Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

VERSIÓN PRELIMINAR

Índice general

VERSIÓN PRELIMINAR

Parte I
Python

Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Interfaces gráficas

Las interfaces que nos ofrecen los programas en un sistema informático se dividen principalmente entre si están pensada para otras aplicaciones o para personas. Estas últimas a su vez se dividen en varios tipos, siendo las gráficas nuestro motivo de estudio en este capítulo.

También conocidas como GUI (del inglés *graphical user interface*), las interfaces gráficas utilizan un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles. Es una herramienta muy poderosa a la hora de proporcionar un entorno visual sencillo para permitir la utilización de nuestro sistema por quienes necesiten usarlo.

Módulo	Versión
matplotlib	3.9.2
PyQt6	6.7.1
scipy	1.14.1

Código disponible

No es un concepto para nada nuevo, las interfaces gráficas datan de la década de 1970, y por eso hay infinidad de variaciones. Sólo a modo de ejemplo, podemos citar algunos frameworks que son multiplataforma, como Qt, GTK+ o Tk, y otros que son específicos a algún sistema operativo, como Cocoa para MacOS o las Microsoft Foundation Classes (MFC) para Windows. Y no sólo tenemos que pensar en computadoras de escritorio o laptops, por ejemplo con Kivy también podemos armar programas con interfaces gráficas para teléfonos.

Para este capítulo vamos a utilizar Qt, aunque la gran cantidad de conceptos sobre los que hablaremos son comunes a todos los frameworks de interfaces gráficas, más allá de sus características particulares.

1.1. Qt

Qt es un framework para construir interfaces gráficas (aunque también tiene su utilización en sistemas no gráficos) que corre en los tres principales sistemas operativos de escritorio (Linux, MacOS y Windows), algunos modelos de teléfonos y sistemas integrados, y hay planes a futuro para que funcione en Android e iOS.

Al ser una sigla vale la pena la aclaración con respecto a su pronunciación: en castellano decimos “cu-té”, mientras que en inglés se pronuncia “quiút” (no “quiú-ti”).

En realidad no vamos a usar Qt directamente, porque tendríamos que programar en C++, sino que vamos a usar PyQt, un *wrapper* alrededor del framework base. Mencionaremos luego algunas diferencias, pero para todos los efectos prácticos, seguiremos hablando de Qt o PyQt indistintamente en el resto del capítulo.

Qt, como la mayoría de los frameworks de interfaces gráficas, es un sistema asincrónico (si todavía no leyeron la sección sobre Async?? es un buen momento, pero no es estrictamente necesario para entender el resto de este capítulo). Por eso podemos decir que un programa de interfaz gráfica está basado en eventos: una vez que arrancó y terminó de armar todos los elementos en una ventana el sistema no estará realmente haciendo nada hasta que reciba eventos y los procese.

Estos eventos pueden venir de distintas fuentes: entrada del usuario (vía el teclado o el mouse, por ejemplo) o del sistema (alarmas luego que pasó determinado tiempo, escritura o lectura en distintas entradas/salidas, etc). Cuando sucede el evento el reactor que maneja el sistema asincrónico (Qt mismo) ejecutará la función correspondiente para manejarlo.

Tengamos en cuenta que no todos los eventos nos interesan en una aplicación de interfaz gráfica. Aunque esos eventos son todo lo que le puede "suceder" a una ventana de nuestra aplicación, hay muchos que están manejados directamente por los componentes que ya agregamos a esa ventana. Por ejemplo, si dejamos el mouse quieto sobre uno de los iconos de la barra de herramientas de nuestra aplicación aparecerá un mensaje, pero supervisar al mouse para detectar la situación y hacer aparecer ese mensaje no es algo que tengamos que programar de nuestro lado, es una funcionalidad integrada del componente mismo que usamos para tener ese icono en esa barra de herramientas, al cual sólo le tenemos que indicar qué mensaje queremos que aparezca en ese caso.

Por otro lado, hay muchas veces que sí queremos reaccionar a los distintos eventos. Por ejemplo, ponemos un botón en nuestra aplicación y queremos ejecutar una función "nuestra" cuando se presiona ese botón. Ya veremos cómo se hace esto.

Estos componentes de la interfaz gráfica se llaman *widgets* (la traducción al castellano es "artilugio" pero en realidad siempre usamos *widget*) y permiten hacer uso de la aplicación y los servicios que esta provee. Como ejemplo, mencionamos aquí algunos widgets utilizando su nombre en Qt y la funcionalidad que ofrecen (estos son widgets bien simples, en la mayoría de otras interfaces gráficas encontraremos widgets similares a estos con nombres parecidos o casi iguales):

- QLabel: muestra un texto o imagen
- QLineEdit: permite ingresar una línea de texto
- QTextEdit: permite ingresar un texto de cualquier cantidad de líneas
- QPushButton: un botón para invocar una acción al ser apretado
- QRadioButton: permite elegir una opción (y sólo una) entre muchas
- QCheckBox: permite elegir una o más opciones
- QDialog: otra ventana con una funcionalidad acotada y específica usada para recabar información desde la ventana que la genera

Si pensamos en cualquier programa de interfaz gráfica reconoceremos que usan alguna combinación de los widgets recién mencionados, entre otros. Más adelante en este capítulo mismo usaremos algunos de esos widgets y otros también.

1.1.1. Unas palabras sobre la documentación

Como mencionamos antes, en este capítulo vamos a utilizar PyQt para explicar los conceptos de interfaces gráficas y armar una aplicación ejemplo.

Si buscamos documentación online sobre PyQt encontraremos muchos tutoriales y ejemplos de cómo hacer distintas aplicaciones o estructurar distintas funcionalidades haciendo el programa en Python. Pero a la hora de buscar la documentación de referencia de los distintos widgets y otros componentes de PyQt, veremos que casi no hay nada, y siempre en algún punto terminan apuntando a la referencia escrita para Qt mismo, en C++.

Esto sucede porque PyQt es una capa arriba de la biblioteca de Qt en C++, permitiendo (más allá de proveer alguna simplicidad extra) poder usarla desde Python. Reescribir toda la documentación para hacerla en PyQt sería un esfuerzo mayúsculo, por el tamaño de la misma y por el hecho de que cambia continuamente (especialmente entre las versiones principales de la librería, de Qt 4 a Qt 5, por ejemplo). Esto, sumado a que entender la documentación en C++ (aunque usemos PyQt) es bastante sencillo, hace que nunca se encare ese esfuerzo.

Un buen punto de entrada a la documentación de referencia es la lista de todas las clases disponibles, porque en general nos va a suceder que sabemos que tenemos que usar un determinado widget, pero no sabemos cómo explotarlo al máximo o queremos aprender de sus distintas funcionalidades. Desde allí podremos ir a cualquiera de las clases que usaremos cuando programamos en (Py)Qt.

Entonces la idea es transmitirles que intenten usar la documentación escrita en C++, que a priori parecerá muy críptica pero es cuestión de acostumbrarse a leerla para poder aprovecharla. Para facilitarles este proceso les recomendamos en hacer foco en las siguientes secciones cuando vayan a la documentación de referencia de alguno de los widgets:

- *Public Functions* y *Reimplemented Public Functions*: aquellos métodos del widget que podemos llamar para explotar todas sus posibilidades.
- *Signals*: aquellas señales a las que les podemos conectar una función nuestra para reaccionar a eventos en el widget (o directamente conectarles un *slot*... hablaremos luego más en profundidad sobre *signals* y *slots*)
- *Detailed Description*: una explicación a fondo de para qué sirve el widget, donde normalmente se mencionan distintas opciones y capacidades funcionales, y es un gran punto de entrada no sólo para aprender sobre el widget sino para entender los contextos en los que se lo puede utilizar.

