégation, traitement des images
- ▶ Bureau : 415, Bat.9

Contact:

Joseph Salmon

joseph.salmon@umontpellier.fr

Github: @josephsalmon



Twitter: @salmonjsph



Enseignant : TPs

- **Pascal Azerad :**

- ▶ Situation actuelle : Maître de conférences à l'université de Montpellier
- ▶ Spécialités : analyse numérique et mécanique des fluides
- ▶ Email : *pascal.azerad@umontpellier.fr*
- ▶ Bureau : 232, Bat 9

- **Florent Bascou :**

- ▶ Situation actuelle : Doctorant à l'université de Montpellier
- ▶ Spécialités : probabilités
- ▶ Email : *florent.bascou@etu.umontpellier.fr*
- ▶ Bureau : 128, Bat. 9

Ressources en ligne

Informations principales : site du cours

<http://josephsalmon.eu/HLMA310.html>

- ▶ Syllabus
- ▶ Slides (au fil de l'eau)
- ▶ Poly (en cours)
- ▶ Feuilles de TP

Rendu TP : Moodle de l'université

<https://moodle.umontpellier.fr/course/view.php?id=5558>

Chat discord : <https://discord.gg/yvvTY9>

Validation (pour la partie Python)

Modifications possible à cause de la situation sanitaire

Partie Python 50% / partie Matlab 50% de la note

- TP noté :
 - ▶ Mise en ligne : mardi 20 octobre, à rendre sur le Moodle du cours pour le vendredi 23 octobre, 23h59
Rendu : fichier `.ipynb` unique (format jupyter notebook)
Plus d'informations à venir en cours et en TP
- Quizzes :
 - ▶ Quiz long : le **19 octobre**
 - ▶ Quiz(s) court(s) : le **5 octobre**



Le rendu est individuel pour le TP noté!!!

Notation pour le TP noté

Rendu : par Moodle en déposant un fichier `nom_prenom.ipynb` dans le dossier adéquat.

Détails de la notation du TP (noté sur 20) :

- ▶ Qualité des réponses aux questions : **14** pts
- ▶ Qualité de rédaction et d'orthographe : **1** pt
- ▶ Qualité des graphiques (légendes, couleurs) : **1** pt
- ▶ Style PEP8 valide : **2** pts
- ▶ Qualité d'écriture du code (nom de variable clair, commentaires utiles, code synthétique, etc.) : **1** pt
- ▶ Notebook reproductible (*i.e.*, *Restart & Run all* marche correctement) et absence de bug : **1** pt

Pénalités :

- ▶ Envoi par mail : **zéro**
- ▶ Retard : **zéro**, sauf excuse validée par l'administration

Bonus

1 pt supplémentaire sur la note finale (partie Python) par contribution à l'amélioration du cours (présentations, codes, etc.)

Contraintes :

- ▶ seule la première amélioration reçue est “rémunérée”
- ▶ déposer un fichier **.txt** (taille <10 ko) en créant une fiche dans la partie du Moodle intitulée “Bonus - Proposition d'amélioration”
- ▶ détailler précisément (ligne de code, page des présentations, etc.) l'amélioration proposée, ce qu'elle corrige et/ou améliore
- ▶ pour les fautes d'orthographe : proposer au minimum 5 corrections par contribution
- ▶ Bonus maximum : **2 points**

Prérequis - à revoir seul

- ▶ Bases de **probabilités** : probabilité, densité, espérance, théorème central limite
Lecture : Foata et Fuchs (1996)
- ▶ Bases de l'**algèbre (bi-)linéaire** : espaces vectoriels, normes, produit scalaire, matrices, diagonalisation
Lecture : Horn et Johnson (1994)

Prérequis - à revoir seul

- ▶ Bases de **probabilités** : probabilité, densité, espérance, théorème central limite
Lecture : Foata et Fuchs (1996)
- ▶ Bases de l'**algèbre (bi-)linéaire** : espaces vectoriels, normes, produit scalaire, matrices, diagonalisation
Lecture : Horn et Johnson (1994)
- ▶ Bases de l'**algèbre linéaire numérique** : résolution de système, pivot de Gauss, factorisation de matrices, conditionnement, etc.
Lecture : Golub et VanLoan (2013), *Applied Numerical Computing* par L. Vandenberghe

Prérequis - à revoir seul

- ▶ Bases de **probabilités** : probabilité, densité, espérance, théorème central limite
Lecture : Foata et Fuchs (1996)
- ▶ Bases de l'**algèbre (bi-)linéaire** : espaces vectoriels, normes, produit scalaire, matrices, diagonalisation
Lecture : Horn et Johnson (1994)
- ▶ Bases de l'**algèbre linéaire numérique** : résolution de système, pivot de Gauss, factorisation de matrices, conditionnement, etc.
Lecture : Golub et VanLoan (2013), [Applied Numerical Computing](#) par L. Vandenberghe

Description du cours

Objectifs : utilisation de Python pour le traitement et la visualisation de données.

- ▶ bases de programmation/algorithmique en Python
- ▶ librairies de méthodes numériques (**numpy**, **scipy**)
- ▶ librairies de traitement de bases de données (pandas)
- ▶ visualisation (matplotlib, seaborn)
- ▶ introduire des bonnes pratiques numériques valables pour tous les langages (lisibilité, documentation)

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

Conditions et boucles

Fonctions en Python

Aspects algorithmiques : quelques conseils

(TODO : Démo sur une machine de l'université)



Python : environnement fonctionnel à l'UM : anaconda3

Démarche :

1. Lancer anaconda3 (cliquer sur l'icône en au haut à gauche)
2. Taper jupyter-notebook et appuyer sur la touche "entrée".

