Introduction au Python

Une brève introduction orientée vers le calcul numérique
VERSION PROVISOIRE

Prof. Gilles Renversez

Aix-Marseille Université, France
gilles.renversez@univ-amu.fr

version 6

Contenu

1. Intro
2. Avantages
3. Environnement
4. Usages
5. Bases
6. Collections
7. Fonctions
8. Modules : survol
9. Retour sur le module Numpy : l'essentiel
   - les bases
   - Fonctionnalités intermédiaires
   - Les entrées/sorties

Introduction au Python I
Sources

Présentations ou documents issus de :

- Guido van Rossum (Google), inventeur du Python en 1989
- Matt Huenerfauth (Penn State University)
- Richard P. Muller (Caltech)
- Collectif de l'Université de Harvard
- Stefano Facchini (AMU)
- Site web `https://docs.python.org/3/tutorial`

Python I
Principaux avantages

- Langage open-source généraliste
- Orienté objet, fonctionnel, procédural
- Facile à interfacer avec le C/C++/Java/Fortran
- Environnement intéractif performant et accessible (langage semi-interprété)
- Multi-plateforme : Linux, MacOS, Windows, Solaris
- Documentation abondante
- Nombreux modules complémentaires (librairies) disponibles dans les distributions standards, relatifs à une multitude de domaine **https ://docs.python.org/3/py-modindex.html**
- Syntaxe : simple, claire, minimale, explicite, uniforme

Python I
Autres caractéristiques

- Typage dynamique (le type d'une expression est déterminé au moment de l'exécution) par opposition au typage statique (e.g. C, Java)
- Site essentiel : **https ://www.python.org/**

Environnement GNU/Linux I

Notions de base sur les systèmes d'exploitation

- Définition : Un système d'exploitation est le logiciel qui gère l'ordinateur (la machine plus généralement) : la lecture/écriture sur le disques, le processeur, les périphériques, ...
- Utilisation : Il permet notamment à un utilisateur de contrôler l'ordinateur.
- Type : Il y a actuellement de deux grandes familles de systèmes d'exploitation : UNIX (ex : linux, True64Unix) et Windows (ex : XP).

Environnement GNU/Linux I

Systèmes de fichiers

Nous travaillerons avec le système d'exploitation GNU/Linux, bien adapté au développement de programmes en Python.
On contrôle le déroulement des opérations soit par l'intermédiaire d'un bureau (*desktop*) soit directement en tapant des lignes de commandes dans une fenêtre de terminal (*shell*).
Nous utiliserons les logiciels `spyder3` ou `emacs` pour éditer les fichiers.

Le système de fichiers

- Un fichier est tout simplement une suite d'octets (une suite donnée de cases de la mémoire ayant certaines valeurs).
- Le système d'exploitation ne fait aucune interprétation du contenu d'un fichier, seul le programme qui produit ou exploite le fichier l'interpréte.

Environnement GNU/Linux II

Systèmes de fichiers

On distingue 3 sortes de fichiers :

1. **Un répertoire** ou catalogue (ou en anglais directory) est un fichier dont le contenu est une liste de noms de fichiers et leurs caractéristiques. Dans cette liste, il peut y avoir des répertoires. . .
2. Un fichier spécial est fichier servant pour la gestion des entrées/sorties (peu ou pas utiliser lors de cet enseignement).
3. **Un fichier ordinaire** est un fichier qui n'est ni un répertoire ni un fichier spécial.

Les règles de constructions des noms de fichiers sont les suivantes pour les systèmes de type UNIX :

1. La manière de nommer les fichiers dans tous les systèmes de type UNIX utilise **la structure arborescente des répertoires**.

Environnement GNU/Linux III
Systèmes de fichiers

2. Tout nom de fichier figure dans un répertoire qui figure à son tour dans un autre répertoire, et ainsi de suite jusqu'au répertoire **racine**.
3. Ce répertoire est le premier répertoire du système de fichier, on le note /.
4. On fait référence à un fichier particulier en écrivant tous les noms de répertoires entre lui et le répertoire racine, séparé par des '/'.

Exemple :

```
/var/home/l3p58/TD1/exo1-TD1.c
```

Environnement GNU/Linux IV

Systèmes de fichiers

On dit alors que `/var/home/l3p58/TD1/` est le **chemin absolu** qui mène au fichier source `exo1-TD1.c`

Le répertoire courant :

Pour alléger les références aux fichiers, un répertoire particulier est distingué comme **répertoire de travail** ou **répertoire courant**.

On parle alors de **chemin relatif**.

Exemple :

Si le répertoire courant est `/var/home/` alors `l3p58/TD1/exo1-TD1.c` est une référence relative au fichier `/var/home/l3p58/TD1/exo1-TD1.c`, le chemin relatif étant `l3p58/TD1/`.

Remarques :

-le caractère '**.**' constitue une référence relative au répertoire de travail (un synonyme, un lien en quelque sorte).

Environnement GNU/Linux V
Systèmes de fichiers

-les caractères '**..**' constituent une référence relative au répertoire ancêtre ou père du répertoire courant.

Environnement GNU/Linux I

Commandes de base du terminal de commandes

L'utilisation directe de commandes demande un apprentissage, mais elle procure bien plus de possibilités et de souplesse que le simple clic de souris. En particulier, elle permet de constituer des fichiers de commandes appelés *scripts* qui peuvent être de véritables programmes destinés au système d'exploitation.
Voici quelques commandes de base.

