Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Tutorial de desarrollo de aplicaciones con interfaz grafica en Python y Qt (PyQt) - II (21738 lectures)

Per **Daniel Rodriguez**, ***DaniRC*** (http://www.ibiza-beach.com/)

Creado el 01/09/2006 22:49 modificado el 01/09/2006 22:49

Hoy veremos un poco de la potencialidad de Python para el calculo cientifico con Numeric y NumPy, el uso de QTableWidget, para ponerle un bonito grid a nuestras aplicaciones. El uso de la libreria pylab para generar una grafica en base a los valores de la tabla. El uso de ficheros para guardar y cargar los datos de la tabla y para acabar veremos las funciones de llamada a procedimientos en un hilo de ejecucion distinto del nuestro.

Todo esto que ahora suena a chino dentro de un rato seguro que te ha quedado super claro, solo tienes que seguir mis explicaciones. Vamos alla ....

Partimos del ejemplo de ayer[1]

En la medida de lo posible el codigo que escriba hoy lo coloreare en la web con ayuda del exportador de Scite[2]

Editar la ventana principal y sus menus

Lo primero es añadir otra opcion al menu de la pantalla principal para nuestro formulario de hoy. Eso lo haremos desde el editor QtDesigner. Si me hicisteis caso, deberiais tener el principal.ui dentro de la carpeta forms ui. Un doble click sobre principal.ui y listos para editar el menu.

Como veis, he cambiado algunas etiquetas. Ahora dice Pruebas en la parte del menu y dice Dia 1 y Dia 2 para las opciones que deben lanzar formularios.

Como habeis imaginado, ahora hay que compilar esto para que sea codigo python y al hacer eso se nos va a machacar todo lo que añadimos el dia anterior.

Sugiero renombrar el actual principal.py a old_principal.py para poder luego copiar el trozo de codigo que nos haga falta del viejo a nuevo.

Desde consola de msdos en el directorio donde esta el fichero .ui hacemos:

pyuic4 -o principal.py -x principal.ui

y obtenemos el principal.py

No olvidemos sacar principal.py de la carpeta de forms ui y ponerlo en la raiz.

Ahora retoquemos el principal.py con pyscripter.

Podemos copiar y pegar del viejo old_principal.py estas 4 primeras lineas y machacar lo que haya

```
from pydanirc.dialogs.dlgAlgo import dlgAlgo

import sys
from PyQt4 import QtCore, QtGui

class Ui_MainWindow(QtGui.QMainWindow):
```

Por un lado importamos el formulario de ayer que sigue existiendo. Por otro lado, os recuerdo que cambiamos Object por QtGui.MainWindow en la creacion de la clase.

La linea de conexion de la opcion del formulario con el metodo sigue igual -he cambiado el texto, pero no el nombre de la accion en el editor de propiedades!-

```
self.connect(self.actionOpcion1, QtCore.SIGNAL("triggered()"), self.abreDlgAlgo)
```

y añadimos otra para el nuevo formulario que aun no hemos hecho

```
self.connect(self.actionOpcion2, QtCore.SIGNAL("triggered()"), self.abreDlgDia2)
```

y ahora los metodos que construyen los dialogos. (copiar pegar del old_principal.py)

```
    def abreDlgAlgo(self):

        d = dlgAlgo()
        d.exec_()

    def abreDlgDia2(self):

        d = dlgDia2()
        d.exec_()
```

Si intentais arrancar esto, puede que recibais un error de compilacion o de ejecucion diciendo que no existe el dialogo dlgDia2 y que no se puede instanciar desde aqui.

