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

Cours de programmation Python            Edouard.Thiel@univ-amu.fr - 16/03/2012
----------------------------

Plan :
        1) Rappel sur les fonctions
        2) La notion de module
        3) Découpage d'un programme en modules
        4) Portée des variables
        5) Notion d'interface de programme : API
        6) Programmation orientée objet (POO) et Classes
        7) Interface graphique (GUI) avec Tkinter
        8) TP traceur de courbes


1) Rappel sur les fonctions


  - Déclarer une fonction sans paramètres :

      def nom_fonction () :
        corps_de_la_fonction

    Ex :
      def bonjour () :
        print ("Hello World!")

    On peut accoler ou non les "()" et ":" avec le nom
      def bonjour():
        print ("Hello World!")

    Tester
      $ python
      >>> def bonjour():
      ...   print ("Hello World!")
      ...

    Appel
      >>> bonjour
      <function bonjour at 0x2b4270>

    --> raté ! On n'a pas appelé la fonction mais demandé ce que python
    connait sur "bonjour" ; c'est une function et il nous donne son adresse
    dans son espace mémoire.

    Le vrai appel : il faut donner les ()
      >>> bonjour()
      Hello World!

    On peut accoler ou non : "bonjour ()" ou "bonjour ( )" marchent aussi.

    RQ: on peut aussi renommer la fonction (en réalité, mémoriser son adresse
    dans une variable qui prend le type fonction) :
    >>> x = bonjour
    >>> x
    <function bonjour at 0x2b4270>
    >>> x()
    Hello World!

    RQ: comment faire une fonction vide ? avec l'instruction pass qui ne fait rien
      def rien():
        pass


  - Déclarer une fonction avec paramètres

      def afficher_vitesse (v) :
        print ("Ma vitesse est de " + v + " km/h")

    Appel :
      >>> afficher_vitesse ()
      Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  TypeError: afficher_vitesse() takes exactly 1 argument (0 given)

--> raté ! On a oublié le paramètre
  >>> afficher_vitesse (60)
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in afficher_vitesse
  TypeError: cannot concatenate 'str' and 'int' objects

--> Le passage de paramètres est bon, mais on a mal concaténé
    (chaîne + entier)
--> C'est une erreur dans la fonction ; on ne la voit que à l'appel

Solutions :
  - Passer une chaîne en paramètre
    >>> afficher_vitesse ("dix")
    Ma vitesse est de dix km/h

  - ou convertir systématiquement le paramètre en chaîne :
  def afficher_vitesse (v) :
    print ("Ma vitesse est de " + str(v) + " km/h")

  >>> afficher_vitesse ("dix")
  Ma vitesse est de dix km/h
  >>> afficher_vitesse (10)
  Ma vitesse est de 10 km/h
  >>> afficher_vitesse (10.5)
  Ma vitesse est de 10.5 km/h

--> str() est elle-même une fonction, qui accepte un paramètre,
    et qui renvoie une valeur (de type chaîne).

RQ: On aurait aussi pu faire

  print "Ma vitesse est de ", v, " km/h"
  (sans les parenthèses ?)
  --> python convertit chaque paramètre en str, puis les concatène en
      rajoutant un espace entre chaque str.

ou encore

  print ("Ma vitesse est de %d km/h" % (v))
  --> utilise la fonction printf du C ("print Formatté") ; %d est remplacé
      par le premier entier dans le tuple (v).

RQ: sur certaines versions de python on peut écrire print sans ()
    il vaut mieux les mettre.


- Comment renvoyer une valeur ? avec return

  def nom_de_la_fonction ( paramètres ) :
    corps_de_la_fonction
    return valeur

  On peut mettre return n'importe où et plusieurs fois ;
  return provoque le retour immédiat de la fonction.

  Exemple :
  def vitesse_est_depassee (v, vmax):
    if v > vmax:
      return True
    else:
      return False

  >>> vitesse_est_depassee (50,60)
  False

--> On a passé plusieurs paramètres :
    séparés par des virgules ; affectés dans l'ordre

RQ: on peut changer l'ordre en donnant les noms à l'appel :
```

```
>>> vitesse_est_depassee (vmax=60,v=50)
False
```

Utilisation d'une fonction renvoyant une valeur :
- dans une expression comme dans l'exemple avec str()
- dans un test de branchement ou de boucle :

```
def tester_vitesse (v, vmax):
  if vitesse_est_depassee(v,vmax):
    print ("Vous roulez trop vite !")
  else:
    print ("Bonne route")
```

- Paramètres optionnels

On peut rendre des paramètres de fonctions optionnels ; pour cela il suffit de les pré-initialiser dans la déclaration. Les paramètres optionnels doivent être déclarés après les paramètres obligatoires.

Ex:

```
def essai (x, y, z=21, t=34):
  print ("x = %d  y = %d  z = %d  t = %d" % (x, y, z, t))

>>> essai (3,5,7,9)
x = 3  y = 5  z = 7  t = 9
>>> essai (3,5,7)
x = 3  y = 5  z = 7  t = 34
>>> essai (3,5)
x = 3  y = 5  z = 21  t = 34
>>> essai (3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: essai() takes at least 2 arguments (1 given)
>>> essai (3,5,t=56)
x = 3  y = 5  z = 21  t = 56
```

2) La notion de module

- En mode interactif, des que l'on quitte python, les fcts que l'on a tapées sont perdues. Une solution est de les enregistrer dans un fichier .py ; on appelle alors ce fichier un module.

Exemple : fichier vitesse.py

```
# Module vitesse.py - E. Thiel - 21/03/2012

def afficher_vitesse (v) :
  print ("Ma vitesse est de %d km/h" % (v))

def vitesse_est_depassee (v, vmax):
  if v > vmax:
    return True
  else:
    return False

def tester_vitesse (v, vmax):
  if vitesse_est_depassee(v,vmax):
    print ("Vous roulez trop vite !")
  else:
    print ("Bonne route")
```

- Utilisation (quitter et relancer python)

```
>>> afficher_vitesse(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'afficher_vitesse' is not defined
```

--> Il faut dire à Python que l'on va utiliser le module !

