GRENOBLE INP UGA Formation Pro

INSTITUT D'INGÉNIERIE ET DE MANAGEMENT

LYNRED

Bien débuter en Python scientifique

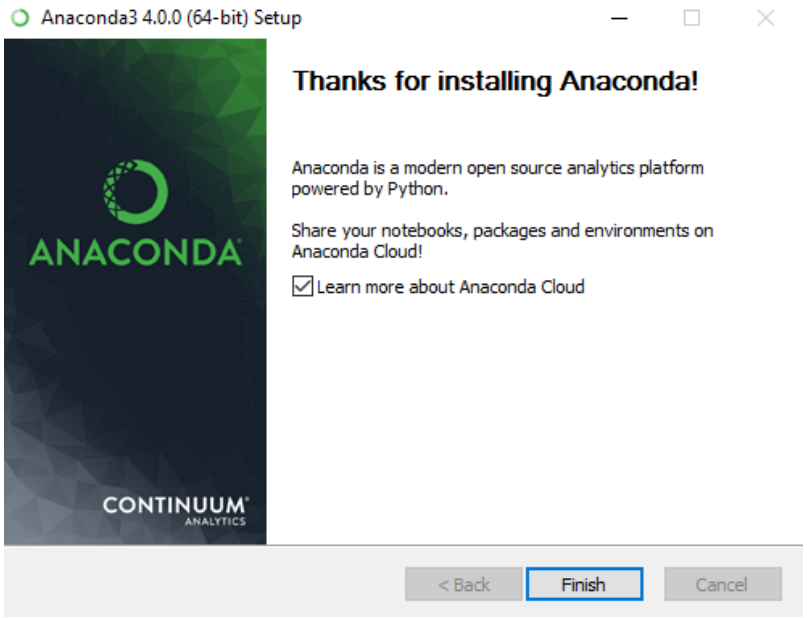
Cours et Travaux dirigés

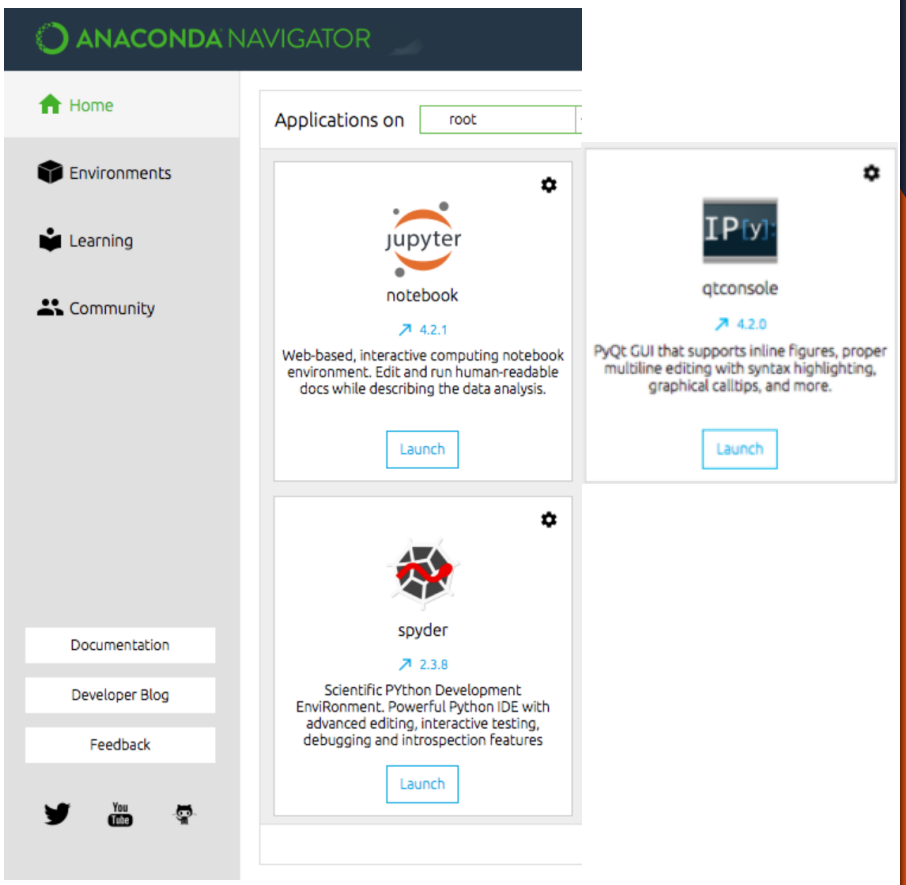
8, 10 et 15 mars 2022

Technologies du numérique | Eau, Environnement | Energies | Matériaux, Mécanique | Conception et organisation industrielle | Micro/Electronique, Nanotechnologies