Rem: la procédure complète est détaillée sur le polycopié du cours

<http://josephsalmon.eu/enseignement/Montpellier/HLMA310/IntroPython.pdf>

Rem: connexion à distance avec x2go sur les machines de l'UM

<https://moodle.umontpellier.fr/mod/page/view.php?id=126345>

Installation personnelle

Installation de Python sur machines personnelles :

Utiliser `conda` / `anaconda` (tous OS) (et `pip` seulement pour les packages les plus rares, non disponibles sous `anaconda`)

Attention : pas d'aide des enseignants sur ce point ;
entraidez-vous !

Format de rendus de TP : `jupyter notebook`

Rem: en TP, prenez vos portables si vous préférez garder votre environnement (packages, versions, etc.)

Python 2 vs. Python 3

Principales différences :

```
print "bonjour"
```

vs.

```
print("bonjour")
```

et aussi

```
xrange
```

vs.

```
range()
```

Migration en cours de toutes les librairies populaires :

Exemple: Numpy : arrête de supporter Python 2.7 en 2019

Rem: détails des différences dans le poly

Rem: je suis passé à Python 3 en 2017

Conseils généraux pour l'année

- ▶ Adoptez des règles d'écriture de code et tenez-y vous !
Exemple: PEP8 pour Python (utiliser **AutoPEP8**, ou pour les notebooks <https://github.com/kenko000/jupyter-autopep8>)
- ▶ Utilisez **Markdown** (.md) pour les parties rédigées / comptes-rendus (e.g., markdown-preview-plus avec **VSCode**)

"A (wo)man must have a code."
– Bunk

Source : [The Wire](#), épisode 7, saison 1.

- ▶ Apprenez de bons exemples (ouvrir les codes sources !) :
<https://github.com/scikit-learn/>,
<http://jakevdp.github.io/>, etc.

TODO : exemple de jupyter notebook, ouvrir
`prise_en_main_notebook.ipynb`

Éditeurs de texte (pas seulement pour Python)

Motivation : jupyter notebook excellent pour des projets courts.
Cas plus long, utiliser : IPython + éditeur de texte avancé

Éditeurs recommandés :

- ▶ Visual Studio Code
- ▶ Atom
- ▶ Sublime Text
- ▶ vim (fort coût d'entrée, déconseillé)
- ▶ emacs (fort coût d'entrée, déconseillé)

Bénéfices : coloration syntaxique, auto-complétion du code, débogueur graphique, correcteur d'orthographe

Environnement intégré (plus avancé)

PyCharm :

- ▶ coloration syntaxique
- ▶ auto-complétion
- ▶ vérification de code dans l'environnement
- ▶ débogueur graphique
- ▶ intégration avec gestionnaires de versions (git)
- ▶ gestion des environnements virtuels (VirtualEnv)
- ▶ gestion des tests, etc.

Versionnement de code

Motivation : garder trace de son code et de l'évolution incrémentale de ses fichiers, notamment (mais pas uniquement) pour le travail en groupe

- ▶ système privilégié **Git**
- ▶ utilisable avec des solutions en ligne (**Gitlab**, **Github**)

Historique : popularisé par la communauté logiciel libre et créé en 2005 par Linus Torvalds (auteur du noyau Linux)

