Bases du langage Python - IG3

POLYTECH MONTPELLIER

1. Python : Introduction au langage

Introduction

Python est un langage interprété.

Les programmes sont lisibles et permettent de rapidement prototyper des solutions. Ceci pour plusieurs raisons :

- les types de données de haut niveau permettent de réaliser des opérations complexes en une seule instruction ;
- le regroupement des instructions se fait par indentation, ce qui facilite la lecture sans rallonger le code source par des `begin` ; end ou des `{ }` ;
- le *typage* et les *déclarations* sont *dynamiques*.

Python 2.7 or Python 3

- Caractéristiques de Python
  - facile à apprendre
  - portable
  - un grand nombre de librairies (packages)
- Python 3 depuis 2008!
- mais rétrocompatibilité non assurée, or beaucoup de librairies sont restées longtemps en 2.7
- plus cohérent, plus de possibilité : l'avenir
- 2.7 ne sera plus maintenu en 2020 et beaucoup de libraires ont maintenant basculé en Python 3

Python: comment l'utiliser ?

- Interactivement:
  - en ouvrant un terminal et en tapant la commande `Python`

```
>>>print "hello world"  #python 2.7
>>>print("hello world") #python 3.0
```

- Avec un fichier :
  - ouvrir un fichier texte, écrire des commandes python et le sauvegarder, puis...
  - taper dans un terminal : `python filename`
- Avec un fichier comme une commande
  - ouvrir un fichier texte, écrire sur la première ligne `#!/usr/bin/python`, et ensuite dans le terminal....
  - `chmod 755 filename`, et `./filename`

Syntaxe : généralité

- Structuration
  - les lignes se termine par une fin de ligne, pas de '`;`'
  - les blocs commencent par '`:`', sont marqués par une indentation `tab`, et sont terminés lors qu'il n'y a plus d'indentation

```
if true:
  print "hello"
print "bye"
```

  - packages

```
import sys # import package system
print "name of program " + sys.argv[0]
sys.exit(0)
```

2. Les éléments de base du langage

sous-titre

2.1 Variables

- les variables et les types
  - implicitement créées en les nommant simplement

```
x = 100    # variable entière x avec 100 pour valeur
y = 100.0 # variable décimale x avec 100 pour valeur
```

  - typage dynamique, cast dynamique

```
x = 100      # x est un entier
x = "hello" # x est une chaîne de caractère maintenant
```

Un type de variable peut donc muter au cours de l'exécution du programme !

2.2 Types

- Même si les types sont déterminés dynamiquement, les variables sont typées
- Types standards :
  - `int` pour les entier
  - `float` pour les nombres décimaux
  - `str` pour le texte (chaîne de caractères)
  - `bool` pour booléens (`True` ou `False`)
  - list, tuple, dictionary
- Déclaration explicite : `x = float(3)`
- calculs : `+, -, *, /, math.sqrt(), math.pow()`

Variables

En langage algorithmique

#déclaration
var nom : **`type`**
#définition explicite
var nom : **`type =`** valeur

En python :

nom = **`type`**(valeur)

2.3 Alternatives

Les *alternatives* ou *instructions de test* permettent de faire des choix en fonction des valeurs de données fournies au programme.

Ce choix dépend d'une condition évaluée comme une expression de type `bool` et aura donc une valeur `True` ou `False`. L'alternative s'écrit alors :

Python

```
if condition:
    action 1
else:
    action 2
```

Pseudo-code

```
if condition then
    action 1
else
    action 2
endif
```

Alternatives ou « tests »

En langage algorithmique

```
# alternative simple
if condition then
    liste d'instructions
endif

# alternative complète
if condition then
    liste d'instructions
else
    liste d'instructions
endif

# alternative complexe
if condition then
    liste d'instructions
elsif condition then
    liste d'instructions
else
    liste d'instructions
endif
```

En python :

```
# alternative simple
if condition:
    liste d'instructions

# alternative complète
if condition:
    liste d'instructions
else:
    liste d'instructions

# alternative complexe
if condition:
    liste d'instructions
elif condition:
    liste d'instructions
else:
    liste d'instructions
```

2.4 Itérations

En python, on itère sur une séquence

```
for i in T do # pour toutes les valeurs de T une séquence
  something
endfor
```

Si l'on a besoin de l'indice, il faut utiliser la fonction `range` la fonction `range` pour générer une séquence d'indices.

