17/09/2019

Python pour l'ingénieur

Langage Python

Cours

PREAMBULE

Pourquoi un cours ?

- Les outils utilisés par un ingénieur sont souvent dotés d'une interface utilisateur graphique et manuelle : appareils de mesure, Excel, outils d'exploitation, de dépouillement, d'analyse, etc.

Travail quotidien de l'ingénieur

- Pour réduire les actions manuelles au strict minimum, pour augmenter la productivité, pour éviter les erreurs de saisie, pour maîtriser la configuration des données utilisées, utiliser un langage de programmation tel que Python est un progrès notable.

Automatisation de son activité à l'aide de scripts Python pilotant les outils et réalisant les traitements

- Python est aussi beaucoup utilisé dans d'autres domaines : prototypage rapide, développement web, langage de script pour progiciel ou jeu vidéo, etc.

Langage versatile, à la fois professionnel et grand public (voir projet Raspberry Pi)

Le cours...

- Objectif : maîtriser les bases du langage Python et savoir ce qu'il est possible de faire.

- Le cours est divisé en sessions de courte durée, théorique (cours et échanges avec l'intervenant) et pratique (exercice et assistance directe de l'intervenant).

- L'ensemble des supports de cours est disponible sur le site du cours :

  http://learnpython.ovh

Les modules...

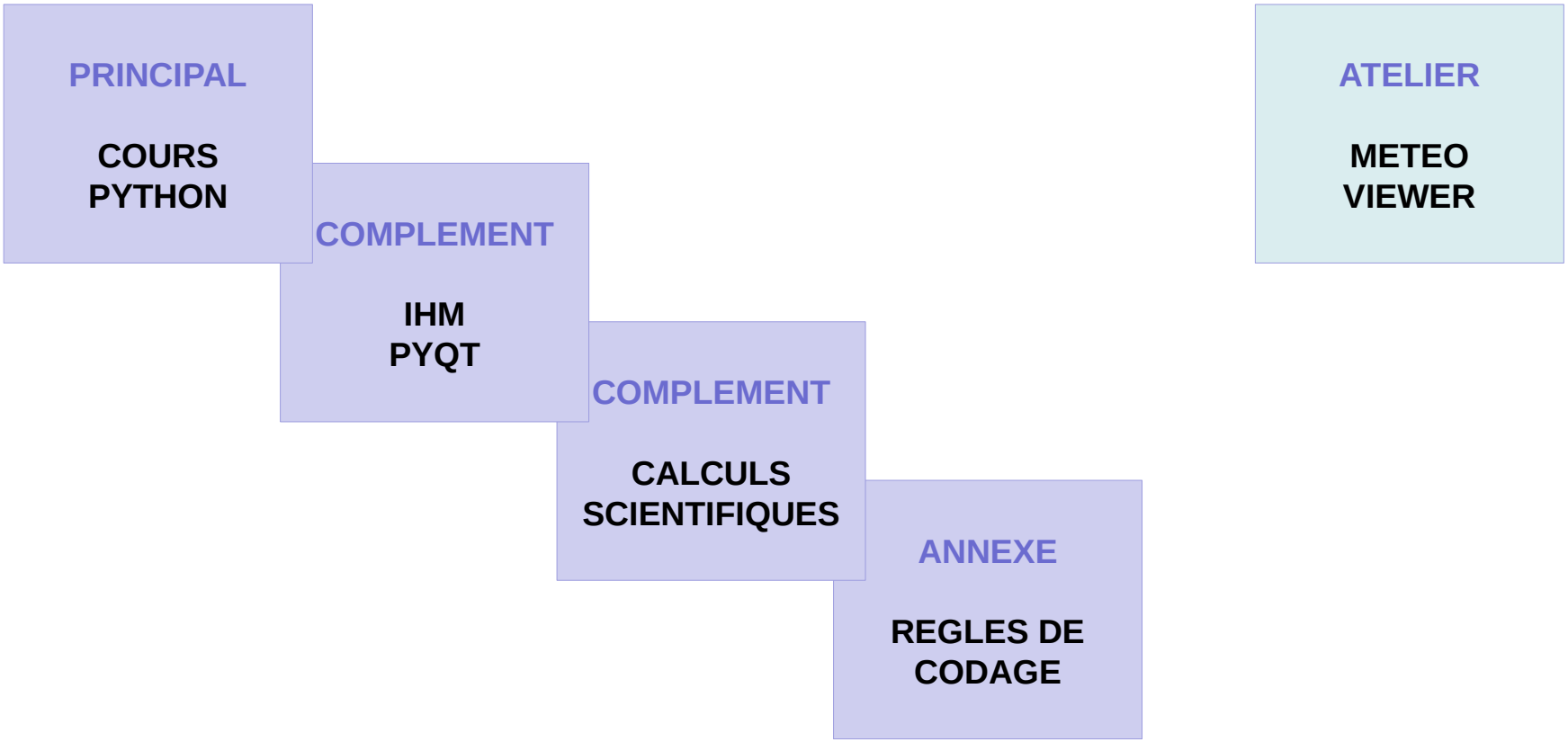
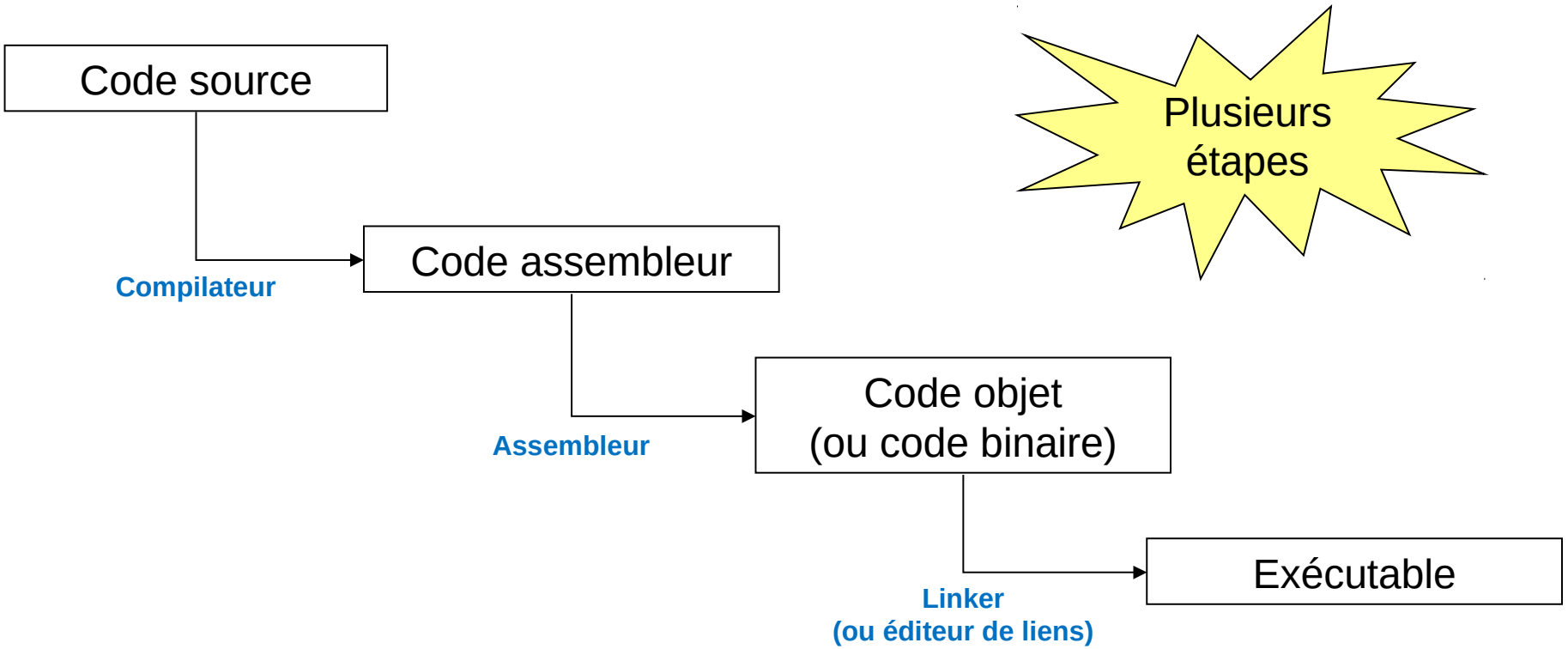


TABLE DES MATIERES

INTRODUCTION

Langage compilé
C, C++, Haskell, Erlang...

