

UED SIM – SysIn

Séance TD : Prise en main de l'environnement de développement *Python*[™]

Jean-luc Charles (jean-luc.charles@ensam.eu)

Éric Ducasse (eric.ducasse@ensam.eu)

Préambule

- Créer un répertoire TD-PEMP dans votre espace de travail
- Télécharger dans ce répertoire les documents du TD proposés par la plateforme E-Learning ENSAM.

Objectifs SysIn

Utiliser un langage moderne de programmation supportant l'**approche procédurale** et l'**approche Orientée Objet** pour :

- ✓ connaître et savoir utiliser les principaux types d'objets (scalaires, listes, chaînes de caractères...)
- ✓ connaître et savoir utiliser les structures de contrôle (tests, boucles...)
- ✓ maîtriser l'utilisation des fonctions (approche procédurale)
- ✓ comprendre et savoir utiliser classes et objets (approche orientée objet)
- ✓ maîtriser l'utilisation d'outils courants de traitement de données (manipulation des nombres et des chaînes de caractères)

Le langage proposé pour la mise en oeuvre informatique est *Python*TM (<http://www.python.org>) :

- simple à prendre en main,
- supportant la programmation procédurale et orientée objet,
- multi plate-forme (Linux, MacOS-X, Windows),
- complet, possédant de nombreuses bibliothèques (modules) de spécialisation,
- distribué sous licence libre,
- gratuit,
- ...

Ressources Python

- ✓ PC du SCI, ou votre ordinateur portable...
- ✓ plateforme E-learning ENSAM
- ✓ première séance TD (3h) : auto-formation en présence d'un enseignant
- ✓ suivi de plusieurs séances de TD (3h)
- ✓ bibliothèque : livres...
- ✓ auto-formation sur InterNet :
 - docs.python.org/release/2.7.2/library/index.html...
avec en particulier les *builtin functions*
 - Document "Prendre en main Python sous Windows" de Guillaume Duriaud,
guigui.developpez.com/Tutoriel/Python/PythonWindows
 - Document "Apprendre à programmer avec Python" de Gérard Swinnen,
python.developpez.com/cours/TutoSwinnen, chapitres 2 à 7.
 - fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-python
 - python.developpez.com/cours/?page=DocGeneral
 - python.developpez.com/livres
 - ...

Séance de prise en main Python : Objectifs

- ① savoir utiliser l'environnement intégré **IDLE**
- ② savoir utiliser les objets de base (scalaires, listes...)
- ③ savoir utiliser les chaînes de caractères et les principaux traitements de la classe **str**
- ④ savoir utiliser les structures de contrôle (tests, boucles...)
- ⑤ savoir définir et utiliser des fonctions (approche procédurale)
- ⑥ savoir utiliser des modules Python (math, numpy...)
- ⑦ savoir écrire/utiliser un algorithme
- ⑧ savoir écrire un programme Python et l'exécuter sous IDLE.

Les activités pédagogiques 2013-2014 requièrent l'installation de **Python2.7**

GNU/Linux

Installation de Python et IDLE

Installer les paquets **python2.7**, **python-numpy**, **python-matplotlib**, **python-scipy**, **python-qt4**, **idle** et **spyder**, avec l'installateur graphique de votre distribution, ou en tapant les commandes :

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

Tester l'installation en tapant la commande `idle` dans un terminal : la fenêtre graphique IDLE doit s'ouvrir.

Windows

Installation de Python et IDLE

Installer la distribution "Python scientifique" **Python(x,y)**, téléchargeable sur :

- ou [www.lamef.bordeaux.ensam.fr/~jlc/Python/Python\(x,y\)-2.7.3.1.exe](http://www.lamef.bordeaux.ensam.fr/~jlc/Python/Python(x,y)-2.7.3.1.exe)

Depuis les versions de PythonXY $\geq 2.7.3.1$, le lancement de IDLE se fait en utilisant le fichier de commande `C:\Python27\Lib\idlelib\idle.bat` :

- créer un "raccourci" avec un "clic droit" sur ce fichier, déplacer le raccourci sur votre bureau.
- tester l'installation en cliquant sur le raccourci IDLE : la fenêtre graphique IDLE doit s'ouvrir.