Installation de la distribution Anaconda

https://docs.continuum.io/anaconda/install

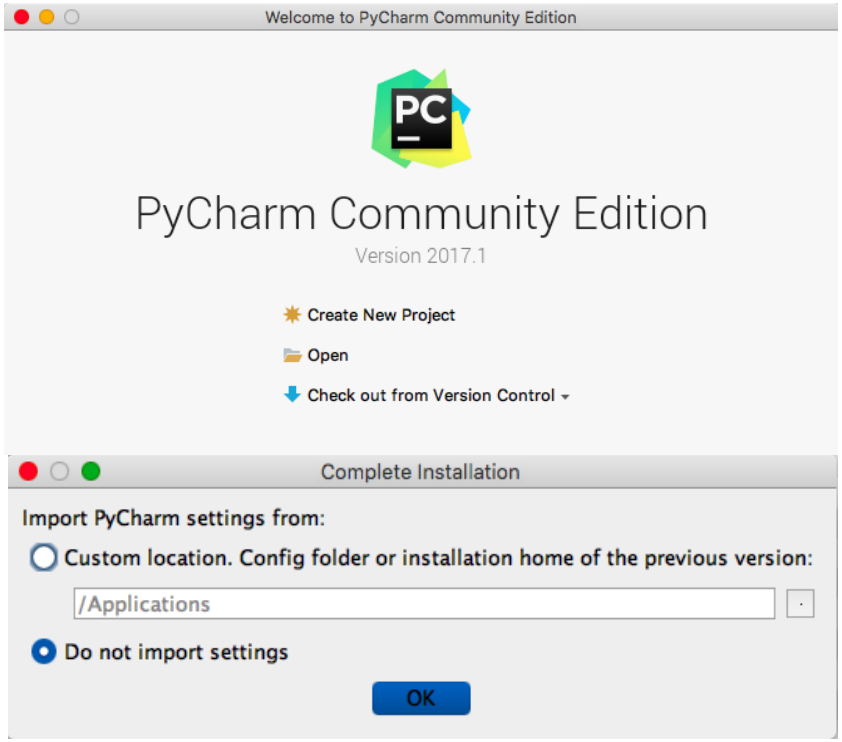


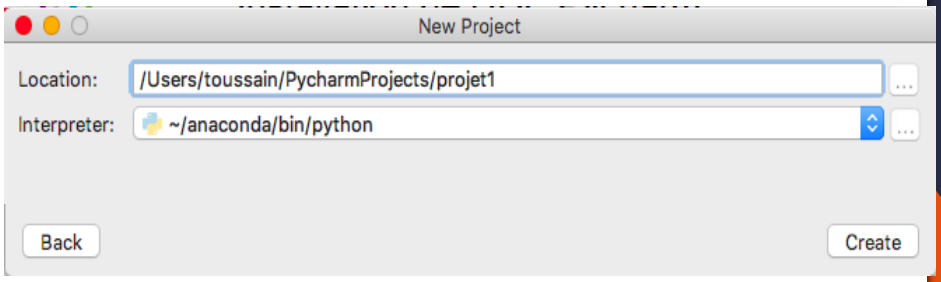


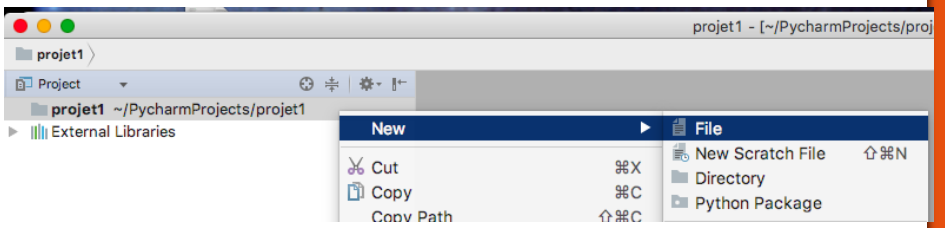
Ipython : interpréteur ligne de commande
Spyder : client GUI
Jupyter : client web

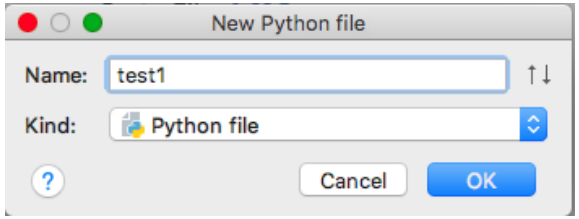
Installation de l'IDE Pycharm

https://www.jetbrains.com/pycharm









Arithmétique de base

TABLE: Operations Python

commande	nom	exemple	sortie
+	addition	4+5	9
*	multiplication	4*5	20
/	division	18/3	6
//	division entière	19//3	6
%	reste	19%3	1
**	puissance	2**4	16

Arithmétique

La priorité des opérations est la même qu'en mathématiques :

1. Parenthèses ()
2. Puissance **
3. Multiplication, Division, reste
4. Addition et Soustraction

Expressions booléennes

- Deux valeurs possibles : `False, True.`

Operateurs de comparaison : == != > >= < <=

```
>>> 2 > 8 # False

>>> 2 <= 8 < 15 # True
```

- Operateurs logiques (notion de shortcut) : `not, or, and`

```
>>> (3==3) or (9 >24) # True (des le premier membre)

>>> (9 > 24) and (3==3) # False (des le premier membre)
```

Les données et les variables

Variable

nom donné à une valeur = **référence** à une adresse mémoire.