Cabe destacar que todas estas clases tienen una estructura de herencia, y también tenemos que acostumbrarnos a eso cuando busquemos información. Por ejemplo, si vamos a la documentación del QPushButton (el botón "normal" de cualquier interfaz gráfica) veremos que no tiene ninguna señal listada... pero encontramos que hereda de QAbstractButton (la clase de la que derivan los distintos tipos de botones) y ahí están las señales que estábamos buscando, como `clicked`, que se emite cuando el botón es activado (o sea, apretado y soltado mientras el cursor del mouse está dentro del botón).

Luego podrán explorar más a fondo, pero ya con esta información podrán acceder a la documentación de la mayor parte de la funcionalidad que van a necesitar.

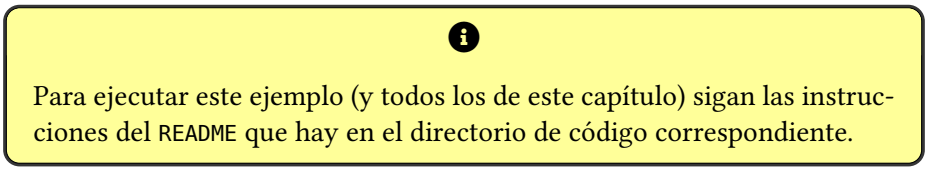
1.2. Una aplicación mínima

Para empezar a entender las distintas funcionalidades que podemos encontrarnos en una aplicación gráfica, y cómo usar esas funcionalidades desde PyQt, armemos una aplicación muy básica.

Este mínimo de código que presentaremos en esta sección nos permitirá charlar sobre distintos conceptos fundamentales, y luego podremos encarar una aplicación más real.

Arranquemos con la "cáscara" mínima que necesitaremos siempre (el código en `01.empty.py`).

```
1  import sys
2
3  from PyQt6.QtWidgets import QApplication, QWidget
4  from PyQt6.QtGui import QIcon
5
6
7  class Window(QWidget):
8      def __init__(self):
9          super().__init__()
10         self.resize(250, 150)
11         self.setWindowTitle("Primer ejemplo")
12         self.setWindowIcon(QIcon("icon.png"))
13
14
15 app = QApplication(sys.argv)
16 window = Window()
17 window.show()
18 sys.exit(app.exec())
```



Para ejecutar este ejemplo (y todos los de este capítulo) sigan las instrucciones del `README` que hay en el directorio de código correspondiente.

Aquí ya vemos las tres secciones que se repetirán en todos los programas. Primero, una secuencia de *imports* (líneas 1 a 4), que irá creciendo a medida que vayamos necesitando más widgets y funcionalidades. Luego, todas las clases que compondrán la estructura de nuestra aplicación; en este caso tenemos sólo una (líneas 7 a 12), que explicaremos luego. Y finalmente una serie de instrucciones para que arranque el sistema (líneas 15 a 18).

Estas últimas líneas no cambiarán para nada en todo nuestro ejemplo, y en aplicaciones más reales sólo se verán afectadas según alguna necesidad de procesamiento de opciones de línea de comando (que en aplicaciones de interfaz gráfica no es normal tener). Lo más importante de esta secuencia es la inicialización de `QApplication`, la cual maneja la configuración a nivel QT y el flujo de control de la aplicación; contiene el *event loop* principal que procesará y despachará todos los eventos, tanto los generados por los elementos de las ventanas como de otras fuentes.

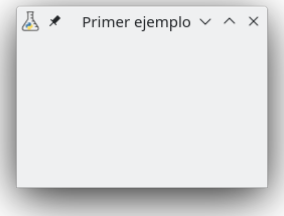
Luego de inicializar `QApplication` (línea 15) instanciamos nuestra ventana y la mostramos (líneas 16 y 17), lo cual no podemos hacer antes (necesitan tener el sistema ya inicializado). Pero si terminamos ahí, nuestro programa no hará nada. Es que recién al ejecutar el `app.exec()` estamos dandole el control a *event loop* de Qt, que se quedará manejando el sistema hasta que nuestra

aplicación termine (devolviendo un código final que lo pasamos directamente a `sys.exit()` para que sea el código de retorno del proceso).

Enfoquémonos entonces en la parte del medio de nuestro programa.

La clase `Window` hereda de `QWidget`, y lo único que hacemos es ajustar tres detalles en tiempo de inicialización (luego de primero inicializar la clase padre en la línea 9: recordemos que cuando redefinimos un método que ya posee la clase de la que derivamos, al llamarse al método sólo se ejecutará el de la clase hija, y si queremos que también se ejecute el método de la clase ancestra debemos llamarlo explícitamente). Las tres líneas donde configuramos el *widget* son bastante autoexplicativas: cambiamos el tamaño de la ventana, le ponemos un título, y le configuramos un icono.

En general construiremos toda nuestra aplicación gráfica heredando de determinados widgets y agregándole la funcionalidad particular que necesitamos en cada caso. Esta estructura no es obligatoria, pero al ser PyQt un framework fuertemente estructurado alrededor de un árbol de clases, si continuamos con esa metodología nos quedará un código prolijo y fácil de entender.



En realidad una ventana vacía tampoco hace a una aplicación, agreguemos algo simple (un texto) y una forma de interacción (un botón que haga algo).

Pero antes, ahora que vamos a agregar widgets en nuestra ventana, unas palabras sobre dónde y cómo agregarlos.

En los comienzos de las aplicaciones gráficas, los elementos dentro de una ventana tenían una posición fija (incluso hoy en día se construyen aplicaciones así), donde cada widget se ubicaba en una coordenada específica dentro de la ventana. Esto siempre termina en una pobre experiencia en el uso, ya que si cambia el largo de los textos (por traducciones), o el tamaño de la ventana, o el tamaño de la tipografía (por configuraciones de accesibilidad del sistema), todo queda amontonado solapándose los elementos de la ventana o demasiado disperso.

Hay una forma superadora de ubicar elementos dentro de una ventana que es a través de *layouts* (“disposiciones”, en castellano, pero se usa el término en inglés). Por ejemplo, si queremos poner tres widgets uno al costado del otro usaremos un *layout* horizontal y agregaremos los tres widgets al layout. Tanto el tamaño del layout como de los widgets dentro se ajustarán dinámicamente en función del espacio mínimo que cada uno tiene que ocupar (por su contenido) y el tamaño máximo manejado por el usuario (por ejemplo al agrandar la ventana). Estos layouts tienen varias configuraciones que permiten controlar su comportamiento y ofrecer una solución elegante, como espacio entre los widgets, qué widget crecería con prioridad si se agranda la ventana, etc.

Obviamente hay layouts verticales y horizontales, y a través de su composición podemos lograr cualquier interfaz, pero también existen algunos más específicos, como un layout de grilla

para aquellas disposiciones que son claramente una cuadrícula.

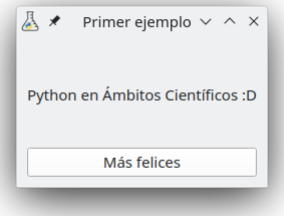
El segundo ejemplo lo tenemos en `02.content.py`, del cual mostraremos aquí sólo la clase del widget, ni la sección donde importamos los módulos necesarios ni las líneas para iniciar el programa. Para nuestra ventana ejemplo usaremos sólo un layout vertical, el cual instanciamos y configuramos como layout del widget en las líneas 15 y 16.

```
8  class Window(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.resize(250, 150)
12         self.setWindowTitle("Primer ejemplo")
13         self.setWindowIcon(QIcon("icon.png"))
14
15         layout = QVBoxLayout()
16         self.setLayout(layout)
17
18         self.message = "Python en Ámbitos Científicos"
19         self.label = QLabel(self.message)
20         self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
21         layout.addWidget(self.label)
22
23         button = QPushButton("Más felices")
24         button.clicked.connect(self.happier)
25         layout.addWidget(button)
26
27     def happier(self):
28         self.message += " :D"
29         self.label.setText(self.message)
```

Los dos bloques de código siguientes crean y agregan al layout el texto y el botón.

