

PyQt4

Definición.-

PyQt es una biblioteca gráfica Qt para el lenguaje de programación Python. La biblioteca está desarrollada por la firma británica Riverbank Computing y está disponible para Windows, GNU/Linux y Mac OS X bajo diferentes licencias.

En agosto de 2009, tras intentar negociar con Riverbank Computing la liberación de PyQt bajo licencia LGPL sin conseguirlo, Nokia, propietaria de Qt, libera bajo esta licencia un binding similar, llamado PySide.

PyQt4 es un conjunto de herramientas para crear aplicaciones GUI. Es una combinación de lenguaje de programación Python y la exitosa librería Qt. La librería Qt es una de las bibliotecas GUI más potentes.

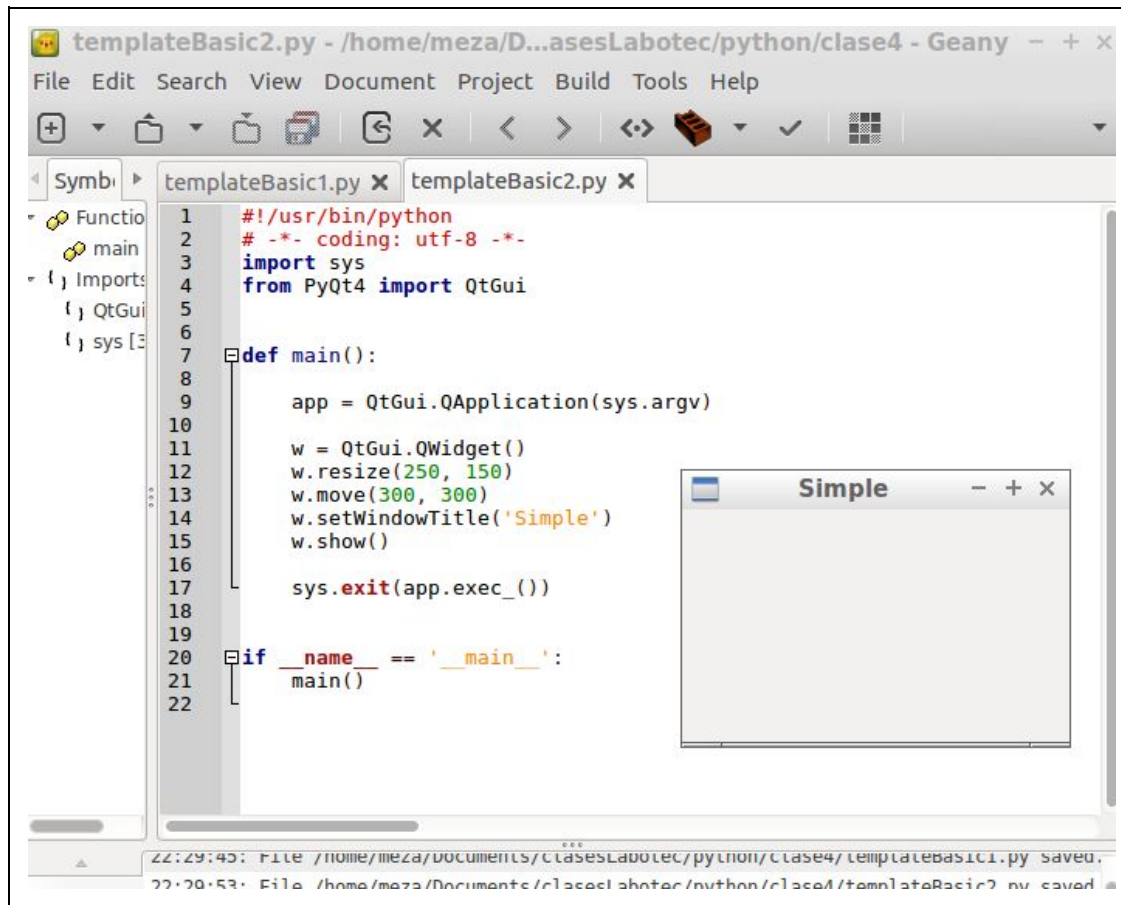
PyQt4 se implementa como un conjunto de módulos Python. Tiene 440 clases y 6000 funciones y métodos. A partir de PyQt versión 4, la licencia GPL está disponible en todas las plataformas soportadas.

Las clases de PyQt4 se dividen en varios módulos:

- QtCore
- QtGui
- QtNetwork
- QtXml
- QtSvg
- QtOpenGL
- QSql

Creación de la Ventana:

Este es un ejemplo sencillo que muestra una pequeña ventana. Sin embargo, podemos hacer mucho con esta ventana. Podemos cambiar su tamaño, maximizarlo o minimizarlo. Esto requiere mucha codificación. Alguien ya ha codificado esta funcionalidad. Debido a que se repite en la mayoría de las aplicaciones, no hay necesidad de código de nuevo. PyQt4 es un conjunto de herramientas de alto nivel. Si codificamos en un conjunto de herramientas de nivel inferior, el siguiente ejemplo de código podría tener fácilmente cientos de líneas.



El código anterior muestra una pequeña ventana en la pantalla. Aquí proporcionamos las importaciones necesarias. Los widgets GUI básicos se encuentran en el módulo QtGui.

```
import sys
from PyQt4 import QtGui
```

Cada aplicación PyQt4 debe crear un objeto de aplicación. El objeto de aplicación se encuentra en el módulo QtGui. El parámetro sys.argv es una lista de argumentos de la línea de comandos. Los scripts de Python se pueden ejecutar desde el shell. Es una forma de cómo podemos controlar el inicio de nuestros scripts.

```
app = QtGui.QApplication(sys.argv)
```

El widget QtGui.QWidget es la clase base de todos los objetos de la interfaz de usuario en PyQt4. Proporcionamos el constructor predeterminado para QtGui.QWidget. El constructor predeterminado no tiene ningún padre. Un widget sin padre se llama ventana.

```
w = QtGui.QWidget()
```

El método `resize ()` redimensiona el widget. Tiene 250px de ancho y 150px de alto.

```
w.resize(250, 150)
```

El método `move ()` mueve el widget a una posición en la pantalla en `x = 300` y `y = 300` coordenadas.

```
w.move(300, 300)
```

Aquí ponemos el título de nuestra ventana. El título se muestra en la barra de título.

```
w.setWindowTitle('Simple')
```

El método `show ()` muestra el widget en la pantalla. Un widget se crea primero en la memoria y más tarde se muestra en la pantalla.

```
w.show()
```