- Les noms des variables sont conventionnellement écrits en minuscule; ils commencent par une lettre ou le caractère souligne (_), puis, eventuellement, des lettres, des chiffres
- Ils doivent etre différents des mots réserves de Python :

```
import keyword
keyword.kwlist  # liste des mots clés python
```

L'affectation

On **affecte** une valeur à une variable en utilisant le signe =

```
a=10 # forme de base

a+=1 # meme chose que a = a + 1, si a est deja connue;
     # avantage : si a est une expression complique, on
     # a besoin d'une seule evaluation de a ;
     # forme standard : a est evalue deux fois.

b = c = 10   # cibles multiples
             # (affectation de droite a gauche)

d, e, f = 1.5, 2.5, 3.5 # affectation de tuple

                        # (par position)

e, d = d, e # echange les valeurs de d et e

m, n = [5,7] # affectation de liste (par position).
```

Typage

Python détecte automatiquement **le type** des variables :

```
# definition d'un entier valant 1
a=1
print(type(a))
b=a//2  # division entière
print(b)
print(type(b))
```

<class ‘int'>

<class 'float'>

```
# definition d'un nombre reel
a=1.0
print(type(a))
```

<class 'float'>

```
b=a/2
print(b)
print(type(b))
```

<class 'float'>

Conversion de type (1/2)

Conversion string → integer ou float

```
s = '789'
print(type(s))

# conversion vers un entier
i = int(s)

# conversion vers un flottant
d = float(s)
```

<class 'string'>

Que se passe-t-il

si str contient '789.12' ?

Conversion integer ou float → string

```
d = 789.12
print(type(d))

# conversion vers un string
s = str(d)
```

<class 'float'>

Conversion de type (2/2)

Conversion tuple → list et list → tuple

```
tup = (1, 2, 3)
print(tup)

lst=list(tup)
print(lst)
```

```
lst = [1, 2, 3]
print(lst)

tup=tuple(lst)
print(tup)
```

Conversion generator → list

```
gen = range(10)
print(type(gen))
print(gen)

lst=list(gen)
print(lst)
```

<class 'range'>
range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Les entrées / sorties

- L'instruction **input** permet de saisir une entrée au clavier. Elle retourne une chaine de caractères. Cela impose de faire une conversion de type explicite

```
n = int(input("Entrez une valeur entière :"))
Entrez une valeur entière : 7

x = float(input("Entrez une valeur réelle :"))
Entrez une valeur réelle : 1.1
```

- L'instruction **print** permet d'afficher des sorties à l'écran :

```
>>> print("La somme : a+b est : ", a+b)
```

Les sorties formatées

- La fonction membre **format** de la classe str permet de formater les sorties

```
x=123.456; y=23.45
s='{:+10.2f}'.format(x)  ou s=f'{x:+10.2f}'
print(s)                                  #___+123.46

s='{:+f} {:+g}'.format(3.14, -3.14)
print(s)                                  # +3.140000 -3.14
```

- De changer l'ordre de sortie

```
s='x={0}, y={1}'.format(x, y)
print(s)                                  #x=123.456, y=23.45

s='{2}, {1}, {0}'.format('a', 'b', 123)
print(s)                                  #123, b, a
```

- Autres exemples

```
s="x : {0:8.2f},  y: {1:8.2f}".format(10.2, 1.2)
print(s)                                  #x :    10.20,  y:     1.20

s="Art: {a:5d},  Prix: {p:8.2f}".format(a=59, p=102.2)
print(s)                                  #Art:    59,  Prix:   102.20
```

Instructions conditionnelles

But : branchement conditionnel

```
if {conditions}:
        {executer ce block}
elif {conditions}:
        {executer ce block}
elif {conditions}:
        {executer ce block}
else:
        {executer ce block}
```

- Il est possible, après un **if** de tester autant de conditions que l'on souhaite avec des **elif** (y compris 0); par contre, il ne peut y avoir qu'un seul **else** à la fin.
- Le caractère **:** est obligatoire a la fin de chaque ligne contenant **if**, **elif** ou **else**.

Instructions conditionnelles

Trouver, par exemple, le minimum de deux nombres

```
x, y = 10, 20

# Ecriture classique
if x<y:
    min=x
else:
    min=y

# Utilisation de l'opérateur ternaire
min=x if x<y else y
```

Boucle WHILE

But : Répéter une portion de code.

```
while {condition a laquelle la boucle continue}:
      {code execute dans la boucle}
      # indentation necessaire, d'habitude 4 espaces;

{code qui ne sera pas execute dans la boucle}
# car il n'est pas indente.
```

Exemple:

```
x = 10
while x != 0:
      x=x-1
      print (" décroit, valeur non nulle egale à", x)
print (f"La boucle est finie, x = {x}")
```

Boucle FOR

But : Répéter une portion de code pour chaque valeur d'une séquence (chaine, liste ou n-uple, range).

```
for {valeur} in {sequence}:
     {code execute dans la boucle}
     # indentation necessaire, d'habitude 4 espaces

{code qui ne sera pas execute dans la boucle}
# car il n'est pas indente
```

Exemple:

```
for lettre in "bonjour":
     print(lettre)
#bonjour

for i in range(10):
    print(i, end=’’)
#0..9
```

Les listes : definition et exemples

Collection **hétérogène, ordonnée** et **modifiable** d'éléments, séparés par des virgules, et entourée de crochets.

```
fruits = ['figue', 'raisin', 'abricot', 'poire']
```

Opérations de base :

```
print(fruits[2])      # abricot
fruits[1]=13          # numérotation des éléments commence à 0
print(fruits)         # ['figue', 13, 'abricot', 'poire']
print(fruits[0:2])    # ['figue', 13]
```