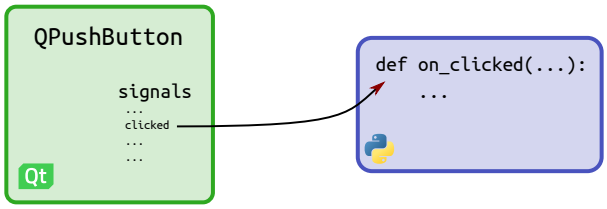
El texto lo agregamos a través de un `QLabel`, al cual le indicamos que lo alinee al centro (atención a la forma de indicar esto, no se usan cadenas como `'center'` sino flags, ya predefinidas en el framework), y finalmente agregamos el widget al layout (líneas 18 a 25). Notar que tanto el mensaje como el widget en sí los creamos en la instancia de la ventana: esto es sólo necesario porque luego usaremos ambos desde un método de la instancia.

Al botón lo creamos directamente con el texto que queremos que tenga, luego conectamos su señal `clicked` a un método que creamos abajo, y agregamos el widget al layout (líneas 23 a 25). El método `happier` entonces se ejecutará cada vez que presionemos el botón: modificará el mensaje y le indicará al `QLabel` que tenemos en la ventana que use este nuevo texto.

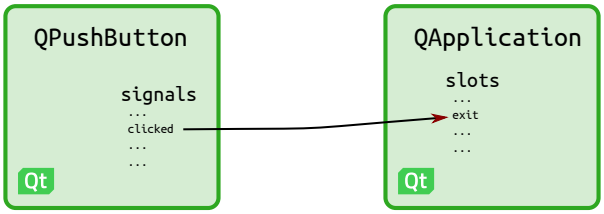


El botón que usamos, como la mayoría de los widgets, está diseñado para emitir señales en respuesta a uno o más eventos. Nosotros utilizamos `clicked`, pero tiene otras (en realidad, como ya mencionamos antes, las hereda de `QAbstractButton`). Incluso podemos crear nosotros nuestras propias señales (y lo haremos casi al final de este capítulo)

Las señales, en general, pueden conectarse a cualquier función de Python (o más de una), las que pueden estar definidas para no recibir nada (como en nuestro caso) o el evento que generó la señal (cuando ese evento nos trae información que necesitemos). Es que las señales pueden generarse por distintas situaciones: muchas son iniciadas por una acción de la persona que usa la interfaz pero eso no es una regla, lo que dispara la señal dependerá de cual sea la señal. Tengamos en cuenta que la señal en sí misma no hace nada, cuando se dispare sólo terminará ejecutando una o más funciones que hayamos conectado manualmente o conectadas por el framework mismo.



Por otro lado, cuando las conectamos no siempre tenemos que conectarlas a funciones de Python. También podemos conectarlas directamente a *slots* que los widgets tienen. De esta manera podemos relacionar widgets entre sí y que unos reaccionen a eventos de otros. Aunque las opciones son limitadas, porque dependemos de los slots que tenga definido cada widget, es interesante estar atentos a la posibilidad ya que es una solución extremadamente eficiente (porque todo se resuelve dentro del framework compilado sin salir a la capa de Python).



Para terminar la sección y antes de arrancar con una aplicación más real, nos permitimos un comentario sobre por qué diseñamos la interfaz gráfica de la aplicación usando código y no un diseñador gráfico como *Qt Designer*. Esta aplicación (no es la única, cada framework tiene algo similar) permite armar la ventana *visualmente*. Esta forma de construir ventanas era de gran ayuda en la época en que los widgets se acomodaban a mano en posiciones específicas, pero al usar layouts esa ventaja se pierde. Construir nuestros widgets especializados en código nos permite lograr una mejor reutilización del código que armamos (por ejemplo, podemos tener un layout con una separación específica entre widgets y reutilizarlo todo el tiempo), y también es mucho más fácil buscar ayuda sobre cómo realizar determinadas cosas.

1.3. Una aplicación real

En esta sección vamos armar una aplicación completa con una doble intención: por un lado mostraremos varios widgets y técnicas necesarias cuando queremos dar una buena usabilidad sobre nuestro programa, y la otra es que ustedes tengan una aplicación "base" para construir las suyas, un código que puedan modificar y adaptar para cumplir vuestras necesidades.

Es por esto último que pensamos en una aplicación que bien podría ser típica en un entorno científico: permitir cargar un archivo de datos y mostrar un gráfico generado a partir de esos datos, según alguna opción. En nuestro caso el archivo está en formato CSV y los datos son coordenadas en el plano, el cálculo que hacemos es ajustar una función polinómica a esos puntos (dejando que se elija en la interfaz el grado de la función) y luego graficar los puntos en sí y la función ajustada.

Por razones pedagógicas partiremos del ejemplo mínimo que traemos de la sección anterior e iremos agregando gradualmente nuevas funcionalidades, explicándolas en cada caso. Pero si no les importa el *spoiler*, pueden arrancar por el final y ejecutar el código `09.complete.py` para ver la aplicación terminada.

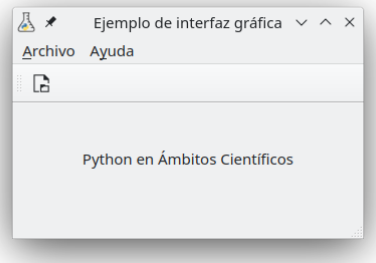
Pero, decíamos, vayamos por partes. El primer gran cambio con respecto a lo que teníamos antes es que no queremos una ventana así no más, sino una ventana "principal" a la que le podamos agregar menú, barra de tareas, e incluso una barra de estado. Tenemos todo esto implementado en `03.app.py`, del que mostramos sólo la clase principal:

```
14 class Window(QMainWindow):
15     """Ventana principal."""
16 
17     def __init__(self):
18         super().__init__()
19         self.resize(350, 200)
20         self.setWindowTitle("Ejemplo de interfaz gráfica")
21         self.setWindowIcon(QIcon("icon.png"))
22 
23         # menú y barra de herramientas
24         toolbar = QToolBar("main-toolbar")
25         self.addToolBar(toolbar)
26         self._status_bar = QStatusBar(self)
27         self.setStatusBar(self._status_bar)
28         menubar = self.menuBar()
29 
30         # las diferentes acciones tanto para el menu como para la barra de herramientas
31         open_action = QAction(QIcon.fromTheme("document-open"), "Abrir archivo", self)
32         open_action.setToolTip("Abrir archivo de datos")
33         open_action.triggered.connect(self.on_open)
34         open_action.setShortcut('Ctrl+A')
35         quit_action = QAction(QIcon(), "Salir", self)
36         quit_action.triggered.connect(app.exit)
37         about_action = QAction(QIcon(), "Acerca de...", self)
38         about_action.triggered.connect(self.on_about)
39 
40         # configuramos la barra de herramientas
41         toolbar.addAction(open_action)
42 
43         # configuramos el menú
44         menu = menubar.addMenu("&Archivo")
```

```
45          menu.addAction(open_action)
46          menu.addAction(quit_action)
47          menu = menubar.addMenu("A&yuda")
48          menu.addAction(about_action)
49
50          self._status_bar.showMessage("Comenzando", 3000)
51
52          label = QLabel("Python en Ámbitos Científicos")
53          label.setAlignment(Qt.AlignmentFlag.AlignCenter)
54          self.setCentralWidget(label)
55
56      def on_open(self):
57          pass
58
59      def on_about(self):
60          pass
```

En este código vemos las distintas secciones donde armamos la funcionalidad recién indicada, pero prestemos atención a que ya no tenemos un widget genérico sino que heredamos de `QMainWindow`. Entonces, además de ese detalle importante y las primeras líneas donde fijamos el tamaño de la ventana y ponemos el título y el icono…

- creamos la barra de herramientas y la barra de estado, las agregamos a la ventana principal, y le pedimos la barra de menú (líneas 24 a 28)
- creamos distintas acciones como paso separado, ya que luego usaremos esas acciones tanto para el menú como para la barra de tareas (líneas 31 a 38); mencionaremos abajo algunos detalles de estas líneas para explicarlas mejor
- agregamos una sola acción a la barra de tareas, para abrir el archivo (línea 41)
- agregamos todas las acciones al menú, bajo dos secciones principales: Archivo y Ayuda (líneas 44 a 48); notar que el nombre de cada sección en el menú tiene un `&` antes de alguna letra: esa letra estará subrayada al mostrarse y permitirá acceder a la sección con la tecla *Alt* más esa letra
- mostramos un mensaje en la barra de estado que desaparecerá luego de 3 segundos (línea 50)
- y finalmente agregamos un texto como todo widget central en la ventana (que luego más adelante reemplazaremos por contenido más avanzado)



Volvamos a la parte de creación de las acciones para resaltar algunos detalles.

