

# Cours 4 : Packages et objets



Ing-info — Ingénierie informatique

Christophe Gonzales

- 1 Les packages
- 2 Survol des objets
- 3 Quelques questions

## ► **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  $\approx$  nom du fichier (sans le `.py`)
- Utiliser le fichier = l'importer
- Module importé  $\implies$  on peut utiliser tout ce qu'il définit

# Et si le monde n'était pas réel ?

- ▶ Python connaît les nombres complexes :

$$z = \text{complex}(4, 2) \implies z = 4 + 2i$$

- ▶ 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` ✓

$\implies$  « 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  $\implies$  espace de nommage

## *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))
```

## *Importation et alias*

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

### ▶ *Exemple :*

```
import numpy as np  
z = np.full(4, 2)
```

## 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)
```

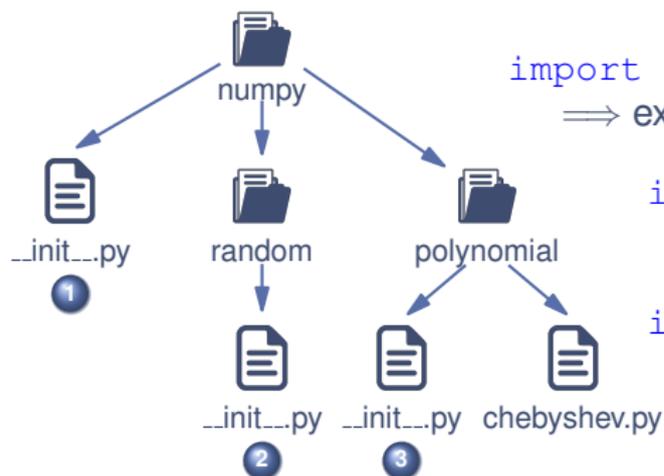
- ▶ **Conseil :** à utiliser avec beaucoup de précautions

- ▶ **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.*
- ⇒ 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

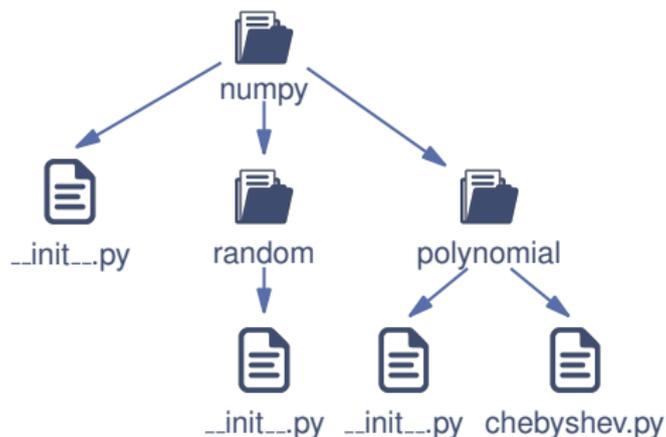
- ▶ Package  $\approx$  ensemble de fichiers
- ▶ Fichiers stockés dans une **arborescence de fichiers** (répertoire, sous-répertoires)
- ▶ Répertoire contient fichier `__init__.py`  $\implies$  module
- ▶ Importations : `import nom_répertoire.nom_module`



```
import numpy.polynomial.chebyshev  
 $\implies$  exécute chebyshev.py
```

```
import numpy  
 $\implies$  exécute __init__.py 1
```

```
import numpy.random  
 $\implies$  exécute __init__.py 2
```



## ► 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']
```

## ② 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**

⇒ 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



class **Ligne**:

constructeur

mot clef : toujours 1er paramètre

```
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`  
⇒ chaque instance possède son propre attribut