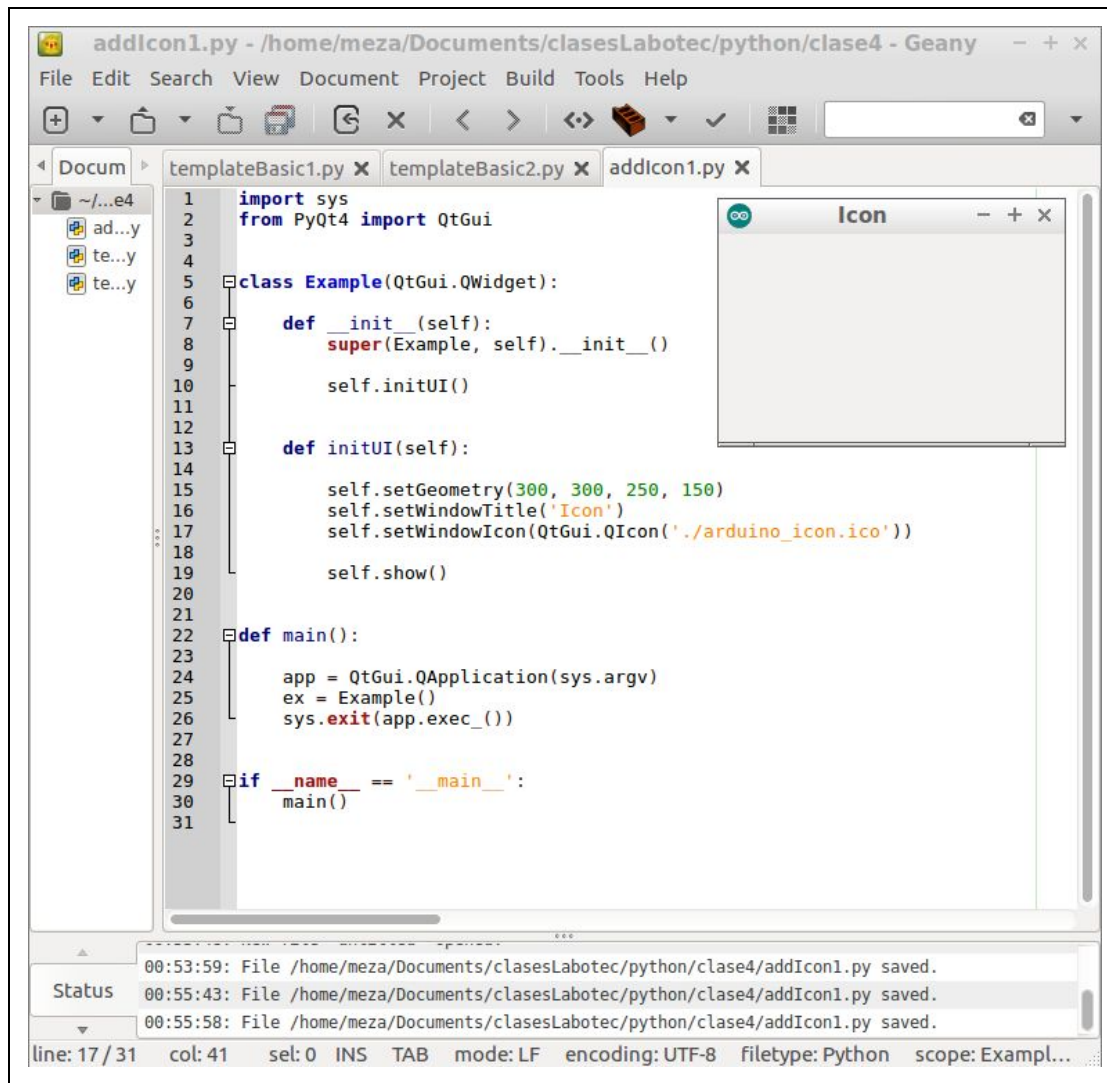
Finalmente, entramos en el `mainloop` de la aplicación. El manejo del evento comienza desde este punto. El `mainloop` recibe eventos del sistema de ventanas y los despacha a los widgets de aplicación. El `mainloop` termina si llamamos al método `exit ()` o el widget principal es destruido. El método `sys.exit ()` asegura una salida limpia. El medio ambiente será informado de cómo terminó la aplicación.

El método `exec_ ()` tiene un subrayado. Es porque el `exec` es una palabra clave de Python. Y así, `exec_ ()` se utilizó en su lugar.

```
sys.exit(app.exec_())
```

Agregando un icono a la aplicación.-

El icono de la aplicación es una imagen pequeña que normalmente se muestra en la esquina superior izquierda de la barra de título. En el siguiente ejemplo mostraremos cómo lo hacemos en PyQt4. También introduciremos algunos nuevos métodos.



```
1 import sys
2 from PyQt4 import QtGui
3
4
5 class Example(QtGui.QWidget):
6
7     def __init__(self):
8         super(Example, self).__init__()
9
10        self.initUI()
11
12
13    def initUI(self):
14
15        self.setGeometry(300, 300, 250, 150)
16        self.setWindowTitle('Icon')
17        self.setWindowIcon(QtGui.QIcon('./arduino_icon.ico'))
18
19        self.show()
20
21
22 def main():
23
24     app = QtGui.QApplication(sys.argv)
25     ex = Example()
26     sys.exit(app.exec_())
27
28
29 if __name__ == '__main__':
30     main()
31
```

00:53:59: File /home/meza/Documents/clasesLabotec/python/clase4/addIcon1.py saved.
Status 00:55:43: File /home/meza/Documents/clasesLabotec/python/clase4/addIcon1.py saved.
00:55:58: File /home/meza/Documents/clasesLabotec/python/clase4/addIcon1.py saved.
line: 17 / 31 col: 41 sel: 0 INS TAB mode: LF encoding: UTF-8 filetype: Python scope: Examl...

El ejemplo anterior se codificó en un estilo de procedimiento. El lenguaje de programación de Python admite estilos de programación orientados a procedimientos y objetos. La programación en PyQt4 significa programar en POO.

