Cours 4 : Packages et objets

Ing-info — Ingénierie informatique

Christophe Gonzales

Plan du cours n° 4

Maths, aléatoire, fichiers. . .

- **Problèmes :**
  - utiliser des fonctions mathématiques (cosinus, *etc.*)
  - ouvrir/lire/écrire un fichier
  - exécuter des commandes système
  - effectuer des requêtes internet
  - *etc.*
- **Solution :** Utiliser du code déjà écrit : un module

Modules

- Module = **un fichier Python**
- Contient variables, fonctions, instructions, *etc.*
- Nom du module ≈ nom du fichier (sans le `.py`)
- Utiliser le fichier = l'importer
- Module importé $\Longrightarrow$ on peut utiliser tout ce qu'il définit

Et si le monde n'était pas réel ?

- Python connaît les nombres complexes :
  `z = complex(4,2)` $\Longrightarrow z = 4 + 2i$
- En maths, $\sqrt{4+2i} \approx 2.058 + 0.486i$
- **Problème :** comment calculer la racine carrée ?
- **3 solutions :**
  - Utiliser la fonction `sqrt` vue en cours ✖
  - Utiliser la fonction `sqrt` du module `math` ✖
  - Utiliser la fonction `sqrt` du module `cmath` ✔

$\Longrightarrow$ « récupérer les fonctions » du module `cmath` et utiliser `sqrt`

- **Problème :** et si on a aussi besoin du `sqrt` de `math` ?
  Comment discriminer `sqrt` de `math` de celui de `cmath` ?
- **Solution :** module $\Longrightarrow$ espace de nommage

Importations et nommage (1/3)

Importation d'un module

- `import` **`nom_module`**
- Fonction accessible via `nom_module.nom_fonction`

- ***Exemples :***

```
import cmath

z = complex(4,2)
print (cmath.sqrt(z))
```

```
import math
import cmath

z = complex(4,2)
print (cmath.sqrt(z))
x = 4.5
print (math.sqrt(x))
```


Importations et nommage (2/3)

Importation et alias

- Si module très utilisé : raccourcir le nom du module $\Longrightarrow$ alias
- `import` **`nom_module`** `as` **`nom_court`**
- Fonction accessible via `nom_court.nom_fonction`

- ***Exemple :***

```
import numpy as np

z = np.full(4,2)
```


Importations et nommage (3/3)

Importation d'une partie d'un module

- Ne charger qu'une seule fonction dans un module :
  `from` **`nom_module`** `import nom_fonction`
- Fonction accessible via `nom_fonction`
- ⚠ on n'utilise plus l'espace de nommage du module
- Importer tout le module :
- `from` **`nom_module`** `import *`

- ***Exemple :***

```
from numpy import *

z = full(4,2)
```

- **Conseil :** à utiliser avec beaucoup de précautions

Les packages : structuration de « gros » modules

- **Problème :** importation de très gros modules (`pyQt5`) :
  - Graphical User Interface (GUI)
  - QtMacExtras (classes pour MacOS et iOS)
  - QtWinExtras (classes pour Windows)
  - QtMultimedia (classes pour caméras, radios)
  - QtNfc (connectivité NFC)
  - Qt3DAnimation (animations 3D)
  - *etc.*

$\Longrightarrow$ côté utilisateur : on n'a pas envie de tout importer

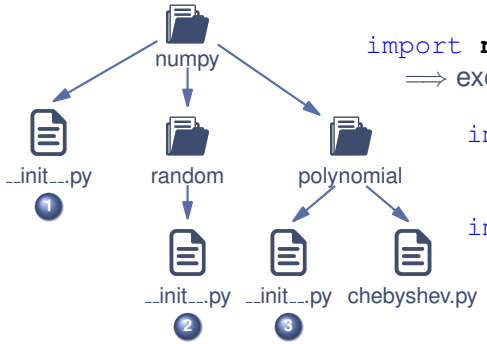
côté développeur : envie de structurer le module en plusieurs fichiers

- **Solution :** les packages

Les packages

Les packages

- Package ≈ ensemble de fichiers
- Fichiers stockés dans une **arborescence de fichiers** (répertoire, sous-répertoires)
- Répertoire contient fichier `__init__.py` ⟹ module
- Importations : `import` **`nom_répertoire.nom_module`**

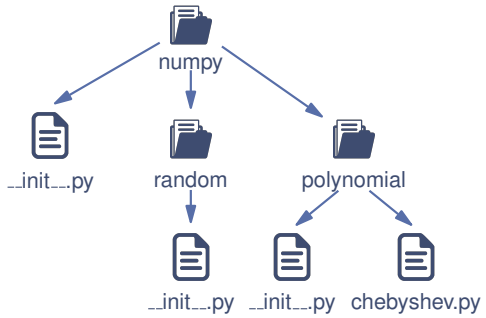


`import` **`numpy.polynomial.chebyshev`**
⟹ exécute `chebyshev.py`

`import` **`numpy`**
⟹ exécute `__init__.py` ❶

`import` **`numpy.random`**
⟹ exécute `__init__.py` ❷

Importer des « sous-modules »



