

Real Python

Python and PyQt: Building a GUI Desktop Calculator

by Leodanis Pozo Ramos Oct 23, 2019     

[Mark as Completed](#)[Tweet](#)[Share](#)[Email](#)

Table of Contents

- [Understanding PyQt](#)
- [Installing PyQt](#)
 - [System-Wide Installation With pip](#)
 - [Virtual Environment Installation With pip](#)
 - [Platform-Specific Installation](#)
 - [Anaconda Installation](#)
- [Creating Your First PyQt Application](#)
- [Considering Code Styles](#)
- [Learning the Basics of PyQt](#)
 - [Widgets](#)
 - [Layout Managers](#)
 - [Dialogs](#)
 - [Main Windows](#)
 - [Applications](#)
 - [Event Loops](#)
 - [Signals and Slots](#)
- [Creating a Calculator With Python and PyQt](#)
 - [Creating the Skeleton](#)
 - [Completing the View](#)
 - [Creating a Basic Controller](#)



- Implementing the Model
- Completing the Controller
- Running the Calculator
- Additional Tools
- Conclusion
- Further Reading

 Remove ads

Even as web and mobile applications appear to overtake the software development market, there's still a demand for traditional **Graphical User Interface (GUI)** desktop applications. For developers who are interested in building these kinds of applications in Python, there are a wide variety of libraries to choose from, including Tkinter, wxPython, PyQt, PySide2, and others. In this tutorial, you'll develop GUI desktop applications with Python and **PyQt**.

You'll learn how to:

- Create Graphical User Interfaces with Python and PyQt
- Give life to your applications by connecting user events to concrete actions
- Create fully-functional GUI applications to solve real-world problems

For this tutorial, you'll create a calculator application with Python and PyQt. This will help you grasp the fundamentals and get you up and running with the library. You can download the source code for the project and all examples in this tutorial by clicking on the link below:

Download Code: Click here to download the code you'll use to build a calculator in Python with PyQt in this tutorial.

Understanding PyQt

PyQt is a Python binding for [Qt](#), which is a set of [C++](#) libraries and development tools that include platform-independent abstractions for Graphical User Interfaces (GUI), as well as networking, threads, [regular expressions](#), SQL databases, SVG, OpenGL, XML, and many other powerful features. Developed by [RiverBank Computing Ltd](#), PyQt is available in two editions:

1. [PyQt4](#): an edition that's built against Qt 4.x and 5.x
2. [PyQt5](#): an edition that's only built against Qt 5.x

Even though PyQt4 can be built against Qt 5.x, only a small subset that is also compatible with Qt 4.x will be supported. This means that if you decide to use PyQt4, then you'll probably miss out on some of the new features and improvements in PyQt5. See the [PyQt4 documentation](#) for more information on this topic.

You'll be covering PyQt5 in this tutorial, as it seems to be the future of the library.

From now on, be sure to consider any mention of PyQt as a reference to PyQt5.

Note: If you want to dive deeper into the differences between the two versions of the library, then you can take a look at the related page on the [PyQt5 documentation](#).

PyQt5 is based on [Qt v5](#) and includes classes that cover Graphical User Interfaces as well as XML handling, network communication, regular expressions, threads, SQL databases, multimedia, web browsing, and other technologies available in Qt.

PyQt5 implements over one thousand of these Qt classes in a set of [Python modules](#), all of which are contained within a top-level Python package called PyQt5.

PyQt5 is compatible with Windows, Unix, Linux, macOS, iOS, and Android. This can be an attractive feature if you're looking for a library or framework to develop multi-platform applications with a native look and feel on each platform.

PyQt5 is available under two licenses:

1. [The Riverbank Commercial License](#)
2. [The General Public License \(GPL\), version 3](#)

Your PyQt5 license must be compatible with your Qt license. If you use the GPL version, then your code must also use a GPL-compatible license. If you want to use PyQt5 to create commercial applications, then you'll need a commercial license for your installation.

Note: The [Qt Company](#) has developed and currently maintains its own Python binding for the Qt library. The Python library is called [Qt for Python](#) and is considered to be the official Qt for Python. In this case, the Python package is called PySide2.

As PyQt5 and PySide2 are both built on top of Qt, their APIs are quite similar, even almost identical. That's why porting PyQt5 code to PySide2 can be as simple as updating some imports. If you learn one of them, then you'll be able to work with the other with minimal effort. If you want to dive deeper into the differences between these two libraries, then you can check out the following resources:

- [PyQt vs Qt for Python \(PySide2\)](#)
- [What's the difference between PyQt5 and PySide2?](#)

 Remove ads

Installing PyQt

You have several options to choose from when you install PyQt on your system or development environment. The first option is to build from source. This can be a bit

complicated, so you might want to avoid it if possible. If you really need to build from source, then you can take a look at what the library's [documentation](#) recommends in those cases.