Bueno, eso es correcto y se correjira cuando este creado el dialogo necesario. Hoy para no hacer siempre lo mismo, estoy haciendolo al reves, primero el menu principal y luego el dialogo, que esto no os confunda :-)

Lo que nos hara falta mas adelante es otra linea como la anterior de importacion de dialogos justo debajo de la primera.

from pydanirc.dialogs.dlgDia2 import dlgDia2

Guardamos los cambios y ahora nos centramos en el dialogo que va a usar tablas y todo lo demas. El nuevo principal ya esta listo, solo falta que existan el nuevo formulario para poder llamarlo. No os pongo el codigo del nuevo principal.py aqui porque no aporta nada nuevo y al final del articulo os paso un link al codigo fuente :-)

Lo primero es lo primero, dibujar la interfaz. NO USAREMOS nada de lo que esté bajo la seccion Qt3 compatible. Por algun motivo en PyQt4 se han desecho de las cosas compatibles de Qt3 y nos piden que usemos los equivalentes nativos de Qt4.

Os pongo aqui un pantallazo de la ventana que quiero que dibujeis y los nombres de cada componente para que podais seguir la leccion.

Repasemos la lista de requisitos para continuar:

1. Hemos creado un nuevo widget en Qt Designer. (frmDia2)
2. Hemos añadido 2 Label la de valores a graficar y la Introduce numero filas entre 1 y 10 (nos da igual como se llamen solo son etiquetas de decorado)
3. Hemos añadido un LineEdit para aceptar la entrada del dato que nos indicara cuantas filas meterle a la tabla. (txtNumFilas)
4. Hemos añadido 3 Buttons, botones para Graficar los datos de la tabla, guardar esos datos introducidos en fichero y leer esos datos desde un fichero. (btnGraficar, btnGuardar, btnCargar respectivamente.)
5. Hemos añadido una tabla de tipo QTableWidget y haciendo click con boton derecho la hemos editado y dejado mas presentable -como muestro en la captura de pantalla- (miTabla)

Si tenemos todo esto podemos compilar el formulario y tratar de sacar adelante la aplicacion dentro del dialogo respectivo.

pyuic4 -o frmDia2.py -x frmDia2.ui

Copiar el frmDia2.py a la carpeta de forms

Crear el dialogo dlgDia2 que extiende el frmDia2 como hicimos ayer. Ok, os dare pistas en forma de codigo :-P

Este es **dlgDia2.py** para empezar -recordad guardarlo en dialogs!-:

```
import sys
from PyQt4 import QtCore, QtGui
from pydanirc.forms.frmDia2 import Ui_frmDia2

class dlgDia2(QtGui.QDialog):
def __init__(self):

        QtGui.QDialog.__init__(self)

        #Inicializar el formulario
        self.ui = Ui_frmDia2()
        self.ui.setupUi(self)
```

Os he marcado en negrita los cambios. Vamos, que un find/replace Algo x Dia2 nos valia, de hecho es como yo lo hago :-P

Ahora conectamos los botones del formulario con las clases que le daran funcionalidad a la aplicacion:

```
# Conectar los botones con funciones del dialogo.
self.connect(self.ui.btnGraficar, QtCore.SIGNAL("clicked()"),self.hazGrafica)
self.connect(self.ui.btnGuardar, QtCore.SIGNAL("clicked()"),self.guarda)
self.connect(self.ui.btnCargar, QtCore.SIGNAL("clicked()"),self.carga)
```

Gestionar otros eventos que no solo sean los de botones, por ejemplo que un cuadro de texto se de por bueno.

```
self.connect(self.ui.txtNumFilas, QtCore.SIGNAL("editingFinished()"),self.ctrlNumFilas)
```

y para usar las validaciones propias de Qt podemos usar esto

```
self.ui.txtNumFilas.setValidator(validaNumFilas(self))
```

El QValidator merece su propia pagina de explicacion...