```
  >>> import vitesse.py
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ImportError: No module named py
```

--> Il ne faut pas donner l'extension

```
  >>> import vitesse
  >>> afficher_vitesse(50)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'afficher_vitesse' is not defined
```

--> Il faut dire que la fonction est dans le module ;

```
  >>> vitesse.afficher_vitesse(50)
  Ma vitesse est de 50 km/h
```

- Espaces de noms

Un espace de nom est un dictionnaire dans lequel sont stockées les noms de variables, de fonctions, etc. Le but est d'éviter des conflits de noms.

Lorsque l'on importe un module, python crée un nouvel espace de noms portant le nom du module ; tout est importé dans cet espace de noms. Pour y accéder, on écrit le nom de l'espace puis "." (ici "vitesse.")

On considère que dans l'interpréteur on est aussi dans un module : le "module principal", dans lequel on accède directement à l'espace de nom principal sans préfixer.

Le nom d'un module est accessible avec __name__ dans son propre espace de noms :

```
  def afficher_nom_module():
    print (__name__)

  >>> import vitesse
  >>> vitesse.afficher_nom_module()
  vitesse
  >>> vitesse.__name__       # c'est pareil
  'vitesse'
  >>> __name__
  '__main__'
```

Dans le dernier exemple on a affiché le nom du module principal. On s'en servira dans la suite.

RQ: on peut modifier l'espace de nom d'un module à l'import

```
  >>> import vitesse as v
  >>> v.afficher_vitesse(50)
```

RQ: on peut afficher les symboles d'un espace de nom (sans les builtin) avec

```
  dir()                # dans l'espace courant
  dir(nom_du_module)   # dans l'espace du module
```

- Autre méthode d'import : import dans l'espace de nom courant

```
  (quitter et relancer)
  >>> from vitesse import afficher_vitesse
  >>> afficher_vitesse (50)
  Ma vitesse est de 50 km/h
```

Comme afficher_vitesse a été inséré dans l'espace de nom courant, on ne préfixe pas pour y accéder.

```
  >>> vitesse.afficher_vitesse(50)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'vitesse' is not defined
```

--> C'est normal, il n'est pas dans l'espace de nom "vitesse" (qui n'existe pas)

.

Les "import" et "from .. import" ne sont pas équivalents.
On peut cumuler les 2 imports (ça ira dans 2 espaces de noms différents, donc pas de conflits).

--> Pour importer plusieurs fcts :

```
>>> from vitesse import afficher_vitesse, vitesse_est_depassee, tester_vitesse
```

--> Pour importer toutes les fonctions du module :

```
>>> from vitesse import *
```

RQ: on peut renommer une fonction à l'import :

```
>>> from vitesse import afficher_vitesse as affi
>>> affi(50)
```

- Quelle méthode d'import choisir ?

Toujours préférer "import" sur "from ... import", car dans la seconde méthode on va mélanger les fonctions et variables du module avec celle de l'espace courant --> dangereux, sauf si on s'y prend bien en préfixant tout --> autant faire "import".

Cet usage est néanmoins utile dans certaines situations.

3) Découpage d'un programme en modules

- Dès qu'un programme a une certaine taille, on le découpe en plusieurs modules pour
  - structurer le programme, en regroupant ce qui va ensemble
  - le rendre plus lisible
  - réutiliser des modules existants
  - faciliter la mise au point et la maintenance.

  Plusieurs approches :
  - découper le problème en modules et fcts = "approche descendante"
  - écrire des fcts et modules puis les assembler = "approche ascendante"

- Dans un programme composé de plusieurs modules, le module que l'on passe à python est le module principal ;
  il importe tous les autres modules nécessaires :
  - directement ou indirectement (1) ;
  - au début du module ou en cours d'exécution (2).

Ex (1) :

```
titi.py                 | titi.py                  | titi.py
  import toto           |   from toto import *     |   import toto
  toto.tata.hop()       |   hop()                  |   toto.hop()
                        |                          |
toto.py                 | toto.py                  | toto.py
  import tata           |   from tata import *     |   from tata import *
                        |                          |
tata.py                 | idem                     | idem
  def hop():            |                          |
    print ("hopla")     |                          |
                        |                          |
$ python titi.py        | idem                     | idem
hopla                   |                          |
```

--> Dans les 2e et 3e solutions, pas besoin de savoir que tata est importée dans toto.

Ex (2) :

```
titi.py
  if 1 == 1 :
    import toto
  else:
    import tata
  hop()
```

```
toto.py
  def hop():
    print ("Bonjour")

tata.py
  def hop():
    print ("Salut")
```

- Que se passe-t'il si un module contient du code hors des fonctions ?

```
Ex :
  titi.py
    print ("titi 1")
    import toto
    print ("titi 2")
    def gros_minet():
      print ("miaou !")
    print ("titi 3")

  toto.py
    print ("toto 1")
    def gros_chien():
      print ("ouaf !")
    print ("toto 2")

  $ python titi.py
  titi 1
  toto 1
  toto 2
  titi 2
  titi 3

  --> Python execute séquentiellement les instructions hors des fcts ;
  lorsqu'un module est importé, il est exécuté de la même manière,
  une seule fois à l'imoprt.

  RQ: tous les print que l'on fait permer de "tracer" le programme ;
  c'est une technique de mise au point et d'exploration, simple et efficace !
```

- Que se passe-t'il si aucun module ne contient de code hors des fcts ?

```
Ex :
  titi.py
    import toto
    def gros_minet():
      print ("miaou !")

  toto.py
    def gros_chien():
      print ("ouaf !")

  $ python titi.py
--> rien !
```

- Dans ce cas, comment dire à Python quelle fonction "principale" appeler ?

```
* Soit en appelant la fct explicitement :
  titi.py
    import toto
    def gros_minet():
      print ("miaou !")