Exemple: <https://github.com/scikit-learn/scikit-learn>

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

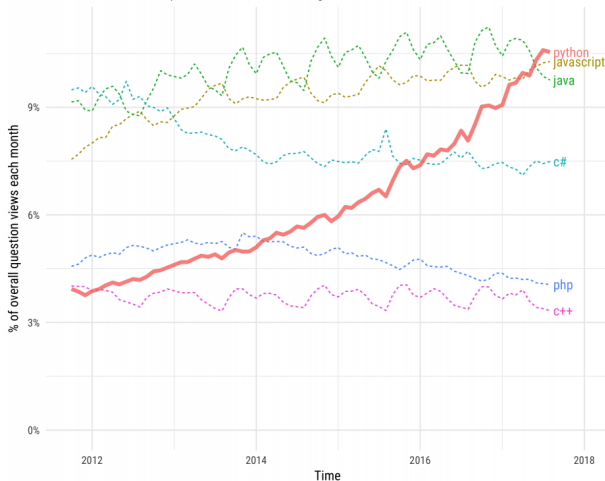
Conditions et boucles

Fonctions en Python

Popularité de Python sur Stackoverflow

Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries

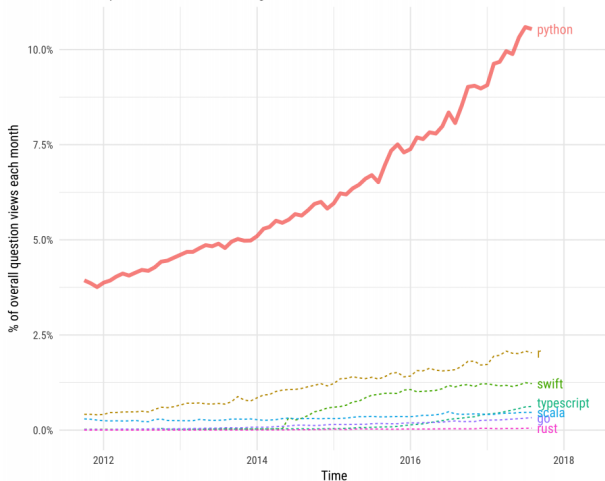


Source : <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>

Popularité de Python sur Stackoverflow

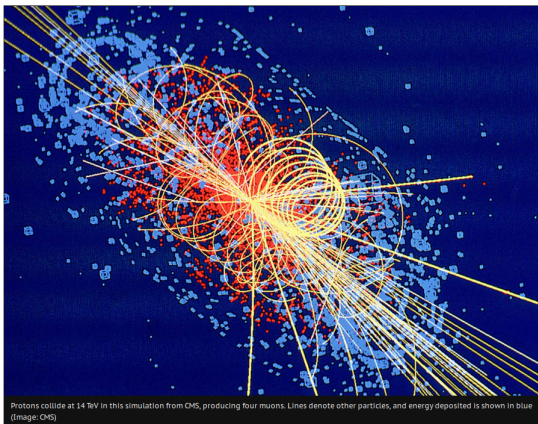
Python compared to smaller, growing technologies

Based on question traffic in World Bank high-income countries



Source : <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>

Python dans les médias : découverte du Boson de Higgs (CERN, 2012)

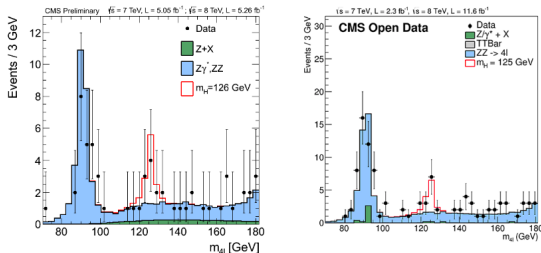


Collisions

Sources :

- ▶ <https://home.cern/fr/science/physics/higgs-boson>
- ▶ <https://cms.cern/news/observing-higgs-over-one-petabyte-new-cms-open-data>

Python dans les médias : découverte du Boson de Higgs (CERN, 2012)



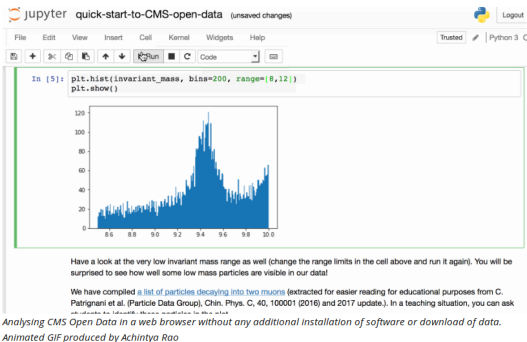
Left: The official CMS plot for the "Higgs to four leptons" channel, shown on the day of the Higgs discovery announcement. Right: A similar plot produced by Nur Zulaiha Jomhari et al. using CMS Open Data from 2011 and 2012. Although the plots appear similar, the analysis with CMS Open Data uses more data (at 8 TeV and overall) than the official CMS one from the original discovery but is a lot less sophisticated and is not scrutinised by the wider CMS community of experts.

Matplotlib (pour la visualisation)

Sources :

- ▶ <https://home.cern/fr/science/physics/higgs-boson>
- ▶ <https://cms.cern/news/observing-higgs-over-one-petabyte-new-cms-open-data>

Python dans les médias : découverte du Boson de Higgs (CERN, 2012)



Jupyter notebook (pour la présentation)

Sources :

- ▶ <https://home.cern/fr/science/physics/higgs-boson>
- ▶ <https://cms.cern/news/observing-higgs-over-one-petabyte-new-cms-open-data>


Python dans l'académique : aspect enseignement

Python est au programme des classes préparatoires aux grandes écoles (depuis 2013)

“Depuis la réforme des programmes de 2013, l'informatique est présente dans les programmes de CPGE à deux niveaux. Un tronc commun à chacune des trois filières MP, PC et PSI se donne pour objectif d'apporter aux étudiants la maîtrise d'un certain nombre de concepts de base : conception d'algorithmes, choix de représentations appropriées des données, etc. à travers l'apprentissage du langage Python.”

Source : <https://info-llg.fr/>

Python dans l'académique : aspect recherche

Exemple d'une package populaire d'apprentissage automatique
( : *machine learning*) : scikit-learn

Scikit-learn: Machine learning in Python

[F Pedregosa](#), [G Varoquaux](#), [A Gramfort](#)... - Journal of machine ..., 2011 - jmlr.org

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level ...


☆ 99 **Cité 11668 fois** Autres articles Les 31 versions »

Source : Google Scholar (8/09/2018)

- ▶ ≈ 1000 pages de documentation
- ▶ ≈ 500, 000 utilisateurs les 30 derniers jours (fin 2017)
- ▶ ≈ 42, 000, 000 pages vues sur le site (2017)

Source : Alexandre Gramfort (INRIA - Parietal)

Python dans l'académique : aspect recherche

Exemple d'une package populaire d'apprentissage automatique
( : *machine learning*) : scikit-learn

Scikit-learn: Machine learning in Python

[F Pedregosa](#), [G Varoquaux](#), [A Gramfort](#)... - Journal of machine ..., 2011 - jmlr.org

Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level ...