Las tres cosas más importantes en la programación orientada a objetos son clases, datos y métodos. Aquí creamos una nueva clase llamada Ejemplo. La clase de ejemplo hereda de la clase QtGui.QWidget. Esto significa que llamamos a dos constructores: el primero para la clase de ejemplo y el segundo para la clase heredada. El método super () devuelve el objeto primario de la clase Example y llamamos a su constructor. El método __init__ () es un método constructor en Python.

```
class Example(QtGui.QWidget):  
  
    def __init__(self):  
        super(Example, self).__init__()  
        ...
```

La creación de la GUI se delega en el método initUI().

```
self.initUI()
```

Los tres métodos han sido heredados de la clase QtGui.QWidget. SetGeometry () hace dos cosas. Localiza la ventana en la pantalla y establece su tamaño. Los dos primeros parámetros son las posiciones x e y de la ventana. El tercero es el ancho y el cuarto es la altura de la ventana. De hecho, combina los métodos resize() y move() en un método. El último método establece el icono de la aplicación. Para ello, hemos creado un objeto QtGui.QIcon. QtGui.QIcon recibe la ruta de acceso a nuestro icono para que se muestre.

```
self.setGeometry(300, 300, 250, 150)  
self.setWindowTitle('Icon')  
self.setWindowIcon(QtGui.QIcon('./arduino_icon.ico'))
```

El código de inicio se ha colocado en el método main ().

```
def main():  
  
    app = QtGui.QApplication(sys.argv)  
    ex = Example()  
    sys.exit(app.exec_())  
  
if __name__ == '__main__':  
    main()
```

Cerrando una ventana.-

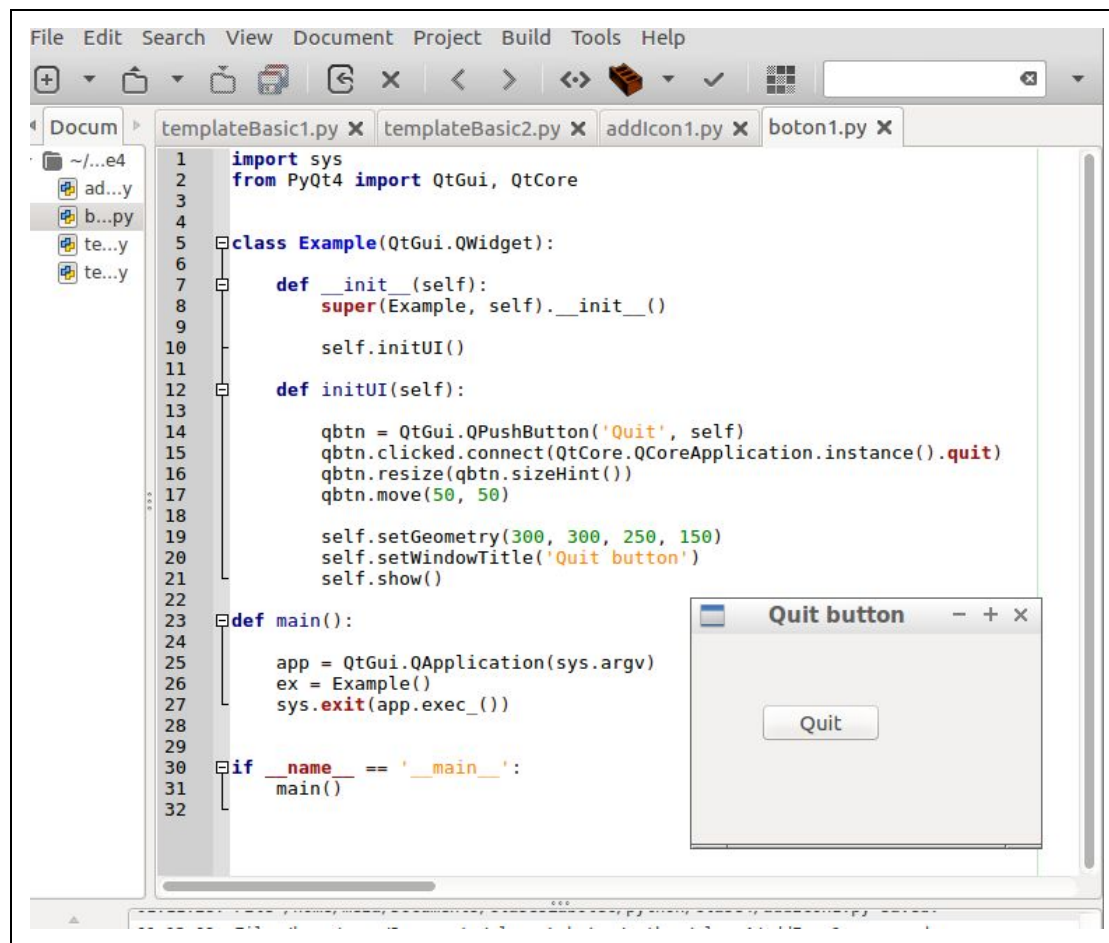
Podemos proporcionar una ayuda para cualquiera de nuestros widgets.

La manera obvia de cómo cerrar una ventana es hacer clic en la marca x en la barra de título. En el siguiente ejemplo, mostraremos cómo podemos cerrar temporalmente nuestra ventana. Vamos a tocar brevemente las señales y las ranuras.

Lo siguiente es el constructor de un `QtGui.QPushButton` que usaremos en nuestro ejemplo.

`QPushButton (texto de cadena, QWidget padre = Ninguno)`

El parámetro `text` es un texto que se mostrará en el botón. El padre es un widget en el que colocamos nuestro botón. En nuestro caso será un `QtGui.QWidget`.



En este ejemplo, creamos un botón de salir. Al hacer clic en el botón, la aplicación termina.

```
from PyQt4 import QtGui, QtCore
```

Se necesitará un objeto del módulo QtCore. Por lo tanto, importamos el módulo.

```
Qbtn = QtGui.QPushButton ('Quit', self)
```