    # Programme principal
    gros_minet()
    toto.gros_chien()

  toto.py
    idem

  $ python titi.py
```

```
  miaou !
  ouaf !

* Soit en testant la variable __name__ :
  titi.py
    import toto
    def gros_minet():
      print ("miaou !")

    def gros_ours():
      print ("gromf !")

    def fct_principale():
      gros_minet()
      toto.gros_chien()

    if __name__ == "__main__":
      fct_principale()

  toto.py
    idem

  $ python titi.py
  miaou !
  ouaf !
```

Avantage : titi peut être utilisé à la fois comme prog principal, et être utilisé comme module dans un autre programme :

```
  tutu.py
    import titi
    titi.gros_ours()

  $ python tutu.py
  gromf !
```

- Bilan : que fait exactement Python lorsqu'on importe un module ?

Il consulte d'abord la liste des modules chargés (sys.modules) pour voir si le module est déjà chargé. Si c'est le cas, il ignore le nouvel import.

Sinon :
- il crée un nouvel objet module vide (en gros, un dictionnaire)
- il insère cet objet module dans sys.modules
- si le module n'a pas été compilé, il le compile et l'enregistre dans le fichier nom_du_modume.pyc ; si ce fichier existe déjà mais qu'il est plus vieux que le module, il est recompilé.
- il charge le fichier .pyc (contient du bytecode python)
- il execute le module dans son espace de nom.

--> On peut donc importer des modules sans crainte de faire des erreurs si
- on importe le module 2 fois (a importe b et c, b importe c)
- on fait des imports cycliques (a importe b qui importe a)

- Quels sont les modules chargés ?

```
  >>> import sys
  >>> sys.modules.keys()
  [ ..., 'titi', ... ]
  >>> sys.modules['titi']
  <module 'titi' from 'titi.pyc'>
  >>> import math
  >>> sys.modules['math']
  <module 'math' from '/opt/local/Library/..../math.so'>
```

Il y a différents types de modules :
- les modules de votre programme (fichiers .py) ;
- les "bibliothèques dynamiques" = DLL de suffixe .pyd sur Windows, .so sous linux, .dylib sous MacOS.
  Par ex le module math : math.sqrt(), math.sin(), ...
- les répertoires contenant des fichiers .py

Dans le cas des répertoires, ils doivent contenir un fichier __init__.py
qui inclut les autres fichiers du répertoire :

```
ga/bu.py
  def affibu():
    print ("Bu !")

ga/zo.pu
  def affizo():
    print ("Zo !")

ga/__init__.py
  from bu import *
  from zo import *
  __all__ = ["bu", "zo"]  # pour aider Python lors d'un "from ga import *"

meu.py                ou encore :
  import ga          |    from ga import *
  ga.affibu()        |    affibu()
  ga.affizo()        |    affizo()
```

On appelle encore un répertoire de modules un "package" ; c'est une
collection de modules.

Les modules sont cherchés dans une liste ordonnée de répertoires :

```
>>> import sys
>>> sys.path
[ ... ]
```

- Pourquoi écrire du code exécutable dans un module (hors test __main__) ?

  --> Pour initialiser le module

  --> Pour tester le module (code de validation)

4) Portée des variables

- On appelle portée (ou visibilité) des variables, les endroits du programme
  où l'on peut accéder aux variables.

- Python recherche les variables dans différents espaces de noms :
  - d'abord dans l'espace local (de la fonction)
  - puis dans l'espace global (du module)
  - enfin dans l'espace builtin

  Il s'arrête dès qu'il a trouvé.

  On peut voir les globales et locales en appelant les fcts :

```
  globals()
  locals()
```

- Examinons des cas de figure :

```
a = 25
def f():
  print (a)

>>> f()
25
```

--> a est défini dans l'espace global ; la fonction y a accès
en consultation.

```
def g():
  b = 30
  print (b)

>>> b
NameError: name 'b' is not defined
```

```
>>> g()
30
>>> b
NameError: name 'b' is not defined
```

--> b est défini dans l'espace local de la fonction ; il est détruit dès la sortie de la fonction ; b n'est pas accessible dans l'espace global.

```
c = 20
def h():
  c = 30
  print (c)

>>> c
20
>>> h()
30
>>> c
20
```

--> dans h(), affecter c crée une variable dans l'espace de nom de la fct ; cette variable "masque" alors la variable c de l'espace global.

```
c = 20
def h():
  global c
  c = 30
  print (c)

>>> c
20
>>> h()
30
>>> c
30
```

--> en déclarant c globale, l'affectation ne créera pas de variable dans l'espace de nom de la fct, mais utilisera la variable de l'espace global.

```
def i():
  global d
  d = 60

>>> d
NameError: name 'd' is not defined
>>> i()
>>> d
60
```

--> comme l'espace global ne contient pas d, et que d est déclarée globale, elle est créé dans l'espace global.

```
def j():
  global e

>>> j()
>>> e
NameError: name 'e' is not defined
```

--> c'est bien l'affectation qui crée la variable.

```
global f
f = 70
def k():
  f = 80

>>> f
70
>>> k()
>>> f
70
```

--> l'étiquette "global" n'est pas transmise dans les fcts ; il faut chaque

fois préciser dans les fct les globales.

Exercice :

```
h = 20
def x():
  global h
  h = 30
def y():
  x()
  h = 40

>>> y()
que vaut maintenant h ? (réponse : 30)
```

Moralité : éviter au maximum les variables globales et tout passer en paramètres pour éviter des "effets de bord", c-à-d des erreurs de portées de variables.

- Portée des variables dans les modules

```
riri.py
  a = 30
  def f():
    print (a)

$ python
>>> import riri
>>> a
NameError: name 'a' is not defined
>>> riri.a
30
>>> riri.f()
30
>>> riri.a = 40
>>> riri.f()
40
```

--> Tout est normal, a et f() résident dans l'espace de nom riri