Validaciones en Qt con QValidator

```
class validaNumFilas(QtGui.QValidator):
    def __init__(self, parent=None, name=None):
        QtGui.QValidator.__init__(self, parent)

    def validate(self, input, posn):
        if "None".find(str(input)) == 0:
            return (QtGui.QValidator.Intermediate, posn)
        elif ((int(input) >0) and (int(input)<=10)):
            return (QtGui.QValidator.Acceptable, posn)
        else:
            return (QtGui.QValidator.Invalid, posn)

    def fixup(self, input):
        input.insert(0,"None")
```

Cada validacion que hagamos necesita una clase de validacion como la anterior que extienda QValidator. Podemos poner la clase en el mismo archivo que el dialogo, de hecho, casi lo recomiendo porque separar estas cosas en ficheros puede ser un caos.

Bueno el tema del validate es malo de entender a la primera. Cuando tecleas algo en un cuadro de texto pueden pasar 3 cosas:

a) Que sea correcto - Acceptable
b) Que sea totalmente incorrecto - Invalid
c) Que este a medio camino entre lo correcto y lo incorrecto.

Por ejemplo, quiero que pongas en un cuadro de texto "si" o "no".

Si pones "si" es correcto. Si pones "no" es correcto.
Si pones "z" ya es imposible que sea correcto, por tanto es invalido poner "z"
Si pones una "s" no es que sea imposible que eso acabe siendo correcto, de hecho si pongo un "i" sera correcto y si pongo otra cosa sera "Invalido".

Pues ese es el meollo de la cuestion. Pero imaginemos ahora que quiero cifras entre 10 y 100. ¿Que ocurre si pongo un 2? Es menor que 10 por tanto es invalido, pero esta a medio camino de ser un 20 ... no deberia rechazarlo ¿verdad? Ahi entra en juego el parametro posn, que dice en que posicion esta el caracter que estamos validando y nos permiter ser mucho mas finos con nuestras validaciones.

El metodo fixup es el metodo que se llama para impedir que quede un valor invalido en el cuadro de texto. En el ejemplo, simplemente pone el cuadro en blanco, o mejor aun a NULL -en Python el null es None-

¿Cual es la diferencia entre el QValidate y validar una entrada a posteriori? Pues la diferencia esta en que QValidate impide el tecleo de valores invalidos, mientras que la otra opcion deja al usuario escribir lo que quiera y luego hay que corregirlo. Corregir al usuario no es tan bueno como impedir que meta la pata.

En el ejemplo anterior, podreis ver a que me refiero si tratais de meter alguna cifra mayor que 10 o menor que 1.

Pero si os habeis fijado, a pesar del QValidate, no reniego de una comprobacion manual de los datos por si acaso:

self.connect(self.ui.txtNumFilas, QtCore.SIGNAL("editingFinished()"),self.ctrlNumFilas)

Asi que tengo un metodo dentro de la clase del dialogo que controla el numero de filas valido. Si Ocurre un fallo saco una ventanita de aviso de informacion con un boton de OK (cte: 1) en el ultimo parametro. El primer texto es cabecera de ventana y lo otro es el texto del aviso.

```
def ctrlNumFilas(self):
    if (int(self.ui.txtNumFilas.text())<1) or (int(self.ui.txtNumFilas.text())>10) :
        QMessageBox.information(self,'Datos incorrectos','Los datos de Numero de Filas no son validos',1)
    else:
        self.ui.miTabla.setRowCount(int(self.ui.txtNumFilas.text()))
```

Si es correcto el numero de filas lo actualizamos para que la tabla tenga tantas filas como queremos llenar.

else:

self.ui.miTabla.setRowCount(int(self.ui.txtNumFilas.text()))

Manipular la tabla y meter los datos de la tabla en un fichero.

```
def guarda(self):
 fileName = QtGui.QFileDialog.getSaveFileName(self,
                        self.tr("Guardar archivo"),
                        "C:\\Python24\\Lib\\site-packages\\pydanirc\\datos\\",
                        self.tr("All Files (*);;Text Files (*.txt)"))
 if not fileName.isEmpty():
    t = fileName
    f=open(str(t),'w+')
    for fila in range(int(self.ui.txtNumFilas.text())):
```

```
            texto = self.ui.miTabla.item(fila,0).text() + ";" +self.ui.miTabla.item(fila,1).text()+"\r\n"
            f.write(texto)
        f.close()
```

El QFileDialog.getSaveNameFile(...) Nos resuelve todos los problemas de conseguir un path y un nombre de fichero para que el usuario guarde lo que quiera donde quiera.