Les listes : initialisation

Utilisation de la répétition et de l'instruction 'range()':

```
debut = []                  # liste vide
repet = [0.0] * 3           # [0.0, 0.0, 0.0]

liste_1=list(range(4))      # [0, 1, 2, 3]
liste_2=list(range(4,10))   # [4, 5, 6, 7, 8, 9]
```

Concaténation de listes:

```
liste_3=liste_1+liste_2
```

Liste de listes :

```
liste_3=[liste_1, liste_2]
print(liste_3)
# [[0, 1, 2, 3], [4, 5, 6, 7, 8, 9]]
```

Les listes : méthodes

Pour ajouter une valeur en fin de liste, utiliser la fonction 'append()'

```
fruits = ['figue', 'raisin', 'abricot', 'poire', 'abricot']
fruits.append(18)
print(fruits)
# ['figue', 'raisin', 'abricot', 'poire', 'abricot', 18]
```

Pour effacer une valeur d'une liste, utiliser la fonction 'del' :

```
del fruits[1]       # supprime 'raisin'

print(fruits)
# ['figue', 'abricot', 'poire', 'abricot', 18]
```

Pour effacer la première occurrence d'une liste, utiliser la fonction 'remove' :

```
fruits.remove('abricot')
```

Pour afficher les méthodes de la classe "liste": `help(list)`

Les listes : techniques de "slicing"

1) Extraction d'une sous-liste : `print(fruits[1:3])` **# ['poire', 'abricot']**
   1 => extraction à partir de l'élément 1 = 2ème élément de la liste
   3 => extraction jusqu'à l'élément 3 **non inclus**

2) Insertion d'éléments dans une liste.

Règle : dans le membre de gauche d'une affectation, il faut obligatoirement indiquer une tranche pour effectuer une insertion.

Le membre de droite doit lui-même être une liste.

```
print(fruits)
#  ['figue', 'poire', 'abricot', 18]

fruits[2:2]=['orange'] # insertion en 3eme position


print(fruits)
#['figue', 'poire', 'orange', 'abricot', 18]
```

Les listes : techniques de "slicing"

Suppression et remplacement d'éléments dans une liste.

Règle : une tranche dans le membre de gauche, une liste dans le membre de droite

```
print(fruits)
#  ['figue', 'poire', 'orange', 'abricot', 18]

fruits[1:3]=[] # effacement par affectation d'une liste vide

print(fruits)
# ['figue', 'abricot', 18]


fruits[2:4] = [1.5, 2.5]  # remplacement des éléments 2 et 3


print(fruits)
#  ['figue', 'abricot', 1.5, 2.5]
```

Les listes : parcours

Parcourir tous les éléments d'une liste.

Utilisation d'une boucle for :

```
for e in [11,12,13]:
    print(e)
```

```
11
12
13
```

Utilisation du générateur enumerate :

```
liste = [1,2,3,5,7,11,13]

for index, elem in enumerate (liste):
    print(index, elem)
```

```
0 1
1 2
2 3
3 5
4 7
5 11
6 13
```

Les tuples : définition et exemples

Collection **hétérogène, ordonnée** et **non-modifiable** d'éléments, séparés par des virgules, et entourée de parenthèses.

```
parametres = (200, 'pression', 10, 'vitesse')

v1, leg1, v2, leg2 = parametres
```

• A la différence des listes, les n-uples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

• Leur parcours est plus rapide que celui des listes.

• ils sont utiles pour définir des constantes, paramètres etc.

Les dictionnaires : définition et exemples

Collection de couples clé-valeur entourée d'accolades

```
d = {   'Bureau Marie' : 201 , 'Bureau Pierre' : 202 , \
        'Bureau Alain' : 203 , 'Bureau Jeanne' : 204}
```

- On les appelle aussi "tableaux associatifs"
- Les indices sont appelés "clés"
- d[Nom] affiche l'element dont la clé est Nom : d['Bureau Marie'] → 201
- d.keys() retourne un itérable contenant les clés de d :

  dict_keys(['Bureau Marie', 'Bureau Alain', 'Bureau Pierre', 'Bureau Jeanne'])

- d.items() retourne un itérable contenant les clés et éléments du dictionnaire.

  dict_items([('Bureau Marie', 201), ('Bureau Alain', 203), ('Bureau Pierre', 202), ('Bureau Jeanne', 204)])

Les dictionnaires : parcours

Parcourir toutes les clés d'un dictionnaire.

```
d = {   'Bureau Marie' : 201 , 'Bureau Pierre' : 202 , \
        'Bureau Alain' : 203 , 'Bureau Jeanne' : 204}
```

Utilisation d'une boucle for et du générateur d.keys():

```
for key in d.keys() :
    print(key)
```

```
Bureau Pierre
Bureau Jeanne
Bureau Alain
Bureau Marie
```

Utilisation du générateur d.items :

```
for key, elem in d.items() :
    print(key, elem)
```

Attention, ordre bouleversé !

```
Bureau Pierre 202
Bureau Jeanne 204
Bureau Alain 203
Bureau Marie 201
```

Les fonctions : définition et exemples

Groupe d'instructions regroupé sous un **nom** et s'exécutant à la demande de l'utilisateur, lors de **l'appel.**

Exemple :

```
import math
math.abs(x)

# Return the absolute value of a number. The argument
# may be a plain or long integer or a floating point
# number. If the argument is a complex number, its
# magnitude is returned.
```

Les fonctions : syntaxe

```
def nom_fonction(liste des parametres):
"""Documentation de la fonction."""   # doc-string
    { code execute dans la fonction }
# indentation necessaire
    return {valeur retournee dans le programme principal}
```

Dans la définition précédente, **nom_fonction** est le nom qui sera utilisée ultérieurement pour appeler la fonction ;

• Le bloc d'instructions est obligatoire. S'il ne fait rien ou si le code n'est pas encore écrit, on utilise l'instruction **`pass`**.