Définition : *fonction range*

La fonction range possède la syntaxe suivante :

```
range([start],fin,[pas])
```

Les arguments `start` et `pas` sont optionnels. `start` indique le début de la progression, par défaut 0. `fin` indique la fin de la progression. `pas` indique la raison de la progression.

Exemple

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

range(5,10)==[5,6,7,8,9]

range(0,10,3)==[0,3,6,9]

range(-10,-100,-30)==[-10,-40,-70]
```

Le fonctionnement d'une boucle pour utilisant « `i in range(a,b,r)` » est le suivant :

- au début de la première itération, le variable compteur de boucle `i` est initialisée à la valeur de début, c'est à dire ici `a` ;
- à la fin de chaque itération, la variable compteur de boucle `i` est incrémentée du pas, c'est à dire ici `r` ;
- une nouvelle itération est alors effectuée si la valeur de la variable compteur de boucle `i` est inférieure strictement à la valeur de fin `b`.

Ainsi pour la boucle suivante :

```
for i in range(a,b):
  séquence d'instruction
```

la séquence d'instructions est répétée `|b - a|` fois.

Itérations à nombre d'itérations borné

En langage algorithmique

```
# itération sur un ensemble
for var in Ensemble do
      liste d'instructions
endf

# itération par compteur de boucle
for i in a..<b do
      séquence d'instruction
endf
# ou
for i in a…b-1 do
      séquence d'instruction
endf

#itération par compteur de boucle
for i in a…b-1 step p do
      séquence d'instruction
endf
```

En python :

```
# itération sur un ensemble
for var in Ensemble:
      liste d'instructions

# itération par compteur de boucle
# de a inclus à b exclus
for i in range(a,b):
      séquence d'instructions

# itération par compteur de boucle
for i in range(a,b,p):
      séquence d'instructions
```

Itérations conditionnelles

En langage algorithmique

while *condition* **do**
    liste d'instructions
endw

En python :

while *condition*:
    liste d'instructions

En langage algorithmique

repeat
    liste d'instructions
until *condition d'arrêt*

En python :

while true:
    liste d'instructions
    if *condition d'arrêt*:
        break

3 Listes et séquences

3.1 Listes Python

En Python, la structure de donnée privilégiée est la liste.

Une liste est une structure de donnée dynamique qui permet d'ajouter ou retirer des éléments en fonction des besoins.

Elles ont généralement la particularité de ne permettre qu'un accès séquentiel aux données qu'elles contiennent.

Les listes python sont assez intéressantes puisqu'elles masquent toutes les difficultés d'implémentation des listes, utilisent la notation des tableaux ainsi que leurs avantages, c'est à dire un accès direct aux données stockées.

Comparaison Tableaux et Listes

Tableaux : en langage algorithmique

nom_tableau :[**Type**](n)
Pour obtenir la longueur (nombre d'éléments) du tableau, il faut faire appel à la fonction **len**

```
len(nom_tableau)
```

ou

```
nom_tableau.count
```

Listes : en langage python

nom_tableau =[]
Pour obtenir un tableau vide.

Il faut ajouter les éléments par l'appel à **append**
nom_tableau.append(3) par exemple

Pour obtenir un tableau à n éléments initialisés à 0, il faut faire :
nom_tableau = [0]*n

Pour obtenir la longueur (nombre d'éléments) du tableau, il faut faire appel à la fonction **len**

```
len (nom_tableau)
```

Exemple 4.1 (Utilisation d'une liste Python).

```
>>>T=['coucou','hello','bonjour',1,2,3]
>>>T
  ['coucou','hello','bonjour',1,2,3]
>>>T[0]
  'coucou'
>>>T[3]
  1
>>>T[-2]
  2
>>>T[1 :3]
  ['hello', 'bonjour']
>>>T[1 :-1]
  ['hello', 'bonjour', 1, 2]
>>>len(T )
  6
```

Exemple 4.2 (Modification d'une liste Python).

```
>>>T.append(4)
>>>T
  ['coucou', 'hello', 'bonjour', 1, 2, 3, 4]
>>>T[1 :1]=['Guten Tag']
>>>T
  ['coucou', 'Guten Tag', 'hello', 'bonjour', 1, 2, 3, 4]
>>>T[2 :4]=['hola','saluton']
>>>T
  ['coucou', 'Guten Tag', 'hola', 'saluton', 1, 2, 3, 4]
>>>T[2 :2]=['hello','bonjour']
>>>T
  ['coucou', 'Guten Tag', 'hello', 'bonjour', 'hola',
'saluton', 1, 2, 3, 4]
```

Exemple 4.3 (Modification d'une liste Python - suite).

