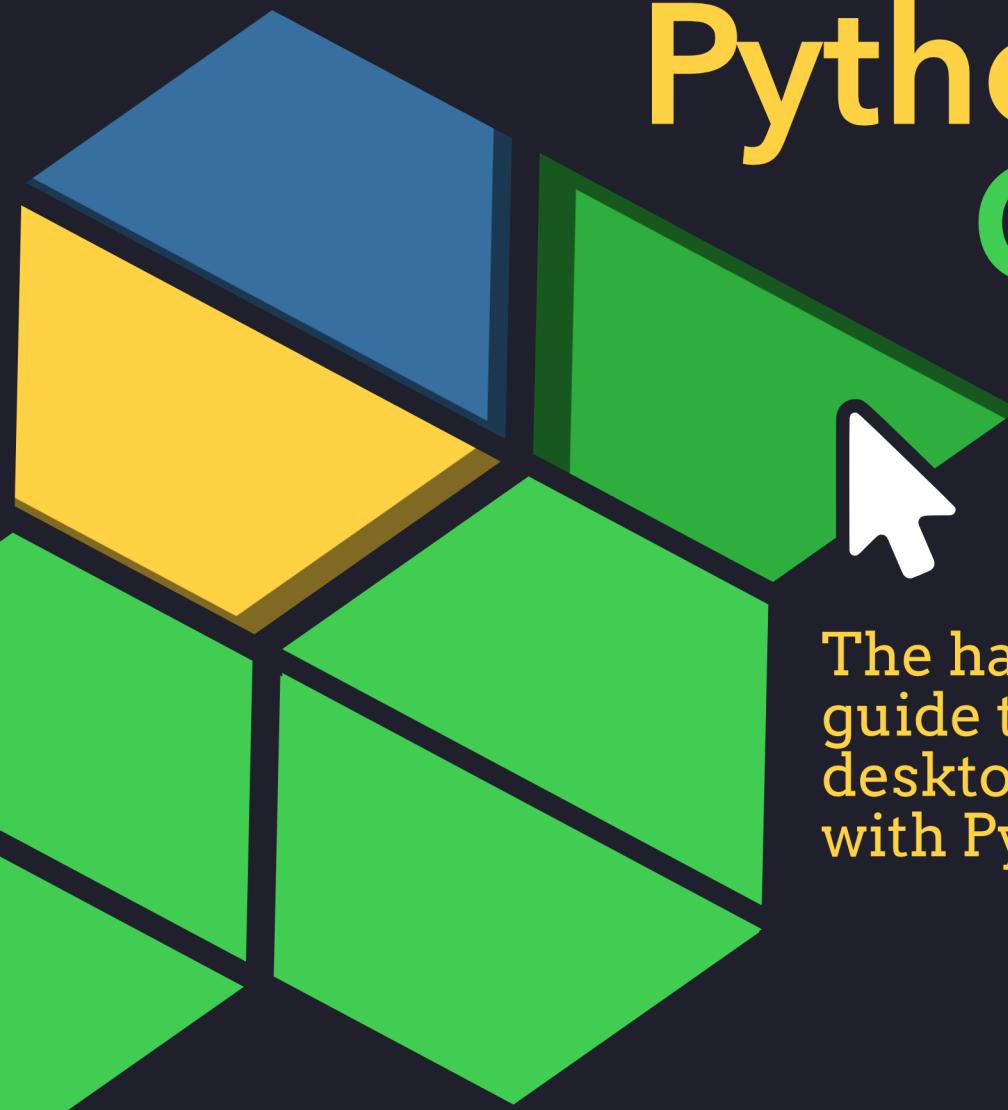


Create Simple GUI Applications *with* **Python &** **Qt5**



The hands-on
guide to making
desktop apps
with Python

Martin Fitzpatrick

Copyright ©2016-2019 CC BY-NC-SA

Cree aplicaciones GUI simples, con Python y Qt5

La guía práctica para crear aplicaciones de escritorio con Python.

Martín Fitzpatrick

Este libro está a la venta en <http://leanpub.com/create-simple-gui-applications>

Esta versión fue publicada el 2019-08-11



Esto es un [Leanpub](#) libro. Leanpub empodera a los autores y editores con el proceso de Lean Publishing. [Publicación ajustada](#) es el acto de publicar un libro electrónico en progreso utilizando herramientas livianas y muchas iteraciones para obtener comentarios de los lectores, girar hasta que tenga el libro correcto y generar tracción una vez que lo tenga.



Esta obra está autorizada bajo una [Licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported](#)

¡Tweet este libro!

Ayude a Martin Fitzpatrick haciendo correr la voz sobre este libro en [Gorjeo!](#) El hashtag sugerido para este libro es [#createsimpleguis](#).

Descubra lo que otras personas están diciendo sobre el libro haciendo clic en este enlace para buscar este hashtag en Twitter:

[# createdimpleguis](#)

Contenido

Introducción	1
Formato de libro.....	Qt y
PyQt.....	Python
3.....	2
Empezando	4
Instalación de Windows.....	PyQt5 para
Python 3.....	PyQt5 para Python
2.7.....	Instalación
Mac.....	Instalación Linux
(Ubuntu).....	6
.....	8
Características básicas de Qt	9
Mi primera ventana.....	Señales, Slots,
Eventos.....	Acciones, barras de
herramientas y menús.....	Widgets
.....	21
Diseños	37
Diálogos.....	52
.....	64
Creador de Qt	70
Creando un archivo .ui.....	70
Disposición de su ventana principal.....	Usando
su archivo .ui generado.....	Añadiendo lógica
a la aplicación.....	81
.....	84
Señales extendidas	85
Modificación de datos de señal.....	85
Señales personalizadas.....	91

Gráficos de mapa de bits y QPainter	92
QPainter.....	Dibujo de
primitivas.....	Un poco de diversión con
QPainter.....	95
116	
Creación de widgets personalizados	127
Empezando	129
paintEvent.....	131
Colocación.....	131 Actualización de la
pantalla.....	134 Dibujar la
barra.....	137 Personalización de la
barra.....	146 Adición de la interfaz
QAbstractSlider.....	151 Actualización desde la pantalla
del medidor.....	152 El código
final.....	153
La arquitectura de la vista del modelo	159
Controlador de vista de modelo.....	159 La
vista de modelo.....	160 Una vista de
modelo simple: una lista de tareas pendientes.....	160
Un almacén de datos persistente.....	174
Subprocesos múltiples	178
Preparación.....	178 El enfoque
tonto.....	181 Subprocesos y
procesos.....	184
QRunnable y QThreadPool.....	185 corredores
extendidos.....	186 Hilo
IO.....	188
QRunnable Ejemplos.....	191
Ejemplos de aplicaciones PyQt	204
Mozzarella Ashbadger.....	204
Moonsweeper.....	214
Empaquetado de aplicaciones PyQt	233
fbs: sistema de compilación fman.....	233
¿Que sigue?	254

CONTENIDO

El curso de video255
Recursos256
Tutoriales	256
Documentación.....	256 conjuntos de
iconos.....	256 Código
fuente.....	257
Derechos de autor258

Introducción

Bienvenido a *Cree aplicaciones GUI simples* la guía práctica para crear aplicaciones de escritorio profesionales con Python & Qt.

Si desea aprender a escribir aplicaciones GUI, comenzar puede ser bastante complicado. Hay muchos conceptos nuevos que debe comprender para obtener *cualquier cosa* a trabajar. Muchos tutoriales ofrecen nada más que fragmentos de código cortos sin ninguna explicación de los sistemas subyacentes y cómo funcionan juntos. Pero, como cualquier código, escribir aplicaciones GUI requiere que aprenda a pensar en el problema de la manera correcta.

En este libro, le daré los conceptos básicos realmente útiles que necesita para construir aplicaciones funcionales con el marco PyQt. Incluiré explicaciones, diagramas, tutoriales y código para asegurarme de que sepa lo que está haciendo en cada paso del camino. En poco tiempo, tendrá una aplicación Qt completamente funcional, lista para personalizar a su gusto.

Se incluye el código fuente de cada paso, pero no se limite a copiar, pegar y seguir adelante. ¡Aprenderás mucho más si experimentas en el camino!

¡Entonces empecemos!

Formato de libro

Este libro tiene el formato de una serie de ejercicios de codificación y fragmentos para permitirle explorar y aprender gradualmente los detalles de PyQt5. Sin embargo, no es posible darle una *completa* descripción general del sistema Qt en un libro de este tamaño (es enorme, este no lo es), por lo que se le anima a experimentar y explorar en el camino.

Si te encuentras pensando "Me pregunto si puedo hacer *ese*" Lo mejor que puede hacer es dejar este libro y luego *ve y descúbrelo!* Simplemente mantenga copias de seguridad periódicas de su código a lo largo del camino para que siempre tenga algo a lo que volver si lo estropea realmente.



A lo largo de estos libros también hay recuadros como este, que brindan información, consejos y advertencias. Todos ellos se pueden omitir de forma segura si tiene prisa, pero leerlos le dará un conocimiento más profundo y completo del marco Qt.

Qt y PyQt

Cuando escribe aplicaciones usando PyQt, lo que está *De Verdad* hacer es escribir aplicaciones en Qt. La biblioteca PyQt es simplemente¹ un contenedor alrededor de la biblioteca C ++ Qt, para permitir su uso en Python.

Debido a que esta es una interfaz de Python para una biblioteca de C ++, las convenciones de nomenclatura utilizadas dentro de PyQt no se adhieren a los estándares PEP8. caso mixto en vez de caso_serpiente. Si se adhiere a este estándar en sus propias aplicaciones basadas en PyQt depende completamente de usted, sin embargo, puede resultarle útil para ayudar a aclarar dónde termina el código PyQt y dónde comienza el suyo.

Además, si bien hay documentación específica de PyQt disponible, a menudo se encontrará leyendo la documentación de Qt en sí, ya que es más completa. Si lo hace, deberá traducir la sintaxis del objeto y algunos métodos que contengan nombres de funciones reservados de Python de la siguiente manera:

Qt	PyQt
Qt :: SomeValue	Qt. SomeValue
object.exec ()	object.exec_ ()
object.print ()	object.print_ ()

Python 3

Este libro está escrito para ser compatible con Python 3.4+. Python 3 es el futuro del lenguaje, y si está comenzando ahora es donde debe concentrar sus esfuerzos. Sin embargo, en reconocimiento del hecho de que muchas personas están atascadas en el soporte o el desarrollo de sistemas heredados, los ejemplos y el código utilizados en este libro también se prueban y se confirma que funcionan en Python 2.7. Cualquier incompatibilidad notable o

¹No realmente *ese* sencillo.

Los errores se marcarán con una meh-face para representar con precisión el sentimiento, por ejemplo



Python 2.7

En Python 2.7 `mapa()` devuelve un lista.

Si está utilizando Python 3, puede ignorar con seguridad su mirada indiferente.

Empezando

Antes de comenzar a codificar, primero deberá tener una instalación funcional de PyQt y Qt en su sistema. Las siguientes secciones lo guiarán a través de este proceso para las principales plataformas disponibles. Si ya tiene una instalación funcional de PyQt en su sistema Python, puede omitir esta parte de manera segura y pasar directamente a la diversión.

El código fuente completo, todos los ejemplos de este libro, está disponible para descargar desde [aquí](#).



Solo GPL

Tenga en cuenta que las siguientes instrucciones son **sólo para la instalación de la versión con licencia GPL de PyQt**. Si necesita utilizar PyQt en un proyecto no-GPL, deberá adquirir una licencia alternativa en [Computación Riverbank](#) para lanzar su software.

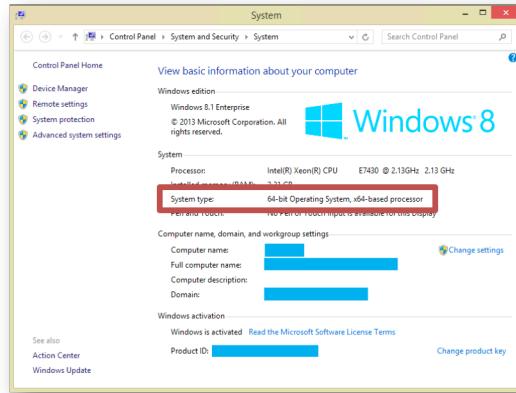


¿Documentación?

Los paquetes de ThePyQt de Riverbank no incluyen la documentación de Qt. Sin embargo, esto está disponible en línea en [docs.qt.io](#). Si tu *hacer* desea descargar la documentación puede hacerlo desde [www.qt.io](#).

Instalación Windows

PyQt5 para Windows se puede instalar como cualquier otra aplicación o biblioteca. La única pequeña complicación es que primero debe determinar si su sistema es compatible con software de 32 bits o de 64 bits. Puede determinar si su sistema admite 32 bits o 64 bits mirando el panel Sistema al que se puede acceder desde el panel de control.



El panel del sistema de Windows, donde puede averiguar si está ejecutando 64 o 32 bits.

Si su sistema *o hace* admite 64 bits (y la mayoría de los sistemas modernos lo hacen), entonces también debe verificar si su instalación actual de Python es de 32 o 64 bits. Abra un símbolo del sistema (Inicio>cmd):

1 C:\> python3

Mire la línea superior de la salida de Python, donde debería poder ver si tiene Python de 32 bits o de 64 bits instalado. Si desea cambiar a Python de 32 bits o 64 bits, debe hacerlo en este punto.

PyQt5 para Python 3

Un instalador de PyQt5 para Windows está disponible directamente del desarrollador [Computación Riverbank](#). Descargar el .exe archivos de la página vinculada, asegurándose de descargar la versión actual de 64 bits o 32 bits para su sistema. Puede instalar este archivo como cualquier otra aplicación / biblioteca de Windows.

Una vez finalizada la instalación, debería poder ejecutar pitón y importar PyQt5.

PyQt5 para Python 2.7

Desafortunadamente, si desea utilizar PyQt5 en Python2.7, hay instaladores oficiales disponibles para hacer esto. Esta parte de una política de Riverbank Computing para fomentar la transición a Python 3 y reducir su carga de soporte.

Sin embargo, técnicamente no hay nada que impida que PyQt5 se compile para Python 2.7 y las personas serviciales de [Fábrica abstracta](#) tengo [hecho exactamente eso](#).

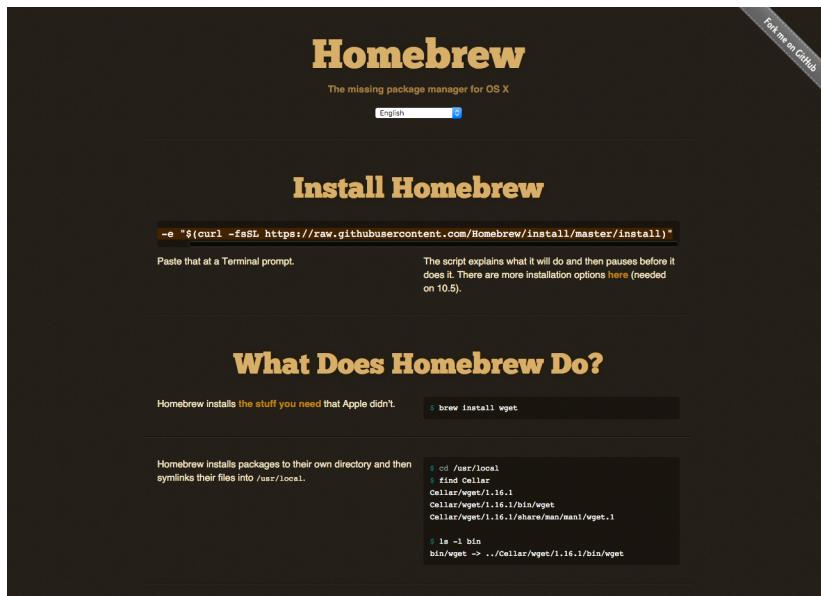
Simplemente descargue lo anterior .rar archivo y descomprimalo con 7zip (u otra aplicación de descompresión). Luego puede copiar la carpeta resultante a sus paquetes de sitio de Python carpeta - generalmente en C:\Python27\lib\site-packages\

Una vez hecho esto, debería poder ejecutar pitón y importar PyQt5.

Instalación Mac

OS X viene con una versión preinstalada de Python (2.7), sin embargo, intentar instalar PyQt5 en esto es más problemático de lo que vale la pena. Si está planeando hacer mucho desarrollo de Python, y debería hacerlo, será más fácil a largo plazo crear una instalación distinta de Python separada del sistema.

De lejos, la forma más sencilla de hacer esto es usar [Cerveza casera](#). Homebrew es un administrador de paquetes para software de línea de comandos en MacOS X. De manera útil, Homebrew también tiene una versión prediseñada de PyQt5 en sus repositorios.



Homebrew: el administrador de paquetes que falta para OS X

Para instalar Homebrew, ejecute lo siguiente desde la línea de comando:

- 1 ruby -e "**PS**curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master\Instalar en pc"
- 2



Esto también está disponible para copiar y pegar desde la página de inicio de Homebrew.

Una vez que se haya completado la instalación de Homebrew, puede instalar Python 3 y PyQt5 de la siguiente manera:

- 1 brew install python3
- 2 brew install pyqt5 --with-python-3

Una vez que se haya completado, debería poder ejecutar `python3` y importar PyQt5.

Instalación Linux (Ubuntu)

La instalación en Linux es muy sencilla ya que los paquetes para PyQt5 están disponibles en los repositorios de la mayoría de las distribuciones. En Ubuntu, puede instalar desde la línea de comandos o mediante el "Centro de software". Los paquetes que está buscando se nombran `python3-pyqt5` o `python-pyqt5` dependiendo de la versión para la que esté instalando.

También puede instalarlos desde la línea de comandos de la siguiente manera:

```
1 apt-get install python3-pyqt5
```

O para Python 2.7:

```
1 apt-get install python-pyqt5
```

Una vez finalizada la instalación, debería poder ejecutar `python3` o `pitón` y importar PyQt5.

Características básicas de Qt

¡Bienvenido a sus primeros pasos en la creación de aplicaciones gráficas! En este capítulo, se le presentarán las características básicas clave de Qt (PyQt) que se encontrará utilizando en cualquier aplicación que cree. Desarrollaremos una serie de aplicaciones pequeñas, agregando (¡y eliminando!) Características paso a paso. Utilice el código proporcionado como guía y no dude en experimentar con él, especialmente con referencia a la [Documentación de Qt](#).

Mi primera ventana

Entonces, creamos nuestra primera aplicación con ventana. Antes de mostrar la ventana en la pantalla, hay algunos conceptos clave para presentar sobre cómo se organizan las aplicaciones en el mundo Qt. Si ya está familiarizado con los bucles de eventos, puede pasar de forma segura a la siguiente sección.

El bucle de eventos y QApplication

El núcleo de todas las aplicaciones Qt es el `QApplication` clase. Cada aplicación necesita una, y solo una, `QApplication` objeto para funcionar. Este objeto contiene el *bucle de eventos* de su aplicación: el bucle central que gobierna toda la interacción del usuario con la GUI.

Cada interacción con su aplicación, ya sea presionar una tecla, hacer clic con el mouse o mover el mouse, genera una *evento* que se coloca en el *cola de eventos*. En el bucle de eventos, la cola se verifica en cada iteración y si se encuentra un evento en espera, el evento y el control se pasan al *controlador de eventos* Para el evento. El controlador de eventos se ocupa del evento y luego devuelve el control al bucle de eventos para esperar más eventos. Solo hay *una* ejecución de bucle de eventos por aplicación.



Puntos clave

- QApplication contiene el bucle de eventos Qt
- Se requiere una instancia de QApplication
- Su aplicación se queda esperando en el bucle de eventos hasta que se tome una acción.
- Solo hay *uno* bucle de eventos

Creando tu aplicación

Para comenzar a construir su aplicación, cree un nuevo archivo Python; puede llamarlo como quiera (p. Ej. MyApp.py).



¡Respaldo!

Editaremos este archivo a medida que avancemos, y es posible que desee volver a versiones anteriores de su código, así que recuerde mantener copias de seguridad periódicas durante el proceso. Por ejemplo, después de cada sección, guarde un archivo con el nombre `MyApp_<sección>.py`

El código fuente de su primera aplicación se muestra a continuación. Escríbalo literalmente y tenga cuidado de no cometer errores. Si te equivocas, Python debería hacerte saber qué está mal cuando lo ejecutes. Si no tiene ganas de escribirlo todo, [puededescargar el código fuente](#).

```
1 de PyQt5.QtWidgets importar *de
2 PyQt5.QtCore importar *de
3 PyQt5.QtGui importar *
4
5 # Solo se necesita para acceder a los argumentos de la línea de comandos
6 importar sys
7
8 # Necesita una (y solo una) instancia de QApplication por aplicación.
9 # Pase sys.argv para permitir argumentos de línea de comando para su aplicación.
10 # Si sabe que no utilizará argumentos de línea de comandos, QApplication ([] también funciona.
11 aplicación = QApplication (sys.argv)
12
13 # Inicie el ciclo de eventos.
14 aplicación.exec_()
15
16
17 # Su aplicación no llegará aquí hasta que salga y el evento
18 # bucle se ha detenido.
```

Repasemos el código línea por línea.

Empezamos importando las clases PyQt5 que necesitamos para la aplicación, desde el `QtWidgets`, `QtGui` y `QtCore` submódulos.



Este tipo de importación global desde `<module> importar *` generalmente está mal visto en Python. Sin embargo, en este caso sabemos que los nombres de clase de PyQt no entran en conflicto entre sí, o con Python en sí. Importarlos todos ahorra mucho escribir y ayuda con la compatibilidad con PyQt4.

A continuación, creamos una instancia de `QApplication`, pasando en `sys.argv` (que contiene argumentos de línea de comando). Esto nos permite pasar argumentos de línea de comando a nuestra aplicación. Si sabe que no aceptará argumentos de línea de comando, puede pasar una lista vacía en su lugar, p. Ej.

```
1 aplicación = QApplication ([])
```

Finalmente, llamamos `app.exec_()` para iniciar el bucle de eventos.



El guión bajo está ahí porque `ejecutivo` es una palabra reservada en Python y no se puede usar como nombre de función. PyQt5 maneja esto agregando un guión bajo al nombre usado en la biblioteca de C++. También lo verás por `. impresión_()`.

Para iniciar su aplicación, ejecútela desde la línea de comandos como cualquier otro script de Python, por ejemplo:

1 Python MyApp.py

O, para Python 3:

1 python3 MyApp.py

La aplicación debería ejecutarse sin errores, pero no habrá ninguna indicación de que esté sucediendo nada, aparte de quizás un indicador de ocupado. Esto es completamente normal - ¡Todavía no le hemos dicho a Qt que cree una ventana!

Cada aplicación necesita al menos una QMainWindow, aunque puede tener más de uno si lo necesita. Sin embargo, no importa cuántos tenga, su aplicación siempre se cerrará cuando se cierre la última ventana principal.

Agreguemos un QMainWindow a nuestra aplicación.

```
1 de PyQt5.QtWidgets importar *
2 PyQt5.QtCore importar *
3 PyQt5.QtGui importar *
4
5 importar sys
6
7
8 aplicación = QApplication(sys.argv)
9
10 ventana = QMainWindow()
11 ventana.show() # !!!!!IMPORTANTE!!!! Las ventanas están ocultas de forma predeterminada.
12
13 # Inicie el ciclo de eventos.
14 aplicación.exec_()
```



QMainWindow

- Enfoque principal para el usuario de su aplicación
- Cada aplicación necesita al menos una (... pero puede tener más)
- La aplicación se cerrará cuando se cierre la última ventana principal

Si inicia la aplicación, ahora debería ver su ventana principal. Tenga en cuenta que Qt crea automáticamente una ventana con las decoraciones de ventana normales, y puede arrastrarla y cambiar su tamaño como cualquier ventana normal.



¡No puedo ver mi ventana!

Usted debe *siempre* llamar `.show()` en un recién creado `QMainWindow` ya que se crean invisibles de forma predeterminada.

¡Felicitaciones, ha creado su primera aplicación Qt! No es muy interesante en este momento, así que a continuación agregaremos algo de contenido a la ventana.

Si desea crear una ventana personalizada, el mejor enfoque es una subclase `QMainWindow` y luego incluir la configuración de la ventana en el `_en eso_ cuadra`. Esto permite que el comportamiento de la ventana sea autónomo. En el siguiente paso creamos nuestra propia subclase de `QMainWindow` - podemos llamarlo Ventana principal para mantener las cosas simples.

```
1 de PyQt5.QtWidgets importar *de
2 PyQt5.QtCore importar *de
3 PyQt5.QtGui importar *
4
5 importar sys
6
7
8 # Subclase QMainWindow para personalizar la ventana principal de su aplicaciónclase
9 Ventana principal(QMainWindow):
10
11     def __init__(self, *args, **kwargs):
12         super(Ventana principal, self).__init__(*args, **kwargs)
13
14         self.setWindowTitle("Mi increíble aplicación")
15
16         etiqueta = QLabel("¡¡ESTO ES IMPRESIONANTE !!!")
17
18         # El espacio de nombres `Qt` tiene muchos atributos para personalizar
19         # widgets. Ver: http://doc.qt.io/qt-5/qt.html#etiqueta.
20         self.setAlignment(Qt.AlignCenter)
21
22         # Establecer el widget central de la ventana. El widget se expandirá
23         # para ocupar todo el espacio de la ventana de forma predeterminada.
24         self.setCentralWidget(etiqueta)
25
26
27         aplicación = QApplication(sys.argv)
28
29         ventana = Ventana principal()
30         ventana.show()
31
32         aplicación.exec_()
```

Observe cómo escribimos el `__init__` bloque con un poco de texto repetitivo para tomar los argumentos (ninguno actualmente) y pasarlos al `__init__` del parent `QMainWindow` clase.



Cuando subclases una clase Qt debes *siempre* llamar al super `__init__`- función para permitir que Qt configure el objeto.

A continuación usamos `.setWindowTitle()` para cambiar el título de nuestra ventana principal.

Luego agregamos nuestro primer widget: un `QLabel` - al medio de la ventana. Este es uno de los widgets más simples disponibles en Qt. El objeto se crea pasando el texto que desea que muestre el widget.

Establecemos la alineación del widget en el centro, por lo que se mostrará en el medio de la ventana.



El espacio de nombres Qt (qt.) está lleno de todo tipo de atributos que puede utilizar para personalizar y controlar los widgets de Qt. Cubriremos eso un poco más después,[vale la pena echarle un vistazo](#).

Finalmente, llamamos `.setCentralWidget()` en la ventana. Esto es una `QMainWindow` función específica que le permite configurar el widget que va en el medio de la ventana.

Si inicia su aplicación, debería ver su ventana nuevamente, pero esta vez con el `QLabel` widget en el medio.



¿Tienes hambre de widgets?

Cubriremos más widgets en detalle en breve, pero si está impaciente y desea avanzar, puede echar un vistazo a la [Documentación de QWidget](#). ¡Intente agregar los diferentes widgets a su ventana!

En esta sección hemos cubierto el `QApplication` clase, la `QMainWindow` class, el bucle de eventos y experimentó con la adición de un widget simple a una ventana. En la siguiente sección, veremos los mecanismos que Qt proporciona para que los widgets y las ventanas se comuniquen entre sí y con su propio código.



Guarde una copia de su archivo como `MyApp_window.py` ya que lo necesitaremos de nuevo más tarde.

Señales, ranuras, eventos

Como ya se describió, cada interacción que el usuario tiene con una aplicación Qt provoca un Evento. Hay varios tipos de eventos, cada uno de los cuales representa un tipo de interacción diferente, por ejemplo, eventos de mouse o teclado.

Los eventos que ocurren se transmiten al controlador específico del evento en el widget donde ocurrió la interacción. Por ejemplo, hacer clic en un widget provocará un QMouseEvent para ser enviado al .mousePressEvent controlador de eventos en el widget. Este controlador puede interrogar el evento para encontrar información, como qué desencadenó el evento y dónde ocurrió específicamente.

Puede interceptar eventos subclasicando y anulando la función del controlador en la clase, como lo haría con cualquier otra función. Puede optar por filtrar, modificar o ignorar eventos, pasándolos al controlador normal del evento llamando a la función de clase padre `super()`.

```
1  clase CustomButton(Botón Q):
2
3      def keyPressEvent(uno mismo, e):
4          # Mi manejo personalizado de eventos
5          super(Botón personalizado, uno mismo).keyPressEvent (e)
```

Sin embargo, imagina que quieres ver un evento en 20 botones diferentes. Subclasicar como esta ahora se convierte en una forma increíblemente tediosa de capturar, interpretar y manejar estos eventos.

```
1  clase CustomButton99(Botón Q)
2
3      def keyPressEvent(uno mismo, e):
4          # Mi manejo personalizado de eventos
5          super(CustomButton99, uno mismo).keyPressEvent (e)
```

Afortunadamente, Qt ofrece un enfoque para recibir una notificación de lo que está sucediendo en su aplicación: *Señales*.

Señales

En lugar de interceptar eventos sin procesar, las señales le permiten 'escuchar' notificaciones de ocurrencias específicas dentro de su aplicación. Si bien estos pueden ser similares a los eventos —un clic en un botón— también se pueden modificar —el texto actualizado en un cuadro. Los datos también se pueden enviar junto con una señal, por lo que, además de recibir una notificación del texto actualizado, también puede recibirla.

Los receptores de señales se llaman *Ranuras* en terminología Qt. Se proporcionan varias ranuras estándar en las clases Qt para permitirle conectar diferentes partes de su aplicación. Sin embargo, también puede usar cualquier función de Python como ranura y, por lo tanto, recibir el mensaje usted mismo.



Cargue una copia nueva de `MyApp_window.py` y guárdelo con un nuevo nombre para esta sección.

Señales básicas

Primero, veamos las señales disponibles para nuestro `QMainWindow`. Puede encontrar esta información en el [Documentación de Qt](#). Desplácese hacia abajo hasta la sección Señales para ver las señales implementadas para esta clase.

```
void iconSizeChanged(const QSize & iconSize)
void toolButtonStyleChanged(Qt::ToolButtonStyle toolButtonStyle)

> 4 signals inherited from QWidget
> 2 signals inherited from QObject
```

Documentación Qt 5 - Señales `QMainWindow`

Como puede ver, junto a los dos `QMainWindow` señales, hay 4 señales heredadas de `QWidget` y 2 señales heredadas de `QObject`. Si hace clic en el `QWidget` documentación de la señal se puede ver un `.windowTitleChanged` señal implementada aquí. A continuación, demostraremos esa señal dentro de nuestra aplicación.

```
void customContextMenuRequested(const QPoint & pos)
void windowIconChanged(const QIcon & icon)
void windowIconTextChanged(const QString & iconText)
void windowTitleChanged(const QString & title)
```

Documentación de Qt 5: señales de widget

El siguiente código ofrece algunos ejemplos de uso de la windowTitleChanged señal.

```
1  clase Ventana principal(QMainWindow):
2
3      def __en eso__(uno mismo, *argumentos, **kwargs):
4          súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kwargs)
5
6          # SEÑAL: La función conectada se llamará siempre que la ventana
7          # título ha cambiado. El nuevo título se pasará a la función.uno mismo.
8          windowTitleChanged.conectar(uno mismo.onWindowTitleChange)
9
10         # SEÑAL: La función conectada se llamará siempre que la ventana
11         # título ha cambiado. El nuevo título se descarta en el lambda y el
12         # se llama a la función sin parámetros.
13         uno mismo.windowTitleChanged.conectar(lambda X: uno mismo.my_custom_fn ())
14
15         # SEÑAL: La función conectada se llamará siempre que la ventana
16         # título ha cambiado. El nuevo título se pasa a la función.
17         # y reemplaza el parámetro predeterminado
18         uno mismo.windowTitleChanged.conectar(lambda X: uno mismo.my_custom_fn (x))
19
20         # SEÑAL: La función conectada se llamará siempre que la ventana
21         # título ha cambiado. El nuevo título se pasa a la función.
22         # y reemplaza el parámetro predeterminado. Se pasan datos adicionales desde
23         # dentro de la lambda.
24         uno mismo.windowTitleChanged.conectar(lambda X: uno mismo.my_custom_fn (x, 25))
25
26         # Esto establece el título de la ventana que activará todas las señales anteriores.
27         # enviando el nuevo título a las funciones adjuntas o lambdas como el
28         # primer parámetro.
29         uno mismo.setWindowTitle ("Mi increíble aplicación")
30
31         etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
32         etiqueta.setAlignment (Qt.Alinear al centro)
33
34         uno mismo.setCentralWidget (etiqueta)
35
36
```

```
37     # SLOT: acepta una cadena, por ejemplo, el título de la ventana, y la imprimedef
38         onWindowTitleChange(uno mismo, s):
39             impresión(s)
40
41     # SLOT: tiene parámetros predeterminados y se puede llamar sin un valordef my_custom_fn(
42         uno mismo, a="¡HELLLO!", B= 5):
43             impresión(a, b)
```

Intente comentar las diferentes señales y ver el efecto en lo que imprime la ranura.

Comenzamos creando una función que se comportará como una 'ranura' para nuestras señales.

Luego usamos .connect en el .windowTitleChanged señal. Pasamos la función que queremos llamar con los datos de la señal. En este caso, la señal envía una cadena que contiene el título de la nueva ventana.

Si ejecutamos eso, vemos que recibimos la notificación de que el título de la ventana ha cambiado.

Eventos

A continuación, echemos un vistazo rápido a los eventos. Gracias a las señales, para la mayoría de los propósitos, puede evitar felizmente el uso de eventos en Qt, pero es importante comprender cómo funcionan cuando son necesarios.

Como ejemplo, vamos a interceptar el .contextMenuEvent sobre QMainWindow. Este evento se activa siempre que se abre un menú contextual.*a punto de ser* se muestra, y se pasa un valor único evento de tipo QContextMenuEvent.

Para interceptar el evento, simplemente anulamos el método del objeto con nuestro nuevo método del mismo nombre. Entonces, en este caso, podemos crear un método en nuestroVentana principal subclase con el nombre contextMenuEvent y recibirá todos los eventos de este tipo.

```
1 def contextMenuEvent(uno mismo, evento):
2     impresión("¡Evento del menú contextual!")
```

Si agrega el método anterior a su Ventana principal class y ejecute su programa, descubrirá que al hacer clic con el botón derecho en su ventana ahora se muestra el mensaje en la declaración de impresión.

A veces, es posible que desee interceptar un evento, pero aún así desencadenar el controlador de eventos predeterminado (principal). Puede hacer esto llamando al controlador de eventos en la clase principal usando super como es normal para los métodos de la clase Python.

```
1 def contextMenuEvent(uno mismo, evento):
2     impresión("¡Evento del menú contextual!")
3     super(Ventana principal, uno mismo).contextMenuEvent (evento)
```

Esto le permite propagar eventos hacia arriba en la jerarquía de objetos, manejando solo aquellas partes de un controlador de eventos que deseé.

Sin embargo, en Qt hay otro tipo de jerarquía de eventos, construida alrededor de las relaciones de la interfaz de usuario. Los widgets que se agregan a un diseño, dentro de otro widget, pueden optar por pasar sus eventos a su interfaz de usuario principal. En widgets complejos con múltiples subelementos, esto puede permitir la delegación del manejo de eventos al widget contenedor para ciertos eventos.

Sin embargo, si ha tratado con un evento y no desea que se propague de esta manera, puede marcarlo llamando a `.aceptar()` en el evento.

```
1 clase CustomButton(Botón Q):
2
3     def evento(uno mismo, e):
4         mi.aceptar()
```

Alternativamente, si desea que se propague la llamada `.ignorar()` logrará esto.

```
1 clase CustomButton(Botón Q):
2     def evento(uno mismo,
3                e): e.ignorar()
```

En esta sección hemos cubierto señales, ranuras y eventos. Hemos demostrado algunas señales simples, que incluyen cómo pasar cada vez más datos usando lambdas. Creamos señales personalizadas y mostramos cómo interceptar eventos, transmitir el manejo y uso de eventos `.aceptar()` y `.ignorar()` para ocultar / mostrar eventos en el widget principal de la interfaz de usuario. En la siguiente sección, analizaremos dos características comunes de la GUI: barras de herramientas y menús.

Acciones, barras de herramientas y menús

A continuación, veremos algunos de los elementos comunes de la interfaz de usuario, que probablemente haya visto en muchas otras aplicaciones: barras de herramientas y menús. También exploraremos el sistema ordenado que proporciona Qt para minimizar la duplicación entre diferentes UI áreas - QAcción.

Barras de herramientas

Uno de los elementos de la interfaz de usuario más comunes es la barra de herramientas. Las barras de herramientas son barras de iconos y / o texto que se utilizan para realizar tareas comunes dentro de una aplicación, para las cuales el acceso a través de un menú sería engorroso. Son una de las características de interfaz de usuario más comunes que se ven en muchas aplicaciones. Si bien algunas aplicaciones complejas, particularmente en el paquete de Microsoft Office, han migrado a interfaces contextuales de 'cinta', la barra de herramientas estándar es generalmente suficiente para la mayoría de aplicaciones que creará.



Elementos estándar de la GUI: la barra de herramientas

Las barras de herramientas Qt admiten la visualización de iconos, texto y también pueden contener cualquier widget Qt estándar. Sin embargo, para los botones, el mejor enfoque es hacer uso del QAction sistema para colocar botones en la barra de herramientas.

Comencemos agregando una barra de herramientas a nuestra aplicación.



Cargue una copia nueva de `MyApp_window.py` y guárdelo con un nuevo nombre para esta sección.

En Qt, las barras de herramientas se crean a partir de la clase QToolBar. Para comenzar, crea una instancia de la clase y luego llama `addToolbar` en `QMainWindow`. Pasando una cadena como primer parámetro a `QToolBar` establece el nombre de la barra de herramientas, que se utilizará para identificar la barra de herramientas en la interfaz de usuario.

<< (código / toolbars_and_menus_1.py)



:Ejecutarlo!

Verás una delgada barra gris en la parte superior de la ventana. Esta es tu barra de herramientas. Haga clic derecho y haga clic en el nombre para desactivarlo.



:¿No puedo recuperar mi barra de herramientas !?

Desafortunadamente, una vez que eliminas una barra de herramientas, no hay lugar para hacer clic con el botón derecho para volver a agregarla. Por lo tanto, como regla general, desea mantener una barra de herramientas inamovible o proporcionar una interfaz alternativa para activar y desactivar las barras de herramientas.

Deberíamos hacer que la barra de herramientas sea un poco más interesante. Podríamos simplemente agregar un QPushButton widget, pero hay un mejor enfoque en Qt que le brinda algunas características interesantes, y eso es a través de QAcción. QAcción es una clase que proporciona una forma de describir interfaces de usuario abstractas. Lo que esto significa en inglés es que puedes definir múltiples elementos de interfaz dentro de un solo objeto, unificados por el efecto que tiene interactuar con ese elemento. Por ejemplo, es común tener funciones que están representadas en la barra de herramientas pero también en el menú; piense en algo como Editar-> Cortar, que está presente tanto en el menú Edición como en la barra de herramientas como un par de tijeras, y también a través de la atajo de teclado Ctrl-X (Cmd-X en Mac).

Sin QAcción tendrías que definir esto en varios lugares. Pero con QAcción puedes definir un solo QAcción, definiendo la acción desencadenada y luego agregue esta acción tanto al menú como a la barra de herramientas. Cada QAcción tiene nombres, mensajes de estado, íconos y señales a las que puede conectarse (y mucho más).

En el código a continuación, puede ver esto primero. QAcción adicional.

```
1  clase Ventana principal(QMainWindow):
2
3      def __en eso__(uno mismo, *argumentos, **kwargs):
4          súper(Ventana principal, uno mismo).__en eso__(*argumentos, * *kwargs)
5
6          uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8          etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9          etiqueta.setAlignment(Qt.Alinear al centro)
10
11         uno mismo.setCentralWidget (etiqueta)
12
13         barra de herramientas = QToolBar ("Mi barra de herramientas principal")uno
14         mismo.addToolBar (barra de herramientas)
15
16         button_action = QAcción ("Tu botón", uno mismo) button_action.setStatusTip (
17             "Este es tu botón") button_action.motivado.conectar(uno mismo.
18             onMyToolBarButtonClick) barra de herramientas.addAction (button_action)
19
20
21
22
```

```
23     def onMyToolBarButtonClick(uno mismo, s):  
24         impresión("hacer clic", s)
```

Para empezar, creamos la función que aceptará la señal de QAction para que podamos ver si está funcionando. A continuación definimos elQAcción sí mismo. Al crear la instancia podemos pasar una etiqueta para la acción y / o un ícono. También debe pasar en cualquierQObject para actuar como padre de la acción, aquí estamos pasando uno mismo como referencia a nuestra ventana principal. Extrañamente paraQAcción el elemento padre se pasa como parámetro final.

A continuación, podemos optar por establecer una sugerencia de estado: este texto se mostrará en la barra de estado una vez que tengamos una. Finalmente conectamos el.motivado señal a la función personalizada. Esta señal se disparará siempre queQAcción es *motivado* (o activado).



¡Ejecutarlo!

Debería ver su botón con la etiqueta que ha definido. Haga clic en él y nuestra función personalizada emitirá un "clic" y el estado del botón.



¿Por qué la señal es siempre falsa?

La señal pasada indica si el botón está *comprobado*, y dado que nuestro botón no se puede marcar, solo se puede hacer clic, siempre es falso. En breve le mostraremos cómo hacerlo comprobable.

A continuación, podemos agregar una barra de estado.

Creamos un objeto de barra de estado llamando QStatusBar para obtener un nuevo objeto de la barra de estado y luego pasarlo a .setStatusbar. Dado que no necesitamos cambiar la configuración de la barra de estado, también podemos pasarlo a medida que la creamos, en una sola línea:

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __en eso__(uno mismo, *argumentos, **kwargs):  
4         super(Ventana principal, uno mismo).__en eso__(*argumentos, *kwargs)  
5  
6         uno mismo.setWindowTitle ("Mi increíble aplicación")  
7  
8         etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")  
9         etiqueta.setAlignment (Qt.Alinear al centro)  
10  
11         uno mismo.setCentralWidget (etiqueta)  
12  
13         barra de herramientas = QToolBar ("Mi barra de herramientas principal")uno  
14         mismo.addToolBar (barra de herramientas)  
15  
16         button_action = QAcción ("Tu botón", uno mismo) button_action.setStatusTip (diseñar  
17         "Este es tu botón") button_action.motivado.conectar(uno mismo.  
18         onMyToolBarButtonClick) barra de herramientas.addAction (button_action)  
19  
20  
21         uno mismo.setStatusBar (QStatusBar (uno mismo))  
22  
23  
24     def onMyToolBarButtonClick(uno mismo, s):  
25         impresión("hacer clic", s)
```



¡Ejecutarlo!

Pase el mouse sobre el botón de la barra de herramientas y verá el texto de estado en la barra de estado.

A continuación, vamos a convertir nuestro QAcción comutable, por lo que al hacer clic se activará, al hacer clic de nuevo se desactivará. Para hacer esto, simplemente llamamos setCheckable (verdadero) sobre el QAcción objeto.

```

1 clase Ventana principal(QMainWindow):
2
3     def __en eso__(uno mismo, *argumentos, **kwargs):
4         super(Ventana principal, uno mismo).__en eso__(*argumentos, *kwargs)
5
6         uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8         etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9         etiqueta.setAlignment (Qt.Alinear al centro)
10
11        uno mismo.setCentralWidget (etiqueta)
12
13        barra de herramientas = QToolBar ("Mi barra de herramientas principal")uno
14        mismo.addToolBar (barra de herramientas)
15
16        button_action = QAcción ("Tu botón", uno mismo) button_action.setStatusTip (
17            "Este es tu botón") button_action.motivado.conectar(uno mismo.
18            onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra de
19            herramientas.addAction (button_action)
20
21
22        uno mismo.setStatusBar (QStatusBar (uno mismo))
23
24
25    def onMyToolBarButtonClick(uno mismo, s):
26        impresión("hacer clic", s)

```



¡Ejecutarlo!

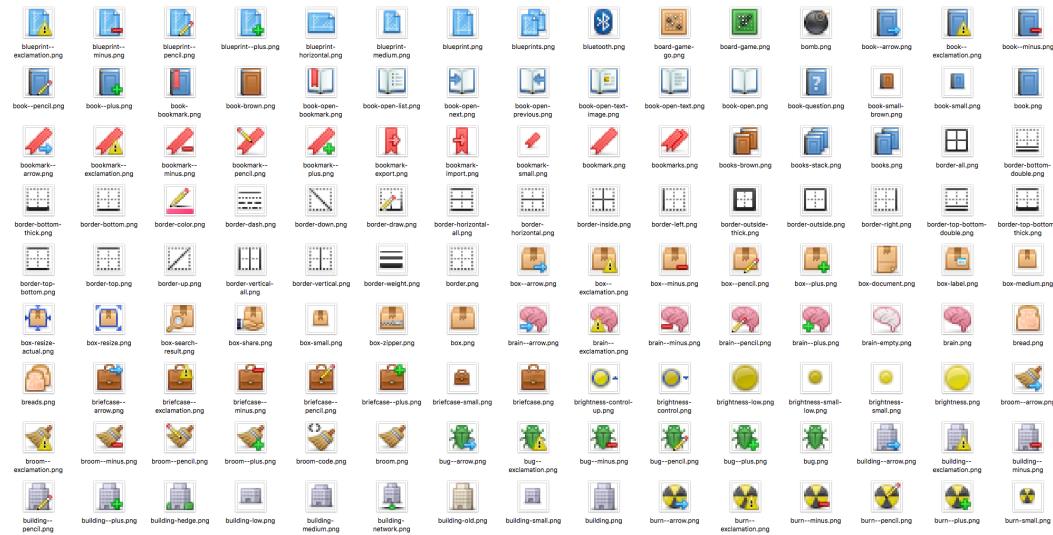
Haga clic en el botón para ver cómo cambia de estado marcado a no marcado. Tenga en cuenta que la función de ranura personalizada que creamos ahora alterna la salida Cierto y Falso.



. alternado

También hay una .alternado señal, que solo emite una señal cuando se alterna el botón. Pero el efecto es idéntico, por lo que en su mayoría no tiene sentido.

Las cosas se ven bastante mal en este momento, así que agreguemos un ícono a nuestro botón. Para esto te recomiendo descargar el [conjunto de iconos de fuga](#) por el diseñador Yusuke Kamiyamane. Es un gran conjunto de hermosos iconos de 16x16 que pueden dar a tus aplicaciones un aspecto profesional agradable. Está disponible gratuitamente y solo se requiere atribución cuando distribuye su aplicación, aunque estoy seguro de que el diseñador también agradecería algo de dinero en efectivo si tiene algo de repuesto.



Conjunto de iconos de fuga - Yusuke Kamiyamane

Seleccione una imagen del conjunto (en los ejemplos aquí he seleccionado el archivo `bug.png`) y cópielo en la misma carpeta que su código fuente. Para agregar el ícono al QAcción (y por lo tanto el botón) simplemente lo pasamos como el primer parámetro al crear el QAcción. Si el ícono está en la misma carpeta que su código fuente, puede copiarlo a

También debe informar a la barra de herramientas qué tan grandes son sus íconos, de lo contrario, su ícono estará rodeado por una gran cantidad de relleno. Puedes hacer esto llamando `setIconSize()` con un `QSize` objeto.