```
(quitter et relancer python)
>>> from riri import *
>>> a
30
>>> f()
30
>>> a = 40
>>> f()
30
```

--> On a importé a et f() dans l'espace global du module __main__ ; or f() voit la variable globale au module riri qui est différente. On en déduit que l'import "from import" se fait par recopie. Autre essai pour vérifier :

```
riri.py
  a = 30
  def f():
    global a
    a = 90
    print (a)

>>> from riri import *
>>> a
30
>>> f()
90
>>> a
30
```

--> l'import "from import" fait donc bien une recopie ; il interdit donc la modification des variables globales du module depuis un

autre module !

--> l'adjectif "global" rend une variable globale au module, pas au programme !

5) Notion d'interface de programme : API

Lorsqu'on écrit un module ou un package (= un répertoire de modules) dans le but de le distribuer ou le réutiliser, il faut définir une API, c'est-à-dire fixer la façon dont un programmeur va s'en servir.

- Fcts publiques et privées

Il faut distinguer les fonctions et variables qui
- peuvent être manipulées par le programmeur : la partie publique
- sont internes et ne doivent pas être manipulées par le programmeur : la partie privée.

En python, on indique à l'interpréteur que les variables et fonctions sont privées en les préfixant par '_' :

```
truc.py
  # -*- coding: utf-8 -*-
  a = 5
  _b = 7
  def x():
    print ("fct publique")
  def _y():
    print ("fct privée")

>>> from truc import *
>>> a
5
>>> _b
NameError: name '_b' is not defined
>>> x()
fct publique
>>> _y()
NameError: name '_y' is not defined
```

--> from ... import n'importe pas ce qui est préfixé par '_' dans l'espace de nom courant.

Mais :

```
>>> import truc
>>> truc.a
5
>>> truc._b
7
>>> truc.x()
fct publique
>>> truc._y()
fct privée
```

--> ce n'est pas vraiment une protection, mais une indication au programmeur.

- Setter et getter

On a vu que les variables d'un module sont globales aux module mais que il vaut mieux éviter de les modifier de l'extérieur (c-a-d d'un autre module, de __main__ ou de l'interpréteur).

De plus, certaines variables ne peuvent pas être modifiées sans conséquence pour les autres données du module : seul le module "sait" comment peuvent évoluer ses variables.

C'est pourquoi le module doit fournir des fonctions spécilisées qui modifient ou renvoient les valeurs de ces variables : on les appelle les

setter et les getter. Du coup, toutes ces variables peuvent être privées !

Ex :
```
foo.py
  _bar = 10
  def set_bar(val):
    global _bar    # sinon on va créer une var locale !!
    _bar = val
  def get_bar():
    return _bar    # on aurait pu la déclarer globale par prudence

>>> import foo
>>> foo.get_bar()
10
>>> foo.set_bar(20)
>>> foo.get_bar()
20
```

- Documentation :

Chaque fonction publique doit être documentée. On peut mettre des explications en commentaires avant la fonction, mais c'est encore mieux d'utiliser le mécanisme de documentation intégrée à Python : les docstring

```
def carre (x):
  """Calcule le carré de x.

  Cette fonction prend en argument un nombre x ;
  Elle renvoie le carré de x."""
  return x*x
```

La docstring se place au début de la fct.
La première ligne commence par une maj puis fini par un point.
Elle est suivie d'une ligne vide, puis d'un texte libre.
Le tout est entre triple quotes car c'est une chaîne de car multi-lignes.

```
>>> carre(3)
9
>>> carre.__doc__
'Calcule le carr\xc3\xa9 de x.\n\n        Cette fonction prend en argument
un nombre x ;\n        Elle renvoie le carr\xc3\xa9 de x.'

>>> help(carre)
Help on function carre in module __main__:

carre(x)
    Calcule le carré de x.

    Cette fonction prend en argument un nombre x ;
    Elle renvoie le carré de x.
(END)
```

- Versions

Le module ou package est amélioré au fil du temps par le développeur ;
or si l'API est changée, il faudra modifier le programme qui l'utilise.
Il est donc primordial de numéroter les versions pour s'y retrouver.
En général on choisit une numérotation x.y.z :

- le x est un changement général d'architecture
- pour x.y donné, on garantit que la partie publique est inchangée
- le z est une numérotation concernant la partie privée. D'un z à l'autre, la partie privée peut changer complètement, sans affecter la partie publique

.

On commence souvent avec x = 0, cela signifie au le logiciel est en version alpha, et que tout peut changer très vite.
Une version béta est une version distribuable, mais pas encore stable et qu'il ne faut pas l'utiliser en production.
Une release candidate (rc) est une version presque finale, dans laquelle on ne change plus rien, on corrige les derniers bugs.

La version finale est la version stable = débuggée et figée.

--> les modules ou packages sont souvent proposés en deux versions,
la version stable et la version de développement.

Pour gérer les versions avec plusieurs programmeurs, on utilise des
outils tels que svn, git, Google code, etc.

- Tests

Chaque fonction obéit à des spécifications = décrivent ce que la fonction fait,
ce qui rentre et ce qui sort ; la façon de le faire n'est pas décrite dans
les spec (boîte noire).

Lors d'une modification, une fonction peut (temporairement) ne plus obéir
aux spécifications : on parle alors de régression.

Il faut écrire des batteries de tests pour chaque fonction, surtout avec
des langages de typage dynamique comme python. Il suffit alors de lancer les
tests pour trouver les éventuelles régressions.

On appelle ces tests des "tests unitaires". Ils sont importants lorsqu'on
développe à plusieurs ou lorsqu'on distribue un module. Il faut les écrire
très "tôt", avant même le code des fonctions !

- Avant d¿écrire le code, ils obligent a préciser le détail des spécification d¿une manière utile.
- Pendant l¿écriture du code, ils empêchent de trop programmer : quand tous les cas de test passent, la fonction est terminée.
- Pendant la refactorisation de code, ils garantissent que la nouvelle version se comporte comme l¿ancienne.
- Pendant la maintenance du code, ils permettent d¿être couvert si quelqu¿un se plaint que votre dernière modification fait planter son code.
- lorsqu'on écrit du code en équipe, on se partage les tâches, le code et les tests.

Python fournit le module unittest pour faire ces tests unitaires.

Voir http://docs.python.org/library/unittest.html

6) Programmation orientée objet (POO) et Classes

- La programmation orientée objet (POO) est un style de programation
(on dit un paradigme), apparue dans les années 70, et très à la mode
actuellement.

Il consiste a définir des briques logicielles appelées objets.
Chaque objet représente quelque chose et a un comportement : il sait
communiquer, interagir avec d'autres objets, etc.

Il y a énormément de notions liées à la POO ; on va simplifier au maximum
et ne voir que quelques éléments : en effet, Python est un langage orienté
objets et de nombreux modules (dont le module unittest) utilisent ce style
de programmation --> il faut avoir ces quelques notions de POO pour les
utiliser.

- Les classes

Une classe est une structure de données qui implémente un objet.
  - elle contient des données, que l'on appelle des attributs ;
  - elle contient des fonctions sachant manipuler les données, on les appelle des méthodes.
  - Les attributs et les méthodes peuvent être publiques ou privées (on les préfixe avec _).

Une variable de type classe s'appelle une instance de la classe ;
déclarer une telle variable s'appelle "instancier la classe".
Les attributs s'appellent aussi les "variables d'instance" de la classe.

Un module python peut contenir plusieurs classes (contrairement à Java).

Un avantage d'une classe sur un module est que l'on peut instancier plusieurs fois une classe, chaque instance ayant ses propres variables.

- Voyons comment déclarer une classe en Python :
(l'usage est de mettre une majuscule au nom de type)