Les activités pédagogiques 2013-2014 requièrent l'installation de **Python2.7**

Mac OS X

Installation de Python et IDLE

Créer un compte avec votre adresse mail académique sur le site de la société **Enthought**

<http://www.enthought.com> :

- Télécharger et installer la distribution académique *Canopy Express Installer*, version 1.1 pour Mac OS X, version 32 ou 64 bits selon le CPU équipant votre ordinateur
- Télécharger le script de lancement [macIDLE32.sh](#) (ou [macIDLE64.sh](#) selon le CPU choisi)
- Ouvrir un terminal et taper la commande :
`sh Downloads/macIDLE32.sh` (ou `sh Downloads/macIDLE64.sh`) : la fenêtre graphique IDLE doit apparaître à l'écran
- avec le menu "Options>Configure IDLE", configurer la taille de la police de caractères pour avoir un rendu visuel satisfaisant.

Les élèves et les personnels du centre ENSAM Bordeaux peuvent aussi utiliser :

R:\Documents\Math-Info\UED-SIM\Ressources(CD-Mma Pythonxy etc)

PostInstallation Linux/Mac OS X/Windows

Installation de IDLEX



- numérotation des lignes,
- présentations des fichiers ouverts dans des onglets,
- gestion de l'historique des commandes avec le curseur,
- ...

```
1 # exemple simple:
2 message = 'Choisissez un nombre...'
3 reponse = raw_input(message)
4
5 nombre = int(reponse)
6
7 if nombre % 2 == 0:
8     print "nombre pair !"
9 else:
10    print "nombre impair !"
```

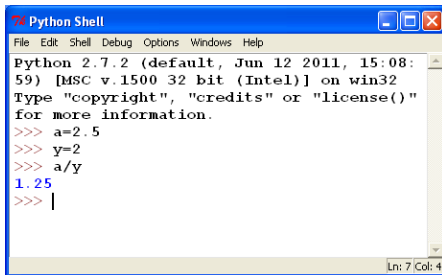
Installation :

- aller sur <http://idlex.sourceforge.net>
- page "Download and Run" : Télécharger l'archive zip et suivre les instructions de la page.

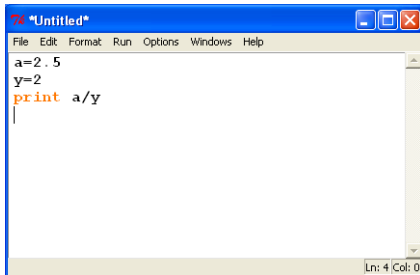
Environnement de développement simple : IDLE

IDLE est un IDE (Integrated Development Environment : environnement de développement intégré) simple qui ne comprend que 2 composants :

- un **interpréteur** Python (on parle aussi de **shell** Python), capable d'analyser et d'exécuter **immédiatement** des **lignes de commandes** écrites en langage Python dans la fenêtre de l'interpréteur,
- un **éditeur** *simple* permettant d'écrire des **programmes** en langage Python et de les faire exécuter simplement par l'interpréteur.



```
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=2.5
>>> y=2
>>> a/y
1.25
>>> |
```



```
a=2.5
y=2
print a/y
|
```

- la fenêtre interpréteur est très utile pour essayer des commandes Python, vérifier la syntaxe, comprendre le fonctionnement, tester des idées...
- un programme écrit dans la fenêtre éditeur peut être envoyé très simplement à l'interpréteur en appuyant sur la touche F5 (ou menu *Run > Run Module*).

IDLE



Sous Linux, MacOS-X ou Windows, lancer le logiciel IDLE...

Accéder à la doc Python depuis IDLE : menu *Help > Python_Docs (F1)*

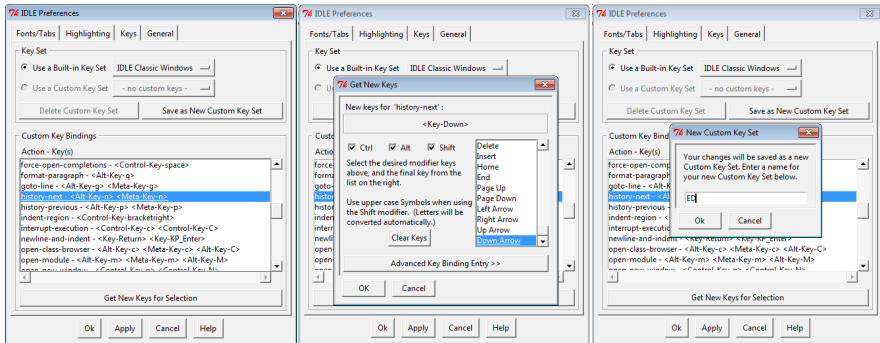
Taper quelques expressions Python simples dans l'interpréteur python pour *se faire la main* :

```
>>> a=2
>>> a+3
>>> range(5)
>>> help(range)
>>> y=range(6)
>>> len(y)
>>> type(y)      # y est un objet de type 'list'
>>> dir(y)       # voir les méthodes utilisables sur un l'objet y
>>> y
>>> y[0]
>>> y[5]
>>> sum(y)       # sum() est une fonction intrinsèque ( builtin)
>>> cos(3.14)    # cos, sin ... ne sont pas dans le langage de base.
                  # Les messages d'erreur Python sont verbeux : noter la dernière ligne !
```