- mkdir `nom-repertoire` crée un nouveau répertoire (*make directory*)
- pwd affiche le nom du répertoire courant (*present working directory*)
- cd `nom-repertoire` pour aller dans le répertoire nommé (*change directory*)
- ls -l liste du contenu d'un répertoire (*long list*)

Environnement GNU/Linux II

Commandes de base du terminal de commandes

cp `source destination` copie d'un ou plusieurs fichier(s) ou répertoire(s) (*copy*)
Par exemple, `cp nom-du-fichier-1 nom-du-fichier-2` : copie le fichier `nom-du-fichier-1` sur le fichier `nom-du-fichier-2`.

rm `fichier` efface un ou plusieurs fichiers (*remove*)

rmdir `répertoire` efface un répertoire (*remove directory*)

mv `source destination` déplace un fichier ou un dossier ou change son nom (*move*)

touch `fichier-ordinaire` crée un fichier ordinaire s'il n'existe pas

Environnement GNU/Linux III

Commandes de base du terminal de commandes

man `com-commande` affiche la notice d'aide d'une commande. (*manual*)
Par exemple, taper `man cp` pour savoir comment utiliser la commande `cp`.

gcc `fichier-source.c` compile le fichier source indiqué avec le compilateur C par défaut dans les systèmes GNU/Linux.

python3 lance l'interpréteur de commandes de la branche 3 du python.

- Toutes ces commandes ont de nombreuses options possibles. Pour en avoir un résumé, rajouter `--help` après la commande envisagée.

Environnement GNU/Linux IV

Commandes de base du terminal de commandes

- La combinaison de touches *Ctrl-c* permet d'interrompre une commande (ou un programme) en cours d'exécution dans le même terminal.
- Les flèches permettent de rappeler la pile des commandes déjà exécutées et de l'éditer.
- De nombreuses autres commandes existent.
- Exemples
  Pour créer le dossier `TD1` dans le répertoire courant, vérifier sa présence puis entrer dans ce répertoire, on tapera donc successivement au clavier :

```
mkdir TD1
ls -l
cd TD1
```

Environnement GNU/Linux V

Commandes de base du terminal de commandes

- Aspect pratique : la complétion automatique
  Dans un terminal (et aussi dans `emacs`), la saisie des noms de fichier (ou de fonction) bénéficie du mécanisme de la complétion automatique : en tapant le début du nom, puis en appuyant sur la touche de tabulation [1], le logiciel tente de compléter pour vous le nom. Si le choix est unique (e.g. un seul répertoire commençant par ce nom), le nom est complété automatiquement, sinon, un bip retentit et vous avez le choix entre donner plus de caractères ou exercer une deuxième pression sur *Tab* pour faire apparaître la liste des choix possibles.

1. La touche sur la gauche du clavier avec les deux flêches tête-bêche.

Python I
Usages

- mode intéractif via l'interpréteur python
- mode script

Usage du Python I
Interpréteur

- Interface interactive au Python (en lançant la commande `python3` dans le terminal de commande)

```
toto@dupont:~$ 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
>>>
```

- Interpréteur Python évalue les entrées

```
>>> (4.5-8.5)+3**2
5.0
```

Le symbole `>>>` est le "prompt" ou l' "invite" du Python.
Pour sortir de l'interpréteur : on tape soit `Ctrl-D` soit `exit()`

Usage du Python I

Directement via le terminal de commandes

- `...:~$ python3 programme-python.py`
  exécute les instructions en Python du programme `programme-python.py`.

Bases du Python I

Un exemple simple de code Python

```
x = 2029 - 11   # Un commentaire
y = "No"   # Un autre commentaire
z = 2.1416
if z == 2.1416 or y == "No":
    x = x + 1
    y = y + " planet B" # concatenation de chaines de caract.
print(x)
print(y)
```

Programme : exemple-code-bases.py

Bases du Python I

Règles de codage

- Les affectations sont faites via =.
- Les tests d'égalité sont faits via ==.
- L'opérateur de test "différent de" est !=.
- Les opérateurs de comparaison sont <, <=, >, et >=.
- Pour les nombres, les opérateurs + - * / % fonctionnent comme attendu.
- Pour les chaînes de caractères, l'opérateur +, réalise la concaténation des chaînes et l'opérateur % est utilisé dans le formattage d'affichage des chaînes.
- Les opérateurs logiques sont des noms and or not, pas des symbôles.
- ; sert à séparer des instructions.

Bases du Python II
Règles de codage

- La première affectation d'une variable la crée.
  - Le type des variables peut ne pas être déclaré.
  - L'interpréteur Python déduit les types des variables non explicitement définis.
- La commande d'impression de base est print.

Bases du Python I
Types de base des variables

- Booléens, mot clé bool, 2 valeurs possibles : True et False

  prop1=(2!=3) *#operateur de test " different de"*

  prop2=(2<3) and (4**2!=16)
- Entiers, mot clé int

  x= 3+1 *#valeur de x = 4 en python 3.5, et x entier*
- Flottants (rationnels), mot clé float

  x= 1/3 *# valeur de x = 0.3333333... en python 3.5, et x flottant*

  y= 1.5
- Complexes, mot clé complex

  z=1.0+ 3.0j *# attention la notation j est la fin*

  z.imag==2 or z.real==1
- Chaînes de caractères, mot clé str

Bases du Python II

Types de base des variables

- Utilisation de " " ou de ' ' pour définir une chaîne `"azerty"` `'azerty'` mais
- `"le nombre d'australiens "` et
- Utilisation des triples guillemets pour des chaînes de plusieurs lignes ou qui contiennent à la fois " et ' *""" le nombre d'aliens "bleus" """*

- L'instruction type affiche le type de l'objet en argument :

```
In [62]: type(3/2)
Out[62]: float
In [63]: c="CO2"; type(c)
Out[63]: str
In [64]: type(z)
Out[64]: complex
In [65]: type(prop1)
Out[65]: bool
```

Bases du Python I
Rôle des espaces

L'espace joue un rôle en python : via l'indentation et le positionnement des sauts de ligne.