☆ 99 **Cité 11668 fois** Autres articles Les 31 versions »


Source : Google Scholar (8/09/2018)


- ▶ ≈ 1000 pages de documentation
- ▶ ≈ 500, 000 utilisateurs les 30 derniers jours (fin 2017)
- ▶ ≈ 42, 000, 000 pages vues sur le site (2017)

Source : Alexandre Gramfort (INRIA - Parietal)

Explication du succès de Python

Proverbe (récent) :

 : Python; *the second best language for everything!*

 : Python; *le deuxième meilleur langage pour tout!*

Autres bénéfices de Python :

- ▶ langage compact (5X plus compact que Java ou C++) : l'indentation (**4 espaces**) remplacent les {...} ou (...)
- ▶ d'autres langages
- ▶ ne requiert pas d'étape — potentiellement longue — de compilation (comme le C) \implies débogage plus facile
- ▶ tous systèmes d'exploitation (Linux, MacOS, Windows)

En résumé : Python = excellent “couteau suisse” numérique

Les limites (car il en existe !)

- ▶ vitesse d'exécution souvent inférieure vs. C/C++, Fortran (langage compilé de plus bas niveau)
- ▶ langage permissif, un programme peut "tourner" avec des "erreurs" non dépistées ; vigilance donc !
- ▶ l'indentation : les caractères sont implicites (espaces !), il peut être difficile de dépister une erreur de ce type
- ▶ les notebooks utiles et beaux visuellement, mais dangereux
TO DO: exemples sur cas simple

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

Conditions et boucles

Fonctions en Python

Essor des librairies “libres” (: *open source*)⁽¹⁾

Apprentissage automatique ( : *Machine learning*) :

- sklearn (2010)
- tensorflow (2015)

Traitement du langage ( : *Natural Language Processing*) :

- nltk (2001)

Traitement des images ( : *image processing*) :

- skimage (2009)

Développement web :

- django (2005)

...

(1) ce n'est pas le cas de Matlab par exemple ; dont le coût = plusieurs dizaines de milliers d'euros pour l'université

Librairies indispensables en Python (pour ce cours)

- ▶ **Numpy**

<https://www.labri.fr/perso/nrougier/from-python-to-numpy/index.html>

https://github.com/agramfort/liesse_telecom_paristech_python/blob/master/2-Numpy.ipynb

- ▶ **Scipy :**

https://github.com/agramfort/liesse_telecom_paristech_python/blob/master/3-Scipy.ipynb

- ▶ **Matplotlib :**

<https://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html>

<https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833>

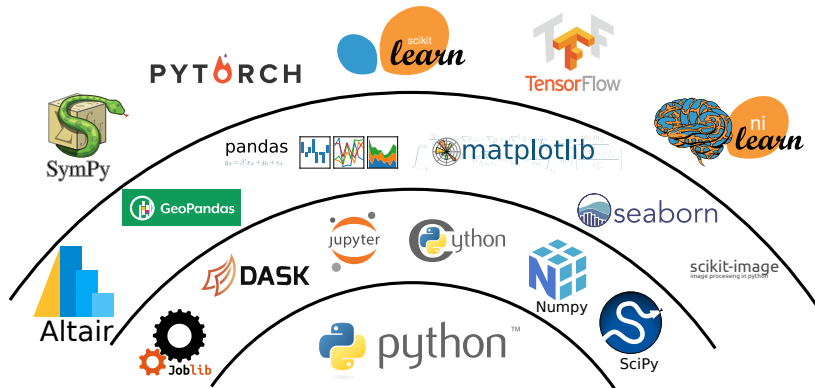
- ▶ **Pandas :** <https://github.com/jorisvandenbossche/pandas-tutorial>

Tutos/Vidéos de Jake Vanderplas **OBLIGATOIRES** ! : en particulier les vidéos 1 ; 2 ; 7 ; 8 ; 9 et 10.

Lien : [Reproducible Data Analysis in Jupyter](#)

Écosystème Python :

Panorama partiel et partiel



Livres et ressources en ligne complémentaires

Général : Skiena, The algorithm design manual, 1998 (en anglais)

Général : Courant *et al.*, Informatique pour tous en classes préparatoires aux grandes écoles : Manuel d'algorithmique et programmation structurée avec Python, 2013

Général / Science des données : Guttag, Introduction to Computation and Programming, 2016 (en anglais)

Science des données : J. Van DerPlas, With Application to Understanding Data, 2016 (en anglais)

Code et style : Boswell et Foucher, The Art of Readable Code, 2001 (en anglais)

Python : <http://www.scipy-lectures.org/>

Visualisation (sous R) : <https://serialmentor.com/dataviz/>

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

- Affichage et aide

- Chaînes de caractères et nombres

- Listes et variantes

Conditions et boucles

Fonctions en Python

Affichage

Afficher :

```
>>> print('Salut tout le monde')  
Salut tout le monde
```

Obtenir de l'aide sur une fonction/commande que l'on connaît :


```
>>> print?
```

Erreur et bugs

Exemple: une erreur simple

```
>>> print('Salut  
SyntaxError: EOL while scanning string literal
```

Python est (assez) explicite dans les messages d'erreur. Il faut généralement commencer par lire ces messages par le bas.

Ici, l'erreur indique une mauvaise syntaxe, et que la fin de la ligne ( : *End Of Line*) a été atteinte pendant la lecture de la chaîne de caractères.

Chaînes de caractères (: *string*)

Chaînes de caractères : permettent de représenter du texte. Elles peuvent être entourées par des apostrophes ou des guillemets.