```
>>>L=[]
>>>L
  []
>>>L.append(0)
>>>L
  [0]
>>>L[1]=1
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module> IndexError: list
assignment index out of range
>>>L[ :1]=1
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: can only assign an iterable
```

Quelques fonctions des listes Python


```
>>> l.append(x) #ajoute un élément à la liste
```

```
>>> l.insert(i, x) # insère un élément à la position i
```

l'élément x est inséré avant l'élément à la position i

```
>>> l.remove(x) # enlève la première occurence de l'élément x
```

```
>>> l.pop() # enlève et retourne le dernier élément de la liste
```

```
>>> l.clear() # efface tous les éléments de la liste
```

```
>>> l.sort() # trie la liste
```

```
>>> l.copy() # permet d'obtenir une nouvelle liste, copie de l
```

Initialisation des listes

On peut initialiser une liste en donnant ses valeurs :

```
l = [1,2,3,4,5]
```

Mais on peut aussi utiliser des expressions pour initialiser plusieurs valeurs en les calculant :

```
l = [ x for x in range(1,6) ]
```

Voir même en faisant des calculs :

```
l = [x*2 for x in range(1,6) ]
```

ou tout simplement

```
l = [0 for i in range(1,6)]
```

3.2 Piles et Files

On utilisera les listes et on se restreindra à certaines fonctions, ainsi...

pour les piles, on utilisera exclusivement les fonctions `p.append(x)` et `p.pop()` pour obtenir le fonctionnement d'une pile

pour les files, on utilisera exclusivement les fonctions `p.append()` et `p.popleft()` pour obtenir le fonctionnement d'une file.

3.3 Les Tuples


```
>>> T=('salut','hello','bonjour',1,2,3)
('salut','hello','bonjour',1,2,3)
>>> T[0]
'salut'
>>> T[1 :3]
['hello', 'bonjour']
```

les tuples semble être juste des listes mais avec des parenthèses à la place des crochets

Mais

- la taille des tuples est immuable (pas de append)
- les tuples ne peuvent pas être mis à jour (pas d'affectation)

3.4 Les string

En Python, une chaîne de caractères (`str`) est considérée comme une séquence

On peut donc les parcourir, ce qui peut être très utile pour analyser une texte :

```
for lettre in "Polytech":
  print(lettre)
#affiche 'P', 'o', 'l', 'y', 't', 'e', 'c', 'h'
```

On peut extraire une partie (comme pour les listes)

```
>>>s = "Polytech IG3"
>>>s[9:len(s)]
"IG3"
>>>s[4:8]
"tech"
```

On dispose également des opérations classiques sur les chaînes de caractères :

- On peut concaténer des chaînes :

```
>>> s = "Python" + " en IG3"
"Python en IG3"
```

On peut comparer des chaînes

```
>>> "IG3" == "MEA3"
False

>>> "IG3" < "MEA3"
True
```

Majuscules et minuscules

```
>>> s = "ig3"
>>> t = s.upper()
"IG3"
>>> u = t.lower()
"ig3"
>>> u.capitalize()
"Ig3"
```

Transcription de texte

```
>>> s ="python en ig3"
>>> traduction = str.maketrans('hnopty','f tsiw')
>>> s.translate(traduction)
"swift  e  ig3"
```

4. Les Dictionnaires Python

Dictionnaires Python

Les dictionnaires peuvent être vus comme des tableaux associatifs qui associent une valeur à une clef

Mais contrairement aux listes ou aux tableaux, il n'y a pas d'ordre défini entre les éléments

Les dictionnaires sont des séquences et l'on peut donc itérer sur les éléments, mais l'ordre peut être différent à chaque fois.

On peut itérer sur les

- valeurs
- valeurs et clés
- clés

d'un dictionnaire

Exemples

```
>>>d = {'pierre':12, 'jean':8, 'paul':19}
>>>d['pierre']
12
>>>for k in d:
>>>    print(k)
>>>#endfor
jean
pierre
paul
>>>for k,v in d.items():
>>>   print(k, v)
>>>#endfor
jean 8
pierre 12
paul 19
>>>for v in d.values():
>>>   print(v)
>>>#endfor
8
12
19
```

5. Les fonction

Les fonctions Python

En python, la notion de procédure n'existe pas. On ne peut définir que des fonctions.

Le schéma de déclaration d'une fonction est le suivant :

```
def nomFonction(paramètres):
  """string documentation """
  instructions
  …
```

Les paramètres sont passés par *valeur* mais pour les structures de données comme les *listes*, cette valeur est l'adresse de la liste. Le passage de paramètre se passe alors comme si il était par *adresse*.

Exemple 5.1 (Fonction de calcul de la suite de Fibonacci).