```
class MaClasse:
  """Ma premiere classe."""
  i = 12345
  def f(self):
    print ("La valeur de i est %d" % self.i)
```

Les classes vivent dans leur propre espace de nom.
La seule façon pour les methodes pour accéder aux attributs est d'utiliser une référence à l'instance elle même : self
C'est pourquoi chaque méthode doit déclarer self en premier ;
mais on ne le donne jamais à l'appel.

Instancier la classe :

```
>>> x = MaClasse()
```

Afficher l'attribut i de l'objet x :

```
>>> x.i
12345
```

Appeler la méthode f de l'objet x :

```
>>> x.f()
La valeur de i est 12345
```

Afficher la documentation :

```
>>> help(x)  # ou help(MaClasse)

Help on instance of MaClasse in module __main__:

class MaClasse
 |  Ma premiere classe.
 |
 |  Methods defined here:
 |
 |  f(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  i = 12345
(END)
```

Attention : dans une définition de classe,
- hors méthode, toute affectation crée un attribut
- dans une méthode, les attributs sont préfixés par self
  --> toute affectation de self.x crée l'attribut x
  --> l'affectation de y créé une variable locale y, détruite à la fin de l'appel de la méthode.

- Augmenter une classe :

On peut ajouter des attributs et des méthodes dynamiquement (= au runtime) à une classe ou à une instance. Si on l'ajoute à la classe, toutes les instances le reçoivent aussi.

```
class A:
  i = 21
  def f(self):
    print ("hop")

>>> help(A)
Help on class A ...
>>> dir(A)
['__doc__', '__module__', 'f', 'i']
```

```
>>> a = A()
>>> dir(a)
['__doc__', '__module__', 'f', 'i']

On ajoute un attribut :
>>> A.b = 32     # à a classe et à toutes les instances
>>> a.c = 41     # à cette instance
>>> dir(A)
['__doc__', '__module__', 'b', 'f', 'i']
>>> dir(a)
['__doc__', '__module__', 'b', 'c', 'f', 'i']

Pour ajouter une méthode, il faut d'abord créer la fonction :
def ggg(self):
  print ("pop")
def hhh(self):
  print ("job")

>>> A.g = ggg   # rajoute la méthode g à la classe et à toules les instances
>>> dir(A)
['__doc__', '__module__', 'b', 'f', 'g', 'i']
>>> a.g()
pop

Attention ! Pour rajouter une méthode à une instance il faut faire :
>>> import types
>>> a.g = types.MethodType(ggg,a)
>>> dir(A)
['__doc__', '__module__', 'b', 'f', 'g', 'i']    # inchangé
>>> dir(a)
['__doc__', '__module__', 'b', 'c', 'f', 'g', 'h', 'i']
>>> a.g()
pop
```

Conclusion : c'est pratique, mais ça peut provoquer des effets de bords : si on se trompe dans le nom d'un attribut dans une affectation, on peut en créer un autre silencieusement !

- Constructeurs et destructeurs :

Lorsqu'on crée une instance, une fonction d'initialisation des variables d'instance est automatiquement appelée : c'est le "constructeur". Lorsqu'on libère/détruit une instance, une fonction de terminaison est appelée : c'est le "destructeur".

Déclarer un constructeur :

```
  def __init__(self):
    self.i = 33
```

On peut aussi paramétrer le constructeur :

```
  class MaClasse:
    def __init__(self, k):
      self.i = k
    def f(self):
      print ("La valeur de i est %d" % self.i)

  >>> x = MaClasse(32)
  >>> x.f()
  La valeur de i est 32
  >>> y = MaClasse()
  TypeError: __init__() takes exactly 2 arguments (1 given)
```

Déclarer un destructeur :

```
  class Toto:
    def __del__(self):
      print ("Instance de Toto détruite !")

  >>> x = Toto()
  >>> del(x)
  Instance de Toto détruite !
```

En fait, del() ne détruit pas x, elle décrémente le nombre de références

sur l'objet. Lorsque ce nombre passe à 0, l'instance est détruite :

```
>>> x = Toto()    # 1 ref
>>> y = x         # 2 refs
>>> del(x)        # 1 ref
>>> del(y)        # 0 ref
Instance de Toto détruite !
```

- Héritage :

On peut fabriquer une classe à partir d'une autre : c'est l'héritage.
La nouvelle classe hérite des attributs et méthodes de la classe mère.