- **Importer le sous-module polynomial :**
  1. `import` **`numpy.polynomial`**
     ⟹ fonction division : `numpy.polynomial.division`
  2. `from` **`numpy`** `import polynomial`
     ⟹ fonction division : `polynomial.division`

Contenu d'un module/package

- Après import, on peut lister le contenu d'un module/package :
  `dir(nom_module)`

```
dir(numpy)
```

```
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP',
'ComplexWarning', 'DataSource', 'ERR_CALL',
.......
'Inf', 'Infinity', 'MAXDIMS', 'MAY_SHARE_BOUNDS',
.......
'__all__', '__builtins__', '__cached__', '__config__',
'__doc__', '__file__', '__git_revision__', '__loader__',
'__name__', '__package__', '__path__', '__spec__',
'__version__', '_add_newdoc_ufunc', '_arg',
.......
'arange', 'arccos', 'arccosh', 'arcsin', 'arcsinh',
'arctan', 'arctan2', 'arctanh', 'argmax', 'argmin',
'argpartition', 'argsort', 'argwhere', 'around',
'array', 'array2string', 'array_equal',
.......
'vstack', 'where', 'who', 'zeros', 'zeros_like']
```


❷ Survol de la programmation objet

Qu'est-ce qu'un objet ?

Définition d'une classe

Une classe = type qui regroupe :

- des informations : **attributs**
- des fonctions de manipulation de ces infos : **méthodes**

$\Longrightarrow$ classe = type d'objets

Définition d'un objet

- Objet = 1 valeur de type « une classe »
- Objet = **instance** d'une classe

- Exemple :

```
a = "toto" ⟵ objet de la classe str
print(type(a))
print(a.upper()) ⟵ upper = méthode
```

```
<class 'str'>
TOTO
```


Création d'une classe et d'un objet

Création d'une classe :

mot clef

constructeur

mot clef : toujours 1er paramètre

```
class Ligne:
  def __init__(self, width, height):
    print('constructeur de Ligne')
    self.width = width
    self.height = height
```

attributs

- **Constructeur :** fonction appelée pour créer les instances
- **self :** référence à l'objet lui-même
- **Attribut :** `self.nom_attribut`
  $\Longrightarrow$ chaque instance possède son propre attribut

Création d'objets

Création d'un objet : `variable = nom_classe(paramètres)`

⚠ On ne spécifie pas le paramètre `self`

```
1  class Ligne:
2    def __init__(self, width, height):
3      print('constructeur de Ligne')
4      self.width = width
5      self.height = height
6
7  ligne1 = Ligne(4,6)
8  print(ligne1.width, ligne1.height)
9  ligne2 = Ligne(4,6)
10 ligne2.width = 10
11 print(ligne1.width, ligne2.width)
```

```
constructeur de Ligne
4 6
constructeur de Ligne
4 10
```

- Créations de 2 objets : lignes 7 et 9

Les méthodes

Méthode = fonction définie dans une classe

```
1  class Ligne:
2    def __init__(self, width, height):
3      print('constructeur de Ligne')
4      self.width = width
5      self.height = height
6
7    def double_taille(self): ⟵ méthode
8      self.width *= 2
9      self.height *= 2
10
11 ligne = Ligne(4,6)
12 print(ligne.width, ligne.height)
13 ligne.double_taille() ⟵ appel de la méthode
14 print(ligne.width, ligne.height)
```

```
constructeur de Ligne
4 6
8 12
```

- 1er paramètre des méthodes : `self`
- Appel de la méthode : `variable.nom_methode(paramètres)`

Intérêts des classes

- Les classes regroupent les informations au même endroit
- Comportement des objets bien défini : les méthodes

```
import numpy as np
x = np.full(6,0)
y = x.reshape((2,3))
print(y)
z = x.reshape((5,5))
```

```
[[0 0 0]
 [0 0 0]]
Traceback (most recent call last):
  File "/home/gonzales/tableau.py", line 5, in <module>
    z = x.reshape((5,5))
ValueError: cannot reshape array of size 6 into shape (5,5)
```

- **Sécurité :** l'objet contrôle les accès à son contenu
- Mais il y a bien plus : **héritage**, **surchage**, **polymorphisme**...