Configuration des touches clavier pour **IDLE** (plateforme Windows)



Configurer les touches du clavier   pour naviguer dans l'historique des commandes tapées :

- 1 Menu *Options* > *Configure IDLE* puis onglet *Keys*
Cocher *Use a Built-in Key Set* et choisir *Idle Classic Windows*
Dans *Custom Key Bindings* sélectionner *history-next*, puis cliquer *Get New Keys for Selection*
- 2 Choisir *Down Arrow* dans la fenêtre *Get New Keys*, puis OK
- 3 La fenêtre *New Custom Key Set* apparaît : vous pouvez utiliser vos initiales...puis OK

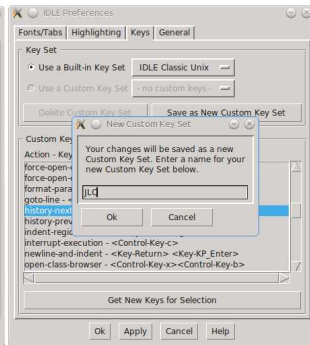
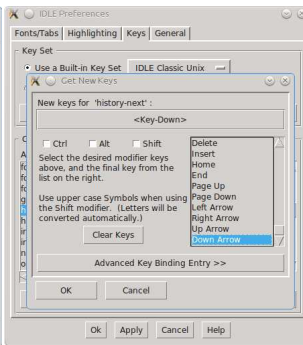
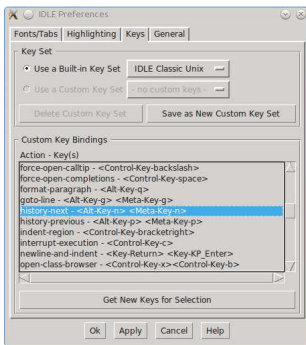


- 4 idem pour *history-previous*, à associer à la touche *Up Arrow*

Configuration des touches clavier pour **IDLE** (plateforme Linux)



Configurer les touches du clavier   pour naviguer dans l'historique des commandes tapées :

- 1 Menu *Options>Configure IDLE* puis onglet *Keys*
Cocher *Use a Built-in Key Set* et choisir *Idle Classic Unix*
Dans *Custom Key Bindings* sélectionner *history-next*, puis cliquer *Get New Keys for Selection*
- 2 Choisir *Down Arrow* dans la fenêtre *Get New Keys*, puis OK
- 3 La fenêtre *New Custom Key Set* apparaît : vous pouvez utiliser vos initiales...puis OK

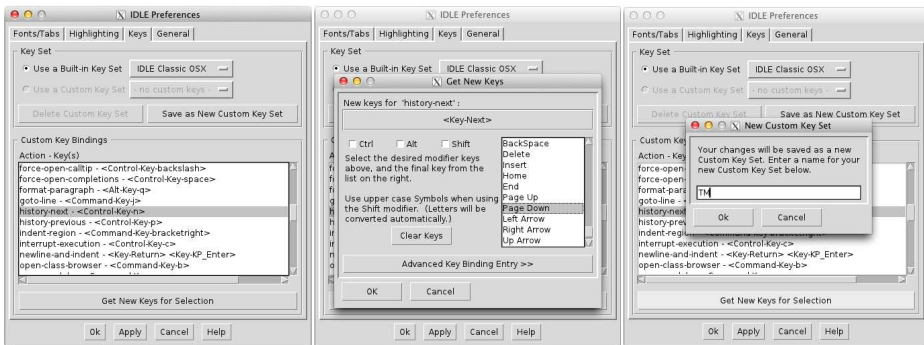


- 4 idem pour *history-previous*, à associer à la touche *Up Arrow*

Configuration des touches clavier pour **IDLE** (plateforme Mac OS X)

Configurer les touches du clavier   pour naviguer dans l'historique des commandes tapées :

- 1 Menu *Options* > *Configure IDLE* puis onglet *Keys*
Cocher *Use a Built-in Key Set* et choisir *Idle Classic OS X*
Dans *Custom Key Bindings* sélectionner *history-next*, puis cliquer *Get New Keys for Selection*
- 2 Choisir *Down Arrow* dans la fenêtre *Get New Keys*, puis OK
- 3 La fenêtre *New Custom Key Set* apparaît : vous pouvez utiliser vos initiales...puis OK



- 4 idem pour *history-previous*, à associer à la touche *Up Arrow*

A – Les objets de base

Builtin types (types intrinsèques livrés avec le langage Python) :

- les nombres : 1234, 3.1416, 1.4e-12, 3+4j... (non modifiables)
- les chaînes de caractères (*strings*) : "Hello", "Jack's home", 'Yes' (non modifiables)
- les listes : homogènes [1,2,3,5,7] ou hétérogènes [1,"two",[3,'Four'],5.0]
- les dictionnaires : { 'nom' : 'Durand', 'tel' : '06233456', 'note' : 18 }
- les tuples : (1,'Yes',4) ("listes" non modifiables)
- les fichiers : myFile=open('data.txt','r')
- les ensembles (*sets*)
- ...