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

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)`

- ▶ 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  $A$

⇒  $B$  contient déjà les méthodes et attributs de  $A$

⇒ 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
        super = parent. Ici, on appelle son constructeur
ligne = LigneColoree(4, 6, "vert")
ligne.double_taille() ← on appelle la méthode définie dans Ligne
print(ligne.width, ligne.height, ligne.couleur)
```

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

Surcharge = on réécrit une méthode « héritée »

⇒ permet d'adapter le code à la nouvelle classe

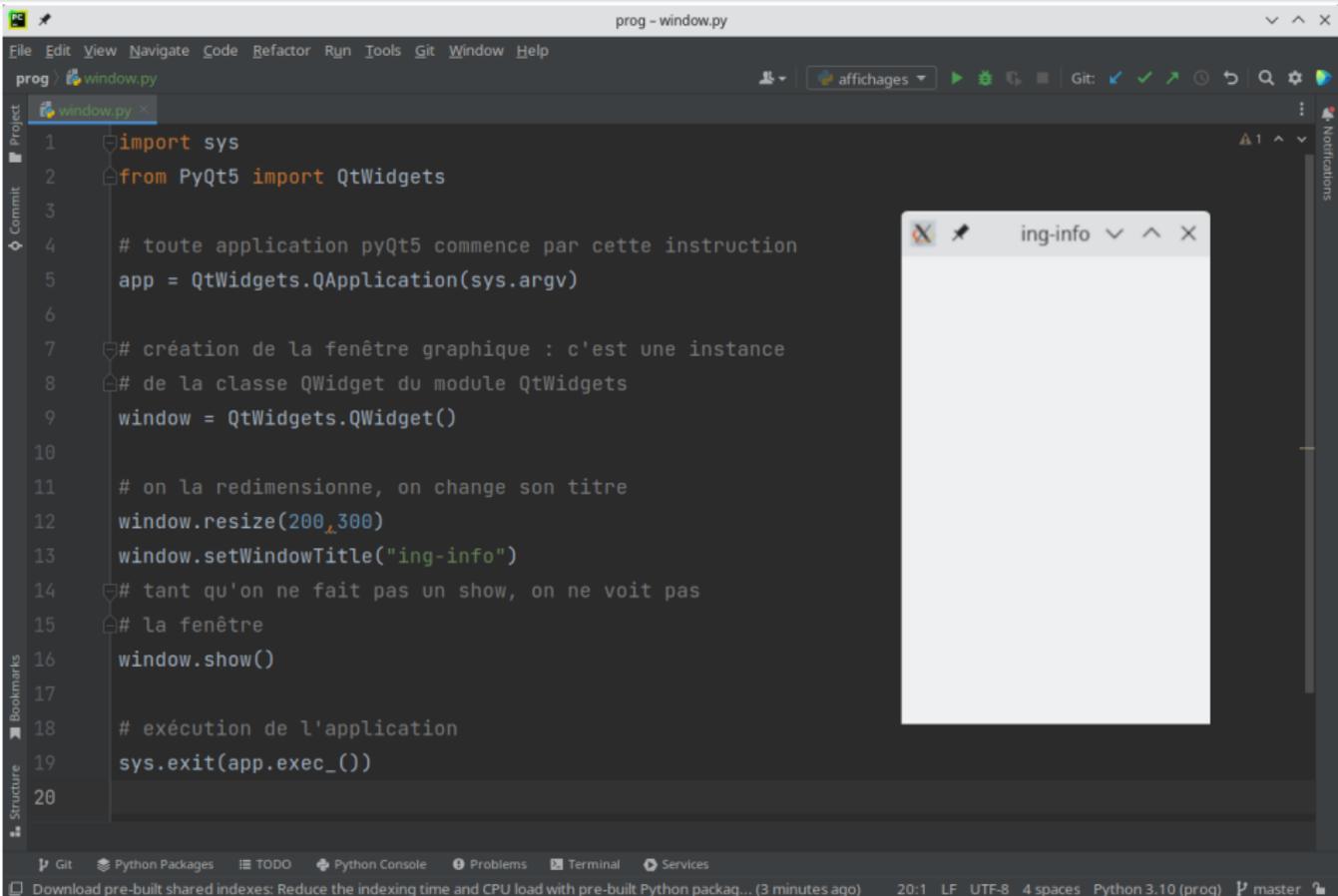
```
class LigneColoree(Ligne):
    def __init__(self, width, height, couleur):
        super().__init__(width, height)
        self.couleur = couleur