```

1  clase Ventana principal(QMainWindow):
2
3      def __en eso__(uno mismo, *argumentos, **kwargs):
4          super(Ventana principal, uno mismo).__en eso__(*argumentos, * kwargs)
5
6          uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8          etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9          etiqueta.setAlignment (Qt.Alinear al centro)
10
11         uno mismo.setCentralWidget (etiqueta)
12
13         barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14         barra de herramientas.setIconSize (QSize (dieciséis,dieciséis))uno mismo.
15         addToolBar (barra de herramientas)

dieciséis
17         button_action = QAction (QIcon ("bug.png"), "Tu botón", uno mismo) button_action.
18         setStatusTip ("Este es tu botón") button_action.motivado.conectar(uno mismo.
19         onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra de
20         herramientas.addAction (button_action)
21
22
23         uno mismo.setStatusBar (QStatusBar (uno mismo))
24
25
26     def onMyToolBarButtonClick(uno mismo, s):
27         impresión("hacer clic", s)

```



¡Ejecutarlo!

La QAction ahora está representada por un ícono. Todo debería funcionar exactamente como antes.

Tenga en cuenta que Qt utiliza la configuración predeterminada de su sistema operativo para determinar si debe mostrar un ícono, texto o un ícono y texto en la barra de herramientas. Pero puede anular esto usando.setStyle. Esta ranura acepta cualquiera de los siguientes indicadores del Qt. espacio de nombres:

Bandera	Comportamiento
Qt.ToolButtonIconOnly	Solo ícono, sin texto
Qt.ToolButtonTextOnly	Solo texto, sin ícono
Qt.ToolButtonTextBesideIcon	Ícono y texto, con texto junto al ícono
Qt.ToolButtonTextUnderIcon	Ícono y texto, con texto debajo del ícono
Qt.ToolButtonIconOnly	Solo ícono, sin texto
Qt.ToolButtonFollowStyle	Siga el estilo del escritorio del host



¿Qué estilo debo usar?

El valor predeterminado es `Qt.ToolButtonFollowStyle`, lo que significa que su aplicación seguirá de forma predeterminada la configuración estándar / global para el escritorio en el que se ejecuta la aplicación. Esto generalmente se recomienda para que su aplicación se sienta *comonativo* como sea posible.

Finalmente, podemos agregar algunos bits y sacudidas más a la barra de herramientas. Agregaremos un segundo botón y un widget de casilla de verificación. Como se mencionó, literalmente puede colocar cualquier widget aquí, así que siéntase libre de volverse loco. No te preocupes por el `QCheckBox` tipo, lo cubriremos más tarde.

```

1  clase Ventana principal(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(Ventana principal, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle ("Mi increíble aplicación")
7
8          etiqueta = QLabel ("¡¡ESTO ES IMPRESIONANTE !!!")
9          etiqueta.setAlignment(Qt.AlignCenter)
10
11         self.setCentralWidget (etiqueta)
12
13         barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14         barra de herramientas.setIconSize (QSize (diecisésis,diecisésis))
15         addToolBar (barra de herramientas)
16
17         button_action = QAction (QIcon ("bug.png"), "Tu botón", self)
```

```
18     button_action.setStatusTip ("Este es tu botón") button_action.motivado.conectar(  
19         uno mismo.onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra  
20         de herramientas.addAction (button_action)  
21  
22  
23     barra de herramientas.addSeparator ()  
24  
25     button_action2 = QAcción (QIcon ("bug.png"), "Tu botón2", uno mismo) button_action2  
26         .setStatusTip ("Este es tu botón2") button_action2.motivado.conectar(uno mismo.  
27         onMyToolBarButtonClick) button_action2.setCheckable (Cierto) barra de herramientas.  
28         addAction (button_action)  
29  
30  
31     barra de herramientas.addWidget (QLabel ("Hola"))  
32     barra de herramientas.addWidget (QCheckBox ())  
33  
34     uno mismo.setStatusBar (QStatusBar (uno mismo))  
35  
36  
37     def onMyToolBarButtonClick(uno mismo, s):  
38         impresión("hacer clic", s)
```

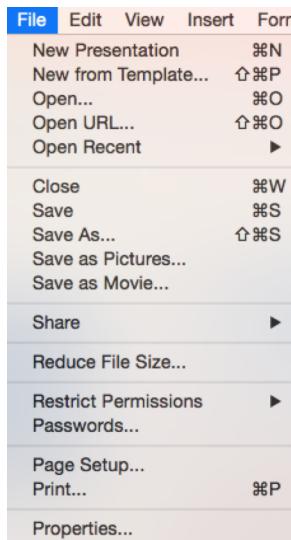


¡Ejecutarlo!

Ahora verá varios botones y una casilla de verificación.

Menús

Los menús son otro componente estándar de UIS. Por lo general, se encuentran en la parte superior de la ventana o en la parte superior de una pantalla en una Mac. Permiten el acceso a todas las funciones estándar de la aplicación. Existen algunos menús estándar, por ejemplo Archivo, Editar, Ayuda. Los menús se pueden anidar para crear árboles jerárquicos de funciones y, a menudo, admiten y muestran atajos de teclado para acceder rápidamente a sus funciones.



Elementos estándar de la GUI: menús

Para crear un menú, creamos una barra de menú que llamamos `.Barra de menús()` en `QMainWindow`. Agregamos un menú en nuestra barra de menú llamando `.addMenu()`, pasando el nombre del menú. Lo he llamado 'YExpediente'. El ampersand define una tecla rápida para saltar a este menú al presionar Alt.

```

1   clase Ventana principal(QMainWindow):
2
3       def __en eso__(uno mismo, *argumentos, **kkwargs):
4           súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kkwargs)
5
6           uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8           etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9           etiqueta.setAlignment (Qt.Alinear al centro)
10
11           uno mismo.setCentralWidget (etiqueta)
12
13           barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14           barra de herramientas.setIconSize (QSize (dieciséis,dieciséis))uno mismo.
15           addToolBar (barra de herramientas)
```

```
17 button_action = QAction (QIcon ("bug.png"), "& Tu botón", uno mismo) button_action.  
18 setStatusTip ("Este es tu botón") button_action.motivado.conectar(uno mismo).  
19 onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra de herramientas.  
20 addAction (button_action)  
21  
22  
23 barra de herramientas.addActionSeparator ()  
24  
25 button_action2 = QAction (QIcon ("bug.png"), "Tu & button2", uno mismo)  
26 button_action2.setStatusTip ("Este es tu botón2") button_action2.motivado.conectar(uno  
27 mismo.onMyToolBarButtonClick) button_action2.setCheckable (Cierto) barra de  
28 herramientas.addAction (button_action)  
29  
30  
31 barra de herramientas.addWidget (QLabel ("Hola"))  
32 barra de herramientas.addWidget (QCheckBox ())  
33  
34 uno mismo.setStatusBar (QStatusBar (uno mismo))  
35  
36 menú = uno mismo.Barra de menús()  
37  
38 menú Archivo = menú.addMenu ("&Expediente")  
39 menú Archivo.addAction (button_action)  
40  
41  
42 def onMyToolBarButtonClick(uno mismo, s):  
43     impresión("hacer clic", s)
```



Teclas rápidas en Mac

Esto no será visible en Mac. Tenga en cuenta que esto es diferente a un método abreviado de teclado; lo cubriremos en breve.

A continuación, agregamos algo al menú. Aquí es donde el poder de QAction entra en juego. Podemos reutilizar lo ya existente QAction para agregar la misma función al menú. Haga clic en él y notará que se puede alternar: hereda las características de los QAction.

Ahora agreguemos algunas cosas más al menú. Aquí agregaremos un separador al menú, que aparecerá como una línea horizontal en el menú, y luego agregaremos el segundo QAcción creamos.

```
1 clase Ventana principal(QMainWindow):
2
3     def __en eso__(uno mismo, *argumentos, **kwargs):
4         super(Ventana principal, uno mismo).__en eso__(*argumentos, * kwargs)
5
6         uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8         etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9         etiqueta.setAlignment(Qt.Alinear al centro)
10
11        uno mismo.setCentralWidget (etiqueta)
12
13        barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14        barra de herramientas.setIconSize (QSize (dieciséis,dieciséis))uno mismo.
15        addToolBar (barra de herramientas)
16
17        button_action = QAcción (QIcon ("bug.png"), "& Tu botón", uno mismo) button_action.
18        setStatusTip ("Este es tu botón") button_action.motivado.conectar(uno mismo.
19        onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra de herramientas.
20        addAction (button_action)
21
22
23        barra de herramientas.addSeparator ()
24
25        button_action2 = QAcción (QIcon ("bug.png"), "Tu & button2", uno mismo)
26        button_action2.setStatusTip ("Este es tu botón2") button_action2.motivado.conectar(uno
27        mismo.onMyToolBarButtonClick) button_action2.setCheckable (Cierto) barra de
28        herramientas.addAction (button_action)
29
30
31        barra de herramientas.addWidget (QLabel ("Hola"))
32        barra de herramientas.addWidget (QCheckBox ())
33
34        uno mismo.setStatusBar (QStatusBar (uno mismo))
```

```

35
36 menú = uno mismo.Barra de menús()
37
38 menú Archivo = menú.addMenu ("&Expediente")
39 menú Archivo.addAction (button_action)
40 file_menu.addSeparator () file_menu.addAction
41 (button_action2)
42
43
44 def onMyToolBarButtonClick(uno mismo, s):
45   impresión("hacer clic", s)

```



¡Ejecutarlo!

Debería ver dos elementos del menú con una línea entre ellos.

También puede usar ampersand para agregar teclas de aceleración al menú y permitir que se use una sola tecla para saltar a un elemento del menú cuando está abierto. Nuevamente, esto no funciona en Mac.

Para agregar un submenú, simplemente cree un nuevo menú llamando `addMenu()` en el menú principal. A continuación, puede agregarle acciones de la forma habitual. Por ejemplo:

```

1 clase Ventana principal(QMainWindow):
2
3   def __en eso__(uno mismo, *argumentos, **kwargs):
4     súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kwargs)
5
6     uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8     etiqueta = QLabel ("¡¡¡ESTO ES IMPRESIONANTE !!!")
9     etiqueta.setAlignment (Qt.Alinear al centro)
10
11     uno mismo.setCentralWidget (etiqueta)
12
13     barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14     barra de herramientas.setIconSize (QSize (dieciséis,dieciséis))

```

```
15     uno mismo.addToolBar (barra de herramientas)

diciéndole
17     button_action = QAction (QIcon ("bug.png"), "& Tu botón", uno mismo) button_action.
18         setStatusTip ("Este es tu botón") button_action.motivado.conectar(uno mismo).
19             onMyToolBarButtonClick) button_action.setCheckable (Cierto) barra de herramientas.
20                 addAction (button_action)

21
22
23     barra de herramientas.addSeparator ()

24
25     button_action2 = QAction (QIcon ("bug.png"), "Tu & button2", uno mismo)
26         button_action2.setStatusTip ("Este es tu botón2") button_action2.motivado.conectar(uno
27             mismo.onMyToolBarButtonClick) button_action2.setCheckable (Cierto) barra de
28                 herramientas.addAction (button_action)

29
30
31     barra de herramientas.addWidget (QLabel ("Hola"))
32     barra de herramientas.addWidget (QCheckBox ())

33
34     uno mismo.setStatusBar (QStatusBar (uno mismo))

35
36     menú = uno mismo.Barra de menús()

37
38     menú Archivo = menú.addMenu (
39         "&Expediente") menú Archivo.addAction
40             (button_action) file_menu.addSeparator ()

41
42     file_submenu = menú Archivo.addMenu ("Submenú")
43         submenu_archivo.addAction (button_action2)

44
45
46     def onMyToolBarButtonClick(uno mismo, s):
47         impresión("hacer clic", s)
```

Finalmente, agregaremos un atajo de teclado al QAction. Usted define un atajo de teclado pasando setKeySequence () y pasando la secuencia de teclas. Cualquier secuencia de teclas definida aparecerá en el menú.



Atajos ocultos

Tenga en cuenta que el método abreviado de teclado está asociado con el QAcción y seguirá funcionando si el QAcción se agrega a un menú o una barra de herramientas.

Las secuencias de teclas se pueden definir de varias formas, ya sea pasándolas como texto, utilizando nombres de teclas del espacio de nombres Qt o utilizando las secuencias de teclas definidas en el espacio de nombres Qt. Utilice este último siempre que pueda para garantizar el cumplimiento de los estándares del sistema operativo.

El código completo, que muestra los botones y menús de la barra de herramientas, se muestra a continuación.

```
1  clase Ventana principal(QMainWindow):
2
3      def __en eso__(uno mismo, *argumentos, **kwargs):
4          super(Ventana principal, uno mismo).__en eso__(*argumentos, * *kwargs)
5
6          uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8          etiqueta = QLabel ("¡¡ESTO ES IMPRESIONANTE !!!")
9          etiqueta.setAlignment(Qt.Alinear al centro)
10
11         uno mismo.setCentralWidget (etiqueta)
12
13         barra de herramientas = QToolBar ("Mi barra de herramientas principal")
14         barra de herramientas.setIconSize (QSize (dieciséis,dieciséis))uno mismo.
15         addToolBar (barra de herramientas)
16
17         button_action = QAcción (QIcon ("bug.png"), "& Tu botón", uno mismo) button_action.
18         setStatusTip ("Este es tu botón") button_action.motivado.conectar(uno mismo.
19         onMyToolBarButtonClick) button_action.setChecked (Ciento)
20
21         # Puede ingresar atajos de teclado usando nombres de teclas (por ejemplo, Ctrl + p)
22         # Identificadores de espacio de nombres de Qt (por ejemplo, Qt.CTRL + Qt.Key_P)
23         # o identificadores independientes del sistema (por ejemplo,
24         QKeySequence.Print)button_action.setShortcut (QKeySequence ("Ctrl + p"))
25
26         barra de herramientas.addAction (button_action)
27
```

```
28     barra de herramientas.addSeparator ()  
29  
30     button_action2 = QAcción (QIcon ("bug.png"), "Tu & button2", uno mismo)  
31     button_action2.setStatusTip ("Este es tu botón2") button_action2.motivado.conectar(uno  
32     mismo.onMyToolBarButtonClick) button_action2.setCheckable (Cierto) barra de  
33     herramientas.addAction (button_action)  
34  
35  
36     barra de herramientas.addWidget (QLabel ("Hola"))  
37     barra de herramientas.addWidget (QCheckBox ())  
38  
39     uno mismo.setStatusBar (QStatusBar (uno mismo))  
40  
41     menú = uno mismo.Barra de menús()  
42  
43     menú Archivo = menú.addMenu (  
44         "&Expediente") menú Archivo.addAction  
45         (button_action) file_menu.addSeparator ()  
46  
47     file_submenu = menú Archivo.addMenu ("Submenú")  
48     submenú_archivo.addAction (button_action2)  
49  
50  
51     def onMyToolBarButtonClick(uno mismo, s):  
52         impresión("hacer clic", s)
```



Guarde una copia de su archivo como MyApp_menus.py ya que lo necesitaremos de nuevo más tarde.

Widgets

En Qt (y en la mayoría de las interfaces de usuario), 'widget' es el nombre que se le da a un componente de la interfaz de usuario con el que el usuario puede interactuar. Las interfaces de usuario se componen de varios widgets, organizados dentro de la ventana.

Qt viene con una gran selección de widgets disponibles e incluso le permite crear sus propios widgets personalizados y personalizados.



Cargue una copia nueva de `MyApp_window.py` y guárdelo con un nuevo nombre para esta sección.

Gran lista de widgets

Una lista completa de widgets está disponible en la documentación de Qt, pero echémosle un vistazo rápidamente en acción.

Todos los widgets de Qt5.

```
1 de PyQt5.QtWidgets importar *
2 PyQt5.QtCore importar *
3 PyQt5.QtGui importar *
4
5 # Solo se necesita para acceder a los argumentos de la línea de comandos
6 importar sys
7
8
9 # Subclase QMainWindow para personalizar la ventana principal de su aplicación
10 clase Ventana principal(QMainWindow):
11
12     def __en eso__(uno mismo, *argumentos, **kargs):
13         súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kargs)
14
15         uno mismo.setWindowTitle("Mi Aplicación impresionante")
16
17         diseño = QVBoxLayout()
18         widgets = [QCheckBox,
19                    QComboBox,
20                    QDateEdit,
21                    QDateTimeEdit,
22                    QDial,
23                    QDoubleSpinBox,
24                    QFontComboBox,
25                    QLCDNumber,
26                    QLabel,
27                    QLineEdit,
28                    QProgressBar,
29                    QPushButton,
30                    QRadioButton,
31                    QSlider,
32                    QSpinBox,
33                    QTimeEdit]
34
35
36     por w en widgets:
37         diseño.addWidget(w())
```

```
38
39
40     widget = QWidget()
41     widget.setLayout(diseño)
42
43     # Establecer el widget central de la ventana. El widget se expandirá
44     # para ocupar todo el espacio de la ventana de forma predeterminada.uno
45     mismo.setCentralWidget(widget)
46
47
48     # Necesita una (y solo una) instancia de QApplication por aplicación.
49     # Pase sys.argv para permitir argumentos de línea de comando para su aplicación.
50     # Si sabe que no utilizará argumentos de línea de comandos, QApplication ([] también funciona.
51     aplicación=QApplication(sys.argv)
52
53     ventana = Ventana principal()
54     ventana.show() # ¡¡¡¡IMPORTANTE!!!! Las ventanas están ocultas de forma predeterminada.
55
56     # Inicie el ciclo de eventos.
57     aplicación.exec_()
58
59
60     # Su aplicación no llegará aquí hasta que salga y el evento
61     # bucle se ha detenido.
```

Todo esto vamos a tomar el esqueleto de nuestra aplicación y reemplazar el QLabel con un QWidget. Esta es la forma genérica de un widget Qt.

Aquí no lo estamos usando directamente. Aplicamos una lista de widgets, en un diseño, que cubriremos en breve, y luego agregamos elQWidget como el widget central de la ventana. El resultado es que llenamos la ventana con widgets, con elQWidget actuando como un contenedor.



Widgets compuestos

Tenga en cuenta que es posible utilizar este QWidget truco de diseño para crear widgets compuestos personalizados. Por ejemplo, puedes tomar una baseQWidget y superponga un diseño que contenga varios widgets de diferentes tipos. Este 'widget' puede insertarse en otros diseños como de costumbre. Más adelante cubriremos los widgets personalizados con más detalle.

Echemos un vistazo a todos los widgets de ejemplo, de arriba a abajo:

Widget	Que hace
QCheckbox	Una casilla de verificación
QComboBox	Un cuadro de lista desplegable
QDateEdit	Para editar fechas y horarios
QDateTimeEdit	Para editar fechas y horarios
QDial	Dial giratorio
QDoubleSpinBox	Una ruleta numérica para flotadores
QFontComboBox	Una lista de fuentes
QLCDNumber	Una pantalla LCD bastante fea Solo
QLabel	una etiqueta, no interactiva Ingrese
QLineEdit	una línea de texto
QProgressBar	Una barra de progreso
QPushButton	Un botón
QRadioButton	Un conjunto de alternancia, con un solo elemento activo Un control
QSlider	deslizante
QSpinBox	Una ruleta de números enteros
QTimeEdit	Para tiempos de edición

En realidad, hay más widgets que este, ¡pero no encajan tan bien! Puedes verlos todos consultando la documentación. Aquí vamos a echar un vistazo más de cerca a un subconjunto de los más útiles.

QLabel

Empezaremos el recorrido con QLabel, posiblemente uno de los widgets más simples disponibles en la caja de herramientas de Qt. Este es un texto simple de una línea que puede colocar en su aplicación. Puede configurar el texto pasando una cadena a medida que lo crea:

```
1 widget = QLabel ("Hola")
```

O, usando el .setText () método:

```
1 widget = QLabel ("1")          # La etiqueta se crea con el texto 1.  
2 widget.setText ("2")          # La etiqueta ahora muestra 2.
```

También puede ajustar los parámetros de la fuente, como el tamaño de la fuente o la alineación del texto en el widget.

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __en eso__(uno mismo, *argumentos, **kwargs):  
4         súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kwargs)  
5  
6         uno mismo.setWindowTitle ("Mi Aplicación impresionante ")  
7  
8         widget = QLabel ("Hola")  
9         fuente = widget_fuente()  
10        fuente.setPointSize (30)  
11        widget.setFont (fuente)  
12        widget.setAlignment (Qt.AlignHCenter | Qt.AlignVCenter)  
13  
14        uno mismo.setCentralWidget (widget)
```



Sugerencias de fuentes

Tenga en cuenta que si desea cambiar las propiedades de la fuente de un widget, generalmente es mejor *Actual/fuente*, actualícela y luego aplíquela nuevamente. Esto asegura que la fuente se mantenga de acuerdo con las convenciones del escritorio.

La alineación se especifica mediante el uso de una bandera de la Qt. espacio de nombres. Las banderas disponibles para alineación horizontal son:

Bandera	Comportamiento
Qt.AlignLeft	Se alinea con el borde izquierdo.
Qt.AlignRight	Se alinea con el borde derecho.
Qt.AlignHCenter	Se centra horizontalmente en el espacio disponible.
Qt.AlignJustify	Justifica el texto en el espacio disponible.

Las banderas disponibles para alineación vertical son:

Bandera	Comportamiento
Qt.AlignTop	Se alinea con la parte superior. Se alinea con la parte inferior.
Qt.AlignBottom	Se centra verticalmente en el espacio disponible.

Puede combinar banderas juntas usando tuberías (|), sin embargo, tenga en cuenta que solo puede usar una bandera de alineación vertical u horizontal a la vez.

1 align_top_left = Qt.Alinear a la izquierda | Qt.AlignTop



Banderas de Qt

Tenga en cuenta que utiliza un *Otubería|* para combinar las dos banderas (no A y B). Esto se debe a que los indicadores son máscaras de bits que no se superponen. p.ej Qt.AlignLeft tiene el valor hexadecimal 0x0001, tiempo Qt.AlignBottom es 0x0040. Al hacer OR juntos obtenemos el valor 0x0041 representando 'abajo a la izquierda'. Este principio se aplica a todas las demás banderas Qt combinatorias.

Si esto le parece un galimatías, no dude en ignorarlo y seguir adelante. ¡Solo recuerde usar |!

Finalmente, también hay una bandera taquigráfica que se centra en ambas direcciones simultáneamente:

Bandera	Comportamiento
Qt.AlignCenter	Centra horizontalmente y verticalmente

Extrañamente, también puedes usar QLabel para mostrar una imagen usando .setPixmap(). Esta

acepta un *mapa de píxeles*, que puede crear pasando un nombre de archivo de imagen a QPixmap. En los archivos de ejemplo proporcionados con este libro, puede encontrar un archivo `otje.jpg` que puede mostrar en su ventana de la siguiente manera:

```
1  widget.setPixmap (QPixmap ('otje.jpg'))
```



Otgon "Otje" Ginge el gato.

Qué cara más hermosa. De forma predeterminada, la imagen se escala manteniendo su relación de aspecto. Si desea que se estire y escale para que se ajuste completamente a la ventana, puede configurar .setScaledContents (verdadero)sobre el QLabel.

1 `widget.setScaledContents (Cierto)`

QCheckBox

El siguiente widget a mirar es QCheckBox () que, como su nombre indica, presenta una casilla de verificación para el usuario. Sin embargo, al igual que con todos los widgets de Qt, hay varias opciones configurables para cambiar el comportamiento del widget.

```

1  clase Ventana principal(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(Ventana principal, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle ("Mi increíble aplicación")
7
8          self.widget = QCheckBox ()
9          self.widget.setCheckState (Qt.Comprobado)
10
11         # Para tristate: self.setCheckState (Qt.PartiallyChecked)
12         # O: self.setTriState (True)
13         self.stateChanged.connect(self.show_state)
14
15         self.setCentralWidget (self.widget)

diecisés
17
18     def show_state(self, s):
19         if s == Qt.Comprobado:
20             print(s)

```

Puede marcar el estado de la casilla de verificación mediante programación.setCheckState(). El primero acepta `O` cierto o Falso representando marcado o no marcado respectivamente. Sin embargo, `con.setCheckState` también especifica un estado marcado particular usando un Qt. bandera de espacio de nombres:

Bandera	Comportamiento
Qt. Sin marcar	El elemento está desmarcado
Qt.PartiallyChecked	El elemento está parcialmente marcado. El
Qt. Revisado	elemento está desmarcado.

Una casilla de verificación que admite un (Qt.PartiallyChecked) estado es com-

conocido como 'tri-estado', que no está activado ni desactivado. Una casilla de verificación en este estado se muestra comúnmente como una casilla de verificación atenuada y se usa comúnmente en arreglos de casillas de verificación jerárquicas donde los subelementos están vinculados a casillas de verificación principales.

Si establece el valor en `Qt.PartiallyChecked` la casilla de verificación se convertirá en triestado. Tú también puedes `setTriState (verdadero)` para configurar el soporte triestado en un `También`. También puede configurar una casilla de verificación para que sea triestado sin configurar el estado actual como parcialmente marcado por utilizando `.setTriState (verdadero)`

Puede notar que cuando el script se está ejecutando, el número de estado actual se muestra como un `Ent` con `marcado = 2`, `sin marcar = 0`, y `parcialmente marcado = 1`. No es necesario recordar estos valores, el `Qt.Revisado` variable de espacio de nombres `== 2` por ejemplo. Este es el valor de las banderas respectivas de estos estados. Esto significa que puede probar el estado usando `estado == Qt.Checked`.

QComboBox

los QComboBox es una lista desplegable, cerrada por defecto con una flecha para abrirla. Puede seleccionar un solo elemento de la lista, y el elemento seleccionado actualmente se muestra como una etiqueta en el widget. El cuadro combinado se adapta a la selección de una opción de una larga lista de opciones.



Probablemente haya visto el cuadro combinado utilizado para la selección de tipos de letra, o tamaño, en aplicaciones de procesamiento de texto. Aunque Qt en realidad proporciona un cuadro combinado de selección de fuente específico como QFontComboBox.

Puede agregar elementos a un QComboBox pasando una lista de cadenas a .addItems(). Los elementos se agregarán en el orden en que se proporcionan.

```
1 clase Ventana principal(QMainWindow):
2
3     def __en eso__(uno mismo, *argumentos, **kkwargs):
4         súper(Ventana principal, uno mismo).__en eso__(*argumentos, * kkwargs)
5
6         uno mismo.setWindowTitle ("Mi Impresionante Aplicación ")
7
8         widget = QComboBox ()
9         widget.addItems(["Uno", "Dos", "Tres"])
10
11         # La señal predeterminada de currentIndexChanged envía el índicewidget.
12         currentIndexChanged.conectar( uno mismo.index_changed)
13
14         # La misma señal puede enviar una cadena de texto
15         widget.currentIndexChanged [str].conectar( uno mismo.text_changed)
16
17         uno mismo.setCentralWidget (widget)
18
19
20     def index_changed(uno mismo, I): # yo es un int
21         impresión(I)
22
23     def text_changed(uno mismo, s): # s es una str
24         impresión(s)
```

los `.currentIndexChanged` La señal se activa cuando se actualiza el elemento seleccionado actualmente, pasando por defecto el índice del elemento seleccionado en la lista. Sin embargo, al conectarse a la señal, también puede solicitar una versión alternativa de la señal agregando [str] (Piense en la señal como una dictar). En cambio, esta interfaz alternativa proporciona la etiqueta del elemento seleccionado actualmente, que a menudo es más útil.

QComboBox también puede ser editable, lo que permite a los usuarios ingresar valores que no están actualmente en la lista e insertarlos o simplemente usarlos como un valor. Para hacer que el cuadro sea editable:

- 1 `widget.setEditable (Cierto)`

También puede establecer una bandera para determinar cómo se maneja la inserción. Estas banderas se almacenan en el `QComboBox` clase en sí y se enumeran a continuación:

Bandera	Comportamiento
<code>QComboBox.NoInsert</code>	Sin inserto
<code>QComboBox.InsertAtTop</code>	Insertar como primer elemento
<code>QComboBox.InsertAtCurrent</code>	Reemplazar el elemento seleccionado actualmente Insertar
<code>QComboBox.InsertAtBottom</code>	después del último elemento
<code>QComboBox.InsertAfterCurrent</code>	Insertar después del elemento
<code>QComboBox.InsertBeforeCurrent</code>	actual Insertar antes del elemento
<code>QComboBox.InsertAlfabéticamente</code>	actual Insertar en orden alfabético

Para usarlos, aplique la bandera de la siguiente manera:

- 1 `widget.setInsertPolicy (QComboBox.Insertar alfabéticamente)`

También puede limitar la cantidad de elementos permitidos en la caja usando `.setMaxCount`, p.ej

- 1 `widget.setMaxCount (10)`

QListBox

próximo QListBox. Es muy similar a QComboBox, difiriendo principalmente en las señales disponibles.

```
1 clase Ventana principal(QMainWindow):
2
3     def __init__(self, *args, **kwargs):
4         super(Ventana principal, self).__init__(*args, **kwargs)
5
6         self.setWindowTitle("Mi Impresionante Aplicación")
7
8         widget = QListWidget()
9         widget.addItem("Uno", "Dos", "Tres")
10
11         # En QListWidget hay dos señales separadas para el elemento, y el str
12         tr
13         widget.currentItemChanged.connect(widget.itemChanged)
14         currentTextChanged.connect(self.itemChanged)
15
16         self.setCentralWidget(widget)
17
18
19     def index_changed(self, I): # No es un índice, yo es un QListWidgetItem
20         impresión(I.text())
21
22     def text_changed(self, s): # s es una str
23         impresión(s)
```

QListBox ofrece un currentItemChanged señal que envía el QListWidgetItem (los elemento del cuadro de lista), y un currentTextChanged señal que envía el texto.

QLineEdit

los QLineEdit widget es un cuadro de edición de texto simple de una sola línea, en el que los usuarios pueden ingresar datos. Se utilizan para campos de formulario o configuraciones donde no hay una lista restringida de entradas válidas. Por ejemplo, al ingresar una dirección de correo electrónico o el nombre de una computadora.

```
1  clase Ventana principal(QMainWindow):
2
3      def __init__(self, *args, **kwargs):
4          super(Ventana principal, self).__init__(*args, **kwargs)
5
6          self.setWindowTitle("Mi increíble aplicación")
7
8          self.widget = QLineEdit()
9          self.widget.setMaxLength(10)
10         self.widget.setPlaceholderText("Ingresa tu texto")
11
12         # self.widget.setReadOnly(True) # descomente esto para hacer de solo lectura
13
14         self.widget.volverPresionado.connect(self.retorno_primido)
15         self.selectionChanged.connect(self.widget.selectionChanged)
16         self.textChanged.connect(self.widget.textChanged)
17         self.textEdited.connect(self.widget.textEdited)
18
19         self.setCentralWidget(self.widget)
20
21
22     def return_pressed(self):
23         impresión("¡Retorno presionado!")
24         self.centralWidget().setText("¡AUGE!")
25
26     def selection_changed(self):
27         impresión("Selección cambiada")
28         impresión(self.centralWidget().selectedText())
29
30     def text_changed(self, s):
31         impresión("Texto cambiado ...")
```

```

32     impresión(s)
33
34     def text_edited(uno mismo, s):
35         impresión("Texto editado ...")
36     impresión(s)

```

Como se demuestra en el código anterior, puede establecer una longitud máxima para el texto en una edición de línea.

Diseños

Hasta ahora, hemos creado una ventana con éxito y le hemos agregado un widget. Sin embargo, normalmente queremos agregar más de un widget a una ventana y tener cierto control sobre dónde termina. Para hacer esto en Qt usamos *diseños*. Hay 4 diseños básicos disponibles en Qt, que se enumeran en la siguiente tabla.

Diseño	Comportamiento
QHBoxLayout	Disposición horizontal lineal
QVBoxLayout	Disposición vertical lineal
QGridLayout	En cuadrícula indexable XxY Apilados
QStackedLayout	(z) uno frente al otro



Diseñador Qt

En realidad, puede diseñar y diseñar su interfaz gráficamente utilizando el diseñador Qt, que cubriremos más adelante. Aquí estamos usando código, ya que es más sencillo comprender y experimentar con el sistema subyacente.

Como puede ver, hay tres diseños posicionales disponibles en Qt. los QVBoxLayout, QHBoxLayout y QGridLayout. Además también hay QStackedLayout cuales le permite colocar widgets uno encima del otro dentro del mismo espacio, pero mostrando solo un diseño a la vez.



Cargue una copia nueva de `MyApp_window.py` y guárdelo con un nuevo nombre para esta sección.

Antes de comenzar a experimentar con los diferentes diseños, primero vamos a crear un widget personalizado muy simple que podemos usar para visualizar los diseños que usamos. Agregue el siguiente código a su archivo como una nueva clase en el nivel superior:

Widget de color personalizado

```
1 clase Color(QWidget):  
2  
3     def __init__(self, color, *arguments, **kwargs):  
4         super(Color, self).__init__(*arguments, **kwargs)  
5         self.setAutoFillBackground(True)  
6  
7         palette = self.palette()  
8         palette.setColor(QPalette.Window, QColor(color))self.  
9         setPalette(palette)
```

En este código subclasificamos QWidget para crear nuestro propio widget personalizado Color. Aceptamos un solo parámetro al crear el widget:color (a str). Primero establecemos. setAutoFillBackground para True para decirle al widget que llene automáticamente su fondo con el enfriador de ventana. A continuación, obtenemos la paleta actual (que es la paleta de escritorio global por defecto) y cambiamos la paleta actual.QPalette.Window color a un nuevo QColor descrito por el valor color pasamos. Finalmente aplicamos esta paleta al widget. El resultado final es un widget que se rellena con un color sólido, que especificamos cuando lo creamos.

Si lo anterior le resulta confuso, no se preocupe demasiado. Cubriremos los widgets personalizados con más detalle más adelante. Por ahora, es suficiente que comprenda que al llamar puede crear un widget rojo con relleno sólido haciendo lo siguiente:

```
1 Color('rojo')
```

Primero probemos nuestro nuevo Color widget usando para llenar toda la ventana en un solo color. Una vez que esté completo, podemos agregarlo al QMainWindow utilizando .setCentralWidget y obtenemos una ventana roja sólida.

Agregar un widget al diseño

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle("Mi increíble aplicación")  
7  
8         widget = Color('rojo')self.  
9         setCentralWidget(widget)
```



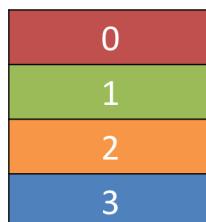
¡Ejecutarlo!

Aparecerá la ventana, llena completamente con el color rojo. Observe cómo el widget se expande para llenar todo el espacio disponible.

A continuación, veremos cada uno de los diseños de Qt disponibles por turno. Tenga en cuenta que para agregar nuestros diseños a la ventana necesitaremos un maniquí QWidget para mantener el diseño.

QVBoxLayout widgets dispuestos verticalmente

Con QVBoxLayout organizas los widgets uno encima del otro de forma lineal. Agregar un widget lo agrega al final de la columna.



Un QVBoxLayout, llenado de arriba a abajo.

Agreguemos nuestro widget a un diseño. Tenga en cuenta que para agregar un diseño al QMainWindow tenemos que aplicarlo a un maniquí QWidget. Esto nos permite luego usar.setCentralWidget para aplicar el widget (y el diseño) a la ventana. Nuestro

Los widgets de colores se organizarán en el diseño, contenido dentro del QWidget en la ventana. Primero agregamos el widget rojo como antes.

QVBoxLayout

```
1 clase Ventana principal(QMainWindow):
2
3     def __init__(self, *args, **kwargs):
4         super(Ventana principal, self).__init__(*args, *kwargs)
5
6         self.setWindowTitle("Mi increíble aplicación")
7
8         diseño = QVBoxLayout()
9
10        diseño.addWidget(Color('rojo'))
11
12        widget = QWidget()
13        widget.setLayout(diseño)
14        self.setCentralWidget(widget)
```



¡Ejecutarlo!

Observe el borde ahora visible alrededor del widget rojo. Este es el espaciado del diseño; veremos cómo ajustarlo más adelante.

Si agrega algunos widgets de colores más al diseño, notará que se alinean verticalmente en el orden en que se agregan.

QVBoxLayout

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle ("Mi increíble aplicación")  
7  
8         diseño = QVBoxLayout ()  
9  
10        diseño.addWidget (Color ('rojo')) diseño.  
11        diseño.addWidget (Color ('verde')) diseño.  
12        diseño.addWidget (Color ('azul'))  
13  
14        widget = QWidget ()  
15        widget.setLayout (diseño) self.setCentralWidget (widget)
```

QHBoxLayout widgets dispuestos horizontalmente

QHBoxLayout es el mismo, excepto que se mueve horizontalmente. Agregar un widget lo agrega al lado derecho.



Un QHBoxLayout, llenado de izquierda a derecha.

Para usarlo, simplemente podemos cambiar el QVBoxLayout a un QHBoxLayout. Las cajas ahora fluyen de izquierda a derecha.

QHBoxLayout

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle("Mi increíble aplicación")  
7  
8         diseño = QHBoxLayout()  
9  
10        diseño.addWidget(Color('rojo')) diseño.  
11        addWidget(Color('verde')) diseño.  
12        addWidget(Color('azul'))  
13  
14        widget = QWidget()  
15        widget.setLayout(diseño) self.setCentralWidget(widget)
```

Diseños de anidación

Para diseños más complejos, puede utilizar diseños más.addLayout en un diseño. A continuación agregamos un QVBoxLayout en el principal QHBoxLayout. Si agregamos algunos widgets al QVBoxLayout, se organizarán verticalmente en la primera ranura del diseño principal.

Diseños anidados

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle("Mi Aplicación impresionante")  
7  
8         layout1 = QHBoxLayout()  
9         layout2 = QVBoxLayout()
```

```
10    layout3 = QVBoxLayout ()  
11  
12    layout2.addWidget (Color ('rojo')) layout2.  
13    addWidget (Color ('amarillo')) layout2.  
14    addWidget (Color ('púrpura'))  
15  
16    layout1.setLayout (diseño2)  
17  
18    layout1.addWidget (Color ('verde'))  
19  
20    layout3.addWidget (Color ('rojo')) layout3.  
21    addWidget (Color ('púrpura'))  
22  
23    layout1.setLayout (diseño3)  
24  
25    widget = QWidget ()  
26    widget.setLayout (diseño1) uno mismo.  
27    setCentralWidget (widget)
```

**¡Ejecutarlo!**

Los widgets deben organizarse en 3 columnas horizontalmente, con la primera columna también conteniendo 3 widgets apilados verticalmente. ¡Experimentar!

Puede establecer el espacio alrededor del diseño usando `.setContentsMargins` o establecer el espacio entre elementos usando `.setSpacing`.

```
1 layout1.setContentsMargins (0,0,0,0) layout1.  
2 setSpacing (20)
```

El siguiente código muestra la combinación de widgets anidados y márgenes y espacio de diseño. Experimente con los números hasta que los sienta.

Márgenes y espaciado

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle("Mi Aplicación impresionante")  
7  
8         layout1 = QHBoxLayout()  
9         layout2 = QVBoxLayout()  
10        layout3 = QVBoxLayout()  
11  
12        layout1.setContentsMargins(0,0,0,0) layout1.  
13        setSpacing(20)  
14  
15        layout2.addWidget(Color('rojo')) layout2.  
diciéndole  
16        addWidget(Color('amarillo')) layout2.  
17        addWidget(Color('púrpura'))  
18  
19        layout1.setLayout(diseño2)  
20  
21        layout1.addWidget(Color('verde'))  
22  
23        layout3.addWidget(Color('rojo')) layout3.  
24        addWidget(Color('púrpura'))  
25  
26        layout1.setLayout(diseño3)  
27  
28        widget = QWidget()  
29        widget.setLayout(diseño1) self.  
30        setCentralWidget(widget)
```

QGridLayout widgets dispuestos en una cuadrícula

Tan útiles como son, si intentas usar QVBoxLayout y QHBoxLayout para diseñar varios elementos, por ejemplo, para un formulario, le resultará muy difícil asegurarse de que los widgets de diferentes tamaños se alineen. La solución a esto es QGridLayout.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

Un QGridLayout que muestra las posiciones de la cuadrícula para cada ubicación.

QGridLayout le permite colocar elementos específicamente en una cuadrícula. Usted especifica posiciones de fila y columna para cada widget. Puede omitir elementos y se dejarán vacíos.

Útil, para QGridLayout no es necesario llenar todas las posiciones en la cuadrícula.

			0,3
	1,1		
		2,2	
3,0			

Un QGridLayout con espacios vacíos.

QGridLayout

```

1 clase Ventana principal(QMainWindow):
2
3     def __init__(self, *argumentos, **kwargs):
4         super(Ventana principal, self).__init__(*argumentos, **kwargs)
5
6         self.setWindowTitle("Mi increíble aplicación")
7
8         diseño = QGridLayout()

```

```
9  
10     diseño.addWidget (Color ('rojo'), diseño. 0, 0)  
11     addWidget (Color ('verde'), diseño.           1, 0)  
12     addWidget (Color ('azul'), diseño.addWidget1, 1)  
13     (Color ("púrpura"),                      2, 1)  
14  
15     widget = QWidget ()  
diseño  
16     widget.setLayout (diseño) uno mismo.  
17     setCentralWidget (widget)
```

QStackedLayout varios widgets en el mismo espacio

El diseño final que cubriremos es el QStackedLayout. Como se describe, este diseño le permite colocar elementos directamente uno frente al otro. A continuación, puede seleccionar qué widget desea mostrar. Puede usar esto para dibujar capas en una aplicación de gráficos o para imitar una interfaz similar a una pestaña. Tenga en cuenta que también hay QStackedWidget que es un widget contenedor que funciona exactamente de la misma manera. Esto es útil si desea agregar una pila directamente a un QMainWindow con .setCentralWidget.



QStackedLayout: en uso, solo el widget superior está visible, que es por defecto el primer widget agregado al diseño.



QStackedLayout, con el segundo (1) widget seleccionado y traído al frente.

QStackedLayout

```
1 clase Ventana principal(QMainWindow):
2
3     def __en eso__(uno mismo, *argumentos, **kwargs):
4         super(Ventana principal, uno mismo).__en eso__(*argumentos, * *kwargs)
5
6         uno mismo.setWindowTitle ("Mi increíble aplicación")
7
8         diseño = QStackedLayout ()
9
10        diseño.addWidget (Color ('rojo')) diseño.
11        addWidget (Color ('verde')) diseño.
12        addWidget (Color ('azul')) diseño.addWidget
13        (Color ('amarillo'))
14
15        diseño.setCurrentIndex (3)
16
17        widget = QWidget ()
18        widget.setLayout (diseño)uno mismo.
19        setCentralWidget (widget)
```

QStackedWidget es exactamente cómo funcionan las vistas con pestañas en las aplicaciones. Solo una vista ('pestaña') es visible a la vez. Puede controlar qué widget mostrar en cualquier momento

mediante el uso `.setCurrentIndex()` o `.setCurrentWidget()` para configurar el elemento por el index (en el orden en que se agregaron los widgets) o por el propio widget.

A continuación se muestra una breve demostración con `QStackedLayout` en combinación con `QPushButton` para proporcionar una interfaz similar a una pestaña para una aplicación:

Interfaz con pestañas

```
1 clase Ventana principal(QMainWindow):
2
3     def __init__(self, *args, **kwargs):
4         super(Ventana principal, self).__init__(*args, **kwargs)
5
6         self.setWindowTitle("Mi Aplicación impresionante")
7
8         diseño_de_página = QVBoxLayout()
9         button_layout = QHBoxLayout()
10        diseño = QStackedLayout()
11
12        diseño_de_página.addLayout(button_layout)
13        diseño_de_página.addLayout(diseño)
14
15        for n, color in enumerate(['rojo', 'verde', 'azul', 'amarillo']):
16            btn = QPushButton(str(color))
17            btn.pressed.connect(lambda n=n: diseño.setCurrentIndex(n))
18            diseño_botón.diseño.addWidget(btn)
19
20            widget = QWidget()
21            widget.setLayout(diseño_de_página)
22            self.setCentralWidget(widget)
```

Con mucha ayuda. Qt en realidad proporciona un `TabWidget` incorporado que proporciona este tipo de diseño listo para usar, aunque en forma de widget. Debajo, la demostración de la pestaña se recrea usando `QTabWidget`:

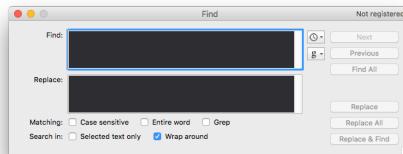
QTabWidget

```
1 clase Ventana principal(QMainWindow):  
2  
3     def __init__(self, *args, **kwargs):  
4         super(Ventana principal, self).__init__(*args, **kwargs)  
5  
6         self.setWindowTitle ("Mi increíble aplicación")  
7  
8  
9         pestañas = QTabWidget ()  
10        pestañas.setDocumentMode (True)  
11        pestañas.setTabPosition (QTabWidget.Este)  
12        pestañas.setMovable (True)  
13  
14        for n, color in enumerate(['rojo','verde','azul','amarillo']):  
15            pestañas.addTab (Color (color), color)  
16  
17        self.setCentralWidget (pestañas)
```

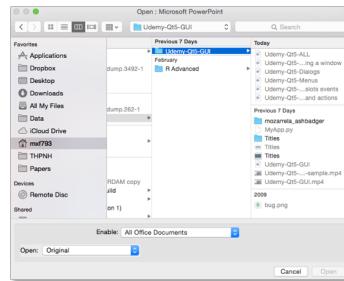
Como puede ver, es un poco más sencillo y un poco más atractivo. Puede establecer la posición de las pestañas usando las direcciones cardinales, alternar si las pestañas se pueden mover con `setMoveable` y activar y desactivar un 'modo de documento' que (en OS X) muestra una interfaz de pestañas más delgada. Encontraremos más de estos widgets avanzados más adelante.

Diálogos

Los cuadros de diálogo son componentes GUI útiles que le permiten *comunicar* con el usuario (de ahí el diálogo de nombre). Se utilizan comúnmente para abrir / guardar archivos, configuraciones, preferencias o para funciones que no encajan en la interfaz de usuario principal de la aplicación. Son pequeños modales (*bloqueo*) ventanas que se encuentran frente a la aplicación principal hasta que se cierran. Qt en realidad proporciona una serie de cuadros de diálogo 'especiales' para los casos de uso más comunes, lo que le permite aprovechar las herramientas específicas de escritorio para una mejor experiencia de usuario.



Funciones de GUI estándar: un cuadro de diálogo de búsqueda



Características estándar de la GUI: un cuadro de diálogo Abrir archivo

En Qt, los cuadros de diálogo son manejados por el QDialog clase. Para crear un nuevo cuadro de diálogo, simplemente cree un nuevo objeto de QDialog tipo (o una subclase), pasando un widget padre, p. ej. QMainWindow como su parent.

Creemos el nuestro QDialog, usaremos nuestro código de ejemplo de menú para que podamos iniciar una ventana de diálogo cuando se presiona un botón en la barra de herramientas.



Cargue una copia nueva de MyApp_menus.py y guárdelo con un nuevo nombre para esta sección.

```
1 clase Ventana principal(QMainWindow):  
2  
3     # def __init__ etc.  
4     #... no se muestra para mayor claridad  
5  
6     def onMyToolBarButtonClick(uno mismo, s):  
7         impresión("hacer clic", s)  
8  
9  
10    dlg = QDialog (uno mismo)  
11    dlg.setWindowTitle ("¡HOLA!") dlg.  
12    exec_()  
13
```

En la función disparada (que recibe la señal del botón) creamos la instancia de diálogo, pasando nuestro QMainWindow instancia como parent. Esto hará que el diálogo sea modal de QMainWindow. Esto significa que el diálogo bloqueará completamente la interacción con la ventana principal.

Una vez que hemos creado el diálogo, lo iniciamos usando `.exec_()` - al igual que lo hicimos para QApplication para crear el bucle de eventos principal de nuestra aplicación. Eso no es una coincidencia: cuando ejecuta el QDialog Se crea un bucle de eventos completamente nuevo, específico para el diálogo.



¡Recuerde que solo puede haber un bucle de eventos Qt ejecutándose en cualquier momento! Los QDialog bloquean completamente la ejecución de su aplicación. No inicie un diálogo y espere que suceda algo más en cualquier otro lugar de su aplicación.

Cubriremos cómo puede usar el subproceso múltiple para salir de este aprieto en un capítulo posterior.



Ejecutarlo! Aparecerá la ventana, ahora haga clic en el botón de error y debería aparecer una ventana modal. Puede salir haciendo clic en la [x].

Como nuestra primera ventana, esto no es muy interesante. Arreglemos eso agregando un título de diálogo y un conjunto de botones Aceptar y Cancelar para permitir al usuario aceptar o rechazar el modal.

Para personalizar el QDialog podemos subclásificarlo, de nuevo tú *pueden* personalice el diálogo sin subclases, pero es mejor si lo hace.