L'héritage de classe

Nouvelle classe *B* = extension d'une autre classe *A*

$\Longrightarrow$ *B* contient déjà les méthodes et attributs de *A*

$\Longrightarrow$ évite de les réécrire dans *B*

héritage : enfant de la classe Ligne

```
class LigneColoree(Ligne):
  def __init__(self, width, height, couleur):
    super().__init__(width, height)
    self.couleur = couleur
ligne = LigneColoree(4, 6, "vert")
ligne.double_taille()
print(ligne.width, ligne.height, ligne.couleur)
```

super = parent. Ici, on appelle son constructeur

on appelle la méthode définie dans Ligne

```
constructeur de Ligne
8 12 vert
```


Surcharge

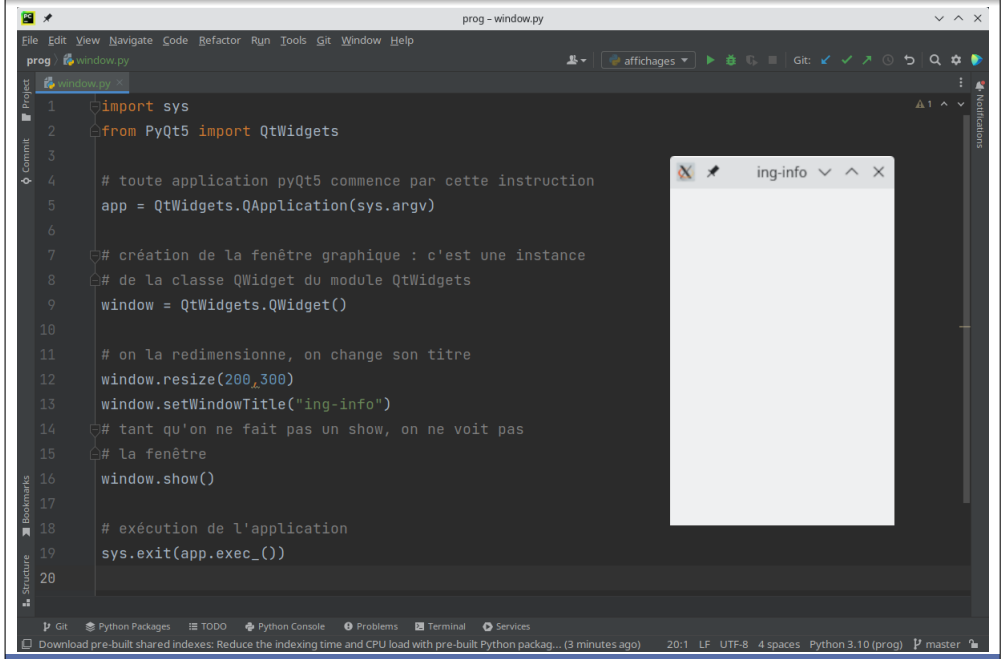
Surcharge = on réécrit une méthode « héritée »

$\Longrightarrow$ permet d'adapter le code à la nouvelle classe

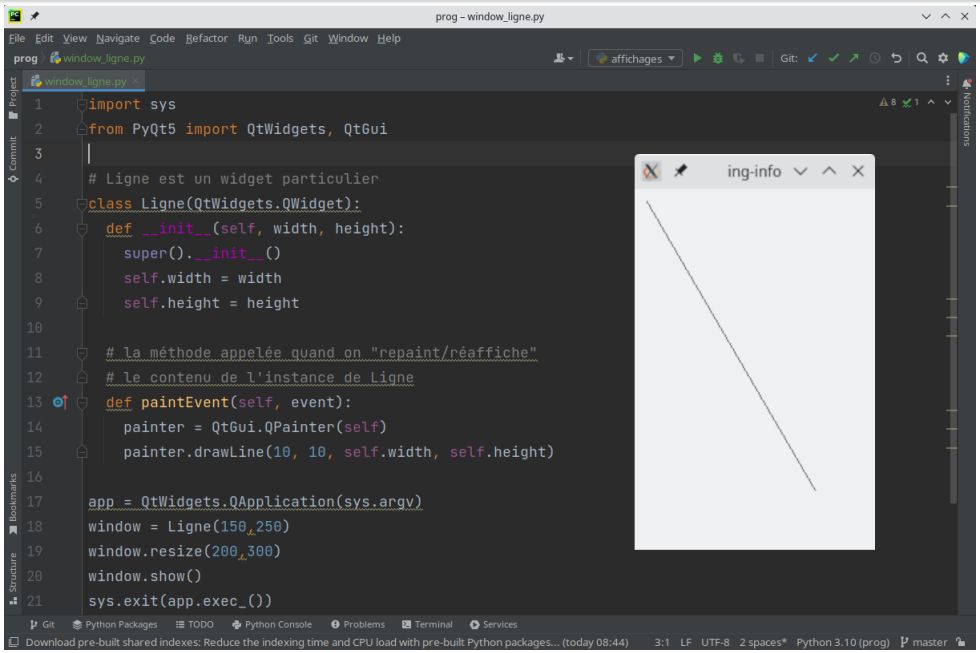
```
class LigneColoree(Ligne):
  def __init__(self, width, height, couleur):
    super().__init__(width, height)
    self.couleur = couleur

  def double_taille(self):
    print("seul width est modifié")
    self.width *= 2

ligne = LigneColoree(4, 6, "vert")
ligne.double_taille()
print(ligne.width, ligne.height, ligne.couleur)
```

```
constructeur de Ligne
seul width est modifié
8 6 vert
```