Ainsi :

```
>>> salutation_apostrophe = 'Salut tout le monde'  
>>> salutation_guillemet = "Salut tout le monde"
```

sont en fait deux chaînes identiques ⁽²⁾

Rem: pour connaître la taille ( : *length*) il suffit d'utiliser **len**

```
>>> len(salutation_apostrophe)  
>>> len(salutation_guillemet)  
20  
20
```

(2) on pourra tester ce fait en utilisant la commande : `salutation_apostrophe == salutation_guillemet`, ou plus naïvement, en visualisant les deux chaînes

Propriétés des chaînes de caractères

Polymorphisme : le signe plus '+' signifie différentes choses pour différents objets : l'addition pour les entiers et la concaténation pour les chaînes de caractères
Propriété générale de Python : la nature d'une opération dépend des objets sur lesquels elle agit

Immuabilité : Les chaînes sont immuables en Python- elles ne peuvent pas être modifiées sur place (en anglais *in place*) après leur création. L'immuabilité : utilisée pour assurer qu'un objet reste constant tout au long d'un programme

Chaînes de caractères (raffinements)

Une version de `print` qui permet d'afficher des listes de caractères plus évoluées et contenant des paramètres est la suivante :

```
>>> first_name = 'Joseph'
>>> last_name = 'Salmon'
>>> print(type(first_name))
<class 'str'>
>>> print('Prénom : {} ;
           nom : {} ;'.format(first_name, last_name))
Prénom : Joseph ;nom : Salmon;
```

Rem: pour l'épineuse question des accents et des encodages, le lecteur curieux pourra découvrir les joies de l'utf8 ici :

- ▶ <http://sdz.tdct.org/sdz/comprendre-les-encodages.html>
- ▶ <http://sametmax.com/lencoding-en-python-une-bonne-fois-pour-toute/>

Caractères spéciaux

Quelques chaînes spéciales méritent d'être connues comme l'apostrophe, le retour à la ligne ou la tabulation :

```
>>> print('l\'apostrophe dans une chaîne')
      print('Hacque ,\n post \n postridie')
      print('\t \t descriptione, post \npostridie')
l'apostrophe dans une chaîne
Hacque ,
  post
  postridie
      descriptione, post
postridie
```

Opérations sur les chaînes de caractères

Concaténation :

```
>>> 'Et un,' + ' et deux,' + ' et trois zéros'  
'Et un, et deux, et trois zéros'
```

Répétition :

```
>>> 'Et un,' * 8  
'Et un,Et un,Et un,Et un,Et un,Et un,Et un,Et un,'
```


Remplacement :

```
>>> 'Et un,'.replace('un', 'deux')  
'Et deux,'
```

Découpage :

```
>>> 'aaa, bbb, cccc, dd'.split(',')  
['aaa', ' bbb', ' cccc', ' dd']
```

Opérations sur les chaînes de caractères

Extraction et découpage ( : *slicing*) :

```
>>> chef_gaulois = 'Abraracourcix'  
    print(chef_gaulois[0], chef_gaulois[-1])  
    print(chef_gaulois[:4], chef_gaulois[4:])  
    print(chef_gaulois[2:5])  
    print(chef_gaulois[::2])  
    print(chef_gaulois[::-1])
```

A x

Abra racourcix

rar


Arrcucx

xicruocararbA

plus d'info sur les chaînes :

http://www.fil.univ-lille1.fr/~wegrzyno/portail/Info/Doc/HTML/seq4_chaines_caracteres.html

Les entiers

Nombres les plus simples manipulables : les entiers ( : *integers*).

Type le plus simple et disposant des 4 opérations usuelles :

+, -, *, //

Exemple: calcul de 5!

```
>>> fact5 = 1 * 2 * 3 * 4 * 5
>>> print(fact5)
120
>>> type(fact5)
int
```

Attention aux divisions !

L'opération de division est particulière : la division entière est en fait obtenue par la commande `//` et le reste par `%`. Ainsi,

```
>>> 7 // 3
2
>>> 7 % 3
1
```

mais en revanche :

```
>>> 7 / 3
2.3333333333333335
```

Une manière de comprendre cela est que

```
>>> type(7 / 3)
float
>>> type(7 // 3)
int
```

Explications

Python a changé le type d'entier vers flottant (float), que nous allons voir maintenant.

Vigilance donc sur ce type d'opération !

Nombres flottants (: *float*)

Bref aperçu ⁽³⁾ : ce concept sera revu en détails par Vanessa Lleras (Partie Matlab)

Rem: les nombres réels mathématiques sont représentés par des suites infinies de chiffres, *e.g.*, π ; impossible pour un ordinateur !

Nombres flottants (numériques) : nombres qui permettent de représenter, à précision choisie (ou donnée par défaut), les nombres réels que l'on manipule en mathématique.

```
>>> nb_flottant = 3.2  
>>> print(type(nb_flottant))  
float
```

(3) J.-M MULLER et al. *Handbook of Floating-Point Arithmetic (2nd Ed.)* Springer, 2018.

Flottants opérations usuelles

Notation exponentielle : Permet d'afficher les nombres voulus en format scientifique (plus lisible)

```
>>> 1e-3  
0.001  
>>> 1e10  
10000000000.0
```

Opérations usuelles : ici il n'y pas de soucis pour les 4 opérations usuelles : +, -, *, /



le carré d'un nombre x s'écrit par contre $x ** 2$

Flottants et arithmétique étrange

Arrondi : “l’arithmétique” associée aux flottants est troublante

- ▶ quand on retranche des quantités petites du même ordre
- ▶ quand on ajoute des nombres trop grands

Pour comprendre l’aspect délicat des flottants :

```
>>> 0.1 + 0.2  
0.30000000000000004
```

ou


```
>>> 1e-41 - 1e-40  
-8.999999999999999e-41
```

et pour des grands nombres :

```
>>> nb_big1 = 1e150  
>>> nb_big2 = 1e150  
>>> nb_big1 * nb_big2  
9.999999999999999e+299
```

inf et nan

Pour des nombres trop grands on atteint la valeur “infinie”, représentée par `inf` en Python.