- Utilisez un saut de ligne pour terminer une ligne du code
  - Utilisez \ pour aller à ligne suivante prématurément.
- Ne pas utilisez d'accolades { } pour définir des blocs d'instruction, utilisez une indentation consistante
  - La première ligne avec *moins* d'indentation est en dehors du bloc.
  - La première ligne avec *plus* d'indentation débute un bloc imbriqué.
- Souvent : apparaît à la fin de la première ligne d'un nouveau bloc (exemple pour les fonctions et pour les définitions des classes)

Bases du Python I
Les commentaires

- Les commentaires débutent avec #, ce qui suit sur la ligne est ignoré par l'interpréteur.
- Dans les fonctions, une mini-documentation peut être ajoutée en tant que première ligne de la fonction ou de la classe dont la définition suit :

```
# def est le mot clef pour definir une fonction
def ma_fonction(x): # noter les : a la fin de cette ligne
    """Ceci est le texte de la mini-documentation de la fonction. Cette
    fonction retourne x.(x-3)^2"""
    print('Dans la fct')
    if (x==0):
        result=0
    else:
        result=x*(x-3)**2
    return result
```

Bases du Python II
Les commentaires

```
print('appel sans affichage')
ma_fonction(3.0)
print('appel avec affichage')
y=3.0
print(ma_fonction(y))
```

Exemple de fonction simple

Bases du Python I
Les affectations

- En Python, faire une affectation dans une variable correspond à définir **un nom pour référence à un objet** i=3
  - L'assignation crée un référence à un objet pas une copie.
- Les noms en Python n'ont pas de type intrinsèque, les objets eux ont un type.
  - Le Python détermine le type de la référence automatiquement en se basant sur l'objet donné qui lui est assigné.
  - On crée un nom la première fois qu'il apparaît du côté gauche d'une expression d'affectation i=3
  - Une référence est effacée par l'interpréteur Python (via la fonctionnalité "garbage collection") après que tous les noms qui lui sont liés se trouvent hors de portée (fin de la session, ou sortie de la fonction).

Bases du Python II
Les affectations

- Si vous tentez d'accéder à un nom avant qu'il n'ait été correctement créé, vous obtiendrez un erreur.

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

Erreur générée par la tentative d'accès à un nom pas défini, ici `t`, puis affectation correcte suivie d'un accès

Bases I

Règles de construction des noms de variables

- Les noms peuvent contenir des lettres, des chiffres, des blancs soulignés mais ne peuvent commencer par un chiffre. Les noms dépendent des majuscules et des minuscules.
- Les mots clefs sont réservés, ci-dessous une liste non-exhaustive :

  access, and, break, class, continue, def, del, elif , else, except, exec, finally , for , from, global, if , import, in , is , lambda, not, or, pass, print , raise , return , try , while

  Des fonctions pré-définies du Python et donc réservées existent aussi notamment :

  abs, all , any, basestring, bin, bool, bytearray, callable , chr, classmethod, cmp, compile, complex, delattr , dict , dir , divmod, enumerate, eval, execfile , file , filter , float , format, frozenset, getattr , globals, hasattr , hash, help, hex, id , input , int , isinstance , issubclass, iter , len , list , locals , long, map, max, memoryview, min, next, object, oct,

Bases II
Règles de construction des noms de variables

open, ord, pow, property, range, reduce, reload, repr, reversed, round, set,
setattr , slice , sorted, staticmethod, str , sum, super, tuple, type, unichr, unicode, vars
xrange, zip , apply, buffer , coerce, intern

Bases I
Comprendre la sémantique de référencement en Python

- Un assignation manipule des références
  - x=y n'effectue pas de copie de l'objet que la variable y référence
  - x=y fait que x référence l'objet que la variable y référence
- Très puissant mais . . .très subtil
- Example :

```
>>> A = [1, 2, 3]    # A reference la liste [1, 2, 3]
>>> B = A  # B reference ce que A reference
>>> A.append(4)    # ceci change la liste que A reference (ajout de 4)
>>> print(B)    # si on fait afficher ce que B reference
[1, 2, 3, 4]

# SURPRISE ! Cela a change.
```

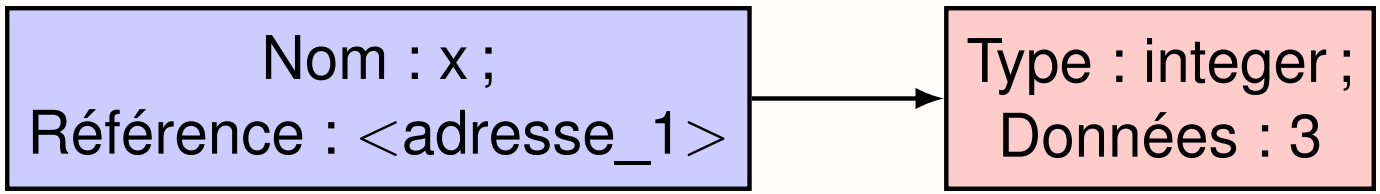
Exemple illustrant les subtilités du référencement en Python

Bases I

Comprendre la sémantique de référencement en Python : explications

Que se passe t'il quand l'interpréteur Python traite la ligne x=3 ?