En Python, les objets sont typés automatiquement d'après leur contenu !

Les nombres, chaînes de caractères et tuples sont des types **non mutables** : on ne peut pas les modifier !

Les chaînes de caractères, listes, tuples sont des **séquences** que l'on peut parcourir facilement avec des boucles (on parle aussi d'**objet itérable**).

A – Les objets de base

Essayer des expressions simples :

```
>>> 2/5          # Python 2.7 => division entière !!!!
>>> 2./5
>>> 5/4          # Python 2.7 => division entière !!!!
>>> 5./4
>>> 5//4         # // : floor division => troncature
>>> a=2;print a
>>> a
...
```

Visiter les autres opérateurs arithmétiques, par exemple :

```
>>> 123+45*2
>>> 13%3         # % : modulo
>>> 2**10
>>> 2**100       # le 'L' à la fin de l'affichage signifie 'long'
>>> 2**1000
>>> a=3;b=4;
>>> b/(2+a)
>>> b/(2.+a)
>>> print b/(2.+a)
...
```

B – Les Listes

Une liste est une collection d'objets (éventuellement hétérogènes), que l'on peut parcourir (concept de **séquence**)

```
>>> L=[1,2,3,4]      # les crochets indiquent le type liste !
>>> len(L)
>>> L[0]              # les listes commencent au rang zéro
>>> L[1]
>>> L[4]              # rang en dehors de la liste!
>>> L[len(L)-1]
>>> L[-1]             # un rang négatif parcourt la liste en sens inverse !
>>> L[-2]
>>> L[1:2]            # plage avec un ':' L[début:fin_exclue]
>>> L[1:4]
>>> L[1:len(L)]
>>> L[:3]
>>> L[1:]
>>> L[:-1]
>>> L[:]
>>> L[::2]            # plage avec deux ':' -> L[début:fin_exclue:pas]
>>> dir(L)             # que sait faire une liste ?
>>> range(5)           # génère la liste : [0,1,2,3,4] (5 exclu !)
>>> range(1,5)         # <=> Mathematica : Range[4]
>>> L1=L+["a","b"]     # + permet de concaténer (mettre bout à bout) des listes
>>> L=range(20)
>>> L[6:-2:2]
>>> L[::-1]
>>> L[-1:0:-1]
>>> L[10:5:-2]
```

B – Les Listes

Essayer aussi les méthodes de la classe **list** : `append()`, `pop()`, `reverse()`, `sort()`...

```
>>> u=[]
>>> dir(u)           # que sait faire l'objet u ?
>>> u.append(67)
>>> u
>>> u += range(10)    # += ajoute à la liste
>>> u.sort()
>>> u
>>> u.reverse()
>>> u
>>> u.insert(5,"Yes")
>>> u
>>> ...
```

Les listes sont des **objets itérables (séquences)** => on peut les parcourir avec une boucle **for** :

```
>>> for i in L:           # après le caractère ':' les lignes qui suivent sont
    print i               # automatiquement indentée pour marquer le "bloc for".
                           # taper un return de plus finit le bloc for.
```

Python supporte le concept de *"list comprehension expression"* :

```
>>> [t for t in range(5)]      # <=> Mathematica : Table[t, {t,0,4}]
>>> range(1,11)                # <=> Mathematica : Range[10]
>>> [t**2 for t in range(1,11)] # <=> Mathematica : Table[t^2, {t,10}]
```


C – Les chaînes de caractères

Ce sont des *listes* de caractères ...

```
>>> s="Vert"
>>> s
>>> s+" or Green ?"  # + : concaténation
>>> s[0]
>>> s[4]
>>> s[len(s)-1]
>>> s[-1]
>>> s[-2]
>>> s[1:2]           # les caractères de rang de 1 à 2, 2 exclu
>>> s[1:4]
>>> s[1:len(L)]
>>> s[:3]
>>> s[1:]
>>> s[:-1]           # -1 : dernier caractère
>>> s[:]              # toute la chaîne
>>> s[::-1]
>>> s[0]="v"         # s est non modifiable !!
>>> type(s)
>>> dir(s)
```

Essayer des méthodes de la classe **string** : upper(), find(), replace()...

```
>>> s1=s.upper()
>>> print s,s1
>>> s.find("r")
>>> print "'r' est le",s.find("r")+1,"me caractère de ",s
...
```

Cette partie peut être abordée dans une deuxième lecture (pas au programme des objectifs de la diapo 5).