    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

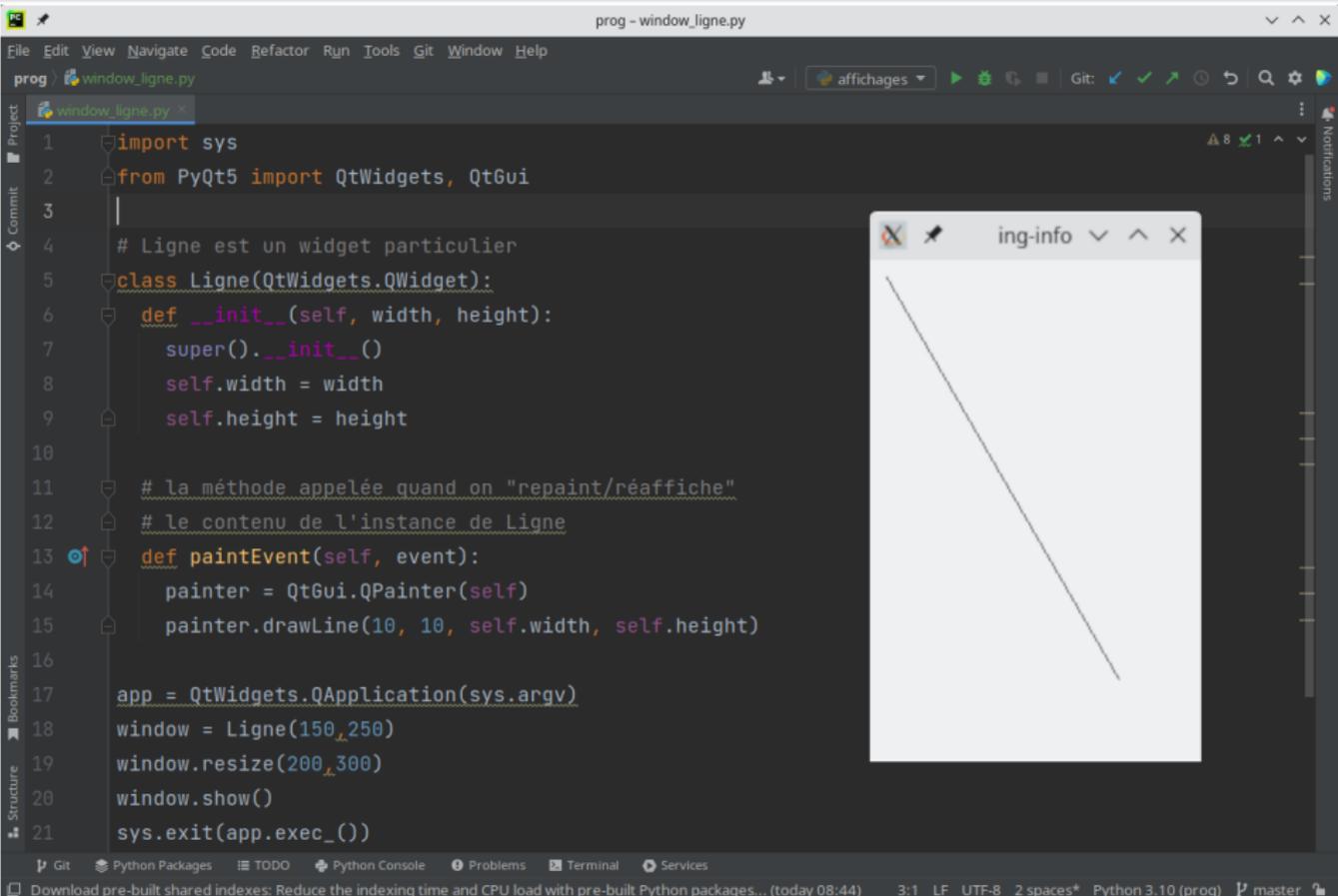


The image shows a code editor window titled "prog - window.py" with the following Python code:

```
1 import sys
2 from PyQt5 import QtWidgets
3
4 # toute application pyQt5 commence par cette instruction
5 app = QtWidgets.QApplication(sys.argv)
6
7 # création de la fenêtre graphique : c'est une instance
8 # de la classe QWidget du module QtWidgets
9 window = QtWidgets.QWidget()