La primer acción, para abrir el archivo de datos, es especial porque es la que usaremos también en la barra de tareas. Por eso es que nos interesa que tenga un icono con sentido (que no cargamos de un archivo, sino que lo tomamos del stock de imágenes del sistema operativo) y también un *tooltip* (que es el texto que aparece cuando dejamos unos segundos el mouse quieto arriba del icono de la barra de tareas).

Por otro lado, las acciones de abrir el archivo y de mostrar el "acerca de" de la aplicación tienen su señal `triggered` conectada a métodos de nuestra clase (definidos en las líneas 56 a 60, pero todavía sin implementar). Notar que a la acción para salir del programa la conectamos directamente al slot `exit` de la aplicación ya que no hace falta que hagamos nada específico en el medio.

El próximo paso en la aplicación es permitir abrir un archivo y cargar los datos correspondientes. Es hora de implementar el método `on_open`, el cual tenemos en `04.load.py`:

```
59      def on_open(self):
60          """Abre un nuevo archivo de datos y refresca el widget principal."""
61          self._status_bar.showMessage("Abriendo archivo con datos", 3000)
62
63          # abrimos un diálogo para que se seleccione un archivo
64          filepath, filters = QFileDialog.getOpenFileName(self, "Abrir archivo")
65          if not filepath:
66              self._status_bar.showMessage("Ningún archivo seleccionado", 3000)
67              return
68          filepath = pathlib.Path(filepath)
69
70          # abrimos el archivo y levantamos los datos
71          try:
72              with open(filepath) as csvfile:
73                  reader = csv.reader(csvfile)
74                  raw_data = list(reader)
75          except Exception as exc:
76              self._status_bar.showMessage(f"No se pudo abrir el archivo correctamente: {exc}", 3000)
77              return
78
79          # convertimos los datos a punto flotante, validando largos y descartando las listas vacias
80          data = []
81          for idx, row in enumerate(raw_data, 1):
82              if not row:
83                  continue
84              if len(row) != 2:
85                  self._status_bar.showMessage(
86                      f"La linea {idx} del archivo no tiene exactamente dos columnas", 3000)
87                  return
88              try:
89                  converted = [float(datum) for datum in row]
90              except ValueError:
91                  self._status_bar.showMessage(
92                      f"La linea {idx} del archivo tiene datos inválidos", 3000)
93                  return
94              data.append(converted)
95
96          # tenemos los datos ok!
97          self._status_bar.showMessage("Archivo abierto correctamente", 3000)
```

```
98
99          label = QLabel(f"Registros cargados: {len(data)}")
100         label.setAlignment(Qt.AlignmentFlag.AlignCenter)
101         self.setCentralWidget(label)
```

Podemos separar este código en estas partes conceptuales:

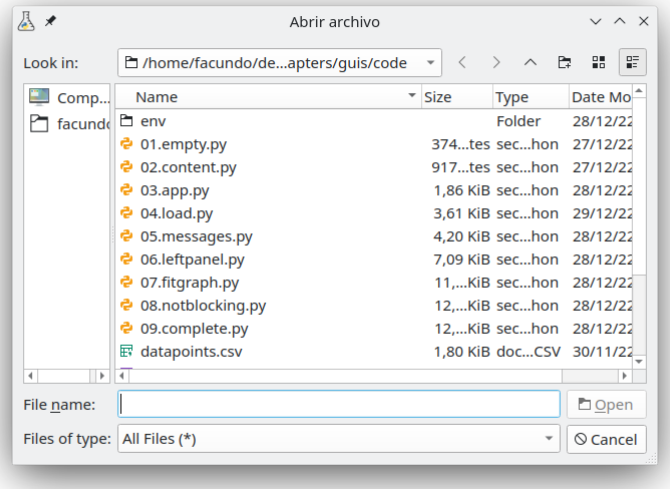
- mostramos un mensaje en la barra de estado informando que estamos abriendo el archivo de datos (línea 61)
- abrimos un "diálogo" para dejar elegir un archivo (hablaremos más sobre diálogos a continuación) en la línea 64, el cual nos puede dar un resultado vacío (en ese caso informamos que no hay archivo seleccionado), o el path a un archivo
- si tenemos un path válido lo abrimos, leemos, y validamos que tenga dos columnas de números, saliendo por error ante cualquier problema (líneas 70 a 94)
- si procesamos todo bien llegamos a la línea 97 donde informamos que se abrió el archivo correctamente
- finalmente cambiamos el texto central con un mensaje que muestra la cantidad de registros cargados (líneas 99 a 101)

Esa gran cantidad de líneas, de la 71 a la 93, es donde procesamos el archivo de datos, validando en cada uno de los pasos que la estructura y el contenido estén como esperamos (reportando un error y abortando la carga si no). Primero abrimos el archivo y lo leemos completamente con el módulo `csv`, para abortar si no se pudo abrir o está corrupto a nivel estructura (líneas 70 a 77). Luego procesamos cada línea validando que tenga dos columnas, y convirtiendo a número cada uno de los valores con `float` (líneas 79 a 94). Al final tenemos en `data` a toda la información correctamente convertida.

Este procesamiento puede parecer demasiado detallado, pero es importante considerar todos los casos de error e informar correctamente cuando algo sale mal: es la diferencia entre un programa que es fácil de usar y nos guía ante cualquier problema, y un programa que falla por "razones misteriosas". Es verdad que la forma de mostrar cada error parece repetitiva y un poco oscura (sólo un mensaje en la barra de estado), pero vamos a mejorar esto en próximas iteraciones del código.

Vemos en la imagen cómo quedaría el error informado cuando abrimos un archivo malformado:

Antes de continuar hablemos sobre el widget que usamos para abrir el archivo. Un "diálogo" o "ventana diálogo" (en inglés *dialog window*) es una ventana que aparece arriba de la que la genera, usada en general para tareas cortas o comunicaciones breves con las personas que usan el programa. En el caso de nuestro ejemplo usamos `QFileDialog`, un diálogo con la función específica de permitir elegir un elemento del sistema de archivos. Particularmente, usamos su método de clase `getOpenFileName` que permite elegir un archivo que ya existe, pero el diálogo tiene otros métodos que permiten también seleccionar muchos archivos, elegir un directorio, elegir un path que quizás no exista todavía (útil para grabar un archivo nuevo), etc. Qt provee varios diálogos pre-armados, pero siempre se puede heredar QDialog para armar uno específico para lo que se necesite.



Como regla general, los diálogos son modales. Una ventana es modal cuando bloquea la utilización de otras ventanas de la misma aplicación hasta que esa ventana se cierre, lo cual es muy útil para los diálogos, justamente, como en el caso de nuestro ejemplo donde no tiene sentido continuar a menos que se elija un archivo o se cancele el diálogo porque no se quiere elegir uno.