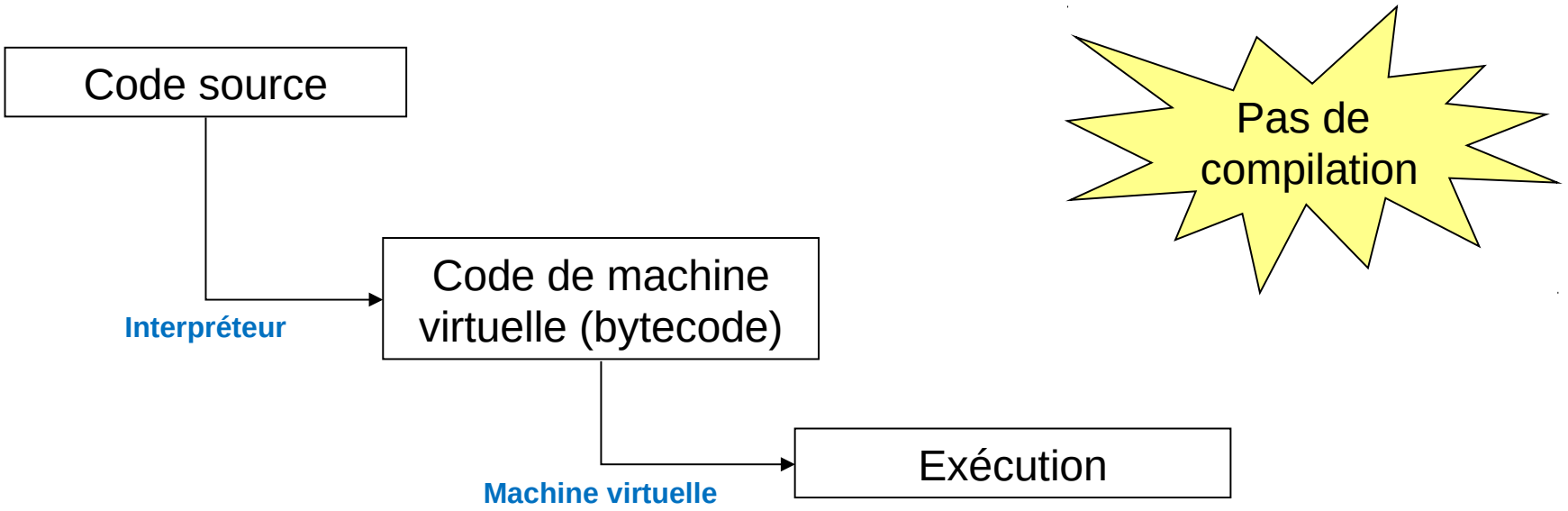


MAÎTRISE DE LA MACHINE

Traduction en code machine avant l'exécution

Langage interprété

Python, Javascript, Lua, Java, Ruby, Perl...



PRODUCTIF / MAQUETTAGE RAPIDE

Traduction en code machine au fur et à mesure de l'exécution
Exécution plus lente qu'un programme compilé

Programme Hello,world!

C
(1971)

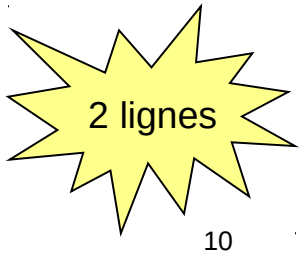
```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    char msg[100] = "Hello,world!";
    printf("%s\n", msg);
    exit(0);
}
```

C++
(1983)

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string msg = "Hello,world!";
    cout << msg << endl;
    return 0;
}
```

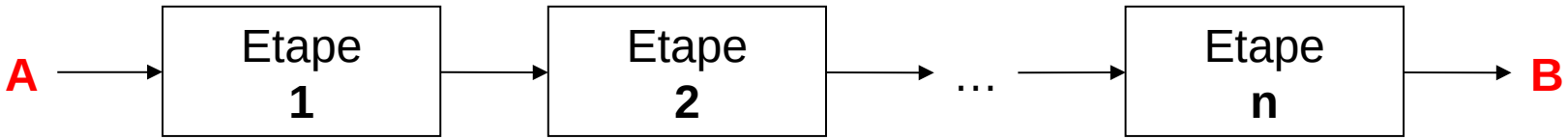
Python
(1991)