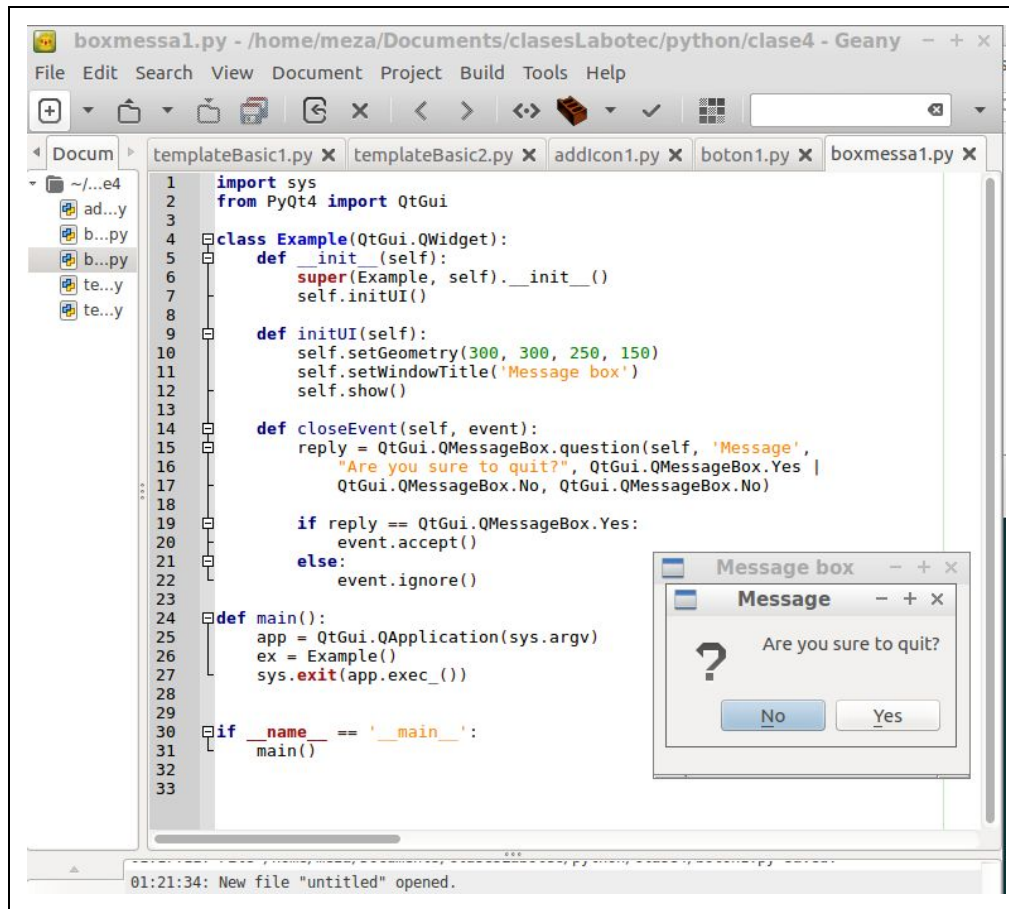
Creamos un botón. El botón es una instancia de la clase QtGui.QPushButton. El primer parámetro del constructor es la etiqueta del botón. El segundo parámetro es el widget principal. El widget principal es el widget de ejemplo, que es un QtGui.QWidget por herencia.

```
Qbtn.clicked.connect (QtCore.QCoreApplication.instance (). Quit)
```

El sistema de procesamiento de eventos en PyQt4 se construye con el mecanismo de señal y ranura. Si hacemos clic en el botón, se emite la señal. La ranura puede ser una ranura Qt o cualquier Python callable. El QtCore.QCoreApplication contiene el bucle de eventos principal. Procesa y distribuye todos los eventos. El método instance () nos da su instancia actual. Tenga en cuenta que QtCore.QCoreApplication se crea con QtGui.QApplication. La señal clicada se conecta al método quit () que termina la aplicación. La comunicación se realiza entre dos objetos: el remitente y el receptor. El remitente es el pulsador, el receptor es el objeto de la aplicación.

Buzon de mensaje.-

De forma predeterminada, si hacemos clic en el botón x de la barra de título, el QtGui.QWidget se cierra. A veces queremos modificar este comportamiento predeterminado. Por ejemplo, si tenemos un archivo abierto en un editor al que hicimos algunos cambios. Se muestra un cuadro de mensaje para confirmar la acción.



Si cerramos QtGui.QWidget, se genera QtGui.QCloseEvent. Para modificar el comportamiento del widget necesitamos volver a implementar el controlador de eventos closeEvent ().

```
Reply = QtGui.QMessageBox.question (self, 'Mensaje',  
    "¿Está seguro de salir?", QtGui.QMessageBox.Yes |  
    QtGui.QMessageBox.No, QtGui.QMessageBox.No)
```

Se muestra un cuadro de mensaje con dos botones: Sí y No. La primera cadena aparece en la barra de título. La segunda cadena es el texto del

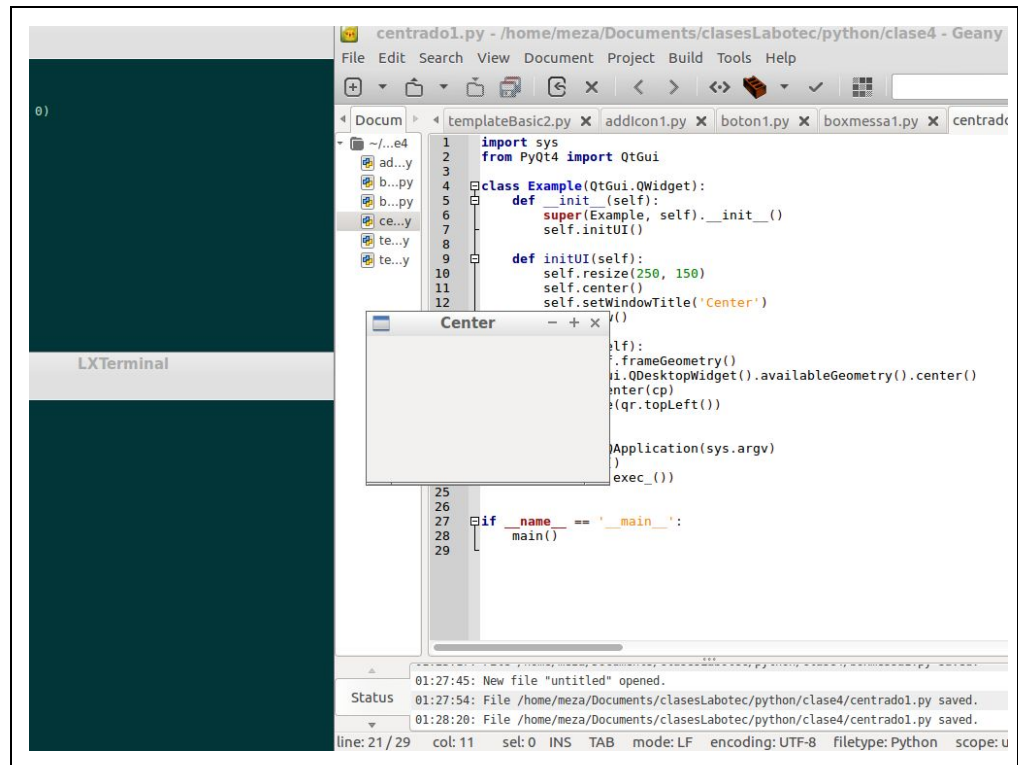
mensaje que se muestra en el cuadro de diálogo. El tercer argumento especifica la combinación de botones que aparecen en el diálogo. El último parámetro es el botón predeterminado. Es el botón que inicialmente tiene el enfoque del teclado. El valor de retorno se almacena en la variable de respuesta.

```
if reply == QtGui.QMessageBox.Yes:
    event.accept ()
else:
    event.ignore ()
```

Aquí probamos el valor de retorno. Si hacemos clic en el botón Sí, aceptamos el evento que conduce al cierre del widget ya la terminación de la aplicación. De lo contrario, ignoraremos el evento cercano.