```
class Mere:
  i = 22
  def f(self):
    print ("Coucou")

class Fille(Mere):
  j = 33
  def g(self):
    print ("Hopla")

m = Mere()
>>> m.i
22
>>> m.f()
Coucou
>>> m.j
AttributeError: Mere instance has no attribute 'j'
>>> m.h()
AttributeError: Mere instance has no attribute 'h'

>>> e = Fille()
>>> e.i
22
>>> e.j
33
>> e.f()
Coucou
>>> e.g()
Hopla
```

RQ: python permet l'héritage multiple.

Si une méthode de la fille veut appeler une méthode de la mère,
elle peut préfixer par le nom de la classe Mère en cas d'ambiguité.
Ex :

```
class Mere:
  def __init__(self):
    print ("Coucou")

class Fille(Mere):
  def __init__(self):
    Mere.__init__(self)
    print ("Hopla")

>>> e = Fille()
Coucou
Hopla
```

--> On voit aussi que l'instanciation d'une classe ne déclenche pas
le constructeur de la classe mère ; il faut l'appeler explicitement.

- Setter et Getter pour une classe

```
class C():
  _x = 1  # attribut privé
  def getx(self):
    return self._x
```

```
        def setx(self, value):
          self._x = value
        def delx(self):
          del self._x
        x = property(getx, setx, delx, "Je suis la propriété 'x'.")
```

Maintenant on peut écrire

```
>>> c = C()
>>> print (c.x)
1
>>> c.x = 5
>>> print (c.x)
5
```

Voir aussi http://docs.python.org/library/functions.html#property

Il y a plein d'autres notions mais ça devait suffire pour aborder la partie suivante !

7) Interface graphique (GUI) avec Tkinter

Un GUI toolkit est un module permettant de créer des fenêtres, les décorer avec des widgets (windows gadgets : boutons, champs de saisie, etc) et réagir aux évènements déclenchés par l'utilisateur.

Python propose de nombreux GUI toolkit, des plus simples aux plus complets (PyGTK, PyQt, etc).

Le GUI toolkit "standard" est Tkinter (pour inter-face), basé sur Tk. Tk est une "vieille" librairie graphique issue de tcl/tk, avec un graphisme un peu "carré", mais il est assez simple, suffisament riche, et normalement installé en standard avec python.

Une fois que l'on maîtrise Tkinter, il est relativement aisé de passer à un toolkit plus riche car de nombreuses notions sont communes.

http://wiki.python.org/moin/TkInter

- Première fenêtre

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

# Tkinter est l'un des rares modules à préconiser un "from import"
# Si cet import échoue, essayer en minuscules : from tkinter import *
from Tkinter import *

# Crée une fen principale, 1 par prog
root = Tk()

# Crée un widget label, son parent est root
w = Label(root, text="Hello, world!")

# calcule sa taille et se rend visible
w.pack()

# Boucle d'évènements ; rien n'apparait avant
root.mainloop()
```

RQ: Pour créer d'autres fenêtres, utiliser le widget Toplevel

- Fenêtre avec boutons

```
from Tkinter import *

# On met toute l'interface dans une classe application
class App:
```

```
        def __init__(self, root):

          # Crée un conteneur
          frame = Frame(root)
          frame.pack()

          button = Button(frame, text="QUIT", fg="red", command=root.quit)
          button.pack(side=LEFT)  # Par défaut TOP

          button = Button(frame, text="Hello", command=self.on_hello_button_press)
          button.pack(side=LEFT)

        def on_hello_button_press (self):
          print ("Button hello pressé !")

      root = Tk()
      app = App(root)
      root.mainloop()
```

L'objet root a une méthode quit qui permet de quitter le programme.

On passe la fonction à appeler lorsque le bouton est pressé avec
le paramètre optionnel command.

Pour le bouton Hello, on définit une nouvelle action ; cette fonction,
destinée à être appelée par un widget, s'appelle une callback.

- Le widget Entry : champ de saisie

On initialise le champ avec .delete(0,END) puis .insert(0,"valeur").
On récupère la valeur avec .get()

```
  from Tkinter import *

  class App:
    def __init__(self, root):

      # Crée un conteneur
      frame = Frame(root)
      frame.pack()

      button = Button(frame, text="QUIT", fg="red", command=root.quit)
      button.pack(side=LEFT)  # Par défaut TOP

      button = Button(frame, text="Hello", command=self.on_hello_button_press)
      button.pack(side=LEFT)

      self.entry1 = Entry(frame, width=15)
      self.entry1.pack()
      self.entry1.delete(0, END)
      self.entry1.insert(0, "Valeur initiale")

    def on_hello_button_press (self):
      print ("Texte entré = \"%s\"" % self.entry1.get())

  root = Tk()
  app = App(root)
  root.mainloop()
```

- Fenêtre avec une zone de dessins : widget Canvas

On crée un canvas puis on enregistre des dessins dedans ; le widget se
charge de les dessiner.

Les coordonnées sont : origine 0,0 en haut à gauche, y vers le bas
Les dessins qui dépassent le canvas sont "coupés".

```
  from Tkinter import *

  class App:
    def __init__(self, root):
```

```
          # Conteneur principal
          frame_princ = Frame(root)
          frame_princ.pack(anchor=NW)   # par défaut side=TOP et anchor=CENTER

          # Crée un conteneur pour les boutons
          frame_top = Frame(frame_princ)
          frame_top.pack(anchor=NW)

          button = Button(frame_top, text="QUIT", fg="red", command=root.quit)
          button.pack(side=LEFT)

          button = Button(frame_top, text="Hello", command=self.on_hello_button)
          button.pack(side=LEFT)

          # Crée un canvas sous la zone de boutons
          canvas = Canvas(frame_princ, bg="white", width=400, height=300);
          canvas.pack();

          # Enregistre des dessins dans le canvas
          canvas.create_line(0, 0, 100, 50, 500, 100, fill="blue")
          canvas.create_rectangle (50, 100, 250, 170, outline="green", fill="yellow"
)

        def on_hello_button (self):
          print ("Button hello pressé !")

      root = Tk()
      app = App(root)
      root.mainloop()
```

- Effacer ou rajouter des dessins

  Il faut mémoriser canvas comme attribut de App pour y accéder ensuite.