```
1  clase Diálogo personalizado(QDialog):
2
3      def __init__(self, *argumentos, **kwargs):
4          super(Diálogo personalizado, self).__init__(*argumentos, *kwargs)
5
6          self.setWindowTitle ("¡HOLA!")
7
8          QBtn = QDialogButtonBox.OK | QDialogButtonBox.Cancelar
9
10         self.buttonBox = QDialogButtonBox(QBtn)
11         self.buttonBox.accepted.connect(self.accept)
12         self.buttonBox.rejected.connect(self.reject)
13
14         self.diseño = QVBoxLayout()
15         self.diseño.addWidget(self.buttonBox)
16
17
18
19  clase Ventana principal(QMainWindow):
20
21
22      # def __init__ etc.
23      #... no se muestra para mayor claridad
24
25
26      def onMyToolBarButtonClick(self, s):
27          impresión("hacer clic", s)
28
29
30          dlg = CustomDialog(self)
31          dlg.exec_()
32          impresión("Éxito!")
33
34          demás:
35              impresión("Cancelar!")
```

En el código anterior, primero creamos nuestra subclase de QDialog que hemos llamado Diálogo personalizado. En cuanto a QMainWindow lo personalizamos dentro del __en eso__ bloque para garantizar que nuestras personalizaciones se crean a medida que se crea el objeto. Primero establecemos un título para el QDialog utilizando .setWindowTitle (), exactamente lo mismo que hicimos para nuestra ventana principal.

El siguiente bloque de código está relacionado con la creación y visualización de los botones de diálogo. Probablemente esto sea un poco más complicado de lo que esperaba. Sin embargo, esto se debe a la flexibilidad de Qt para manejar el posicionamiento de los botones de diálogo en diferentes plataformas.



Puede optar por ignorar esto y utilizar un estándar QPushButton en un diseño, pero el enfoque descrito aquí garantiza que su diálogo respete los estándares del escritorio del host (Ok, a la izquierda frente a la derecha, por ejemplo). Romper estas expectativas puede ser increíblemente molesto para sus usuarios, por lo que no lo recomendaría.

El primer paso para crear un cuadro de diálogo de botones es definir los botones que se quieren mostrar, utilizando los atributos del espacio de nombres de QDialogButtonBox. Construir una línea de varios botones es simple o combinarlos con un solo (|). La lista completa de botones disponibles se encuentra a continuación:

Tipos de botones

QDialogButtonBox.Ok
QDialogButtonBox.Open
QDialogButtonBox.Save
QDialogButtonBox.Cancel
QDialogButtonBox.Close
QDialogButtonBox.Discard
QDialogButtonBox.Apply
QDialogButtonBox.Reset
QDialogButtonBox.RestoreDefaults
QDialogButtonBox.Help
QDialogButtonBox.SaveAll
QDialogButtonBox.Yes
QDialogButtonBox.YesToAll
QDialogButtonBox.No
QDialogButtonBox.NoToAll |
QDialogButtonBox.Abort

```
QDialogButtonBox.Retry  
QDialogButtonBox.Ignore  
QDialogButtonBox.NoButton
```

Estos deberían ser suficientes para crear cualquier cuadro de diálogo que se te ocurra. Por ejemplo, para mostrar un botón Aceptar y Cancelar usamos:

```
1 botones = QDialogButtonBox.OK | QDialogButtonBox.Cancelar
```

La variable botones ahora contiene un indicador de máscara de bits que representa esos dos botones. A continuación, debemos crear el QDialogButtonBox instancia para mantener los botones. La bandera para que se muestren los botones se pasa como primer parámetro.

Para que los botones tengan un efecto visual, debe conectarlos correctamente. QDialogButtonBox señales a las ranuras en el diálogo. En nuestro caso hemos conectado el.aceptado y .rechazado señales del QDialogButtonBox a los manipuladores para .aceptar() y .rechazar() en nuestra subclase de QDialog.

Por último, para hacer QDialogButtonBox aparece en nuestro cuadro de diálogo, debemos agregarlo al diseño del cuadro de diálogo. Entonces, en cuanto a la ventana principal, creamos un diseño y agregamos nuestro QDialogButtonBox a eso QDialogButtonBox es un widget), y luego establezca ese diseño en nuestro diálogo.



Ejecutarlo! Haga clic para iniciar el diálogo y verá un cuadro de diálogo con botones.

¡Felicitaciones! Ha creado su primer cuadro de diálogo. Por supuesto, puede continuar agregando cualquier otro contenido al cuadro de diálogo que desee. Simplemente insértelo en el diseño como de costumbre.

Creador de Qt

Hasta ahora hemos estado creando aplicaciones usando código Python. Esto está bien para aplicaciones simples, pero a medida que sus aplicaciones se hacen más grandes o las interfaces se vuelven más complicadas, puede resultar un poco engorroso definir todos los elementos mediante programación de esta manera. La buena noticia es que Qt viene con un editor gráfico, Qt Creator, que contiene un editor de interfaz de usuario de arrastrar y soltar.

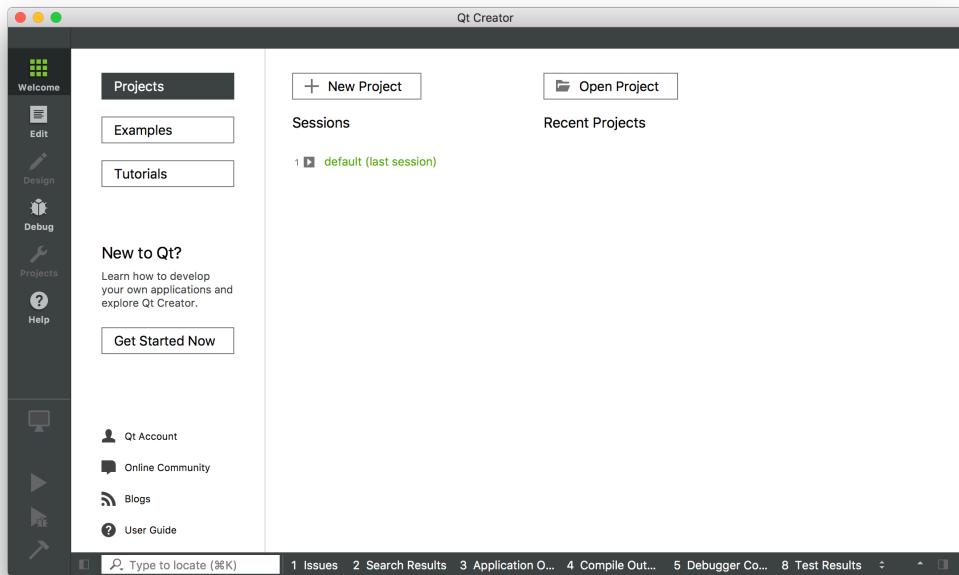
En este capítulo, cubriremos los conceptos básicos de la creación de interfaces de usuario con Qt Creator. Los principios, diseños y widgets son idénticos, por lo que puede aplicar todo lo que ya ha aprendido. También necesitará su conocimiento de la API de Python para conectar su aplicación lógica más tarde.



Puede descargarlo del sitio web de Qt. Simplemente vaya a <https://www.qt.io/download> y descargue el paquete Qt. Puede optar por instalar solo Creator durante la instalación.

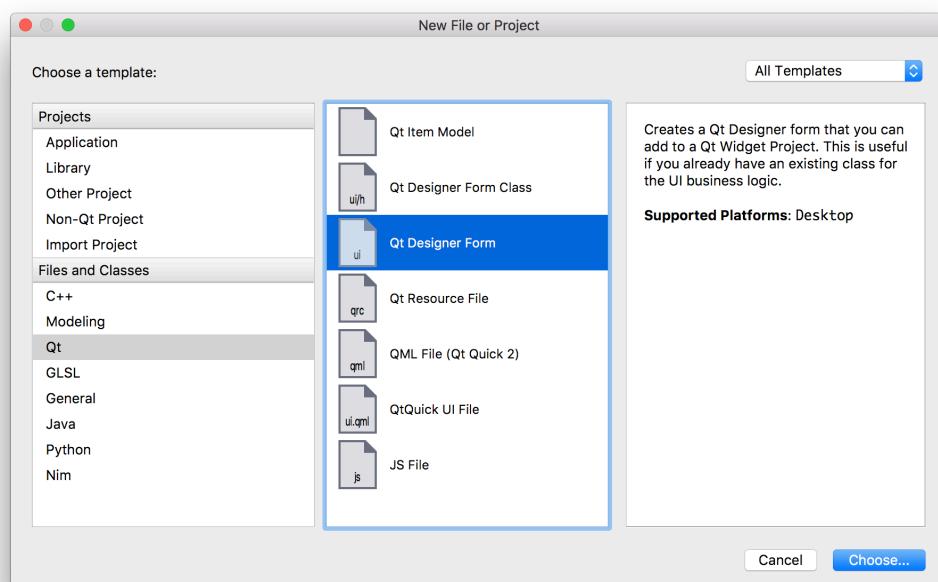
Crear un archivo .ui

Abra Qt Creator y se le presentará la ventana principal. El diseñador está disponible a través del tabón del lado izquierdo. Sin embargo, para activar esto, primero debe comenzar a crear un.ui expediente.



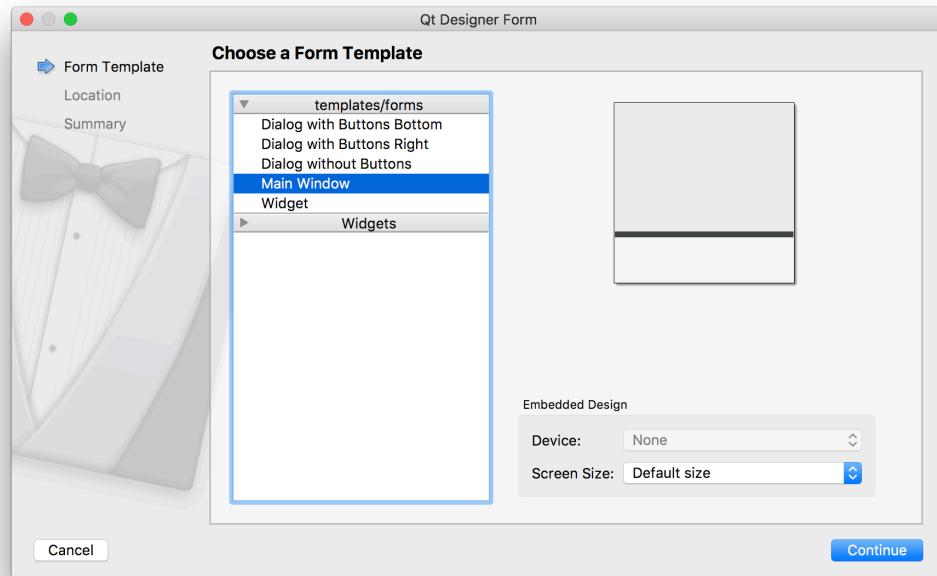
La interfaz de Qt Creator, con la sección de Diseño que se muestra a la izquierda.

Para crear un .ui archivo, vaya a Archivo -> Nuevo archivo o proyecto ... En la ventana que aparece, seleccione *Qt* debajo *Archivos y clases* a la izquierda, luego seleccione *Formulario de Qt Designer* A la derecha. Notarás que el ícono tiene "ui", que muestra el tipo de archivo que estás creando.



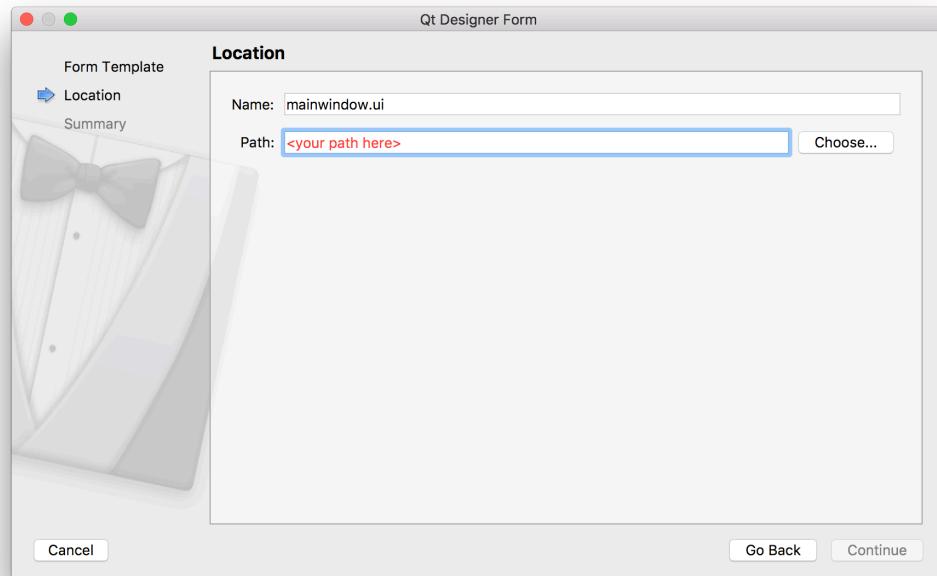
Cree un nuevo archivo Qt .ui.

En el siguiente paso, se le preguntará qué tipo de widget desea crear. Si está iniciando una aplicación, *Ventana principal* es la elección correcta. Sin embargo, también puede crear.ui archivos para cuadros de diálogo, formularios y widgets compuestos personalizados.



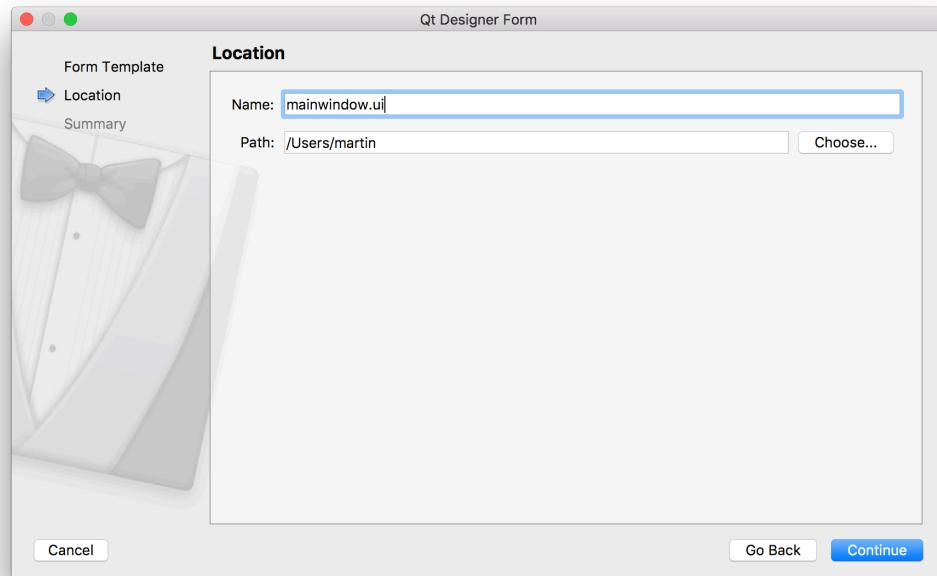
Seleccione el tipo de widget que desea crear; para la mayoría de las aplicaciones, será la ventana principal.

A continuación, elija un nombre de archivo y guarde la carpeta para su archivo. Guarda tu.ui archivo con el mismo nombre que la clase que creará, solo para simplificar los comandos posteriores.



Elija guardar el nombre y la carpeta de su archivo.

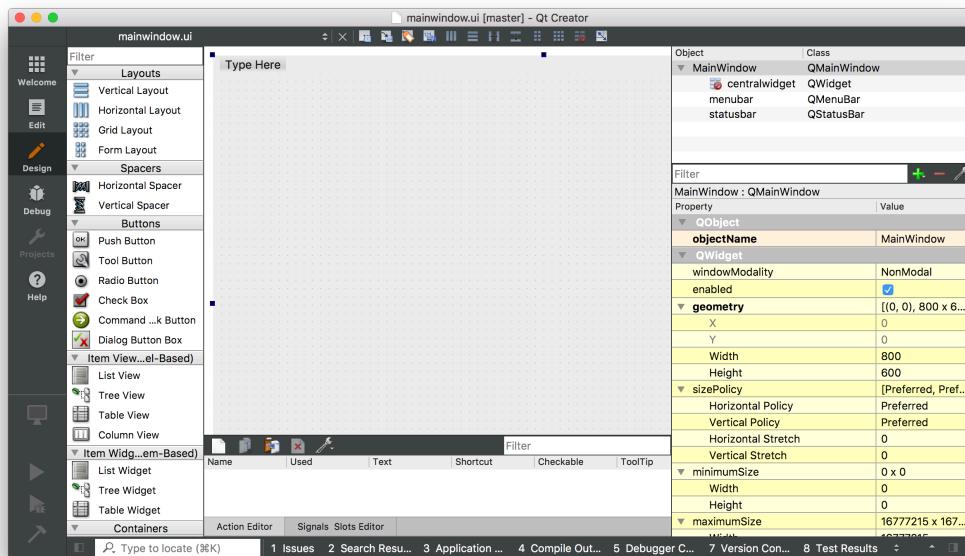
Finalmente, puede optar por agregar el archivo a su sistema de control de versiones si está usando uno. No dude en omitir este paso, no afecta su interfaz de usuario.



Opcionalmente, agregue el archivo a su control de versiones, por ejemplo, Git.

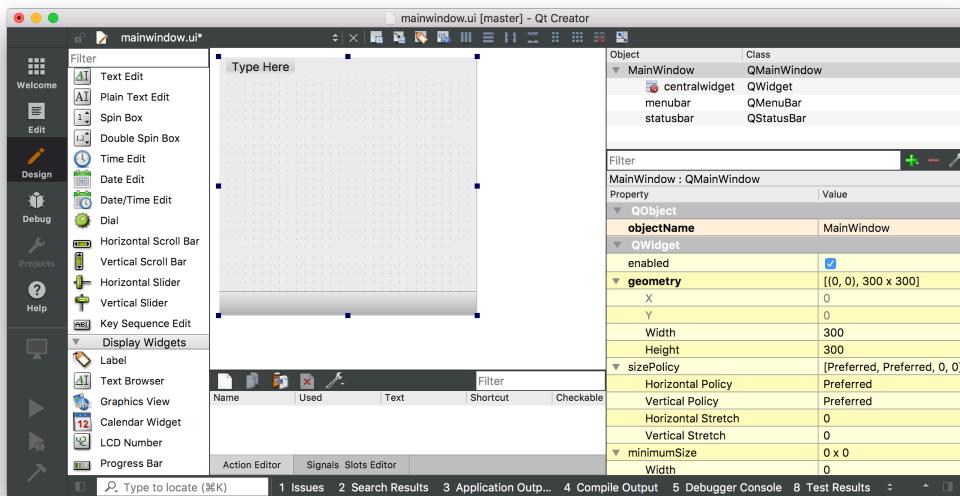
Disposición de su ventana principal

Se le presentará con su ventana principal recién creada en el diseñador de interfaz de usuario. Para empezar, no hay mucho que ver, solo un área de trabajo gris que representa la ventana, junto con el comienzo de una barra de menú de ventana.



La vista inicial de la ventana principal creada.

Puede cambiar el tamaño de la ventana haciendo clic en la ventana y arrastrando los tiradores azules en cada esquina.

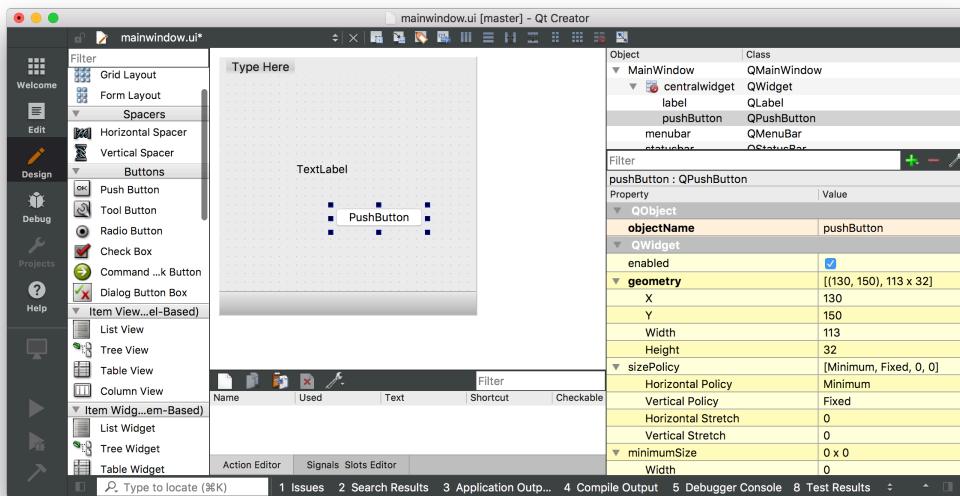


La vista inicial de la ventana principal creada.

El primer paso para crear una aplicación es agregar algunos widgets a su ventana. En nuestras primeras aplicaciones aprendimos que configurar el widget central para un QMainWindow necesitamos usar `.setCentralWidget()`. También vimos que para agregar varios widgets con un diseño, necesitamos un intermediario QWidget para aplicar el diseño, en lugar de agregar el diseño a la ventana directamente.

Qt Creator se encarga de esto automáticamente, aunque no es particularmente obvio al respecto.

Para agregar varios widgets a la ventana principal con un diseño, primero arrastre sus widgets a la QMainWindow. Aquí estamos arrastrando 3 etiquetas. No importa dónde los dejes caer.

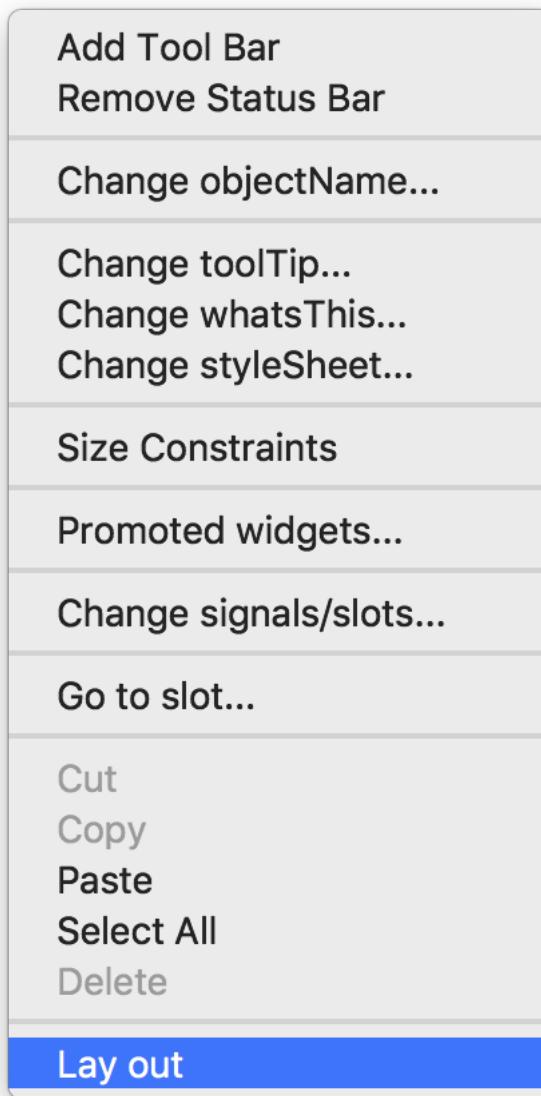


Ventana principal con 1 etiquetas y 1 botón agregado.

Hemos creado 2 widgets arrastrándolos a la ventana, los convertimos en hijos de esa ventana. Ahora podemos aplicar un diseño.

Encuentra el QMainWindow en el panel de la derecha (debería estar justo en la parte superior). Debajo ves *widget central* que representa el widget central de la ventana. El ícono del widget central muestra el diseño actual aplicado. Inicialmente tiene un círculo rojo que lo atraviesa, lo que muestra que no hay ningún diseño activo.

Haga clic derecho en el QMainWindow objeto y busque 'Diseño' en el menú desplegable resultante.



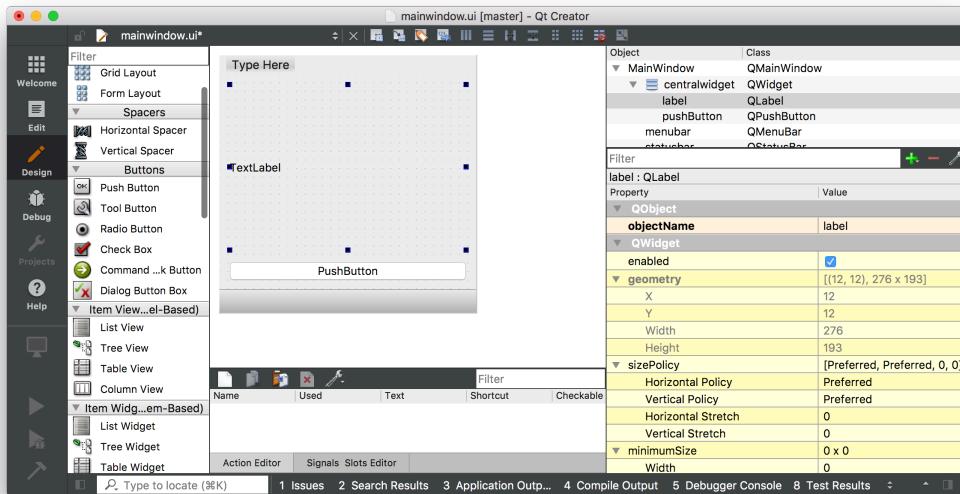
Haga clic derecho en la ventana principal y elija diseño.

A continuación, verá una lista de diseños que puede aplicar a la ventana. Seleccione *Disposición horizontal* y el diseño se aplicará al widget.

- Adjust Size
- Lay Out Horizontally**
- Lay Out Vertically
- Lay Out Horizontally in Splitter
- Lay Out Vertically in Splitter
- Lay Out in a Grid
- Lay Out in a Form Layout
- Break Layout
- Simplify Grid Layout

Seleccione el diseño para aplicar a la ventana principal.

El diseño seleccionado se aplica al *widget central* de El QMainWindow y los widgets se agregan al diseño, que se distribuyen en función del diseño seleccionado. Tenga en cuenta que en Qt Creator puede arrastrar y reordenar los widgets dentro del diseño, o seleccionar un diseño diferente, como desee. Esto hace que sea especialmente agradable crear prototipos y probar cosas.



Diseño vertical aplicado a los widgets en la ventana principal.

Usando su archivo .ui generado

Hemos creado una interfaz de usuario muy simple. El siguiente paso es introducirlo en Python y usarlo para construir una aplicación que funcione.

Primero guarda tu .ui archivo: de forma predeterminada, se guardará en la ubicación que eligió al crearlo, aunque puede elegir otra ubicación si lo desea.

Los .ui archivos están en formato XML. Para usar nuestra interfaz de usuario de Python, tenemos dos métodos alternativos disponibles:

1. cargue en una clase usando el `.loadUI()` método
2. conviéntalo a Python usando la herramienta `pyuic5`.

Estos dos enfoques se tratan a continuación. Personalmente, prefiero convertir la interfaz de usuario a un archivo de Python para mantener las cosas similares desde el punto de vista de la programación y el empaquetado.

Cargando el archivo .ui directamente

Cargar .ui archivos podemos usar el `uic` módulo incluido con PyQt5, específicamente el `uic.loadUI()` método. Esto toma el nombre de un archivo de IU y lo carga creando un objeto PyQt5 completamente funcional.

```
1 importar sys
2 de PyQt5 importar QtWidgets, uic
3
4 aplicación = QtWidgets.QApplication(sys.argv)
5
6 ventana = uic.loadUi ("mainwindow.ui")
7 ventana.show()
8 aplicación.ejecutivo()
9 ...
10
11 T> Como el `uid.loadUI ()` el método convierte una instancia objeto no puedes adjuntar cust \ om `__init __ ()` código. Sin embargo, puede manejar esto a través de una función de configuración personalizada.
12
13
14
15 """pitón
diálogos
importar sys
17 de PyQt5 importar QtWidgets, uic
18
19 def mainwindow_setup(w):
20     w.setWindowTitle ("Ventana principal Título")
21
22 aplicación = QtWidgets.QApplication(sys.argv)
23
24 ventana = uic.loadUi ("mainwindow.ui")
25 ventana mainwindow_setup
26 (ventana).show()
27 aplicación.ejecutivo()
```

Conversión de su archivo .ui a Python

Para generar un archivo de salida de Pythonpyuic5 desde la línea de comando, pasando el.ui archivo y el archivo de destino para la salida, con un -o parámetro. Lo siguiente generará un archivo Python llamadoMainWindow.py que contiene nuestra interfaz de usuario creada.

1 pyuic5 mainwindow.ui -o MainWindow.py



Si está utilizando PyQt4, la herramienta se llama pyuic4, pero por lo demás es completamente idéntico.

Puede abrir el resultado MainWindow.py archivo en un editor para echar un vistazo, aunque debe *no* editar este archivo. El poder de usar Qt Creator es poder editar, ajustar y actualizar su aplicación mientras desarrolla. Cualquier cambio realizado en este archivo se perderá cuando lo actualice. Sin embargo, *tu puedes* anule y modifique lo que quiera cuando importe y utilice el archivo en sus aplicaciones.

La importación del archivo Python resultante funciona como cualquier otro. Puede importar su clase de la siguiente manera. Lospyuic5 la herramienta agrega ui_ al nombre del objeto definido en *Creador de Qty* es este objeto el que desea importar.

1 **de Ventana principal importar** Ui_MainWindow

Para crear la ventana principal en su aplicación, cree una clase como de costumbre pero subclasicando de ambos QMainWindow y tu importado Ui_MainWindow clase. Finalmente, llamaself.setupUi (self) desde dentro del _en eso_ para activar la configuración de la interfaz.

```
1 clase Ventana principal(QMainWindow, Ui_MainWindow):
2     def __en eso__(uno mismo, *argumentos, **kwargs):
3         súper(Ventana principal, uno mismo).__en eso__(*argumentos, uno      * *kwargs)
4         mismo.setupUi (uno mismo)
```

Eso es todo. Su ventana ahora está completamente configurada.

Agregar lógica de aplicación

Puede interactuar con los widgets creados a través de Qt Creator tal como lo haría con los creados con código. Para hacer las cosas más simplesuic agrega todos los childwidgets al objeto de la ventana por su nombre de identificación.

Señales extendidas

Anteriormente, cubrimos los conceptos básicos de qué son las señales Qt y cómo puede usarlas para hacer que su aplicación responda a acciones y otras ocurrencias. Sin embargo, esto simplemente raspa la superficie de lo que puede lograr con el sistema de señal / ranura Qt.

En este capítulo veremos las formas en que puede ampliar y modificar el comportamiento de la señal desde Python y cómo puede crear señales personalizadas usted mismo.

Modificación de datos de señal

A medida que utilice señales con más frecuencia, a menudo encontrará que desea poder personalizar los datos que se envían con ellas. Desafortunadamente, no hay forma de hacer esto en Qt directamente, pero podemos aprovechar algunas características de Python para que nos funcione.

Para empezar, veremos cómo enviar *menos* datos.

Imagina que tenemos una función que acepta dos parámetros, con valores predeterminados. Sin embargo, ninguno de estos es un factor importante..windowTitleChanged señal a esta función?

Al usar una función contenedora (o una lambda) podemos aceptar los datos de la señal, descartarlos y luego llamar a nuestra ranura de destino. Entonces, por ejemplo, para descartar datos de una señal que emite un solo valor, podríamos usar la siguiente construcción.

```
1 def wrapper_function(X):
2     función_real () # Para llamar al objetivo, descartando x
```

los wrapper_function acepta el x valor, pero no lo pasa al llamar función_real. También podemos escribir esto usando una lambda de la siguiente manera:

```
1 lambda x: fn()
```

Al igual que la función, la lambda acepta un solo parámetro x, luego lo descarta llamando al *verdadero* objetivo fn sin parámetros:



No importa si usa una función normal o lambda (función anónima) para estos. Sin embargo, tiendo a usar la sintaxis lambda porque hace un código más ordenado.

Si queremos enviar *más* data podemos usar una construcción similar, pero en lugar de descartar un parámetro agregamos otro. Por ejemplo:

```
1 def wrapper_function(X)
2     función_real(x, algunos_más_datos)
```

O de nuevo, con lambda sintaxis:

```
1 lambda x: fn(x, algunos_más_datos)
```

Sin embargo, hay un problema que hay que tener en cuenta. Si envuelve una serie de señales a su vez y usa un bucle, debe conocer el comportamiento de alcance de Python para su variable de bucle y la función envuelta.

Aquí usaremos un diseño para crear una lista de widgets (no se preocupe por los diseños todavía, eso se explicará más adelante). Copie el siguiente código en un archivo y ejecútelo con Python.

```
1 de PyQt5.QtWidgets importar *de
2 PyQt5.QtCore importar *de
3 PyQt5.QtGui importar *
4
5
6 clase Ventana principal(QMainWindow):
7
8     def __en eso__(uno mismo, *argumentos, **kwargs):
9         súper(Ventana principal, uno mismo).__en eso__(*argumentos, *kwargs)
10
11         uno mismo.setWindowTitle ("Mi increíble aplicación")
12
13     diseño = QVBoxLayout ()
14
15     por norte en distancia(10):
16         btn = QPushButton (str(norte))
17         btn.presionado.conectar( lambda: uno mismo.my_custom_fn (n))
18
19         diseño.addWidget (btn)
20
21     widget = QWidget ()
22     widget.setLayout (diseño)
23
24     uno mismo.setCentralWidget (widget)
25
26     def my_custom_fn(uno mismo, n):
27         impresión("Botón %D se hizo clic " % norte)
28
29
30     aplicación = QApplication ([])
31
32     ventana = Ventana principal()
33     ventana.show()
34     aplicación.exec_()
```

Notará que a medida que iteramos para agregar los widgets, redirigimos usando una lambda a nuestra clickeado función, pasando la variable de bucle. La expectativa es que al hacer clic en cada botón se imprima un mensaje junto con el número del botón en el

consola. Intentalo.

¿Notó que al hacer clic en todos los widgets se obtiene el mismo resultado?

```
1 python3 señales_lambda.botón py
2 9 se hizo clic en el botón 9 se hizo
3 clic en el botón 9 se hizo clic en el
4 botón 9 se hizo clic
5
```

¿Qué pasa? Las reglas de alcance de variables de Python significan que cuando usamos la variable de bucle dentro de la lambda, esta es *el mismo objeto* como variable de bucle. Cada método envuelto contendrá una referencia a la misma variable, que, una vez que se complete el ciclo, contendrá el mismo valor final del ciclo. Cada llamada a cada contenedor enviará el mismo valor.

Para evitar esto, necesitamos pasar los datos adicionales como un parámetro con nombre al lambda o función de envoltura. Esto crea un nuevo objeto, único para ese nuevo espacio de nombres, que contiene el valor del bucle en el momento de su creación.

```
1 lambda x, datos=datos: fn (x, datos)
```

Entonces, ¡ahora debería poder pasar casi cualquier cosa a cualquier función usando señales! El código final ahora se ve así:

```
1 clase Ventana principal(QMainWindow):
2
3     def __init__(self, *args, **kwargs):
4         super(Ventana principal, self).__init__(*args, **kwargs)
5
6         self.setWindowTitle ("Mi increíble aplicación")
7
8         # QBoxLayout es un diseño de apilamiento horizontal con nuevos widgets
9         # agregado a la derecha de los widgets anteriores.
10        diseño = QVBoxLayout ()
11
12        por norte en distancia(10):
13            # Cree un pulsador etiquetado con el número de bucle 0-9
14            btn = QPushButton(str(norte))
15
16            # SEÑAL: La señal presionada se dispara cada vez que se presiona el botón.
17            # Conectamos esto a self.my_custom_fn a través de una lambda para pasar
18            # datos adicionales.
19            # IMPORTANTE: Debe pasar los datos adicionales como un nombre
20            # parámetro en la lambda para crear un nuevo espacio de nombres. De lo contrario
21            # el valor de n estará ligado al valor final en el parente
22            # para bucle (siempre 9).
23            btn.pressed.connect(lambda norte=norte: self.my_custom_fn (n))
24
25            # Agregue el botón al diseño. Irá a la derecha de forma predeterminada.
26            diseño.addWidget(btn)
27
28        # Cree un widget vacío para contener el diseño que contiene nuestros botones.
29        widget = QWidget ()
30
31        # Establezca el diseño que contiene nuestros botones en el widget en blanco. Nosotros sólo
32        # Necesito hacer esto aquí porque no podemos establecer un diseño en una QMainWindow.
33        # Entonces, en su lugar, estamos configurando un diseño en un widget y luego agregando eso
34        # widget a la ventana (!)widget.
35        widget.setLayout (diseño)
36
37        self.setCentralWidget (widget)
38
39        # SLOT: esta función recibirá el valor único pasado de la señal
```

```
40     def my_custom_fn(uno mismo, n):  
41         impresión("Botón %D se hizo clic " % norte)
```

Señales personalizadas

El último bit de señales que vamos a cubrir son las señales personalizadas. Estos le permiten usar el bucle de eventos Qt para enviar datos alrededor de su aplicación. Es una excelente manera de mantener su aplicación modular y receptiva.

Puede definir sus propias señales utilizando el `pyqtSignal` método proporcionado por PyQt5. Las señales deben definirse como atributos de la clase, pasando el tipo que se enviará con las señales al crearla. Puede elegir cualquier nombre de variable Python válido para el nombre de la señal.

```
1 def Ventana_principal(QMainWindow):
2     mensaje = pyqtSignal(str)
3     valor = pyqtSignal(En t)
```

Estas señales se pueden utilizar normalmente:

```
1 ventana.valor.emitir(23) # Señal en otro objeto.uno mismo.mensaje.
2 emitir("mi mensaje") # Señal en uno mismo.
```

Puede crear sus propias señales en cualquier clase que sea una subclase de `QObject`. Eso incluye todos los widgets, incluida la ventana principal, los cuadros de diálogo, etc.

Puede enviar cualquier tipo de Python, incluidos varios tipos y tipos compuestos (por ejemplo, diccionarios, listas).

```
1 def Mi_clase(QObject):
2     valor clave = pyqtSignal(dictar)
3     datos = pyqtSignal(tupla)
```



Si define su señal como `pyqtSignal(objeto)` podrá enviar cualquier tipo de Python. Pero esto no es recomendable, ya que los controladores de recepción deberán tratar con todos los tipos.

Gráficos de mapa de bits y QPainter

El primer paso para crear widgets personalizados en PyQt5 es comprender las operaciones gráficas de mapa de bits (basadas en píxeles). Todos los widgets estándar se dibujan a sí mismos como mapas de bits en un "lienzo" rectangular que forma la forma del widget. Una vez que comprenda cómo funciona, ¡puede dibujar cualquier widget que desee!



Los mapas de bits son cuadrículas rectangulares de *píxeles*, donde cada píxel (y su color) está representado por una serie de "bits". Son distintos de los gráficos vectoriales, donde la imagen se almacena como una serie de líneas (*o vector*) instrucciones de dibujo que se repiten desde la imagen. Si está viendo gráficos vectoriales en su pantalla, se están *rasterizado* (es decir, convertida en una imagen de mapa de bits) para que se muestre como píxeles en la pantalla.

En este tutorial veremos QPainter, La API de Qt para realizar operaciones gráficas de mapa de bits y la base para dibujar sus propios widgets. Pasaremos por algunas operaciones básicas de dibujo y finalmente lo uniremos todo para crear nuestra propia pequeña aplicación Paint.

QPainter

Las operaciones de dibujo de mapa de bits en Qt se manejan a través del QPainter clase. Esta es una interfaz genérica que se puede utilizar para dibujar en variossuperficies incluyendo, por ejemplo, QPixmap. En este capítulo veremos el QPainter métodos de dibujo, primero usando operaciones primitivas en un QPixmap Surface, y luego construyendo una aplicación Paint simple usando lo que hemos aprendido.

Para que esto sea fácil de demostrar, usaremos la siguiente aplicación de código auxiliar que maneja la creación de nuestro contenedor (un QLabel) creando un lienzo de mapa de píxeles, configurándolo en el contenedor y agregando el contenedor a la ventana principal.

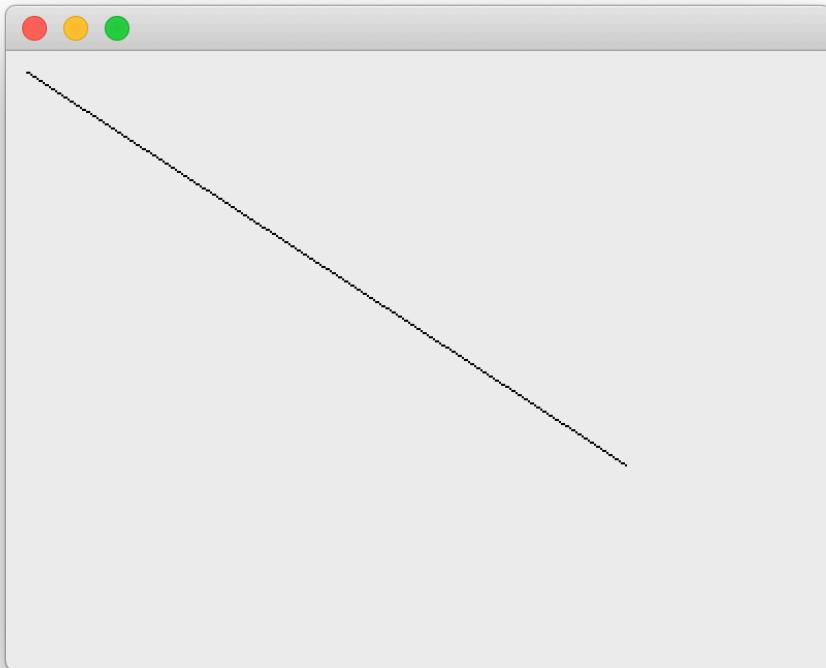
```
1 importar sys
2 de PyQt5 importar QtCore, QtGui, QtWidgets, uicde
3 PyQt5.QtCore importar Qt
4
5
6 clase Ventana principal(QtWidgets.QMainWindow):
7     def __en eso__(uno mismo):
8         súper().__en eso__()
9
10    uno mismo.etiqueta = QtWidgets.Lienzo QLabel
11    () = QtGui.QPixmap (400, 300)uno mismo.
12    etiqueta.setPixmap (lienzo)uno mismo.
13    setCentralWidget (uno mismo.etiqueta)uno
14    mismo.dibujar algo()
15
16
17    def dibujar algo(uno mismo):
18        pintor = QtGui.QPainter (uno mismo.etiqueta.pixmap ())
19        pintor.Dibujar linea(10, 10, 300, 200) pintor.fin()
20
21
22    aplicación = QtWidgets.QApplication (sys.argv)
23    ventana = Ventana principal()
24    ventana.show()
25    aplicación.exec_()

```



¿Por qué usamos QLabel ¿recurrir a? los QLabel El widget también se puede usar para mostrar imágenes, y es el widget más simple disponible para mostrar un QPixmap.

Guarde esto en un archivo y ejecútelo y debería ver lo siguiente: una sola línea negra dentro del marco de la ventana:

**Una sola línea negra en el lienzo.**

Todo el dibujo ocurre dentro del dibujar algo método - creamos un QPainter ejemplo, pasando en el lienzo (self.label pixmap ()) y luego emita un comando para dibujar una línea. Finalmente llamamos .fin () para cerrar el pintor y aplicar los cambios.



Por lo general, también necesitaría llamar .actualizar () para activar una actualización del widget, pero como estamos dibujando antes de que se muestre la ventana de la aplicación, una actualización ya se producirá automáticamente.

Dibujo de primitivas

QPainter proporciona una gran cantidad de métodos para dibujar formas y líneas en una superficie de mapa de bits (en 5.12 hay 192 QPainter métodos específicos sin eventos). La buena noticia es que la mayoría de estos son métodos sobrecargados que son simplemente formas diferentes de llamar a los mismos métodos base.

Por ejemplo, hay 5 diferentes Dibujar linea métodos, todos los cuales dibujan la misma línea, pero difieren en cómo se definen las coordenadas de qué dibujar.

Método	Descripción
Dibujar linea (const QLineF <i>y/línea</i>)	Dibuja un QLineF instancia
Dibujar linea (const QLine <i>y/línea</i>) Dibujar linea (En t <i>x1</i> , En t <i>y1</i> , En t <i>x2</i> , En t <i>y2</i>)	Dibuja una QLine instancia Dibuja una línea entre <i>x1</i> , <i>y2</i> y <i>x2</i> , <i>y2</i>
Dibujar linea (const QPoint <i>yp1</i> , const QPoint & <i>p2</i>) Dibujar linea (const QPointF <i>yp1</i> , const QPointF & <i>p2</i>)	Dibuja una línea entre QPoint 1 y QPoint 2 Dibuja una línea entre QPointF 1 y QPointF 2

Si se pregunta cuál es la diferencia entre un QLine y un QLineF , este último tiene sus coordenadas especificadas como flotador. Esto es conveniente si tiene posiciones flotantes como resultado de otros cálculos, pero por lo demás no tanto.

Ignorando las variantes F, tenemos 3 formas únicas de dibujar una línea: con un objeto de línea, con dos conjuntos de coordenadas (*x1*, *y1*), (*x2*, *y2*) o con dos QPoint objetos. Cuando descubra que unQLine en sí mismo se define como QLine (const QPoint & *p1*, const QPoint *y p2*)OQLine (int *x1*, int *y1*, int *x2*, int *y2*)ves que son todos de hecho, exactamente lo mismo. Las diferentes firmas de llamada están ahí simplemente por conveniencia.



Dadas las coordenadas *x1*, *y1*, *x2*, *y2*, los dos QPoint los objetos serían definido como QPoint (*x1*, *y1*) y QPoint (*x2*, *y2*).

Entonces, dejando de lado los duplicados, tenemos las siguientes operaciones de dibujo:drawArc , drawChord , drawConvexPolygon , dibujarEllipse , Dibujar linea , drawPath , empate , drawPoint , drawPolygon , drawPolyline , drawRect , drawRects y drawRoundedRect . Toavoidget abrumado, nos centraremos primero en las formas y líneas primitivas primero y regresaremos a las operaciones más complicadas una vez que tengamos los conceptos básicos.



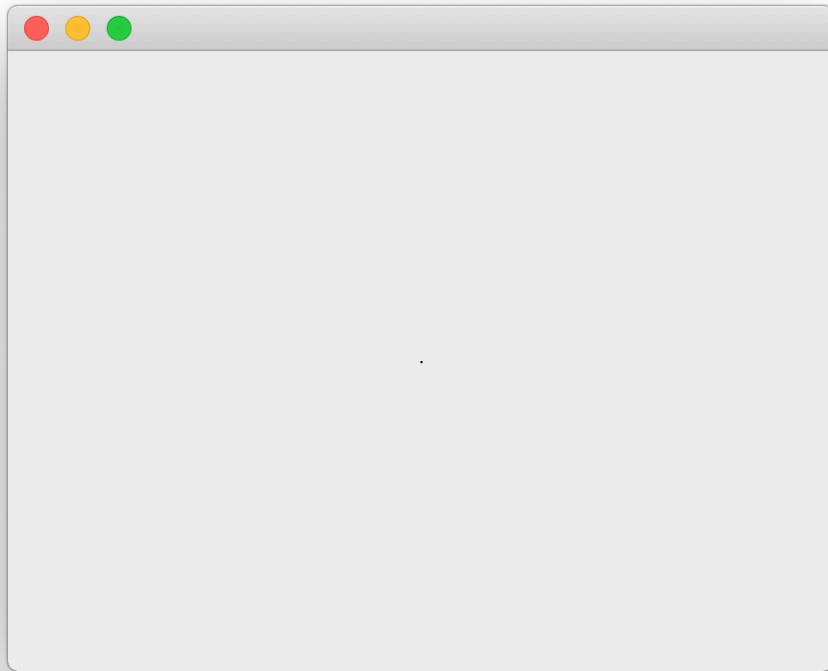
Para cada ejemplo, reemplace el dibujar algo en su aplicación auxiliar y vuelva a ejecutarlo para ver el resultado.

drawPoint

Esto dibuja un punto, o *píxel*, en un punto dado del lienzo. Cada llamada a `drawPoint` dibuja un píxel. Reemplaza tu dibujar algo código con lo siguiente.