Lo demas es tan facil como parece. Hay algunas cosas "raras", como por ejemplo que use una variable para guardar el fileName, y luego haga un str() -convertir a string- de esa variable en lugar de pasar directamente fileName como parametro de open() . La verdad es que lo intente de la otra manera y me dio algun tipo de error que no quise perder el tiempo en resolver. Sospecho que lo que devuelve el QFileDialog sea un Qstring y no un string de python o cualquier tonteria de ese estilo, si os molesta, lo arreglais vosotros mismos :-).

Bueno, ahora viene la parte en que se recorre la tabla fila a fila desde 0 hasta el numero de filas-1. (eso es lo que hace el range())

En la tabla cada celda se llama ITEM i se puede acceder a un item concreto por sus coordenadas de FILA, COLUMNA. Luego los items puede ser cualquier tipo de QWidget, asi que segun el tipo de QWidget que tenga el item cambia la manera de accederlo. Por defecto, lo que se crean son QWidgetItem y tiene un metodo .text() para acceder a su contenido.

Meto los dos campos de la tabla en una sola linea separada por ";" y pongo los caracteres de retorno de carro de windows "\r\n" para que cambie de linea cada vez. Estoy creando algo que se llama comunmente un texto "separado por comas".

Bueno, con estas simples lineas ya tenemos la tabla volcada a fichero.

Leyendo los datos desde un fichero y metiendolos en la tabla

```
def carga(self):
    fileName = QtGui.QFileDialog.getOpenFileName(self,
                            self.tr("Abrir archivo"),
                            "C:\\Python24\\Lib\\site-packages\\pydanirc\\datos\\",
                            self.tr("All Files (*);;Text Files (*.txt)"))
    if not fileName.isEmpty():
        t = fileName

        #Para leer el fichero.
        f = open(str(t),'r')
        info = f.readlines()

        numfilas=0
        self.ui.miTabla.setRowCount(len(info))
        self.ui.txtNumFilas.setText(str(len(info)))
        for record in info:
            a=record.split(";")
            miItemc1 = QtGui.QTableWidgetItem()
            miItemc1.setText(strip(a[0]))
            self.ui.miTabla.setItem(numfilas,0,miItemc1)

            miItemc2 = QtGui.QTableWidgetItem()
            miItemc2.setText(strip(a[1]))
            self.ui.miTabla.setItem(numfilas,1,miItemc2)
            numfilas = numfilas + 1
```

Aqui lo que hacemos es capturar el nombre del fichero a abrir gracias al dialogo de Qt que nos hace todo el trabajo.
len(info) -len = longitud de**-** nos dice cuantas lineas hay y con eso actualizo las lineas de la tabla en pantalla y el

numero del cuadro de texto para que sea consistente con lo que se ve en pantalla. Son estas dos lineas:

```
self.ui.miTabla.setRowCount(len(info))
self.ui.txtNumFilas.setText(str(len(info)))
```

Leemos el fichero y hacemos un readlines que lee todas las filas y las mete en un array. Seguidamente recorremos el array al estilo python. Para cada registro en el array de informacion hacer:

Separo el contenido de cada linea por el ";" que puse antes. Y meto eso en un array auxiliar (a) a[0] sera la primera columna y a[1] la segunda columna.

Ahora lo gracioso es que no podemos hacer un item(x,y).setText() ... primero hay que crear un QTableWidgetItem, luego hacer un setText() sobre dicho item, y por ultimo colocar dicho item en la tabla.