D – Les Tuples

Un tuple est un objet non modifiable composés d'éléments hétérogènes (un peu comme une liste, mais non modifiable!).

```
>>> t=(1,2,3,4)      # les parenthèses indiquent le type tuple !
>>> t[0]              # même mécanisme d'index que les listes
>>> t[3]
>>> t[3]=56           # Erreur : un tuple n'est pas modifiable
>>> u=1,2,3           # tuple implicite (objets séparés par des virgules)
>>> u
>>> type(u)
>>> a,b,c=u           # extraction des objets du tuple par affectation multiple
>>> z=(1,)            # création d'un tuple à un seul élément (noter la virgule)
>>> z
>>>
```

Quand on n'a pas besoin de modifier une séquence, il vaut mieux utiliser un tuple (optimisation mémoire/CPU et préservation du caractère non-mutable).

E – Les modules Python

Les modules Python étendent les fonctions, objets... disponibles dans de très nombreux domaines...

On peut charger un module avec un des mots-clé **import**, **from**. Exemples :

```
# syntaxe de base :
>>> import math           # chargement de TOUTES les fonctions/données du module math!
>>> x=math.sin(math.pi/3.) # => toute fonction, donnée du module math doit être
                           # préfixée par 'math.'

# syntaxe avec renommage :
>>> import math as m      # le module math est importé et désigné par "m"
>>> x=m.sin(m.pi/3.)      # => le préfixage devient plus court : 'm.'

# solution conseillée : importation sélective
>>> from math import sin, pi # import de sin et pi dans le module math
>>> x=sin(pi/3.)           # => utilisation directe de sin et pi, sans préfixe.
```

Le module **math**

- Importer le module "math" avec "m" comme alias (solution acceptable en raison de la faible taille du module math).
- Essayer la commande `dir(m)`...
- Essayer l'aide sur une des fonctions : `help(m.trunc)` par exemple...
- Essayer d'utiliser des fonctions du module ... dont `ceil()`, `floor()` et `trunc()`.
- Essayer d'utiliser des données : par exemple `pi` et `e`.

Le module **numpy**

- Importer le module **numpy** avec "np" comme alias (observer le temps de chargement !).
- Essayer `dir(np)`...
- Essayer les fonctions du module ... dont `mean()`, `std()`, `sum()`...

```
>>> L=range(11)
>>> np.mean(L)      # fonction moyenne du module numpy importé avec l'alias 'np'
>>> np.sum(L)       # fonction somme du module numpy
>>> np.std(L)       # fonction écart-type (Standard deviation) du module numpy
...
```

- Essayer la fonction `uniform()` du sous-module `random` du module `numpy` :

```
>>> np.random.uniform() # un tirage aléatoire uniforme dans [0,1]
>>> np.random.uniform() # un autre tirage aléatoire uniforme dans [0,1]
>>> [np.random.uniform() for t in range(10)] # une liste de 10 nombres aléatoires
...
```

- Remarque : si l'on veut utiliser la fonction `random` sans le préfixe du module, on peut utiliser la syntaxe **from ...moduleName... import ...objet...** :

```
>>> from numpy.random import uniform
>>> uniform()
>>> uniform(-5,5,20)    # 20 tirages uniformes dans l'intervalle [-5,5]
...
```

F – Les Fonctions

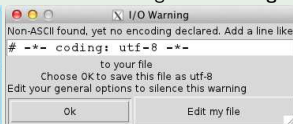
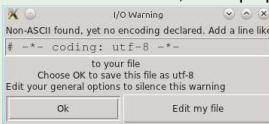
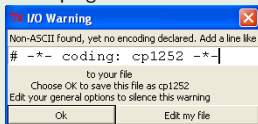
- Les fonctions sont définies avec le mot clef **def**, suivi d'un :
- une fonction est exécutée en invoquant son nom suivi des parenthèses '(' et ')'
- une fonction peut prendre **0, 1** ou **plusieurs** arguments (mentionnés entre les parenthèses)
- une fonction peut renvoyer des données, avec l'instruction **return** suivi des données à retourner.