Quantité tout aussi étrange mais parfois utile à connaître : le `nan` ( : *not a number*) représente un nombre qui n'en est pas un !

Exemple: pour obtenir des `nan`'s

```
>>> too_big = nb_big1 * nb_big2 ** 2
>>> print(too_big)
inf
>>> too_big / too_big
nan
>>> too_big - too_big
nan
```

Pour aller plus loin : <https://www.stat.berkeley.edu/~nolan/stat133/Spr04/chapters/representations.pdf>

Conversion float/int/str

Passer de **float** à **int** (perte possible !)

```
>>> int(3.0), type(int(3.0))  
(3, int)
```

Passer de **int** à **float**

```
>>> float(3), type(float(3))  
(3.0, float)
```

Passer de **float** à **str**

```
>>> str(3.), type(str(3.))  
( '3.0', str)
```

Passer de **str** à **float**

```
>>> float('3.'), type(float('3.'))  
(3.0, float)
```

Nombres complexes

Représentation des nombres complexes : le nombre complexe dont le carré est -1 s'écrit $1j$ sous Python.

Exemple:

```
>>> (1j)**2  
(-1+0j)
```

On peut aussi utiliser la librairie `cmath` pour avoir accès au module, à la phase, etc.

```
>>> import cmath  
>>> z = complex(4, 3)  
>>> print(abs(z), cmath.polar(z))  
>>> print(z.real, z.imag)  
5.0 (5.0, 0.6435011087932844)  
4.0 3.0
```

Booléens

Booléens⁽⁴⁾ : variables qui valent zéro ou un, ou de manière équivalente en Python **True** (vrai) ou **False** (faux).

Une opération commune pour créer des booléens est un test :

```
>>> a = 3
>>> b = 4
>>> print(a == 4)
False
```

En revanche noter que :

```
>>> petit_nb = 0.1 + 0.2 - 0.3
>>> mon_test = petit_nb == 0
>>> print(mon_test)
False
```

(4) nom est associé à [George Boole](#) (1815-1864)

Test sur des flottants

Pour tester l'égalité (?) entre nombres flottants, on peut utiliser la librairie `math` :

```
>>> import math
>>> print(petit_nb)
5.551115123125783e-17
>>> print(math.isclose(0., petit_nb, abs_tol=1e-5))
True
>>> print(math.isclose(0., petit_nb, abs_tol=1e-17))
False
```

`isclose` teste (à la précision voulue) si deux nombres sont "égaux" ou non.

Opérations sur les booléens

Comme en logique on a les opérations suivantes :

- ▶ “ou” (🇬🇧 : *or*), commande **or**
- ▶ “et” (🇬🇧 : *and*), commande **and**
- ▶ “non” (🇬🇧 : *not*), commande **not**
- ▶ “ou exclusif” (🇬🇧 : *xor*), commande **!=**



test d'égalité :

- ▶ **is** retourne **True** si deux variables pointent sur le même objet
- ▶ **==** retourne **True** si les objets auxquels les variables se réfèrent sont identiques⁽⁵⁾

(5) <https://stackoverflow.com/questions/132988/is-there-a-difference-between-and-is#133024> pour quelques exemples

Tables logiques : and

Expression	Résultat
True and True	True
True and False	False
False and True	False
False and False	False

TABLE – Table logique de **and**

Tables logiques : or

Expression	Résultat
True or True	True
True or False	True
False or True	True
False or False	False

TABLE – Table logique de **or**

Tables logiques : not

Expression	Résultat
True	False
False	True


TABLE – Table logique de **not**

Tables logiques : !=


Expression	Résultat
True != True	False
True != False	True
False != True	True
False != False	False

TABLE – Table logique de **xor**

Listes

Les listes ( : *list*) peuvent être créées en écrivant une suite de valeurs séparées par des virgules, le tout entre crochet

```
>>> ma_liste = ["bras", "jambes", 10, 12]
>>> ma_liste
['bras', 'jambes', 10, 12]
```

Rem: une liste peut mélanger des éléments ( : *items*) de types différents

Comme les chaînes de caractères :

- ▶ le premier item d'une liste **a l'indice 0**
- ▶ les listes supportent le *slicing*
- ▶ les listes supportent la concaténation

Listes (suite)

```
>>> a = ["bras", "jambes", 10, 12]
>>> a[0:2] = [] # Enlève quelques items
>>> a
[10, 12]
>>> a[1:1] = ['main', 'coude'] # Insertion
[10, 'main', 'coude', 12]
>>> a[:] = [] # Vider la liste
>>> a, len(a), type(a)
([], 0, list)
```

Dictionnaire :

clefs ( : *keys*) / valeurs ( : *values*)

Le **dictionnaire** est une structure dont les pseudo-indices (les clés) peuvent avoir un sens, et renvoient à des valeurs :

```
>>> dico = {}  
>>> dico['car'] = 'voiture'  
>>> dico['plane'] = 'avion'  
>>> dico['bus'] = 'bus'  
>>> print(dico)  
{'car': 'voiture', 'plane': 'avion', 'bus': 'bus'}
```