Desde luego, los de Qt son la leche! vais a ver pocos grids tan potentes como este -de hecho en VB cuesta encontrar cosas tan potentes hasta pagando!- (vease, FlexGrid o TrueDBGrid)

Lo malo de ser la leche, es que cuando uno quiere una cosa sencillita tiene que dar 3 rodeos.

Bueno, pues ya esta, ya teneis el fichero de texto separado por comas metido en vuestra tabla. Solo una puntualizacion sobre el uso del strip.

El strip hace en python lo que trim hace en otros lenguajes. Quita blancos y caracteres de retorno de carro a una cadena. ¿Recordais que al guardar pongo un "\r\n" en cada linea, pues al leer si no los quitas esos caracteres tambien se leen y si luego guardas otra vez estas guardando "X\r\n"+"\r\n" y acabas con un fichero lleno de lineas en blanco y no sabes de donde vienen ... pues vienen de no haber quitado los retornos de carro con el strip :-)

y ahora hagamos alguna cosillas con los datos.

Graficas de los datos introducidos

Nota: Guardad estas lineas con el nombre **graficas.py** en el mismo directorio que principal.py para usarlo más adelante.

```
import sys, string, os
from pylab import *

x = []
y = []

f = open('datos.txt','r')
info = f.readlines()
for record in info:

        a = record.split(";")

        x.append(float(a[0]))
        y.append(float(a[1]))

plot(x, y, "-bo")
xlabel('Este es el eje X')
ylabel('Este es el eje Y')
title('Grafica del copon en 3 lineas mal contadas')
show()
```

Esto basicamente lee las filas del documento de texto una a una, separa los campos por ";" (punto y coma) y mete cada campo en un vector, luego esos vectores son los puntos X e Y de la grafica. Plot genera la grafica y luego show la muestra. El "-bo" dice "bolitas azules" para los puntos ;-) ... para todo lo demas usad de manual de matplotlib.

Si probais esto en python con la libreria mathplotlib, seguro que conseguis una grafica que os guste. De momento yo me voy a centrar en las 4 cosas que os pueden dar problemas y luego ya os leereis los manuales adecuados para sacarle partido a todo lo que os estoy intentando enseñar.

1. Pylab es el nombre del modulo que provee la libreria MathPlotLib.
2. MathPlotLib usa los modulos Numeric, NumPy y NumArray -pero todo eso lo tenemos instalado desde el primer dia, (ver articulo anterior)-
3. MathPlotLib esta perfectamente documentada y tiene cantidad de ejemplos para iniciarse.
4. MathPlotLib usa la interfaz Tk
5. Aunque existen versiones de Matplotlib que usan la interfaz Qt no os lo aconsejo porque es complicarse la vida para nada. (No van con Qt4 por ejemplo)
6. Efectivamente, podemos llamar una ventana Tk desde Qt y no pasa nada.
7. Efectivamente despues de haber llamado a la ventanita Tk es posible que nuestra apliacion Qt no responda o lo haga de forma incorrecta.

Bueno, el caso es que si mezclamos Qt y Tk como no usan el mismo modelo de gestion de eventos todo acaba siendo un caos y el cuelgue está casi asegurado. La solucion elegante es usar algo que embeba MathPlotLib en Qt con las herramientas de dibujo de Qt -que las hay y son un lujazo!-

Pero ya os habeis dado cuenta de que elegante no es la palabra que mejor me define. Asi que usaremos la opcion 2, que ademas introduce una tecnica muy util para mil y una cosas mas.