```python
msg = 'Hello,world!'
print(msg)
```



Programmation impérative
C, bash…

- Le problème à résoudre est vu comme une suite d'étapes modifiant l'état du système jusqu'à la résolution :



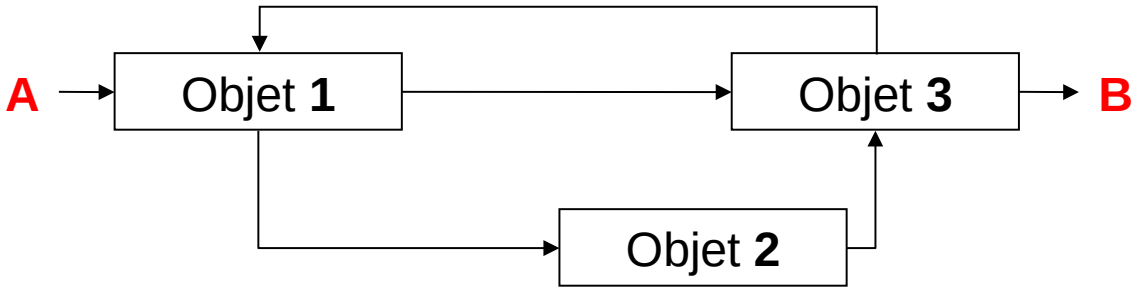
- Un développeur peut donc coder chaque étape et la tester unitairement. OK avec Python

- A la fin, tout est assemblé et le logiciel permet d'obtenir B à partir de A !

- *Inconvénient : une étape isolée de son contexte n'est pas forcément réutilisable…*

Programmation impérative
Langages objet : C++, Java…

- Le problème à résoudre est vu comme la collaboration d'entités autonomes encapsulant un état évolutif :



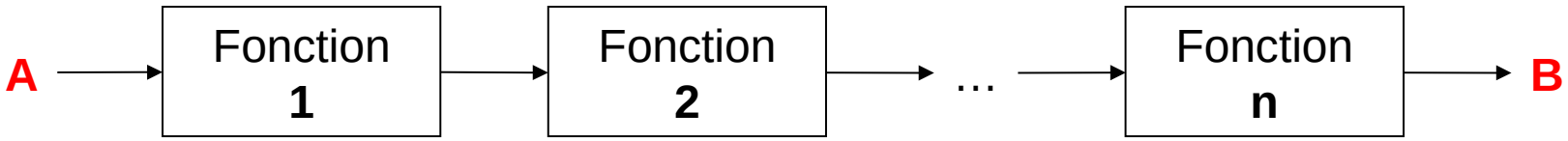
- Un développeur peut donc coder chaque objet et le tester unitairement. OK avec Python

- A la fin, tout est assemblé et le logiciel permet d'obtenir B à partir de A !

- *Avantage : un objet isolé de son contexte est très souvent réutilisable !*

Programmation déclarative
Langages fonctionnels : Haskell, Erlang…

- Le problème à résoudre est vu comme une suite de fonctions autonomes sans effet de bord :



- Un développeur peut donc coder chaque fonction et la tester unitairement. En partie OK avec Python

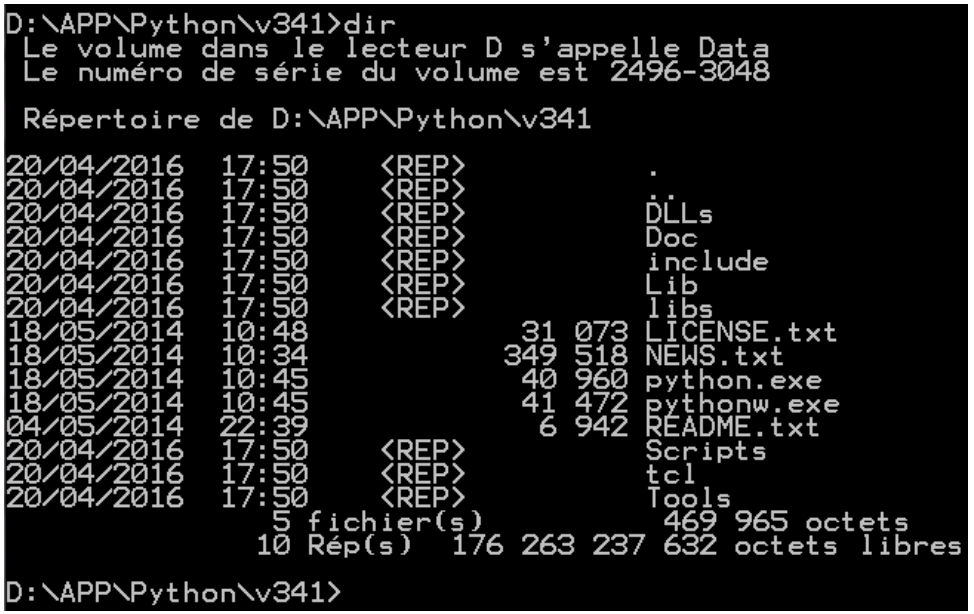
- A la fin, tout est assemblé et le logiciel permet d'obtenir B à partir de A !

- *Avantage : une fonction isolée de son contexte est réutilisable et sûre !*

ENVIRONNEMENT DE DEVELOPPEMENT

Écrire et exécuter un programme

- Python est un langage interprété.
- En pratique, Python est un logiciel installé dans un système (Windows, Linux, Mac, etc), et se présente sous la forme d'un exécutable.
- Deux modes d'exécution :
  - Mode interactif :
    - Au sein d'une console texte interactive (*shell*).
    - Les commandes sont saisies au clavier et exécutées une par une.
  - Mode fichier :
    - Dans une console de l'OS ou au sein d'un éditeur, spécialisé ou non.
    - Le fichier entier est lu et exécuté.
    - Exemple dans une console Windows : `python.exe test.py`



Vocabulaire

Fichier .py	module
Script Python	module
Programme Python	plusieurs modules
Paquetage Python	plusieurs modules

IDLE - Integrated DeveLopment Environment

- IDLE est un environnement très simple fourni avec Python.
- Il permet de combiner le mode interactif et le mode fichier.
- Mode d'emploi :
  - Lancer IDLE. Un *shell* Python apparaît.
  - Choisir File / New File (Ctrl-N) pour créer un nouveau module.
  - Saisir le code du module dans la nouvelle fenêtre.
  - Exécuter le module : F5. Les résultats s'affichent dans la première fenêtre.

Utiliser un encodage UTF-8.
Ne pas utiliser de vraies tabulations.
(touche TAB = 4 espaces)

EXERCICE

Stocker le résultat de l'opération 5/2 dans une variable, puis afficher cette variable à l'écran.

1. En mode interactif
2. En mode fichier

Indice :

```
print('Hello,World!\n')
```

Autres éditeurs :
Eclipse avec Pydev
PyCharm

Documentation

- La documentation de Python est très bien faite, et contient de très nombreux exemples de mise en œuvre.

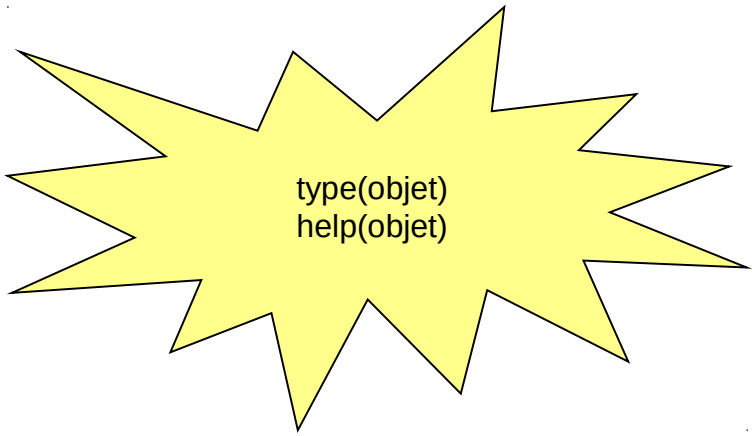
- Il est indispensable de savoir naviguer dans la documentation pour trouver l'information cherchée.

- Deux chapitres de la documentation sont utilisés couramment :
  - ***Tutorial***
    (start here)
  - ***Library Reference***
    (keep this under your pillow)

Le site python.org

Il contient beaucoup d'informations sur le langage… ainsi que la documentation complète et à jour.

https://www.python.org/



BASES DU LANGAGE

Les listes (1/5)

- Le type liste est fondamental en programmation Python.
- L'utilisation de listes doit être un réflexe, et la syntaxe doit être connue sans hésitation.
- Une liste peut contenir des données de tout type.

Créer une liste vide