Notions de clef / valeur :

```
>>> print(dico.keys())  
>>> print(dico.values())
```

Dictionnaire : flexibilité et opérations

Un dictionnaire peut mélanger plusieurs types de valeur :

```
>>> dico_3 = {'nom': {'first': 'Alexandre', 'last': 'PetitChateau'},  
              'emploi': ['enseignant', 'chercheur'],  
              'age': 36}
```

Effacer un couple clef / valeur :

```
>>> del(dico['bus'])  
>>> dico  
{'car': 'voiture', 'plane': 'avion'}
```

Pour aller plus loin : https://fr.wikibooks.org/wiki/Programmation_Python/Listes

Limite des dictionnaires

Attention : les opérations comme la concaténation et l'extraction n'ont pas de sens pour un dictionnaire :

```
>>> print(dico[1:2])
TypeError: unhashable type: 'slice'
>>> print(dico + dico)
unsupported operand type(s) for +: 'dict' and 'dict'
```

Rem: pour fusionner deux dictionnaires on peut utiliser l'argument `update`

Autres types non décrits dans le cours

- ▶ tuple, sorte de liste immuable
- ▶ set (■ ■ : *ensemble*), notamment pour avoir les opérations de la théorie des ensembles

Pour plus de détails sur les types, voir par exemple :

https://github.com/sergulaydore/EE551A_Fall_2018/blob/master/lectures/lecture3/Lecture3.ipynb

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

Conditions et boucles

if, **then**, **else** et autres tests

Boucle **for**

Boucle **while**

Fonctions en Python

Instruction “if ... then ... else ...”

Instruction “**if ... then ... else ...**” : permet de créer des tests pour prendre des “décisions” en fonction de la valeur d’un booléen (une instruction) est vraie ou fausse.

Exemple: tester si un nombre est pair

```
>>> nb_a = 14
>>> if nb_a%2==0:
    print("nb_a est pair")
else:
    print("nb_a est impair")
nb_a est pair
```

Rem: une variante possible de == est **is**



Ne pas oublier les “deux points” : en fin de ligne, et les espaces (ou une tabulation) pour indenter correctement l’instruction

Instruction “if ... then ... else ...”


Pour des tests impliquant plus de deux cas possibles, la syntaxe nécessite d'utiliser **elif**

Exemple: trouver le reste d'un nombre modulo 3

```
nb_a = 11
if nb_a % 3 == 0:
    print("nb_a est congru à 0 modulo 3")
elif nb_a % 3 == 1:
    print("nb_a est congru à 1 modulo 3")
else:
    print("nb_a est congru à 2 modulo 3")
nb_a est congru à 2 modulo 3
```

Rem: **elif** est la concaténation de **else** et **if**

Boucle for : premiers pas

Boucle **for** ( : *pour*) : permet d'itérer des opérations sur un objet dont la taille est prédéfinie

Exemple: pour afficher (avec 3 chiffres après la virgule) la racine des nombres de 0 à 10 il suffit d'écrire



```
>>> for i in range(4):  
    print("La racine de {:d} est {:.3f}".  
          format(i, i**0.5))
```

Racine de 0: 0.000

Racine de 1: 1.000

Racine de 2: 1.414

Racine de 3: 1.732

Rem: d renvoie aux nombre décimaux ( : *decimal*) et f aux nombres flottants ( : *float*)

Commande enumerate

Commande utile en Python : **enumerate**, permet d'obtenir les éléments d'une liste et leurs indices pour itérer sur eux

Exemple: supposons que l'on ait créé la liste des carrés de 1 à 10

```
>>> liste_carres = [] # init: liste vide
    for i in range(4):
        liste_carres.append(i**2)
```

Maintenant pour itérer sur cette liste et par exemple trouver la racine des nombres de cette liste on écrit

```
>>> for i, carre in enumerate(liste_carres):
    print(int(carre**0.5))
```

```
0
1
2
3
```

Compréhensions et boucle for

Python inclut une opération avancée connue sous le nom de **compréhension de liste** ( : *list comprehension*)

```
>>> a = [1,4,2,7,1,9,0,3,4,6,6,6,8,3]
>>> [x for x in a if x > 5]
[7, 9, 6, 6, 6, 8]
```

Un second exemple :

```
>>> print([c * 2 for c in 'ohé'])
['oo', 'hh', 'ée']
```


Boucle while

while (en français : “tant que”) permet d’itérer des opérations tant qu’une condition n’est pas satisfaite

```
>>> i = 0
      print("Nb x t.q. 2^x < 10:")
      while (2**i < 10):
          print("{0}".format(i))
          i += 1 # incrémente le compteur
Nb x t.q. 2^x < 10:
0
1
2
3
```

En effet $16 = 2^4 > 10$ mais $8 = 2^3 < 10$



on peut créer des boucles sans fin (!)

Sommaire

Conseils numériques : pour le cours et delà

Introduction et motivation

Écosystème Python : les librairies

Formats et commandes usuelles en Python

Conditions et boucles

Fonctions en Python

Premiers éléments

Définition d'une fonction en Python :

- ▶ utiliser le mot clé **def**
- ▶ faire suivre du nom de la fonction
- ▶ puis de la signature entre parenthèses ()
- ▶ terminer la ligne par un symbole ' : ' ligne.
- ▶ de nouveau il faut quatre espaces pour indenter après les ' : '

```
>>> def fonction1():  
    print("mon test")
```

Pour la lancer il suffit alors de taper la commande suivante

```
>>> fonction1()  
mon test
```

Aide (: *docstring*)

Aide d'une fonction : **primordiale** !