Ventana de centrado en la pantalla.-

El siguiente script muestra cómo podemos centrar una ventana en la pantalla del escritorio.



La clase QtGui.QDesktopWidget proporciona información sobre el escritorio del usuario, incluido el tamaño de la pantalla.

```
self.center()
```

El código que centrará la ventana se colocará en el método personalizado center ().

```
qr = self.frameGeometry ()
```

Obtenemos un rectángulo especificando la geometría de la ventana principal. Esto incluye cualquier marco de ventana.

```
cp = QtGui.QDesktopWidget (). AvailableGeometry ().Center ()
```

Calculamos la resolución de pantalla de nuestro monitor. Y de esta resolución, obtenemos el punto central.

```
qr.moveCenter (cp)
```

Nuestro rectángulo ya tiene su anchura y altura. Ahora ajustamos el centro del rectángulo al centro de la pantalla. El tamaño del rectángulo no cambia.

```
self.move (qr.topLeft ())
```

Movemos el punto superior izquierdo de la ventana de la aplicación hacia el punto superior izquierdo del rectángulo qr, centrando así la ventana en nuestra pantalla.

[Layout en PyQt4.-](#)

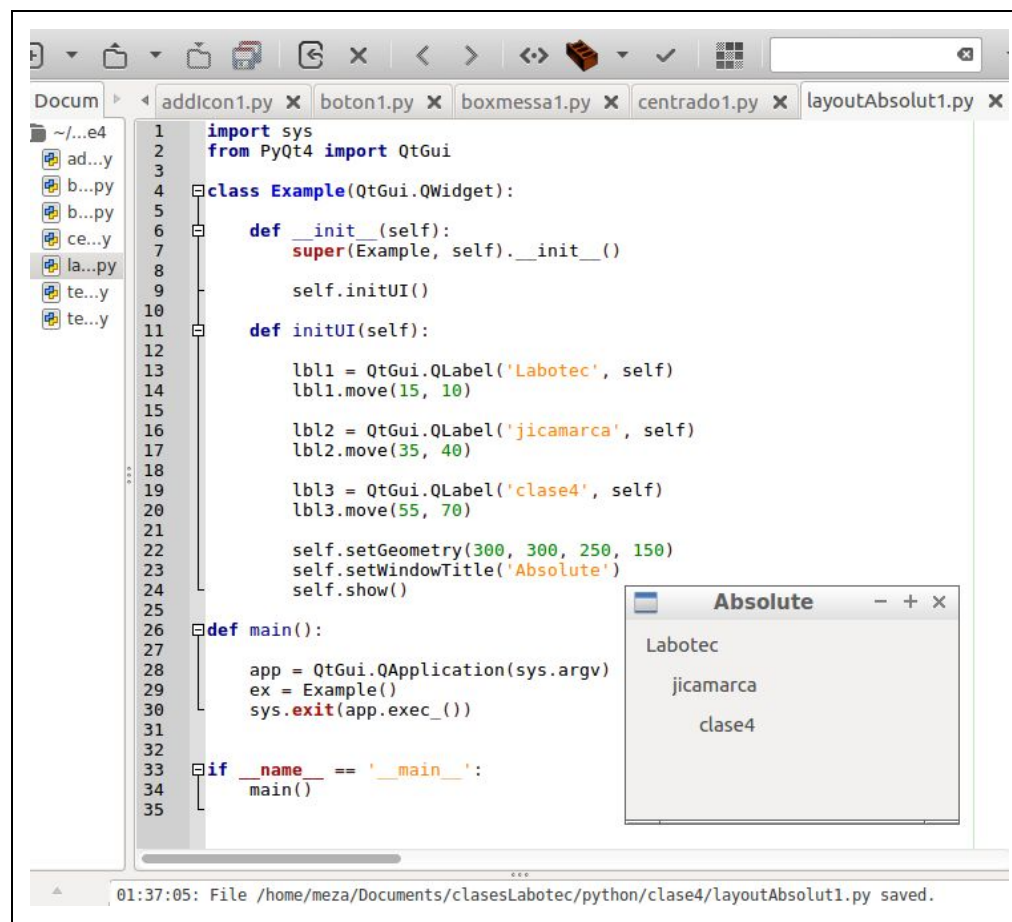
Un aspecto importante en la programación GUI es la gestión de la disposición. La gestión de diseño es la forma en que colocamos los widgets en la ventana. La gestión se puede hacer de dos maneras básicas. Podemos utilizar el posicionamiento absoluto o clases de diseño.

Posicionamiento absoluto

El programador especifica la posición y el tamaño de cada widget en píxeles. Cuando se utiliza el posicionamiento absoluto, tenemos que entender las siguientes limitaciones:

- El tamaño y la posición de un widget no cambian si cambiamos el tamaño de una ventana
- Las aplicaciones pueden verse diferentes en varias plataformas
- Cambiar fuentes en nuestra aplicación podría estropear el diseño
- Si decidimos cambiar nuestro diseño, debemos rehacer completamente nuestro diseño, que es tedioso y requiere mucho tiempo

El siguiente ejemplo colocará widgets en coordenadas absolutas.



Utilizamos el método `move()` para posicionar nuestros widgets. En nuestro caso son etiquetas. Las posicionamos proporcionando las coordenadas `key`. El comienzo del sistema de coordenadas se encuentra en la esquina superior izquierda. Los valores `x` crecen de izquierda a derecha. Los valores `y` crecen de arriba a abajo.