Note: Most of the installation options you'll cover here require that you have a working Python installation. If you need to dive deeper into how to install Python, then check out [Python 3 Installation & Setup Guide](#).

Another option would be to use binary [wheels](#). Wheels are a very popular way to manage the installation of [Python packages](#). However, you need to consider that wheels for PyQt5 are only available for Python 3.5 and later. There are wheels for:

- Linux (64-bit)
- macOS
- Windows (32-bit and 64-bit)

All of these wheels include copies of the corresponding Qt libraries, so you won't need to install them separately.

Your third option is to use package managers on Linux distributions and macOS. For Windows, you can use a binary .exe file. Your fourth and final option is to use the Anaconda distribution to install PyQt on your system. The next few sections will walk you through some of the options you have to properly install PyQt5 from different sources and on different platforms.

System-Wide Installation With pip

If you're using Python 3.5 or later, then you can install PyQt5 from [PyPI](#) by running the following command:

Shell

```
$ pip3 install pyqt5
```

With this command, you'll install PyQt5 in your base system. You can start using the library immediately after the installation finishes. Depending on your operating system, you may need root or administrator privileges for this installation to work.

Virtual Environment Installation With pip

You may decide not to install PyQt directly on your base system to avoid messing up your configuration. In this case, you can use a Python [virtual environment](#). If you're on a UNIX system, then you can use the following commands to create one and install PyQt5:

Shell

```
$ python3 -m venv pyqtvenv  
$ source pyqtvenv/bin/activate  
(pyqtvenv) $ pip install pyqt5
```

```
Collecting pyqt5
...
Successfully installed PyQt5-sip-4.19.17 pyqt5-5.12.2
```

Here, you create a virtual environment with `venv`. Once you activate it, you install `pyqt5` in that environment with `pip install pyqt5`. This installation alternative is the most recommended option if you want to keep your base system clean.

Platform-Specific Installation

In the Linux ecosystem, several distributions include binary packages for PyQt in their repositories. If this is true for your distribution, then you can install the library using the distribution's package manager. On Ubuntu 18.04 for example, you can use the following command:

Shell

```
$ sudo apt install python3-pyqt5
```

With this command, you'll install PyQt5 and all its dependencies in your base system, so you can use the library in any of your GUI projects. Note that root privileges are needed, which you invoke here with `sudo`.

If you're a Mac user, then you can install PyQt5 using the [Homebrew](#) package manager. To do that, open a terminal and type in the following command:

Shell

```
$ brew install pyqt5
```

If all goes well, then you'll have PyQt5 installed on your base system, ready for you to use.

Note: If you use a package manager on Linux or macOS, then there's a chance you won't get the latest version of PyQt5. A pip installation would be better if you want to ensure that you have the latest release.

If you prefer to use Windows, but you decide not to use the binary wheels, then your path through PyQt installation can be painful. That's because the PyQt5 [download page](#) appears to no longer provide Windows binaries (.exe files) for later versions. Still, you can find some Windows binary packages for older versions of the library at the [project page](#). The latest binary file was built for [PyQt v5.6](#).

If you really need to install PyQt this way, then you'll need to:

1. Determine what version of Python you're running and whether you have 32-bit or 64-bit Python
2. Download the right version for your Python installation
3. Install PyQt by running the .exe file and following the on-screen instructions

Anaconda Installation

Another alternative you can use to install PyQt is [Anaconda](#), a Python distribution for data science. Anaconda is a free and multi-platform package and environment manager that includes a collection of over 1,500 open source [packages](#).

Anaconda provides a user-friendly installation wizard that you can use to install PyQt on your system. You can [download](#) the appropriate version for your current platform and follow the on-screen instructions. If you install the latest version of Anaconda, then you'll have the following packages:

- [pyqt](#): a Python binding for the cross-platform GUI toolkit Qt (Commercial, GPL-2.0, GPL-3.0 licenses)
- [anyqt](#): a compatibility layer for PyQt4/PyQt5 (GPL-3.0 license)
- [qtpy](#): an abstraction layer PyQt5/PyQt4/PySide (MIT license)
- [pyqtgraph](#): a Python library for Scientific Graphics (MIT license)

With this set of packages, you'll have all that you need to develop GUI desktop applications with Python and PyQt.