Rappel : sklearn dispose par exemple de plus de 1000 pages de documentation / aide.

Motivations multiples :

- ▶ permet d'accélérer l'écriture sur un gros projet
- ▶ facilite l'échange en travail en groupe
- ▶ facilite l'échange avec soi-même dans le futur

Rem: **test unitaire** (autre pratique utile) : créer un test de la fonction, potentiellement avant même de terminer son écriture.
cf. pytest, <https://docs.pytest.org/en/latest/>, pour automatiser cette pratique

Entrées interactives

On peut créer des boîtes de dialogues avec Python en utilisant la commande `input` ; ainsi en lançant

```
>>> nom = input('Entrer nom de famille: ')
>>> prenom = input('Entrer prénom: ')
```

On obtient des boîtes de dialogues où l'on peut interagir avec le clavier. En y tapant "Salmon" et "Joseph", on peut obtenir :

```
>>> print("Je m'appelle {} {}".format(prenom, nom))
Je m'appelle Joseph Salmon.
```

Création / visualisation

Création de la documentation : commencer la fonction par des commentaires entre `"""` et `"""`

```
>>> def fonction2(s):  
    """  
    Affichage d'une chaîne et de sa longueur  
    """  
    print("{} est de longueur: {}".format(s) + str(len(s)))
```

Vérifier la fonction

```
>>> fonction2("Cette chaîne de test")  
Cette chaîne de test est de longueur : 20
```

Afficher l'aide avec la commande `help` ou la commande ?

```
>>> fonction2?
```

Usage de help / ?

Affichage de l'aide : diffère selon l'environnement (IPython, notebook, etc.) mais syntaxe identique

```
>>> fonction2?  
Signature: fonction2(s)  
Docstring: Affichage d'une chaîne et de sa longueur
```

Fonctionne pour toutes les fonctions standard de Python, e.g.,

```
>>> print?
```

Sortie de fonctions

Définition d'une sortie de fonction : mot clef **return** :

```
>>> def square(x):  
    """  
    Retourne le carré de x.  
    """  
    return x ** 2
```

Vérification de la fonction : sur des **int** ou des **float** (type d'entrée/sortie non précisé : polymorphisme) :

```
>>> square(3), square(0.4)  
(9, 0.16000000000000003)
```


Sortie multiples

Il est aussi possible de retourner plusieurs valeurs :

```
>>> def powers(x):  
    """  
    Retourne les premières puissances de x.  
    """  
    return x ** 2, x ** 3, x ** 4
```

L'appel de la fonction renvoie un **triplet** :

```
>>> out = powers(3)  
>>> print(out)  
>>> print(type(out))  
>>> x2, x3, x4 = powers(3)  
>>> print(x2, x3)  
(9, 27, 81)  
<class 'tuple'>  
9 27
```

Arguments optionnels

Arguments optionnels : s'ils ne sont pas donnés en appel de la fonction ils prennent la valeur donnée par défaut

```
>>> def ma_puissance(x, exposant=3, verbose=True):  
    """  
    Fonction calculant une puissance  
  
    Parameters  
    -----  
    x : float,  
        Valeur du nombre dont on calcule la puissance  
  
    exposant : float, default 3  
        Paramètre de l'exposant choisi  
  
    verbose : bool, default False  
        Paramètre d'affichage  
  
    Returns:  
    -----  
    Retourne x élevé à la puissance exposant (default=3)  
    """  
    if verbose is True:  
        print('version verbeuse')  
    return x ** exposant
```

Arguments optionnels (suite)

Arguments optionnels : appel possible avec ou sans eux

```
>>> ma_puissance(5)
version verbeuse
125
>>> ma_puissance(5, verbose=False, exposant=2)
25
```

Rem: ordre de saisie des paramètres sans importance en Python



ce n'est pas vrai pour les arguments non optionnel

Bibliographie I

- ▶ BOSWELL, D. et T. FOUCHER. *The Art of Readable Code*. O'Reilly Media, 2011.
- ▶ COURANT, J. et al. *Informatique pour tous en classes préparatoires aux grandes écoles : Manuel d'algorithmique et programmation structurée avec Python*. Eyrolles, 2013.
- ▶ FOATA, D. et A. FUCHS. *Calcul des probabilités : cours et exercices corrigés*. Masson, 1996.
- ▶ GOLUB, G. H. et C. F. VAN LOAN. *Matrix computations*. Fourth. Johns Hopkins University Press, Baltimore, MD, 2013, p. xiv+756.
- ▶ GUTTAG, J. V. *Introduction to Computation and Programming Using Python : With Application to Understanding Data*. MIT Press, 2016.
- ▶ HORN, R. A. et C. R. JOHNSON. *Topics in matrix analysis*. Corrected reprint of the 1991 original. Cambridge : Cambridge University Press, 1994, p. viii+607.

Bibliographie II

- ▶ MULLER, J.-M et al. *Handbook of Floating-Point Arithmetic (2nd Ed.)* Springer, 2018.
- ▶ SKIENA, S. S. *The algorithm design manual*. T. 1. Springer Science & Business Media, 1998.
- ▶ VANDERPLAS, J. *Python Data Science Handbook*. O'Reilly Media, 2016.