Exemples :

```
>>> def ajouter(a,b):          # attention au caractère ':'
    return a+b                # Le corps de la fonction est un bloc indenté
>>>
>>> ajouter(5.7,9)            # appel de la fonction ajouter
>>>
>>> def Info():
    print("Hello")
>>>
>>> Info()                    # appel de la fonction Info, qui ne prend
                              # pas d'argument et qui ne retourne rien!
>>>
>>> def divide(a,b):
    q = a // b                # division entière
    r = a - q*b               # calcul du reste
    return (q,r)              # renvoi du tuple (q,r)
>>>
>>> divide(7,3)                # appel de la fonction divide
>>> a,b = divide(7,3)          # récupération du retour de divide
>>> a
>>> b
```

L'écriture de programmes Python

- Après la prise en main de l'interpréteur Python, il faut maintenant aller plus loin en écrivant des programmes Python avec l'éditeur proposé par IDLE.
Ces programmes seront ensuite exécutés par l'interpréteur.
- Le lancement de l'éditeur se fait avec le menu IDLE **File > New**
- Attention : dans un programme, on doit utiliser l'instruction **print** pour faire afficher le contenu d'un objet !
- En Python, les blocs ne sont pas notés entre accolades '{...}' comme avec les langages C, C#, C++... mais simplement marqués par une **indentation** (tabulation).
- L'exécution d'un programme créé dans l'éditeur IDLE se fait avec le menu **Run** (F5) :
 - IDLE sauvegarde le programme dans un fichier dont on doit donner l'**emplacement** et le **nom**
 - il FAUT inclure l'extension '.py' dans le nom du fichier (pour avoir la colorisation syntaxique!)
 - si le programme contient des **caractères accentués**, IDLE propose d'insérer une ligne d'**encodage**



⇒ pour tous les systèmes, cliquer sur **Edit my File**

⇒ Windows : il faut remplacer dans la première ligne du fichier **cp1252** par **utf-8**

- IDLE relance (restart) l'interpréteur pour exécuter le programme...
- en fin d'exécution du programme, on peut accéder dans l'interpréteur aux données du programme...

Premier programme Python : saisie clavier / sortie écran

La forme la plus simple de dialogue utilisateur-programme consiste à utiliser :

- l'entrée clavier pour entrer les données dans un programme :
l'utilisateur tape des touches du clavier en terminant par la touche <Entrée> (Return)
- la sortie écran (instruction print) pour afficher les données traitées par le programme.

L'entrée clavier peut se faire avec la fonction `raw_input("...message à afficher...")` :

- elle affiche le message à l'écran,
- elle retourne une **chaîne de caractères** contenant la séquence des touches tapées.

Éditer le programme suivant, prévoir ce qu'il fait... le lancer avec F5 :

```
while True :
    y = raw_input(u"Tapez des caractères au clavier puis <Entrée> : ")
    if y == "q" :
        print "(fin)"
        break
    else :
        print u"vous avez saisi <", y, ">"
```

En programmation, on utilise les sorties **formatées** pour maîtriser l'affichage des nombres :

```
>>> num = 1./30
>>> print num                # affichage par défaut de Python...
>>> print "%e" % num         # observer la différence...
>>> print "num = %15.7e" % num # format scientifique sur 15 caractères
                                # dont 7 chiffres après le '.'
>>> a, b = 1.234e-12, 56.78
>>> print "a=%15.6E et b=%15.6e" % (a,b)
```

Tests et opérations booléennes

Observer dans l'interpréteur la sortie des commandes suivantes :

```
>>> a =5                # affectation
>>> a == 5              # test 'égal à'
>>> a != 5              # test 'différent de'
>>> y = raw_input("Nombre ? ") # entrée par le clavier, avec affichage du message
                                # `"Nombre ?". Entrer la valeur '5'
>>> y == 5              # la réponse est False, pourquoi ?
>>> type(y)             # This is the reason why !!!
>>> float(y) == 5.      # maintenant, la réponse est True... of course! ;-)
>>> 2>3
>>> "abc"=="ab"
>>> 1<3 and 2<3        # and est l'opérateur logique "ET"
>>> not True            # not : opérateur logique "NOT", True : constante booléenne "vrai"
```

Programme : structures de contrôle avec les tests if

Avec l'éditeur IDLE, éditer le programme suivant :

```
def ask():                # fonction ask, attention au ':', puis bloc indenté
    y=raw_input("Nombre entier... ") # saisie clavier
    if int(y) % 2 == 0 :      # ':' est indispensable, puis bloc indenté
        print y, " est pair"  # indentation (tabulation) => corps du if
    else :                  # ':' est indispensable !
        print y, " est impair" # indentation (tabulation) => corps du if
    return
```

Exécuter

le programme avec F5, puis lancer ask() dans l'interpréteur...