```
a = []
a = list()
```

Créer une liste

```
a = ['lorem', 'ipsum', 12]
```

Ajouter un élément

```
a.append(34)
```

Concaténer deux listes

```
c = a + b
```

Accéder au n^{ième} élément

```
a[n-1]
```

EXERCICE

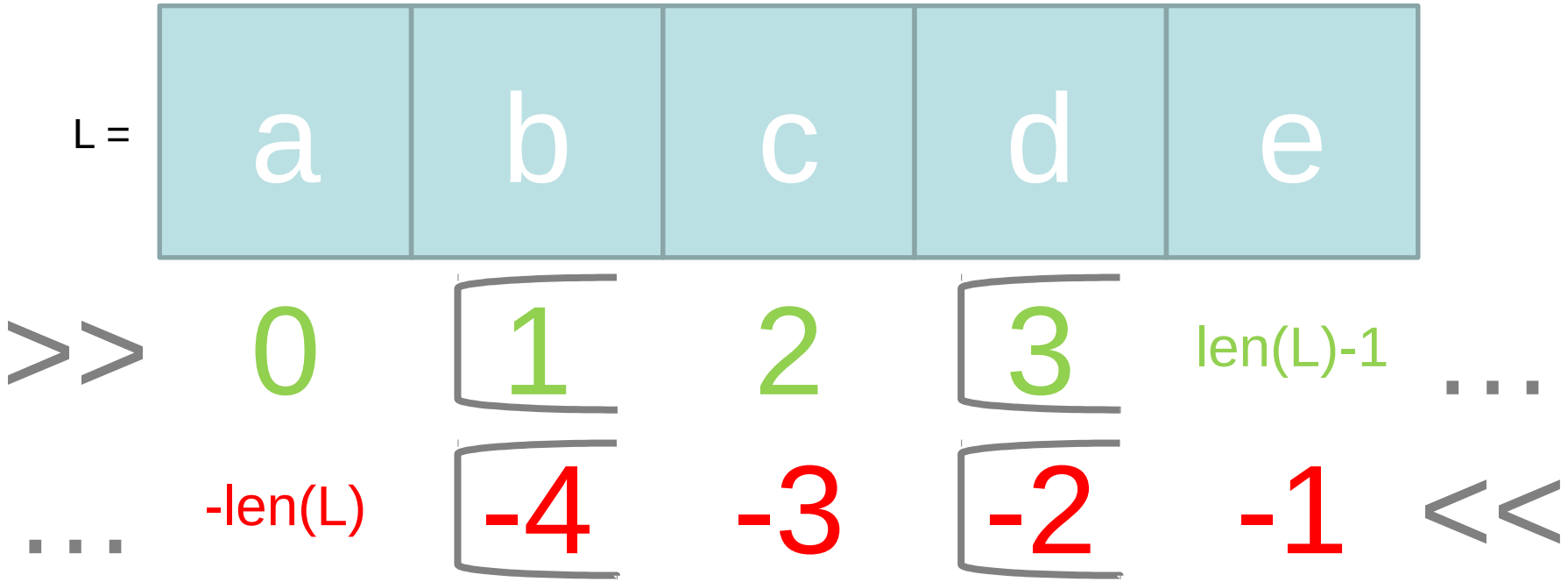
Créer deux listes de 3 éléments, a et b.

Créer une troisième liste contenant, dans l'ordre :

- Les éléments de a
- Le dernier élément de b
- Le premier élément de a
- Les éléments de b

Afficher la liste créée.

Les listes (2/5)



1 inclus, 3 exclus, -4 inclus, -2 exclus
Sous-liste de 2 éléments : 3-1 = -2-(-4) = 2

```
L[1:3] == L[-4:-2] == L[1:-2] == L[-4:3]
```

- En limite gauche, le début de la liste est noté ainsi : **`L[:3] == L[0:3]`**
- En limite droite, la fin de la liste est notée ainsi : **`L[-2:]`**
- Si besoin, limites gauche et droite ramenées aux bornes : **`L[-99:99] == L`**
- Si incohérence, liste vide : **`L[4:-1] == []`**

Les listes (3/5)

- Accéder à des éléments :

  Le premier / le dernier
  ```
  a[0] / a[-1]
  ```
  Le n^{ième} élément depuis le début / la fin
  ```
  a[n-1] / a[-n]
  ```

- Extraire des sous-listes :

  Toute la liste (ie copier une liste !)
  ```
  b = a[:]
  ```
  Tous les éléments, sauf le premier
  ```
  a[1:]
  ```
  Les deuxième et troisième éléments
  ```
  a[1:3]
  ```
  Les trois premiers éléments
  ```
  a[:3]
  ```
  Les trois derniers éléments
  ```
  a[-3:]
  ```

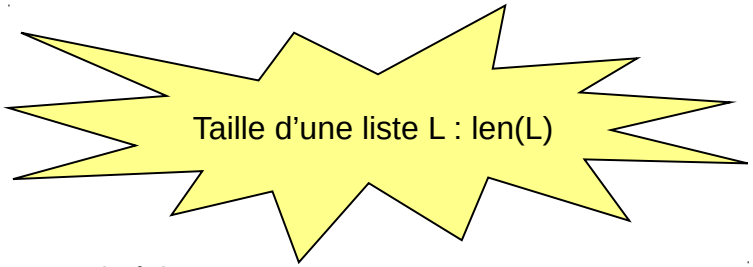
EXERCICE

Créer deux listes de 3 éléments, a et b.

Créer une troisième liste contenant, dans l'ordre :

1. Les deux premiers éléments de a
2. Les deux derniers éléments de b

Afficher la liste créée.



Les listes (4/5)

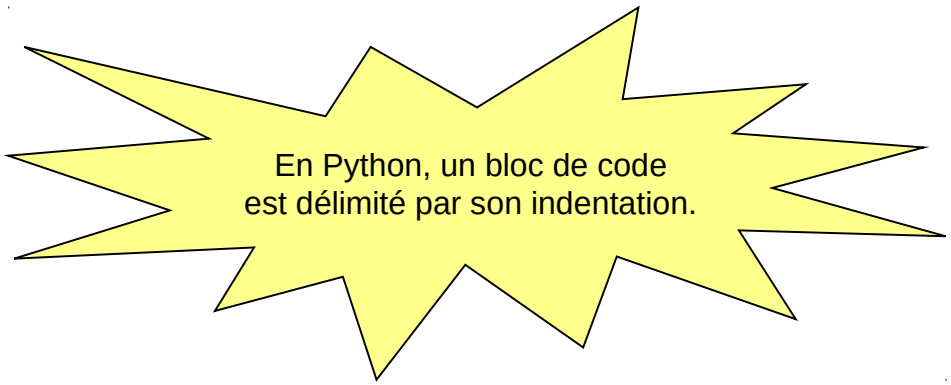
- En Python, une boucle **`for`** opère sur une liste.
- La boucle **`for`** commence par le premier élément de la liste, et se termine après avoir « consommé » le dernier.
- A chaque itération, la variable muette prend la valeur d'un élément de la liste (dans l'ordre).

```
for val in [-1, 99, 'foo', 'bar']:
  print(val)

for item in mylist:
  print(item)
```

EXERCICE

Soit le texte :
« Roma in Italia est »
Afficher le texte, un mot par ligne.

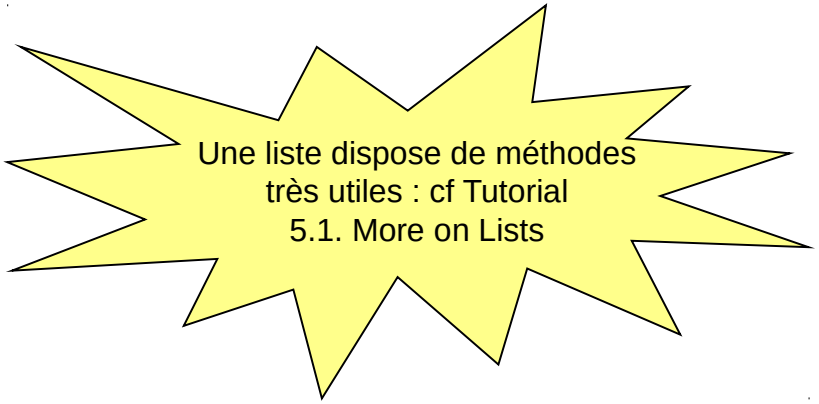
Indice :
`'Lorem ipsum'.split()`



Les listes (5/5)

- Il est possible d'utiliser une syntaxe très concise pour créer des listes (appelée *list comprehension*).

- En voici différents exemples :
  - Création à partir d'une liste d'indices et d'une transformation.
  - Création à partir d'une liste source et d'une sélection avec condition.
  - Création à partir d'une liste source et d'une transformation avec condition.