```
1 def dibujar algo(uno mismo):
2     pintor = QtGui.QPainter (uno mismo.etiqueta.pixmap ())
3     pintor.drawPoint (200, 150) pintor.fin()
4
```

Si vuelve a ejecutar el archivo, verá una ventana, pero esta vez hay un solo punto, en negro en el medio. Probablemente necesites mover la ventana para verla.

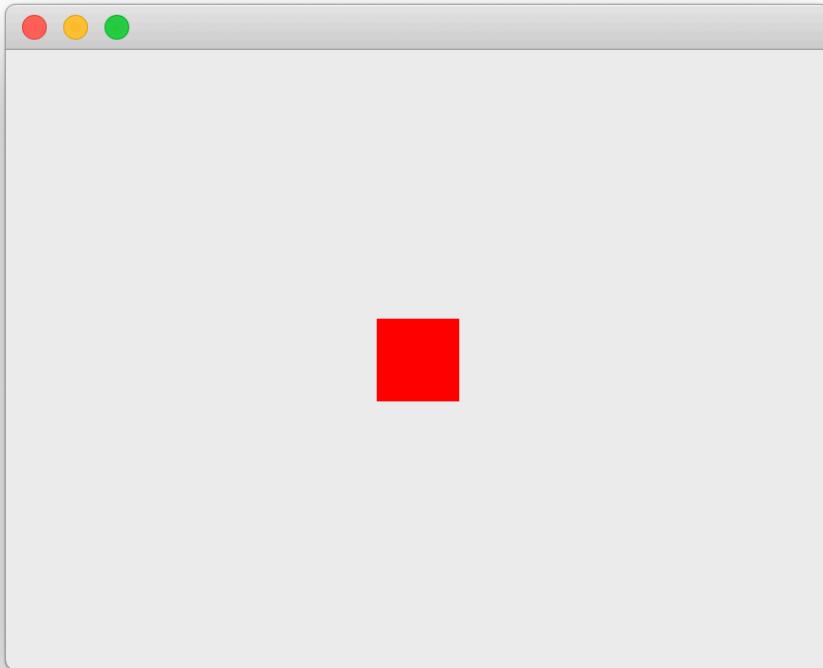


Dibujar un solo punto (píxel) con QPainter

Eso realmente no es mucho para mirar. Para hacer las cosas más interesantes, podemos cambiar el color y el tamaño del punto que estamos dibujando. En PyQt, el color y el grosor de las líneas se definen utilizando el *activo/ápiz* en el QPainter. Puede configurar esto creando un QPen instancia y aplicándola.

```
1 def dibujar algo(uno mismo):
2     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
3     ()) = QtGui.Bolígrafo QPen ().setWidth (40)
4
5     lápiz.setColor (QtGui.QColor ('rojo')) pintor.
6     setPen (bolígrafo)
7     pintor.drawPoint (200, 150) pintor.
8     fin()
```

Esto dará el siguiente resultado ligeramente más interesante.



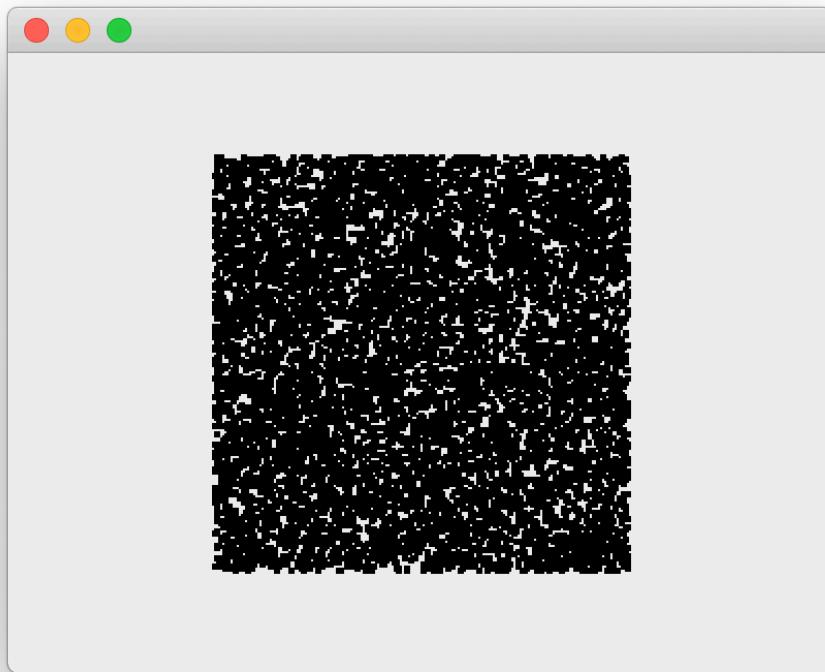
usted

son libres de realizar múltiples operaciones de dibujo con su QPainter hasta que el pintor es terminó. Dibujar en el lienzo es muy rápido: aquí estamos dibujando 10k puntos en

aleatorio.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4     ()) = QtGui.Bolígrafo QPen ().setWidth (3) pintor.setPen (bolígrafo)
5
6
7
8     por norte en distancia(10000):
9         pintor.drawPoint (
10             200+randint-100,150+ 100),      # X
11             randint-100,)           100)      # y
12
13     pintor.fin()
```

Los puntos tienen 3 píxeles de ancho y son negros (el lápiz predeterminado).

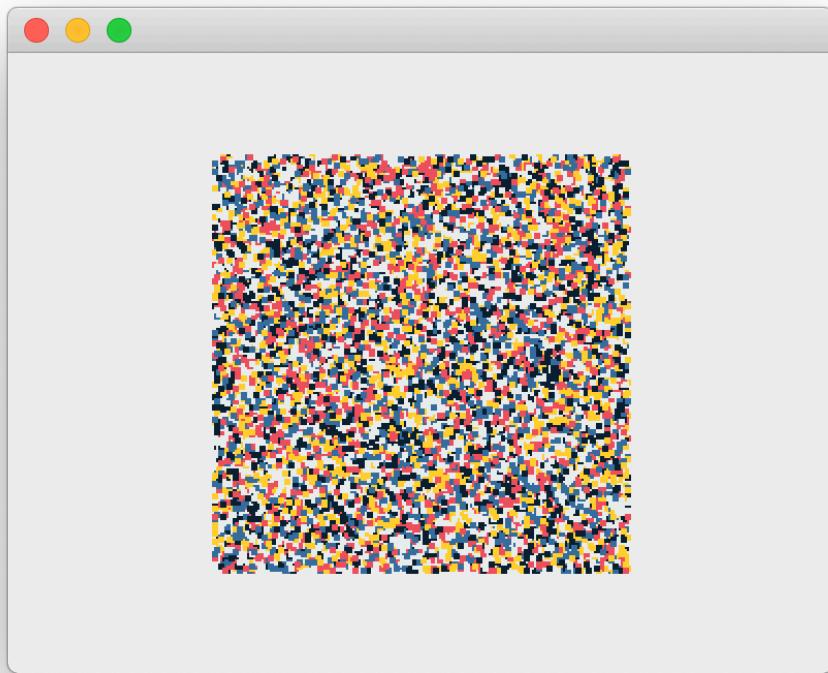


10k puntos de 3 píxeles en un lienzo

A menudo, deseará actualizar el dibujo actual del lápiz, por ejemplo, dibujar varios puntos en diferentes colores mientras se mantienen las otras características (ancho) iguales. Para hacer esto sin recrear un nuevo `QPen` instancia cada vez que pueda obtener el lápiz activo actual del `QPainter` utilizando `pen = pintor.pen()`. También puede volver a aplicar una pluma existente varias veces, cambiándola cada vez.

```
1 def dibujar algo(uno mismo):
2     de aleatorio importar randint, elección
3     colores = ['# FFD141', '# 376F9F', '# 0D1F2D', '# E9EBEF', '# EB5160']
4
5     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
6     ()= QtGui.Bolígrafo QPen ().setWidth (3) pintor.setPen (bolígrafo)
7
8
9
10    por norte en distancia(10000):
11        # pen = painter.pen () puede obtener el lápiz activo aquí lápiz.setColor
12        (QtGui.Pintor QColor (elección (colores))).setPen (bolígrafo)
13
14        pintor.drawPoint (
15            200+randint-100,150+ 100),      # X
16            randint-100,)                 100)      # y
17
18    pintor.fin()
```

Producirá la siguiente salida:



Patrón aleatorio de 3 puntos de ancho



Solo puede haber uno QPen activo en un QPainter - la pluma actual.

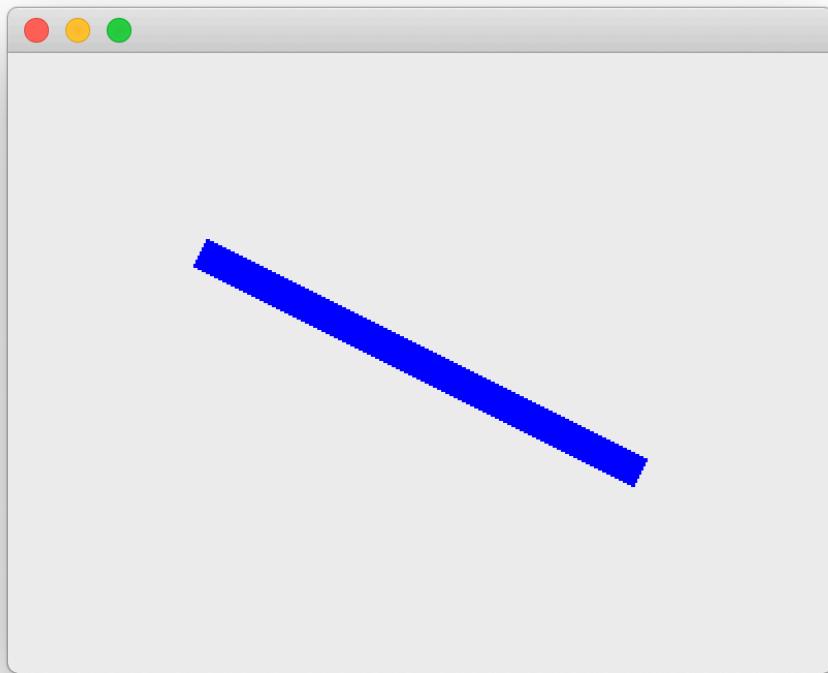
Eso es todo lo emocionante que puedes tener al dibujar puntos en una pantalla, así que pasaremos a ver algunas otras operaciones de dibujo.

Dibujar línea

Ya dibujamos una línea en el lienzo al principio para probar que todo funciona. Pero lo que no probamos fue configurar el lápiz para controlar la apariencia de la línea.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4     ()) = QtGui.Bolígrafo QPen ().setWidth (15)
5
6     lápiz.setColor (QtGui.QColor ('azul')) pintor.
7     setPen (bolígrafo)
8     pintor.Dibujar linea(
9         QtCore.QPoint (100,      100),
10        QtCore.QPoint (300,      200)
11    )
12    pintor.fin()
```

En este ejemplo también usamos QPoint para definir los dos puntos para conectar con una línea, en lugar de pasar individual x1, y1, x2, y2 parámetros: recuerde que ambos métodos son funcionalmente idénticos.



Una gruesa línea azul

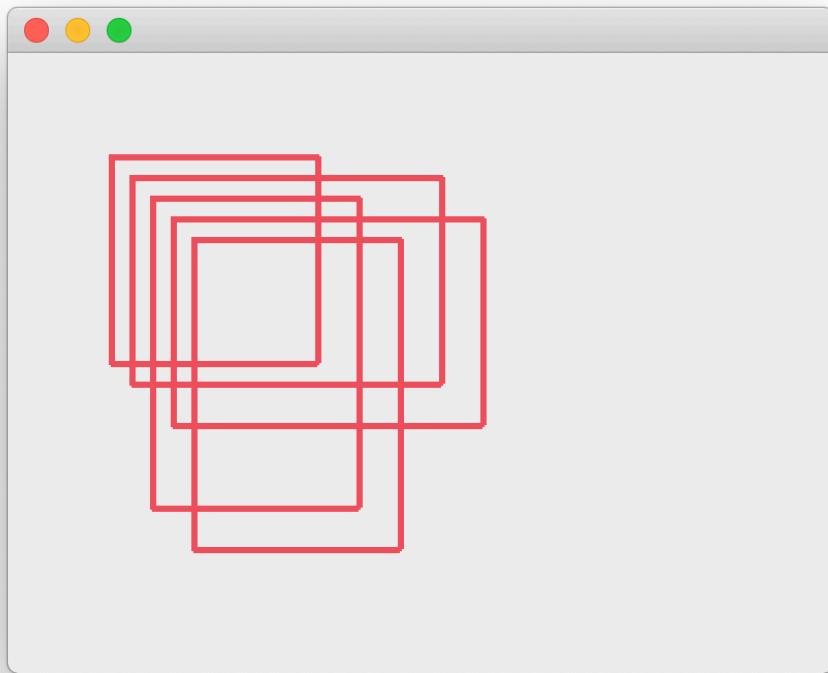
drawRect, drawRects y drawRoundedRect

Todas estas funciones dibujan rectángulos, definidos por una serie de puntos, o por QRect o QRectF instancias.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4     ()) = QtGui.Bolígrafo QPen ().setWidth (3)
5
6     lápiz.setColor (QtGui.QColor ("# EB5160")) pintor.
7     setPen (bolígrafo)
8     pintor.drawRect (50, 50, 100, 100) pintor.
9     drawRect (60, 60, 150, 100) pintor.drawRect (
10    70, 70, 100, 150) pintor.drawRect (80, 80, 150,
11    100) pintor.drawRect (90, 90, 100, 150) pintor.
12    fin()
13
```



Un cuadrado es solo un rectángulo con el mismo ancho y alto



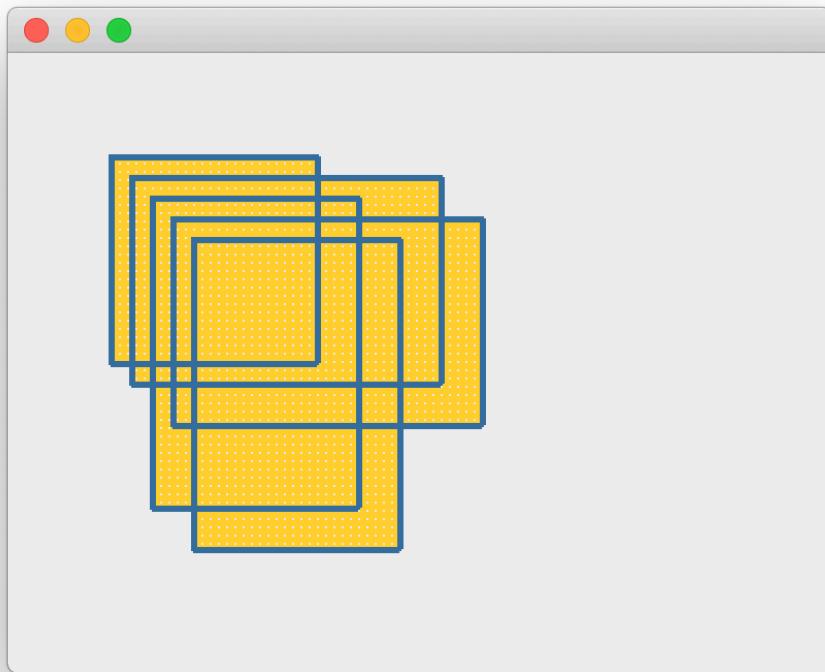
Dibujar rectángulos

También puede reemplazar las múltiples llamadas a `drawRect` con una sola llamada a `drawRects` pasando en múltiples `QRect` objetos. Esto producirá exactamente el mismo resultado.

```
1 pintor.drawRects (
2     QtCore.QRect(50, 50, 100, 100), QtCore.
3     QRect(60, 60, 150, 100), QtCore.QRect(70,
4         70, 100, 150), QtCore.QRect(80, 80, 150,
5         100), QtCore.QRect(90, 90, 100, 150),
6
7 )
```

Las formas dibujadas se pueden llenar en PyQt configurando el pintor activo actual *cepillar*, pasando en un QBrush instancia a painter.setBrush (). El siguiente ejemplo llena todos los rectángulos con un color amarillo estampado.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4     ()) = QtGui.Bolígrafo QPen ().setWidth (3)
5
6     lápiz.setColor (QtGui.QColor ("# 376F9F")) pintor.
7     setPen (bolígrafo)
8
9     cepillar = QtGui.QBrush ()
10    cepillar.setColor (QtGui.QColor ("# FFD141")) cepillar.
11    setStyle (Qt.Dense1Pattern) pintor.setBrush (cepillo)
12
13
14    pintor.drawRects (
15        QtCore.QRect (50, 50, 100, 100), QtCore.
16        QRect (60, 60, 150, 100), QtCore.QRect (70,
17            70, 100, 150), QtCore.QRect (80, 80, 150,
18            100), QtCore.QRect (90, 90, 100, 150),
19
20    )
21    pintor.fin()
```



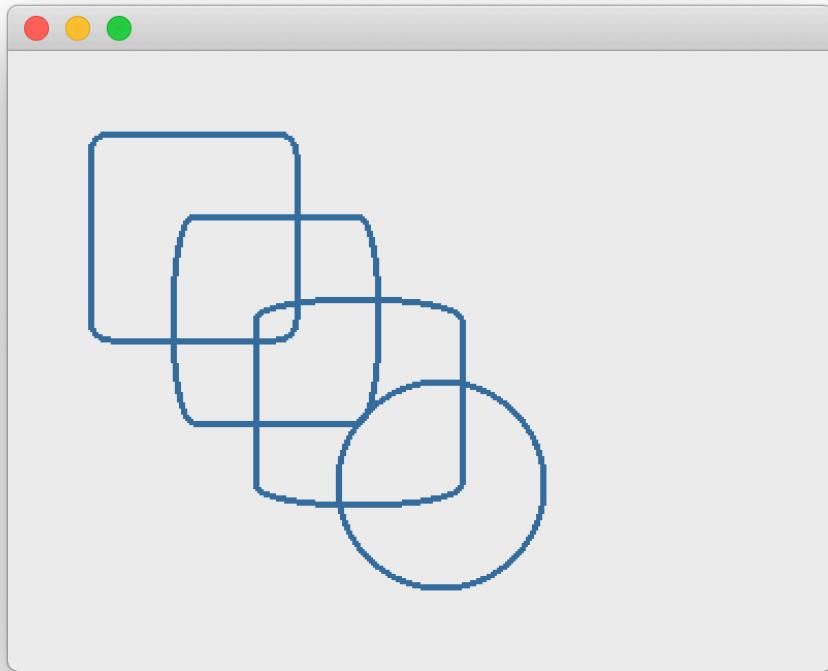
Como para el lápiz, solo hay un pincel activo en un pintor determinado, pero puede alternar entre ellos o cambiarlos mientras dibuja. Existen [varios patrones de estilo de pincel disponibles](#). Probablemente usarás Qt. De patrón sólido aunque más que cualquier otro.



usted *debe* establecer un estilo para ver cualquier relleno, ya que el valor predeterminado es `Qt.NoBrush`.

los `drawRoundedRect` Los métodos dibujan un rectángulo, pero con bordes redondeados, por lo que toman dos parámetros adicionales para `x & y radio` de las esquinas.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4     ()) = QtGui.Bolígrafo QPen ().setWidth (3)
5
6     lápiz.setColor (QtGui.QColor ("# 376F9F")) pintor.
7     setPen (bolígrafo)
8     pintor.drawRoundedRect (40, 40, 100, 100, 10, 10) pintor.
9     drawRoundedRect (80, 80, 100, 100, 10, 50) pintor.drawRoundedRect
10    (120, 120, 100, 100, 50, 10) pintor.drawRoundedRect (160, 160, 100,
11    100, 50, 50) pintor.fin()
12
```



Rectángulos redondeados



Hay un parámetro final opcional para alternar entre los radios de elipse x e y de las esquinas que se definen en términos de píxeles absolutos. `Qt.RelativeSize` (el predeterminado) o relativo al tamaño del rectángulo (pasado como un valor 0... 100). Aprobar `Qt.RelativeSize` para habilitar esto.

dibujarEllipse

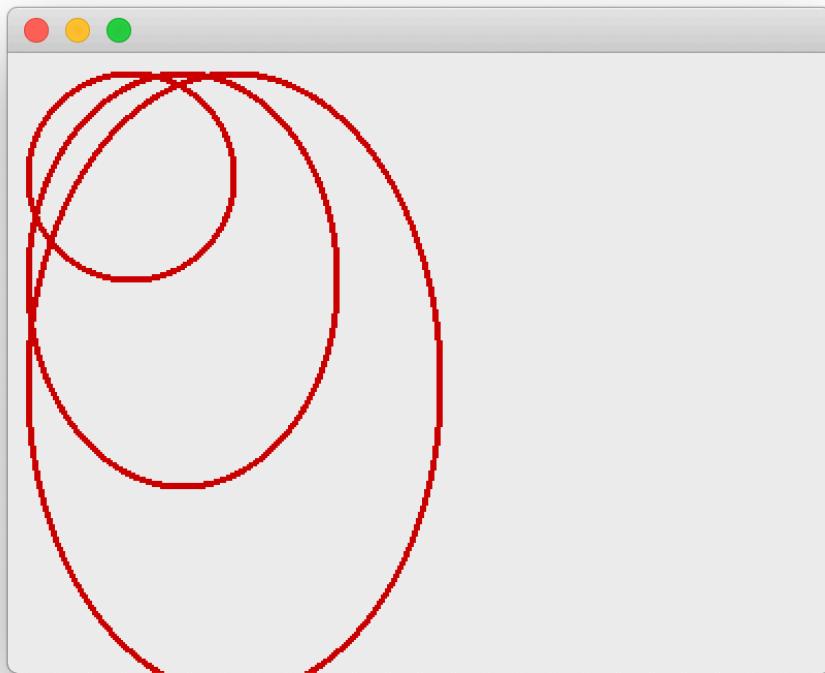
El último método primitivo que veremos ahora es `dibujarEllipse` que se puede utilizar para dibujar un *elipse* o un *círculo*.



Un círculo es solo una elipse con el mismo ancho y alto.

```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.bolígrafo pixmap
4         ()) = QtGui.Bolígrafo QPen ().setWidth (3)
5
6     lápiz.setColor (QtGui.QColor (204,0,0)) # r, g, bpintor.setPen
7     (bolígrafo)
8
9     pintor.dibujarEllipse (10, 10, 100, 100) pintor.
10    dibujarEllipse (10, 10, 150, 200) pintor.
11    dibujarEllipse (10, 10, 200, 300) pintor.fin()
12
```

En este ejemplo dibujarEllipse está tomando 4 parámetros, siendo los dos primeros la posición x & y del *arriba a la izquierda del rectángulo* en el que se dibujará la elipse, mientras que los dos últimos parámetros son el ancho y el alto de ese rectángulo respectivamente.

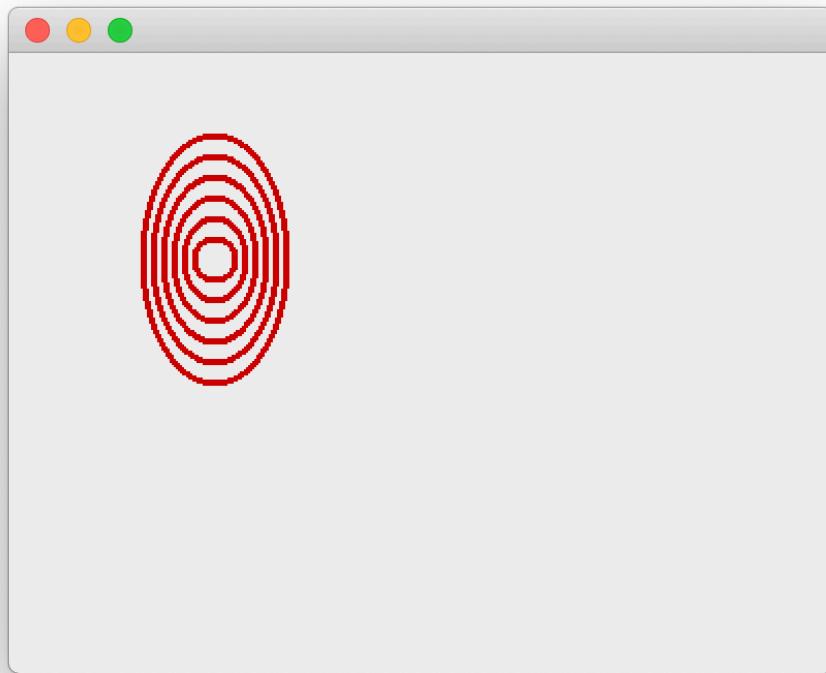


T>

Puede lograr lo mismo pasando un QRect

Hay otra firma de llamada que toma el *centro de la elipse* como primer parámetro, proporcionado como QPoint o QPointF objeto, y luego *ax* y *y radio*. El siguiente ejemplo lo muestra en acción.

```
1 pintor.drawEllipse (QtCore.QPoint (100, 100), 10, 10) pintor.drawEllipse  
2 (QtCore.QPoint (100, 100), 15, 20) pintor.drawEllipse (QtCore.QPoint (  
3 100, 100), 20, 30) pintor.drawEllipse (QtCore.QPoint (100, 100), 25, 40)  
4 pintor.drawEllipse (QtCore.QPoint (100, 100), 30, 50) pintor.drawEllipse  
5 (QtCore.QPoint (100, 100), 35, 60)  
6
```



Dibujar una elipse usando Punto y radio

Puede llenar elipses con el mismo QBrush enfoque descrito para rectángulos.

Texto

Finalmente, haremos un breve recorrido por los métodos de dibujo de texto. Para controlar la fuente actual en un QPainter tu usas setFont pasando en un QFont ejemplo. Con esto puedes controlar la familia, el peso y el tamaño (entre otras cosas) del texto que escribes. Sin embargo, el color del texto todavía se define con el lápiz actual.

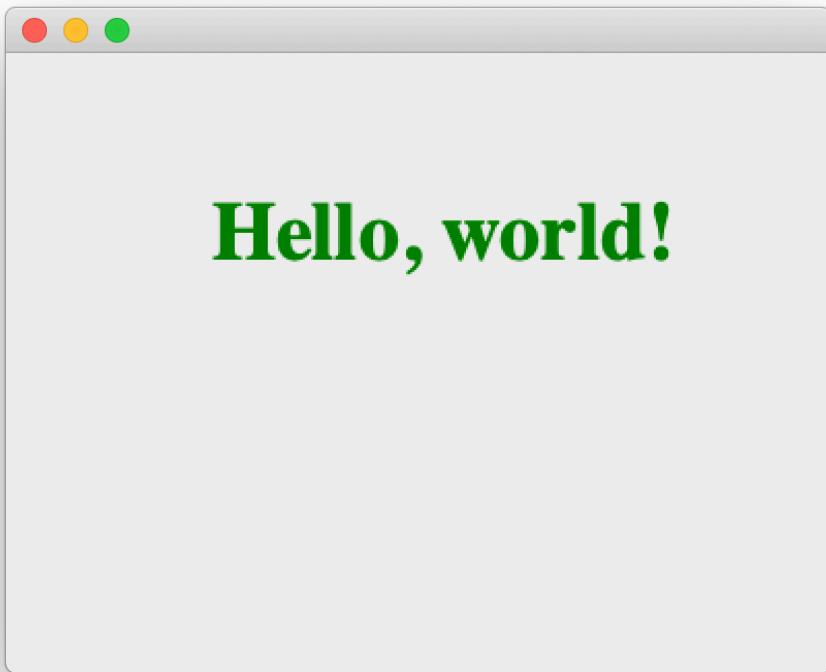
```
1 def dibujar algo(uno mismo):de
2     aleatorio importar randint
3     pintor = QtGui.QPainter (uno mismo.etiqueta.pixmap ())
4
5     lápiz = QtGui.Bolígrafo
6     QPen ().setWidth (1)
7     lápiz.setColor (QtGui.QColor ('verde')) pintor.
8     setPen (bolígrafo)
9
10    fuente = QtGui.Fuente QFont () .
11    setFamily ('Veces') fuente.
12    setBold (Cierto) fuente.
13    setPointSize (40) pintor.setFont
14    (fuente)
15
16    diecisés
17    pintor.dibujarTexto (100, 100, '¡Hola Mundo!') pintor.fin()
```



También puede especificar la ubicación con QPoint o QPointF.



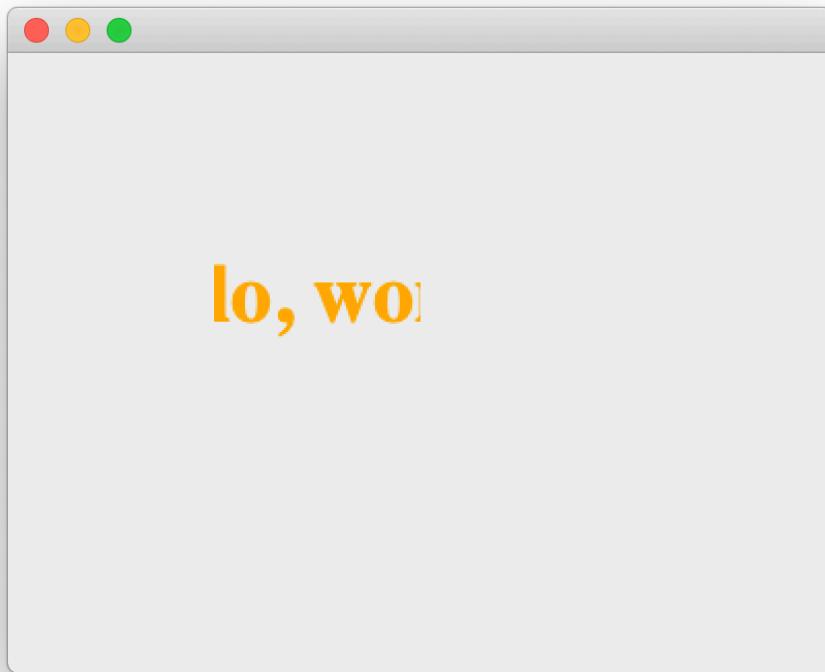
El ancho del bolígrafo no tiene ningún efecto sobre la apariencia del texto.



Ejemplo de hola mundo de texto de mapa de bits

También existen métodos para dibujar texto dentro de un área específica. Aquí los parámetros definen la posición x & y y el ancho y alto del cuadro delimitador. El texto fuera de este cuadro está recortado (oculto). El quinto parámetro *banderas* se puede utilizar para controlar la alineación del texto dentro del cuadro, entre otras cosas.

1 pintor.dibujarTexto (100, 100, 100, 100, Qt.AlignHCenter, '**Hola Mundo!**)



Cuadro delimitador recortado en `drawText`

Tiene control total sobre la visualización del texto configurando la fuente activa en el pintor a través de un QFont objeto. Revisar la [Documentación de QFont](#) para más información.

Un poco de diversión con QPainter

Se puso un poco pesado, así que tomemos un respiro y hagamos algo divertido. Hasta ahora hemos estado definiendo programáticamente las operaciones de dibujo para realizar en el QPixmap superficie. Pero podemos dibujar con la misma facilidad en respuesta a la entrada del usuario, por ejemplo, permitir que un usuario garabatee en todo el lienzo. Tomemos lo que hemos aprendido hasta ahora

y utilícelo para crear una aplicación de Paint rudimentaria.

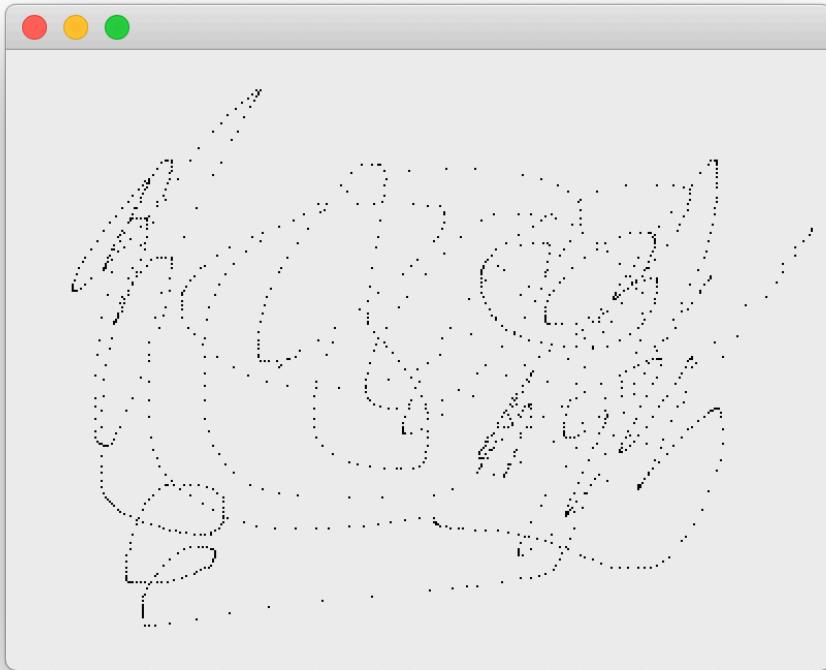
Podemos comenzar con el mismo esquema de aplicación simple, agregando un mouseMoveEvent manejador de la Ventana principal class en lugar de nuestro método de dibujo. Aquí tomaremos la posición actual del mouse del usuario y la dibujaremos en el lienzo.

```
1 importar sys
2 de PyQt5 importar QtCore, QtGui, QtWidgets, uicde
3 PyQt5.QtCore importar Qt
4
5
6 clase Ventana principal(QtWidgets.QMainWindow):
7
8     def __en eso__(uno mismo):
9         súper().__en eso__()
10
11     uno mismo.etiqueta = QtWidgets.Lienzo QLabel
12     () = QtGui.QPixmap (400, 300)uno mismo.
13     etiqueta.setPixmap (lienzo)uno mismo.
14     setCentralWidget (uno mismo.etiqueta)
15
16     def mouseMoveEvent(uno mismo, e):
17         pintor = QtGui.QPainter (uno mismo.etiqueta.pixmap ())
18         pintor.drawPoint (e.x (), e.y ()) pintor.fin()
19
20         uno mismo.actualizar()
21
22
23     aplicación = QtWidgets.QApplication (sys.argv)
24     ventana = Ventana principal()
25     ventana.show()
26     aplicación.exec_()
```



¿Por qué ningún evento de clic? Los widgets de forma predeterminada solo reciben eventos de movimiento del mouse cuando se presiona un botón del mouse, a menos que *seguimiento del mouse* esté habilitado. Esto se puede configurar usando el.setMouseTracking método - estableciendo esto en Cierto (está Falso por defecto) rastreará el mouse continuamente.

Si guarda esto y lo ejecuta, debería poder mover el mouse sobre la pantalla y hacer clic para dibujar puntos individuales. Debería verse algo como esto:



Dibujar puntos individuales de mouseMoveEvent

El problema aquí es que cuando mueve el mouse rápidamente, en realidad salta entre ubicaciones en la pantalla, en lugar de moverse suavemente de un lugar a otro. Los `mouseMoveEvent`s se dispara para cada ubicación en la que se encuentra el mouse, pero eso no es suficiente para dibujar una línea continua, a menos que mueva *muy lentamente*.

La solución a esto es dibujar *líneas* en lugar de *puntos*. En cada evento, simplemente trazamos una línea desde donde estábamos (`anteriorEx()` y `ey()`) a donde estamos ahora (`actual ex()` y `ey()`). Podemos hacer esto rastreando `last_x` y `last_y` nosotros mismos.

También necesitamos *olvidar* la última posición al soltar el mouse, o comenzaremos

dibujar desde esa ubicación nuevamente después de mover el mouse por la página, es decir, no podremos romper la línea.

```
1 importar sys
2 de PyQt5 importar QtCore, QtGui, QtWidgets, uicde
3 PyQt5.QtCore importar Qt
4
5
6 clase Ventana principal(QtWidgets.QMainWindow):
7
8     def __en eso__(uno mismo):
9         súper().__en eso__()
10
11     uno mismo.etiqueta = QtWidgets.Lienzo QLabel
12     () = QtGui.QPixmap (400, 300)uno mismo.
13     etiqueta.setPixmap (lienzo)uno mismo.
14     setCentralWidget (uno mismo.etiqueta)
15
16     uno mismo.last_x, uno mismo.last_y = Ninguno, Ninguno
17
18     def mouseMoveEvent(uno mismo, e):
19         si uno mismo.last_x es Ninguno: # Primer evento.
20             uno mismo.last_x = mi.X()
21             uno mismo.last_y = mi.y ()
22             regreso # Ignore la primera vez.
23
24         pintor = QtGui.QPainter (uno mismo.etiqueta.pixmap ()) pintor.Dibujar linea(
25             uno mismo.last_x, uno mismo.last_y, e.x (), e.y ()) pintor.fin()
26
27         uno mismo.actualizar()
28
29         # Actualice el origen para la próxima vez.uno
30         mismo.last_x = mi.X()
31         uno mismo.last_y = mi.y ()
32
33     def mouseReleaseEvent(uno mismo, mi):
34         uno mismo.last_x = Ninguno
35         uno mismo.last_y = Ninguno
```

```
36
37
38 aplicación = QtWidgets.QApplication(sys.argv)
39 ventana = Ventana principal()
40 ventana.show()
41 aplicación.exec_()
42 """
43
44 Si ejecuta esto, debería poder hacer garabatos en la pantalla como usted esperaría.
45
46
47 ![Dibujo con el mouse, usando una línea continua] (imágenes/mapa de bits-dibujar.png)
48
49 Esotodavía es un poco aburrido, así que vamoss agregue una paleta simple que nos permita cambiar el \
50 color del lápiz.
51
52 Esto requiere un poco de re-Arquitectura para asegurar la posición del mouse. es detectado \
53 precisamente. Hasta ahora nosotrosHe estado usando el `mouseMoveEvent` en el` QMainWindow`. Wh\es,
54 solo tenemos un widget en la ventana, esto está bien, siempre y cuando no\`t cambiar el tamaño de la
55 ventana, las coordenadas del contenedor y la única alineación de wi \ dget anidada. Sin embargo, si
56 Agregamos otros widgets al diseño que ganó.'t hold - \las coordenadas de la ` QLabel` se desplazarán de la
57 ventana, y nosotros 'Estaré dr \ awing en la ubicación incorrecta.
58
59
60 Esta es se arregla fácilmente moviendo la manija del mouse en el ` QLabel` sí mismo- eso'\Las coordenadas
61 del evento son siempre relativas a sí mismas. Esto lo envolvemos como un indivi\El objeto dual `Canvas`, que
62 maneja la creación de la superficie del mapa de píxeles, configura\
63 las ubicaciones x e y contiene el color actual del lápiz (establecido en negro por defecto\ult).
64

sesenta y cinco
65 T> Esta uno mismo-contenido `Lienzo` es una gota-en superficie dibujable que podrías usar en \tus propias
66 aplicaciones.
67
68
69 """pitón
70 importar sys
71 de PyQt5 importar QtCore, QtGui, QtWidgets, uicde
72 PyQt5.QtCore importar Qt
73
74 clase Lienzo(QtWidgets.QLabel):
```

```
75
76     def __init__(self):
77         super().__init__()
78         self.mapa_de_píxeles = QtGui.QPixmap(600, 300)
79         self.setPixmap(self.mapa_de_píxeles)
80
81         self.last_x, self.last_y = Ninguno,
82         Ninguno
83         self.pen_color = QColor('# 000000')
84
85     def set_pen_color(self, C):
86         self.pen_color = QColor(c)
87
88     def mouseMoveEvent(self, e):
89         if self.last_x is Ninguno: # Primer evento.
90             self.last_x = mi.X()
91             self.last_y = mi.y()
92             regreso # Ignore la primera vez.
93
94             pintor = QtGui.QPainter(self.pixmap())
95             pintor.setPen(p)
96             pintor.setBrush(self.pen_color)
97             pintor.drawLine(self.last_x, self.last_y, e.x(), e.y())
98             fin()
99
100            self.update()
101
102        # Actualice el origen para la próxima vez.
103        self.last_x = mi.X()
104        self.last_y = mi.y()
105
106    def mouseReleaseEvent(self, mi):
107        self.last_x = Ninguno
108        self.last_y = Ninguno
```

Para la selección de colores vamos a construir un widget personalizado, basado en QPushButton. Este widget acepta un color como parámetro, que puede ser una instancia de QColour, o un nombre de color ('rojo', 'negro') o un valor hexadecimal. Este color se establece en el fondo del widget.

para hacerlo identificable. Podemos usar el estándar QPushButton.pressed señal para conectarlo a cualquier acción.

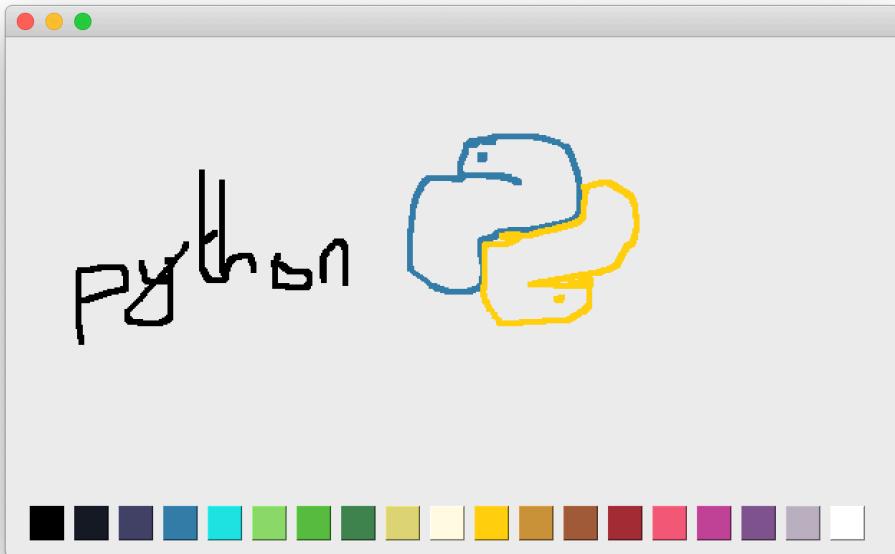
```
1 COLORES = [
2 # 17 matices https://lospec.com/palette-list/17undertones
3 '# 000000', '# 141923', '# 414168', '# 3a7fa7', '# 35e3e3', '# 8fd970', '# 5ebb49', '# 458352', '# dcd37b', '# 
4 fffee5', '# ffd035', '# cc9245', '# a15c3e', '# a42f3b', '# f45b7a', '# c24998', '# 81588d', '# bcb0c2', '#ffffff', ]
5
6
7
8
9 clase QPaletteButton(QtWidgets.QPushButton):
10
11     def __init__(self, color):
12         super().__init__()
13         self.setFixedSize(QtCore.QSize(24,24))
14         self.color = color
15         self.setStyleSheet("background-color: %s; color: %s" % (color, color))
```

Con esas dos partes definidas, simplemente necesitamos iterar sobre nuestra lista de colores, QPaletteButton pasando en el color, conecte su señal presionada a la set_pen_color controlador en el lienzo (indirectamente a través de un lambda para pasar los datos de color adicionales) y agregarlo al diseño de la paleta.

```
1 clase Ventana_principal(QtWidgets.QMainWindow):
2
3     def __init__(self):
4         super().__init__()
5
6         self.lienzo = Lienzo()
7
8         w = QtWidgets.QWidget()
9         l = QtWidgets.QVBoxLayout()
10        w.setLayout(l)
11        l.addWidget(self.lienzo)
12
13        paleta = QtWidgets.QHBoxLayout()
```

```
14     uno mismo.add_palette_buttons (paleta) l.  
15         addLayout (paleta)  
16  
17         uno mismo.setCentralWidget (w)  
18  
19     def add_palette_buttons(uno mismo, diseño):por  
20         C en COLORES:  
21             B = QPaletteButton (c)  
22             B.presionado.conectar(lambda C=c:     uno mismo.lienzo.set_pen_color (c))  
23             diseño.addWidget (b)  
24  
25  
26     aplicación = QtWidgets.QApplication (sys.argv)  
27     ventana = Ventana principal()  
28     ventana.show()  
29     aplicación.exec_()
```

Esto debería darle una aplicación de pintura multicolor completamente funcional, donde puede dibujar líneas en el lienzo y seleccionar colores de la paleta.



Desafortunadamente, eso no te hace bien.

Desafortunadamente, eso no te convierte en un buen artista.

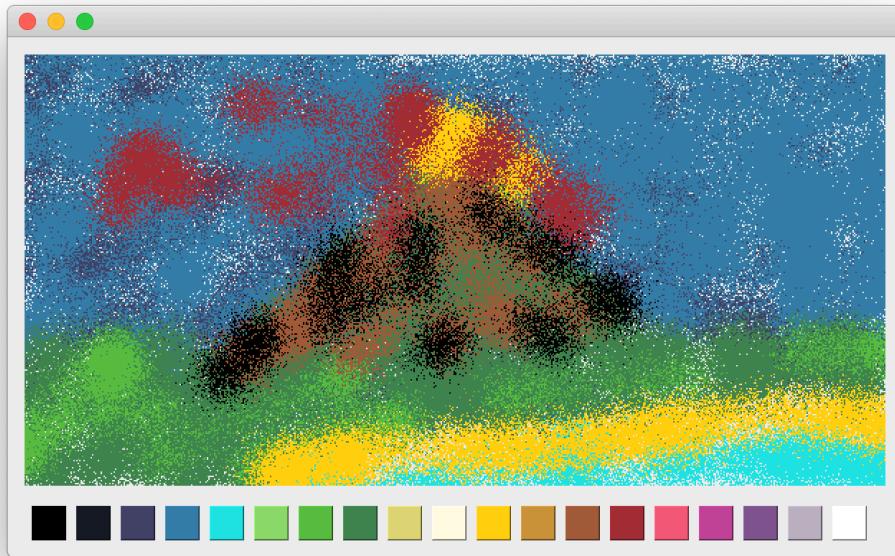
Rociar

Para divertirse un poco, puede cambiar el `mouseMoveEvent` con lo siguiente para dibujar con un efecto de "lata de aerosol" en lugar de una línea. Esto se simula usando `random.gauss` para generar una serie de *Normalmente distribuido* puntos alrededor de la posición actual del mouse que trazamos con `drawPoint`.

```
1     def mouseMoveEvent(uno mismo, e):
2         pintor = QtGui.QPainter (uno mismo.pixmap ()) p =
3             pintor.bolígrafo () p.setWidth (1)
4
5         pag.setColor (uno mismo.
6             pen_color) pintor.setPen (p)
7
8         por norte en distancia(SPRAY_PARTICLES):
9             xo = aleatorio.gauss(0, SPRAY_DIAMETER) yo = aleatorio
10                .gauss(0, SPRAY_DIAMETER) pintor.drawPoint (e.X() +
11                    xo, e.y () + yo)
12
13         uno mismo.actualizar()
```

Definir el SPRAY_PARTICLES y SPRAY_DIAMETER variables en la parte superior de su archivo e importe el aleatorio módulo de biblioteca estándar. La siguiente imagen muestra el comportamiento de la pulverización cuando se utilizan los siguientes ajustes:

```
1 importar aleatorio
2
3 SPRAY_PARTICLES    = 100
4 SPRAY_DIAMETER    = 10
```



Solo llámame Picasso

Solo llámame Picasso.

N> Para la lata de aerosol, no necesitamos rastrear la posición anterior, ya que siempre rociamos alrededor del punto actual.

Si desea un desafío, puede intentar agregar un botón adicional para alternar entre los modos de dibujo y rociado, o una entrada para definir el diámetro del cepillo / rociado.



Para obtener un programa de dibujo completamente funcional escrito con PyQt5, consulte mi aplicación de 15 minutos “Picasso” disponible aquí: <https://github.com/mfitzp/15-minute-apps/tree/master/paint>

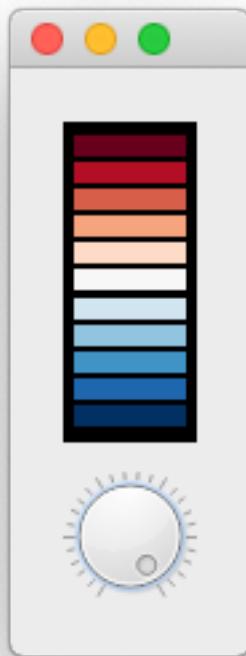
Esta introducción debería haberle dado una buena idea de lo que puede hacer con QPainter. Como se describe, este sistema es la base de todos los dibujos de widgets. Si quieras buscar más, echa un vistazo al `widget.pintura()` método, que recibe un QPainter ejemplo, para permitir que el widget se dibuje sobre sí mismo. Los mismos métodos que ha aprendido aquí se pueden utilizar en `pintura()` para dibujar algunos widgets personalizados básicos.

Creación de widgets personalizados

En el capítulo anterior presentamos QPainter y examinó algunas operaciones básicas de dibujo de mapas de bits que puede utilizar para dibujar puntos, líneas, rectángulos y círculos en un QPainter *superficie* como una QPixmap.

Este proceso de *dibujando en una superficie* con QPainter es de hecho la base por la cual se dibujan todos los widgets en Qt. Ahora ya sabes cómo usar QPainter ¡Sabes cómo dibujar tus propios widgets personalizados!