1. Un entier 3 est crée et stocké en mémoire
2. Un nom x est crée
3. Une référence à l'emplacement mémoire stockant le 3 est alors assigné au nom x
4. Par conséquent, quand on dit que la valeur de x est 3, en fait on veut signifier que x référence maintenant l'entier 3 via son adresse (adresse_1 dans le schéma)

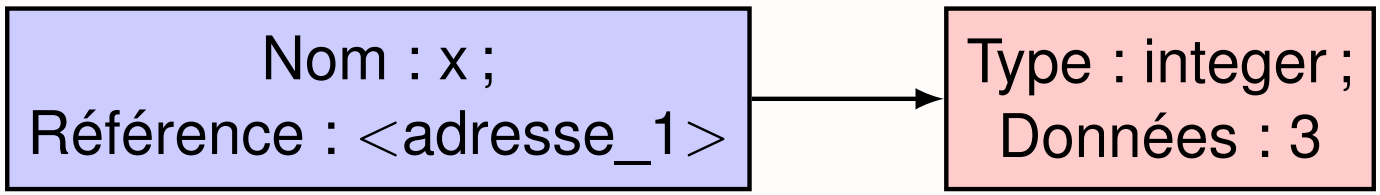


Bases

Comprendre la sémantique de référencement en Python : explication

- La donnée 3 créée est de type entier. En Python les types de base (int , float , str ,, complex, bool, tuple) sont non-modifiables ("immutable" en anglais technique)
- Ceci ne signifie pas que la valeur de x ne peut pas changer, on va simplement changer ce que x référence.

```
>>> x = 3
>>> x = x + 1
>>> print  x
4
```

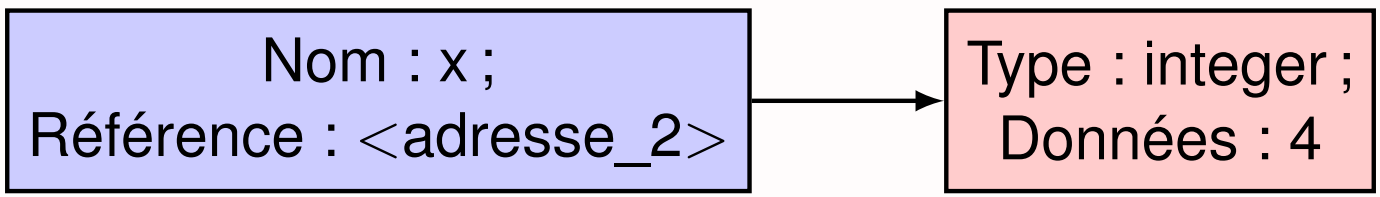


Bases

Comprendre la sémantique de référencement en Python : explication

- La donnée 3 créée est de type entier. En Python les types de base (int , float , str ,, complex, bool, tuple) sont non-modifiables ("immutable" en anglais technique)
- Ceci ne signifie pas que la valeur de x ne peut pas changer, on va simplement changer ce que x référence.

```
>>> x = 3
>>> x = x + 1
>>> print  x
4
```



Bases

Comprendre la sémantique de référencement en Python : première conclusion

- Pour les données de types simples (int , float , str), les assignations se comportent comme attendues :

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print(x,y)
3 4
```

Bases

Comprendre la sémantique de référencement en Python : première conclusion

- Pour les données de types simples (`int`, `float`, `str`), les assignations se comportent comme attendues :

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print(x,y)
3 4
```

- Mais, **pour les autres types de données** (liste, dictionnaire, types spécifiques définis par l'utilisateur dont ceux de `numpy`)**, les affections fonctionnent différemment.**
  - Ces type sont dits "mutable" par opposition au type "immutable"

Bases I

Comprendre la sémantique de référencement en Python : variable *mutable*

- Quand on change ces données, on effectue le changement sur place.
- On ne copie pas ces données dans une nouvelle adresse mémoire à chaque fois.
- Quand on exécute `b=a`, et que l'on modifie par la suite `a`, à la fois `a` et `b` sont modifiés.

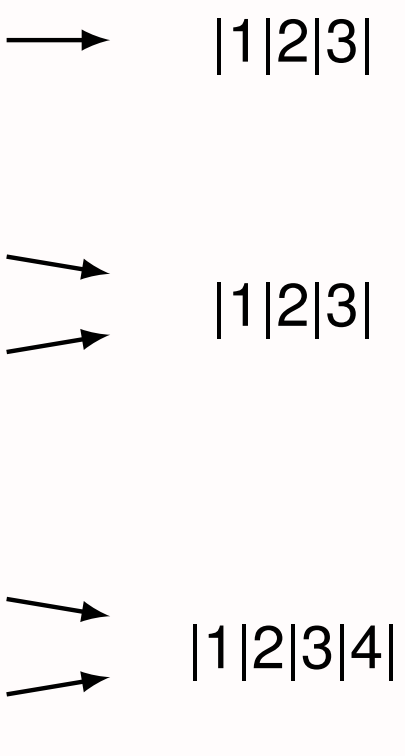
Bases II

Comprendre la sémantique de référencement en Python : variable *mutable*

Illustration :

```
In [18]: a=[1,2,3]
In [19]: type(a)
Out[19]: list
In [20]: b=a
In [21]: b
Out[21]: [1, 2, 3]
In [22]: a.append(4)
In [23]: a
Out[23]: [1, 2, 3, 4]
In [24]: b
Out[24]: [1, 2, 3, 4]
```

Interpréteur Python 3 : affectation pour une liste



Les objets de type collection I

Les 3 grands types

1. Tuple
   - Une simple collection/séquence ordonnée *immutable* d'objets
   - Les objets concernés peuvent être de types différents
   - Définit en utilisant des paranthèses et des virgules
   - Exemple : >>> tu= (10, `'SUV'`, 0.0, 99, `'CO2'`)
2. Chaîne de caractères (string en anglais)
   - *immutable*
   - Conceptuellement très proche d'un tuple
   - Définit en utilisant des simples guillemets ou des triples
   - Exemple : >>> st= `"Pas beaucoup de pistes cyclables dans cette ville"`
3. List
   - *mutable*
   - Une simple collection ordonnée *mutable* d'objets éventuellement de type différent