```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

a = [0, 1] * 5
print(a)

b = [i*i for i in range(10)]
print(b)

c = [i for i in b if i < 50]
print(c)

d = [(i if i < 50 else -1) for i in b]
print(d)

e = [3, 2, 1]
f = sorted(e)
print(f)
e.sort()
print(e)
```

Les dictionnaires

- Le type dictionnaire est très utile : il permet d'associer une valeur à une clé.

- Comme une liste, un dictionnaire peut contenir des données de tout type. Une clé est généralement une chaîne de caractères ou un nombre.

Créer un dictionnaire vide

```
a = {}
a = dict()
```

Créer un dictionnaire non vide

```
a = {k1:v1, k2:v2, k3:v3}
```

Ajouter ou modifier un couple (clé, valeur)

```
a[k] = v
```

Accéder à la valeur associée à une clé

```
a[k]
```

EXERCICE

Créer un dictionnaire à 3 entrées.

Afficher les clés et les valeurs à l'aide d'une boucle for.

Indice :

`a.keys()`

Les tuples

- Le type tuple est une généralisation de la notion de couple : un couple est un 2-tuple.

- Un tuple permet de grouper des données disparates. Si une fonction doit retourner plusieurs valeurs, elle peut le faire via un tuple :
  `return val1, val2, val3`

- A connaître :
  - Créer un tuple (*tuple packing*)
    - `a = x, y, z`
  - Opération inverse (*tuple unpacking*)
    - `x, y, z = a`
  - Obtenir le i^{ème} élément
    - `a[i-1]`

EXERCICE

Créer un 3-tuple : une chaîne de caractères, une liste, un dictionnaire.

Vérifier qu'il est impossible de remplacer la liste ou le dictionnaire par autre chose, mais qu'il est possible de faire évoluer leur contenu !

Point important

Un tuple est "immutable", contrairement à une liste, qui est "mutable" :

a[i] = object provoque une erreur

Attention… a[i] n'est pas forcément constant !

Les named tuples (1/2)

- Un named tuple est un tuple dont les champs sont nommés, ce qui permet une manipulation aisée : **`Employee.name`** au lieu de **`Employee[0]`**

- Un named tuple fonctionne comme un tuple ! Il est cependant nécessaire de le définir au préalable.

- A connaître :
  - Créer une classe namedtuple spécifique
    - `Point = namedtuple('Point', ['x', 'y'])`
  - Créer un namedtuple
    - `p = Point(x=11, y=22)  # ou Point(11,22)`
  - Manipuler les éléments d'un namedtuple
    - `p.x + p.y  # ou p[0] + p[1]`

      `>>> 33`

EXERCICE

Créer un namedtuple Friend permettant de stocker nom, prenom, telephone.

Créer une liste d'amis et trier cette liste avec sort(). Constat ?

Créer un namedtuple Friend2 permettant de stocker en plus une adresse, puis convertir votre liste de Friend en Friend2 grâce à l'opérateur *.

Point important

En début de module, écrire :
`from collections import namedtuple`

(fonctionnalité standard mais non native)

Les named tuples (2/2)

- Depuis Python **3.6**, un named tuple peut être déclaré comme suit :

```
from typing import NamedTuple

class Employee(NamedTuple):
    name: str
    id: int
```

- Ce code est équivalent à :

```
Employee = namedtuple(
  'Employee', ['name', 'id'])
```

EXERCICE

Trier votre liste par ordre décroissant de numéro de téléphone.

Indice :

Utiliser le paramètre key pour spécifier la fonction de calcul de la clé.

```
def getkey(x):
  return x.telephone
sort(key=getkey)
```

ou bien :

```
sort(key=lambda x:x.telephone)
```

Les itérables

- Un itérable est un objet pouvant être utilisé par une boucle, avec les caractéristiques d'une liste (début, fin, élément suivant).

  Note : un itérable peut être converti en une vraie liste. Si x est un itérable, list(x) est une liste contenant tous les éléments que fournirait progressivement x. A utiliser avec précaution !

- Exemples de fonctions built-in renvoyant un itérable :

  `range`
  `enumerate`
  `zip`

```
for i in range(10):
    print(i)


for i,item in enumerate(mylist):
  print(i, item)


# klist : liste des clés
# vlist : liste des valeurs
d = dict(zip(klist, vlist))
```

Les fichiers

- Ouvrir un fichier : **open**
  - En écriture
    - `f = open('myfile.txt', 'w')`
  - En lecture
    - `f = open('myfile.txt', 'r')`
  - Ecrire une ligne
    - `f.write('Lorem ipsum\n')`
    - `f.write('Result: ' + str(result) + '\n')`
  - Lire une ligne
    - `line = f.readline()`
  - Fermer un fichier **NE PAS OUBLIER !**
    - `f.close()`

- Utiliser un bloc **with** et **for** :

```
with open('myfile.txt', 'r') as f:
  for line in f:
    data = line.strip()[:72]
    print(data)
# fin du bloc with
# appel automatique de f.close() !
```

EXERCICE

Demander à l'utilisateur de saisir deux valeurs numériques.

Enregistrer la somme et la différence dans un fichier texte.

Relire et afficher le fichier texte créé.

Indice :

```
s = input()
```

En Python 2, écrire : `s = raw_input()`

Les conditions

- En Python, l'instruction **`if`** permet de définir un test.

EXERCICE

Créer une liste vide, et une autre non vide.

Utiliser successivement chacune de ces deux variables comme expression d'une condition.

Idem avec des chaînes de caractères.

```
if x > 0:
  x -= 1

if x > 0:
  x -= 1
elif x == 0:
  x = 10
else:
  x = -1

if s == 'Lorem':
if s != 'Lorem':

if x:
if not x:
if x and not y:

if item in mylist:
if item not in mylist:

if x is y:
if x is None:
if x is not None:
```

```
Test sur des nombres

Test sur des chaînes de
caractères

Test sur des booléens
if True:       if 1:
if False:      if 0:

Test sur des listes
Très important !

Test sur des objets
is compare l'identité de
deux objets. None = null
```

L'instruction **`pass`** ne fait rien !
(mais remplace un bloc de code)

Les boucles

- En Python, deux types de boucle sont disponibles.
- La boucle **`for`**, déjà étudiée, permet d'exécuter un bloc de code pour chaque élément d'une liste, successivement et dans l'ordre.
- La boucle **`while`** permet d'exécuter un bloc de code tant qu'une condition est vraie.

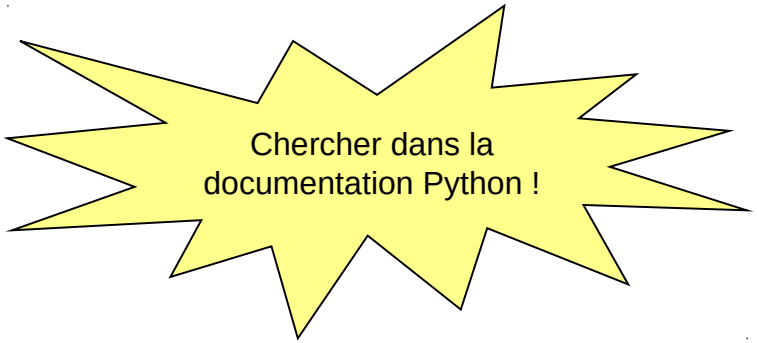
```
for item in mylist:
  pass

while expression:
  pass
```

EXERCICE

Quel est l'effet d'une instruction **break** dans une boucle ?

Coder un exemple.

Idem avec l'instruction **continue**.



Exceptions

- Certains morceaux de code sont risqués et peuvent provoquer une erreur *run-time*.

- Gérer une erreur *run-time* est possible grâce aux exceptions.

- Voici un exemple très simple de code fatal :

```
1/0
```

- Et de gestion d'exceptions :