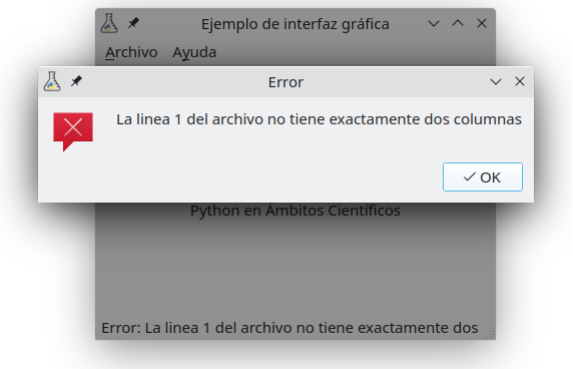
Por otro lado, es normal tener ventanas no modales cuando la aplicación es de ventanas múltiples, donde se tienen simultáneamente varias ventanas abiertas de la misma aplicación y se puede usar cualquiera indistintamente.

En la próxima iteración de nuestro código, a continuación (en `05.messages.py`), nos ocuparemos de los mensajes.

Para simplificar el código de toda la aplicación creamos dos métodos, uno que muestre un mensaje en la barra de estado (y reemplazamos todas las llamadas a `self._status_bar.showMessage` en las que había que poner siempre la misma duración), y otro para informar errores de forma genérica (al que llamamos en todos los casos de error al abrir y procesar el archivo fuente de datos).

```
61     def set_status(self, message):
62         """Muestra un mensaje efímero en la barra de estado."""
63         self._status_bar.showMessage(message, 3000)
64 
65     def show_error(self, message):
66         """Muestra un mensaje de error via un diálogo y en la barra de estado."""
67         self.set_status(f"Error: {message}")
68         QMessageBox.critical(self, "Error", message)
```

Este último método no sólo muestra el mensaje en la barra de estado (que era lo que hacían las distintas llamadas originales) sino que también levanta un diálogo informativo (con un icono correspondiente al estado `critical`, con el título "Error" y el mensaje necesario).



Vemos en la imagen que tenemos una ventana modal (notar que la ventana "padre" está griseada porque no la podemos usar) con el mensaje de error, y ese mensaje aparece también en la barra de estado.

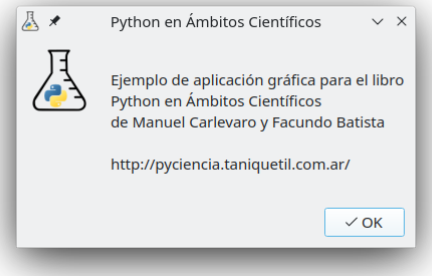
También nos ocupamos del método `on_about` que ya teníamos vinculado a la acción del menú.

```
111 
112     def on_about(self):
113         """Muestra el diálogo de Acerca de."""
114         title = "Python en Ámbitos Científicos"
115         text = textwrap.dedent("""
116             Ejemplo de aplicación gráfica para el libro
```

```
            Python en Ámbitos Científicos
            de Facundo Batista y Manuel Carlevaro

            http://pyciencia.taniquetil.com.ar/
        """)
        QMessageBox.about(self, title, text)
```

Aquí volvemos a usar `QMessageBox` pero ahora para mostrar un dialogo con la estructura específica del *about*: un título, un mensaje largo para el cuerpo, y un icono que no hace falta especificárselo si es el mismo que ya le configuramos a la aplicación.



Es hora de ocuparnos del panel principal de la aplicación, en `06.leftpanel.py`.

Como es en ese panel donde tendremos toda la complejidad de mostrar los datos, es buena idea separar esa funcionalidad toda en otra clase, `MainPanel`, que explicaremos luego. En el código que teníamos hasta ahora lo único que haremos es eliminar esos textos temporales que teníamos (el inicial al abrir la aplicación y el que indicaba la cantidad de registros al abrir un archivo) y justamente integraremos el panel principal a la aplicación:

```
        # armamos el área principal dentro de la ventana
        if self.main_panel is not None:
            self.main_panel.close()
        self.main_panel = MainPanel(self, filepath, data)
        self.setCentralWidget(self.main_panel)
```

Este código es el que se ejecutará luego de abrir exitosamente un archivo: instanciamos `MainPanel` (pasando una referencia a la ventana principal, el path del archivo y los datos cargados), y ponemos esta instancia como widget central de la aplicación.

Notar como antes de instanciar un nuevo `MainPanel` validamos no tener uno de antes, y en tal caso lo cerramos. Esto es para soportar abrir distintos archivos todas las veces que sea necesario: cada vez que se abra un nuevo archivo se descartará el panel anterior y se colocará uno nuevo.

En `MainPanel`, decíamos, tendremos toda la funcionalidad de la aplicación: mostraremos los datos, dejaremos elegir con qué tipo de función se ajustarán esos datos, y mostraremos el gráfico resultante. Por ahora, sin embargo, ocupémonos de todo eso menos el gráfico.

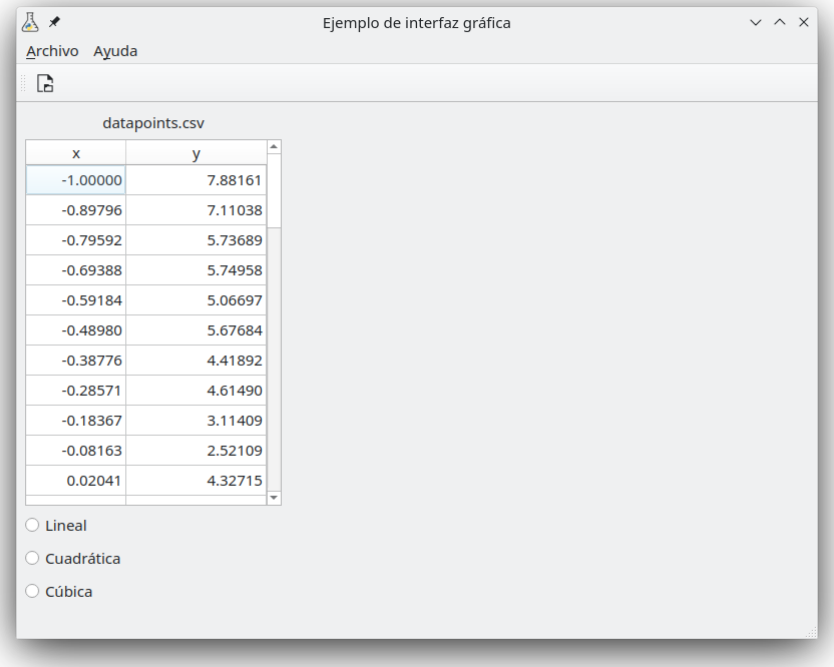
```
class MainPanel(QWidget):
    """Panel principal."""
```

```
63
64      def __init__(self, main_window, filepath, data):
65          super().__init__()
66          self.main_window = main_window
67          self.data = data
68
69          # layout principal con dos layouts verticales a izquierda y derecha
70          main_layout = QHBoxLayout()
71          self.left_layout = QVBoxLayout()
72          main_layout.addLayout(self.left_layout)
73          self.right_layout = QVBoxLayout()
74          self.right_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
75          main_layout.addLayout(self.right_layout, stretch=1)
76          self.setLayout(main_layout)
77
78          # la etiqueta con el nombre del archivo que estamos usando
79          label = QLabel(filepath.name)
80          label.setAlignment(Qt.AlignmentFlag.AlignCenter)
81          self.left_layout.addWidget(label)
82
83          # armamos una tabla, le asociamos un modelo, y configuramos su estética
84          table = QTableView()
85          table.setModel(TableModel(data))
86          table.verticalHeader().setVisible(False)
87          horizontal_header = table.horizontalHeader()
88          horizontal_header.setStretchLastSection(True)
89          horizontal_header.setSectionResizeMode(1, QHeaderView.ResizeMode.Interactive)
90          self.left_layout.addWidget(table)
91
92          # radiobuttons para la selección de qué función utilizar
93          options = QVBoxLayout()
94          self.left_layout.addLayout(options)
95          for degree, name in enumerate(["Lineal", "Cuadrática", "Cúbica"], 1):
96              radio = QRadioButton(name, self)
97              options.addWidget(radio)
```

Lo primero que hacemos al inicializar `MainPanel` es inicializar a su vez `QWidget` del cual heredamos y guardarnos en la instancia a esa ventana y los datos, que los vamos a usar más adelante.