```
def fib(n):
  """Return a list containing the Fibonacci series up to n,
i.e une liste où chaque terme est la somme des deux termes
qui le précèdent."""
  result = []
  a, b = int(0), int(1)
  while a < n:
    result.append(a) # see below
    a, b = b, a+b# a+b évaluée avant a=b
  return result
```

Exemple 5.2 (Valeur par défaut des paramètres).

```
def demander_confirmation(message,nbessai=4,avertissement='Pour la
dernière fois, Oui ou Non?!'):
  reponse_faite = bool(False)
  reponseok = bool(False)
  reponse = Text('')
  while (not reponse_faite) and (nbessai> 0):
    if nbessai>1:
      reponse = raw_input(message)
    else:
      reponse = raw_input(avertissement)
    if reponse in ('o','oui','ok'):
      reponseok = True
      reponse_faite = True
    if reponse in ('n','non'):
      reponseok = False
      reponse_faite = True
    nbessai = nbessai-1
  if reponse_faite:
    return reponseok
  else:
    raise IOError('utilisateur récalcitrant')
```

Il est donc possible de spécifier des valeurs par défaut pour les arguments d'une fonction sous la forme d'une affectation dans la déclaration du paramètre.

La fonction `demander_confirmation` peut être appelée des façons suivantes :

- en ne donnant que le paramètre obligatoire :
  `demander_confirmation`('Confirmez-vous votre réponse?')
- en donnant le premier paramètre optionnel :
  `demander_confirmation`('Confirmez-vous votre réponse?',3)
- en donnant tous les paramètres :
  `demander_confirmation`('Confirmez-vous votre réponse?',3, 's'il vous plaît, répondez oui ou non')

- *Attention !* la valeur par défaut n'est définie *qu'une seule fois* au premier appel ;
  ceci a une conséquence pour les données structurées qui donc pourront ne pas garder la valeur par défaut au cours d'appels ultérieurs si celle-ci est modifié lors du premier appel :

```
def f(a,L=[]):
  L.append(a)
  return L
print f(1)
print f(2)
print f(3)

# Ceci donnera à l'exécution :
[1]
[1,2]
[1,2,3]
```

Fonctions

En langage algorithmique

```
func nomf(a: T1, b: T2) -> Tr
# données: a : T1 explications
#          b : T2 explications
# résultat  Tr explications
# déclaration/définition des variables locales
var v : Type
    liste d'instructions
    return résultat
end
```

Note: par défaut les paramètres de la fonction sont des paramètres de type donnée et donc le mot clef in n'est pas requis ; si l'on veut spécifier un autre mode de passage de paramètres, il faut utiliser les mots clefs **in** ou **inout**

par exemple :

```
func nomf(in a: T1, inout b: T2) -> T2
var v : Type
    liste d'instructions
    return résultat
end
```

En python :

```
def nomf(a,b):
# données: in a : T1 explications
#          in b : T2 explications
# résultat  Tr explications
    liste d'instructions
    return résultat
```

Fonctions et tableaux

En langage algorithmique

```
func nomf(a: [T1](n1), b: [T2](n2)) -> Tr
# données: a : tableau à n1 éléments de type T1
#          b : tableau à î éléments de type T2
# résultat  Tr explications
# déclaration/définition des variables locales
      liste d'instructions
      return résultat
end
```

Note: la taille d'un tableau est intrinsèque au type correspondant, un tableau passé en paramètre devra indiquer la taille et le paramètre sera donc considéré comme connu par défaut, même si il est toujours possible de connaître sa taille grâce à la propriété **len**.

par exemple :

```
func nomf(in a: [Int](n), inout b: [Int](m)) -> Int
    for i in 0..<n do
        do something
    endfor
endfunc
```

En python :