• Une fonction peut retourner plusieurs valeurs sous la forme d'un tuple.

•La documentation (doc-string) est fortement conseillée.

•help(nom_fonction) retourne la doc-string

Les fonctions : exemples

```
def somme():
        """ somme des 10 premiers entiers"""
        s=0
        for x in range(1, 11):
            s+=x
        return s

# appel de la fonction
print ("La somme est ", somme())# La somme est 55
help(somme)
```

Avec un paramètre d'entrée :

```
def somme(N):
        """ somme des N premiers entiers"""
        s=0
        for x in range(1, N+1):
            s+=x
        return s
# appel de la fonction
print ("La somme des 20 premiers entiers est", somme(20))
```

Les fonctions : exemples

Avec plusieurs valeurs de retour :

```
def som_dif(x, y):
        """ somme et différence """
        return x+y, x-y

# appel de la fonction
print ("somme et difference sont :", som_dif(10, 20))
help(som_dif)
```

somme et difference sont : (30, -10) *# valeurs de retour dans un tuple*
Help on function som_dif in module __main__:
som_dif(x, y)
    somme et difference

```
# affectation des valeurs retournées à des variables
som, dif = som_dif(10, 20)
```

Les fonctions : valeurs par défaut et passage de paramètres par nom

```
def f(x, y, z=100): # 100 est la valeur par défaut de z
        print('x=', x)
        print('y=', y)
        print('z=', z)

# appel de la fonction
f(10, 20, 30)

# On peut aussi passer les paramètres en spécifiant
# explicitement leur nom (passage par nom)
f(x=10, y=20) # z prend sa valeur par défaut

# passage de paramètres en spécifiant leur nom
f(y=20, x=10) # z prend sa valeur par défaut

f(y=20) # ERREUR manque x car pas de valeur par défaut
```

Les fonctions à nombre d'arguments variable (1/2)

```
# fonction à nombre d'arguments variable
# Les arguments sont transformés en tuple

def varArg(*ar) :
    print(type(ar))
    for e in ar :
        print(type(e), ' ', e)

varArg('toto', [1, 2, 3], [4, 5])


# <class 'tuple'>
# <class 'str'>   toto
# <class 'list'>   [1, 2, 3]
# <class 'list'>   [4, 5]
```

Les fonctions à nombre d'arguments variable (2/2)

 # fonction à nombre d'arguments variable
 # Les arguments sont transformés en dictionnaire (ils doivent être nommés !)

```
def varKwarg(**ar) :
    print(type(ar))
    print(ar.keys())
    print(ar.items())
    for key,value in ar.items() :
        print(key, value)  # ar[key]

varKwarg(k=10, l='too')
#  <class 'dict'>
#  dict_keys(['l', 'k'])
#  dict_items([('k', 10), ('l', 'too')])
#    l too
#    k 10
```

Philosophie Objet

Dès la conception d'une classe, l'un des points importants en POO est le choix des méthodes proposées à un utilisateur extérieur de la classe.

Il faut donc adopter une approche en termes de services rendus par la classe.

Pour établir des **services** efficaces, se placer en tant qu'**utilisateur extérieur** de la classe et définir les méthodes qui lui seront utiles.

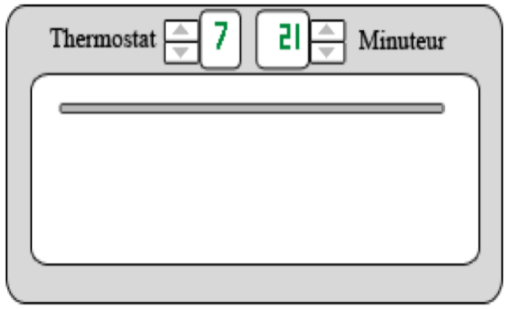
Donc il faut construire la classe depuis l'extérieur sans réfléchir à ses données privées (utiles pour son fonctionnement).

Ne pas offrir des accès inutiles aux données membres.

Exemples pratiques

Soit un four électrique, seuls 4 boutons sont accessibles pour l'utiliser :

ce sont les services offerts à l'utilisateur



Le four gère beaucoup de choses en interne :

- S'allumer si minuteur > 0 mn
- Décrémenter le compteur associé au temps de cuisson
- Réguler la température d'après la consigne

Autre exemple : classe string en python

L'utilisateur ne se soucie pas de la gestion interne de la classe et de ses optimisations.

Programmation Objet avec Python

- Instanciation : construction d'un objet, appel de la méthode __init__(self) ou __init__(self, args)

- Quand une méthode est appelée lors de l'exécution, une référence vers l'objet principal est passée en tant que premier argument (self).