Luego armamos el layout principal, horizontal, el cual tendrá dos layouts verticales en lo que llamaremos "izquierda" y "derecha", y lo ponemos como layout del widget. Notar que al layout de la derecha (que tendrá el gráfico o un símbolo de espera si el cálculo tarda mucho), le indicamos que tome todo el espacio disponible con `stretch=1` al agregarlo al layout principal y le configuramos su alineación para que el contenido salga centrado. Esto no lo hacemos con el layout de la izquierda porque queremos que tenga el tamaño ajustado a su contenido.

Del layout de la izquierda, justamente, nos ocupamos ahora: le agregamos el nombre del archivo que abrimos, una tabla con los datos, y una serie de *radiobuttons* para permitir elegir la función de ajuste. Poner el nombre del archivo es sencillo (líneas 78 a 81); ya usamos `QLabel` anteriormente, el único detalle es que le seteamos un flag para que el texto quede centrado. Poner los radiobuttons es apenas más complicado (líneas 92 a 97): agregamos los tres que necesitamos directamente en un layout vertical, sin necesitar hacer más nada porque como tenemos un sólo grupo de radiobuttons podemos usar el default (recordemos que los radiobuttons trabajan en grupo porque al habilitar uno se deshabilita el resto); notemos que por ahora no conectamos estos botones a ninguna acción.



Para la tabla con datos necesitamos un poco más de explicación. Vemos en el código que hay un `QTableView` y un `TableModel`. Esta estructura con distintas partes existe porque las tablas ofrecen toda una funcionalidad que en realidad no estamos usando en nuestro ejemplo (piensen que "una planilla de cálculos" también termina siendo una tabla de datos compleja con una gran cantidad de detalles e interacciones).

Así y todo no es tan complicado, pero el concepto fundamental que tenemos que entender es que nos tenemos que encargar de esas dos partes: la "vista" de los datos (que se ocupa de cómo mostrar la información y cómo interactuar con la misma) y el "modelo" de los datos, que se encarga de recuperar el valor de cada celda (que puede ser de una tabla fija como tenemos nosotros o de una base de datos), guardar también cambios en esos valores (si la tabla fuese editable), y en general proveer toda la información con respecto a los datos (como cuantas filas o columnas hay).

En el panel principal entonces nos encargamos directamente de la vista de la tabla (líneas 83 a 90). Creamos una `QTableView`, le ponemos un modelo (una clase propia que veremos a continuación), y antes de agregarla al layout le configuramos su *look & feel*: indicamos que no se vea la cabecera vertical (lo que sería el número de cada fila), y le indicamos que a la cabecera horizontal se le puede cambiar los tamaños (permitiendo ajustar el ancho de cada columna), y que la última columna se "estire" hasta ocupar el ancho de toda la tabla. Noten como en ese código no hay nada con respecto a los datos en sí, sólo se los pasamos al modelo al instanciar `TableModel`.

```
25 class TableModel(QAbstractTableModel):
26     """Modelo para la tabla de datos fuente."""
27
28     def __init__(self, data):
29         super(TableModel, self).__init__()
```

```
30          self._src_data = data
31
32      def headerData(self, section_index, orientation, role):
33          """Devuelve información para armar los headers."""
34          if orientation == Qt.Orientation.Horizontal and role == Qt.ItemDataRole.DisplayRole:
35              return ["x", "y"][section_index]
36
37      def data(self, index, role):
38          """Devuelve información necesaria para mostrar los datos.
39
40          - DisplayRole: el dato a mostrar
41          - TextAlignmentRole: la alineación
42          """
43          if role == Qt.ItemDataRole.DisplayRole:
44              # devolvemos el dato fuente según fila y columna, formateado lindo
45              datum = self._src_data[index.row()][index.column()]
46              return format(datum, ".5f")
47
48          if role == Qt.ItemDataRole.TextAlignmentRole:
49              # al centro en vertical, a la derecha en horizontal
50              return Qt.AlignmentFlag.AlignVCenter + Qt.AlignmentFlag.AlignRight
51
52      def rowCount(self, index):
53          """Devuelve el total de filas."""
54          return len(self._src_data)
55
56      def columnCount(self, index):
57          """Devuelve el total de columnas."""
58          return len(self._src_data[0])
```

Nuestro modelo es una clase que hereda de `QAbstractTableModel` y su funcionamiento se basa en proveer distintos métodos estándares que la vista va a consultar para obtener distintas informaciones. En nuestra clase definimos algunos métodos, los necesarios para proveer información sobre los datos de nuestra tabla simple y estática. Y el `__init__` para inicializar la clase, donde guardamos la tabla en la instancia luego de inicializar la clase ancestra.

Cada uno de estos métodos están bien definidos en la documentación: para qué sirve cada uno, qué información reciben al ser ejecutados y qué tienen que devolver. No vamos a explicar todo lo que podría proveer `QAbstractTableModel` porque sería demasiado largo, pero entremos en detalle al menos con los que usamos en nuestro modelo para ayudarles a entender el código de nuestro ejemplo y facilitarles la entrada a la documentación más compleja cuando la tengan que usar.

- `headerData`: este método devuelve información relacionada con las cabeceras; en nuestro caso sólo nos interesa proveer los "títulos" de cada columna, por eso sólo contestamos información para el caso de la cabecera horizontal (`orientation == Qt.Orientation.Horizontal`) y con respecto a cómo se muestran (`role == Qt.ItemDataRole.DisplayRole`), devolviendo `X` para la primer columna e `Y` para la segunda.

- `data`: este método es el que usa la vista para obtener los datos de cada celda; implementamos dos roles, `DisplayRole` donde devolvemos cada dato (formateado) según fila y columna, y `TextAlignmentRole` para la alineación, mucho más sencillo porque todas las celdas están alineadas igual (al centro en lo vertical y a la derecha en lo horizontal).

- `rowCount` y `columnCount`: devuelven la cantidad de filas y de columnas, dato que se obtiene fácilmente de la tabla fuente más allá de por qué indice hayan sido consultados (ya que la tabla es rectangular).

El próximo paso es usar una función para ajustar los datos a una función polinómica, generar el gráfico y mostrarlo. Para ello, ya en el código `07.fitgraph.py`, tenemos una función `fit` que recibe los datos y el grado de la función a utilizar.

```
# funciones para ajustar los datos; lineal, cuadrática y cúbica
functions_to_fit = {
    1: lambda x, a, b: a * x + b,
    2: lambda x, a, b, c: a * x**2 + b * x + c,
    3: lambda x, a, b, c, d: a * x**3 + b * x**2 + c * x + d,
}


def fit(data, degree):
    """Ajusta los datos con una función según el grado especificado."""
    try:
        fit_function = functions_to_fit[degree]
    except KeyError:
        raise ValueError(f"No hay una función para el grado requerido: {degree!r}")

    # los datos fuentes es una tabla de dos columnas, x e y; necesitamos esas dos columnas
    # por separado como entrada a la función de ajuste
    x_data, y_data = (np.array(x) for x in zip(*data))

    params, pcovs = curve_fit(fit_function, x_data, y_data)

    param_texts = []
    for idx, (pvalue, pname) in enumerate(zip(params, string.ascii_lowercase)):
        pcov = pcovs[idx, idx]
        param_texts.append(f"{pname} = {pvalue:.3f} ± {pcov:.3f}")

    fig = plt.figure()
    ax = fig.subplots()
    ax.plot(x_data, y_data, '.')
    ax.plot(x_data, fit_function(x_data, *params))
    ax.set_xlabel("x")
    ax.set_ylabel("y")

    return param_texts, fig
```

A las funciones polinómicas las tenemos definidas en el diccionario `functions_to_fit` de acuerdo a lo que vamos a necesitar para `curve_fit` de la biblioteca SciPy. La función `fit` toma entonces la función polinómica según el grado recibido, separa los datos en dos columnas y obtiene (usando SciPy) los parámetros de la función polinómica y las covarianzas, y con esta información genera dos productos: los textos indicando el valor y error de cada parámetro, y una figura de `matplotlib` con todo dibujado (los puntos originales y la función polinómica calculada).