```
def nomf(a,b):
# données: a : tableau à n1 éléments de type T1
#          b : tableau à î éléments de type T2
# résultat  Tr explications
# déclaration/définition des variables locales
      liste d'instructions
      return résultat
```

Note: la taille d'une liste python est une propriété de la liste et peut être interrogée grâce à la fonction **len**.

par exemple :

```
def nomf(a, b):
    for i in range(len(a)):
        do something
```

Fonctions et tuples

En langage algorithmique

func nomf(a: **T1**, b: **T2**) -> (**Tr1, Tr2)**
    liste d'instructions
    return (*résultat1, résultat2*)
end

Note: une fonction peut retourner plusieurs valeurs en utilisant des tuples. On accède aux éléments des tuples par leur position :

```
t = nomf(a,b)
print(t.0+ ‘,’+t.1)
```

Ces tuples peuvent être nommées, par exemple :

func nomf(a: T1, b: T2) -> (first: Tr1, second: Tr2)

et on accède alors aux valeurs par leur nom

```
t = nomf(a,b)
print(t.first+ ‘,’+t.second)
```

on peut également affecter directement à 2 variables le résultat d'un tuple, par exemple :

```
t,u = nomf(a,b)
```

En python :

def nomf(a,b):
    liste d'instructions
    return (*résultat1, résultat2*)

Note: une fonction peut retourner plusieurs valeurs en utilisant des tuples. On accède aux éléments des tuples par leur position :

```
t = nomf(a,b)
print(t[0]+ ‘,’+t[1])
```

On peut également affecter directement à 2 variables le résultat d'un tuple, par exemple :

```
(t,u) = nomf(a,b)
```

Procédures

En langage algorithmique

Proc nomp(a: **T1**, b: **T2**)
données: a : T1 *explications*
b : T2 *explications*
déclaration/définition des variables locales
var v : **Type**
    liste d'instructions
end

Note: par défaut les paramètres de la procédure sont des paramètres de type donnée et donc le mot clef in n'est pas requis ; si l'on veut spécifier un autre mode de passage de paramètres, il faut utiliser les mots clefs **in** ou **inout**
par exemple :
Proc nomp(**in** a: T1, **inout** b: T2)
var v : **Type**
    liste d'instructions
end

En python :

def nomp(a,b):
données: in a : T1 *explications*
in b : T2 *explications*
    liste d'instructions

6. Entrée-Sorties

Saisie et impression dans un terminal
Fichiers

6.1 Print : affichage sur un terminal

Attention le format n'est pas le même en Python 2.7 et Python 3

Python 2.7

```
c = 'chevalier'
print 'Le {chev} qui dit {chev} Ni!'.format(chev=c)
```

Python 3

```
print('Le {c} qui dit Ni!')
```

Pour les entrées au clavier, même syntaxe : `input()`

```
s = input("entrez votre nom - n'oubliez pas les '\n")
```

6.2 Format de chaînes

On peut formater une chaîne de caractères en sortie :

Le caractère `'f'` devant une chaîne indique un format :

`f"l'{event} aura lien en {year}"` est une chaîne de caractères où `{event}` et `{year}` seront remplacées par l'évaluation des variables `event` et `year`.

On peut également utiliser la fonction str.format() qui permet de formater des nombres notamment :

```
f"il y a {:2.2}% d'élèves qui ont réussi sur une
promo de {:-3} élèves".format(preussite,neleves)
```

ou plus simplement

6.3 Lecture dans un fichier

Pour lire dans un fichier, il faut commencer par l'ouvrir :

la fonction `open()` renvoie un 'fichier' et on l'utilise généralement avec la clause `with` :

```
with open('data.csv','r') as file:
  data = f.read()
f.closed
```

Le deuxième argument de la fonction open indique le mode d'ouverture :

- `'r'` pour lire le fichier
- `'w'` pour écrire dans le fichier (écrase les données existantes)
- `'a'` pour append : ajoute au fichier

- `file.read([size])` : lit size (toutes) données du fichier
- `file.readline()` : lit une ligne d'un fichier texte - la chaîne retournée se termine par '\n'
- `file.write("chaine")` : écrite "chaine" dans le fichier

Exemple de lecture d'un fichier texte

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

file = open("data.csv","r",encoding="utf-8")

for line in file:
  print(line)
  data = line.split(",")
  for val in data:
  print(val)
```

7. Les packages

Comment faire des packages et quelques packages bien utiles

7.1 Créer un package

Un simple fichier python contenant du code et des fonctions peut toujours être importé dans un autre fichier (on parle alors de module) : `import tp1.py`

permet d'importer votre fichier `tp1.py` et toutes les définitions de fonctions s'y trouvant.