Les objets de type collection II

Les 3 grands types

- Définit en utilisant des crochets et des virgules
- Exemple : `>>> li = [10, 11, 'SUV', 0.0, 'CO2']`

Les objets de type collection I
Accès aux élements, indices, et copie

- On peut accéder aux éléments individuels de ces variables de type collection en utilisant des crochets (comme les tableaux), les indices débutant à 0 :

```
>>> tu[1]
'azerty'
>>> st[1]
'a'
>>> li [1]
11
```

- Indice positif : on compte à partir de la gauche, en partant de 0.

```
>>> tu[1]
'azerty'
```

Les objets de type collection II
Accès aux élements, indices, et copie

- Indice négatif : on compte à partir de la droite, en partant de -1.

  >>> tu[−2]

  99

- Extraction d'une partie contigue de la collection : le premier indice est celui du premier élément sélectionné et le second indice est celui de l'élément *suivant* le dernier sélectionné. En anglais, cette opération est nommée *slicing*.

  >>> tu[2:4]

  0.0 99

  - Si le premier indice est omis, l'extraction débute au début de la collection.
  - Si le dernier indice est omis, l'extraction s'achève à la fin de la collection.

Les objets de type collection III

Accès aux élements, indices, et copie

- Copie de l'intégralité d'une collection, l'instruction `li2 = li` et `li2 = li [:]` ne sont pas équivalentes : avec la première, les deux variables font référence à la même séquence (mêmes emplacements mémoires) alors que la seconde crée de nouvelles données :

```
In [28]: li = [10, 11, 'azerty', 0.0, 'qwerty']
In [29]: li2=li
In [30]: li3=li[:]
In [31]: li[0]=99
In [32]: li
Out[32]: [99, 11, 'azerty', 0.0, 'qwerty']
In [33]: li2
Out[33]: [99, 11, 'azerty', 0.0, 'qwerty']
In [34]: li3
Out[34]: [10, 11, 'azerty', 0.0, 'qwerty']
```

Illustration des différences de référencement

Les objets de type collection I
Opération sur les listes

Soit une liste ma_liste :

- ma_liste.append(valeur) : rajout d'une valeur supplémentaire en fin de la liste
- ma_liste.insert(indice,valeur) : rajout d'une valeur supplémentaire à l'indice donné en argument, puis décalage des autres éléments de la liste
- ma_liste.extend(valeur_1,valeur_2 ,..., valeur_N) : rajout de N valeurs en fin de la liste
- ma_liste.index(valeur) : indication de l'indice de la première occurence de la valeur fournie en argument dans la liste
- ma_liste.count(valeur) : comptage du nombre d'occurences de la valeur fournie en argument dans la liste

Les objets de type collection II
Opération sur les listes

- ma_liste.remove(valeur) : retrait de la liste de la première occurence de la valeur fournie en argument
- ma_liste.reverse() : inversion de l'ordre des éléments de la liste
- ma_liste.sort(fonction_de_tri) : trie les éléments de la liste en utilisant la fonction fournie en argument
- tuple(ma_liste) : transformation de la liste en collection de type tuple

Les fonctions I

Modularité et lisibilité des programmes

- L'instruction def crée une fonction et lui assigne un nom
- L'instruction return retourne un résultat à la partie ayant appelée la fonction concernée
- Les arguments sont passés pas affectation (transmission par valeur)
- Les types des arguments et du résultat ne sont pas déclarés

Syntaxe :

```
def <nom>(argu1, argu2, ...., arguN):
    <instructions>
    return <valeur>
```

Les fonctions II

Modularité et lisibilité des programmes

Exemple :

```
def ma_somme(x,y): # noter les " :" à la fin de cette ligne
    """Elle effectue une somme ou une concatenation
    en fonction des arguments."""
    return x+y

print(ma_somme(1,2))
print(ma_somme('aze','rty'))
```

Exemple d'une fonction avec 2 arguments et d'appels à la fonction

Les fonctions I

Passage des arguments d'une fonction

- Les arguments sont passés par affectation/par valeur
- Les arguments passés (leurs valeurs) sont assignés aux noms locaux dans la fonction
- Les affectations globales des noms locaux (dans la fonction) ne modifient pas les variables d'appel (dans la partie appelante)
- Le changement d'un argument de type *mutable* peut modifier la variable d'appel

Les fonctions II

Passage des arguments d'une fonction

1er exemple :

```
def change(x,y):
    """revoir les transparents sur la difference mutable/immutable."""
    x=1
    y[0]='attention'
    print("""dans 'change':""",x,y)
    return
x=0
y=['bonjour',' les terriens']
print(x,y)
change(x,y)
print(x,y)
```

Les fonctions III

Passage des arguments d'une fonction

Exemple d'une fonction avec changement de son argument d'appel

Résultat :

0 ['bonjour', ' les terriens']

dans 'change': 1 ['attention', ' les terriens']

0 ['attention', ' les terriens']

Les fonctions IV
Passage des arguments d'une fonction

2ème exemple :

```
def change2(x,y):
    """revoir les transparents sur la difference mutable/immutable."""
    x=1
    a=['bonsoir',' les martiens']
    y=a
    print("""dans 'change2 :""",x,y)
    return
x=0
y=['bonjour',' les terriens']
print(x,y)
change2(x,y)
print(x,y)
```

Les fonctions V
Passage des arguments d'une fonction

Exemple d'une fonction sans changement de son argument d'appel
Résultat :

0 ['bonjour', ' les terriens']

dans 'change2': 1 ['bonsoir', ' les martiens']

0 ['bonjour', ' les terriens']

Les fonctions
Autres règles importantes

- Toutes les fonctions en Python retournent une valeur y compris celles sans instruction return
- Dans ce cas, les fonctions retournent la valeur spéciale None
- Pas de surcharge de fonction en Python : deux fonctions différentes ne peuvent pas avoir le même nom même si elles ont des arguments différents
- Les fonctions peuvent être utilisées comme n'importe quel autre type d'objet. Les fonctions peuvent être :
  - des arguments de fonction
  - des valeurs de retour de fonction
  - assignées à des variables
  - des éléments de listes, tuples, ...