Para utilizar esta función conectamos a los radiobuttons que creamos anteriormente a un método nuestro.

```
168         for degree, name in enumerate(["Lineal", "Cuadrática", "Cúbica"], 1):
169             radio = QRadioButton(name, self)
170             radio.clicked.connect(partial(self.calculate, degree))
171             options.addWidget(radio)
```

Vemos en la línea 170 que conectamos la señal `clicked` de cada radiobutton a una función creada ahí mismo con `partial`, que al ejecutarse llamará a `self.calculate` pasando automáticamente `degree` como parámetro

```
176     def calculate(self, function_degree):
177         """Pone a realizar los cálculos."""
178         self.main_window.set_status("Calculando")
179 
180         try:
181             result = fit(self.data, function_degree)
182         except Exception as exc:
183             self.on_error(exc)
184         else:
185             self.show_result(result)
186 
187     def on_error(self, exc):
188         """Muestra el error que sucedió al hacer el cálculo."""
189         # un resumen del problema en la interfaz
190         self.main_window.show_error(f"Error al calcular! {exc!r}")
191         # y el traceback completo por la terminal
192         print("ERROR al calcular:")
193         traceback.print_exception(exc)
194 
195     def show_result(self, result):
196         """Arma el panel con el resultado."""
197         self.main_window.set_status("Mostrando el resultado")
198         fit_parameters, matplotlib_figure = result
199         result_panel = ResultPanel(self, fit_parameters, matplotlib_figure)
200 
201         if self.result_panel is None:
202             # primera vez, no hay resultados de antes
203             self.right_layout.addWidget(result_panel)
204         else:
205             # reemplazamos los resultados previos
206             self.right_layout.replaceWidget(self.result_panel, result_panel)
207             self.result_panel.close()
208         self.result_panel = result_panel
```

Este método `calculate` es el que ejecutará la función `fit`. En caso de error llamará a otro método que mostrará el error utilizando nuestro método de la ventana principal, más todo el *traceback* por la terminal.

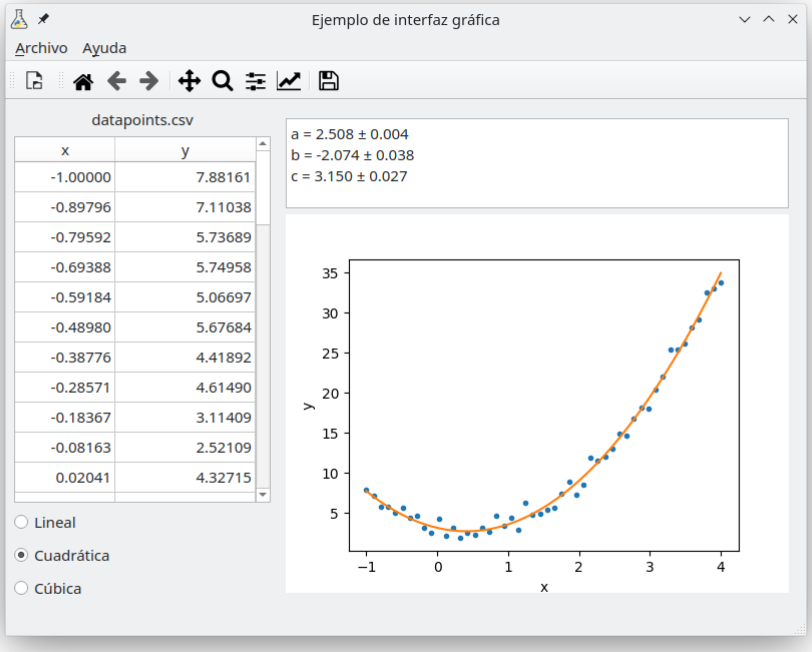
En caso de que haya salido todo bien pasará ese resultado a otro método, `show_result`, cuya responsabilidad es crear un panel de resultados (que veremos a continuación) y agregarlo al layout de la derecha, o en caso de tener otro panel de antes, cerrar el viejo y reemplazarlo.

Veamos entonces `ResultPanel` que es el que se encarga de mostrar la info resultado. Recibe la instancia del panel principal, los parámetros de la función, y la figura de matplotlib:

```
109 class ResultPanel(QWidget):
110     """Panel para mostrar el resultado: parámetros y el gráfico en sí."""
111     def __init__(self, parent, fit_parameters, matplotlib_figure):
112         super().__init__()
113         self.parent = parent
114         layout = QVBoxLayout()
115         self.setLayout(layout)
116 
117         # los parámetros de la ecuación
118         params_textedit = QPlainTextEdit()
119         params_textedit.setPlainText("\n".join(fit_parameters))
120         layout.addWidget(params_textedit)
121 
122         # el gráfico, y su toolbar relacionado que lo agregamos a la ventana principal
123         figure_canvas = FigureCanvasQTAgg(matplotlib_figure)
124         layout.addWidget(figure_canvas)
125         self.toolbar = NavigationToolbar2QT(figure_canvas, self)
126         self.parent.main_window.addToolBar(self.toolbar)
127 
128     def close(self):
129         """Remueve la toolbar antes de cerrar el widget."""
130         self.parent.main_window.removeToolBar(self.toolbar)
131         super().close()
```

Como ya vimos en otros casos, la instancia primero inicializa el widget del que hereda, guarda algún parámetro recibido y crea un layout para agregar contenido. En este caso tenemos dos elementos: un *text edit* al que le ponemos varias líneas con los parámetros recibidos, y el gráfico.

Para el gráfico usaremos funcionalidad que `matplotlib` misma provee para integrarse con Qt: un *canvas* para agregar la figura y una barra de herramientas (conectada a esa canvas) para que se pueda interactuar con el gráfico. A la barra de tareas la podemos ubicar en cualquier lado, pero a nivel usabilidad lo mejor es que sea una barra más de la ventana principal, entonces en la línea 126 la agregamos allí. Pero al hacer esto tenemos que tener la precaución de que cuando se cambie el gráfico tenemos que sacar la barra del gráfico anterior, y eso lo hacemos en el método `close` de nuestra clase (línea 130).



La próxima iteración de código, en `08.notblocking.py`, está pensada para ejemplificar cómo podemos soportar aquellas situaciones donde algo tarda demasiado. Como Qt es un sistema asincrónico en realidad no podemos bloquearlo con ninguna operación que tarde más que algunas decenas de milisegundos, porque mientras el sistema está bloqueado no va a poder procesar ningún evento, y esto a nivel de experiencia de usabilidad se traduce en que la ventana se "congela", no responde al teclado o el mouse, e incluso puede aparecer a medio dibujar en la pantalla (recordemos que Qt no sólo procesa eventos nuestros sino también del sistema operativo, y si no puede procesar nada porque está bloqueado no se va a comportar correctamente).