Programme : structures de contrôle avec les boucles for et while

Les blocs des boucles sont également marqués par une indentation.

Éditer le programme suivant :

```
L = ['a', 1, [6, 7], "jour"]
i = 0
for e in L:                # attention au caractère ':' indispensable !
    print "item de rang " + str(i) + " de L : ", e
    i += 1
print
# autre façon (plus efficace) de balayer la liste,
# en générant automatiquement le rang i:

for i,e in enumerate(L): # enumerate() renvoie le rang et l'élément
    print "item de rang " + str(i) + " de L : ", e

print
# boucle while:
i = 0
while i < len(L):
    print "item de rang", i, "de L : ", L[i]
    i += 1
```

Exécuter le programme avec F5 dans l'interpréteur...

Autre exemple de programme

Écrire ce programme et l'exécuter :

```
from numpy.random import random_integers
a=random_integers(1,10)

print 'The computer has choosen a random integer number between 1 and 10'
print 'Can you guess it ?'
while True:
    y=raw_input('enter an integer (or type q to quit) ... ')
    if y == 'q':
        break;

    x=int(y)
    if x<1 or x>10:
        print 'Wrong value... try again'
        continue
    if x < a:
        print 'too small...'
    elif x > a:
        print 'too big...'
    else:
        print 'Good!'
        break

print 'Good bye...'
```

Exemple d'algorithme simple : calcul d'une fonction polynôme

Algorithme 1 : Calcul naïf de la valeur du polynôme P en t

1 **Function** $P(a, t)$

input : $a \rightarrow$ la liste des $n+1$ coefficients du polynôme de degré n ($a[0] \rightarrow a_0, \dots, a[n] \rightarrow a_n$)

input : $t \rightarrow$ la valeur en laquelle calculer le polynôme

output : La valeur du polynôme en t : $\sum_{i=0}^n a_i t^i$

2 $val \leftarrow a[0]$

3 $n \leftarrow \text{len}(a) - 1$

4 **for** $i \in [1, n]$ **do**

5 $val \leftarrow val + a[i] * t^i$ # 1 add., 1 mult., 1 puiss.

6 **return** val

Exemple d'algorithme simple : calcul d'une fonction polynôme

Algorithme 2 : Calcul plus malin de la valeur du polynôme P en t

7 **Function** P(a,t)

input : a → la liste des n+1 coefficients du polynôme de degré n ($a[0] \rightarrow a_0, \dots, a[n] \rightarrow a_n$)

input : t → la valeur en laquelle calculer le polynôme

output : La valeur du polynôme en t : $\sum_{i=0}^n a_i t^i$

8 val ← a[0]

9 n ← len(a) - 1

10 tpi ← 1.0 # tpi : t puissance i

11 **for** i ∈ [1,n] **do**

12 tpi ← tpi * t # 1 mult.

13 val ← val + a[i] * tpi # 1 add., 1 mult.

14 **return** val

Exemple d'algorithme simple : calcul d'une fonction polynôme

Algorithme 3 : Calcul plus malin de la valeur du polynôme P en t

15 **Function** P(a,t)

input : a \rightarrow la liste des n+1 coefficients du polynôme de degré n ($a[0] \rightarrow a_0, \dots, a[n] \rightarrow a_n$)

input : t \rightarrow la valeur en laquelle calculer le polynôme

output : La valeur du polynôme en t : $\sum_{i=0}^n a_i t^i$

16 val \leftarrow a[0]

17 tpi \leftarrow 1.0 # tpi : t puissance i

18 **for** c \in a **do** # c balaye la liste des coefficients

19 val \leftarrow val + c * tpi # 1 add., 1 mult. tpi \leftarrow tpi * t # 1 mult.

20 **return** val

Exemple d'algorithme simple : calcul d'une fonction polynôme

Algorithme 4 : Algorithme de Horner

21 **Function** $P(a, t)$

input : $a \rightarrow$ la liste des $n+1$ coefficients du polynôme de degré n ($a[0] \rightarrow a_0, \dots, a[n] \rightarrow a_n$)

input : $t \rightarrow$ la valeur en laquelle calculer le polynôme

output : La valeur du polynôme en t : $\sum_{i=0}^n a_i t^i$

22 $val \leftarrow$ dernier élément de a

23 **for** $c \in [a_{n-1}, a_{n-2}, \dots, a_0]$ **do**

24 $val \leftarrow c + t * val$ # 1 add., 1 mult.

25 **return** val

Les exercices qui suivent proposent la réalisation de générateurs pseudo aléatoires discrets ou continus, utilisant diverses méthodes de simulation.