```
try:
  1/0
except ZeroDivisionError:
  print('Division par 0.')
except:
  print('Erreur inattendue.')
```

EXERCICE

Demander à l'utilisateur de saisir un entier.

Si l'utilisateur saisit une valeur invalide, gérer l'exception et renvoyer -1.

Indice :

Déterminer d'abord l'exception à gérer en faisant un essai sans bloc `try`.

Pour finir...

- Une sélection de morceaux de code à connaître et à réutiliser !
  - Listes, boucle et condition
  - Conversions nombre / chaîne de caractères
  - Echange de deux valeurs
  - Affichage formaté
  - dict comprehension

```
liste = ['a', 'b', 'c']
for item in liste:
  if item not in ['a', 'z']:
    print(item)
```

```
result1 = 2.54
result2 = float('+00002.540')
print('Result: ' + str(result1))
print('Result: ' + str(result2*100))
print('Result: ' + str(int(result2)*100))
```

```
a = 1
b = 11
a, b = b, a
print(a, b)
```

```
print('{} + {} = {}'.format(a, b, a+b))
print(f'{a} + {b} = {a+b}')  # Python 3.6
```

```
d = {name[:3].upper():name for name in
['France', 'Australie', 'USA']}
print(d)
```

FONCTIONS & MODULES

Les fonctions (1/4)

- Une fonction est un regroupement de code. En Python, le mot-clé **`def`** permet de définir une fonction.

- Ce mot-clé permet aussi de créer des fonctions membres d'une classe en programmation orientée objet. Un exemple sera donné dans la suite du cours.

```
def myfunction(param1, param2):
  """Ma fonction."""
  sum = param1 + param2
  diff = param1 - param2
  return sum, diff

print(myfunction(5, 3))
```

EXERCICE

Développer la fonction spécifiée par :

ENTREE : une liste de phrases.

TRAITEMENT :

- Créer un dictionnaire associant le premier mot de chaque chaîne à la chaîne elle-même.
- Trier la liste fournie en entrée par ordre alphabétique.

SORTIE : le dictionnaire et la liste triée.

Les fonctions (2/4)

- Lors de la déclaration d'une fonction, il est possible de donner une valeur par défaut à chaque argument.

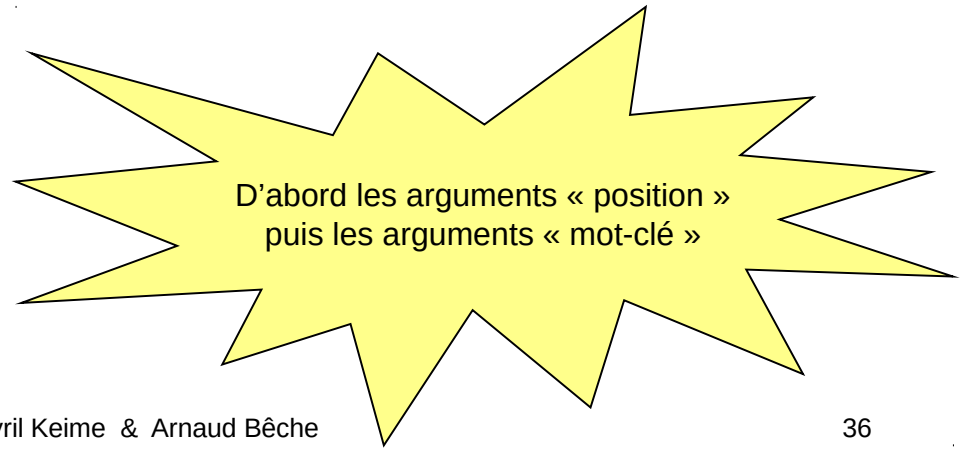
- La valeur par défaut d'un argument est la valeur que prend l'argument s'il n'est pas spécifié lors de l'appel de la fonction.

- Par ailleurs, lors de l'appel d'une fonction, il est possible de spécifier la valeur d'un argument quelconque de deux façons :

  Par sa position : la valeur souhaitée est écrite directement, et Python associe la valeur au bon argument grâce à sa position dans l'appel.

  Par son nom : le nom de l'argument devient un mot-clé permettant de spécifier sa valeur. Dans ce cas, la position n'intervient pas.

```
def myfunctionA(param1, param2):
    print(param1, param2)

def myfunctionB(param1, param2, param3=False, param4=0):
    print(param1, param2, param3, param4)

def myfunctionC(param1, param2='a', param3=False, param4=0):
    print(param1, param2, param3, param4)

myfunctionA(1, 'a')
myfunctionB(1, 'a')
myfunctionB(1, 'a', param3=True)
myfunctionB(1, 'a', param4=1)
myfunctionB(1, 'a', param4=1, param3=True)
myfunctionB(1, 'a', True, 1)
myfunctionC(1)
myfunctionC(1, 'b')
myfunctionC(1, param4=1, param3=True, param2='b')
```

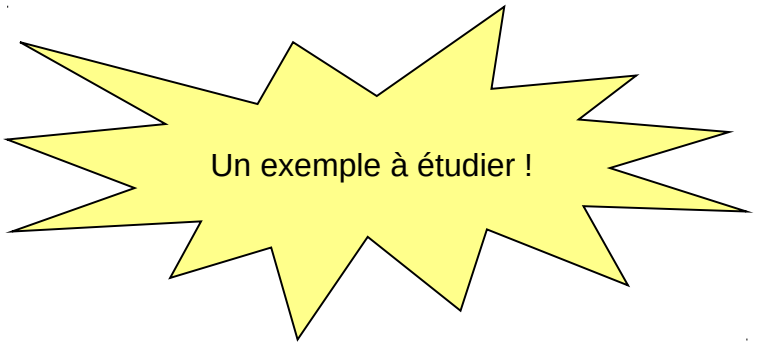


Les fonctions (3/4)

```
def getheadtemperature(config):
    """Get head temperature (°C)."""
    simu, t = config
    if simu:
        headtemperature = [25 + i*2.5 for i in range(24)]
        return headtemperature[t]
    else:
        return testbench.gettemperature()

def getemittedpower(config):
    """Get emitted power (W)."""
    simu, t = config
    if simu:
        emittedpower =\
          [1000, 1025, 1050, 1075] +\
          [1100 - i*50 for i in range(20)]
        return emittedpower[t]
    else:
        return testbench.getpower()
```

Les fonctions (4/4)

```
def runtest(simu=False):
    """Run test (24 h)."""
    reporttemplate = '{:4d}; {:4.1f}; {:4.0f}'
    report = []
    report.append('{:4}; {:4}; {:4}'.format('time','temp','pow'))
    for t in range(24):
        headtemperature = getheadtemperature((simu, t))
        emittedpower = getemittedpower((simu, t))
        report.append(reporttemplate.format(
          t,
          headtemperature,
          emittedpower))
    for line in report:
        print(line)

#runtest()
simu = True
runtest(simu)
```



Affichage formaté

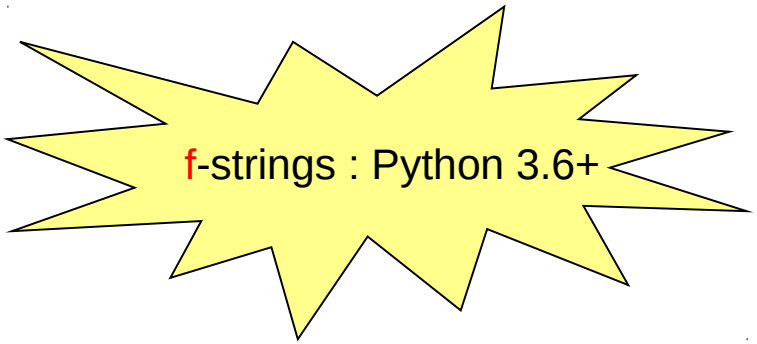
- Formater une chaîne de caractères pour un affichage sur écran ou un enregistrement sur disque est une bonne pratique.

- Méthode 1 : définir une chaîne de formatage, puis l'utiliser pour formater plusieurs lignes : format strings →