  Par convention, self est toujours le premier argument

Une classe Python

Définition : l'argument self est obligatoire pour la définition

```
class Personne(object):
    def __init__(self , nom=None):   # None valeur par défaut
        self.__nom = nom              # __nom donnée privée

    def who(self):
        return self.__nom
```

Utilisation de la classe Personne :

```
pers=Personne('titeuf')   # pers est une instance de la classe Personne
print(pers.who())         # op • donne accès aux méthodes publiques

# pers.__nom='bill'   → impossible __nom est privée
```

Accesseur (getter) et mutateur (setter)

Le concepteur de la classe se pose la question : Quelles sont les données et Fonctions de ma classe, utiles aux autres classes, i.e. au monde extérieur ?

•Ce qui est utile pour le monde extérieur est public.
•Ce qui relève du fonctionnement interne de ma classe est private.

⇒L'extérieur ne peut modifier le comportement interne de la classe et voulu par le concepteur : c'est l'encapsulation

En pratique, les attributs d'une classe sont souvent privés
•Si besoin, autoriser la lecture de l'attribut par un accesseur (getter) à définir
•Parfois, autoriser la modification d'un attribut par un mutateur (setter) à définir

•C'est le concepteur de la classe qui décide de l'accès possible aux données membres privées de la classe.

Une classe Robot

```
class Robot(object):
    def __init__(self, name=None, build_year=None):
        self.__name = name
        self.__build_year = build_year

    def say_hi(self):
        if self.__name:
            print("Hi, I am " + self.__name)   # les fonctions membres ont accès
        else:                                  # aux données privées
            print("Hi, I am a robot without a name")

    def set_name(self, name):    # setter
        if isinstance(s, str):
            self.__name = name

    def get_name(self):      # getter
        return self.__name

    def set_build_year(self, by):
        self.__build_year = by

    def get_build_year(self):
        return self.__build_year
```

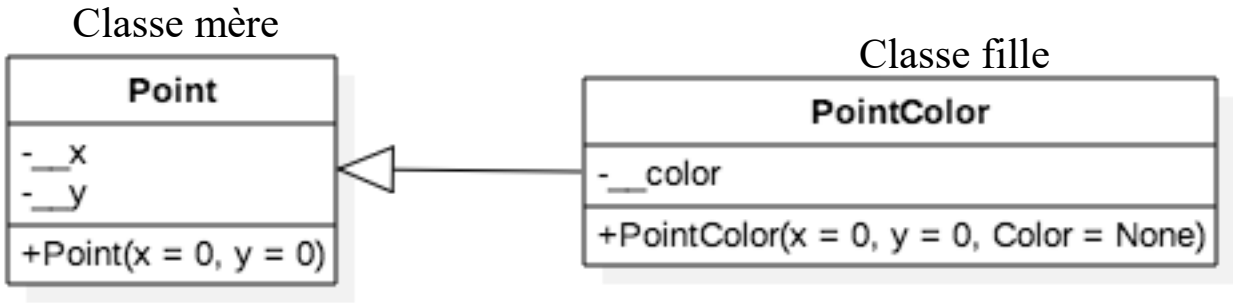
```
rob1 = Robot("Henry", 2008)
rob2 = Robot()

name=rob1.get_name()
rob2.set_name(name)

print(rob1.get_name(), rob1.get_build_year())
print(rob2.get_name(), rob2.get_build_year())
```

Héritage en Programmation Objet

- Héritage : une classe dérivée issue d'une classe parente
- données membres et méthodes héritées de la classe parente ne sont plus à redéfinir. Les méthodes de la classe parente sont utilisables (sous condition) dans la classe dérivée.
- Ajout dans la classe dérivée de nouvelles fonctionnalités sans tout redévelopper.
- Cloisonnement des nouveaux éléments pour sécuriser le code existant.

Héritage en python

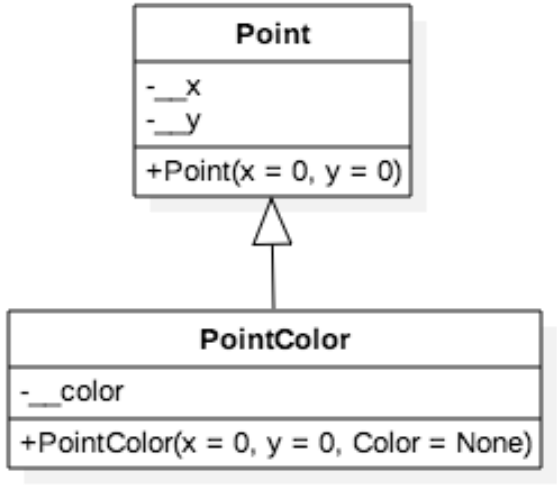
L'héritage en python est exclusivement public comme en java



Exercice : développer ces deux classes, afficher le contenu d'une instance de chaque classe

Correction : classes Point et PointColor

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.__x=x
        self.__y=y

    def __repr__(self):   # appelé lors d'un print
        return str(self.__x) + ', ' + str(self.__y)
```




```
class PointColor(Point):
    def __init__(self, x=0, y=0, color=None):
        super().__init__(x, y)        # appel du constructeur de la classe Point
        self.__color=color

    def __repr__(self):   # appelé lors d'un print
        return super().__repr__()+ ',' + str(self.__color)

point=Point(10, 20)                   # super().__repr__() appelle la fonction
print(point)                          # magique __repr__() de la classe Point

pointcol=PointColor(10, 20, 255)
print(pointcol)
```

Les modules : utilisation

Import d'un module : deux syntaxes possibles.

•La commande **`import`** `<nom module>`

importe la totalite des objets du module.