Ejecución de aplicaciones externas en hilos de ejecucion distintos al nuestro. (spawnv)

Lo que queremos nosotros es lanzar un script de python que funcione de forma independiente a nuestra aplicacion Qt, que tenga en cierta forma "vida propia" y no mezcle su gestion de eventos con la nuestra ni nos moleste para nada. Para ello:

1. Debemos tener un script de Python listo para ser invocado y hacer la grafica que queremos.
2. Debemos tener un documento estatico con los datos a graficar, porque no podemos mandar parametros a la otra aplicacion. -Nota: si que podemos, pero da mucho trabajo y estoy muy cansado para ponerme a parametrizar la linea de comandos ;-)-
3. Debemos lanzar una aplicacion en un hilo de ejecucion diferente al nuestro desde nuestra propia aplicacion.

Por suerte Python tiene el modulo os -de Operating System- que nos facilita el acceso a todas la llamadas de sistema que nos puedan hacer falta.

Una de tantas llamadas es la de spawnv que hace precisamente lo que queremos. Su sintaxis no es del todo sencilla y menos aun en windows -debido a los jaleos con la barras invertidas- asi que veamos un ejemplo y lo tendremos todo mas claro:

```
def hazGrafica(self):
    f=open("datos.txt",'w+')
    for fila in range(int(self.ui.txtNumFilas.text())):
        texto = self.ui.miTabla.item(fila,0).text() + ";" +self.ui.miTabla.item(fila,1).text()+"\r\n"
        f.write(texto)
    f.close()

    file=os.path.join(os.getcwd(),'grafica.py')
    os.spawnv(os.P_NOWAIT,r'C:\Python24\python.exe',('python.exe',file,))
```

El os.path.join, hace justo lo que dice, junta cadenas para hacer un path, el path es el camino hacia el fichero. El getCurrentWorkDirectory ... dice justo eso. Captura el directorio actual de trabajo, normalmente el directorio en el que esta el principal.py que es desde donde lo hemos llamado.

El os.spawnv se encarga de lanzar la aplicacion que le indicamos como si el usuario hubiese abierto el mismo la aplicacion por su cuenta. el P_NOWAIT hace referencia a que tiene que esperar a nuestra aplicacion, ni para abrirse ni para cerrarse, que es independiente de la aplicacion que lo lanzo.

Lo siguiente es la ruta al programa que queremos abrir. El nombre concreto del ejecutable a usar, el fichero que queremos que abra y la ultima coma no se que pinta aqui, pero bueno, en su dia lo hice asi y funciono y no lo he vuelto a pensar más. Solo se que si la quitais ya no funciona. :-P El el manual del modulo os de python seguro que teneis mas informacion.

Por cierto para usar el modulo os hay que hacer el import os a principio del fichero!

OJO

Bueno, con esto esta casi todo listo, pero falta el CASI. Resulta que la grafica se hace leyendo los datos de un fichero de texto que debe contener los mismos datos que la tabla que hay en pantalla. ¿Como lo hacemos?

Pues lo que haremos sera generar ese fichero **datos.txt** con los datos que hay en pantalla en el mismo momento en que se hace el click, es decir, en el metodo hazGrafica y justo antes de lanzar la grafica. Podemos "casi" copiar y pegar el codigo de guardar y ponerle el nombre del fichero "datos.txt" a mano.

Acordaros de usar "w+" que es el modo de escritura que crea el fichero si no existe y si existe lo machaca.

Aqui os dejo una captura de lo que hemos hecho hoy funcionando y en la siguiente pagina os meto el codigo coloreado por el Scite ademas de un enlace al codigo fuente en .zip

---

Codigo fuente del dialogo que es lo mas "critico".