En nuestro ejemplo el *cálculo científico* que hacemos es bastante rápido, ya que es un simple ajuste de datos, pero simulemos un potencial cálculo más costoso agregando una demora artificial de cinco segundos en la función `fit`:

```
72      # simulamos que la tarea tarda mucho tiempo, y es bloqueante, así mostramos cómo
73      # lidiar con esta situación a nivel GUI
74      time.sleep(5)
```

Para soportar casos de funciones que tardan mucho en ejecutarse y no bloquear la interfaz gráfica debemos ejecutar esa función en otro hilo o *thread*, para lo cual creamos una clase auxiliar, `ThreadedTask`:

```
115 class TaskSignals(QObject):
116     """Señales para conectar de nuestra ThreadedTask."""
117     success = pyqtSignal(object)
118     error = pyqtSignal(Exception)
```

```
119
120
121 class ThreadedTask(threading.Thread):
122     """Ejecuta una función en un hilo.
123
124     Recibe la función y los argumentos. En 'signals' provee 'success' para cuando la función
125     se completa satisfactoriamente (envía el resultado) y 'error' para cuando no (envía
126     la excepción).
127     """
128     def __init__(self, func, *args, **kwargs):
129         self.func = func
130         self.args = args
131         self.kwargs = kwargs
132         self.signals = TaskSignals()
133         super().__init__()
134
135     def run(self):
136         try:
137             result = self.func(*self.args, **self.kwargs)
138         except Exception as exc:
139             self.signals.error.emit(exc)
140         else:
141             self.signals.success.emit(result)
```

Esta clase es bien simple, pero el truco es que hereda de `threading.Thread`, que es una de las maneras de armar hilos en Python (más sobre hilos en el capítulo de procesamiento paralelo **??**). En el método de inicialización recibe la función a ejecutar y cualquier cantidad de argumentos posicionales y nombrados, y guarda todo en la instancia. Además inicializa un objeto con dos señales (definido en las líneas 115 a 118), para los casos de terminar satisfactoriamente o en error, y por supuesto llama al inicializador de la clase ancestra (requisito fundamental para que el hilo funcione).

Luego en el método `run` (que es el punto de entrada cuando se ejecuta el hilo) simplemente se ejecuta la función y guarda el resultado, con la precaución de soportar cualquier error. Si todo estuvo bien se emite la señal `success` pasando el resultado, y si algo salió mal se emite la señal `error` pasando la excepción ocurrida, y luego el hilo termina.

Volvamos al código de `MainPanel` para ver cómo usamos esa clase auxiliar. Aquí tenemos que reescribir el método `calculate` que es donde ejecutábamos la función:

```
211     def calculate(self, function_degree):
212         """Pone a realizar los cálculos en un hilo."""
213         self.main_window.set_status("Calculando")
214
215         # disparamos el nuevo cálculo y conectamos la función que muestra el resultado
216         tt = ThreadedTask(fit, self.data, function_degree)
217         tt.signals.success.connect(self.show_result)
218         tt.signals.error.connect(self.on_error)
219         tt.start()
```

Ahora, además de poner el estado en "Calculando", inicializamos `ThreadedTask` pasando la función `fit`, los datos y el grado de la función de ajuste, y conectamos sus señales: la de éxito para mostrar el resultado, y la de error para mostrar la excepción ocurrida (¡notar que esas

dos funciones no cambiaron! ya recibían el resultado o la excepción desde antes). Finalmente llamamos al método `start` de la clase auxiliar para que el hilo arranque.

Todo este trabajo que hicimos no tiene un impacto directo en la interfaz del usuario (no hay un botón nuevo, o se muestra distinta alguna información), pero es fundamental a la hora de tener una aplicación que no se congele y que no de la sensación al usuario de que "algo salió mal".

Por otro lado, aunque ahora la aplicación se mantiene responsiva, en realidad no nos está dando ninguna indicación que nos ayude a darnos cuenta que está haciendo un cálculo que lleva tiempo y que tenemos que esperar. Implementamos esto en la próxima iteración del código, `09.complete.py`, que es la versión final de nuestra aplicación.

Históricamente la forma de las interfaces gráficas de mostrar que "se está realizando alguna tarea y hay que esperar" es mediante un `throbber` (un término sin traducción al castellano para esta acepción), que es una imagen animada, en general circular, que va "girando" mientras tenemos que esperar.

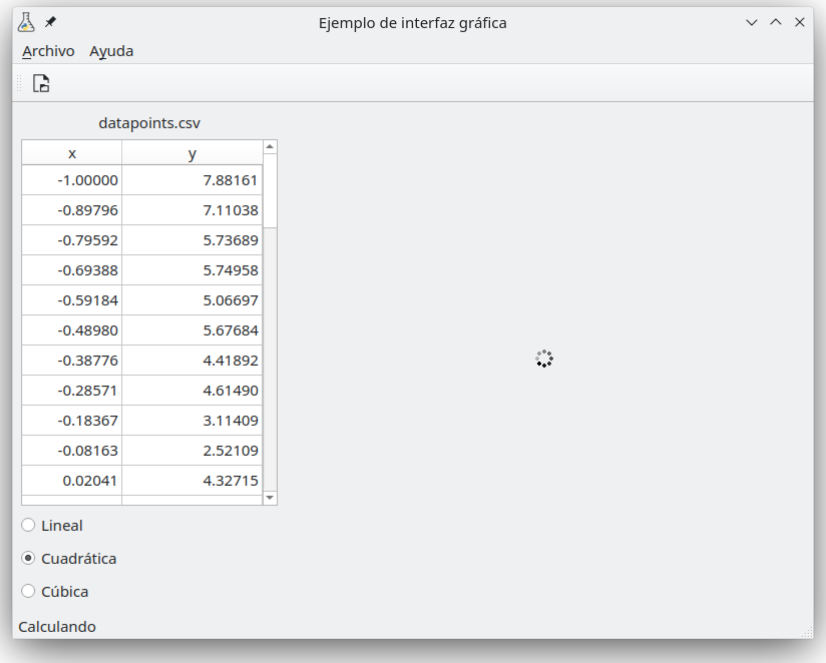
Agreguemos entonces un throbber a nuestro programa. El mismo tiene que verse cuando arranca el cálculo, y sacarse cuando tenemos el gráfico resultado. Si elegimos otro grado para la función de ajuste tenemos que sacar el gráfico y poner el throbber para esperar el nuevo cálculo. En la práctica, lo que sucede es que en el layout derecho, luego de iniciar el primer cálculo, vamos a ir alternando entre mostrar un throbber y el panel de resultados.

En el método `MainPanel.calculate` tenemos entonces que poner el throbber antes de crear e iniciar el hilo con el cálculo:

```
216         self.throbber = QLabel()
217         movie = QMovie("throbber.gif")
218         self.throbber.setMovie(movie)
219         movie.start()
220
221         if self.result_panel is None:
222             # primera vez, no hay resultados de antes, sólo agregamos el throbber
223             self.right_layout.addWidget(self.throbber, stretch=1)
224         else:
225             # reemplazamos los resultados previos con el throbber
226             self.right_layout.replaceWidget(self.result_panel, self.throbber)
227             self.result_panel.close()
```

Crear el throbber es sencillo (líneas 216 a 219) porque nos basamos en tener un GIF animado en un archivo `throbber.gif` que cargamos con `QMovie` (un widget para mostrar animaciones simples sin sonido), que para insertarlo en la interfaz ponemos dentro de un `QLabel` (como haríamos con un texto o una imagen).

Inmediatamente después en las líneas 221 a 227 agregamos el throbber al layout derecho, directamente si no había un panel de antes, o reemplazando un panel anterior (y cerrándolo).



Luego de agregarlo, tenemos que considerar cuando sacarlo. Esto sucede en los métodos `on_error` y `show_result`, que se ejecutan cuando terminó el cálculo por error o satisfactoriamente, cerrando y eliminando el throbber en el primer caso…

```
243         self.right_layout.removeWidget(self.throbber)
244         self.throbber.close()
```

…, o cerrándolo y reemplazándolo por el panel de resultados en el segundo.

```
250         self.result_panel = ResultPanel(self, fit_parameters, matplotlib_figure)
251         self.right_layout.replaceWidget(self.throbber, self.result_panel)
252         self.throbber.close()
```

Llegamos entonces al final de nuestro recorrido, con una interfaz completa y funcional.

Vuestras necesidades de interfaz van a ser distintas, obviamente, pero siempre serán en mayor o menor medida alguna variación de lo que ya vimos en este capítulo, usando también otros widgets más o menos complejos.

Lo importante es que ya saben cómo arrancar.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [2].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.