Modules : survol I

Introduction et importation des modules

- Les modules permettent d'utiliser des **fonctions pré-programmées**.
- Dans d'autres langages de programmation, ces modules sont appelés **librairies**.
- Il existe un très grand nombre de modules disponibles couvrant aussi bien le calcul scientifique, le traitement d'images, la gestion des fichiers de données, ...
- Il y a plusieurs manières, différentes en terme de fonctionalités, d'importer un module.

Modules : survol II
Introduction et importation des modules

- Nous utiliserons : import nom_du_module as alias_choisi
  qui permet de conserver une trace du module d'origine de la fonction utilisée. On procédera ainsi car des fonctions semblables (mais différentes dans les détails) peuvent exister dans différents modules. exemple : la fonction racine carrée sqrt.
  exemple : import numpy as np permet d'appeler la fonction numpy.exp simplement par np.exp.

Modules : survol I

Introduction et importation des modules

- Les modules permettent d'utiliser des **fonctions pré-programmées**.
- Dans d'autres langages de programmation, ces modules sont appelés **librairies**.
- Il existe un très grand nombre de modules disponibles couvrant aussi bien le calcul scientifique, le traitement d'images, la gestion des fichiers de données, ...
- Il y a plusieurs manières, différentes en terme de fonctionalités, d'importer un module.

Modules : survol II
Introduction et importation des modules

- Nous utiliserons : `import nom_du_module as alias_choisi`
  qui permet de conserver le module d'origine de la fonction utilisée.
  On procédera ainsi car des fonctions semblables (mais différentes dans les détails) peuvent exister dans différents modules, exemple la fonction racine carrée `sqrt`.
  Exemple : `import numpy as np` permet d'appeler la fonction `numpy.exp` simplement par `np.exp`.

Module : numpy, une très brève description I

Le module incontournable pour une utilisation efficace des tableaux

- tableaux 2D, multidimensionnels
- opérations d'algèbre linéaire
- manipulations rapides des tableaux
- fonctionnalités semblables à Matlab

```
import numpy as np
x = np.linspace(0, 2, 9)
print(`x=`,x)
print("nbre d'elts de x=",np.size(x))
y= -2*np.ones(9)
print('y=',y)
print("nbre d'elts de y",np.size(y))
z=x*y
print('z=',z)
print("somme des elts de z=",np.sum(z))
```

Module : numpy, une très brève description II

Le module incontournable pour une utilisation efficace des tableaux

Programme : exemple-numpy.py

```
x= [  0.  0.25 0.5  0.75 1.  1.25 1.5  1.75 2.  ]
nbre d'elts de x= 9
y= [−2. −2. −2. −2. −2. −2. −2. −2. −2.]
nbre d'elts de y= 9
z= [−0. −0.5 −1. −1.5 −2. −2.5 −3. −3.5 −4. ]
somme des elts de z= −18.0
```