Les solutions Mathematica sont proposées : elles vous permettront de formuler un algorithme de résolution, à **implémenter en Python**...

1 - Tirage selon une loi géométrique

Simulation de l'expérience - type (premier succès obtenu)

Loi de Bernoulli de paramètre p : 1 (succès, probabilité p) ou 0 (échec, probabilité $q = 1 - p$).

Mathematica

```
In[13]:= TirageLoiBernoulli = If[RandomReal[] < #, 1, 0] &;  
In[14]:= Table[TirageLoiBernoulli[0.4], {15}]  
Out[14]= {1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0}  
In[15]:= Table[TirageLoiBernoulli[0.4], {15}]  
Out[15]= {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0}
```

Écrire un programme python :

- qui importe la fonction `uniform` du sous module **numpy.random**
- qui définit la fonction `TirageBernoulli(x)` (s'inspirer de la solution Mathematica...)

Exécuter avec F5, et essayer plusieurs fois de suite l'appel `TirageBernoulli(0.4)`...

1 - Tirage selon une loi géométrique

Simulation de l'expérience - type (premier succès obtenu)

Loi géométrique de paramètre p :

Mathematica

```
In[21]:= TirageLoiGeometrique1 =  
Module[{NombreEssais},  
  NombreEssais = 1;  
  While[TirageLoiBernoulli[#] == 0, NombreEssais++];  
  NombreEssais  
&];  
  
In[22]:= ValeursSimulees = Table[TirageLoiGeometrique1[0.2], {15}]  
  
Out[22]:= {10, 2, 3, 9, 1, 6, 8, 2, 5, 4, 16, 2, 17, 3, 2}  
  
In[23]:= N[Mean[ValeursSimulees]]  
  
Out[23]:= 6.
```

Compléter le programme précédent python :

- ajouter la définition de la fonction TirageLoiGeometrique1 (s'inspirer de la solution Mathematica...)

Exécuter avec F5, appeler plusieurs fois TirageLoiGeometrique1(0.2)...

Calculer la moyenne d'une quinzaine de tirages ...

Exercices ...

Ouvrir le document Mathematica *Petits-Algos-pseudo-aléatoires.nb*, continuer les exercices "Mise en oeuvre de la méthode générale de tirage aléatoire d'une loi discrète" et "Tirage d'une loi continue, par la méthode du rejet"

1 - Tirage selon une loi géométrique : corrigé

Simulation de l'expérience - type (premier succès obtenu)

Loi de Bernoulli de paramètre p : 1 (succès avec une probabilité p) ou 0 (échec avec une probabilité $q = 1 - p$)

Mathematica

```
In[13]:= TirageLoiBernoulli = If[RandomReal[] < #, 1, 0] &;
In[14]:= Table[TirageLoiBernoulli[0.4], {15}]
Out[14]= {1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0}
In[15]:= Table[TirageLoiBernoulli[0.4], {15}]
Out[15]= {1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0}
```

Python

IDLE-éditeur

```
from numpy.random import uniform

def TirageBernoulli(x):
    if uniform() <= x: return 1
    else: return 0
```

IDLE-Interpréteur

```
>>> TirageBernoulli(0.4)
0
>>> TirageBernoulli(0.4)
1
>>> [TirageBernoulli(0.4) for i in range(15)]
[1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1]
>>> [TirageBernoulli(0.4) for i in range(15)]
[0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1]
```

1 - Tirage selon une loi géométrique : corrigé

Loi géométrique de paramètre p

Mathematica

```
In[21]:= TirageLoiGeometrique1 =
Module[{NombreEssais},
  NombreEssais = 1;
  While[TirageLoiBernoulli[#] == 0, NombreEssais++];
  NombreEssais
] &;

In[22]:= ValeursSimulees = Table[TirageLoiGeometrique1[0.2], {15}]

Out[22]= {10, 2, 3, 9, 1, 6, 8, 2, 5, 4, 16, 2, 17, 3, 2}

In[23]:= N[Mean[ValeursSimulees]]

Out[23]= 6.
```

Python

IDLE-éditeur

```
from numpy.random import uniform
from numpy import mean

def TirageBernoulli(x):
    if uniform() <= x: return 1
    else: return 0

def TirageLoiGeometrique1(x) :
    NbEssais=1
    while TirageBernoulli(x) == 0:
        NbEssais += 1
    return NbEssais
```

IDLE-Interpréteur

```
>>> V=[TirageLoiGeometrique1(0.2) for i in range(15)]
>>> V
[5, 5, 15, 4, 4, 2, 9, 9, 3, 12, 2, 1, 3, 2, 18]
>>> mean(V)
6.2666666666666666
```