```
mytemplate = '{:2d} + {:2d} = {:3d}'
for a,b in [(1,2),(3,4),(99,99)]:
  print(mytemplate.format(a,b,a+b))
```

- Méthode 2 : formater une chaîne de manière immédiate : f-strings →

```
a = 5
b = 6
print(f'{a} + {b} = {a+b}')
print(f'{a:2d} + {b:2d} = {a+b:3d}')
```



Les modules (1/3)

- Un module est un fichier Python (.py) contenant du code réutilisable, et définissant un espace de noms.
- Un module peut donc être réutilisé, via la commande **`import`**, dans un autre module ou script Python.
- Ce mécanisme est fondamental pour organiser le code correctement, et permettre sa maintenance.
- La commande **`import`** permet aussi de réutiliser un ensemble complet de modules, appelé paquetage (ou *package*).

EXERCICE

Importer le paquetage standard permettant de gérer la date et l'heure.

Afficher l'heure courante.

```
import module
import module as name
from module import object
from module import object as name
```

Les modules (2/3)

- Un module doit être organisé, et sa structure peut être normalisée.

- Ci-contre un exemple de normalisation, à suivre dorénavant pour toute création de module.
  - Fichier source unicode.
  - Description du module.
  - Utilisation des paquetages standards `os` et `sys`.
  - Définition d'une fonction (plusieurs fonctions peuvent être définies, ainsi que des classes).
  - Définition d'une fonction de test du module.
  - Test de lancement du module par la commande `import`.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""
Multi-line comment.
(module)
"""

import os
import sys

def myfunction(myarg):
    """Multi-line comment.
    (function)"""
    print('myfunction')
    print(myarg)
    pass  # Do nothing.

def mytest():  # Module test.
    print('mytest')
    myfunction('This is a test.')
    pass  # Do nothing.

if __name__ == '__main__':
    mytest()
```

Les modules (3/3)

- Un module peut être utilisé de deux manières différentes :
  - Exécution directe.
  - Importation par la commande `import`.

- Exécution directe :
  - `python mymodule.py`
  - Dans ce cas la condition de fin est vraie, et la fonction de test est exécutée.
  - Autrement dit, il s'agit d'une méthode très simple pour implémenter et exécuter un test unitaire de module.

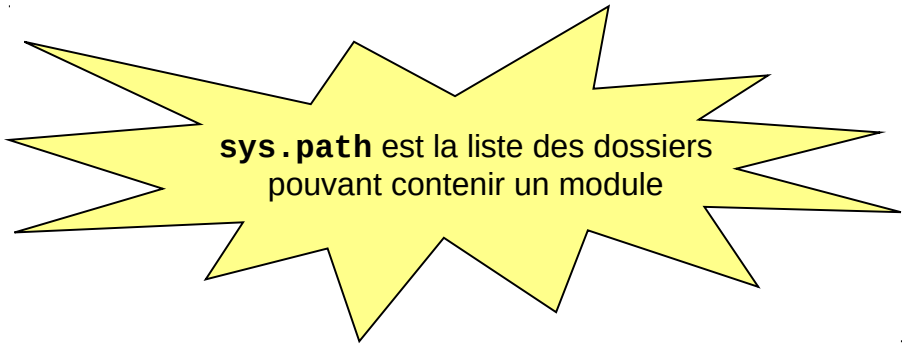
- Commande **`import`** :
  - `import mymodule`
  - Dans ce cas la condition de fin est fausse, et la fonction de test n'est pas exécutée.
  - Les constantes, fonctions et classes du module importé sont maintenant disponibles !

EXERCICE

Saisir le code donné en exemple.

Dans une console système, exécuter le module. Que constatez-vous ?

Dans une console Python, importer le module. Que constatez-vous ?



CLASSES & POO

Définition d'une classe

- Une classe est un regroupement de données et de fonctions. En Python, le mot-clé **`class`** permet de définir une classe.

- Une fonction membre (ou méthode) est définie comme une fonction normale, avec deux contraintes :
  - Elle doit être définie dans le bloc de code de la classe.
  - Elle a forcément un premier argument, habituellement nommé `self`. Cet argument permet d'identifier l'objet appelant.

- Une fois l'objet créé, une méthode peut être appelée en omettant le premier argument (il est implicite).

```
class MyClass:
  """Ma classe."""
  def __init__(self):
    self.wordlist = ['Lorem', 'ipsum']

  def addword(self, word):
    """Add a word."""
    self.wordlist.append(word)

  def getid(self, n):
    """Compute id of word list."""
    result = ''
    for word in self.wordlist:
      result += word[:n]
    return result

c = MyClass()
c.addword('New word...')
print(c.getid(1))
print(c.getid(3))
print(c.wordlist)
```

Exercice : getid() en une seule ligne !

Exercice : classe Complexe

EXERCICE

Créer une classe capable de représenter un nombre complexe et de traiter les opérations suivantes :

- Changer la valeur du nombre complexe
- Retourner la partie réelle
- Retourner la partie imaginaire
- Retourner le module
- Retourner l'argument
- Modifier la valeur en inversant le nombre complexe
- Modifier la valeur en ajoutant un autre nombre complexe.

Important ! Tester cette classe pendant le codage !

Pour cela, créer une fonction de test dès le début et tester les fonctionnalités au fur et à mesure qu'elles sont implémentées.

Pour les plus rapides : définir des opérateurs pour cette classe : +, *

Note : complexe est un type de base en Python. Voir la documentation !

Notions de programmation orientée objet

- **Classes** et instances de classe (les objets).

- **Encapsulation** : les attributs d'une classe ne doivent être connus/modifiés que par la classe elle-même. (Utilisation de méthodes get/set.)
  Mécanisme Python décrit ci-contre.

- **Héritage** : une classe peut être définie comme l'extension d'une autre classe.
  Mécanisme Python décrit ci-contre.

- **Polymorphisme** : capacité d'exécuter un code identique sur des objets différents, mais partageant une interface.
  Mécanisme Python implicite.

```
Note : utiliser la décoration @staticmethod pour
définir une méthode de classe
```

```
Préfixer par __ (double underscore) un attribut
ou une méthode qui doit être inaccessible de
l'extérieur de la classe (membre privé).

Utiliser le mécanisme property() pour
implémenter les méthodes get/set d'un attribut.
```

```
class BaseClass:
    def __init__(self, a):
        self.a = a

class MyClass(BaseClass):
    def __init__(self, a, b):
        super().__init__(a)
        self.b = b

En Python 2, écrire :
        BaseClass.__init__(self, a)
à la place de :
        super().__init__(a)
```

Exercice : classe FigureGeometrique

EXERCICE

Attention ! Ne pas oublier **( )** lors de l'appel d'une fonction ou d'une instanciation

1. Écrire une classe FigureGeometrique définissant une interface de classe contenant les fonctions Perimetre() et Surface(). Ajouter également à la classe FigureGeometrique un attribut privé nommé « NomObjet » de type chaîne de caractères et une méthode appelée Nom() retournant cet attribut. La classe FigureGeometrique est-elle abstraite ?

2. Écrire une classe Rectangle et une classe Cercle héritées de la classe FigureGeometrique et implémentant les fonctions Perimetre() et Surface(). Le constructeur des classes Rectangle et Cercle permet d'initialiser le nom et les dimensions des objets créés.