FIGURE 1 – Résultats générés par le programme précédent

Module : matplotlib I

Le module privilégié pour les tracés de courbes et de figures

- réalisation assez rapide de figures de grande qualité

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2, 100) # 100 pts équiréparties entre 0 et 2 inclus
plt.plot(x, x,'+', label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.grid(True)
plt.savefig('exemple-simple.pdf')
plt.show()
```

Programme : exemple-matplotlib.py

Module : matplotlib II

Le module privilégié pour les tracés de courbes et de figures

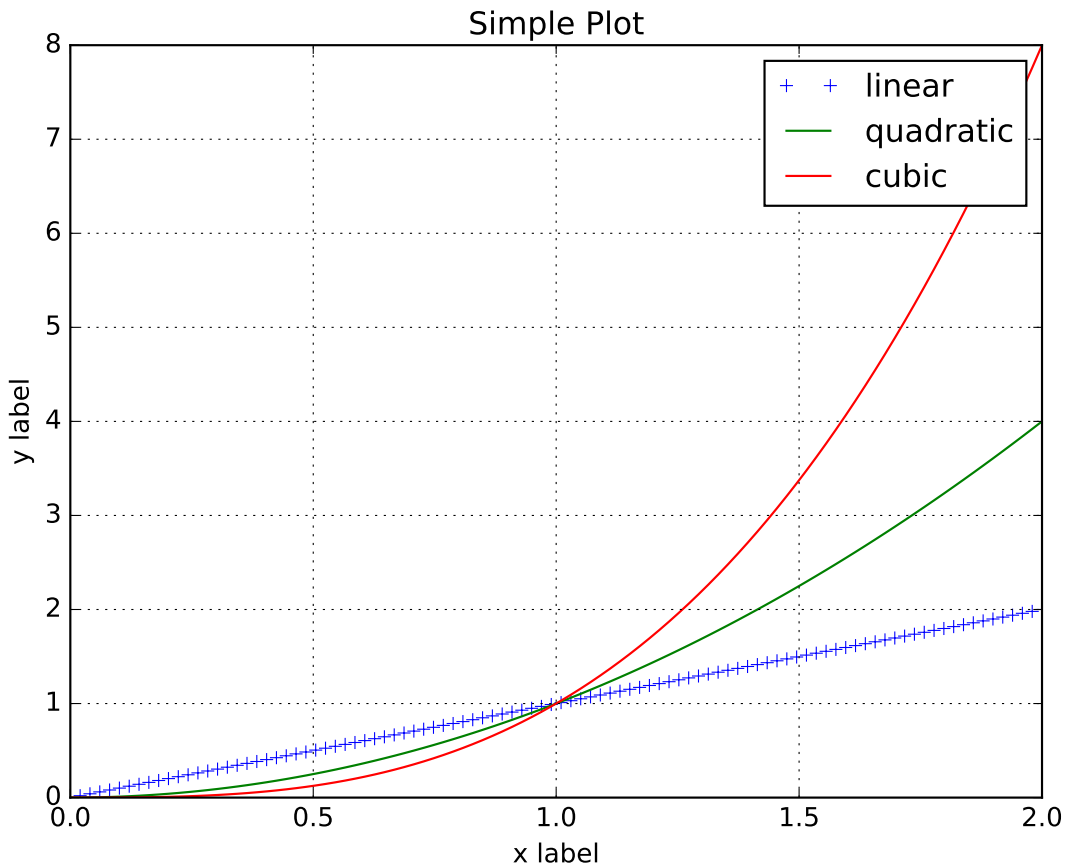


FIGURE 2 – Figure `exemple-simple.pdf` générée par le programme précédent.

Module : matplotlib III

Le module privilégié pour les tracés de courbes et de figures

Module : Numpy I
Les bases

Généralités :

- Le module Numpy permet de créer et de gérer des tableaux multidimensionnels homogènes, leur type est numpy.ndarray.
- Ils sont indexés par un objet de type tuple d'entiers positifs.
- Les dimensions d'un tableau Numpy sont appelées *axes*.
- Les tableaux de base du python ont moins de fonctionnalités attachées que ceux générés par Numpy.

Création d'un tableau Numpy

- manuelle : tab1 = np.array ([1,2,3],[4,5,6])
- automatique :

Module : Numpy II
Les bases

Création automatique de tableaux : exemple-numpy-bases-creat.py

```
import numpy as np
tab2 = np.zeros((3,4));print("tab2=",tab2)
tab3 = np.ones((2,3)) ;print("tab3=",tab3)
tab4 = np.arange(1,2,0.5) ;print("tab4=",tab4)
tab5 = np.arange(2,18).reshape(2, 8) ;print("tab5=",tab5)
tab6 = np.eye(2) ;print("tab6=",tab6)
tab7 = np.empty_like(tab3) ;print("tab7=",tab7)
tab8 = np.linspace(0.0,10.5,5) ;print("tab8=",tab8)
```

Module : Numpy III

Les bases

Affichage suite à l'éxécution de exemple-numpy-bases-creat.py

```
tab2= [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
tab3= [[ 1.  1.  1.]
 [ 1.  1.  1.]]
tab4= [ 1.   1.5]
tab5= [[ 2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17]]
tab6= [[ 1.  0.]
 [ 0.  1.]]
tab7= [[ 0.  0.  0.]
 [ 0.  0.  0.]]
tab8= [  0.     2.625   5.25    7.875  10.5  ]
```

Module : Numpy I
Les bases

- Le type des éléments peut être fixé lors de la création du tableau avec le mot clef `dtype` :

```
tab=np.ones((1,5),dtype=complex)
```

Module : Numpy I

Les bases : fonctionnalités agissant sur tout le(s) tableau(x)

- `tab2.size` fournit le nombre total d'éléments de `tab2`
- `tab2.ndim` fournit le nombre de dimensions de `tab2`
- `tab2.shape` fournit les dimensions de `tab2`
- `tab2.dtype` fournit le type des éléments de `tab2`
- `tab1.sum(axis=0)` fournit les sommes des éléments de chaque colonne de `tab1`, la sommation s'effectue selon l'axe `axis` fourni en argument
- `tab5.min(axis=1)` fournit les minima de chaque ligne de `tab5`
- `tab5.argmax(axis=1)` fournit les indices des maxima de chaque ligne de `tab5`
- l'opérateur `+` effectue l'addition élément par élément des tableaux mis en opérande

Module : Numpy II

Les bases : fonctionnalités agissant sur tout le(s) tableau(x)

- l'opérateur unaire `**2` effectue le carré élément par élément du tableau mis en opérande
- l'opérateur `*` effectue la multiplication élément par élément des tableaux mis en opérande
- l'opérateur `@` effectue la multiplication matricielle des tableaux mis en opérande
- les fonctions usuelles de numpy admettent en argument un tableau de type `np.ndarray` en effectuant l'opération requise élément par élément, exemple `np.exp(tab1)`
- la fonction `np.dot(a,b)` calcule le produit scalaire entre les deux vecteurs `a` et `b`
- l'appel à `np.all(tab1 == tab2)` permet de tester l'égalité globale, élément par élement, entre les tableaux `tab1` et `tab2`

Module : Numpy I

Les bases : indexation, découpage/extraction

- **tous les indices commencent à 0**
- `tab4[ind_deb:ind_fin+1:p]` extrait les éléments de `tab4` de l'indice `ind_deb` à `ind_fin` inclus avec un pas `p`
- `tab5[:,ind]` extrait la colonne d'indice `ind`
- `tab2[-1]` est le dernier élément du tableau `tab2`
- `tab2[-2]` est l'avant-dernier élément du tableau `tab2`, et ainsi de suite

Module : Numpy II

Les bases : indexation, découpage/extraction

- `tab2[tab2>5]` extrait les éléments de `tab2` (> 5) (`tab2` de type `float` ou `int`)
- `np.where(tab2>5)` renvoie le `tuple` associé aux valeurs sélectionnées vérifiant la condition alors que
- `np.where(tab2>5,tab2, -tab2)` renvoie un `nd.array` des valeurs sélectionnées vérifiant la condition et sinon les valeurs associées au 3ème argument

Module : Numpy I

Les bases : transformation, redimensionnement

- `tab5.ravel()` fournit le tableau aplati (sur une seule dimension)
- `tab5.T` fournit le tableau transposé (pour deux dimensions)
- `tab5.reshape(dim_1,dim_2)` fournit le tableau en un tableau de `dim_1` lignes par `dim_2` colonnes, le tableau original `tab` n'étant pas modifié.
- `tab5.resize(dim_1,dim_2)` transforme le tableau en un tableau de `dim_1` lignes par `dim_2` colonnes, le tableau original `tab` est modifié.
- `tab_a=np.append(tab_a,elt)` ajoute l'élément donné à la fin de `tab_a`.

Module : Numpy II

Les bases : transformation, redimensionnement

- `np.vstack((tab_a,tab_b))` fournit la concaténation, effectuée verticalement, des deux tableaux `tab_a` et `tab_b`. Les tailles des deux tableaux doivent être identiques.
- `np.hstack((tab_a,tab_b))` fournit la concaténation, effectuée horizontalement, des deux tableaux `tab_a` et `tab_b`

Manipulations de tableaux : exemple-numpy-stack.py