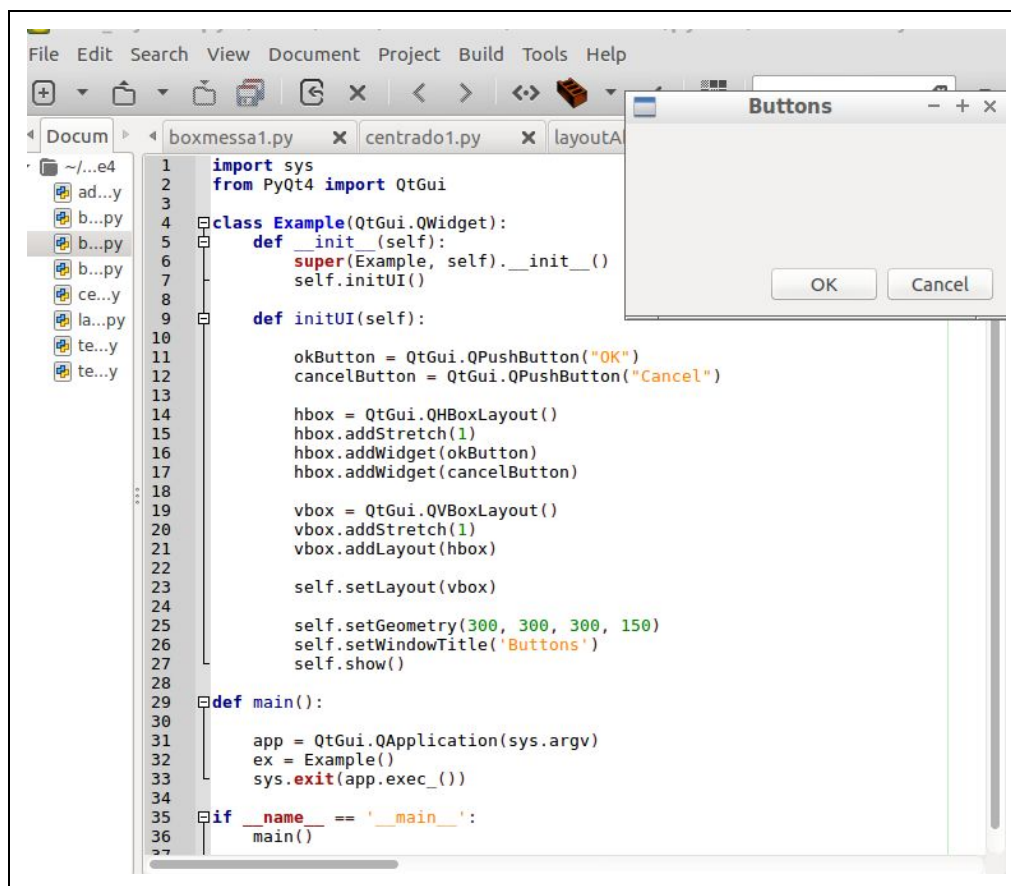
```
lbl1 = QtGui.QLabel ('Zetcode', self)
lbl1.move (15, 10)
```

El widget de etiqueta se posiciona en x = 15 e y = 10.

Box Layout.-

La gestión de diseño con clases de diseño es mucho más flexible y práctica. Es la forma preferida de colocar widgets en una ventana. QtGui.QHBoxLayout y QtGui.QVBoxLayout son clases de diseño básicas que alinean los widgets horizontal y verticalmente.

Imagine que queríamos colocar dos botones en la esquina inferior derecha. Para crear dicho diseño, usaremos una caja horizontal y una vertical. Para crear el espacio necesario, agregaremos un factor de estiramiento.



El ejemplo coloca dos botones en la esquina inferior derecha de la ventana. Se quedan allí cuando cambiamos el tamaño de la ventana de la aplicación. Utilizamos un QtGui.HBoxLayout y un QtGui.QVBoxLayout.

```
okButton = QtGui.QPushButton ("Aceptar")  
cancelButton = QtGui.QPushButton ("Cancelar")
```

Aquí creamos dos pulsadores.

```
hbox = QtGui.QHBoxLayout ()  
hbox.addStretch (1)  
hbox.addWidget (okButton)  
hbox.addWidget (cancelButton)
```

Creamos un diseño de caja horizontal y agregamos un factor de estiramiento y ambos botones. El estiramiento añade un espacio estirable antes de los dos botones. Esto los empujará a la derecha de la ventana.

```
vbox = QtGui.QVBoxLayout ()  
vbox.addStretch (1)  
vbox.addLayout (hbox)
```

Para crear el diseño necesario, ponemos un diseño horizontal en uno vertical. El factor de estiramiento en la caja vertical empujará la caja horizontal con los botones a la parte inferior de la ventana.

```
self.setLayout (vbox)
```

Finalmente, establecemos el diseño principal de la ventana.

Eventos y Señales en PyQt4

En esta parte del tutorial de programación PyQt4, exploraremos eventos y señales que ocurren en aplicaciones.

Eventos

Todas las aplicaciones GUI están orientadas a eventos. Los eventos son generados principalmente por el usuario de una aplicación. Pero también pueden generarse por otros medios: p. Una conexión a Internet, un administrador de ventanas o un temporizador. Cuando llamamos al método `exec_()` de la aplicación, la aplicación entra en el bucle principal. El bucle principal recupera los eventos y los envía a los objetos.

En el modelo de eventos, hay tres participantes:

- Fuente del evento
- Objeto de evento
- Objetivo del evento

El origen de eventos es el objeto cuyo estado cambia. Genera eventos. El objeto de evento (evento) encapsula los cambios de estado en el origen de eventos. El objetivo del evento es el objeto que desea ser notificado. El objeto de origen de eventos delega la tarea de gestionar un evento al destino de evento.

PyQt4 tiene una señal única y mecanismo de ranura para hacer frente a los eventos. Las señales y las ranuras se utilizan para la comunicación entre objetos. Se emite una señal cuando se produce un evento en particular. Una ranura puede ser cualquier Python callable. Se llama una ranura cuando se emite una señal conectada a ella.

Nueva API

PyQt4.5 introdujo una nueva API de estilo para trabajar con señales y ranuras.

```
QtCore.QObject.connect(botón,QtCore.SIGNAL('clicked()'),self.onClicked)
```