10
11 # on la redimensionne, on change son titre
12 window.resize(200,300)
13 window.setWindowTitle("ing-info")
14 # tant qu'on ne fait pas un show, on ne voit pas
15 # la fenêtre
16 window.show()
17
18 # exécution de l'application
19 sys.exit(app.exec_())
20
```

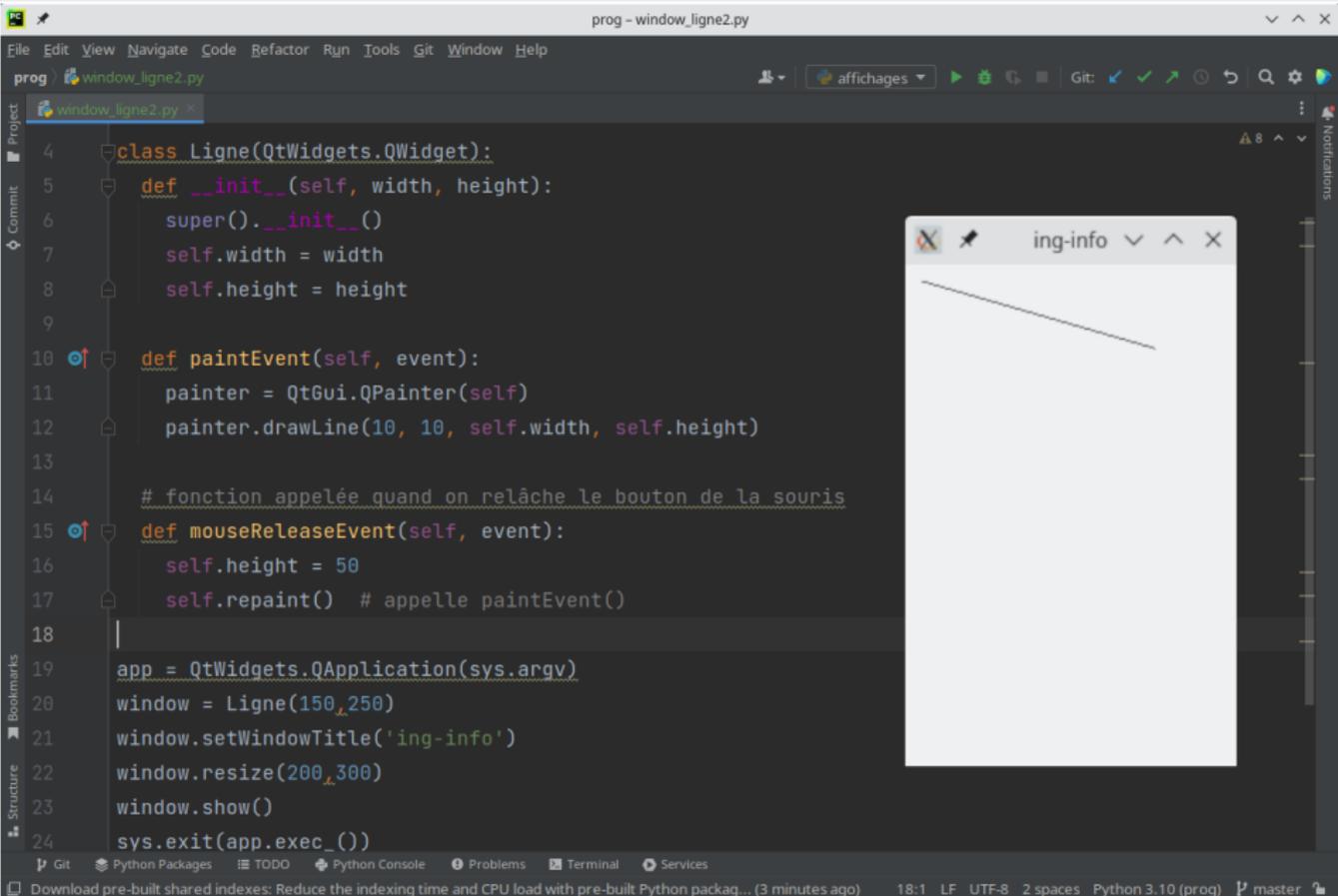
Overlaid on the right side of the editor is a small window titled "ing-info" with a white background and standard window controls (close, maximize, minimize).