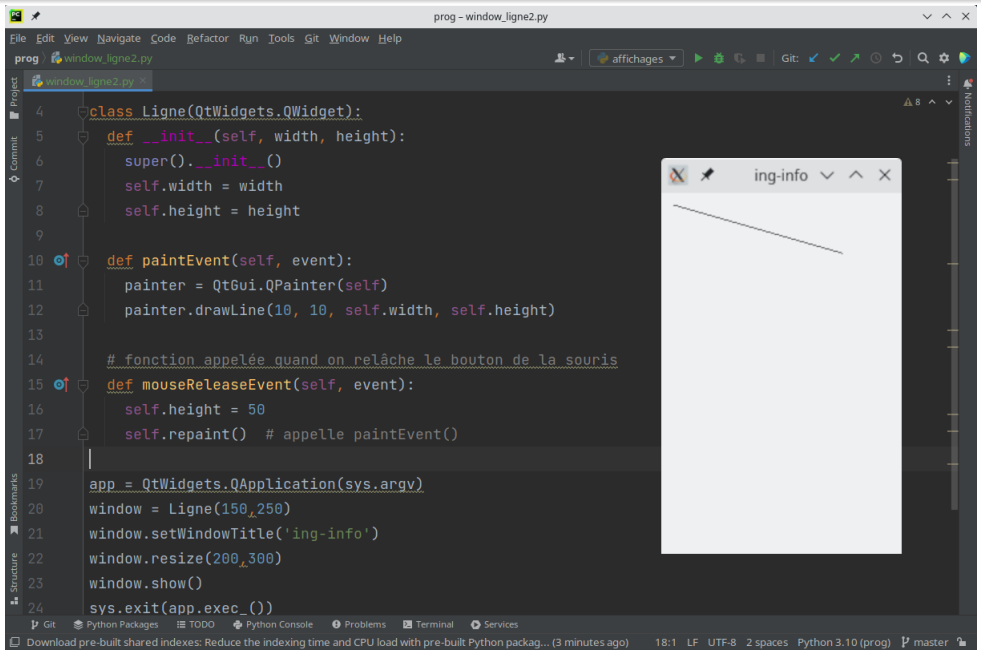

Application avec pyQt5 : création d'une fenêtre



Application avec pyQt5 : OK sur toute la ligne



Application avec pyQt5 : souris, tu es filmé



3. Quelques questions ouvertes

Exécution de fonction

```
1  def initialise (tab) :
2    tab = np.full((3,3), 0)
3
4  def remplit_inc (tab) :
5    initialise (tab)
6    x = 0
7    while x < 3:
8      tab[0,x] = x
9      x = x + 1
```

2	2	2
2	2	2
2	2	2

- ***Question :*** Contenu de `tab` après exécution de `remplit_inc` ?

Qui a raison ?

```
1  def fonction (tab_3x3) :
2    somme  = 0
3
4    y = 0
5    x = 0
6    while y < 3:
7      while x < 3:
8        somme = somme + tab_3x3[y,x]
9        x = x + 1
10     y = y + 1
11
12   return somme
```

11	13	16
1	0	3
2	1	1

- ***Question :*** somme = 40 ou 48 ?
- ***Question :*** Que faire pour obtenir 48 ?

Écriture de fonctions (1/2)

- Fonction `myfunc(x0, x1, y0, y1) :`
- renvoie : $\begin{cases} 0 \text{ si } x0 = x1 \text{ et } y01 > y11 \\ 1 \text{ si } x0 = x1 \text{ et } y01 < y11 \\ 2 \text{ si } x0 > x1 \text{ et } y01 = y11 \\ 3 \text{ si } x0 < x1 \text{ et } y01 = y11 \\ 4 \text{ sinon} \end{cases}$

- Fonction `myfunc2 (tab, nb_lignes, nb_colonnes) :`
- remplit le tableau `tab` de 0
- Si `x = np.full((2,2),5)`, après `myfunc2 (x, 2, 2)`, `x` ne contient que des 0

Écriture de fonctions (2/2)

- fonction `remplitDiag1 (tab, nb_lignes, nb_cols) :`

  Remplit la diagonale avec des 1

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

$\Longrightarrow$

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

- fonction `remplitDiag2 (tab, nb_lignes, nb_cols) :`

  Remplit l'autre diagonale avec des 1

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

$\Longrightarrow$

0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0