```
import math

y = math.cos(x) # seul math.cos(x) est chargé en mémoire ici
```

•La commande **`from`** `<nom module>` **`import`** `*` existe mais déconseillée.

•La commande **`from`** `<nom module>` **`import`** `obj1, obj2 …`

n'importe que les objets `obj1, obj2 ...` du module.

```
from math import pi, sin, log
y= cos(x)
```

Le module : math

Le module math permet d'importer les fonctions et les constantes mathématiques usuelles.

commande Python	constante / fonction math
`pi, e, exp(x)`	π, e = exp(1), exp(x)
`log(x), log(x,a)`	ln(x), *loga(x)*
`pow(x,y), floor(x), abs(x), factorial(n)`	x^y, [x], |x|, n!
`sin(x), cos(x), tan(x), asin(x), ...`	fonctions trigonometriques
`sinh(x), cosh(x), tanh(x), asinh(x), ...`	fonctions hyperboliques

Le module : random

Le module `random` propose diverses fonctions permettant de generer

des nombres pseudo-aléatoires qui suivents differentes distributions math.

```
random.seed(123) # initialise la graine
random.randrange(p, n, h)# choisit un entier aleatoirement dans range(p,n,h)

random.randint(a, b)
# choisit un entier N aleatoirement tel que
# a <= N <= b.
random.choice(seq)# choisit un aleatoirement dans la sequence seq

random.random()# renvoie un decimal aleatoire dans [0,1[

random.uniform(a, b)
# choisit un reel N aleatoirement, tel que
# a <= N <= b si a <= b et b <= N <= a si b < a.
```

Comment créer son propre module ?

On crée un fichier `fib_gen.py` regroupant l'unique fonction python `fib` pour calculer la suite de Fibonacci. Le pgm `test_fib.py` importe le module

```
""" fibonacci series """

def fib_gen(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

# test de la fonction fib
if __name__ == "__main__":
    print(fib_gen(10))
```

```
# test_fib_gen.py
import fib_gen as fm

# test de la fonction fib_gen
print(fm.fib_gen(10))
```

Uniquement exécuté lorsque `fib_gen.py` est le programme principal

Inconvénient : désordre car tous les fichiers se trouvent dans le même répertoire

Ouverture et fermeture des fichiers

Le type fichier est un type prédéfini de Python. Principaux modes d'ouverture :

```
with open("monFichier", "r") as f:
# ouverture en lecture seule

with open("monFichier", "w") as f:
# ouverture pour ecriture : si le fichier n'existe pas,
# il est créé, sinon son contenu est écrasé

with open("monFichier", "a") as f:
# ouverture pour ajout : si le fichier n'existe pas,
# il est créé, sinon l'ecriture s'effectue a la suite
# du contenu déjà existant

with open("monFichier", "+") as f:
# ouverture pour lecture et écriture
```

Une seule méthode de fermeture : `f.close()`

Utilisation des fichiers

```
with open('data', 'w') as f:
    s='toto'
    f.write(s + '\n')
    l=['a', 'b', 'c']
    f.writelines(l)
    f.close() # pas nécessaire avec un with


with open('data', 'r') as f:
    s=f.readline()
    print(s)
    s=f.readline()
    print(s)
```

Sauvegarde de données dans fichier ASCII

Exemple de sauvegarde de données dans un fichier ASCII en format libre

```
import math
with open('data', 'w') as f:
    for k in range(4):
        x=math.pi*k/4
        str='{:+g} {:+g}'.format(x, math.cos(x))
        print(str)
        f.write(str + '\n')
```

```
+0 +1
+0.785398 +0.707107
+1.5708 +6.12323e-17
+2.35619 -0.707107
```

Sauvegarde de données dans fichier ASCII

Exemple de sauvegarde de données dans un fichier ASCII format float

```
import math
with open('data', 'w') as f:
    for k in range(4):
        x=math.pi*k/4
        str='{:+10.5f} {:+10.5f}'.format(x, math.cos(x))
        print(str)
        f.write(str + '\n')
```

```
+0.00000   +1.00000
+0.78540   +0.70711
+1.57080   +0.00000
+2.35619   -0.70711
```

Lecture de données ASCII

Exemple de lecture de données depuis un fichier ASCII

1ème version : lecture ligne à ligne

```
x=[]
y=[]

with open('data', 'r') as f:
    for line in f:
        lst = line.split()
        x.append(float(lst[0]))
        y.append(float(lst[1]))

print(x)
print(y)
```

Lecture de données ASCII

Exemple de lecture de données depuis un fichier ASCII:

2ère version : lecture de tout le fichier

```
x=[]
y=[]

with open('data', 'r') as f:
    lines=f.readlines() # toutes les lignes dans la liste lines
    for line in lines:
        lst = line.split()
        x.append(float(lst[0])) # conversion vers float
        y.append(float(lst[1]))

print(x)
print(y)
```

Le module de calcul numérique Numpy

Le module incontournable du calcul scientifique avec Python.

- Il s'installe comme un module Python standard. Il contient les fonctions de manipulation des tableaux pour le calcul numérique vectoriel.
- Bibliothèque mathématique importante.