En este capítulo tomaremos lo que hemos aprendido hasta ahora y lo usaremos para construir un *personalizado* widget. Como ejemplo práctico, crearemos el siguiente widget: un medidor PowerBar personalizable con un control de dial.



PowerBar-metro

Este widget es en realidad una mezcla de *widget compuesto* y *widget personalizado* en que estamos usando el Qt incorporado QDial componente para el dial, mientras dibujamos la barra de potencia nosotros mismos. Luego, ensamblamos estas dos partes juntas en un widget principal que se puede colocar en su lugar sin problemas en cualquier aplicación, sin necesidad de saber cómo se ensamblan. El widget resultante proporciona el común QAbstractSlider interfaz con algunas adiciones para configurar la visualización de la barra.

Después de seguir este ejemplo, podrá crear sus propios widgets personalizados, ya sean compuestos de elementos integrados o maravillas autodibujadas completamente novedosas.

Empezando

Como hemos visto anteriormente, los widgets compuestos son simplemente widgets con un diseño aplicado, que a su vez contiene > 1 widget más. El "widget" resultante se puede utilizar como cualquier otro, con los componentes internos ocultos / expuestos como desee.

El esquema de nuestro *Barra de poder* El widget se proporciona a continuación: construiremos nuestro widget personalizado gradualmente a partir de este código auxiliar.

```
1 de PyQt5 importar QtCore, QtGui, QtWidgetsde
2 PyQt5.QtCore importar Qt
3
4
5 clase _Bar(QtWidgets.QWidget):
6     aprobar
7
8 clase Barra de poder(QtWidgets.QWidget):
9     """
10     Widget Qt personalizado para mostrar una barra de potencia y
11     marcar. Demostración de widget compuesto y personalizado. """
12
13
14     def __en eso__(uno mismo, pasos= 5, *argumentos, **kwargs):
15         súper(Barra de poder,  uno mismo).__en eso__(*argumentos, **kwargs)
16
17         diseño =    QtWidgets.QVBoxLayout ()
18         uno mismo._bar  = _Bar()
19         diseño.addWidget (uno mismo._bar)
20
21         uno mismo._marcar = QtWidgets.QDial()
22         uno mismo._marcar.setRange (-pasos, pasos)
23         uno mismo._marcar.setValue (0)
24         uno mismo.setLayout (diseño)
```

Esto simplemente define nuestra barra de potencia personalizada se define en el _Bar objeto - aquí solo una subclase inalterada QWidget. LosBarra de poderWidget (que es el widget completo) combina esto, usando un QVBoxLayout con el incorporado QDial para mostrarlos juntos.

Guarde esto en un archivo llamado power_bar.py

También necesitamos una pequeña aplicación de demostración para mostrar el widget.

```
1 de PyQt5 importar QtCore, QtGui, QtWidgetsde barra
2 de poder importar Barra de poder
3
4
5 aplicación = QtWidgets.QApplication []
6 volumen = Barra de poder()
7 volumen.show()
8 aplicación.exec_()
```

N> No necesitamos crear un QMainWindow ya que cualquier widget sin un parente es una ventana por derecho propio. Nuestra costumbreBarra de poder El widget aparecerá como cualquier ventana normal.

Esto es todo lo que necesita, simplemente guárdelo en la misma carpeta que el archivo anterior, en algo como demo.py. Puede ejecutar este archivo en cualquier momento para ver su widget en acción. Ejecútelo ahora y debería ver algo como esto:



PowerBar-dial

Si estira la ventana hacia abajo, verá que el dial tiene más espacio por encima que por debajo; esto lo ocupa nuestro (actualmente invisible) _Bar widget.

paintEvent

los paintEvent handler es el núcleo de todos los dibujos de widgets en PyQt.

Cada nuevo dibujo completo y parcial de un widget se activa a través de un paintEvent que maneja el widget para dibujarse a sí mismo. ApaintEvent puede ser provocado por -

- `repintar()` o `actualizar()` fue llamado
- el widget estaba oculto y ahora se ha descubierto
- se ha cambiado el tamaño del widget

- pero también puede ocurrir por muchas otras razones. Lo importante es que cuando un paintEvent se activa su widget es capaz de volver a dibujarlo.

Si un widget es lo suficientemente simple (como el nuestro), a menudo puede salirse con la suya simplemente redibujando todo en cualquier momento *cualquier cosa* sucede. Pero para widgets más complicados, esto puede volverse muy ineficiente. Para estos casos el paintEvent incluye la región específica que debe actualizarse. Usaremos esto en ejemplos posteriores más complicados.

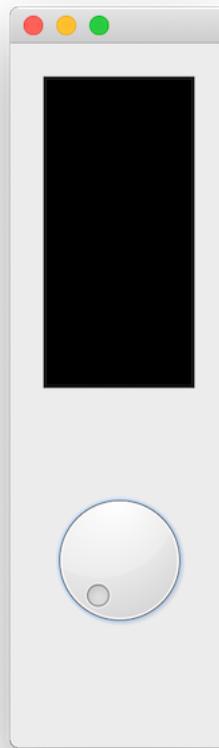
Por ahora haremos algo muy simple y solo llenaremos todo el widget con un solo color. Esto nos permitirá ver el área con la que estamos trabajando para comenzar a dibujar la barra.

```
1  def paintEvent(uno      mi):
2      mismo, pintor = QtGui.QPainter (uno mismo)
3      cepillar =  QtGui.QBrush ()
4      cepillar.setColor (QtGui.QColor ('negro')) cepillar.
5      setStyle (Qt.SolidPattern)
6      rect = QtCore.QRect (0, 0, pintor.dispositivo().ancho (), pintor.dispositivo().el\ 
7  ight ())
8      pintor.fillRect (rect, cepillo)
```

Posicionamiento

Ahora podemos ver el _Bar widget podemos modificar su posicionamiento y tamaño. Si arrastra alrededor de la forma de la ventana, verá que los dos widgets cambian de forma para adaptarse

el espacio disponible. Esto es lo que queremos, pero el QDial también se está expandiendo verticalmente más de lo que debería y dejando un espacio vacío que podríamos usar para la barra.

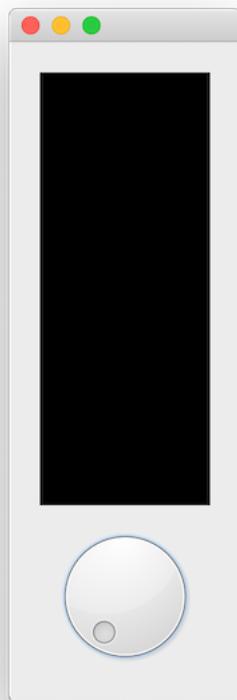


PowerBar-stretch

Nosotros podemos usar `setSizePolicy` en nuestro `_Bar` widget para asegurarse de que se expanda tanto como posible. Usando el `QSizePolicy.MinimumExpanding` el proporcionado `sizeHint` voluntad se utilizará como mínimo y el widget se expandirá tanto como sea posible.

```
1 clase _Bar(QtWidgets.QWidget):
2
3     def __init__(self, *args, **kwargs):
4         super().__init__(*args, **kwargs)
5
6         self.setSizePolicy(
7             QtWidgets.QSizePolicy.MinimumExpanding,
8             QtWidgets.QSizePolicy.MinimumExpanding
9         )
10
11    def sizeHint(self):
12        return QtCore.QSize(40, 120)
```

Todavía no es *Perfecto* como el `QDial`. El widget cambia de tamaño de forma un poco incómoda, pero nuestra barra ahora se está expandiendo para llenar todo el espacio disponible.



Política de PowerBar

Con el posicionamiento ordenado, ahora podemos pasar a definir nuestros métodos de pintura para dibujar nuestro medidor PowerBar en la parte superior (actualmente negra) del widget.

Actualizar la pantalla

Ahora tenemos nuestro lienzo completamente lleno de negro, a continuación usaremos QPainter dibujar comandos para dibujar algo en el widget.

Antes de comenzar con la barra, tenemos que hacer algunas pruebas para asegurarnos de que podemos actualizar la pantalla con los valores de nuestro dial. Actualizar el paintEvent con el siguiente código.

```

1     def paintEvent(uno mi):
2         mismo, pintaQPainter (uno mismo)
3
4         cepillar = QtGui.QBrush ()
5         cepillar.setColor (QtGui.QColor ('negro')) cepillar.
6         setStyle (Qt.SolidPattern)
7         rect = QtCore.QRect (0, 0, pintor.dispositivo().ancho (), pintor.dispositivo().él\
8 ight ())
9         pintor.fillRect (rect, cepillar)
10
11     # Obtener estado actual.
12     marcar = uno mismo.padre()._marcar
13     vmin, vmax = marcar.mínimo (), marcar.valor máximo
14     = marcar.valor()
15
16     lápiz = pintor.lápiz()
17     lápiz.setColor (QtGui.QColor ('rojo')) pintor.
18     setPen (bolígrafo)
19
20     fuente = pintor.font () fuente.
21     setFamily ('Veces') fuente.
22     setPointSize (18) pintor.setFont
23     (fuente)
24
25     pintor.dibujarTexto (25, 25, "{} -> {} <- {}".formato (vmin, valor, vmax)) pintor.fin()
26

```

Esto dibuja el fondo negro como antes, luego usa .padre() para acceder a nuestro parente. Barra de poder widget y a través de eso el QDial vía _marcar. A partir de ahí obtenemos el valor actual, así como los valores mínimo y máximo del rango permitido. Finalmente dibujamos los que usan el pintor, tal como hicimos en la parte anterior.



Dejamos el manejo del valor actual, los valores mínimo y máximo al QDial aquí, pero también podríamos almacenar ese valor nosotros mismos y usar señales hacia / desde el dial para mantener las cosas sincronizadas.

Ejecute esto, mueva el dial y... ...no pasa nada. Aunque hemos definido el paintEvent handler no vamos a activar un repaintado cuando cambia el dial.



Puede forzar una actualización cambiando el tamaño de la ventana, tan pronto como lo haga, debería ver aparecer el texto. UX ordenada, pero terrible: "¡solo cambia el tamaño de tu aplicación para ver tu configuración!"

Para solucionar esto, necesitamos conectar nuestro `_BarWidget` para repintarse a sí mismo en respuesta a los valores cambiantes en el dial. Podemos hacer esto usando `elQDial.valueChangedseñal`, conectándolo a un método de ranura personalizado que llama `.actualizar()` - desencadenando una pintura completa.

Agregue el siguiente método al `_Bar` widget.

```
1 def _trigger_refresh(uno mismo):uno  
2     mismo.actualizar()
```

... y agregue lo siguiente al `_en eso_` bloque para el padre Barra de poder widget.

```
1 uno mismo._marcar.valueChanged.conectar(uno mismo._bar._trigger_refresh)
```

Si vuelve a ejecutar el código ahora, verá que la pantalla se actualiza automáticamente a medida que gira el dial (haga clic y arrastre con el mouse). El valor actual se muestra como texto.

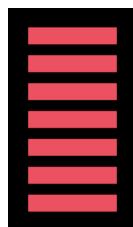


PowerBar-texto

Dibujando la barra

Ahora que tenemos la pantalla actualizando y mostrando el valor actual del dial, podemos pasar al dibujo de la barra actual. Esto es un poco complicado, con un poco de matemáticas para calcular las posiciones de las barras, pero lo repasaremos para dejar en claro lo que está sucediendo.

El siguiente esquema muestra lo que estamos buscando: una serie de *norte* recuadros, insertados desde los bordes del widget, con espacios entre ellos.



meta de poder

Calculando que dibujar

El número de cuadros a dibujar está determinado por el valor actual y qué tan lejos está entre el valor mínimo y máximo configurado para el QDial. Ya tenemos esa información en el ejemplo anterior.

```
1 marcar = uno mismo.padre()._marcar  
2 vmin, vmax = marcar.mínimo (), marcar.valor máximo =  
3 marcar.valor()
```

Si valor está a medio camino entre vmin y vmax luego queremos dibujar la mitad de las casillas (si tenemos 4 casillas en total, dibuja 2). Si valor Me senté vmax queremos dibujarlos todos.

Para hacer esto, primero convertimos nuestro valor en un número entre 0 y 1, donde 0 = vmin y 1 = vmax. Primero restamos vmin de valor para ajustar el rango de valores posibles para comenzar desde cero, es decir, desde vmin ... vmax para 0... (vmax-vmin). Dividiendo este valor por vmax-vmin (el nuevo máximo) luego nos da un número entre 0 y 1.

El truco entonces es multiplicar este valor (llamado ordenador personal abajo) por el número de pasos y eso nos da un número entre 0 y 5 - el número de casillas a dibujar.

```
1 ordenador personal = (valor - vmin) / (vmax - vmin)  
2 n_steps_to_draw = Ent(ordenador personal * 5)
```

Estamos envolviendo el resultado en Ent para convertirlo en un número entero (redondeando hacia abajo) para eliminar los cuadros parciales.

Actualizar el drawText en su evento de pintura para escribir este número en su lugar.

```
1 ordenador personal = (valor - vmin) / (vmax - vmin)  
2 n_steps_to_draw = Ent(ordenador personal * 5)  
3 pintor.dibujarTexto (25, 25, "0".formato (n_steps_to_draw))
```

Al girar el dial, verá un número entre 0 y 5.

Cajas de dibujo

A continuación, queremos convertir este número 0... 5 en un número de barras dibujadas en el lienzo. Empiece por quitar `eldrawText` y configuración de fuente y lápiz, ya que ya no los necesitamos.

Para dibujar con precisión, necesitamos saber el tamaño de nuestro lienzo, es decir, el tamaño del widget. También agregaremos un poco de relleno alrededor de los bordes para dar espacio alrededor de los bordes de los bloques contra el fondo negro.



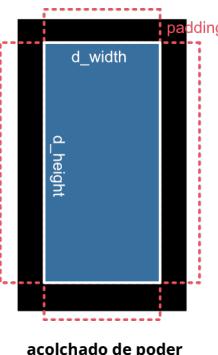
Todas las medidas en el QPainter están en píxeles.

```

1     relleno    = 5
2
3     # Definir    nuestro lienzo.
4     d_height = pintor.dispositivo().altura() - (relleno * 2) d_width = pintor.
5           dispositivo().ancho() - (relleno * 2)

```

Tomamos la altura y el ancho y restamos $2 \times$ acolchado fromeach— es 2x porque estamos llenando los bordes izquierdo y derecho (y superior e inferior). Esto nos da nuestro resultado/lienzo activo área en `d_height` y `d_width`.



Necesitamos romper nuestro `d_height` en 5 partes iguales, una para cada bloque; podemos calcular esa altura simplemente por `d_height / 5`. Además, como queremos espacios

entre los bloques, necesitamos calcular cuánto de este tamaño de paso ocupa el espacio (superior e inferior, por lo que se divide a la mitad) y cuánto es el bloque real.

- 1 Numero de pie = d_height / 5bar_height =
- 2 Numero de pie * 0,6espaciador_barras =
- 3 Numero de pie * 0,4 / 2

Estos valores son todo lo que necesitamos para dibujar nuestros bloques en nuestro lienzo. Para hacer esto, contamos hasta el número de pasos-1 comenzando desde 0 usando distancia y luego dibuja un fillRect sobre una región para cada bloque.

```

1 cepillar.setColor (QtGui.QColor ('rojo'))
2
3 por norte en distancia(5):
4     rect = QtCore.QRect (
5         relleno,
6         relleno + d_height - ((norte+1) * Numero de pie) + espaciador_barras,
7         ancho_d,
8         bar_height
9     )
10 pintor.fillRect (rect, cepillo)

```

N> El relleno se establece en un pincel rojo para empezar, pero lo personalizaremos más adelante.

La caja para dibujar fillRect se define como un QRect objeto al que pasamos, a su vez, la x izquierda, y superior, ancho y alto.

los *ancho* es el ancho completo del lienzo menos el relleno, que calculamos y almacenamos previamente en *d_width*. los *izquierda* x es igualmente solo el relleno valor (5px) desde el lado izquierdo del widget.

los *altura* de la barra *bar_height* calculamos como 0,6 veces el Numero de pie.

Esto deja el parámetro *2 altura_d - ((1 + n) * tamaño_paso) + espaciador_barras* cuales da el *arriba* y posición del rectángulo a dibujar. Este es el único cálculo que cambia a medida que dibujamos los bloques.

Un hecho clave para recordar aquí es que *y* coordenadas en QPainter comience en la parte superior y aumente hacia abajo en el lienzo. Esto significa que trazar *end_height* se trazará en

la parte inferior del lienzo. Cuando dibujamos un rectángulo desde un punto, se dibuja en el *Derecha y abajo* desde la posición inicial.



Para dibujar un bloque en la parte inferior, debemos comenzar a dibujar en `d_-altura-paso_tamaño` es decir, un bloque hacia arriba para dejar espacio para dibujar hacia abajo.

En nuestro medidor de barras estamos dibujando bloques, a su vez, comenzando desde abajo y trabajando hacia arriba. Así que nuestro primer bloque debe colocarse `end_height-step_size` y el segundo en `d_height- (step_size * 2)`. Nuestro ciclo itera desde 0 hacia arriba, por lo que podemos lograr esto con la siguiente fórmula:

1 `d_height - ((1 + norte) * Numero de pie)`

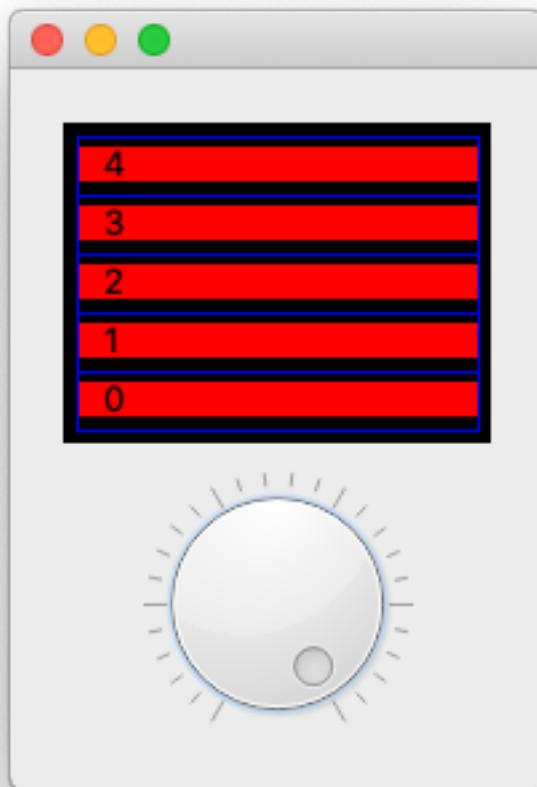
El ajuste final es tener en cuenta que nuestros bloques solo ocupan parte de cada `Numero de pie` (actualmente 0,6). Agregamos un poco de relleno para alejar el bloque del borde de la caja y colocarlo en el medio, y finalmente agregamos el relleno para el borde inferior. Eso nos da la fórmula final:

1 `relleno + d_height - ((norte+1) * Numero de pie) + espaciador_barras,`

Esto produce el siguiente diseño.



En la siguiente imagen, el valor actual de `norte` se ha impreso sobre la caja, y se ha dibujado un cuadro azul alrededor de la `Numero de pie` para que pueda ver el relleno y los espaciadores en efecto.



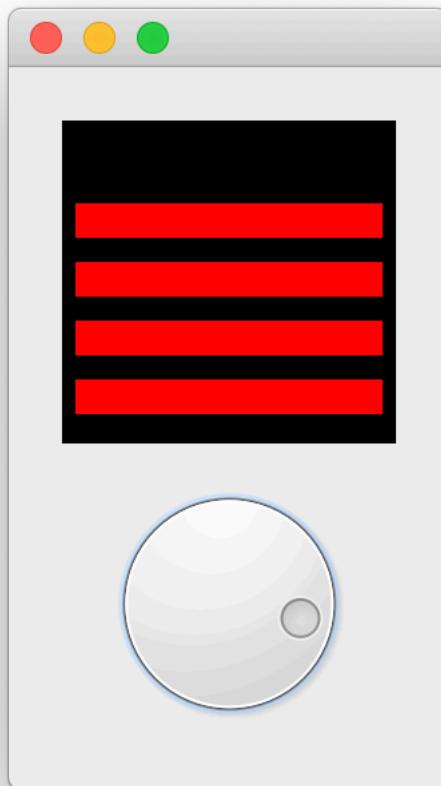
PowerBar-espaciador

Poniendo todo esto junto da el siguiente código, que cuando se ejecuta producirá un widget de barra de energía con bloques en rojo. Puede arrastrar la rueda hacia adelante y hacia atrás y las barras se moverán hacia arriba y hacia abajo en respuesta.

```
1 de PyQt5 importar QtCore, QtGui, QtWidgetsde
2 PyQt5.QtCore importar Qt
3
4 clase _Bar(QtWidgets.QWidget):
5
6     def __init__(uno mismo, *argumentos, **kwargs):
7         super().__init__(*argumentos, **kwargs)
8
9         uno mismo.setSizePolicy(
10             QtWidgets.QSizePolicy.Mínimo en expansión,
11             QtWidgets.QSizePolicy.Mínimo en expansión
12         )
13
14     def sizeHint(uno mismo):regreso QtCore
15         .QSize (40,120)
16
17     def paintEvent(uno mismo, pintor):
18         pintor.setBrush (QtGui.QColor ('negro'))
19
20         cepillar = QtGui.QPen ()
21         cepillar.setWidth (2)
22         cepillar.setStyle (Qt.SolidLine)
23
24         rect = QtCore.QRect (0, 0, pintor.dispositivo().ancho (), pintor.dispositivo().alto ())
25         pintor.drawRect (rect, cepillar)
26
27         # Obtener el estado actual.
28         marcar = uno mismo.padre().__marcar
29         vmin, vmax = marcar.mínimo (), marcar.valor máximo =
30             marcar.valor()
31
32         relleno = 5
33
34         # Definir nuestro lienzo.
35         d_height = pintor.dispositivo().altura () - (relleno * 2) d_width = pintor.
36         dispositivo().ancho () - (relleno * 2)
37
38         # Dibuja las barras.Numero de
39         pie = d_height / 5
```

```
40     bar_height = Numero de pie * 0,6
41     espaciador_barras = Numero de pie * 0,4 / 2
42
43     ordenador_personal = (valor - vmin) / (vmax - vmin)
44     n_steps_to_draw = En t(ordenador_personal * 5)
45     cepillar.setColor (QtGui.QColor ('rojo'))por norte
46     en distancia(n_pasos_para_dibujar):
47         rect = QtCore.QRect (
48             relleno,
49             relleno + d_height - ((norte+1) * Numero de pie) + espaciador_barras,
50             ancho_d,
51             bar_height
52         )
53         pintor.fillRect (rect, cepillo)
54
55     pintor.fin()
56
57     def _trigger_refresh(uno mismo):uno
58         mismo.actualizar()
59
60
61     clase Barra de poder(QtWidgets.QWidget):
62         """
63             Widget Qt personalizado para mostrar una barra de potencia y
64             marcar. Demostración de widget compuesto y personalizado. """
65
66
67     def __en eso__(uno mismo, pasos= 5, *argumentos, **kwargs):
68         super(Barra de poder, uno mismo).__en eso__(*argumentos, **kwargs)
69
70         diseño = QtWidgets.QVBoxLayout ()
71         uno mismo._bar = _Bar()
72         diseño.addWidget (uno mismo._bar)
73
74         uno mismo._marcar = QtWidgets.QDial ()uno
75         mismo._marcar.valueChanged.conectar(
76             uno mismo._bar._trigger_refresh
77         )
78
```

```
79     diseño.addWidget(uno mismo._marcar)  
80     uno mismo.setLayout(diseño)
```



PowerBar-básico

Eso ya funciona, pero podemos ir más allá para proporcionar más personalización, agregar algunas mejoras de UX y mejorar la API para trabajar con nuestro widget.

Personalizando la barra

Ahora tenemos una barra de potencia en funcionamiento, controlable con un dial. Pero es bueno al crear widgets proporcionar opciones para configurar el comportamiento de su widget para hacerlo más flexible. En esta parte, agregaremos métodos para establecer números personalizables de segmentos, colores, relleno y espaciado.

Los elementos que vamos a proporcionar personalización son los siguientes:

Opción	Descripción
número de barras	¿Cuántas barras se muestran en el widget
colores	Colores individuales para cada una de las barras
color de fondo	El color del lienzo de dibujo (predeterminado negro)
relleno	Espacio alrededor del borde del widget, entre barras y borde de lona.
altura de la barra / porcentaje de la barra	Proporción (0... 1) de la barra que es sólida (el resto estará espaciado entre barras adyacentes)

Podemos almacenar cada uno de estos como atributos en el `_bar` objeto, y utilizarlos desde el `paintEvent` método para cambiar su comportamiento.

los `_Barra.__init__` se actualiza para aceptar un argumento inicial para el número de barras (como un número entero) o los colores de las barras (como una lista de `QColor`, valores hexadecimales o nombres). Si se proporciona un número, todas las barras serán de color rojo. Si se proporciona una lista de colores, el número de barras se determinará a partir de la longitud del lista de colores. Valores predeterminados para `self._bar.Solid_percent`, `self._background_color`, `self._padding` también se establecen.

```
1 clase _Bar(QtWidgets.QWidget):
2     clickedValue = QtCore.pyqtSignal (En t)
3
4     def __en eso__(uno mismo, pasos, *argumentos, **kwargs):
5         super().__en eso__(*argumentos, **kwargs)
6
7         uno mismo.setSizePolicy (
8             QtWidgets.QSizePolicy.Mínimo en expansión,
9             QtWidgets.QSizePolicy.Mínimo en expansión
10        )
11
12     si isinstance(pasos, lista):
13         # lista de colores.uno
14         mismo.n_pasos = len(pasos)
15         uno mismo.pasos = pasos
16
17     elif isinstance(pasos,      En t):
18         # int número de barras, predeterminado en rojo.
19         uno mismo.n_pasos = pasos uno mismo.pasos = [
20             'rojo'] * pasos
21
22     demás:
23         aumentar Error de tecleado('los pasos deben ser una lista o int')
24
25     uno mismo._bar_solid_percent = 0,8
26     uno mismo._color de fondo = QtGui.QColor ('negro')
27     uno mismo._relleno = 4.0 # Espacio de n píxeles alrededor del borde.
```

Así mismo actualizamos la PowerBar `__init__` para aceptar el parámetro de pasos y pasarlo.

```
1 clase Barra de poder(QtWidgets.QWidget):
2     def __en eso__(uno mismo, pasos= 5, *argumentos, **kwargs):
3         super().__en eso__(*argumentos,      * kwargs)
4
5         diseño =    QtWidgets.QVBoxLayout ()
6         uno mismo._bar   = _Bar (pasos)
7
8         #... continuado como antes.
```

Ahora tenemos los parámetros en su lugar para actualizar el `paintEvent` método. El código modificado se muestra a continuación.

```
1 def paintEvent(uno      mi):
2     mismo, pintor = QtGui.QPainter (uno mismo)
3
4     cepillar =  QtGui.QBrush ()
5     cepillar.setColor (uno mismo._background_color)
6     pincel.setStyle (Qt.SolidPattern)
7     rect = QtCore.QRect (0, 0, pintor.dispositivo().ancho (), pintor.dispositivo().él\
8 ight ())
9     pintor.fillRect (rect,          cepillar)
10
11     # Obtener estado actual.
12     padre   = uno mismo.padre()
13     vmin, vmax = padre.mínimo (), padre.valor máximo =
14             padre.valor()
15
16     # Definir nuestro lienzo.
17     d_height = pintor.dispositivo().altura() - (uno mismo._relleno * 2) d_width =
18     pintor.dispositivo().ancho() - (uno mismo._relleno * 2)
19
20     # Dibuja las barras.
21     Numero de pie = d_height / uno mismo.n_pasos bar_height = Numero de pie *
22     uno mismo._bar_solid_percent bar_spacer = Numero de pie * (1 - uno mismo.
23     _bar_solid_percent) / 2
24
25     # Calcule la posición de parada y, a partir del valor en el rango.ordenador
26     personal = (valor - vmin) / (vmax - vmin)
```

```

27     n_steps_to_draw = En t(ordenador personal * uno mismo.n_pasos)
28
29     por norte en distancia(n_pasos_para_dibujar):
30         cepillar.setColor (QtGui.QColor (uno mismo.pasos [n])) rect
31         = QtCore.QRect (
32             uno mismo.relleno,
33             uno mismo.relleno + d_height - ((1 + norte) * Numero de pie) + espaciador_barras,
34             ancho_d,
35             bar_height
36         )
37         pintor.fillRect (rect, cepillo)
38
39     pintor.fin()

```

Ahora puede experimentar pasando diferentes valores para el init a Barra de poder, por ejemplo, aumentando el número de barras o proporcionando una lista de colores. A continuación se muestran algunos ejemplos: una buena fuente de paletas hexadecimales es el[Fuente de bokeh](#).

```

1 Barra de poder(10)
2 Barra de poder(3)
3 Barra de poder(["# 5e4fa2", "# 3288bd", "# 66c2a5", "# abddaa4", "# e6f598", "#ffffbf", "#F\ee08b ", "# fdae61", "# f46d43", "# d53e4f", "# 9e0142"])
4 Barra de poder(["# a63603", "# e6550d", "# fd8d3c", "# fdae6b", "# fdd0a2", "#feedde"])

```



ejemplos de poder

Puede jugar con la configuración de relleno a través de las variables, por ejemplo self._bar_solid_percent pero sería mejor proporcionar métodos adecuados para configurarlos.

N> Estamos siguiendo el estándar Qt de los nombres de método de camelCase para estos métodos externos para mantener la coherencia con los otros heredados de QDial.

```

1   def setColor(uno mismo, color):
2       uno mismo._bar.pasos = [color] * uno mismo._bar.n_pasosuno
3       mismo._bar.actualizar()
4
5   def setColors(uno mismo, colores):uno
6       mismo._bar.n_pasosuno mismo.mismo._bar.actualizar()
7       pasos = uno mismo._bar.actcolores$#
8
9
10  def setBarPadding(uno mismo, I):
11      uno mismo._bar._relleno = En t(I)
12      mismo._bar.actualizar()
13
14  def setBarSolidPercent(uno mismo, uno F):
15      mismo._bar._bar_solid_percentuno = flotador(F)
16      mismo._bar.actualizar()
17
18  def setBackgroundColor(uno mismo, color):
19      uno mismo._bar._color de fondo = QtGui.QColor (color)uno mismo.
20      _bar.actualizar()

```

En cada caso, establecemos la variable privada en el `_bar` objeto y luego llamar `_bar.update()` para activar un nuevo dibujo del widget. El método admite cambiar el color a un solo color o actualizar una lista de ellos; también se puede usar la configuración de una lista de colores para cambiar el número de barras.

N> No hay ningún método para establecer el recuento de barras, ya que expandir una lista de colores sería falso.
¡Pero siéntase libre de intentar agregar esto usted mismo!

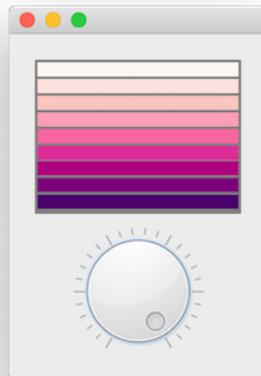
A continuación, se muestra un ejemplo con un relleno de 25 píxeles, una barra completamente sólida y un fondo gris.

```

1  bar = Barra de poder(["# 49006a", "# 7a0177", "# ae017e", "# dd3497", "# f768a1", "# fa9fb5\\"", "# fcc5c0", "#"
2  fde0dd", "# fff7f3"]) bar.setBarPadding (2) bar.setBarSolidPercent (0,9) bar.setBackgroundColor ('gris')
3
4
5

```

Con esta configuración obtienes el siguiente resultado.



atributos de poder

Adición de la interfaz QAbstractSlider

Hemos agregado métodos para configurar el comportamiento de la barra de energía. Pero actualmente no proporcionamos ninguna forma de configurar el estándar `QDial` métodos, por ejemplo, establecer el tamaño mínimo, máximo o de paso, desde nuestro widget. Podríamos trabajar y agregar métodos de envoltura para todos estos, pero se volvería muy tedioso muy rápidamente.

- 1 *# Ejemplo de un contenedor único, necesitaríamos más de 30 de estos.*
- 2 `def setNotchesVisible(uno mismo, B):regreso uno`
- 3 `mismo._marcar.setNotchesVisible (b)`

En su lugar, podemos agregar un pequeño controlador en nuestro widget externo para buscar automáticamente métodos (o atributos) en el `QDial` ejemplo, si no existen directamente en nuestra clase. De esta manera podemos implementar nuestros propios métodos, pero aún así obtener todos los `QAbstractSlider` bondad gratis.

El contenedor se muestra a continuación, implementado como un personalizado `__getattribute__` método.

```

1 def __getattr__(uno mismo, nombre):si
2     nombre en uno mismo.__dict__:
3         regreso uno mismo[nombre]
4
5     tratar:
6         regreso getattr(uno mismo._marcar, nombre)
7     excepto AttributeError:
8         aumentar AttributeError(
9             "El objeto '{}' no tiene atributo '{}'".formato(uno mismo.__clase__.nombre_, n \
10 ame)
11     )

```

Al acceder a una propiedad (o método), por ejemplo, al llamar PowerBar.setNotchesVisible (tr Python utiliza internamente `__getattr__` para obtener la propiedad del objeto actual. Este controlador hace esto a través del diccionario de objetos.`yo __ dict__`. Hemos anulado este método para proporcionar nuestra lógica de manejo personalizada.

Ahora, cuando llamamos PowerBar.setNotchesVisible (verdadero), este controlador primero mira nuestro objeto actual (un Barra de poder instancia) para ver si `.setNotchesVisible` existe y si lo hace lo usa. Si no luego llama `getattr()` sobre `self._dial` en su lugar, devuelve lo que encuentra allí. Esto nos da acceso a todos los métodos de `QDial` de nuestra costumbre Barra de poderwidget.

Si `QDial` tampoco tiene el atributo y genera un `AttributeError` lo capturamos y lo volvemos a subir desde nuestro widget personalizado, donde pertenece.



Esto funciona para cualquier propiedad o método, incluidas las señales. Entonces el estándar `QDial` señales como `.valueChanged` también están disponibles.

Actualización desde la pantalla del medidor

Actualmente puede actualizar el valor actual del medidor PowerBar jugando con el dial. Pero sería bueno si también pudiera actualizar el valor haciendo clic en una posición en la barra de energía o arrastrando el mouse hacia arriba y hacia abajo. Para hacer esto podemos actualizar nuestro `_Bar` widget para manejar eventos del mouse.

```

1 clase _Bar(QtWidgets.QWidget):
2
3     clickedValue = QtCore.pyqtSignal (En t)
4
5     #... código existente ...
6
7     def _calculate_clicked_value(uno mismo, e): padre
8         = uno mismo.padre()
9         vmin, vmax = padre.mínimo (), padre.máxima () d_altura = uno mismo.
10        Talla().altura() + (uno mismo._relleno * 2) Numero de pie = d_height /
11        uno mismo.n_pasos click_y = mi.y () - uno mismo._relleno - Numero de
12        pie / 2
13
14        ordenador personal = (d_altura - click_y) / valor
15        d_height = vmin + ordenador personal * (vmax - vmin)
16        uno mismo.clickedValue.emitir (valor)
17
18    def mouseMoveEvent(uno mismo, e):uno
19        mismo._calculate_clicked_value (e)
20
21    def mousePressEvent(uno mismo, e):uno
22        mismo._calculate_clicked_value (e)

```

En el_en eso_bloque para elBarra de poderWidgetspodemos conectarnos al_Bar.clickedValue señal y envía los valores a self._dial.setValue para establecer el valor actual en el dial.

```

1 # Reciba comentarios de los eventos de clic en el medidor.uno mismo.
2 _bar.clickedValue.conectar(unos mismo._marcar.valor ajustado)

```

Si ejecuta el widget ahora, podrá hacer clic en el área de la barra y el valor se actualizará y el dial girará en sincronía.

El código final

A continuación se muestra el código final completo para nuestro widget de medidor PowerBar, llamado Barra de poder. Puede guardar esto sobre el archivo anterior (por ejemplo, llamadopower_bar.py) y luego úsalo

en cualquiera de sus propios proyectos, o personalícelo según sus propios requisitos.

```
1  de PyQt5 importar QtCore, QtGui, QtWidgetsde
2  PyQt5.QtCore importar Qt
3
4
5  clase _Bar(QtWidgets.QWidget):
6
7      clickedValue = QtCore.pyqtSignal (En t)
8
9      def __en eso__(uno mismo, pasos, *argumentos, **kwargs):
10          súper().__en eso__(*argumentos, **kwargs)
11
12          uno mismo.setSizePolicy (
13              QtWidgets.QSizePolicy.Mínimo en expansión,
14              QtWidgets.QSizePolicy.Mínimo en expansión
15          )
16
17          si isinstance(pasos, lista):
18              # lista de colores.uno
19              mismo.n_pasos = len(pasos)
20              uno mismo.pasos =  pasos
21
22          elif isinstance(pasos,     En t):
23              # int número de barras, predeterminado en rojo.
24              uno mismo.n_pasos = pasosuno mismo.pasos = [
25                  'rojo'] * pasos
26
27          demás:
28              aumentar Error de tecleado('los pasos deben ser una lista o int')
29
30          uno mismo._bar_solid_percent = 0,8
31          uno mismo._color de fondo = QtGui.QColor ('negro')
32          uno mismo._relleno = 4.0 # Espacio de n píxeles alrededor del borde.
33
34      def paintEvent(uno mismo, e): pintor = QtGui
35          .Painter (uno mismo)
```

```
37     cepillar = QtGui.QBrush ()
38     cepillar.setColor (uno mismo._background_color)
39     pincel.setStyle (Qt.SolidPattern)
40     rect = QtCore.QRect (0, 0, pintor.dispositivo().ancho (), pintor.dispositivo().él\
41   ight ())
42     pintor.fillRect (rect, cepillar)
43
44     # Obtener estado actual.
45     padre = uno mismo.padre()
46     vmin, vmax = padre.mínimo (), padre.valor máximo =
47           padre.valor()
48
49     # Definir nuestro lienzo.
50     d_height = pintor.dispositivo().altura () - (uno mismo._relleno * 2) d_width =
51     pintor.dispositivo().ancho () - (uno mismo._relleno * 2)
52
53     # Dibuja las barras.
54     Numero de pie = d_height / uno mismo.n_pasos bar_height = Numero de pie *
55     uno mismo._bar_solid_percent bar_spacer = Numero de pie * (1 - uno mismo.
56     _bar_solid_percent) / 2
57
58     # Calcule la posición de parada y, a partir del valor en el rango.ordenador
59     personal = (valor - vmin) / (vmax - vmin) n_steps_to_draw = En t(ordenador
60     personal * uno mismo.n_pasos)
61
62     por norte en distancia(n_pasos_para_dibujar):
63         cepillar.setColor (QtGui.QColor (uno mismo.pasos [n])) rect
64         = QtCore.QRect (
65             uno mismo._relleno,
66             uno mismo._relleno + d_height - ((1 + norte) * Numero de pie) + espaciador_barras,
67             ancho_d,
68             bar_height
69         )
70         pintor.fillRect (rect, cepillo)
71
72     pintor.fin()
73
74     def sizeHint(uno mismo):
75         regreso QtCore.QSize (40, 120)
```

```
76
77     def _trigger_refresh(uno mismo):uno
78         mismo.actualizar()
79
80     def _calculate_clicked_value(uno mismo,      mi):
81         padre = uno mismo.padre()
82         vmin, vmax = padre.mínimo (), padre.máxima () d_altura = uno mismo.
83         Talla().altura() + (uno mismo._relleno * 2) Número de pie = d_height /
84         uno mismo.n_pasos click_y = mi.y () - uno mismo._relleno - Número de
85         pie / 2
86
87         ordenador personal = (d_altura - click_y) / valor
88         d_height = vmin + ordenador personal * (vmax - vmin)
89         uno mismo.clickedValue.emitir (valor)
90
91     def mouseMoveEvent(uno mismo, e):uno
92         mismo._calculate_clicked_value (e)
93
94     def mousePressEvent(uno mismo, e):uno
95         mismo._calculate_clicked_value (e)
96
97
98 clase Barra de poder(QtWidgets.QWidget):
99     """
100     Widget Qt personalizado para mostrar una barra de potencia y
101     marcar. Demostración de widget compuesto y personalizado.
102
103     Al hacer clic con el botón izquierdo en el botón se muestra el selector de color, mientras
104     que al hacer clic con el botón derecho se restablece el color a Ninguno (sin color). """
105
106
107     colorChanged = QtCore.pyqtSignal ()
108
109     def __en eso__(uno mismo, pasos= 5, *argumentos, **kwargs):
110         super().__en eso__(*argumentos,      * kwargs)
111
112         diseño = QtWidgets.QVBoxLayout ()
113         uno mismo._bar  = _Bar (pasos)
114         diseño.addWidget (uno mismo._bar)
```

```
115  
116     # Cree el widget QDial y configure los valores predeterminados.  
117     #: proporcionamos descriptores de acceso en esta clase para anularlos.  
118     uno mismo._marcar = QtWidgets.QDial ()uno mismo._marcar.  
119     setNotchesVisible (Cierto)uno mismo._marcar.setWrapping (Falso)  
120  
121     uno mismo._marcar.valueChanged.conectar(uno mismo._bar._trigger_refresh)  
122  
123     # Reciba comentarios de los eventos de clic en el medidor.uno mismo.  
124     _bar.clickedValue.conectar(uno mismo._marcar.valor ajustado)  
125  
126     diseño.addWidget (uno mismo._marcar)  
127     uno mismo.setLayout (diseño)  
128  
129     def __getattr__(uno mismo, nombre):si  
130         nombre en uno mismo.__dict__:  
131             regreso uno mismo[nombre]  
132  
133         regreso getattr(uno mismo._dial, nombre)  
134  
135     def setColor(uno mismo, color):  
136         uno mismo._bar.pasos = [color] * uno mismo._bar.n_pasosuno  
137         mismo._bar.actualizar()  
138  
139     def setColors(uno mismo, colores):uno  
140         mismo._bar.n_pasosuno mismo._bar.colores  
141         pasos =uno mismo._bar.actcolores$  
142  
143  
144     def setBarPadding(uno mismo, I):  
145         uno mismo._bar._rellenouno = En t(I)  
146         mismo._bar.actualizar()  
147  
148     def setBarSolidPercent(uno mismo,uno F):  
149         mismo._bar._bar_solid_percentuno      = flotador(F)  
150         mismo._bar.actualizar()  
151  
152     def setBackgroundColor(uno mismo, color):  
153         uno mismo._bar._color de fondo = QtGui.QColor (color)
```

154

```
uno mismo._bar.actualizar()
```

Debería poder utilizar muchas de estas ideas para crear sus propios widgets personalizados. Para obtener más ejemplos, eche un vistazo a la [Aprenda la biblioteca de widgets de PyQt](#) - estos widgets son de código abierto y están disponibles gratuitamente para usar en sus propios proyectos.

La arquitectura de la vista del modelo

A medida que comience a crear aplicaciones más complejas con PyQt5, probablemente encontrará problemas para mantener los widgets sincronizados con sus datos.

Los datos almacenados en widgets (por ejemplo, un simple QListWidget) no está disponible para manipular desde Python: los cambios requieren que obtenga un elemento, obtenga los datos y luego vuelve a configurarlo. La solución predeterminada para esto es mantener una representación de datos externos en Python y luego duplicar las actualizaciones tanto de los datos como del widget, o simplemente reescribir el widget completo a partir de los datos. Esto puede volverse feo rápidamente y da como resultado una gran cantidad de texto estándar solo para manipular los datos.

Afortunadamente, Qt tiene una solución para esto: ModelViews. ModelViews son una alternativa poderosa a los widgets de visualización estándar, que utilizan una interfaz de modelo regular para interactuar con fuentes de datos, desde estructuras de datos simples hasta bases de datos externas. Esto aísla sus datos, lo que permite que se mantengan en cualquier estructura que desee, mientras que la vista se encarga de la presentación y las actualizaciones.

Este capítulo presenta los aspectos clave de la arquitectura ModelView de Qt y la utiliza para crear una aplicación Todo de escritorio simple en PyQt5.

Controlador de vista de modelo

Modelo – Vista – Controlador (MVC) es un patrón arquitectónico utilizado para desarrollar interfaces de usuario que divide una aplicación en tres partes interconectadas. Esto separa la representación interna de los datos de cómo se presenta y acepta la información al usuario.

El patrón de diseño MVC desacopla tres componentes principales:

- **Modelo** contiene la estructura de datos con la que está trabajando la aplicación.
- **Vista** es cualquier representación de la información que se muestra al usuario, ya sea en forma gráfica o en tablas. Se permiten múltiples vistas del mismo modelo de datos.
- **Controlador** acepta la entrada del usuario, transformándola en comandos para el modelo o la vista.

Si aterriza, la distinción entre Vista y Controlador se vuelve un poco turbia. Qt acepta eventos de entrada del usuario (a través del sistema operativo) y los delega a los widgets (controlador) para que los manejen. Sin embargo, los widgets también manejan la presentación del estado actual al usuario, colocándolos directamente en la Vista. En lugar de agonizar sobre dónde trazar la línea, en Qt-speak la Vista y el Controlador se fusionan creando una arquitectura Modelo / ViewController, llamada "Vista Modelo" por simplicidad.

Es importante destacar que la distinción entre *datos* y *como se presenta* se conserva.

La vista del modelo

El modelo actúa como interfaz entre el almacén de datos y ModelView. El modelo contiene los datos (o una referencia a ellos) y presenta estos datos a través de una API estandarizada que las vistas luego consumen y presentan al usuario. Varias vistas pueden compartir los mismos datos, presentándolos de formas completamente diferentes.

Puede utilizar cualquier "almacén de datos" para su modelo, incluyendo, por ejemplo, una lista o diccionario estándar de Python, o una base de datos (a través de, por ejemplo, SQLAlchemy); depende totalmente de usted.

Las dos partes son esencialmente responsables de:

1. El **modelo** almacena los datos, o una referencia a ellos y devuelve registros individuales o rangos de registros, y metadatos asociados o *monitor* instrucciones.
2. El **vista** solicita datos del modelo y muestra lo que se devuelve en el widget.



Hay una vista en profundidad de la arquitectura Qt aquí: <http://doc.qt.io/qt-5/model-view-programming.html>

Una vista de modelo simple: una lista de tareas pendientes

Para demostrar cómo usar ModelViews en la práctica, crearemos una implementación muy simple de una Lista de tareas de escritorio. Esto consistirá en un QListWidget por

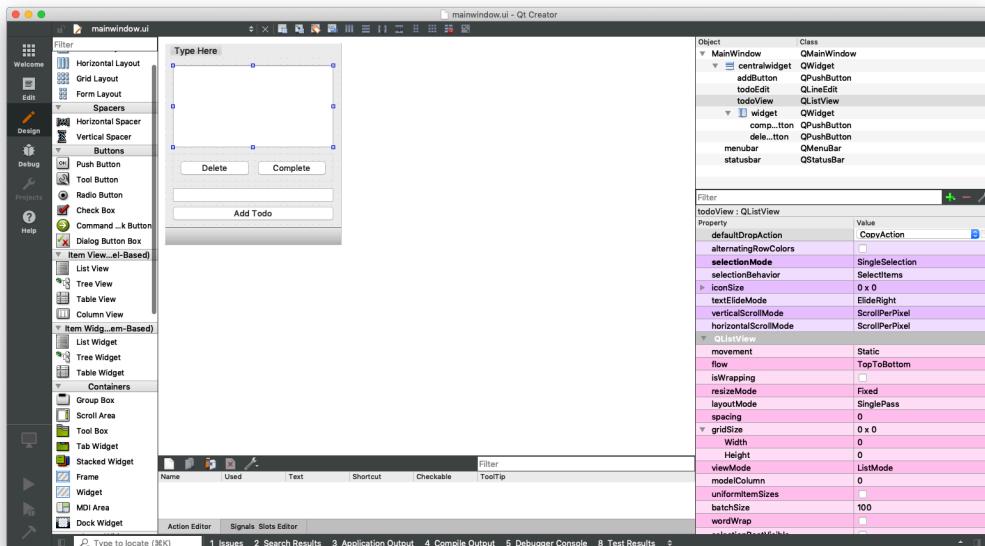
la lista de elementos, un QLineEdit para ingresar elementos nuevos y un conjunto de botones para agregar, eliminar o marcar elementos como hechos.