3. Écrire une classe Carre héritée de la classe Rectangle.

4. Créer et initialiser des objets des classes Rectangle, Carre et Cercle puis stocker ces objets dans une liste.

5. A l'aide d'une boucle, afficher le nom, le périmètre et la surface de chaque objet, sans réaliser de test sur le type ou le nom de l'objet.

Méthodes spéciales

- Une classe peut définir un certain nombre de fonctions spéciales. Parmi ces fonctions :

  `__str__(self)` : cette fonction doit renvoyer une chaîne de caractères décrivant l'objet, et compréhensible par un être humain. Appelée implicitement par **`print()`**.

  `__repr__(self)` : cette fonction doit renvoyer une chaîne de caractères décrivant complètement l'objet d'un point de vue informatique.

Toujours définir **`__repr__()`**

- Remarques :

  `__repr__()` remplace **`__str__()`** si celle-ci n'est pas définie.

  `__str__()` d'un conteneur (list par exemple) utilise **`__repr__()`** pour le contenu.

PAQUETAGES STANDARDS

math — Mathematical functions

- This module is always available. It provides access to the mathematical functions defined by the C standard.

os.path

os.path — Common pathname manipulations

- This module implements some useful functions on pathnames. To read or write files see open(), and for accessing the filesystem see the os module.

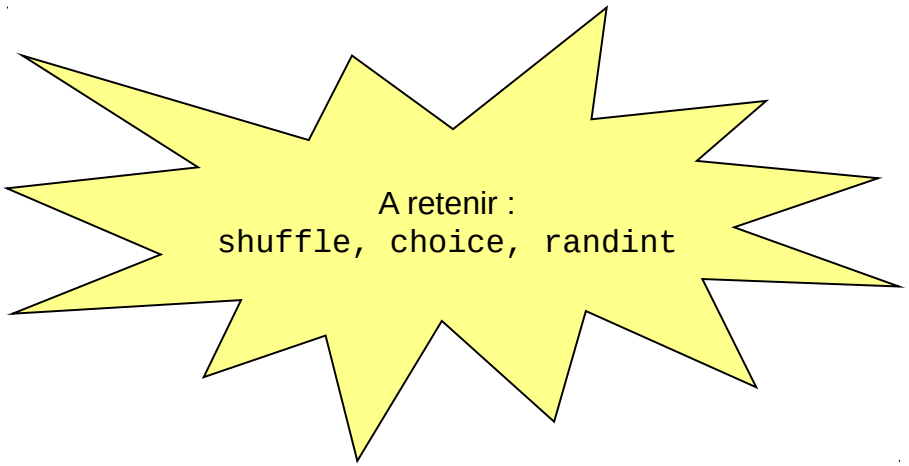
datetime — Basic date and time types

- The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. For related functionality, see also the time and calendar modules.

random

random — Generate pseudo-random numbers

- This module implements pseudo-random number generators for various distributions.
- For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.



pickle

pickle — Python object serialization

- The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” [1] or “flattening”, however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

```
class MyClass:
    def __init__(self, a):
        self.a = a


# Serialize
import pickle
from myclass import MyClass
l = [MyClass(123), MyClass('abc'),
     MyClass(10**10)]
f = open('myclasslist.bin', 'wb')
pickle.dump(l, f,
pickle.HIGHEST_PROTOCOL)
f.close()


# Deserialize
import pickle
f = open('myclasslist.bin', 'rb')
l = pickle.load(f)
f.close()
for instance in l:
    print(instance.a)
```

re — Regular expression operations

- This module provides regular expression matching operations similar to those found in Perl. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings.

collections

collections — Container datatypes

- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

- namedtuple() : factory function for creating tuple subclasses with named fields
- deque : list-like container with fast appends and pops on either end
- ChainMap : dict-like class for creating a single view of multiple mappings
- Counter : dict subclass for counting hashable objects
- OrderedDict : dict subclass that remembers the order entries were added
- defaultdict : dict subclass that calls a factory function to supply missing values
- UserDict : wrapper around dictionary objects for easier dict subclassing
- UserList : wrapper around list objects for easier list subclassing
- UserString : wrapper around string objects for easier string subclassing

unittest — Unit testing framework

- The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

- To achieve this, unittest supports some important concepts in an object-oriented way:
  - **test fixture** : a test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions.
  - **test case** : a test case is the individual unit of testing. It checks for a specific response to a particular set of inputs.
  - **test suite** : a test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.
  - **test runner** : a test runner is a component which orchestrates the execution of tests and provides the outcome to the user.

CONCLUSION

Pour aller plus loin...

Lire la documentation de Python

Consulter régulièrement python.org

Explorer le site http://learnpython.ovh

Apprendre avec la communauté Python sur le web

Apprendre avec des livres (Head First Python) ou des MOOC

Pratiquer !

M E R C I

ANNEXE

Morceaux de code

```
# Fonctions lambda
liste = [('a',1,'dummy'), ('x',2), ('y',2)]
print(liste)
liste.sort(key=lambda item: item[1], reverse=True)
print(liste)


# isinstance
a = 5
if isinstance(a, float):
    print('a -> float')
elif isinstance (a, int):
    print('a -> int')
print(isinstance ('', str))


# Opérateur +
class Book:
    def __init__(self, title, nbpages):
        self.title = title
        self.nbpages = nbpages
    def __repr__(self):
        return '{} : {:d} pages'.format(self.title, self.nbpages)
    def __add__(self, other):
        return Book(' / '.join([self.title, other.title]),
                    self.nbpages + other.nbpages)
a = Book('Pavé', 1000)
b = Book('Résumé', 1)
print(a+b)
```

Morceaux de code

```
# * et **
def myfunc(*args, **kwargs):
    s = ' '.join(args)
    for k,v in kwargs.items():
      s += ' {}={}'.format(k,v)
    return s

print(myfunc('a'))
print(myfunc('a', 'b'))
print(myfunc('a', 'b', hello='world', good='bye'))


# set
a = {1,2,99,4,1,1,2,1,6,1,1,1,1,1,1,9,99,2}
print(a)
print(len(a))

b = [1,2,99,4,1,1,2,1,6,1,1,1,1,1,1,9,99,2]
c = list(set(b))
c.sort()
print(c)
```

Morceaux de code

```
# Générateurs
#   En lecture seule une seule fois et de longueur inconnue !
generator = (x**2 for x in range(3))
for val in generator:
    print(val)
for val in generator:  # vide...
    print(val)


# yield
#   Dans une fonction, la transforme en générateur !
def compute(exp):
    mylist = range(3)
    for i in mylist:
      yield i**exp

generator1 = compute(2)
generator2 = compute(3)
for val in generator1:
    print(val)
for val in generator2:
    print(val)
```

Morceaux de code

```
# for/else
def search(data, value):
    for v in data:
        if v == value:
            print('FOUND')
            break
    else:
        print('NOT FOUND')
        v = None
    return v

print(search([1, 42, 99, 51], 42))
print(search([1, 42, 99, 51], 43))


# with
with open('expourfinir3.py') as f:
    for i,line in enumerate(f):
        if i == 19:
            print(line.strip())
            break
```

Morceaux de code

```
# enum
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

for const in Color:
    print(const.value)

print(Color.RED is Color.RED)
print(Color.RED is Color.GREEN)


# itertools
import itertools
data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
print(data)
print(list(itertools.accumulate(data, max)))
print(list(itertools.compress(data, [1,0,1,0,1,1])))
```

Morceaux de code

```
# next
def gencolours():
    while(True):
        yield 'RED'
        yield 'GREEN'
        yield 'BLUE'
colours = gencolours()
for i in range(10):
    print(next(colours))
```