```
import numpy as np

# les noms numpy sont accessibles avec le prefixe np
```

Les tableaux NumPy

`ndarray`

Collection indexable, contenant des éléments du même type et de la même taille.

- Contrairement aux listes, ils contiennent des objets de types identiques, comme des flottants, ou des entiers.
- Comparés aux listes simples, les objets fournis par NumPy sont optimisés pour le calcul numérique vectoriel
- La manipulation est similaire à tout autre objet Python,
- Les dimensions et parcours sont modifiables, les indexations souples

Création de tableaux

Differentes méthodes :

- par le contenu :

```
a=np.array([1, 3, 5, 7, 9, 11, 13, 17])
# liste de valeurs
```

- par la dimension (une liste : dimension 1, une liste de listes dimension 2, une liste de listes de listes : dimension 3 etc.) :

```
a=np.array([0.1, 0.0, 0.2])
b=np.array([[1, 2, 3], [4, 5, 6]])
```

- par le type d'élément:

```
a=np.array([0.1, 0.0, 0.2],dtype='float64')
b=np.array([[1, 2, 3], [4, 5, 6]],dtype='int8')
# dtype est optionnel dans toutes les creations de
# tableaux et fixe par defaut a float
```

Création de tableaux

```
np.zeros(N)
# tableau de N zeros

np.empty(N)
#tableau de N cases non initialisées


np.ones((N,M))
# tableau de dimension (N,M) de uns

np.zeros_like(a)
# tableau de zeros de meme dimension et type qu'un
# tableau a (idem np.ones_like, np.empty_like).

np.arange(M,N,P)
#tableau unidim allant de M a N avec pas P (M et P optionnels,
# les valeurs par defaut sont M=0 et P=1)
```

Modification du profil d'un tableau

Les dimensions d'un tableau et/ou le nombre de dimensions peuvent

être changés :

- `flat` : vue 1D d'un tableau sans modification :

```
np.eye((3)).flat[:]
# array([ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.])
```

  - `shape` : n-uplet des dimensions d'un tableau

```
a=np.ones((3,5,7))
a.shape
# (3, 5, 7)
```

  - modification des dimensions, en conservant la longueur totale :

```
a=np.ones((4,6))
a=a.reshape(8,3)
```

Copie et référence

```
b=a.copy()
# copie d'un tableau a dans un tableau b

b=a
# ne copie pas les elements, mais cree juste une reference,
i.e. si a est modifie, b l'est aussi.
```

Executer les scripts Python suivants et commenter.

```
a = np.ones((4,3))
b = a.transpose()
b[0,1] = 10
print(a)
```

```
a = np.arange(12)
b = a.reshape((4,3))
b[0,1] = 10
print(a)
```

Référencer les éléments d'un tableau

```
a=np.arange(24).reshape(2,3,4)

# Acces a un element dans le tableau
a[0][2][1]

# Syntaxe avec implementation optimisee pour l'acces
a[0,:]
a[0,:-1]

# La syntaxe fonctionne pour la reference et l'assignation
b=a[0:2]      # de 0 à 2 exclu
a[0:2]=9      # plein de 9
print(a)
print(b)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Opérations

Operations mathématiques : toutes les opérations arithmétiques et fonctions classiques s'appliquent terme à terme pour un tableau a :

```
a=np.eye(3)
2*a+1
a**2
1+np.sin(a) # sin de numpy vectorialisé
```

Operations sur l'ensemble du tableau

```
a=np.array([[0, 1], [2, 3]])
print(a.max())          # valeur max du tableau
print(a.argmax())       # localisation du 1er élément <-> max
print(a.flat[3])

# a.max(0) selon les lignes, a.max(1) selon les colonnes
# voir a.min() a.mean() a.prod()
```

Calcul numerique matriciel

Les matrices sont des tableaux à deux dimensions

Creation de matrices :

```
np.matrix([[0, 1], [2, 3]])

np.diag(v)
# matrice avec le vecteur v sur la diagonale.

np.diag(np.arange(N))
np.diag(np.ones(N))
np.diag(v,i)
# matrice avec le vecteur v sur la ieme diagonale  superieure.

np.diag(v,-i)
# matrice avec le vecteur v sur la ieme diagonale  inferieure.

np.identity(n)
# matrice identite de taille n
```

Opération sur les matrices

Ces opérations sont utilisees dans `numpy.linalg` :

```
np.dot(A,B)
# produit des matrices A et B

np.inner(x,y)
# produit scalaire des vecteurs x et y

np.linalg.det(A)
# determinant de la matrice A

np.linalg.inv(A)
# inverse de la matrice A

np.linalg.solve(A,b)
# solution x du systeme lineaire Ax=b
```

```
np.linalg.eigvals(A)
# valeurs propres de la matrice A

np.linalg.eig(A)
# valeurs et vecteurs propres de A
```

GRENOBLE INP UGA Formation Pro

Grenoble INP - UGA
Département Formation Pro
46 avenue Félix-Viallet
38031 Grenoble Cedex 1
France

formation-pro.grenoble-inp.fr

GROUPE INP — Cti Commission des Titres d'Ingénieur

GRENOBLE INP - UGA
INGÉNIERIE & MANAGEMENT