Le code hors fonction sera exécuté une fois : utile pour initialiser des données

Si vous exécutez un fichier prévu pour être un module, il est possible d'avoir du code spécifique :

```
if __name__ == "__main__":
  import sys
  print(sys.arv[1])
```

Un package est plus qu'un module : c'est un répertoire contenant plusieurs fichiers qui pourront être importés.

Dans ce cas, le nom du répertoire est le nom du package et il doit comporter un fichier `__init__.py`

Structuration d'un package

```
TP1
  __init__.py
  Utils
    module1.py
    module2.py
  module3.py
```

`TP1` est le package, Utils est un sous-package, `module*.py` sont les différents modules.

Le fichier `__init__.py` est nécessaire pour que Python considère un répertoire comme un paquet.

`__init__.py` peut contenir une doc du package

`__init__.py` peut contenir du code d'initialisation

`__init__.py` peut (doit) contenir la liste des fonctions qui seront importées lors d'une commande

```
from TP1 import *
```

Cette liste des fonctions est donnée dans la variable `__all__`

```
__all__ = ["prodintpair","sansdiviseur"]
```

Si `__all__` n'est pas définie, seuls les modules sont importés, cad seuls les `__init__.py` sont exécutés

7.2 Comment utiliser un package

Il faut indiquer à Python que l'on veut utiliser les fonctions d'un module ou d'un package par la commande `import`

```
import math
from math import mt
```

Ensuite on utilisera les fonctions en les préfixant du nom du package :

```
math.sqrt(2)
mt.sqrt(2)
```

On peut aussi importer individuellement des fonctions :

```
from math import sqrt
sqrt(2)
```

On peut aussi importer toutes les fonctions :

```
from math import *
```

7.3 Package Sys

Le package Sys permet d'accéder à certaines commandes du système, et en particulier d'accéder à la ligne de commande.

`sys.argv` est un tableau des arguments de la ligne de commande.

Attention `sys.argv[0]` est le nom du programme.

`len(sysargv)` permet de connaître le nombre d'argument (en comptant le nom du programme)

`sys.stdin, sys.stdout, sys.stderr` permet d'accéder aux entrées/sorties standards

7.4 Exécuter un programme externe

La manière la plus simple est d'utiliser la commande `system` du package `os` :

```
os.system('ls -l')
```

La commande est alors exécuté dans un shell

- Avantage : facilité d'utilisation
- Inconvénient : contrôle limité, la sortie est celle du processus python, pas de contrôle sur l'entrée, pas de maîtrise des entrées/sorties

Le package `subprocess` est à préférer.

Il permet de créer de nouveaux processus qui vont exécuter vos commandes ou programmes

Il permet de contrôler l'entrée standard et la sortie standard de chacun des processus que vous créez

Il permet de gérer des pipes (|)

La fonction `call` permet de remplacer la commande os.system

La fonction `run` de `subprocess` permet de retrouver la syntaxe des anciennes fonction d'exécution de programmes externes

La fonction call permet d'exécuter une commande 'à la `os.system`' :

```
res = subprocess.call(["ls","-l"])
```

est équivalent à

```
res = subprocess.call(["ls","-l"],stdout=sys.stdout,
stderr=sys.stderr)
```

les deux syntaxe impriment le résultat de la commande `'ls -l'` sur la sortie standard, et place le résultat de la commande, cad `0` dans `res`.

Si on veut vérifier que tout se passe bien on peut utiliser :

```
subprocess.check_call(["monscriptpleinderreur"])
```

qui déclenchera une exception `CalledProcessError`

On peut également rediriger la sortie d'une commande call :

```
with open("ls.txt","w+") as file:
  res = subprocess.call(["ls","-l"],stdout=file)
```

On ne verra alors rien à l'écran mais le résultat sera sauvegardé dans un fichier `"ls.txt"`

Depuis la version 3.5, il est recommandé d'utiliser la fonction run à la place de la fonction call. Celle-ci est plus complète (mais aussi plus complexe)

Les changements notables :

- le résultat n'est plus un simple entier
- il n'y a pas de `checkrun` mais `run` avec un argument `check=true`

subprocess.Popen() va créer un sous-processus qui sera exécuté en parallèle.