El resto lo pillais del fichero .zip con todo el codigo ¿vale?

```
import sys,os
from string import strip
from PyQt4 import QtCore, QtGui
from pydanirc.forms.frmDia2 import Ui_frmDia2
```

```
class dlgDia2(QtGui.QDialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)

        #Inicializar el formulario
        self.ui = Ui_frmDia2()
        self.ui.setupUi(self)

        # Conectar los botones con funciones del dialogo.
        self.connect(self.ui.btnGraficar, QtCore.SIGNAL("clicked()"),self.hazGrafica)
        self.connect(self.ui.btnGuardar, QtCore.SIGNAL("clicked()"),self.guarda)
        self.connect(self.ui.btnCargar, QtCore.SIGNAL("clicked()"),self.carga)
        self.connect(self.ui.txtNumFilas, QtCore.SIGNAL("editingFinished()"),self.ctrlNumFilas)

        #Declarar una validacion de ejemplo.
        self.ui.txtNumFilas.setValidator(validaNumFilas(self))

    def hazGrafica(self):
        f=open("datos.txt",'w+')
        for fila in range(int(self.ui.txtNumFilas.text())):
            texto = self.ui.miTabla.item(fila,0).text() + ";" +self.ui.miTabla.item(fila,1).text()+"\r\n"
            f.write(texto)
        f.close()

        file=os.path.join(os.getcwd(),'grafica.py')
        os.spawnv(os.P_NOWAIT,r'C:\Python24\python.exe',('python.exe',file,))

    def guarda(self):
        fileName = QtGui.QFileDialog.getSaveFileName(self,
                                self.tr("Guardar archivo"),
                                "C:\\Python24\\Lib\\site-packages\\pydanirc\\datos\\",
                                self.tr("All Files (*);;Text Files (*.txt)"))
        if not fileName.isEmpty():
            t = fileName
            f=open(str(t),'w+')
            for fila in range(int(self.ui.txtNumFilas.text())):
                texto = self.ui.miTabla.item(fila,0).text() + ";" +self.ui.miTabla.item(fila,1).text()+"\r\n"
                f.write(texto)
            f.close()


    def carga(self):
        fileName = QtGui.QFileDialog.getOpenFileName(self,
                                self.tr("Abrir archivo"),
                                "C:\\Python24\\Lib\\site-packages\\pydanirc\\datos\\",
                                self.tr("All Files (*);;Text Files (*.txt)"))
        if not fileName.isEmpty():
            t = fileName

            #Para leer el fichero.
            f = open(str(t),'r')
            info = f.readlines()

            numfilas=0
            self.ui.miTabla.setRowCount(len(info))
            self.ui.txtNumFilas.setText(str(len(info)))
            for record in info:
                a=record.split(";")
```

```
            miItemc1 = QtGui.QTableWidgetItem()
            miItemc1.setText(strip(a[0]))
            self.ui.miTabla.setItem(numfilas,0,miItemc1)

            miItemc2 = QtGui.QTableWidgetItem()
            miItemc2.setText(strip(a[1]))
            self.ui.miTabla.setItem(numfilas,1,miItemc2)
            numfilas = numfilas + 1


    def ctrlNumFilas(self):
        if (int(self.ui.txtNumFilas.text())<1) or (int(self.ui.txtNumFilas.text())>10) :
            QMessageBox.information(self,'Datos incorrectos','Los datos de Numero de Filas no son validos',1)
        else:
            self.ui.miTabla.setRowCount(int(self.ui.txtNumFilas.text()))

class validaNumFilas(QtGui.QValidator):
    def __init__(self, parent=None, name=None):
        QtGui.QValidator.__init__(self, parent)

    def validate(self, input, posn):
        if "None".find(str(input)) == 0:
            return (QtGui.QValidator.Intermediate, posn)
        elif ((int(input) >0) and (int(input)<=10)):
            return (QtGui.QValidator.Acceptable, posn)
        else:
            return (QtGui.QValidator.Invalid, posn)

    def fixup(self, input):
        input.insert(0,"None")
```

Código fuente de todo lo visto hoy[3]

Lista de enlaces de este artículo:

1. http://bulma.net/~danirc/pyqt/pydanirc_dia1.zip
2. http://gnuwin.epfl.ch/apps/SciTE/en/install/
3. http://bulma.net/~danirc/pyqt/pydanirc_dia2.zip

E-mail del autor: danircJUBILANDOSEbulma.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=2338