# Application avec PyQt5 : OK sur toute la ligne



```
1 import sys
2 from PyQt5 import QtWidgets, QtGui
3
4 # Ligne est un widget particulier
5 class Ligne(QtWidgets.QWidget):
6     def __init__(self, width, height):
7         super().__init__()
8         self.width = width
9         self.height = height
10
11 # la méthode appelée quand on "repaint/réaffiche"
12 # le contenu de l'instance de Ligne
13 def paintEvent(self, event):
14     painter = QtGui.QPainter(self)
15     painter.drawLine(10, 10, self.width, self.height)
16
17 app = QtWidgets.QApplication(sys.argv)
18 window = Ligne(150, 250)
19 window.resize(200, 300)
20 window.show()
21 sys.exit(app.exec_())
```

# Application avec PyQt5 : souris, tu es filmé



The image shows a code editor window titled "prog - window\_ligne2.py". The code defines a class `Ligne` that inherits from `QtWidgets.QWidget`. The class has an `__init__` method that sets the width and height, and a `paintEvent` method that draws a line. A `mouseReleaseEvent` method is also defined, which sets the height to 50 and calls `repaint()`. The main code block creates a `QApplication`, instantiates the `Ligne` class, sets the window title to "ing-info", resizes the window, and shows it.

```
4 class Ligne(QtWidgets.QWidget):
5     def __init__(self, width, height):
6         super().__init__()
7         self.width = width
8         self.height = height
9
10    def paintEvent(self, event):
11        painter = QtGui.QPainter(self)
12        painter.drawLine(10, 10, self.width, self.height)
13
14    # fonction appelée quand on relâche le bouton de la souris
15    def mouseReleaseEvent(self, event):
16        self.height = 50
17        self.repaint() # appelle paintEvent()
18
19    app = QtWidgets.QApplication(sys.argv)
20    window = Ligne(150, 250)
21    window.setWindowTitle('ing-info')
22    window.resize(200, 300)
23    window.show()
24    sys.exit(app.exec_())
```

The preview window titled "ing-info" displays a white background with a single black line drawn from the top-left corner to the bottom-right corner, representing the state of the application after the mouse button is released.

### ③ Quelques questions ouvertes

# Exécution de fonction

```
1 def initialise (tab) :  
2     tab = np.full((3,3), 0)  
3  
4 def remplir_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 remplir\_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 ?

► Fonction `myfunc(x0, x1, y0, y1)` :

► renvoie : 
$$\begin{cases} 0 \text{ si } x_0 = x_1 \text{ et } y_{01} > y_{11} \\ 1 \text{ si } x_0 = x_1 \text{ et } y_{01} < y_{11} \\ 2 \text{ si } x_0 > x_1 \text{ et } y_{01} = y_{11} \\ 3 \text{ si } x_0 < x_1 \text{ et } y_{01} = y_{11} \\ 4 \text{ sinon} \end{cases}$$

---

► 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

 $\Rightarrow$ 

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

 $\Rightarrow$ 

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