La commande wait() permettra d'attendre le résultat du processus créé. Ainsi :

```
p = subprocess.Popen(["ls","-l"]) ; print("start") ;
p.wait() ; print('end')
```

verra certainement s'afficher "start" puis le résultat de `'ls -l'` puis "end"

Avec call, le résultat de `'ls -l'` s'affiche avant "start" et "end"

```
p = subprocess.Popen(["ls","-l"],stdout=subprocess.PIPE,
universal_newlines=True)
```

permet de stocker le résultat dans un PIPE

```
s,e = p.communicate() # récupère le résultat
```

On peut donc enchaîner les commandes en redirigeant la sortie de l'une dans un Pipe, et l'entrée de l'autre depuis le Pipe.

```
import subprocess

p = subprocess.Popen(["ls","-l"],stdout=subprocess.PIPE)

q =
subprocess.Popen(["grep",".py"],stdin=p.stdout,stdout=sub
process.PIPE)

p.stdout.close()

s,e = q.communicate()

print(s)
```

7.5 L'aléatoire

On peut également générer des données aléatoirement comme entrée d'un programme.

Le package `random` contient différentes fonctions permettant de gérer de l'aléa :

- `randint(a,b)` : génère un nombre entier aléatoire entre a et b
- `random()` : génère un nombre aléatoire entre 0 et 1
- `randrange(a,b,step)` : génère un nombre aléatoire sur le modèle de la fonction range()
- `choice(séquence)` : une des valeur de la séquence
- `shuffle(séquence)` : mélange une séquence

Que fait le programme suivant ?

```
import random

resultats = { 'success':0, 'fail':0 }

result = list(resultats.keys())

for i in range(10000):
    resultats[ random.choice(result) ] += 1

print('moyenne de succès :',float(resultats['success']/
(resultats['success']+resultats['fail'])))
```

8.6 tkinter

Les principes sont assez simple :

- créer une fenêtre
- créer éventuellement des conteneurs (Frame)
- créer des widgets associés à des conteneur
- les positionner les uns par rapport aux autres
- lier les widgets d'interaction à des variables de saisi ou des fonctions

```
import tkinter as tk
window = tk.Tk()
```

8. Interface graphique

tk-inter

8.1 Principaux widgets

Label : permet d'afficher du texte

```
l = tk.Label(window, text="Ceci est un label")
```

TextField : permet de saisir du texte

```
tf = tk.Entry(window, textvariable=unevar, width=30)
```

Malheureusement il n'est pas possible de lier une entrée avec n'importe quelle variable ; il faut utiliser les variables de `tkinter` :

```
unevar = tk.StringVar()
```

`unevar.get()` pour récupérer le contenu de la variable

`unevar.set("chaine")` pour mettre à jour une variable

Button : permet de définir un bouton

```
b = tk.Button(window, text="Button", fg='blue',
command=button_fonction)
```

`button_function` est une fonction sans paramètre qui sera appelée lorsque quelqu'un appuie sur le bouton.

On peut aussi lier un widget à l'action de la souris ou de l'appui d'une touche particulière

```
f.bind('<Key-Return>',action)
```

`action` est une fonction avec un paramètre 'event' qui sera appelée lors de l'appui de la touche return.

8.2 Le positionnement

Il y a plusieurs façons de positionner les objets.

Une des façon est de positionner différents conteneurs les uns par rapport aux autres. Les différents conteneurs sont :

- les widgets
- la fenêtre `Tk()`
- les cadres : `Frame`

```
cadre = tk.Frame(window,borderwidth=2,relief='groove')
```

définit un cadre de la fenêtre dans lequel on pourra insérer des widgets ; les widgets devront être créés par la suite avec comme premier argument `cadre` au lieu de `window`.

Pour positionner les objets les uns par rapport aux autres, on pourra utiliser la fonction pack() qui peut prendre les arguments :

expand: boolean

fill: 'x', 'y', 'both', 'none'.

ipadx and **ipady** : une distance désignant une marge interne

padx and **pady** : une distance désignant une marge exterieure

side: 'left', 'right', 'top', 'bottom' indique le positionnement relatif

```
label.pack(side="top",padx=10,pady=10)
```

8.6.3 Exercice

Réalisez l'interface suivante :