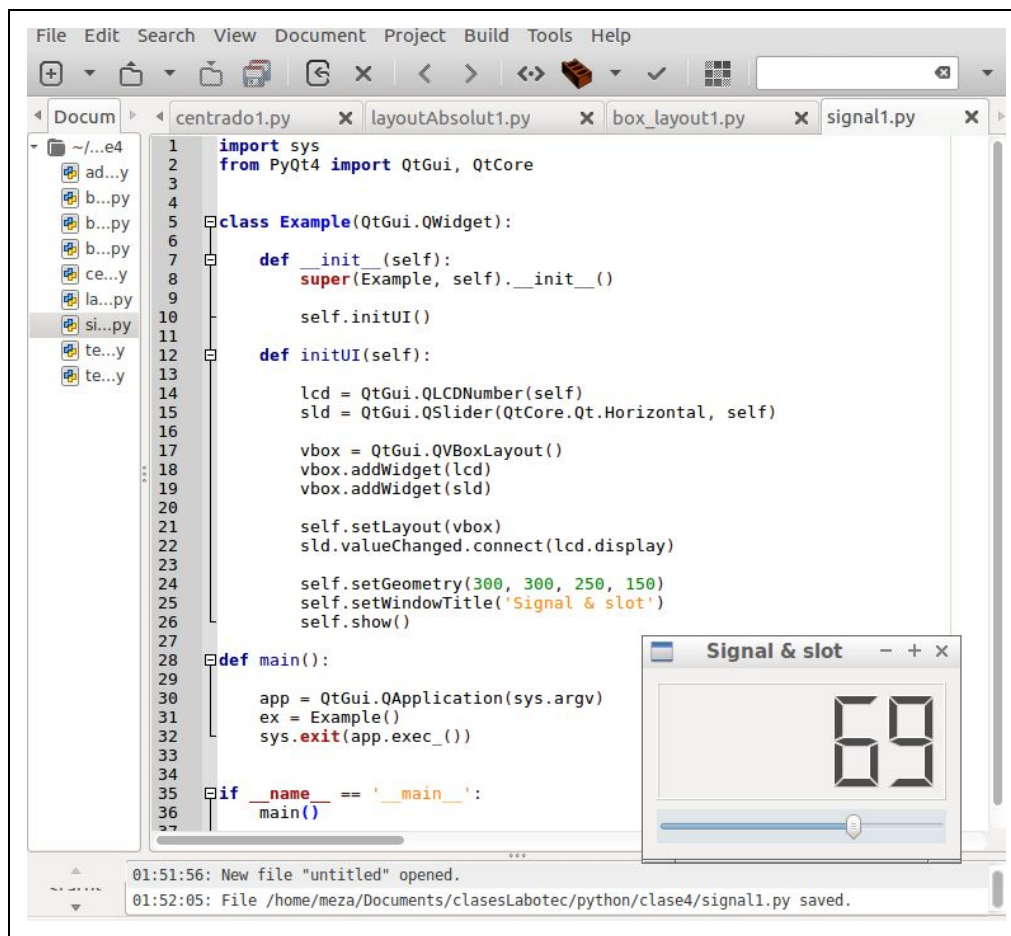
Esta es la API de estilo antiguo.

```
Button.clicked.connect (self.onClicked)
```

El nuevo estilo se adhiere más a los estándares de Python.

Signals & Slots

Este es un ejemplo sencillo que demuestra señales y ranuras en PyQt4.



En nuestro ejemplo, mostramos un QtGui.QLCDNumber y un QtGui.QSlider. Cambiamos el número de lcd arrastrando el mando deslizante.

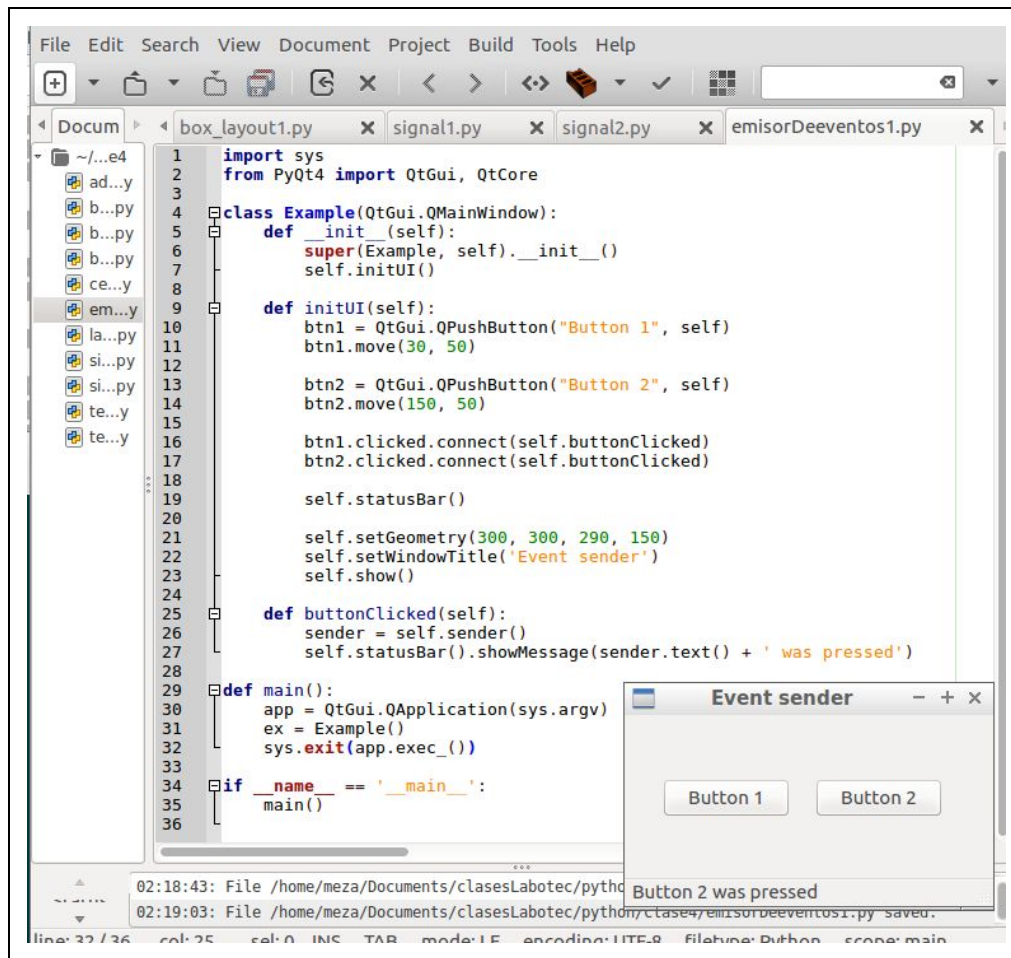
```
sld.valueChanged.connect (lcd.display)
```

Aquí se conecta una señal valueChanged del control deslizante a la ranura de visualización del número lcd.

El remitente es un objeto que envía una señal. El receptor es el objeto que recibe la señal. La ranura es el método que reacciona a la señal.

Transmisor de eventos

A veces es conveniente saber qué widget es el remitente de una señal. Para ello, PyQt4 tiene el método `sender()`.



Tenemos dos botones en nuestro ejemplo. En el método `buttonClicked ()` determinamos qué botón hemos pulsado llamando al método `sender ()`.

```
btn1.clicked.connect (self.buttonClicked)
btn2.clicked.connect (self.buttonClicked)
```