El código y los archivos asociados para este ejemplo se encuentran en el archivo fuente descargable.

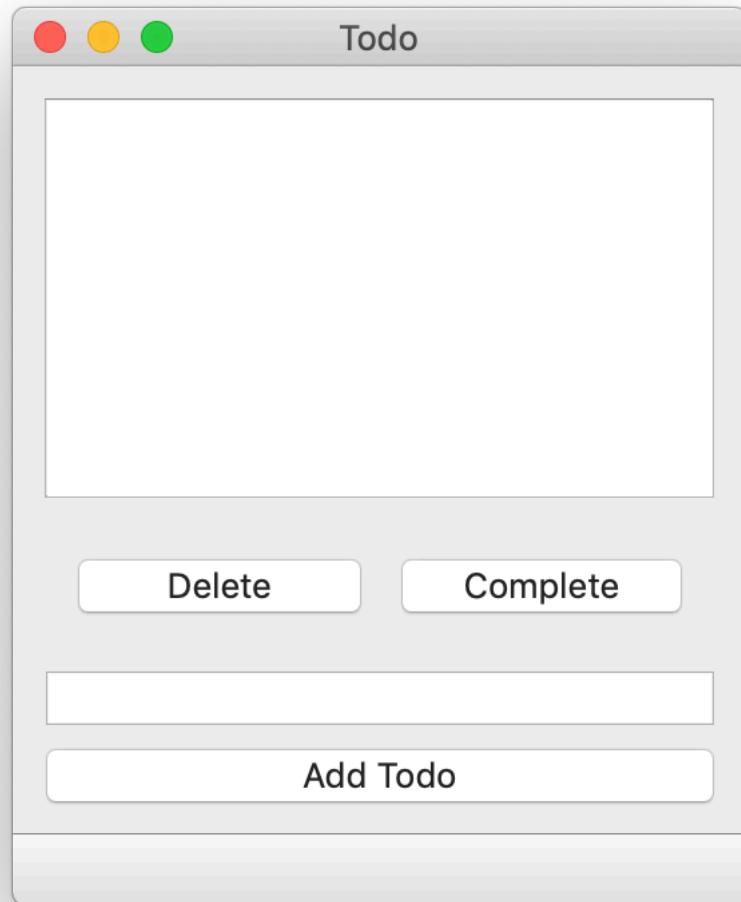
La interfaz de usuario

La interfaz de usuario simple se diseñó con Qt Creator y se guardó como mainwindow.ui. Los.ui El archivo está incluido en las descargas de este libro.



Diseñar la interfaz de usuario en Qt Creator

La aplicación en ejecución se muestra a continuación.



La ventana principal

Los widgets disponibles en la interfaz recibieron los ID que se muestran en la siguiente tabla.

nombre del objeto	Escribe	Descripción
todoView	QListView	La lista de todos los actuales
todoEditar	QLineEdit	La entrada de texto para crear un nuevo
addButton	QPushButton	todo artículo Cree la nueva tarea y agréguela a
deleteButton	QPushButton	la lista de todos Eliminar la tarea actual seleccionada,
completeButton	QPushButton	eliminarlo de la lista de tareas pendientes Marcar la tarea actual seleccionada como
		hecho

Usaremos estos identificadores para conectar la lógica de la aplicación más adelante.

El modelo

Definimos nuestro modelo personalizado subclasiﬁcando a partir de una implementación base, lo que nos permite centrarnos en las partes exclusivas de nuestro modelo. Qt proporciona varias bases de modelos diferentes, incluidas las que admiten datos multidimensionales (piense en una hoja de cálculo).

Pero para este ejemplo solo necesitamos una lista simple para nuestros datos y estamos mostrando el resultado a una QListView. El modelo base coincidente para esto es QAbstractListModel. La deﬁnición del esquema de nuestro modelo se muestra a continuación.

```

1 clase TodoModel(QtCore.QAbstractListModel):
2     def __init__(self, *args, todos=Ninguno, **kwargs):
3         super(TodoModel, self).__init__(*args, **kwargs)
4         self.todos = todos or []
5
6     def datos(self, index, role):
7         if role == Qt.DisplayRole:
8             # Vea a continuación la estructura de datos.estado,
9             texto = self.todos[index.row()]
10            # Devuelve solo el texto de tareas pendientes.
11            return texto
12
13    def número_de_filas(self, index):
14        return len(self.todos)

```

`los.todos` variable es nuestro almacén de datos y los dos métodos `número de filas()` y `datos()` son métodos de modelo estándar que debemos implementar para un modelo de lista. Revisaremos estos a continuación.

. lista de todos

El almacén de datos de nuestro modelo es `.todos`, una lista de Python simple en la que almacenaremos un tupla de valores en el formato `[(bool, str), (bool, str), (bool, str)]` dónde `bool` es el *hecho* estado de una entrada determinada, y `str` es el texto de la tarea.

Nosotros inicializamos `self.todo` a una lista vacía al inicio, a menos que se pase una lista a la vista del `todos` argumento de palabra clave.



`self.todos = todos or []` establecerá `self.todos` al valor de lo proporcionado `todos` variable si es *veraz* (es decir, cualquier otra cosa que no sea una lista vacía, thebool '`Falso`' o `Ninguno` el valor predeterminado), de lo contrario, se establecerá en la lista vacía `[]`.

Para crear una instancia de este modelo, simplemente podemos hacer:

```
1 modelo = TodoModel ()      # crear una lista de tareas vacía
```

O para pasar una lista existente -

```
1 todos = [(Falso, 'un artículo'), (Falso, 'otro artículo')] modelo
2     = TodoModel (todos)
3     ...
4
5     ##### .número de filas()
6
7 los `rowcount ()` La vista llama al método para obtener el número de filas en los\ 
8 datos actuales. Esto es requeridopor la vista para saber cuál es el índice máximo que \puede solicitar del
9 almacén de datos es ('row count-1'). Desde que nosotrosEstás usando una lista de Python como nuestro
10 almacén de datos, el valor de retorno para esto es simplemente el `len ()` de la lista de \.
11
12
```

```
13 # ## #. datos()
14
15 Este es el núcleo de su modelo, que maneja las solicitudes de datos de la vista y devuelve el resultado
16 apropiado. Recibe dos parámetros `index` y `role` .
17
18
19 `index` es la posición / coordenadas de los datos que la vista solicita, especificada por dos métodos` .row ()
20 `y` .column () `que dan la posición en una dimensión particular.
21
22
23 T> Para nuestro `QListView` la columna es siempre 0 y se puede ignorar, pero lo haría \
24 necesita usar esto para datos 2D en una vista de hoja de cálculo.
25
26 `role` es una bandera que indica el * tipo * de datos que solicita la vista. Esto se debe a que el método
27 `.data ()` en realidad tiene más responsabilidad que solo los datos principales. También maneja solicitudes
28 de información de estilo, información sobre herramientas, barras de estado, \
29 etc. - básicamente cualquier cosa que pueda ser informada por los propios datos.
30
31 El nombre de `Qt.DisplayRole` es un poco extraño, pero esto indica que * vie \ w * nos pide" por favor, dame
32 datos para mostrar ". Hay otros * roles * que \ h los `datos` pueden recibir para solicitudes de estilo o para
33 solicitar datos en formato" listo para editar ".
34
35
36 | Papel | Valor | Descripción
37 |-----|----|-----\
38 | ----- | ---- | ----- \
39 |-----| |
40 `Qt.DisplayRole` | `0` | Los datos clave que se van a representar en forma de texto \
41 . ([QString] (https://doc.qt.io/qt-5/qstring.html)) |
42 | `Qt.DecorationRole` | `1` | Los datos que se renderizarán como decoración en forma \ de un ícono.
43 ([QColor] (https://doc.qt.io/qt-5/qcolor.html), [QIcon] (https: / \ /doc.qt.io/qt-5/qicon.html) o [QPixmap ]
44 (https://doc.qt.io/qt-5/q pixmap.html)) \ |
45
46 | `Qt.EditRole` | `2` | Los datos en una forma adecuada para editar en un e \
47 ditor. ([QString] (https://doc.qt.io/qt-5/qstring.html)) |
48 | `Qt.ToolTipRole` | `3` | Los datos que se muestran en el artículo 's información sobre herramientas. ([QSt\
49 anillo] (https://doc.qt.io/qt-5/qstring.html)) |
50 | `Qt.StatusTipRole` | `4` | Los datos mostrados en la barra de estado. ([QString] (https://\
51 doc.qt.io/qt-5/qstring.html)) |
```

```
52 | `Qt.WhatsThisRole` | `5`           | Los datos mostrados por el objeto en "Qué es esto\  
53 ? " modo. ([QString] (https://doc.qt.io/qt-5/qstring.html) /  
54 | `Qt.SizeHintRole` | `13` | La sugerencia de tamaño por el artículo que se suministrará d a las  
55 vistas. ([QSize] (https://doc.qt.io/qt-5/qsize.html) /  
56  
57 Para obtener una lista completa de *roles* que puede recibir ver [el Qt ItemDataR\documentación ole]  
58 (https://doc.qt.io/qt-5/qt.html#ItemDataRole-enum). Nuestro todo lla lista solo estará usando  
59 `Qt.DisplayRole` y `Qt.DecorationRole`.  
60  
61 # ## Implementación básica  
62  
63 A continuación se muestra la aplicación de código auxiliar básica necesaria para cargar la interfaz de usuario y mostrarla. Nosotros'll  
64 agregue nuestro código de modelo y lógica de aplicación a esta base.
```

sesenta y cinco

```
66 `` pitón  
67 importar sys  
68 desde PyQt5 importar QtCore, QtGui, QtWidgets, uic desde  
69 PyQt5.QtCore importar Qt  
70  
71  
72 qt_creator_file = "mainwindow.ui"  
73 Ui_MainWindow, QtBaseClass = uic.loadUiType (qt_creator_file)  
74  
75  
76 clase TodoModel (QtCore.QAbstractListModel):  
77     def __init __ (self, * args, todos = None, ** kwargs):  
78         super (TodoModel, self) .__ init __ (* args, ** kwargs) self.todos =  
79         todos o []  
80  
81     def data (self, index, role):  
82         if role == Qt.DisplayRole:  
83             status, text = self.todos [index.row ()] return text  
84  
85  
86     def rowCount (self, index):  
87         return len (self.todos)  
88  
89  
90 clase MainWindow (QtWidgets.QMainWindow, Ui_MainWindow):
```

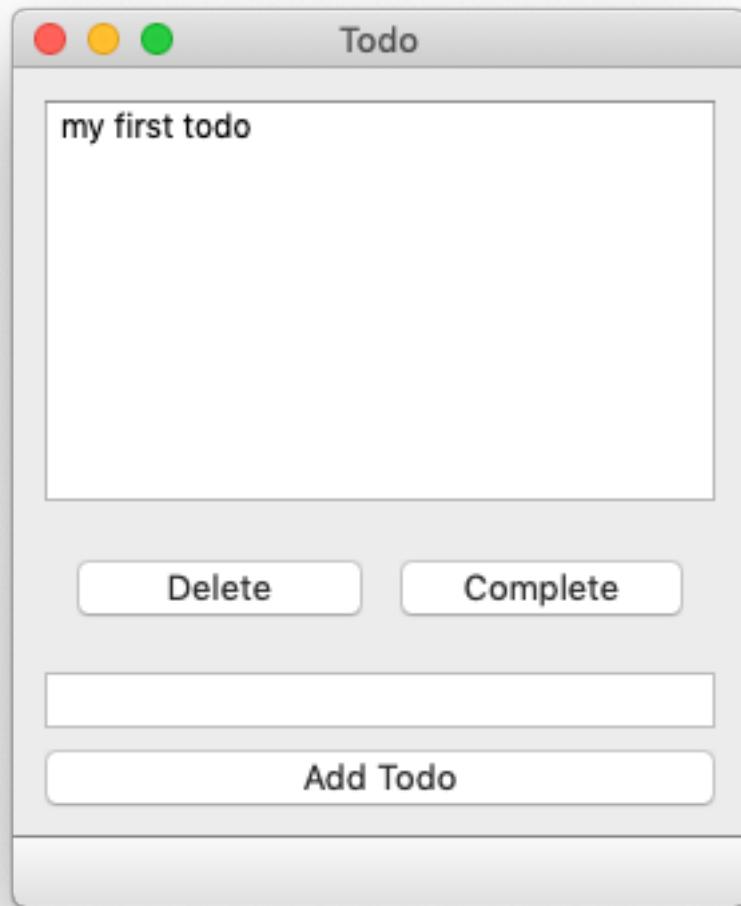
```
91     def __en si mismo):
92         QtWidgets.QMainWindow.__init__(self)
93         Ui_MainWindow.__init__(self)
94         self.setupUi(self)
95         self.model = TodoModel()
96         self.todoView.setModel(self.model)
97
98
99     aplicación = QtWidgets.QApplication(sys.argv) =
100    ventana = MainWindow()
101    window.show()
102    app.exec_()
```

Definimos nuestro TodoModel como antes, e inicialice el Ventana principal objeto. En el __en si mismo para MainWindow creamos una instancia de nuestro modelo de tareas pendientes y configuramos este modelo en el todo_view. Guarde este archivo comotodo.py y ejecutarlo con -

```
1 python3 todo.py
```

Si bien aún no hay mucho que ver, el QListView y nuestro modelo realmente está funcionando; si agrega algunos datos predeterminados, verá que aparecen en la lista.

```
1 uno mismo.modelo = TodoModel(todos=[(Falso, 'mi primera tarea')])
```



QListView que muestra un elemento de tarea codificado de forma rígida

Puede seguir agregando elementos manualmente como este y aparecerán en orden en el QListWidget. A continuación, haremos posible agregar elementos desde dentro de la aplicación.

Primero cree un nuevo método en el Ventana principal llamado `add_todo`. Esta es nuestra devolución de llamada

que se encargará de agregar el texto actual de la entrada como una nueva tarea. Conecte este método aladdButton.pressed señal al final de la `_en eso_ cuadra`.

```

1 clase Ventana principal(QtWidgets.QMainWindow,      Ui_MainWindow):
2     def __init__(self):
3         QtWidgets.QMainWindow.__init__(self)
4         Ui_MainWindow.__init__(self)
5         self.modelo = TodoModel()
6         self.todoView.setModel(self.modelo)
7         self.addButton.clicked.connect(self.agregar)
8
9         # Conecte el botón.
10        self.addButton.pressed.connect(self.agregar)
11
12    def agregar(self):
13        """ """
14
15        Agregue un elemento a nuestra lista de tareas pendientes, obteniendo el texto del QLineEdit.todo
16        Editar
17        y luego limpiarlo. """
18
19        texto = self.todoEdit.text() if texto: #
20        # No agregue cadenas vacías.
21        # Acceda a la lista a través del modelo.
22        self.modelo.todos.adjuntar(False, texto)
23        # Activar actualización.
24        self.modelo.layoutChanged.emit()
25        # Vaciar la entrada
26        self.todoEdit.setText("")
```

En el `add_todo` Bloquear nota la línea `self.modelo.layoutChanged.emit()`. Aquí estamos emitiendo una señal `modelo.layoutChanged` para que la vista sepa que el *form*ade los datos ha sido alterado. Esto desencadena una actualización de la totalidad de la vista. Si omite esta línea, la tarea aún se agregará pero la QListWidget no se actualizará.

Si se modifican los datos justos, pero el número de filas / columnas no se ve afectado, puede utilizar el `dataChanged()` señal para informar a Qt sobre esto. Esta también es una ubicación superior izquierda e inferior derecha en los datos, para evitar volver a dibujar la vista completa.

Conectando las otras acciones

Ahora podemos conectar el resto de las señales del botón y agregar funciones de ayuda para realizar la *Eliminar* y *completo* operaciones. Agregamos las señales de los botones al en eso bloquear como antes.

```
1 uno mismo.addButton.presionado.conectar(unomismo.agregar)uno mismo.  
2 deleteButton.presionado.conectar(unomismo.Eliminar)uno mismo.  
3 completeButton.presionado.conectar(unomismo.completo)
```

Luego defina un nuevo Eliminar método de la siguiente manera -

```
1 def Eliminar(unomismo):  
2     índices = unomismo.todoView.selectedIndexes ()si  
3         índices:  
4             # Índices es una lista de un solo elemento en modo de selección única.índice =  
5             índices [0]  
6             # Elimina el artículo y actualiza.del uno  
7             mismo.modelo.todos [índice.hilera()]uno  
8             mismo.modelo.layoutChanged.emitir()  
9             # Borre la selección (ya que ya no es válida).uno mismo.todoView.  
10            selección clara()
```

Usamos self.todoView.selectedIndexes para obtener los índices (en realidad, una lista de un solo elemento, ya que estamos en el modo de selección única) y luego use el .hilera() como un índice en nuestra lista de todos en nuestro modelo. Eliminamos el elemento indexado usando Python del operador, y luego activar un layoutChanged señal porque se ha modificado la forma de los datos.

Finalmente, borramos la selección activa ya que el elemento con el que se relaciona ahora puede estar fuera de los límites (si había seleccionado el último elemento).



Puede intentar hacer esto más inteligente y seleccionar el último elemento de la lista.

los completo el método se ve así:

```
1     def completo(uno mismo):
2         índices = uno mismo.todoView.selectedIndexes ()si
3             índices:
4                 índice  =  índices [0]
5                 hilera =  índice.hilera()
6                 estado, texto = uno mismo.modelo.todos [fila]
7                 uno mismo.modelo.todos [fila] = (Cierto, texto)
8                 #.dataChanged toma la parte superior izquierda y la inferior derecha, que son iguales
9                 # para una sola selección.
10                uno mismo.modelo.dataChanged.emitir (índice, índice)
11                # Borre la selección (ya que ya no es válida).uno mismo.todoView.
12                selección clara()
```

Esto usa la misma indexación que para eliminar, pero esta vez obtenemos el elemento del modelo .todos lista y luego reemplace el estado con Cierto.



Tenemos que hacer esto de buscar y reemplazar, ya que nuestros datos se almacenan como tuplas de Python que no se pueden modificar.

The key difference here vs. standard QtWidgets is that we make changes directly to our data, and we simply need to notify Qt that something has changed; the update of the state of the widget is handled automatically.

Usando Qt.DecorationRole

If you run the application now, you should find that adding and removing both work, but while completing items works, there is no indication of this in the view. We need to update our model to provide the view with an indicator to show when an article is complete. The updated model is shown below.

```
1 garrapata = QtGui.QImage ('tick.png')
2
3
4 clase TodoModel(QtCore.QAbstractListModel):
5     def __init__(self, *args, todos=Ninguno, **kwargs):
6         super(TodoModel, self).__init__(*args, **kwargs)
7         argumentos, self.todos = todos or []
8
9     def datos(self, index, role):
10        if role == Qt.DisplayRole:
11            _, texto = self.todos[index.row()]
12            return texto
13
14        if role == Qt.DecorationRole:
15            estado, _ = self.todos[index.row()]
16            if estado:
17                return garrapata
18
19    def numero_de_filas(self, index):
20        return len(self.todos)
```

Estamos usando un ícono de marca garrapata.png para indicar elementos completados, que cargamos en un QImage objeto nombrado garrapata. En el modelo hemos implementado un controlador para el Qt.DecorationRole que devuelve el ícono de marca para las filas que es estado es Certo (por completo).

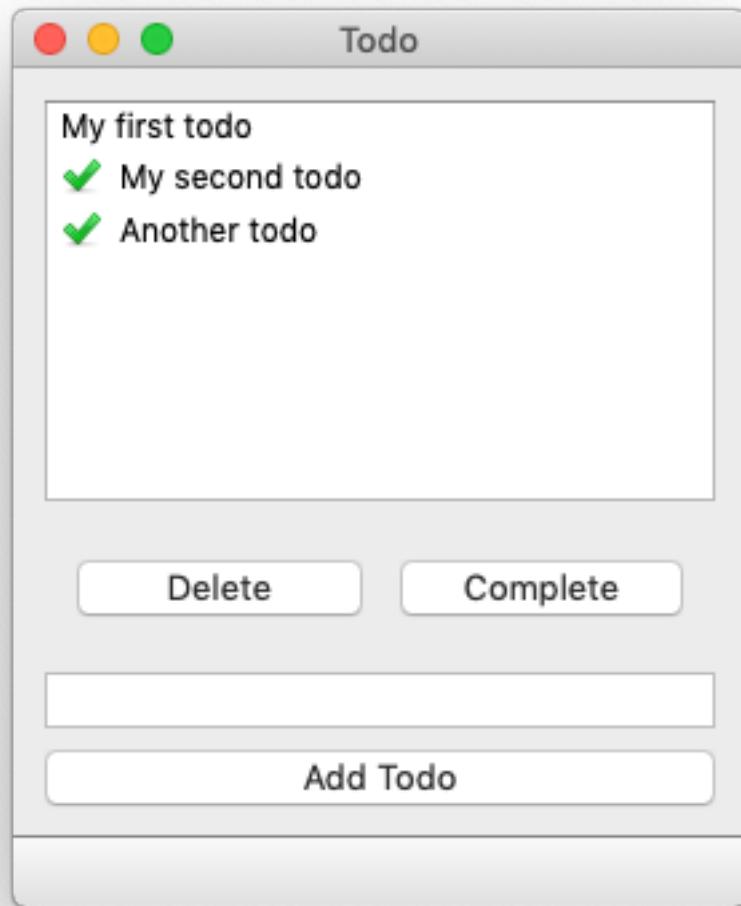


El ícono que estoy usando está tomado del conjunto de Fuga de [p.yusukekamiyamane](#)



En lugar de un ícono, también puede devolver un color, p. Ej. QtGui.QColor ('verde') que se dibujará como un cuadrado sólido.

Al ejecutar la aplicación, ahora debería poder marcar elementos como completos.



Todos completos

Un almacén de datos persistente

Nuestra aplicación de tareas pendientes funciona bien, pero tiene un defecto fatal: olvida tus tareas pendientes tan pronto como cierras la aplicación. una mala idea.

La solución es implementar algún tipo de almacén de datos persistente. El enfoque más simple es un almacén de archivos simple, donde cargamos elementos desde un archivo JSON o Pickle al inicio y escribimos los cambios.

Para hacer esto, definimos dos nuevos métodos en nuestro Ventana principal clase - carga y ahorrar. Estos cargan datos desde un nombre de archivo JSONdata.json (si existe, ignorando el error si no existe) para self.modelo.todos y escribe la corriente self.modelo.todos en el mismo archivo, respectivamente.

```
1  def carga(uno mismo):
2      tratar:
3          con abierto('data.json', 'r') como F:
4              uno mismo.modelo.todos = json.carga (f)
5      excepto Excepción:
6          aprobar
7
8  def ahorrar(uno mismo):
9      con abierto('data.json', 'w') como F:
10         datos = json.verteadero(uno mismo.modelo.todos, f)
```

Para persistir en los cambios en los datos, necesitamos agregar el .ahorrar() manejador al final de cualquier método que modifique los datos, y el .carga() manejador de la __en eso__ bloque después de que se haya creado el modelo.

El código final se ve así:

```
1 importar sys
2 importar json
3 de PyQt5 importar QtCore, QtGui, QtWidgets, uicde
4 PyQt5.QtCore importar Qt
5
6
7 qt_creator_file = "mainwindow.ui"
8 Ui_MainWindow, QtBaseClass = uic.loadUiType (qt_creator_file) marca = QtGui.
9 QImage ('tick.png')
10
11
12 clase TodoModel(QtCore.QAbstractListModel):
13     def __init__(self, *args, todos=Ninguno, **kwargs):
14         super(TodoModel, self).__init__(*args, **kwargs)
15         argumentos, self.todos = todos or []
16
17     def datos(self, index, role):
18         if role == Qt.DisplayRole:
19             _, texto = self.todos [index.row()]
20             return texto
21
22         if role == Qt.DecorationRole:
23             estado, _ = self.todos [index.row()]
24             if estado:
25                 return "check"
26
27     def numero_de_filas(self, index):
28         return len(self.todos)
29
30
31 clase Ventana_principal(QtWidgets.QMainWindow,      Ui_MainWindow):
32     def __init__(self):
33         QtWidgets.QMainWindow.__init__(self)
34         Ui_MainWindow.__init__(self)
35         self.modelo = TodoModel()
36         self.carga()
37
38         self.todoView.setModel (self.modelo)
39         addButton.presionado.conectar(self.agregar)
```

```
40         uno mismo.deleteButton.presionado.conectar(unomismo.Eliminar)uno
41         mismo.completeButton.presionado.conectar(unomismo.completo)
42
43     def agregar(unomismo):
44         """
45             Agregue un elemento a nuestra lista de tareas pendientes, obteniendo el texto del QLineEdit .todo |
46             Editar
47             y luego limpiarlo. """
48
49         texto = unomismo.todoEditar.texto()si texto: #
50             No agregue cadenas vacías.
51                 # Acceda a la lista a través del modelo.unomismo.
52                 modelo.todos.adjuntar((Falso, texto))
53                 # Activar actualización.
54                 unomismo.modelo.layoutChanged.emitir()
55                 # Vaciar la entradaunomismo.
56                 todoEditar.setText("")unomismo.
57                 ahorrar()
58
59     def Eliminar(unomismo):
60         índices = unomismo.todoView.selectedIndexes ()si
61             índices:
62                 # Índices es una lista de un solo elemento en modo de selección única.índice =
63                 índices [0]
64                 # Elimina el artículo y actualiza.del uno
65                 mismo.modelo.todos [índice.hilera()]uno
66                 mismo.modelo.layoutChanged.emitir()
67                 # Borre la selección (ya que ya no es válida).unomismo.todoView.
68                 selección clara()unomismo.ahorrar()
69
70
71     def completo(unomismo):
72         índices = unomismo.todoView.selectedIndexes ()si
73             índices:
74                 índice = índices [0]
75                 hilera = índice.hilera()
76                 estado, texto = unomismo.modelo.todos [fila]
77                 unomismo.modelo.todos [fila] = (Cierto, texto)
78                 #.dataChanged toma la parte superior izquierda y la inferior derecha, que son iguales
```

```
79         # para una sola selección.
80         uno mismo.modelo.dataChanged.emitir (índice, índice)
81         # Borre la selección (ya que ya no es válida).uno mismo.todoView.
82         selección clara(uno mismo.ahorrar())
83
84
85     def carga(uno mismo):
86         tratar:
87             con abierto('data.db', 'r') como F:
88                 uno mismo.modelo.todos = json.carga (f)
89             excepto Excepción:
90                 aprobar
91
92     def ahorrar(uno mismo):
93         con abierto('data.db', 'w') como F:
94             datos = json.verteadero(uno mismo.modelo.todos, f)
95
96
97     aplicación = QtWidgets.QApplication (sys.argv)
98     ventana = Ventana principal()
99     ventana.show()
100    aplicación.exec_()
```

Si los datos de su aplicación tienen el potencial de volverse más grandes o más complejos, es posible que prefiera utilizar una base de datos real para almacenarlos. En este caso, el modelo ajustará la interfaz a la base de datos y la consultará directamente para mostrar los datos.

Para ver otro ejemplo interesante de QListView, vea este ejemplo de aplicación de reproductor de medios: <https://www.learnpyqt.com/apps/failamp-multimedia-player/> Utiliza QMediaPlaylist integrado en Qt como almacén de datos, con el contenido mostrado en un QListview .

Subprocesos múltiples

A medida que empiece a crear aplicaciones más complejas, puede encontrar problemas en los que las tareas de larga duración "bloquean" su interfaz.

El ciclo de eventos comenzó llamando `.exec_()` en tu `QApplication` objeto y se ejecuta dentro del mismo hilo que su código Python. El hilo que ejecuta este bucle de eventos, comúnmente conocido como el *Hilo de la GUI* - también maneja toda la comunicación de la ventana con el sistema operativo del host.

De forma predeterminada, cualquier ejecución desencadenada por el bucle de eventos también se ejecutará sincrónicamente dentro de este hilo. En la práctica, esto significa que cada vez que su aplicación PyQt *pasahaciendo algo* en su código, la comunicación de ventana y la interacción GUI están congeladas.

Si lo que está haciendo es simple y devuelve el control al bucle de la GUI rápidamente, esta congelación será imperceptible para el usuario. Sin embargo, si necesita realizar tareas de ejecución más largas, por ejemplo, abrir / escribir un archivo grande, descargar algunos datos o renderizar alguna imagen compleja, habrá problemas. Para su usuario, la aplicación parecerá no responder (porque lo es). Debido a que su aplicación ya no se comunica con el sistema operativo, en MacOS X, si hace clic en su aplicación, verá la rueda giratoria de la muerte. Y, *nadie* quiere eso.

La solución es simple: saque su trabajo del *GUI* hilo y en otro. PyQt (a través de Qt) proporciona una interfaz sencilla para hacer exactamente eso.

Preparación

Para demostrar la ejecución de subprocessos múltiples, necesitamos una aplicación con la que trabajar. A continuación se muestra una aplicación de código auxiliar mínimo para PyQt que nos permitirá demostrar el subprocesso múltiple y ver el resultado en acción. Simplemente copie y pegue esto en un nuevo archivo y guárdelo con un nombre de archivo apropiado como `multithread.py`. El resto del código se agregará a este archivo (también hay un ejemplo completo de trabajo en la parte inferior si está impaciente).

```
1 de PyQt5.QtGui importar *de
2 PyQt5.QtWidgets importar *de
3 PyQt5.QtCore importar *
4
5 importar tiempo
6
7
8 clase Ventana_principal(QMainWindow):
9     def __init__(self, *args, **kwargs):
10         super(Ventana_principal, self).__init__(*args, **kwargs)
11
12         self.encimera = 0
13
14         diseño = QVBoxLayout()
15
16         self.setWindowTitle("Comienzo")
17         b = QPushButton("¡PELIGRO!")
18         presionado.connect(self.Oh_no)
19
20         diseño.addWidget(self.l)
21         diseño.addWidget(b)
22
23         w = QWidget()
24         w.setLayout(diseño)
25
26         self.setCentralWidget(w)
27
28         self.show()
29
30         self.Temporizador = QTimer()
31         self.Temporizador.setInterval(1000)
32
33         self.Temporizador.timeout.connect(self.Oh_no)
34
35     def Oh_no(self):
36         tiempo.sleep(5)
37
38     def temporizador_recurrente(self):
39         self.encimera += 1
```

```
40         uno mismo.l.setText ("Encimera: %D" % uno mismo.encimera)
41
42
43     aplicación = QApplication []
44     ventana = Ventana principal()
45     aplicación.exec_()
```

Ejecute el archivo como de costumbre:

```
1 python3 multithread.py
```

Debería ver una ventana de demostración con un número ascendente. Esto se genera por un tiempo recurrente simple, disparando una vez por segundo. Piense en esto como nuestro *indicador de bucle de eventos*, una forma sencilla de hacernos saber que nuestra aplicación está funcionando normalmente. También hay un botón con la palabra "*¡PELIGRO!*".

Empujalo.



Presione el botón

Notarás que cada vez que presionas el botón, el contador deja de funcionar y tu aplicación se congela por completo. En Windows, es posible que vea que la ventana se pone pálida, lo que indica que no responde, mientras que en una Mac es posible que vea la rueca de la muerte.

El enfoque tonto

Lo que aparece como *interfaz congelada* de hecho, es causado por el bloqueo del bucle de eventos Qt para no procesar (y responder) eventos de ventana. Sus clics en la ventana aún están registrados por el sistema operativo host y enviados a su aplicación, pero debido a que está incluido en su gran cantidad de código (tiempo de dormir), no puede aceptarlos ni reaccionar ante ellos. Su aplicación no responde al sistema operativo y lo interpreta como una aplicación congelada.

La forma más sencilla de evitar esto es aceptar eventos desde dentro de su código. Esta

permite que Qt continúe respondiendo al sistema operativo host y su aplicación seguirá respondiendo. Puede hacer esto fácilmente mediante el uso de `estática.processEvents ()` función en el `QApplication` clase. Simplemente agregue una línea como la siguiente, en algún lugar de su bloque de código de larga ejecución:

```
1 QApplication.processEvents ()
```

Si podemos tomar nuestro largo plazo tiempo de dormir código y dividirlo en varios pasos, podemos insertar `.processEvents` entre. El código para esto sería:

```
1 def Oh no(uno mismo):
2     por norte en distancia(5):
3         QApplication.processEvents () tiempo.
4         dormir(1)
```

Ahora, cuando presiona el botón, su código se ingresa como antes. Sin embargo, ahora `QApplication.processEvents ()` intermitentemente pasa el control a Qt y le permite responder a los eventos del sistema operativo de forma normal. Qt ahora aceptará eventos y *manejarlos* antes de volver a ejecutar el resto de su código.

Esto funciona, pero es horrible por algunas razones:

1. Cuando devuelve el control a Qt, su código ya no se está ejecutando. Esto significa que cualquier cosa de larga duración que intentes hacer, tomará *más extenso*. Eso definitivamente no es lo que quieras.
2. Solo puede hacer esto si su tarea de ejecución prolongada se divide en varios pasos de ejecución corta. Si aumenta la duración ahora de dormir (30) y vuelve a ejecutarlo, verá que la interfaz de usuario se congela nuevamente. El paso individual es ahora tan largo que el control no se devuelve con suficiente frecuencia.
3. Procesamiento de eventos fuera del bucle de eventos principal (`app.exec_()`) hace que su aplicación se ramifique en código de manejo (por ejemplo, para ranuras activadas o eventos) mientras está dentro de su ciclo. Si su código depende de / responde a un estado externo, esto puede causar un comportamiento indefinido. El siguiente código demuestra esto en acción:

```
1 de PyQt5.QtGui importar *de
2 PyQt5.QtWidgets importar *de
3 PyQt5.QtCore importar *
4
5 importar tiempo
6
7
8 clase Ventana_principal(QMainWindow):
9     def __init__(self, *args, **kwargs):
10         super(Ventana_principal, self).__init__(*args, **kwargs)
11
12         self.encimera = 0
13
14         diseño = QVBoxLayout()
15
16         self.l = QLabel("Comienzo")
17         b = QPushButton("¡PELIGRO!")
18         presionado.connect(self.l.change_message)
19
20         c = QPushButton("?")
21         c.clicked.connect(self.change_message)
22
23         diseño.addWidget(self.l)
24         diseño.addWidget(b)
25
26         diseño.addWidget(c)
27
28         w = QWidget()
29         w.setLayout(diseño)
30
31         self.setCentralWidget(w)
32
33         self.show()
34
35     def change_message(self):
36         self.mensaje = "OH NO"
37
38     def Oh_no(self):
39         self.mensaje = "Presionado"
```

```
40
41     por norte en distancia(100):
42         tiempo.dormir(0,1)
43         uno mismo.l.setText (uno mismo.
44             mensaje) QApplication.processEvents ()
45
46
47     aplicación = QApplication []
48     ventana = Ventana principal()
49     aplicación.exec_()
```

Si ejecuta este código, verá el contador como antes. Pulsando "¡PELIGRO!" cambiará el texto mostrado a "Presionado", como se define en el punto de entrada a la otra función. Sin embargo, si presiona "?" botón mientras no aún se está ejecutando, verá que el mensaje cambia. El estado se está cambiando desde fuera de su bucle.

Este es un ejemplo de juguete. Sin embargo, si tiene varios procesos de larga ejecución dentro de su aplicación, con cada llamada `QApplication.processEvents ()` para que las cosas sigan funcionando, el comportamiento de su aplicación puede ser impredecible.

Hilos y procesos

Si da un paso atrás y piensa en lo que quiere que suceda en su aplicación, probablemente se pueda resumir con "cosas que suceden al mismo tiempo que suceden otras cosas".

Hay dos enfoques principales para ejecutar tareas independientes dentro de una aplicación PyQt: *hilos* y *procesos*.

Hilos comparten el mismo espacio de memoria, por lo que se inician rápidamente y consumen recursos mínimos. La memoria compartida hace que sea trivial pasar datos entre subprocesos, sin embargo, leer / escribir memoria de diferentes subprocesos puede conducir a condiciones de carrera o segfaults. En Python existe el problema adicional de que varios subprocesos están vinculados por el mismo bloqueo de intérprete global (GIL), lo que significa que el código Python que no libera GIL solo se puede ejecutar en un subproceso a la vez. Sin embargo, este no es un problema importante con PyQt, donde la mayor parte del tiempo se pasa fuera de Python.

Procesos usan un espacio de memoria separado (y un intérprete de Python completamente separado). Esto evita cualquier problema potencial con el GIL, pero a costa de

tiempos de inicio más lentos, mayor sobrecarga de memoria y complejidad en el envío / recepción de datos.

En aras de la simplicidad, generalmente tiene sentido usar subprocesos, a menos que tenga una buena razón para usar procesos (consulte [advertencias](#) más tarde). Los subprocesos en Qt se adaptan mejor a la ejecución y la comunicación con programas externos.

QRunnable y QThreadPool

Qt proporciona una interfaz muy simple para ejecutar trabajos en otros subprocesos, que se expone muy bien en PyQt. Esto se basa en dos clases:QRunnable y QThreadPool. El primero es el contenedor del trabajo que desea realizar, mientras que el segundo es el método por el cual pasa ese trabajo a hilos alternos.

Lo bueno de usar QThreadPool es que maneja las colas y la ejecución de los trabajadores por usted. Aparte de poner en cola los trabajos y recuperar los resultados, no hay mucho que hacer en absoluto.

Para definir una costumbre QRunnable puedes subclásificar la base QRunnable class, luego coloque el código que desea ejecutar dentro de la correr() método. La siguiente es una implementación de nuestro tiempo de dormir trabajo como QRunnable. Agregue el siguiente código amultithread.py, sobre el Ventana principal definición de clase.

```
1  clase Trabajador(QRunnable):
2      """
3          Hilo de trabajador
4      """
5
6      @pyqtSlot()
7      def correr(uno mismo):
8          """
9              Tu código va en esta función """
10
11         impresión("Hilo comienzo")
12         tiempo.dormir(5)
13         impresión("Hilo completo")
```

Ejecutar nuestra función en otro hilo es simplemente una cuestión de crear una instancia del Trabajador y luego pasarlo a nuestro QThreadPool instancia y se ejecutará automáticamente.

A continuación, agregue lo siguiente dentro del `_en eso_` block, para configurar nuestro grupo de subprocesos.

```
1 uno mismo.hilo de rosca = QThreadPool ()  
2 impresión("Múltiples subprocesos con máximo %D hilos" % uno mismo.hilo de rosca.maxThreadCount \ ())  
3
```

Finalmente, agregue las siguientes líneas a nuestro Oh no función.

```
1 def Oh no(uno mismo):  
2     trabajador = Trabajador()  
3     uno mismo.hilo de rosca.empezar (trabajador)
```

Ahora, al hacer clic en el botón se creará un trabajador para manejar el proceso (de larga duración) y convertirlo en otro hilo a través del grupo de hilos. Si no hay suficientes subprocesos disponibles para procesar los trabajadores entrantes, se pondrán en cola y se ejecutarán en orden en un momento posterior.

Pruébelo y verá que su aplicación ahora lo maneja golpeando el botón sin problemas.

Compruebe lo que sucede si presiona el botón varias veces. Deberías ver tus hilos ejecutados inmediatamente *hasta* el número informado por `.maxThreadCount`. Si presiona el botón nuevamente después de que ya haya este número de trabajadores activos, los trabajadores posteriores se pondrán en cola hasta que haya un hilo disponible.

Corredores extendidos

Si desea pasar datos personalizados a la función de ejecución, puede configurar su corredor para que tome *argumentos* o *palabras clave* y luego almacenar los datos en el propio corredor. Los datos serán accesibles mientras se ejecutan a través de `uno mismo` de tu QRunnable objeto.

De hecho, incluso puede aprovechar el hecho de que las funciones en Python son objetos y pasar la función para ejecutar en lugar de subclasicar cada vez. En el

después de la construcción, solo requerimos una Trabajador class para manejar todos nuestros trabajos de ejecución.

```
1 clase Trabajador(QRunnable):
2     """
3         Hilo de trabajador
4
5         :param fn: la función a ejecutar
6         :param args: Argumentos para poner a disposición del código de ejecución: param
7         kwargs: Argumentos de palabras clave para poner a disposición de la ejecución: código
8
9     :
10    """
11
12    def __en eso__(uno mismo, fn, *argumentos, **kwargs):
13        súper(Trabajador, uno mismo).__en eso__()
14        uno mismo.fn = fn
15        uno mismo.argumentos = argumentos
16        uno mismo.kwargs = kwargs
17
18    @pyqtSlot()
19    def correr(uno mismo):
20        """
21            Ejecute la función de corredor con self.args, self.kwargs pasados.
22
23        """
24        uno mismo.fn(argumentos, kwargs)
```

Ahora puede pasar cualquier función de Python y hacer que se ejecute en un hilo separado.

```
1 def ejecutar_esto_fn(uno mismo):
2     impresión("¡Hola!")
3
4 def Oh_no(uno mismo):
5     # Pasar la función a ejecutar
6     trabajador = Trabajador(uno mismo.ejecutar_esto_fn) # Cualquier otro argumento, kwargs se pasa t lo la
7     función de ejecución
8
9     # Ejecutar
10    hilo de rosca.empezar (trabajador)
```

Hilo IO

A veces es útil poder transmitir *estado* y *datos* de los trabajadores en ejecución. Esto podría incluir el resultado de los cálculos, las excepciones planteadas o el progreso continuo (piense en las barras de progreso). Qt proporciona *señales y ranuras* framework que le permite hacer precisamente eso y es seguro para subprocesos, lo que permite una comunicación segura directamente desde los subprocesos en ejecución a su interfaz gráfica de usuario. *Señales* permitirte .emitir valores, que luego se recogen en otra parte de su código por *espacio* funciones que se han vinculado con .conectar.

A continuación se muestra un simple `TrabajadorSeñales` clase definida para contener una serie de señales de ejemplo.



Las señales personalizadas solo se pueden definir en objetos derivados de `QObject`. Ya que `QRunnable` no se deriva de `QObject` no podemos definir las señales directamente. Una costumbre `QObject` retener las señales es la solución más simple.

```
1 importar rastrear, sys
2
3
4 clase TrabajadorSeñales(QObject):
5     """
6         Define las señales disponibles de un subproceso de trabajo en ejecución.
7
8     Las señales admitidas son:
9
10    finalizado
11        Sin datos
12
13    error
14        tupla `(exctype, valor, traceback.format_exc ())`
15
16        resultado
17            datos del `objeto` devueltos por el procesamiento, cualquier cosa
18
19        """
20    finalizado = pyqtSignal ()
21    error =    pyqtSignal (tupla)
22    resultado = pyqtSignal (objeto)
```

En este ejemplo, hemos definido 5 señales personalizadas:

1. *finalizado* señal, sin datos para indicar cuando la tarea está completa.
2. *error* señal que recibe una tupla de Excepción escribe, Excepción valor y seguimiento con formato.
3. *resultado* señal que recibe cualquier objeto escriba de la función ejecutada.

Es posible que no encuentre una necesidad para todas estas señales, pero se incluyen para dar una indicación de lo que es posible. En el siguiente código vamos a implementar una tarea de larga duración que hace uso de estas señales para proporcionar información útil al usuario.

```
1 clase Trabajador(QRunnable):
2     """
3         Hilo de trabajador
4
5         Se hereda de QRunnable a la configuración, las señales y la finalización del hilo del trabajador del controlador.
6
7         :param callback: la función de devolución de llamada que se ejecutará en este
8             trabajador: hilo. Args y
9                 los kwargs pasarán al corredor.
10        :escriba devolución de llamada: función
11        :param args: Argumentos para pasar a la función de devolución de llamada:
12        param kwargs: Palabras clave para pasar a la función de devolución de llamada:
13
14        """
15
16
17     def __init__(self, fn, *args, **kwargs):
18         super(Trabajador, self).__init__()
19         # Argumentos del constructor de la tienda (reutilizados para el procesamiento)
20         self.fn = fn
21         self.args = args
22         self.kwargs = kwargs
23         self.signals = WorkerSignals()
24
25     @pyqtSlot()
26     def run(self):
27         """
28             Inicialice la función de corredor con args, kwargs pasados.
29
30         # Recupere args / kwargs aquí, y procesamiento de fuego usándolos tratar:
31
32         resultado = self.fn(
33             *self.args, **self.kwargs,
34             estado=self.signals.estado,
35             progreso=self.signals.progreso,
36         )
37     except:
38         rastrear.print_exc()
39         exctype, valor = sys.exc_info()[:2]
```

```

40         uno mismo.señales.error.emitir ((exctype,      valor, rastreo.format_exc ()))
41     demás:
42         uno mismo.señales.resultado.emitir (resultado)   # Devuelve el resultado del proceso !
43 cantar
44     finalmente:
45         uno mismo.señales.finalizado.emitir() # Hecho

```

Puede conectar sus propias funciones de controlador a estas señales para recibir una notificación de la finalización (o el resultado) de los subprocesos.

```

1 def ejecutar_esto_fn(uno mismo):
2     por norte en distancia(0, 5):
3         tiempo.dormir(1)
4     regreso "Hecho."
5
6 def print_output(uno mismo, s):
7     impresión(s)
8
9 def thread_complete(uno mismo):
10    impresión("¡HILO COMPLETO!")
11
12 def Oh_no(uno mismo):
13     # Pasar la función a ejecutar
14     trabajador = Trabajador(uno mismo.ejecutar_esto_fn) # Cualquier otro argumento, kwargs se pasa t lo la
15     función de ejecución
16     trabajador.señales.resultado.conectar(uno mismo.print_output)
17     trabajador.señales.finalizado.conectar(uno mismo.thread_complete)
18
19     # Ejecutar
20     uno mismo.hilo de rosca.empezar (trabajador)

```

QRunnable Ejemplos de

Las características de QRunnables descrito se puede utilizar para construir corredores adecuados para una variedad de tareas. A continuación, se muestran algunos ejemplos de cómo construir corredores y aplicaciones para usarlos en varios escenarios diferentes.

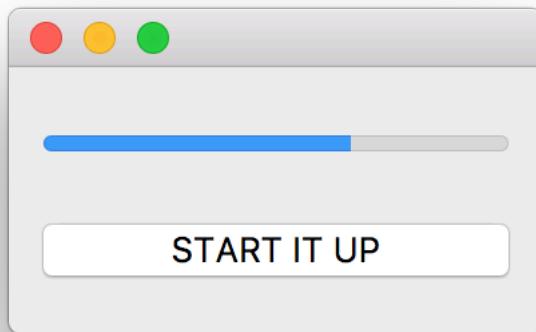
El actualizador

A menudo, desea recibir información sobre el progreso de los hilos de ejecución prolongada. Esto se puede hacer fácilmente definiendo una señal en elTrabajadorSeñales objeto, a través del cual se pasa un número que indica el% de finalización. El siguiente ejemplo usa esto para actualizar una barra de progreso en ejecución.

```
1  de PyQt5.QtGui importar *
2  PyQt5.QtWidgets importar *
3  PyQt5.QtCore importar *
4
5  importar tiempo
6
7
8  clase TrabajadorSeñales(QObject):
9      """
10         Define las señales disponibles de un subproceso de trabajo en ejecución.
11
12     Progreso
13         int progreso completo, de 0-100
14         """
15     Progreso = pyqtSignal (En t)
16
17
18  clase Trabajador(QRunnable):
19      """
20         Hilo de trabajador
21
22         Hereda de QRunnable a la configuración de subprocesos, señales y
23             envolver.
24         """
25
26     def __en eso__(uno mismo):
27         súper(Trabajador, uno mismo).__en eso__()
28
29         uno mismo.señales = WorkerSignals ()
30
31     @pyqtSlot()
```

```
32     def correr(uno mismo):
33         total_n = 1000
34         por norte en distancia(total_n):
35             progress_pc = En t(100 *flotador(norte)/total_n)uno      # Progreso 0-100% como int
36             mismo.señales.Progreso.emitir (progress_pc) tiempo
37             .dormir(0,01)
38
39
40 clase Ventana principal(QMainWindow):
41
42
43     def __en eso__(uno mismo, *argumentos, **kwargs):
44         super(Ventana principal, uno mismo).__en eso__(*argumentos, **kwargs)
45
46
47     diseño = QVBoxLayout ()
48
49     uno mismo.bar = QProgressBar ()
50
51     botón = QPushButton ("INICIO") botón.
52     presionado.conectar(uno mismo.ejecutar)
53
54     diseño.addWidget (uno mismo.
55     barra) diseño.addWidget (botón)
56
57     w = QWidget ()
58     w.setLayout (diseño)
59
60     uno mismo.setCentralWidget (w)
61
62     uno mismo.show()
63
64     uno mismo.hilo de rosca = QThreadPool ()
sesenta y cinco
65     impresión("Múltiples subprocesos con máximo %D hilos" % uno mismo.hilo de rosca.maxThr \
66     eadCount ())
67
68     def ejecutar(uno mismo):
69         trabajador = Trabajador()
70         trabajador.señales.Progreso.conectar(uno mismo.progreso de actualización)
```

```
71  
72     # Ejecutar  
73     uno mismo.hilo de rosca.empezar (trabajador)  
74  
75 def progreso de actualización(uno mismo, Progreso):  
76     uno mismo.bar.setValue (progreso)  
77  
78  
79  
80 aplicación = QApplication ([])  
81 ventana = Ventana principal()  
82 aplicación.exec_()
```



Barra de progreso que muestra el progreso actual de un trabajador de larga duración.