```
import numpy as np
tab_a = np.array([1, 2, 3])
tab_b = np.array([4, 5, 6])
tab_c = np.hstack((tab_a,tab_b)); print('tab_c=',tab_c)
tab_d = np.vstack((tab_a,tab_b)); print('tab_d=',tab_d)
tab_e = np.vstack((tab_a,tab_b)).T; print('tab_e=',tab_e)
```

Module : Numpy III

Les bases : transformation, redimensionnement

Affichage suite à l'éxécution de exemple-numpy-stack.py

```
tab_c= [1 2 3 4 5 6]
tab_d= [[1 2 3]
 [4 5 6]]
tab_e= [[1 4]
 [2 5]
 [3 6]]
```

Module : Numpy I

Les bases : copie de tableaux

Il y a copie de tableaux et vue de tableaux en python. Ces opérations peuvent sembler identiques au premier abord mais ce n'est pas le cas. On distingue les principales opérations suivantes conduisant à la création d'objets de type de tableau et associées à tort ou à raison à une copie de tableaux :

1. Une simple affectation via l'opérateur = n'est pas une copie : l'objet créé est simplement l'original.
2. Une vue (*view en anglais*) est un synonyme d'un tableau ou d'une partie d'un tableau. L'objet créé n'est pas l'original mais les données qui lui sont attachées sont communes. Si l'on change la vue on change l'original. On parle de *shallow copy* en anglais.
3. La vraie copie (*deep copy* en anglais) qui réalise une véritable duplication du tableau initial dans le nouveau.

Module : Numpy II

Les bases : copie de tableaux

Copies et vues : exemple-numpy-copie-vue.py

```
import numpy as np
tab_a=np.arange(8); print(tab_a)
tab_b=tab_a # assignation
print(tab_b is tab_a)
print(tab_b.flags.owndata)
tab_b.resize(2,4); print(tab_a.shape)
tab_c=tab_a.view() # vue
print(tab_c is tab_a)
print(tab_c.flags.owndata)
tab_c[0,0]=99; print(tab_a[0,0])
tab_d=tab_a.copy() # vraie copie ou deep copy
print(tab_d is tab_a)
print(tab_d.flags.owndata)
tab_d[1,1]=-1; print(tab_a[1,1])
```

Module : Numpy I

Les bases : copie de tableaux

Affichage suite à l'éxécution de exemple-copie-vue.py

```
[0 1 2 3 4 5 6 7]
True
True
(2, 4)
False
False
99
False
True
5
```

Module : Numpy I

Fonctionnalités intermédiaires impliquant les fonctions

- `numpy.fromfunction(fonction,forme,dtype=type_choisi)` construit un tableau de la forme indiquée en appliquant la fonction fournie en argument à chacun des indices du tableau.

```
>>>     def ma_fonction(x,y):
>>>     return 10*x+y+2

>>>     b = np.fromfunction(ma_fonction,(3,4),dtype=float
>>>     b
```

- `numpy.vectorize(fonction)` définit une version vectorisée de la fonction donnée en argument.

Module : Numpy II

Fonctionnalités intermédiaires impliquant les fonctions

```
>>>     def heaviside(x):
>>>       if (x>=0):
>>>         return 1.0
>>>       else:
>>>         return 0.0
>>>     heaviside_vect = np.vectorize(heaviside)
>>>     heaviside_vect(np.linspace(-1,1,10))
```

Module : Numpy I

Les entrées/sorties dans des fichiers

Le module `numpy` étant chargé par l'instruction `import numpy as np`, les appels de fonction suivants assurent les entrées/sorties avec des fichiers de type texte (par opposition au type binaire) :

- `colonne1,colonne2,...= np.loadtxt('mon_fichier_texte',skiprows=nbre_ligne_a_ignorer,unpack=True)`
  affecte aux tableaux de type `ndarray` `colonne1, colonne2,...` les colonnes correspondantes du fichier texte donné en argument
- `np.savetxt('mon_fichier_texte',tableau_a_sauver)`