  Chaque fois que l'on appelle self.canvas.create_quelquechose(), on rajoute un dessin à la liste des dessins du canvas, et le canvas est redessiné.

  Pour vider le canvas, on appelle self.canvas.delete(ALL)

```
  from Tkinter import *

  class App:
    def __init__(self, root):

      # Conteneur principal
      frame_princ = Frame(root)
      frame_princ.pack(anchor=NW)   # par défaut side=TOP et anchor=CENTER

      # Crée un conteneur pour les boutons
      frame_top = Frame(frame_princ)
      frame_top.pack(anchor=NW)

      button = Button(frame_top, text="QUIT", fg="red", command=root.quit)
      button.pack(side=LEFT)

      button = Button(frame_top, text="Vider", command=self.on_vider_button)
      button.pack(side=LEFT)

      button = Button(frame_top, text="Nouveau", command=self.on_nouveau_button)
      button.pack(side=LEFT)

      # Crée un canvas sous la zone de boutons
      # On mémorise canvas comme attribut
      self.canvas = Canvas(frame_princ, bg="white", width=400, height=300);
      self.canvas.pack();

      # Enregistre des dessins dans le canvas
      self.canvas.create_line(0, 0, 100, 50, 500, 100, fill="blue")
      self.canvas.create_rectangle (50, 100, 250, 170, outline="green",
        fill="yellow")
```

```
    def on_vider_button (self):
      self.canvas.delete(ALL)

    def on_nouveau_button (self):
      self.canvas.create_oval(100, 100, 300, 250, outline="red")

  root = Tk()
  app = App(root)
  root.mainloop()
```

- Réagir aux évènements

  On peut déclencher des callbacks pour toutes sortes d'évènements, voir
  http://infohost.nmt.edu/tcc/help/pubs/tkinter/events.html

  Il suffit d'attacher une callback à un widget avec la méthode .bind()
  en lui donnant le nom de l'évènement sous la forme d'une chaîne de
  caractères.

```
  from Tkinter import *

  class App:
    def __init__(self, root):

      # Conteneur principal
      frame_princ = Frame(root)
      frame_princ.pack(anchor=NW)   # par défaut side=TOP et anchor=CENTER

      # Crée un conteneur pour les boutons
      frame_top = Frame(frame_princ)
      frame_top.pack(anchor=NW)

      button = Button(frame_top, text="QUIT", fg="red", command=root.quit)
      button.pack(side=LEFT)

      button = Button(frame_top, text="Vider", command=self.on_vider_button)
      button.pack(side=LEFT)

      # Crée un canvas sous la zone de boutons
      self.canvas = Canvas(frame_princ, bg="white", width=400, height=300);
      self.canvas.pack();

      # Attache une callback au bouton 1. Pour une touche "H" du clavier,
      # ce serait "<KeyPress-H>" ou encore "<Control-Shift-KeyPress-H>"
      self.canvas.bind ( "<Button-1>", self.on_canvas_button1_press)
      self.canvas.bind ( "<ButtonRelease-1>", self.on_canvas_button1_release)
      self.canvas.bind ( "<Motion>", self.on_canvas_motion)

    def on_vider_button (self):
      self.canvas.delete(ALL)

    def on_canvas_button1_press (self, event):
      print ("button1_press %d %d" % (event.x, event.y))
      self.canvas.create_oval (event.x-5, event.y-5, event.x+5, event.y+5)

    def on_canvas_button1_release (self, event):
      print ("button1_release %d %d" % (event.x, event.y))

    def on_canvas_motion (self, event):
      print ("Motion %d %d" % (event.x, event.y))

  root = Tk()
  app = App(root)
  root.mainloop()
```

- Exercice : dessiner à la souris

  On déclare des attributs lastx, lasty pour mémoriser l'ancienne position
  de la souris, et un attribut clic1 pour savoir si le bouton est enfoncé.

```
  from Tkinter import *
```

```
class App:

  lastx = lasty = clic1 = 0

  def __init__(self, root):

    # Conteneur principal
    frame_princ = Frame(root)
    frame_princ.pack(anchor=NW)   # par défaut side=TOP et anchor=CENTER

    # Crée un conteneur pour les boutons
    frame_top = Frame(frame_princ)
    frame_top.pack(anchor=NW)

    button = Button(frame_top, text="QUIT", fg="red", command=root.quit)
    button.pack(side=LEFT)

    button = Button(frame_top, text="Vider", command=self.on_vider_button)
    button.pack(side=LEFT)

    # Crée un canvas sous la zone de boutons
    self.canvas = Canvas(frame_princ, bg="white", width=400, height=300);
    self.canvas.pack();

    # Attache une callback au bouton 1. Pour une touche "H" du clavier,
    # ce serait "<KeyPress-H>" ou encore "<Control-Shift-KeyPress-H>"
    self.canvas.bind ( "<Button-1>", self.on_canvas_button1_press)
    self.canvas.bind ( "<ButtonRelease-1>", self.on_canvas_button1_release)
    self.canvas.bind ( "<Motion>", self.on_canvas_motion)

  def on_vider_button (self):
    self.canvas.delete(ALL)

  def on_canvas_button1_press (self, event):
    print ("button1_press %d %d" % (event.x, event.y))
    self.canvas.create_oval (event.x-5, event.y-5, event.x+5, event.y+5)
    self.clic1 = 1
    self.lastx = event.x
    self.lasty = event.y

  def on_canvas_button1_release (self, event):
    print ("button1_release %d %d" % (event.x, event.y))
    self.clic1 = 0

  def on_canvas_motion (self, event):
    print ("Motion %d %d" % (event.x, event.y))
    if (self.clic1 == 1):
      self.canvas.create_line(self.lastx, self.lasty, event.x, event.y,
        fill="blue")
      self.lastx = event.x
      self.lasty = event.y

root = Tk()
app = App(root)
root.mainloop()
```