Si desea admitir corredores de variables pasando una función, no tendrá acceso a la devolución de llamada de progreso. Puede evitar esto pasándolo directamente a su función. Ver el *Registrador genérico* descripción a continuación.

El maderero

El subprocesamiento es una buena opción cuando necesita ejecutar operaciones de E / S y recibir los datos de ellas. Esto puede significar interactuar con API o sitios web, o recibir datos en serie del hardware.

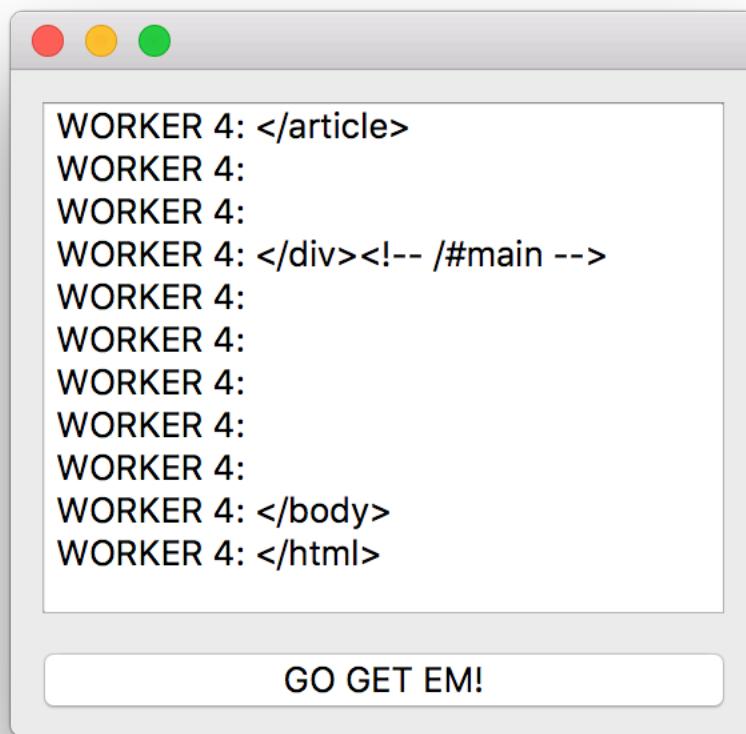
En este ejemplo, creamos múltiples corredores, cada uno enviando sus datos en vivo, etiquetados con su propio identificador. Esto permite que los datos devueltos se asocien con el corredor del que provienen y se envíen a la salida correcta.

```
1  de PyQt5.QtGui importar *de
2  PyQt5.QtWidgets importar *de
3  PyQt5.QtCore importar *
4
5  importar peticiones
6
7  clase TrabajadorSeñales(QObject):
8      """
9          Define las señales disponibles de un subproceso de trabajo en ejecución.
10
11         datos
12             tupla de (identificador, datos)
13         """
14
15         datos = pyqtSignal (tupla)
16
17
18         clase Trabajador(QRunnable):
19             """
20                 Hilo de trabajador
21
22                 Se hereda de QRunnable a la configuración, las señales y la finalización del hilo del trabajador
23                 del controlador.
24
25                 : param id: el id de este trabajador: param
26                 url: la url para recuperar "
27
28         def __en eso__(uno mismo, identificación, url):
```

```
29         súper(Trabajador, uno mismo).__en eso__()
30         uno mismo.identificación = identificación
31         uno mismo.url = url
32
33         uno mismo.señales = WorkerSignals()
34
35     @pyqtSlot()
36     def correr(uno mismo):
37         r = peticiones.obtener(uno mismo.url)
38
39         por línea en r.texto.splitlines():
40             uno mismo.señales.datos.emitir((uno mismo.identificación, línea))
41
42
43     clase Ventana principal(QMainWindow):
44
45
46         def __en eso__(uno mismo, *argumentos, **kargs):
47             súper(Ventana principal, uno mismo).__en eso__(*argumentos, *kargs)
48
49             uno mismo.urls =
50                 "http://www.example.com",'https://www.mfitzp.com','https://www.google.com','https://
51                 www.udemy.com/create-simple-gui-applications-with-python-a\
52
53
54             nd-qt / ',
55                 'https://books.mfitzp.com/create-simple-gui-applications/'
56             ]
57
58             diseño = QVBoxLayout()
59
60             uno mismo.texto = QPlainTextEdit()
61             uno mismo.texto.setReadOnly(Cierto)
62
63             botón = QPushButton ("¡VE A BUSCARLOS!")
64             botón.presionado.conectar(uno mismo.ejecutar)
65
66             diseño.addWidget (uno mismo.texto)
67             diseño.addWidget (botón)
```

```
68
69     w = QWidget ()
70     w.setLayout (diseño)
71
72     uno mismo.setCentralWidget (w)
73
74     uno mismo.show()
75
76     uno mismo.hilo de rosca = QThreadPool ()
77     impresión("Múltiples subprocesos con máximo %D hilos" % uno mismo.hilo de rosca.maxThr \
78     eadCount ())
79
80
81     def ejecutar(uno mismo):
82         por n, url en enumerar(uno mismo.urls):
83             trabajador = Trabajador (n, url)
84             trabajador.señales.datos.conectar(uno mismo.display_output)
85
86             # Ejecutar
87             uno mismo.hilo de rosca.empezar (trabajador)
88
89
90     def display_output(uno mismo, datos):
91         identificación, s = datos
92         uno mismo.texto.appendPlainText ("TRABAJADOR %D: %s" % (identificación, s))
93
94
95
96     aplicación = QApplication []
97     ventana = Ventana principal()
98     aplicación.exec_()
```

Si ejecuta este ejemplo y presiona el botón, verá la salida HTML de varios sitios web, precedida por la ID de trabajador que los recupera. Tenga en cuenta que la salida de diferentes trabajadores está intercalada.



Registrar la salida de varios trabajadores en la ventana principal.

los tupla Por supuesto, es opcional, puede devolver cadenas desnudas si solo tiene un corredor, o no necesita salidas asociadas con una fuente. También es posible enviar unbytesting, o cualquier otro tipo de datos, configurando las señales apropiadamente.

El genérico

Si tiene que ejecutar varios trabajos similares o los corredores no tienen requisitos específicos, es posible que necesite un corredor genérico. Pase la función para no recibir resultados, errores y progreso.

A continuación, se muestra un ejemplo de trabajo completo, que muestra la costumbre QRunnable trabajador junto con el trabajador y señales de progreso. Debería poder adaptar fácilmente este código a cualquier aplicación que desarrolle.

```
1  de PyQt5.QtGui importar *
2  PyQt5.QtWidgets importar *
3  PyQt5.QtCore importar *
4
5  importar tiempo
6  importar rastrear, sys
7
8
9  clase TrabajadorSeñales(QObject):
10    """
11        Define las señales disponibles de un subprocesso de trabajo en ejecución.
12
13        Las señales admitidas son:
14
15        finalizado
16            Sin datos
17
18        error
19            `tupla` (exctype, valor, traceback.format_exc ())
20
21        resultado
22            datos del `objeto` devueltos por el procesamiento, cualquier cosa
23
24        Progreso
25            `int` indica% de progreso
26
27        """
28        finalizado = pyqtSignal ()
```

```
29     error =    pyqtSignal (tupla)
30     resultado =  pyqtSignal (objeto)
31     Progreso   = pyqtSignal (En t)
32
33
34 clase Trabajador(QRunnable):
35     """
36     Hilo de trabajador
37
38     Se hereda de QRunnable a la configuración del subproceso del trabajador del controlador, finalización de las
39     y señales.
40
41     :param callback: la función de devolución de llamada que se ejecutará en este
42     trabajador: hilo. Args y
43     los kwargs pasarán al corredor.
44     :escriba devolución de llamada: función
45     :param args: Argumentos para pasar a la función de devolución de llamada:
46     :param kwargs: Palabras clave para pasar a la función de devolución de llamada:
47
48     """
49
50     def __init__(self, fn, *args, **kwargs):
51         super(Trabajador, self).__init__()
52         # Argumentos del constructor de la tienda (reutilizados para el procesamiento)Uno
53         self.fn = fn
54         self.args = args
55         self.kwargs = kwargs
56         self.signals = WorkerSignals()
57
58         # Agregue la devolución de llamada a
59         self.kwargs['progress_callback'] = self.signals.progress
60
61     @pyqtSlot()
62     def run(self):
63         """
64         Inicialice la función de corredor con args, kwargs pasados. """
65
66         # Recupere args / kwargs aquí; y procesamiento de fuego usándolos
```

```
68     tratar:
69         resultado = uno mismo.fn*uno mismo.arguments, **uno mismo.kwargs)
70     excepto:
71         rastrear.print_exc ()
72         exctype, valor = sys.exc_info () [:2]uno mismo.
73         señales.error.emitir ((exctype,demás:           valor, rastreo.format_exc ()))
74
75         uno mismo.señales.resultado.emitir (resultado)      # Devuelve el resultado del proceso |
76     cantar
77
78     finalmente:
79         uno mismo.señales.finalizado.emitir() # Hecho
80
81
82 clase Ventana principal(QMainWindow):
83
84
85     def __en eso__(uno mismo, *argumentos, **kwargs):
86         súper(Ventana principal,   uno mismo).__en eso__(*argumentos,    * *kwargs)
87
88         uno mismo.encimera = 0
89
90         diseño = QVBoxLayout ()
91
92         uno mismo.l = QLabel ("Comienzo")
93         B = QPushButton ("¡PELIGRO!") B.
94         presionado.conectar(uno mismo.Oh no)
95
96         diseño.addWidget (uno mismo.l)
97         diseño.addWidget (b)
98
99         w = QWidget ()
100        w.setLayout (diseño)
101
102        uno mismo.setCentralWidget (w)
103
104        uno mismo.show()
105
106        uno mismo.hilo de rosca = QThreadPool ()
```

```
107     impresión("Múltiples subprocessos con máximo %D hilos" % uno mismo.hilo de rosca.maxThr \
108     eadCount ())
109
110     uno mismo.Temporizador = QTimer ()uno mismo.
111     Temporizador.setInterval (1000)
112     uno mismo.Temporizador.se acabó el tiempo.conectar(uno mismo.temporizador_recurrente)uno
113     mismo.Temporizador.comienzo()
114
115     def progress_fn(uno mismo, n):
116         impresión("%D%% hecho" % norte)
117
118     def ejecutar_esto_fn(uno mismo, progress_callback):por
119         norte en distancia(0, 5):
120             tiempo.dormir(1) progress_callback.emitir
121             (n* 100/4)
122
123     regreso "Hecho."
124
125     def print_output(uno mismo, s):
126         impresión(s)
127
128     def thread_complete(uno mismo):
129         impresión("¡HILO COMPLETO!")
130
131     def Oh no(uno mismo):
132         # Pasar la función a ejecutar
133         trabajador = Trabajador(uno mismo.ejecutar_esto_fn) # Cualquier otro argumento, los kwargs se pasan
134     led a la función de ejecución
135         trabajador.señales.resultado.conectar(uno mismo.print_output)
136         trabajador.señales.finalizado.conectar(uno mismo.thread_complete)
137         trabajador.señales.Progreso.conectar(uno mismo.progress_fn)
138
139         # Ejecutar
140         uno mismo.hilo de rosca.empezar (trabajador)
141
142
143     def temporizador_recurrente(uno mismo):
144         uno mismo.encimera += 1
145         uno mismo.l.setText ("Encimera: %D" % uno mismo.encimera)
```

```
146  
147  
148 aplicación = QApplication ([])  
149 ventana = Ventana principal()  
150 aplicación.exec_()
```

Uso de subprocessos múltiples de Python en PyQt

Es posible que haya detectado un pequeño defecto en este plan maestro; todavía estamos haciendo uso del bucle de eventos (y el *Hilo de la GUI*) para procesar la salida de nuestros trabajadores.

Esto no es un problema cuando simplemente hacemos un seguimiento del progreso, la finalización o la devolución de metadatos. Sin embargo, si tiene trabajadores que devuelven grandes cantidades de datos, por ejemplo, cargar archivos grandes, realizar análisis complejos y resultados necesarios (grandes) o consultar bases de datos, devolver estos datos a través del hilo de la GUI puede causar problemas de rendimiento y es mejor evitarlo.

Del mismo modo, si su aplicación utiliza una gran cantidad de subprocessos y controladores de resultados de Python, es posible que se enfrente a las limitaciones de GIL. Como se mencionó anteriormente, cuando se utilizan subprocessos, la ejecución de Python se limita a un solo subprocesso a la vez. Los trabajadores pueden bloquear el código Python que maneja las señales de sus subprocessos *y viceversa*. Dado que el bloqueo de las funciones de la ranura bloquea el bucle de eventos, esto puede afectar directamente la capacidad de respuesta de la GUI.

En estos casos, a menudo es mejor usar una implementación de grupo de subprocessos de Python puro (por ejemplo, futuros concurrentes) para mantener su *Procesando* y el manejo de eventos de subprocessos más aislado de su GUI.

Ejemplos de aplicaciones PyQt5

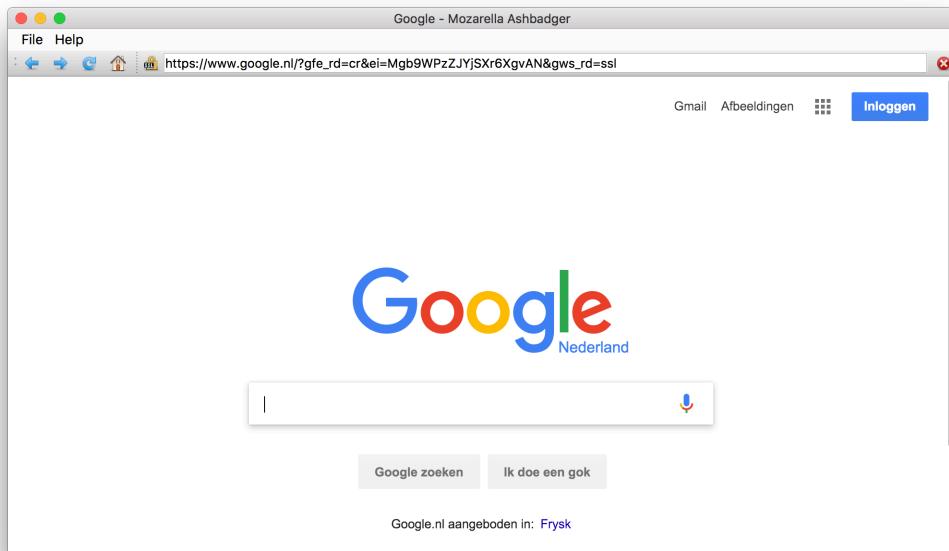
A estas alturas, debería tener un conocimiento firme de cómo crear aplicaciones simples con PyQt. Para mostrar cómo puede poner en práctica lo que ha aprendido, he incluido algunas aplicaciones de ejemplo en este capítulo.

Estas aplicaciones son funcionales, simples y de alguna manera *incompleto*. Úselos como inspiración, para diferenciarse y como una oportunidad para mejorar. Siga leyendo para conocer las partes más interesantes de cada aplicación.

La fuente completa de ambas aplicaciones está disponible para descargar, junto con otras 13 aplicaciones. [en mi repositorio de aplicaciones de 15 minutos en Github](#). ¡Divirtirse!

Ashbadger de mozzarella

¡Mozzarella Ashbadger es la última revolución en la navegación web! ¡Retrocede y avanza! ¡Impresión! ¡Guardar archivos! ¡Consigue ayuda! (lo necesitarás). Cualquier similitud con otros navegadores es pura coincidencia.



Mozzarella Ashbadger.



Esta aplicación hace uso de las funciones cubiertas en [Señales extendidas](#).

El código fuente de Mozzarella Ashbadger se proporciona en dos formatos, uno con navegación con pestañas y otro sin ella. Agregar pestañas complica un poco el manejo de la señal, por lo que la versión sin pestañas se cubre primero.

Código fuente

La fuente completa del navegador sin pestañas se incluye en las descargas de este libro. El código del navegador tiene el nombre `browser.py`.

1 `python3 browser.py`



¡Ejecutarlo! Explorar el *Ashbadger de mozzarella* interfaz y características antes de pasar al código.



Se recomienda que descargue y eche un vistazo rápido al código fuente antes de continuar. El siguiente tutorial destaca las partes clave (e interesantes) del código, pero hay más para ver.

El widget del navegador

El núcleo de nuestro navegador es el `QWebEngineView` del cual importamos `PyQt5.QtWebEngineWidget`. Esto proporciona una ventana de navegador completa, que maneja la representación de las páginas descargadas.

A continuación se muestra el código mínimo requerido para usar el widget del navegador web en PyQt.

```
1  de PyQt5.QtCore importar *
2  PyQt5.QtWidgets importar *
3  PyQt5.QtGui importar *
4  de PyQt5.QtWebEngineWidgets importar *
5
6  importar sys
7
8  clase Ventana principal(QMainWindow):
9
10     def __init__(self, *argumentos, **kwargs):
11         super(Ventana principal, self).__init__(*argumentos, **kwargs)
12
13         self.navegador = QWebEngineView()
14         self.navegador.setUrl(QUrl("http://google.com"))
15
16         self.setCentralWidget(self.navegador)
17
18         self.show()
19
20     aplicacion = QApplication(sys.argv)
21     ventana = Ventana principal()
22
23     aplicacion.exec_()
```

Si hace clic un poco, descubrirá que el navegador se comporta como se esperaba: los enlaces funcionan correctamente y puede interactuar con las páginas. Sin embargo, también

observe que faltan cosas que da por sentado, como una barra de URL, controles o cualquier tipo de interfaz. Esto hace que su uso sea un poco complicado.

Navegación

Para convertir este navegador básico en algo utilizable, agregamos algunos controles, como una serie de QActions en un QToolBar. Agregamos estas definiciones a la_en_eso_- bloque de la QMainWindow.

```

62      navtb = QToolBar ("Navegación") navtb.
63      setIconSize (QSize (dieciséis,dieciséis)uno )
64      mismo.addToolBar (navtb)

sesenta y cinco
66      back_btn = QAction (QIcon (os.sendero.entrar('iconos','flecha-180.png')), "Atrás\"
67      ", uno mismo)
68      back_btn.setStatusTip ("Volver a la página anterior") back_btn.
69      motivado.conectar( uno mismo.navegador.atrás) navtb.
70      addAction (back_btn)
```

los QWebEngineView incluye ranuras para la navegación hacia adelante, hacia atrás y de recarga, a las que podemos conectarnos directamente a nuestra acción .motivado señales.

Usamos lo mismo QAcción estructura para los controles restantes.

```

73      next_btn = QAction (QIcon (os.sendero.entrar('iconos','flecha-000.png')), "Forw\uno mismo
74      ard",
75      next_btn.setStatusTip ("Avanzar a la página siguiente") next_btn.
76      motivado.conectar( uno mismo.navegador.adelante) navtb.
77      addAction (next_btn)
78
79      reload_btn = QAction (QIcon (os.sendero.entrar('iconos','círculo-flecha-315.png')\\"Recargar", uno
80      )), mismo)
81      reload_btn.setStatusTip ("Recargar página") reload_btn.motivado.
82      conectar( uno mismo.navegador.recargar) navtb.addAction
83      (reload_btn)
84
85      home_btn = QAction (QIcon (os.sendero.entrar('iconos','home.png'))), "Hogar", se \
```

```
86    si)
87        home_btn.setStatusTip ("Vete a casa")
88        home_btn.motivado.conectar( uno mismo.navigate_home      )
89        navtb.addAction (home_btn)
```

Tenga en cuenta que mientras que avanzar, retroceder y recargar pueden usar ranuras integradas, el botón de inicio de navegación requiere una función de ranura personalizada. La función de ranura se define en nuestro QMainWindow class y simplemente establece la URL del navegador en la página de inicio de Google. Tenga en cuenta que la URL debe pasarse comoQUrl objeto.

```
197    def Navegar a casa(uno mismo):
198        uno mismo.navegador.setUrl ( QUrl ("http://www.google.com"))
```



Intente configurar la ubicación de navegación de inicio. Podrías crear una PreferencesQDialog con un campo de entrada.

Cualquier navegador web decente también necesita una barra de URL y alguna forma de detener la navegación, ya sea cuando es por error o la página está tardando demasiado.

```
92        uno mismo.httpsicon = QLabel () # ;Sí, en serio; uno
93        mismo.httpsicon.setPixmap (QPixmap (          os.sendero.entrar('iconos','lock-nossal.png')
94    ')  ) )
95        navtb.addWidget (uno mismo.httpsicon)
96
97        uno mismo.urlbar = QLineEdit ()
98        uno mismo.urlbar.volverPresionado.conectar      uno mismo.navigate_to_url)
99        (navtb.addWidget (uno mismo.urlbar)
100
101        stop_btn = QAction (QIcon (os.sendero.entrar('iconos','cross-circle.png')), "S\uno mismo
102    cima",
103        stop_btn.setStatusTip ("Dejar de cargar la página actual") stop_btn.
104        motivado.conectar( uno mismo.navegador.detener) navtb.
105        addAction (stop_btn)
```

Como antes, la función 'detener' está disponible en el QWebEngineView, y podemos simplemente conectar el .motivado señal desde el botón de parada a la ranura existente. Sin embargo, debemos manejar otras características de la barra de URL de forma independiente.

Primero agregamos un QLabel para mantener nuestro icono SSL o no SSL para indicar si la página es segura. A continuación, agregamos la barra de URL que es simplemente una QLineEdit. Para activar la carga de la URL en la barra cuando se ingresa (tecla de retorno presionada) nos conectamos al.volverPresionado señal en el widget para impulsar una función de ranura personalizada para activar la navegación a la URL especificada.

```
202     def navigate_to_url(uno mismo): # No recibe la URL
203         q = QUrl ( uno mismo.urlbar.texto() )si q.
204             esquema() == "":
205                 q.setScheme ("http")
206
207         uno mismo.navegador.setUrl (q)
```

También queremos que la barra de URL se actualice en respuesta a los cambios de página. Para hacer esto nosotros puede usar el .urlChanged y .loadFinished señales del QWebEngineView. Nosotros configurar las conexiones de las señales en el _en eso_ bloquear de la siguiente manera:

```
57     uno mismo.navegador.urlChanged.conectar(uno mismo.update_urlbar)
58     uno mismo.navegador.loadFinished.conectar(uno mismo.update_title)
```

Luego definimos las funciones de la ranura de destino que para estas señales. El primero, para actualizar la barra de URL acepta unQUrl objeto y determina si se trata de un http o https URL, utilizándola para configurar el icono SSL.

ADVERTENCIA: Esta es una forma terrible de probar si una conexión es 'segura'. Para ser correctos debemos realizar una validación de certificado.

los QUrl se convierte en una cadena y la barra de URL se actualiza con el valor. Tenga en cuenta que también establecemos la posición del cursor de nuevo al principio de la línea para evitar la QLineEdit widget desplazándose hasta el final.

```
211     def update_urlbar(uno mismo, q):
212
213         si q.esquema() == 'https':
214             # IIcono de candado seguro uno
215             mismo_httpsicon.setPixmap ( QPixmap ( os.sendero.entrar('iconos','lock-ssl.p\
216             ng ') ) )
217
218         demás:
219             # IIcono de candado inseguro uno
220             mismo_httpsicon.setPixmap ( QPixmap ( os.sendero.entrar('iconos','lock-noss\\
221             .png ') ) )
222
223             uno mismo.urlbar.setText (q.Encadenar()uno )
224             mismo.urlbar.setCursorPosition (0)
```

También es un buen toque actualizar el título de la ventana de la aplicación con el título de la página actual. Podemos conseguir esto a través de browser.page().title() que devuelve el contenido del <título> </title> etiqueta en la página web actualmente cargada.

```
149     def update_title(uno mismo):
150         título = uno mismo.navegador.página().título()
151         uno mismo.setWindowTitle ("%s - Mozarella Ashbadger " % título)
```

Operaciones de archivo

Un menú Archivo estándar con self.menuBar().addMenu ("& Archivo") se crea asignando el F como un atajo Alt (como es normal). Una vez que tenemos el objeto de menú, podemos asignar QAcción objetos para crear las entradas. Creamos dos entradas básicas aquí, para abrir y guardar archivos HTML (desde un disco local). Ambos requieren funciones de ranura personalizadas.

```

110     menú Archivo = uno mismo.Barra de menús().addMenu ("&Expediente")
111
112         open_file_action = QAction (QIcon (os.sendero.entrar('iconos','disk--arrow.pn\gramo')), "Abrir
113 documento...", uno mismo)
114     open_file_action.setStatusTip ("Abierto desde el archivo")
115     open_file_action.motivado.conectar uno mismo.abrir_documento )
116     (menú_archivo.addAction (acción_archivo_abierto)
117
118         save_file_action = QAction (QIcon (os.sendero.entrar('iconos','disco - lápiz.p\ng ')), "Guardar
119 pagina como...", uno mismo)
120     save_file_action.setStatusTip ("Guardar la página actual en un archivo")
121     save_file_action.motivado.conectar( uno mismo.guardar_archivo) file_menu.
122     addAction (guardar_archivo_acción)

```

La función de ranura para abrir un archivo utiliza la función incorporada `QFileDialog.getOpenFileName()` función para crear un diálogo de apertura de archivo y obtener un nombre. Restringimos los nombres de forma predeterminada a los archivos que coinciden `*.htm` o `*.html`.

Leemos el archivo en una variable `.html` usando funciones estándar de Python, luego use `.setHtml()` para cargar el HTML en el navegador.

```

164     def abrir_documento(unو mismo):
165         nombre del archivo, _ = QFileDialog.getOpenFileName (uno mismo, "Abrir documento", "",
166                                         "Lenguaje de marcado de hipertexto (*.htm *.html);;"
167                                         "Todos los archivos (*.*)")
168
169         si nombre del archivo:
170             con abierto(nombre del archivo, 'r') como F:
171                 html = F.leer()
172
173             uno mismo.navegador.setHtml (html)uno
174             mismo.urlbar.setText (nombre de archivo)

```

De manera similar, para guardar el HTML de la página actual, usamos el `QFileDialog.getSaveFileToGetAFilename`. Sin embargo, esta vez obtenemos el HTML de `self.browser.page().toHtml()` y escríbalo en el nombre de archivo seleccionado. Nuevamente usamos funciones estándar de Python para el manejador de archivos.

```

178     def guardar_el_archivo(uno mismo):
179         nombre del archivo, _ = QFileDialog.getSaveFileName (uno mismo, "Guardar pagina como", "",
180                                         "Lenguaje de marcado de hipertexto (*.htm * html) ;;"
181                                         "Todos los archivos (*.*)")
182
183         si nombre del archivo:
184             html = uno mismo.navegador.página().toHtml ()con
185             abierto(nombre del archivo, 'w') como F:
186                 F.escribir (html)

```

Impresión

Podemos agregar una opción de impresión al menú Archivo usando el mismo enfoque que usamos anteriormente. Nuevamente, esto necesita una función de ranura personalizada para realizar la acción de impresión.

```

124     print_action = QAction (QIcon (os.sendero.entrar('iconos','impresora.png')), "\\\\Impresión...", uno
125 mismo)
126     print_action.setStatusTip ("Impresión           página actual")
127     print_action.motivado.conectar
128             uno mismo.Imprimir página )
(menú_archivo.addAction (print_action)

```

Qt proporciona un marco de impresión completo. QPrintPreviewDialog para solicitar la configuración del usuario. El objeto de diálogo tiene un.paintRequested signal, que podemos conectar al controlador de impresión del widget que deseamos imprimir. Afortunadamente, el QWebEngineView proporciona una interfaz compatible para que nos conectemos.

los .paintRequested La señal se activará si se acepta el cuadro de diálogo y se imprimirá la página.

```

190     def Imprimir página(uno mismo):
191         dlg = QPrintPreviewDialog () dlg.
192         paintRequested.conectar (dlg.exec_   uno mismo.navegador.impresión_    )
193             ()

```

Ayudar

Finalmente, para completar la interfaz estándar podemos agregar un menú Ayuda. Esto se define como antes, dos funciones de ranura personalizadas para manejar la visualización de un diálogo Acerca de y para cargar la 'página del navegador' con más información.

```

131     menú de ayuda = uno mismo.Barra de menús().addMenu ("&Ayudar")
132
133     about_action = QAction (QIcon (os.sendero.entrar('iconos','question.png')), \"Acerca de Mozarella
134 Mozarella Ashbadger", uno mismo)
135     about_action.setStatusTip ("Más información sobre Mozarella Ashbadger") # I
136     Hambrero!
137     about_action.motivado.conectar( uno mismo.acerca de)
138     help_menu.addAction (about_action)
139
140     navigate_mozarella_action = QAction (QIcon (os.sendero.entrar('iconos','lifebuoy.png ')), "Página
141 principal de Mozarella Ashbadger", uno mismo)
142     navigate_mozarella_action.setStatusTip ("Ir a la página principal de Mozarella Ashbadger\"
143 la edad")
144     navigate_mozarella_action.motivado.conectar( uno mismo.navigate_mozarella) help_menu.
145     addAction (navigate_mozarella_action)
```

Definimos dos métodos que se utilizarán como espacios para las señales del menú Ayuda. El primero navegar_mozarella abre una página con más información sobre el navegador (o en este caso, este libro). El segundo crea y ejecuta un personalizado QDialog clase Acerca del dialogo que definiremos a continuación.

```

155     def navegar_mozarella(uno mismo):
156         uno mismo.navegador.setUrl (QUrl ("https://www.udemy.com/522076"))
157
158     def sobre(uno mismo):
159         dlg = AboutDialog ()
160         dlg.exec_()
```

La definición del diálogo Acerca de se da a continuación. La estructura sigue la vista anteriormente en el libro, con un QDialogButtonBox y señales asociadas para manejar la entrada del usuario, y una serie de QLabels para mostrar la información de la aplicación y un logotipo.

El único truco aquí es agregar todos los elementos al diseño, luego iterar sobre ellos para establecer la alineación en el centro en un solo bucle. Esto ahorra duplicación para las secciones individuales.

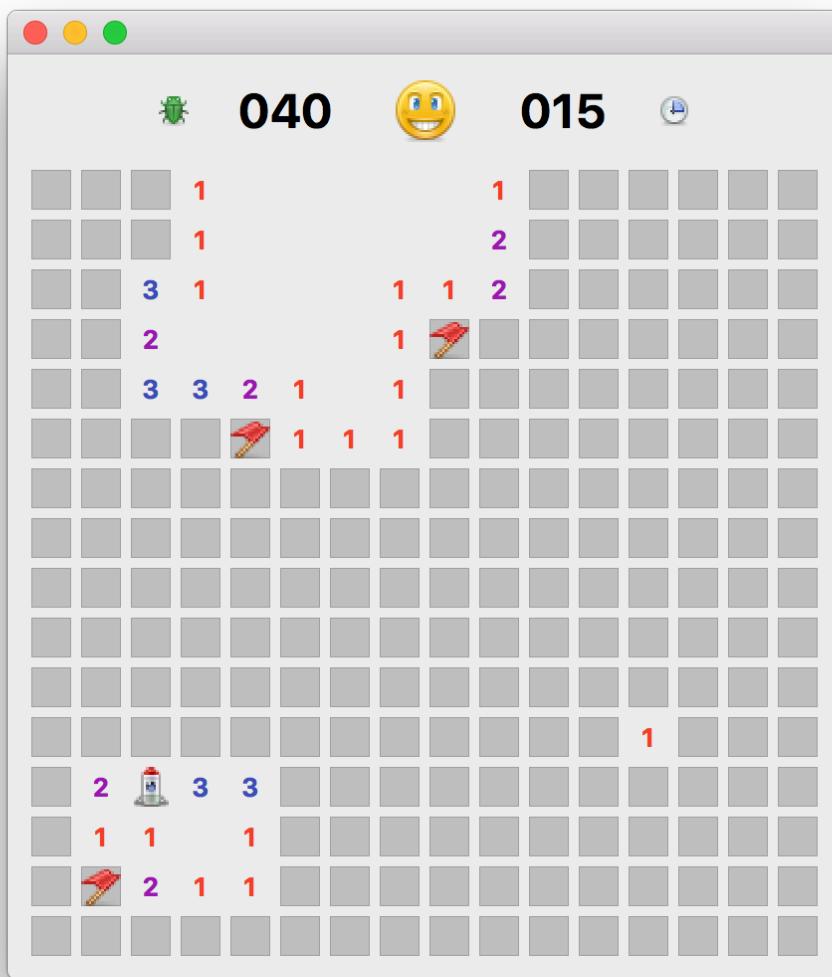
Barrendero

¡Explora la misteriosa luna de Q'tee sin acercarte demasiado a los nativos alienígenas!

Moonsweeper es un videojuego de rompecabezas para un solo jugador. El objetivo del juego es explorar el área alrededor de tu cohete espacial aterrizado, sin acercarte demasiado a los letales alienígenas B'ug. Su confiable tri-contador le dirá el número de B'ugs en las cercanías.



Esta aplicación hace uso de las funciones cubiertas en [Señales extendidas](#).



Barrendero

Esta es una simple exploración para un jugador. *Dragaminas* donde debes revelar todas las fichas sin golpear las minas ocultas. Esta implementación usa `customQWidget` objetos para las baldosas, que individualmente mantienen su estado como minas,

estado y el recuento adyacente de minas. En esta versión, las minas son reemplazadas por insectos alienígenas (B'ug) pero fácilmente podrían ser cualquier otra cosa.

En muchos *Dragaminas* variantes, el turno inicial se considera un paso libre: si golpea una mina con el primer clic, se mueve a otro lugar. Aquí hacemos un poco de trampa al dar la primera oportunidad al jugador, asegurándonos de que esté en un lugar que no sea mío. Esto nos permite no preocuparnos por el mal primer movimiento que nos obligaría a recalcular las adyacencias. Podemos explicar esto como la "exploración inicial alrededor del cohete" y hacer que suene completamente sensato.



Si desea implementar esto, puede capturar el primer clic en una posición y en ese punto generar minas / adyacencias, excluyendo su ubicación, antes de manejar el clic. Deberá dar acceso a sus widgets personalizados al objeto de la ventana principal.

Código fuente

La fuente completa de la *Barrendero* El juego está incluido en las descargas de este libro. El archivo del juego se guarda con el nombrebuscaminas.py.

1 python3 buscaminas.py



Se recomienda que descargue y eche un vistazo rápido al código fuente antes de continuar. El siguiente tutorial destaca las partes clave (e interesantes) del código, pero hay más para ver.

Campo de juego

El área de juego de Moonsweeper es un NxNgrid, que contiene un número determinado de minas. Las dimensiones y recuentos de minas que usaremos se toman de los valores predeterminados para la versión de Windows de Buscaminas. Los valores utilizados se muestran en la siguiente tabla:

. Dimensiones de la tabla y recuento de minas | === | Nivel | Dimensiones | Número de minas |
Fácil | 8 x 8 | 10 | Mediano | 16 x 16 | 40 | Difícil | 24 x 24 | 99 | ===

Almacenamos estos valores como una constante NIVELES definido en la parte superior del archivo. Dado que todos los campos de juego son cuadrados, solo necesitamos almacenar el valor una vez (8, 16 o 24).

```
1 NIVELES = [  
2     ("Fácil", 8,      10),  
3     ("Medio",   dieciséis, 40),  
4     ("Duro",    24,      99)  
5 ]
```

La cuadrícula de juego podría representarse de varias formas, incluida, por ejemplo, una "lista de listas" en 2D que represente los diferentes estados de las posiciones de juego (mía, revelada, marcada).

Sin embargo, en nuestra implementación usaremos un enfoque orientado a objetos, donde las posiciones individuales en el mapa contienen todos los datos relevantes sobre sí mismos. Llevando esto un paso más allá, podemos hacer que estos objetos sean individualmente responsables de dibujarse a sí mismos. En Qt podemos hacer esto simplemente subclasiificando deQWidget y luego implementando una función de pintura personalizada.

Cubriremos la construcción y el comportamiento de estos widgets personalizados antes de pasar a su apariencia. Dado que nuestros objetos de mosaico son subclases deQWidget podemos diseñarlos como cualquier otro widget. Hacemos esto, configurando unQGridLayout.

```
1 uno mismo.red = QGridLayout()  
2 uno mismo.red.setSpacing(5)  
3 uno mismo.red.setSizeConstraint(QLayout.SetFixedSize)
```

A continuación, debemos configurar el campo de juego, crear nuestros widgets de mosaicos de posición y agregarlos a nuestra cuadrícula. La configuración inicial para el nivel se define en el método personalizado, que se lee enNIVELES y asigna una serie de variables a la ventana. Se actualizan el título de la ventana y el contador de minas y, a continuación, se inicia la configuración de la cuadrícula.

```

234     def set_level(uno mismo, nivel):
235         uno mismo.level_name, uno mismo.b_size, uno mismo.n_mines = NIVELES [nivel]
236
237         uno mismo.setWindowTitle ("Moonsweeper - %s" % (uno mismo.nombre_nivel))uno
238         mismo.minas.setText ("% 03d" % uno mismo.n_mines)
239
240         uno mismo.mapa.claro()
241         uno mismo.init_map ()
242         uno mismo.reset_map ()

```

Las funciones de configuración se tratarán a continuación.

Estamos usando una costumbre Pos class aquí, que veremos en detalle más adelante. Por ahora, solo necesita saber que contiene toda la información relevante para la posición relevante en el mapa, incluyendo, por ejemplo, si es una amina, revelada, marcada y el número de minas en las inmediaciones.

Cada Pos el objeto también tiene 3 señales personalizadas *clickeado*, *revelado* y *ampliable* que conectamos a métodos de tragamonedas personalizados. Finalmente, llamamos a resize para ajustar el tamaño de la ventana al nuevo contenido. Tenga en cuenta que esto solo es necesario cuando la ventana *encoge* - crecerá automáticamente.

```

257     def init_map(uno mismo):
258         # Agregar posiciones al mapa por X en
259         distancia(0, uno mismo.b_size):
260             por y en distancia(0, uno mismo.b_size):
261                 w = Pos (x, y)
262                 uno mismo.red.addWidget (w, y, x)
263                 # Conecte la señal para manejar la expansión.w.
264                 clickeado.conectar(uno mismo.trigger_start) w.
265                 revelado.conectar(uno mismo.on_reveal)
266                 w.ampliable.conectar(uno mismo.expand_reveal)
267
268         # Coloque el cambio de tamaño en la cola de eventos, devolviendo el control a Qt antes.QTimer.un
269         solo tiro(0, lambda: uno mismo.cambiar el tamaño1,1)) #<1>

```



1. El un solo tiro Se requiere un temporizador para garantizar que el cambio de tamaño se ejecute después de que Qt tenga conocimiento de los nuevos contenidos. Mediante el uso de un temporizador, `weguaranteecontrol` volverá a Qt antes de que se produzca el cambio de tamaño.

También necesitamos implementar la inversa de la `init_map` función para eliminar los objetos de mosaico del mapa. Será necesario quitar los mosaicos al pasar de un nivel superior a uno inferior. Sería posible ser un poco más inteligente aquí y agregar / eliminar solo los mosaicos que son necesarios para obtener el tamaño correcto. Pero, como ya tenemos la función de sumar todo al tamaño correcto, podemos hacer un poco de trampa.



Actualice este código para agregar / eliminar los mosaicos necesarios para dimensionar las dimensiones del nuevo nivel.

Observe que ambos eliminamos el elemento de la cuadrícula con `self.grid.removeitem (c)` y limpiar al padre `c.widget () .setParent (Ninguno)`. Este segundo paso es necesario, ya que al agregar los elementos se les asigna la ventana principal como principal. Simplemente quitarlos los deja flotando en la ventana fuera del diseño.

```

246     def mapa_claro(uno mismo):
247         # Elimina todas las posiciones del mapa, hasta el tamaño máximo, por X en
248         distancia(0, NIVELES [-1] [1]): # <1>
249             por y en distancia(0, NIVELES [-1] [1]):
250                 C = uno mismo.red.itemAtPosition (y, x) si C: #
251                     <2>
252                         uno mismo.red.removeitem (c) c.
253                             widget () .setParent (Ninguno)

```



1. Para asegurarnos de borrar todos los tamaños de mapas, tomamos la dimensión del nivel más alto.
2. Si no hay nada en la cuadrícula en esta ubicación, podemos omitirlo.

Ahora que tenemos nuestra cuadrícula de objetos de mosaico posicionales en su lugar, podemos comenzar a crear las condiciones iniciales del tablero de juego. Este proceso es bastante complejo, por lo que se divide en varias funciones. Los nombramos _Reiniciar (el subrayado inicial es una convención para indicar una función privada, no diseñada para uso externo). La función principal reset_map llama a estas funciones a su vez para configurarlo.

El proceso es el siguiente -

1. Elimine todas las minas (y restablezca los datos) del campo.
2. Agregue nuevas minas al campo.
3. Calcule el número de minas adyacentes a cada posición.
4. Agregue un marcador de inicio (el cohete) y active la exploración inicial.
5. Reinicie el temporizador.

```
273     def reset_map(uno mismo):  
274         uno mismo._reset_position_data ()uno  
275             mismo._reset_add_mines ()  
276             uno mismo._reset_calculate_adjacency ()  
277             uno mismo._reset_add_starting_marker ()  
278             uno mismo.update_timer ()
```

Los pasos separados del 1 al 5 se describen en detalle a continuación, con el código para cada paso.

El primer paso es restablecer los datos de cada posición en el mapa. Repasamos cada posición en el tablero, llamando .Reiniciar() en el widget en cada punto. El código para el .Reiniciar() la función está definida en nuestra costumbre Pos clase, exploraremos en detalle más adelante. Por ahora, es suficiente saber que limpia minas, banderas y vuelve a poner la posición para no ser revelada.

```

282     def _reset_position_data(uno mismo):
283         # Borrar todas las posiciones de la mina por X
284         en distancia(0, uno mismo.b_size):
285             por y en distancia(0, uno mismo.b_size):
286                 w = uno mismo.red.itemAtPosition (y, x).widget () w.
287                 Reiniciar()

```

Ahora que todas las posiciones están en blanco, podemos comenzar el proceso de agregar minas al mapa. El número máximo de minas `n_mines` se define mediante la configuración de nivel, descrita anteriormente.

```

1      def _reset_add_mines(uno mismo):
2          # Agregar el mío  posiciones
3          posiciones  = []
4          tiempo len(posiciones) < uno mismo.n_mines:
5              x, y = aleatorio.randint0, uno mismo.b_tamaño-1), aleatorio.randint0, uno mismo.b_s \
6              tamaño-1)
7              si (x, y) no en posiciones:
8                  w = uno mismo.red.itemAtPosition (y, x).widget () w.es
9                  mio = Ciertoposiciones.añadir ((x, y))
10
11
12          # Calcular la condición del juego final
13          uno mismo.end_game_n = (uno mismo.b_tamaño * uno mismo.b_size) - (uno mismo.n_mines + 1)
14          regreso posiciones

```

Con las minas en posición, ahora podemos calcular el número de 'adyacencia' para cada posición, simplemente el número de minas en las inmediaciones, usando una cuadrícula de 3x3 alrededor del punto dado. La función personalizada `get_surrounding` simplemente devuelve esas posiciones en torno a un determinado `x` y `y` localización. Contamos el número de estos que es un `mía is_mine ==` verdadero y almacenar.



El cálculo previo de los recuentos adyacentes de esta manera ayuda a simplificar la lógica de revelación más adelante.

```

1   def _reset_calculate_adjacency(uno mismo):
2
3       def get_adjacency_n(x, y):
4           posiciones = uno mismo.get_surrounding(x, y)regreso suma(
5               1 por w en posiciones si w.es mio)
6
7       # Agregar adyacencias a las posicionespor X
8       en distancia(0, uno mismo.b_size):
9           por y en distancia(0, uno mismo.b_size):
10          w = uno mismo.red.itemAtPosition(y, x).widget() w.
11          adyacente_n = get_adjacency_n(x, y)

```

Se usa un marcador de inicio para asegurar que el primer movimiento sea *always_valid*. Esto se implementa como una fuerza _brutebusque a través del espacio de la cuadrícula, probando efectivamente posiciones aleatorias hasta que encontremos una posición que no sea la mía. Como no sabemos cuántos intentos tomará esto, necesitamos envolverlo en un ciclo continuo.

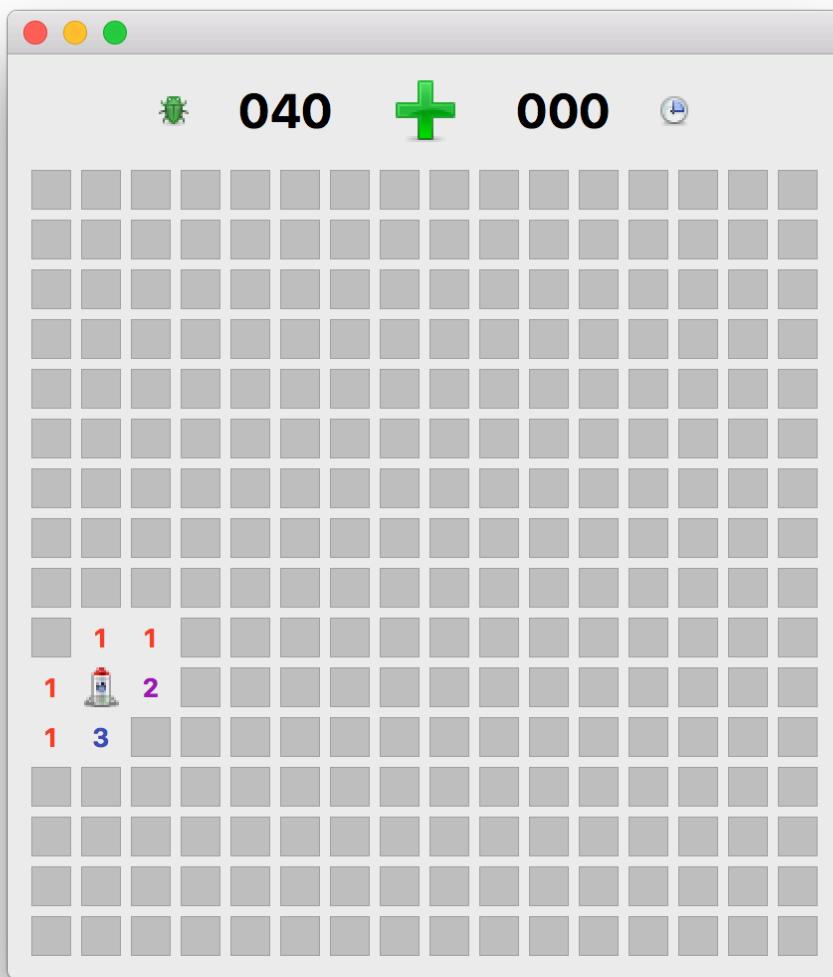
Una vez que se encuentra esa ubicación, la marcamos como la ubicación de inicio y luego activamos la exploración de todas las posiciones circundantes. Salimos del bucle y restablecemos el estado listo.

```

1   def _reset_add_starting_marker(uno mismo):
2       # Coloque el marcador de inicio.
3
4       # Establecer el estado inicial (necesario para que funcione .click)uno
5       mismo.update_status(STATUS_READY)
6
7       tiempo Cierto:
8           x, y = aleatorio.randint0, uno mismo.b_tamaño - 1), aleatorio.randint0, uno mismo.B\(
9           _Talla - 1)
10          w = uno mismo.red.itemAtPosition(y, x).widget()
11          # No queremos empezar con una mina.si no w.
12          es mio:
13              w.is_start = Cierto
14              w.es revelado = Cierto
15              w.actualizar()
16
17          # Revele todas las posiciones en torno a esto, si no son minas eithe |

```

```
18 r.  
19     por w en uno.mismo.get_surrounding(x,y):  
20         si no w.es mio:  
21             w.hacer clic()  
22             rotura  
23  
24         # Restablezca el estado a listo después de los clics iniciales.  
25         uno.mismo.update_status(STATUS_READY)
```



Exposición inicial alrededor del cohete.