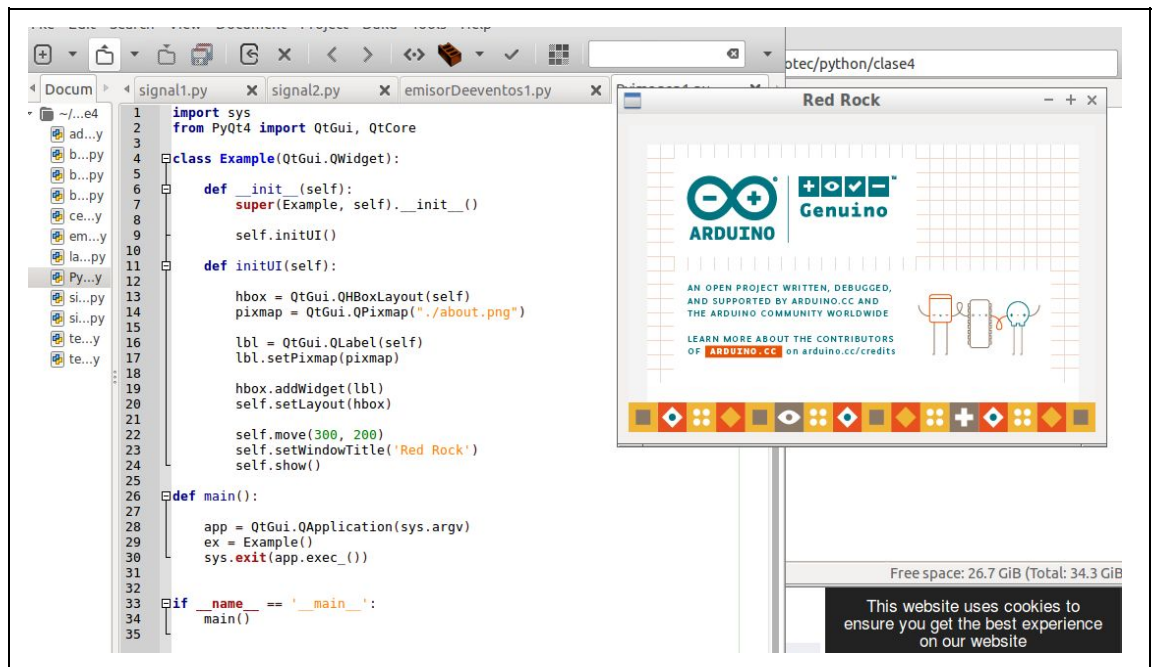
Ambos botones están conectados a la misma ranura.

```
def buttonClicked (self):
    sender = self.sender ()
    self.statusBar (). ShowMessage (sender.text () + 'fue pulsado')
```

Determinamos la fuente de señal llamando al método `sender()`. En la barra de estado de la aplicación, mostramos la etiqueta del botón pulsado.

QtGui.QPixmap

Un QtGui.QPixmap es uno de los widgets utilizados para trabajar con imágenes. Está optimizado para mostrar imágenes en pantalla. En nuestro ejemplo de código, usaremos el QtGui.QPixmap para mostrar una imagen en la ventana.



En nuestro ejemplo, mostramos una imagen en la ventana.

```
pixmap = QtGui.QPixmap ("redrock.png")
```

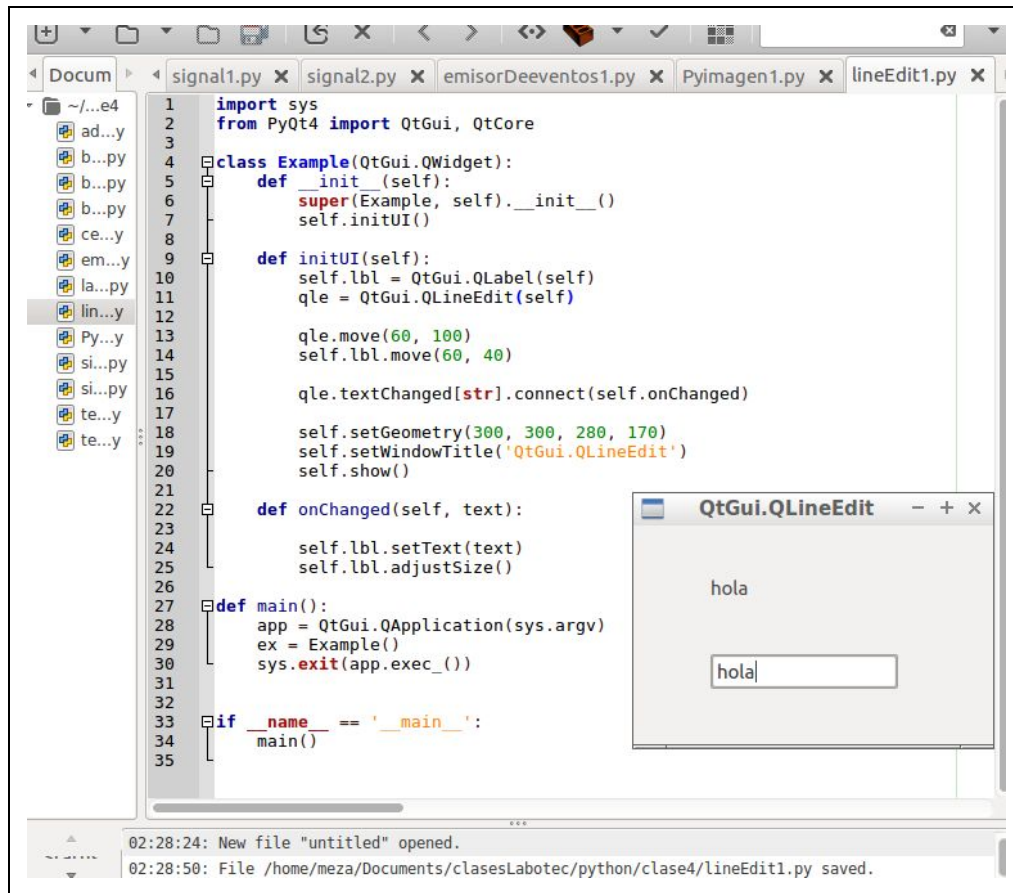
Creamos un objeto QtGui.QPixmap. Se toma el nombre del archivo como un parámetro.

```
lbl = QtGui.QLabel (self)  
lbl.setPixmap (pixmap)
```

Ponemos el pixmap en el widget QtGui.QLabel.

[QtGui.QLineEdit](#)

Un QtGui.QLineEdit es un widget que permite introducir y editar una sola línea de texto sin formato. Hay opciones de deshacer y rehacer, cortar y pegar y arrastrar y soltar disponibles para el widget.



Este ejemplo muestra un widget de edición de línea y una etiqueta. El texto que introducimos en la edición de línea se muestra inmediatamente en el widget de etiquetas.

```
qle = QtGui.QLineEdit (self)
```

Se crea el widget QtGui.QLineEdit.

```
qle.textChanged [str] .connect (self.onChanged)
```

Si cambia el texto en el widget de edición de línea, llamamos al método onChanged ().

```
def onChanged (self, text):  
    Self.lbl.setText (text)  
    Self.lbl.adjustSize()
```

Dentro del método onChanged (), establecemos el texto escrito en el widget de etiqueta. Llamamos el método adjustSize () para ajustar el tamaño de la etiqueta a la longitud del texto.