8) TP traceur de courbes

- Evaluation d'expressions

  On va utiliser la fonction eval de python qui permet d'évaluer une expression arithmétique :

```
>>> t=5
>>> eval ("2*t")
10
```

  On peut aussi spécifier les variables sans les affecter :

```
>>> eval ("2*y+3*x*x+4", {}, {"x":2, "y":5})
```

```
  26
```

La syntaxe est : eval (expression, var globales, var locales)
Attention cette syntaxe vide les espaces de nom, donc empêche d'utiliser
par exemple les fonctions de math, ou d'autres fonctions potentiellement
dangereuses --> il faut donner la liste des fcts autorisées :

```
  >>> import math
  >>> math.cos(math.pi/4)
  0.7071067811865476
  >>> eval ("math.cos(math.pi/4)", {}, {})
  NameError: name 'math' is not defined
  >>> eval ("cos(pi/4)", {}, {"cos":math.cos,"pi":math.pi})
  0.7071067811865476
```

- Le TP consiste à programmer un traceur de courbes y=f(x).
  - placer en haut de la fenêtre des boutons : Quitter, Vider, Tracer ;
  - placer en dessous à gauche le canvas
  - placer en dessous à droite les paramètres :
    "f(x) =", "xmin = ", "ymin = ", "xmax = ", "ymax = ", "nb points ="

  Il faudra créer plusieurs frames emboîtés.

  Lorsqu'on presse le bouton Tracer,
  - le repère est tracé en bleu
  - la fonction f(x) est échantillonnée sur nb points de xmin à xmax,
    et des segments de droites reliant les points sont tracés.

  Ne pas oublier d'inverser le sens des y en dessinant car l'origine de
  la fenêtre est en haut à gauche avec y vers le bas !

  Attention aux changements de coordonnées ...

  Voici une solution :

```
  # ---------------------------------------------------------------------------

  from Tkinter import *

  class App:

    def __init__(self, root):

      # Conteneur principal
      frame_princ = Frame(root)
      frame_princ.pack(anchor=NW)   # par défaut side=TOP et anchor=CENTER

      # Crée un conteneur pour les boutons
      frame_top = Frame(frame_princ)
      frame_top.pack(anchor=NW)

      button = Button(frame_top, text="QUIT", fg="red", command=root.quit)
      button.pack(side=LEFT)

      button = Button(frame_top, text="Vider", command=self.on_vider_button)
      button.pack(side=LEFT)

      button = Button(frame_top, text="Tracer", command=self.on_tracer_button)
      button.pack(side=LEFT)

      # Crée un conteneur pour séparer le canvas des paramètres
      frame_horiz = Frame(frame_princ)
      frame_horiz.pack()

      # Crée un canvas sous la zone de boutons
      self.canvas = Canvas(frame_horiz, bg="white", width=400, height=300);
      self.canvas.pack(side=LEFT);

      # Crée un conteneur pour les paramètres
      frame_params = Frame(frame_horiz)
      frame_params.pack(side=LEFT, anchor=NW)

      # Les paramètres
```

```
        self.entry_fx   = self.creer_entry(frame_params,
                            "f(x) =", "2*x*x-5*x+3", "30")
        self.entry_xmin = self.creer_entry(frame_params, "xmin =", "-5", "15")
        self.entry_ymin = self.creer_entry(frame_params, "ymin =", "-5", "15")
        self.entry_xmax = self.creer_entry(frame_params, "xmax =", "5", "15")
        self.entry_ymax = self.creer_entry(frame_params, "ymax =", "5", "15")
        self.entry_nbp  = self.creer_entry(frame_params,
                            "nb points =", "100", "15")

      def creer_entry(self,owner,titre,valeur,largeur):
        label = Label(owner, text=titre)
        label.pack(anchor=NW)
        entry = Entry(owner, width=largeur)
        entry.pack(anchor=NW)
        entry.delete(0, END)
        entry.insert(0, valeur)
        return entry

      def on_vider_button (self):
        self.canvas.delete(ALL)

      def on_tracer_button (self):
        self.canvas.delete(ALL)

        # On récupère la taille du canvas
        w = self.canvas.winfo_width()-2
        h = self.canvas.winfo_height()-2
        print ("w,h = %d,%d" % (w,h))

        # On récupère les paramètres de l'utilisateur
        fx=self.entry_fx.get()
        xmin=float(self.entry_xmin.get())
        ymin=float(self.entry_ymin.get())
        xmax=float(self.entry_xmax.get())
        ymax=float(self.entry_ymax.get())
        nbp=int(self.entry_nbp.get())

        if (xmin == xmax or ymin == ymax) :
          return;

        # On dessine les axes
        x = w*xmin/(xmin-xmax)
        y = h-h*ymin/(ymin-ymax)  # on inverse les y
        print ("Origine = %d,%d" % (x,y))
        self.canvas.create_line(x, 0, x, h, fill="blue")
        self.canvas.create_line(0, y, w, y, fill="blue")

        for i in range(0,nbp+1):
          x = xmin+i*(xmax-xmin)/nbp
          y = float(eval(fx, {}, {"x" : x} ))
          # print ("i = %d  x = %d  y = %d" % (i,x,y))
          if i > 0:
            self.canvas.create_line(
              w*(ox-xmin)/(xmax-xmin), h - h*(oy-ymin)/(ymax-ymin),
              w*( x-xmin)/(xmax-xmin), h - h*( y-ymin)/(ymax-ymin))
          ox = x
          oy = y

    root = Tk()
    app = App(root)
    root.mainloop()

    # --------------------------------------------------------------------
```