Colocar mosaicos

Como se describió anteriormente, hemos estructurado el juego para que las posiciones individuales de los mosaicos contengan su propia información de estado. Esto significa que los objetos están en una posición ideal para manejar la lógica del juego que reacciona a las interacciones que se relacionan con su propio estado; en otras palabras, aquí es donde está la magia.

Desde el Pos La clase es relativamente compleja, se desglosa aquí en varios temas, que se discuten a su vez. La configuración inicial_en eso bloque es simple, aceptando un x y y posición y almacenarlo en el objeto. Pos las posiciones nunca cambian una vez creadas.

Para completar la configuración .Reiniciar() Se llama a la función que restablece todos los atributos del objeto a los valores predeterminados, cero. Esto marca la mina como_no la posición de inicio, _no es mio, no revelado y no marcado. También restablecemos el recuento adyacente.

```
1  clase Pos(QWidget):
2
3      ampliable    = pyqtSignal (En t,En t)
4      revelado     = pyqtSignal (objeto)
5      clickeado    = pyqtSignal ()
6
7      def __en eso__(uno mismo,x,y,*argumentos,**kwargs):
8          súper(Pos, uno mismo).__en eso__(*argumentos,      **kwargs)
9
10         uno mismo.setFixedSize (QSize (20,uno    20))
11         mismo.X  = X
12         uno mismo.y = y
13         uno mismo.Reiniclar()
14
15     def Reiniciar(uno mismo):
16         uno mismo.is_start  = Falso
17         uno mismo.es mio   = Falso
18         uno mismo.adyacente_n = 0
19         uno mismo.es revelado = Falso
20         uno mismo.is_flagged = Falso
21
22         uno mismo.actualizar()
```

El juego se centra en las interacciones del ratón con las fichas del campo de juego, por lo que detectar y reaccionar a los clics del ratón es fundamental. En Qt detectamos los clics del mouse detectando el `mouseReleaseEvent`. Todo esto por nuestra costumbre `QMouseEvent` widget que define un controlador en la clase. Esto recibe `QMouseEvent` con la información que contiene lo sucedido. En este caso, solo nos interesa si la liberación del mouse se produjo con el botón izquierdo o derecho del mouse.

Para un clic izquierdo del mouse, verificamos si el mosaico está marcado o ya está revelado. Si es así, ignoramos el clic, lo que hace que los mosaicos marcados sean "seguros", ya que no se puede hacer clic por accidente. Si el mosaico no está marcado, simplemente iniciamos el `.hacer clic()` método (ver más adelante).

Para hacer clic con el botón derecho del mouse, en mosaicos *no* revelado, llamamos a nuestro `.toggle_flag()` método para activar y desactivar una bandera.

```
1 def mouseReleaseEvent(uno mismo, e):
2
3     si (mi.botón() == Qt.Botón derecho y no uno mismo.es revelado):
4         uno mismo.toggle_flag()
5
6     elif (mi.botón() == Qt.Botón izquierdo):
7         # Bloquear haciendo clic en minas marcadas.
8         si no uno mismo.is_flagged y no uno mismo.es revelado:
9             uno mismo.hacer clic()
```

Los métodos llamados por el `mouseReleaseEvent` handler se definen a continuación.

los `.toggle_flag` el manejador simplemente establece `.is_flagged` a la inversa de sí misma (Cíerto se convierte en Falso, Falso se convierte en Cíerto) que tiene el efecto de activarlo y desactivarlo. Tenga en cuenta que tenemos que llamar `.actualizar()` para forzar un redibujo ha cambiado el estado. También emitimos nuestra costumbre `.clikcado` señal, que se usa para iniciar el temporizador, porque colocar una bandera también debe contar como inicio, no solo como revelar un cuadrado.

los `.hacer clic()` El método maneja un clic izquierdo del mouse y, a su vez, activa la revelación del cuadrado. Si el número de minas adyacentes a este `Pos` es cero, activamos el `.ampliable` señal para comenzar el proceso de autoexpansión de la región explorada (ver más adelante). Entonces, volvemos a emitir `.clikcado` para señalar el inicio del juego.

Finalmente, el `.revelar()` El método comprueba si el mosaico ya está revelado y, si no, establece `.es_revelado` para Cíerto. De nuevo llamamos `.actualizar()` para activar un repintado del widget.

La emisión opcional del `.revelado` La señal se usa solo para la revelación del mapa completo del final del juego. Debido a que cada revelación desencadena una búsqueda adicional para encontrar qué mosaicos también se pueden revelar, revelar el mapa completo crearía una gran cantidad de devoluciones de llamada redundantes. Suprimiendo la señal aquí evitamos eso.

```
1  def toggle_flag(uno mismo):
2      uno mismo.is_flagged = no uno mismo.is_flagged uno
3      mismo.actualizar()
4
5      uno mismo.clickeado.emitir()
6
7  def hacer clic(uno mismo):
8      uno mismo.revelar()
9      si uno mismo.adyacente_n == 0:
10         uno mismo.ampliable.emitir(uno mismo.X, uno mismo.y)
11
12     uno mismo.clickeado.emitir()
13
14 def revelar(uno mismo, emitir=Cierto):
15     si no uno mismo.es revelado:
16         uno mismo.es revelado = Verdadero
17         ser.actualizar()
18
19     si emitir:
20         uno mismo.revelado.emitir(uno mismo)
```

Finalmente, definimos un personalizado `paintEvent` método para nuestro `Pos` widget para manejar la visualización del estado de la posición actual. Como se describe en el capítulo para realizar una pintura personalizada sobre un lienzo de widgets, `QPainter` y el `event.rect()` que proporciona los límites en los que vamos a trazar, en este caso el borde exterior de la `Pos` widget.

Los mosaicos revelados se dibujan de manera diferente dependiendo de si el mosaico es un *Posición de salida, bomba o espacio vacío*. Los dos primeros están representados por iconos de un cohete y una bomba respectivamente. Estos se dibujan en el mosaico `QRect` utilizando `.drawPixmap`. Tenga en cuenta que necesitamos convertir el `QImage` constantes a mapas de píxeles, pasando a través de `QPixmap` pasando.



Puede pensar "¿por qué no almacenarlos simplemente como QPixmap objetos ya que eso es lo que estamos usando? No podemos hacer esto y almacenarlos en constantes porque no puede crear QPixmap objetos antes QApplication instancia está funcionando.

Para posiciones vacías (no cohetes, no bombas) mostramos opcionalmente el número de adyacencia si es mayor que cero. Para dibujar texto en nuestro QPainter usamos .dibujarTexto () pasando en el QRect, banderas de alineación y el número a dibujar como una cadena. Hemos definido un color estándar para cada número (almacenado enNUM_COLORS) para la usabilidad.

Para baldosas que son *no Revelado* dibujamos un mosaico, rellenando un rectángulo con gris claro y dibujamos un borde de 1 píxel de gris más oscuro. Si.is_flagged está configurado, también dibujamos un icono de bandera sobre la parte superior del mosaico usando drawPixmap y el azulejo QRect.

```
1  def paintEvent(uno mismo, evento): p
2      = QPainter (uno mismo)
3      pag.setRenderHint (QPainter.Antialiasing)
4
5      r = evento.rect ()
6
7      si uno mismo.es revelado:
8          si uno mismo.is_start:
9              pag.drawPixmap (r, QPixmap (IMG_START))
10
11     elif uno mismo.es mio:
12         pag.drawPixmap (r, QPixmap (IMG_BOMB))
13
14     elif uno mismo.adyacente_n > 0:
15         lápiz = QPen (NUM_COLORS [uno mismo.adyacente_n]) p.
16         setPen (bolígrafo)
17         F = pag.fuente()
18         F.setBold (Cierto)
19         pag.setFont (f)
20         pag.dibujarTexto (r, Qt.AlignHCenter | Qt.AlignVCenter, str(uno mismo.adjac \
21 ent_n))
22
23     demás:
24         pag.fillRect (r, QBrush (Qt.gris claro))
```

```

25     lápiz = QPen (Qt.gris)
26     bolígrafo.setWidth (1) pag.
27     setPen (bolígrafo)
28     pag.drawRect (r)
29
30     si uno mismo.is_flagged:
31         pag.drawPixmap (r, QPixmap (IMG_FLAG))

```

Mecánica

Por lo general, necesitamos obtener todos los mosaicos que rodean un punto determinado, por lo que tenemos una función personalizada para ese propósito. Simplemente itera a través de una cuadrícula de 3x3 alrededor del punto, con una verificación para asegurarse de que no salgamos de los límites en los bordes de la cuadrícula ($0 \geq x \leq self.b_-tamaño$). La lista devuelta contiene unPos widget de cada ubicación circundante.

```

1      # etiqueta :: circundante []
2      def get_surrounding(uno mismo, x, y):
3          posiciones = []
4
5          por xi en distancia(max(0, X - 1), min(X + 2, uno mismo.b_tamaño)):
6              por yi en distancia(max(0, y - 1), min(y + 2, uno mismo.b_tamaño)):
7                  si no (xi == X y yi == y):
8                      posiciones.adjuntar( uno mismo.red.itemAtPosition (yi, xi).widget () \
9 )
10
11     regreso posiciones

```

los expand_reveal El método se activa en respuesta a un clic en un mosaico sin minas adyacentes. En este caso, queremos expandir el área alrededor del clic a cualquier espacio que también tenga cero minas adyacentes, y también revelar los cuadrados alrededor del borde de esa área expandida (que no son minas).

Empezamos con una lista expandir que contiene las posiciones para comprobar en la siguiente iteración, una lista Revelar que contiene los tilewidgets para revelar, y una bandera any_added para determinar cuándo salir del bucle. El bucle se detiene la primera vez que no se agregan nuevos widgets aRevelar.

Dentro del bucle reiniciamos any_added para Falso y vaciar el expandir lista, manteniendo una tienda temporal en l para iterar.

Para cada x y y ubicación obtenemos los 8 widgets circundantes. Si alguno de estos widgets no es una amina y no está ya en elRevelar listweadd. Esto asegura que se revelen todos los bordes del área expandida. Si la posición no tiene minas adyacentes, agregamos las coordenadas aexpandir que se comprobará en la siguiente iteración.

Añadiendo baldosas que no sean de mina a Revelary solo expandir mosaicos que aún no están en Revelar, nos aseguramos de no visitar un mosaico más de una vez.

```
1  def expand_reveal(uno mismo, x, y):
2      """
3          Itere hacia afuera desde el punto inicial, agregando nuevas ubicaciones a la cola. Esto nos
4          permite expandirnos todo de una sola vez, en lugar de depender de múltiples devoluciones
5          de llamada.
6      """
7      expandir = [(x, y)]
8      Revelar = []
9      any_added = Cierto
10
11     tiempo any_added:
12         any_added = Falso
13         to_expand, l = [], expandir
14
15     por x, y en l:
16         posiciones = uno mismo.get_surrounding(x, y)por
17         w en posiciones:
18             si no w.es mio y w no en Revelar:
19                 Revelar.append(w)
20                 si w.adyacente_n == 0:
21                     expandir.append((w.x, w.y))
22                     any_added = Cierto
23
24     # Iterar y revelar todas las posiciones que hemos encontrado.por w en
25     Revelar:
26         w.revelar()
```

Finales

Los estados del final del juego se detectan durante el proceso de revelación después de hacer clic en un título. Hay dos posibles resultados:

1. Tile is a mine, game over.
2. El mosaico no es mío, disminuya el self.end_game_n.

Esto continúa hasta self.end_game_n llega a cero, lo que activa el proceso del juego ganador llamando a juego terminado o juego ganado. El éxito / fracaso se desencadena al revelar el mapa y establecer el estado relevante, en ambos casos.

```
1  def on_reveal(uno mismo, w):  
2      si w.es mio:  
3          uno mismo.juego terminado()  
4  
5      demás:  
6          uno mismo.end_game_n -= 1 # disminuir los espacios vacíos restantes  
7  
8      si uno mismo.end_game_n == 0:  
9          uno mismo.juego ganado()  
10  
11     def juego terminado(uno mismo):  
12         uno mismo.revelar mapa()  
13         uno mismo.update_status (STATUS_FAILED)  
14  
15     def juego ganado(uno mismo):  
16         uno mismo.revelar mapa()  
17         uno mismo.update_status (STATUS_SUCCESS)
```

Otras ideas

Si quiere intentar expandir *Barrendero*, Aquí hay algunas ideas -

1. Permita que el jugador tome su primer turno. Intente posponer el cálculo de las posiciones de la mina hasta que el usuario haga clic por primera vez, y luego genere posiciones hasta que se pierda.

2. Agregue potenciadores, por ejemplo, un escáner para revelar un área determinada del tablero automáticamente.
3. Deje que los B'ugs ocultos se muevan entre cada turno. Mantenga una lista de posiciones gratuitas no reveladas y permita que los B'ugs se muevan a ellas. Deberá volver a calcular las adyacencias después de cada clic.

Empaquetado de aplicaciones PyQt

No es muy divertido crear sus propias aplicaciones si no puede compartirlas con otras personas, ya sea que eso signifique publicarlas comercialmente, compartirlas en línea o simplemente dárselas a alguien que conoce. ¡Compartir sus aplicaciones permite que otras personas se beneficien de su arduo trabajo!

Empaquetar aplicaciones de Python para su distribución ha sido típicamente un poco complicado, particularmente cuando se apunta a múltiples plataformas (Windows, MacOS y Linux). Esto se debe en parte a la necesidad de agrupar la fuente, los archivos de datos, el tiempo de ejecución de Python y todas las bibliotecas asociadas de una manera que funcione de manera confiable en el sistema de destino.

La buena noticia es que existe un sistema de construcción de paquetes diseñado específicamente para PyQt - fbs. Esto simplifica y automatiza gran parte del proceso de compilación para garantizar paquetes multiplataforma confiables y reproducibles.



fbs solo funciona con PyQt5 o PySide2 and Python version > 3.5. Sin embargo, está construido sobre PyInstaller que funciona con versiones anteriores de ambos. Si todavía está en Python 2.7, es posible que desee [consultar el manual de PyInstaller](#) directamente.

fbs: sistema de compilación fman

fbs es un sistema de empaquetado PyQt5 multiplataforma que admite la creación de aplicaciones de escritorio para Windows, Mac y Linux (Ubuntu, Fedora y Arch). Construido sobre *PyInstaller* envuelve algunas de las asperezas y define una estructura de proyecto estándar que permite que el proceso de construcción sea completamente automatizado. La API de recursos incluida es particularmente útil, ya que simplifica el manejo de archivos de datos externos, imágenes o bibliotecas de terceros, un problema común al empaquetar aplicaciones.



fbs tiene licencia GPL. Esto significa que puede usar el **fbs** system de forma gratuita en paquetes de código abierto distribuidos con la GPL. Para paquetes comerciales (o no GPL), debe comprar una licencia comercial. Ver la [página de licencias de fbs](#) para obtener información actualizada.

Si está impaciente, puede obtener los instaladores de Moonsweeper directamente para [Ventanas](#), [Mac OS](#) o [Linux \(Ubuntu\)](#).

Requisitos

fbs funciona fuera de la caja con PyQt PyQt5 y Qt para Python PySide2. El único otro requisito es PyInstaller que maneja el propio embalaje. Puede instalarlos en un entorno virtual (o en el entorno virtual de sus aplicaciones) para mantener limpio su entorno.



fbs solo es compatible con las versiones 3.5 y 3.6 de Python

```
1 python3 -m venv fbsenv
```

Una vez creado, active el entorno virtual ejecutando desde la línea de comandos -

```
1 #En Mac / Linux:  
2 fuente fbsenv / bin / activar  
3  
4 #Sobre Ventanas:  
5 llama fbsenv\slisiados\activate.bat
```

Finalmente, instale las bibliotecas necesarias. Para PyQt5 usaría -

```
1 pip3 instalar fbs PyQt5 PyInstaller== 3.4
```

O para Qt para Python (PySide2) -

```
1 pip3 instalar fbs PySide2 PyInstaller== 3.4
```

fbs instala una herramienta de línea de comandos **fbs** en tu camino que proporciona acceso a todos Comandos de gestión ** fbs **. Para ver la lista completa de comandos disponibles, ejecute **fbs**.

```
1 martin @ Martins-Laptop testapp $ uso de fbs:  
2 fbs          [-h]  
3           {startproject, ejecutar, congelar, instalador, sign_installer, repositorio, cargar, lanzamientos,  
4 e, prueba, limpiar, buildvm, runvm, gengpgkey, registrarse, iniciar sesión, init_licensing}  
5           ...  
6  
7 fbs  
8  
9 argumentos posicionales:  
10    {startproject, ejecutar, congelar, instalador, sign_installer, repositorio, cargar, liberar, probar, cl\ean,  
11 buildvm, runvm, gengpgkey, registrarse, iniciar sesión, init_licensing}  
12    inicio proyecto      Inicie un nuevo proyecto en el directorio actual Ejecute su  
13    correr               aplicación desde fuente  
14    congelar              Compile su código en un ejecutable independiente Cree un  
15    instalador            instalador por tu aplicación  
16    sign_installer       Instalador de firmas, para que el usuario's el sistema operativo  
17    repositorio          confía en él Genera archivos por actualizaciones automáticas  
18    subir                Cargue el instalador y el repositorio en fbs.sh Versión de bump y  
19    liberación            ejecute de forma limpia, congelada, ..., cargue Ejecute sus  
20    prueba               pruebas automatizadas  
21    limpio               Elimine los resultados de la compilación anterior Cree una  
22    buildvm              máquina virtual Linux. Por ejemplo: buildvm ubuntu Ejecuta  
23    runvm                una máquina virtual Linux. Por ejemplo: runvm ubuntu  
24    gengpgkey            Genera una clave GPGpor Firma de código Linux Crear una  
25    Registrarse          cuenta por cargando sus archivos Guarde los detalles de su  
26    acceso                cuenta en secret.json Genere claves públicas / privadas por  
27    init_licensing        Licencia  
28  
29 argumentos opcionales:  
30    - h, --ayuda          muestra esto ayuda mensaje y Salida
```

Iniciar una aplicación

Si está comenzando una aplicación PyQt5 de scratch, puede usar `fbs startproject` comando de administración para crear un código auxiliar de aplicación completo, funcional y empaquetable en la carpeta actual. Esto tiene la ventaja de permitirle probar (y continuar probando) la capacidad de empaquetado de su aplicación a medida que la desarrolla, en lugar de dejarla para el final.

- 1 `fbs startproject`

El comando lo guía a través de algunas preguntas, lo que le permite completar los detalles de su solicitud. Estos valores se escribirán en la fuente y la configuración de su aplicación. La aplicación básica se creará bajo `src/` carpeta en el directorio actual.

- 1 martin @ Martins-Laptop ~ \$ `fbs startproject Nombre de`
- 2 `la aplicación[MyApp] : Autor de HelloWorld [Martín] :`
- 3 Martín Fitzpatrick
- 4 Identificador de paquete de Mac (p.ej. `com.martin.helloworld`, opcional):



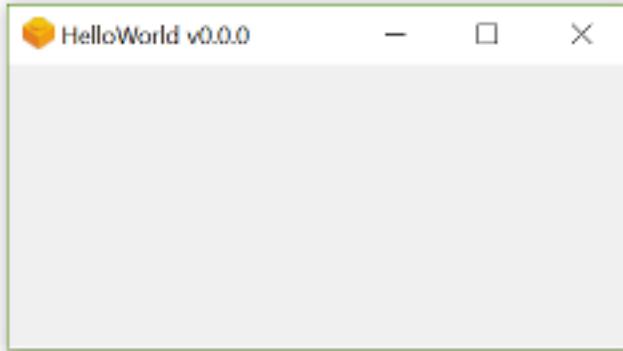
Si ya tiene su propia aplicación PyQt5 en funcionamiento, deberá a) usar la aplicación generada como guía para convertir la suya a la misma estructura, ob) crear una nueva aplicación usando `inioproyecto` y migrar el código.

Ejecutando su nuevo proyecto

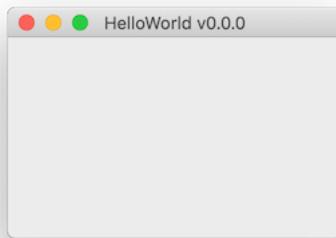
Puede ejecutar esta nueva aplicación utilizando lo siguiente `fbs` comando en la misma carpeta que ejecutó `inioproyecto` de.

- 1 ejecutar `fbs`

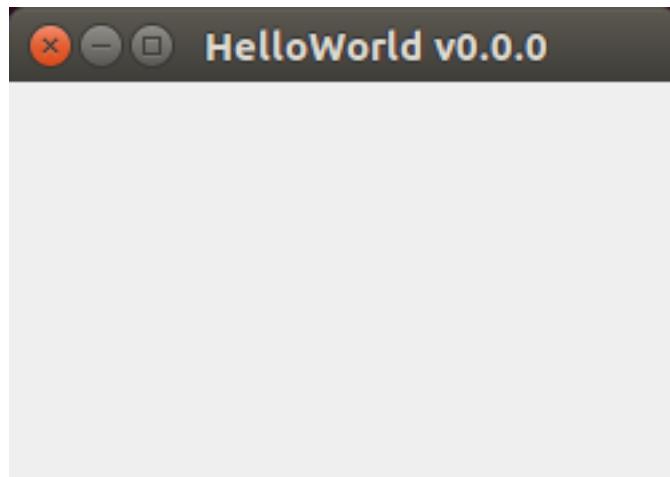
Si todo está funcionando, esto debería mostrarte una pequeña ventana vacía con el título de tus aplicaciones, ¿eh emocionante?



HelloWorld en Windows



HelloWorld en Mac



HelloWorld en Ubuntu

La estructura de la aplicación

los inicio proyecto El comando genera la estructura de carpetas requerida para un **fbs** Aplicación PyQt5. Esto incluye unsrc / build que contiene la configuración de compilación de su paquete, principal / iconos que contiene los iconos de la aplicación, y src / python para la fuente.

```
1 .
2 └── src
3     ├── construir
4     │   └── ajustes
5     │       ├── base.json
6     │       ├── linux.json
7     │       └── mac.json
8     └── principal
9         ├── iconos
10        |   ├── icon.ico
11        |   ├── README.md
12        |   └── base
13        |       ├── dieciséis.png
14        |       ├── 24.png
15        |       └── 32.png
```

```
diecisés
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```

    |
    |   └── 48.png
    |
    |   └── 64.png
    |
    └── linux
        ├── 1024.png
        ├── 128.png
        ├── 256.png
        └── 512.png
    |
    └── Mac
        ├── 1024.png
        ├── 128.png
        ├── 256.png
        └── 512.png
    |
    └── pitón
        └── main.py
```

Su aplicación PyQt5 básica se genera en src / main / python / main.py y es un ejemplo funcional completo que puede utilizar para basar su propio código.

```
1 de fbs_runtime.application_context importar ApplicationContextde
2 PyQt5.QtWidgets importar QMainWindow
3
4 importar sys
5
6 clase AppContext(ApplicationContext): #1. Subclase ApplicationContext
7     def correr(uno mismo): #2. Implementar correr()
8         ventana = QMainWindow()
9         versión = uno mismo.build_settings ['versión']
10        ventana.setWindowTitle ("HelloWorld v" + versión) ventana.
11        cambiar el tamaño250, 150) ventana.show()
12
13     regreso uno mismo.aplicación.exec_() #3. Finaliza run () con esta línea
14
15 si __nombre__ == '__principal__':
16     appctxt = AppContext ()
17     código_salida = appctxt.ejecutar ()
18     sys.salir (código_salida) #4. Crea una instancia de la subclase
#5. Invocar ejecutar ()
```

Si ha creado aplicaciones PyQt5 antes, notará que crear una aplicación con **fbs** introduce un nuevo concepto: el ApplicationContext.

los ApplicationContext

Al crear aplicaciones PyQt5, normalmente hay una serie de componentes o recursos que se utilizan en toda su aplicación. Estos se almacenan comúnmente en el QMainWindow o como variables globales que pueden complicarse un poco a medida que crece la aplicación. losApplicationContext proporciona una ubicación central para inicializar y almacenar estos componentes, además de proporcionar acceso a algunos **fbs** características.

los ApplicationContext El objeto también crea y contiene una referencia a un QApplication objeto - disponible bajo ApplicationContext.app. Cada aplicación de Qt debe tener una (y solo una) QApplication para mantener el bucle de eventos y la configuración principal. Sin **fbs** normalmente definiría esto en la base de su script y llamaría .ejecutivo () para iniciar el ciclo de eventos.

Sin **fbs** esto se vería algo así:

```
1 si __nombre__ == '__principal__':
2     aplicación = QApplication ()
3     w = MyCustomWindow ()
4     aplicación.exec_ ()
```

El equivalente con **fbs** sería -

```
1 si __nombre__ == '__principal__':
2     ctx = ApplicationContext ()
3     w = MyCustomWindow ()
4     ctx.aplicación.exec_ ()
```



Si quieres crear tu propia costumbre QApplication inicialización puede sobrescribir el .aplicación propiedad en su ApplicationContext subclase usando cached_property (vea abajo).

Este ejemplo básico es claro de seguir. Sin embargo, una vez que comience a agregar estilos personalizados y traducciones a su aplicación, la inicialización puede crecer bastante. Para mantener las cosas bien estructuradas **fbs** recomienda crear un `.correr` método en tu `ApplicationContext`.

Este método debe manejar la configuración de su aplicación, como crear y mostrar una ventana, finalmente iniciar el bucle de eventos en el `.aplicación` objeto. Este último paso se realiza llamando `self.app.exec_()` al final del método.

```
1 clase ApplicationContext(ApplicationContext):
2     def correr(uno mismo):
3         ...
4     regreso uno mismo.aplicación.exec_()
```

A medida que su inicialización se vuelve más complicada, puede dividir subsecciones en métodos separados para mayor claridad, por ejemplo:

```
1 clase ApplicationContext(ApplicationContext):
2     def correr(uno mismo):
3         uno mismo.setup_fonts()
4         uno mismo.setup_styles()
5         uno mismo.setup_translations()
6     regreso uno mismo.aplicación.exec_()
7
8     def setup_fonts(uno mismo):
9         #... hacer algo ...
10
11     def setup_styles(uno mismo):
12         #... hacer algo ...
13
14     def setup_translations(uno mismo):
15         #... hacer algo ...
```



Al ejecutarse el `.correr()` se llamará al método y se iniciará el bucle de eventos. La ejecución continúa en este bucle de eventos hasta que se sale de la aplicación, momento en el que `su.correr()` devolverá el método (con el código de salida apropiado).

Construyendo una aplicación real

La aplicación básica no hace mucho, por lo que a continuación veremos algo más completo: el *Barrendero* aplicación del capítulo anterior. El código fuente modificado es[disponible para descargar aquí](#).



Solo los cambios necesarios para convertir *Barrendero* a **fbs** están cubiertos aquí. Si quieres ver como *Barrendero* sí funciona, consulta el capítulo anterior. Los íconos de aplicaciones personalizados fueron creados usando arte de íconos por[Freepik](#).

El proyecto sigue la misma estructura básica que la aplicación estuba que creamos anteriormente.

```
1 .
2   ├── README.Maryland
3   ├── requisitos.TXT
4   ├── captura de pantalla-buscaminas1.
5   ├── captura de pantalla jpg-buscaminas2.
6   └── jpg src
7     ├── construir
8     │   └── ajustes
9     │       ├── base.json
10    │       ├── linux.json
11    │       └── Mac.json
12    └── principal
13      ├── Instalador.nsi
14      └── iconos
15          ├── licono.ico
16          ├── README.Maryland
17          └── base
18              ├── dieciséis.png
19              ├── 24.png
20              ├── 32.png
21              ├── 48.png
22              └── 64.png
23          └── linux
```

```
24      |   |   └── 1024.png
25      |   |   └── 128.png
26      |   |   └── 256.png
27      |   |   └── 512.png
28      |   └── Mac
29          |   └── 1024.png
30          |   └── 128.png
31          |   └── 256.png
32          |   └── 512.png
33      └── pitón
34          └── __en eso__.py
35          └── principal.py
36      └── recursos
37          └── base
38              └── imágenes
39                  └── bomba.png
40                  └── insecto.png
41                  └── reloj-Seleccione.cruz
42                  └── png.png
43                  └── bandera.png
44                  └── más.png
45                  └── cohete.png
46                  └── smiley-jajaja.
47                  └── smiley png.png
48          └── Mac
49              └── Contenido
50                  └── Información.plist
```

los src / build / settings / base.json almacena los detalles básicos sobre la aplicación, incluido el punto de entrada para ejecutar la aplicación con ejecutar fbs o una vez envasado.

```
1  {
2      "nombre de la aplicación": "Moonsweeper", "autor":
3          "Martin Fitzpatrick", "main_module": "src / main /
4          python / main.py", "versión": "0.0.0"
5
6 }
```

La secuencia de comandos *punto de entrada* está en la base de src / main / python / main.py. Esto crea el ApplicationContext objeto y llama al .correr() método para iniciar la aplicación.

```
1  si __nombre__ == '__principal__':
2      appctxt = ApplicationContext()
3      código_salida = appctxt.correr()
4      sys.salir(código_salida)
```

los ApplicationContext define un .correr() método para manejar la inicialización. En este caso eso consiste en crear y mostrar la ventana principal, luego iniciar el bucle de eventos.

```
1  de fbs_runtime.application_context importar ApplicationContext, \
2      cached_property
3
4
5  clase ApplicationContext(ApplicationContext):
6      def correr(uno mismo):
7          uno mismo.ventana principal.show()
8          regreso uno mismo.aplicación.exec_()
9
10     @cached_property
11     def ventana principal(uno mismo):
12         regreso Ventana principal(uno mismo) # Pasa contexto a la ventana.
13
14     #... recordar ...
```

los cached_property decorador

los .correr() accesos al método self.main_window. Notarás que este método está envuelto en `unfbs @cached_property` decorador. Este decorador convierte el método en una propiedad (como Python@propiedad decorador) y almacena en caché el valor de retorno.

La primera vez que se accede a la propiedad, el método se ejecuta y el valor de retorno se almacena en caché. En llamadas posteriores, el valor almacenado en caché se devuelve directamente sin ejecutar nada. Esto también tiene el efecto secundario de posponer la creación de estos objetos hasta que sean necesarios.

Puedes usar `@cached_property` para definir cada componente de la aplicación (una ventana, una barra de herramientas, una conexión a la base de datos u otros recursos). Sin embargo, tu no *tengo* usar el `@cached_property` - alternativamente, podría declarar todas las propiedades en su `ApplicationContext __init__` bloque como se muestra a continuación.

```
1 de fbs_runtime.application_context importar ApplicationContext
2
3 clase ApplicationContext(ApplicationContext):
4
5     def __en eso__(uno mismo, *argumentos, **kwargs):
6         súper(AppContent, uno mismo).__en eso__(*argumentos, **kwargs)
7
8         uno mismo.ventana = Ventana()
9
10    def correr(uno mismo):
11        uno mismo.ventana.show().__regreso
12        uno mismo.aplicación.exec_()
```

Accediendo a recursos con `.get_resource`

Las aplicaciones generalmente requieren archivos de datos adicionales más allá del código fuente, por ejemplo, iconos de archivos, imágenes, estilos (Qt's .qss archivos) o documentación. También es posible que desee agrupar bibliotecas o binarios específicos de la plataforma. Para simplificar esto `fbs` define una estructura de carpetas y un método de acceso que funcionan sin problemas en las versiones distribuidas y de desarrollo.

La carpeta de nivel superior `recursos/` debe contener una carpeta `base` más cualquier combinación de las otras carpetas que se muestran a continuación. La carpeta `base` contiene archivos comunes a todos

plataformas, mientras que las carpetas específicas de la plataforma se pueden utilizar para cualquier archivo específico de un sistema operativo determinado.

```
1 base/      # para archivos requeridos en todos los sistemas operativos
2 ventanas / # para archivos solo necesarios en Windows
3 Mac/       # "   "   "   "   "   Mac
4 linux /    # "   "   "   "   "   Linux
5 arco/      # "   "   "   "   "   Arch Linux
6 fedora /   # "   "   "   "   "   Debian Linux
7 ubuntu /   # "   "   "   "   "   Ubuntu Linux
```



Llevar los archivos al lugar correcto para cargarlos desde una aplicación distribuida en todas las plataformas suele ser una de las partes más fabulosas de la distribución de aplicaciones PyQt. Es realmente útil que **fbs** maneja esto por ti.

Para simplificar la carga de recursos desde su recursos/ carpeta en sus aplicaciones **fbs** proporciona el `ApplicationContext.get_resource()` método. Este método toma el nombre de un archivo que se puede encontrar en algún lugar de la recursos/ carpeta y devuelve la ruta absoluta a ese archivo. Puede utilizar esta ruta absoluta devuelta para abrir el archivo como de costumbre.

```
1 de fbs_runtime.application_context importar ApplicationContext, cached_property
2
3
4 clase ApplicationContext(ApplicationContext):
5
6     #... recordar...
7
8     @cached_property
9     def img_bomb(uno mismo):
10        regreso QImage (uno mismo.get_resource ('images / bug.png'))
11
12     @cached_property
13     def img_flag(uno mismo):
14        regreso QImage (uno mismo.get_resource ('images / flag.png'))
```

```
15  
diciéndole  
16     @cached_property  
17     def img_start(uno mismo):  
18         regreso QImage (uno mismo.get_resource ('images / rocket.png'))  
19  
20     @cached_property  
21     def img_clock(uno mismo):  
22         regreso QImage (uno mismo.get_resource ('images / clock-select.png'))  
23  
24     @cached_property  
25     def status_icons(uno mismo):  
26         regreso {  
27             STATUS_READY: QIcon (uno mismo.get_resource ("images / plus.png")),  
28             STATUS_PLAYING:   QIcon (uno mismo.get_resource ("images / smiley.png")), QIcon (  
29                 STATUS_FAILED: uno mismo.get_resource ("images / cross.png")), QIcon (uno mismo.  
30                 STATUS_SUCCESS: get_resource ("images / smiley-lol.png"))  
31         }  
32  
33     #... recordar ...
```

En nuestro *Barrendero* aplicación anterior, tenemos una *bomba* archivo de imagen disponible en `src / main / resources / base / images / bug.jpg`. Llamando `ctx.get_resource ('imágenes / bug.png')` obtenemos la ruta absoluta a ese archivo de imagen en el sistema de archivos, lo que nos permite abrir el archivo dentro de nuestra aplicación.



Si el archivo no existe `FileNotFoundException` se planteará en su lugar.

Lo útil de este método es que maneja de forma transparente las carpetas de la plataforma bajo `src / main / resources` dando prioridad a los archivos específicos del sistema operativo. Para examen-
Por favor, si el mismo archivo también estaba presente `src / main / resources / mac / images / bug.jpg` y llamamos `ctx.get_resource ('imágenes / bug.jpg')` obtendríamos la versión Mac del archivo.

Adicionalmente `get_resource` funciona tanto cuando se ejecuta desde la fuente como cuando se ejecuta una versión congelada o instalada de su aplicación. Si `turcursos/cargar` correctamente localmente, puede estar seguro de que se cargarán correctamente en sus aplicaciones distribuidas.

Utilizando el ApplicationContext desde la aplicación

Como se muestra arriba, nuestro ApplicationContext El objeto tiene propiedades en caché para cargar y devolver los recursos. Para permitirnos acceder a estos desde nuestro QMainWindowPodemos pasar el contexto y almacenar una referencia a él en nuestra ventana. __en eso__.

```
1 clase Ventana principal(QMainWindow):
2     def __en eso__(uno mismo, ctx):
3         super(Ventana principal, uno mismo).__en eso__()
4
5             uno mismo.ctx = ctx #Almacene una referencia al contexto para los recursos, et
6
7     #...     recortar...
```

Ahora que tenemos acceso al contexto a través de self.ctx podemos usarlo en cualquier lugar en el que queramos hacer referencia a estos recursos externos.

```
1 l = QLabel ()
2 l.setPixmap (QPixmap.desde imagen(uno mismo.ctx.img_bomb)) l.
3 setAlignment (Qt.AlignRight hb.addWidget(l | Qt.AlignVCenter)
4
5
6 #...     recorte ...
7
8 l = QLabel ()
9 l.setPixmap (QPixmap.desde imagen(uno mismo.ctx.img_clock)) l.
10 setAlignment (Qt.Alinear a la izquierda | Qt.AlignVCenter) hb.
11 addWidget (l)
```

La primera vez que accedemos self.ctx.img_bomb el archivo se cargará, el QImage creado y devuelto. En llamadas posteriores, obtendremos la imagen del caché.

```
1     def init_map(uno mismo):
2         # Agregar posiciones al mapa por X en
3         distancia(0, uno mismo.b_size):
4             por y en distancia(0, uno mismo.b_size):
5                 w = Pos(x, y, uno mismo.ctx.img_flag, uno mismo.ctx.img_start, uno mismo.ctx.l
6                 mg_bomb)
7                     uno mismo.red.addWidget(w, y, x)
8                     # Conecte la señal para manejar la expansión. w.
9                     clickeado.conectar(uno mismo.trigger_start) w.
10                    ampliable.conectar(uno mismo.expand_reveal) w.Oh
11                    no.conectar(uno mismo.juego terminado)
12
13     #...     recordar...
14
15     uno mismo.botón.setIcon (uno mismo.ctx.status_icons [STATUS_PLAYING])
16
17     #...     recordar...
18
19     def estado de actualización(uno mismo, estado):
20         uno mismo.estado = estado
21         uno mismo.botón.setIcon (uno mismo.ctx.status_icons [uno mismo.estado])
```

Esos son todos los cambios necesarios para obtener el *Barrendero* aplicación empaquetada con **fbs**. Si abre la carpeta de origen, debería poder iniciarla como antes.

```
1 ejecutar fbs
```

Si eso está funcionando, está listo para pasar a congelar y construir en el instalador.

Congelar la aplicación

Congelación es el proceso de convertir una aplicación Python en un ejecutable independiente que se puede ejecutar en la computadora de otro usuario. Utilice el siguiente comando para convertir el código fuente de la aplicación en un ejecutable independiente:

1 fbs congelar

El ejecutable resultante depende de la plataforma que *congelarencendido*: el ejecutable solo funcionará en el sistema operativo en el que lo construyó (por ejemplo, un ejecutable construido en Windows se ejecutará en otra computadora con Windows, pero no en una Mac).

- Windows creará un .exe ejecutable en la carpeta target / <AppName>
- MacOS X creará un .aplicación paquete de aplicaciones en target / <AppName>.app
- Linux creará un ejecutable en la carpeta target / <AppName>



En Windows, es posible que deba instalar el [SDK de Windows 10](#), a pesar de que **fbsle** indicará si este es el caso.

Creando un instalador

Si bien puede compartir los archivos ejecutables con los usuarios, las aplicaciones de escritorio se distribuyen normalmente con *instaladores* que manejan el proceso de colocar el ejecutable (y cualquier otro archivo) en el lugar correcto. Consulte las siguientes secciones para obtener notas específicas de la plataforma antes de crear



Usted debe *congelartu* aplicación primero *luego* crea el instalador.

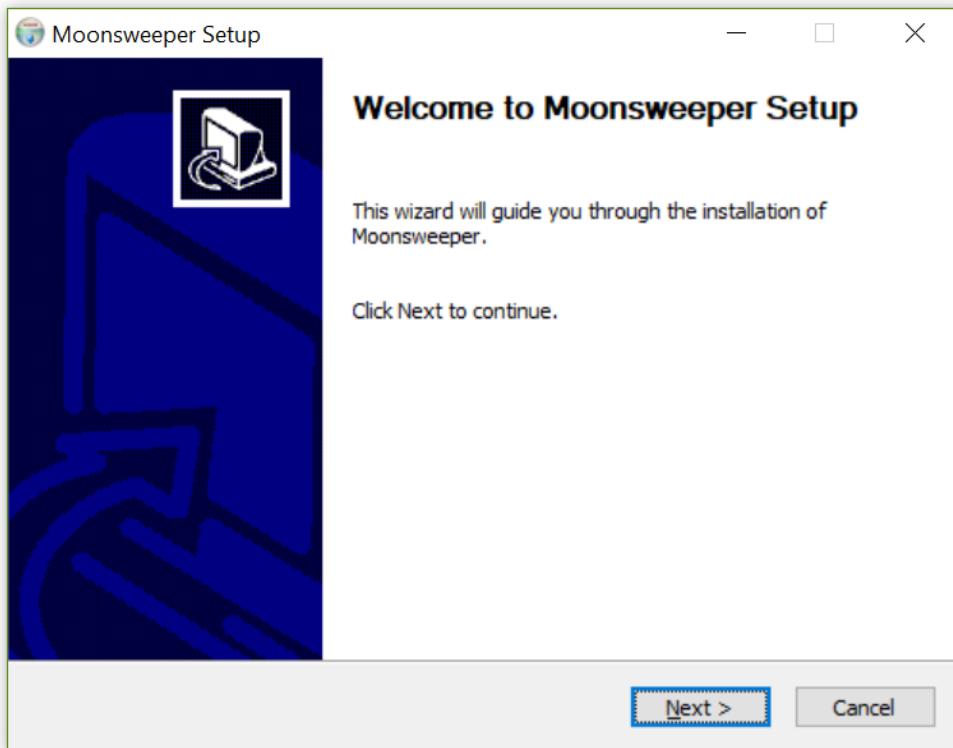
Instalador de ventanas

El instalador de Windows permite a los usuarios elegir el directorio de instalación del ejecutable y agrega su aplicación al menú Inicio del usuario. La aplicación también se agrega a los programas instalados, lo que permite que los usuarios la desinstalen.

Antes de crear instaladores en Windows, deberá instalar [NSIS](#) y asegúrese de que su directorio de instalación esté en su SENDEROS. Luego puede construir un instalador usando -

1 instalador de fbs

El instalador de Windows se creará en target /<AppName> Setup.exe.



El instalador de Windows NSIS

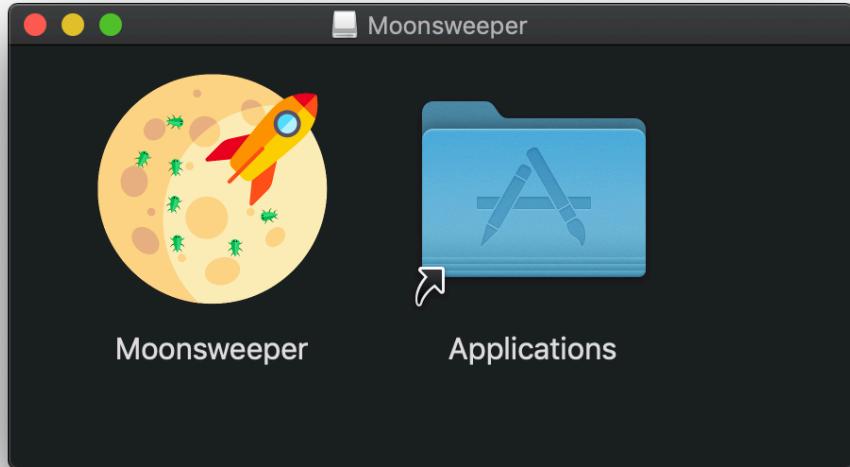
Descargar el *MoonsweeperSetup.exe*

Instalador de Mac

No hay pasos adicionales para crear un instalador de MacOS. Solo ejecuta el **fbs** comando -

1 instalador de fbs

En Mac, el comando generará una imagen de disco en target / <AppName>.dmg. Esta imagen de disco contendrá el paquete de la aplicación y un acceso directo a la carpeta Aplicaciones. Cuando sus usuarios la abren, pueden arrastrar la aplicación a la carpeta Aplicaciones para instalarla.



El instalador .dmg en Mac

Descargar el *Barrendero*.paquete dmg

Instalador de Linux

Para crear instaladores en Linux, debe instalar la herramienta Ruby [¡Effing gestión de paquetes!](#) - usar [la guía de instalación](#) para configurarlo. Una vez que esté en su lugar, puede usar el comando estándar para crear el archivo del paquete de Linux.

1 instalador de fbs

El paquete resultante se creará bajo el objetivo/ carpeta. Dependiendo de su plataforma, el archivo del paquete se denominará <Nombre de la aplicación> .deb, <Nombre de la aplicación> .pkg.tar.xz O <Nombre de la aplicación> .rpm. Sus usuarios pueden instalar este archivo con su administrador de paquetes.

[Descargar el Barrendero .archivo deb](#)

Más información sobre fbs

Más información sobre cómo **fbs** Los trabajos del sistema de embalaje se pueden encontrar en [el manual](#) que también presenta características más avanzadas [distribuir lanzamientos de aplicaciones Linux](#), [informar errores a la plataforma de registro de errores Sentry](#) y [agregar claves de licencia](#) a su software.

¿Que sigue?

Si ha llegado hasta aquí, debería estar bien encaminado para crear sus propias aplicaciones. Pero aún queda mucho por descubrir.

Si recibió una copia de este libro de otra persona (¡muy bien!), Es posible que le interese [el sitio web adjunto](#), que contiene nuevos tutoriales, cursos de vídeo y aplicaciones de demostración con regularidad. Al igual que este libro, todas las muestras tienen licencia del MIT y se pueden mezclar gratis en su propia aplicación.

Finalmente, este libro tiene la licencia CC-BY-NC-SA. Esto significa que debe sentirse libre de darle una copia a alguien que conoce (o no conoce). Ayude a compartir el conocimiento y la diversión de crear aplicaciones de escritorio con Python.

Gracias por leer, y si tiene algún comentario o sugerencia, hágamelo saber.

El curso de video

Gracias por comprar *;Cree aplicaciones GUI simples!*

Si le gustó este libro, también puede disfrutar del curso en video que lo acompaña, que puede comprar en <https://www.learnpyqt.com/purchase>



Si compró este libro, recuerde reclamar su cuenta en <https://www.learnpyqt.com> para obtener el precio de actualización.

Para obtener los tutoriales más recientes, visite [tipssandcodesampleseehttps://www.learnpyqt.com/courses/](https://www.learnpyqt.com/courses/)

Recursos

Esta sección es una breve lista de recursos que pueden resultarle útiles para escribir sus aplicaciones Qt. Incluyen fuentes de buena documentación y ayuda, y también recursos útiles para hacer que sus aplicaciones se vean y funcionen bien.

Tutoriales

Para obtener tutoriales, sugerencias y ejemplos de código actualizados, consulte el sitio de tutoriales asociado para este libro en <https://www.learnpyqt.com>

Documentación

Recurso	Descripción
Documentación Qt5.5	
Documentación Qt 4.8	
Documentación de la biblioteca PyQt4	
Documentación de la biblioteca PyQt5	
Documentación de la biblioteca PySide	

Conjuntos de iconos

Los siguientes conjuntos de iconos son de uso gratuito, con la licencia adecuada, para que las aplicaciones tengan un aspecto más profesional y coherente. El conjunto de Fuga son los iconos sugeridos y utilizados en los ejemplos de este libro, pero los demás también merecen una mirada.

Recurso	Descripción	Licencia
Fuga por p.yusukekamiyamane	3,570 iconos de 16x16 en formato PNG	CC BY 3.0
Diagona por p.yusukekamiyamane	400 16x16 y 10x10 iconos en formato PNG	CC BY 3.0

Recurso	Descripción	Licencia
Iconos de tango por The Tango Desktop Project	Iconos que utilizan el tema de color del proyecto Tango.	Dominio publico

Código fuente

El código fuente completo, todos los ejemplos de este libro, está disponible para descargar desde [aquí](#).

Derechos de autor

Este libro tiene la licencia Creative Commons Attribution Share-alike Noncommercial License (CC BY-NC-SA) y (C) 2015 Martin Fitzpatrick.

- Eres gratis para compartir copias no modificadas de este libro con cualquiera que eliges.
- Si modifica este libro y distribuye su versión alterada, debe distribuirse bajo la misma licencia.
- No se le permite vender este libro o sus derivados en ningún formato.
- Si desea apoyar al autor, puede *legalmente* comprar una copia directamente del autor (es).

Las contribuciones y correcciones de los lectores (CC BY-NC-SA) son bienvenidas.