Note: Note that the Anaconda installation will occupy a large amount of disk space. If you install Anaconda only to use the PyQt packages, then you'll have a serious amount of unused packages and libraries on your system. Keep this in mind when you consider using the Anaconda distribution.

Creating Your First PyQt Application

Now that you have a working PyQt installation, you're ready to start coding. You're going to create a "Hello, World!" application with Python and PyQt. Here are the steps you'll follow:

1. Import [QApplication](#) and all the required widgets from [PyQt5.QtWidgets](#).
2. Create an instance of [QApplication](#).
3. Create an instance of your application's GUI.
4. Show your application's GUI.
5. Run your application's event loop (or main loop).

You can download the source code for the examples you'll cover in this section at the link below:

Download Code: [Click here to download the code you'll use](#) to build a calculator in Python with PyQt in this tutorial.

You'll start with a file called `hello.py` in your current working directory:

Python

```
# Filename: hello.py

"""Simple Hello World example with PyQt5."""

import sys

# 1. Import `QApplication` and all the required widgets
from PyQt5.QtWidgets import QApplication
from PyQt5.QtWidgets import QLabel
from PyQt5.QtWidgets import QWidget
```

First, you import `sys`, which will allow you to handle the exit status of the application. Then, you import `QApplication`, `QWidget`, and `QLabel` from `QtWidgets`, which is part of the package called `PyQt5`. You'll be using these imports later on, but for now, you're done with step one.

For step two, you need to create an instance of `QApplication` as follows:

Python

```
# 2. Create an instance of QApplication
app = QApplication(sys.argv)
```

Here, you create the instance of `QApplication`. Since the `QApplication` object (`app`) does so much initialization, you should create it before you create any other object related to the GUI. The `QApplication` object also deals with common [command line arguments](#), so you also need to pass in `sys.argv` as an argument when you create `app`.

Note: `sys.argv` contains the [list](#) of command-line arguments passed into a Python script. If your application is not going to accept command line arguments, then you can use an empty [list](#) instead of `sys.argv`. That is, you can do something like `app = QApplication([])`.

Step three is to create the application's GUI. For this example, your GUI will be based on `QWidget`, which is the base class of all user interface objects in PyQt. Let's create the GUI:

Python

```
# 3. Create an instance of your application's GUI
window = QWidget()
window.setWindowTitle('PyQt5 App')
window.setGeometry(100, 100, 280, 80)
window.move(60, 15)
helloMsg = QLabel('<h1>Hello World!</h1>', parent=window)
helloMsg.move(60, 15)
```

In this code, `window` is an instance of [QWidget](#), which provides all the features you'll need to create the application's window (or form). With `setWindowTitle()`, you

need to create the application's window (or form). With `setWindowTitle()`, you can add a title to your application's window. In this case, the title to show is PyQt5 App.

You can use `.setGeometry()` to define the size of the window and where to place it on your screen. The first two parameters are the x and y coordinates at which the window will be placed on the screen. The third and fourth parameters are the width and height of the window.

Every functional GUI application needs widgets! Here, you use a `QLabel` object (`helloMsg`) to show the message Hello World! on your application's window. `QLabel` objects can accept HTML text, so you can use the HTML element '`<h1>Hello World!</h1>`' to format the text as an `h1` header. Finally, you use `.move()` to place `helloMsg` at the coordinates (60, 15) on your application's window.

Note: In PyQt5, you can use any widget (a subclass of `QWidget`) as a top-level window, or even a button or a label. The only condition is that you pass no parent to it. When you use a widget like this, PyQt5 automatically gives it a title bar and turns it into a normal window.

The parent-child relationship is used for two complementary purposes:

1. A widget that doesn't have a parent is a **main window** or a **top-level window**.
2. A widget that has a parent (which is always another widget) is contained (or shown) within its parent.

This relationship also defines **ownership**, with parents owning their children. The PyQt5 ownership model ensures that if you delete a parent (for example, a top-level window), then all of its children (widgets) are automatically deleted as well.

To avoid memory leaks, you should always make sure that any `QWidget` object has a parent, with the sole exception of top-level windows.

You're done with step three, so let's code the last two steps and get your first PyQt GUI application ready to go live:

Python

```
# 4. Show your application's GUI  
window.show()  
  
# 5. Run your application's event loop (or main loop)  
sys.exit(app.exec_())
```

Here, you call `.show()` on `window`. The call to `.show()` schedules a paint event. In other words, it adds a new event to the application's event queue. You cover the event loop in a later section.

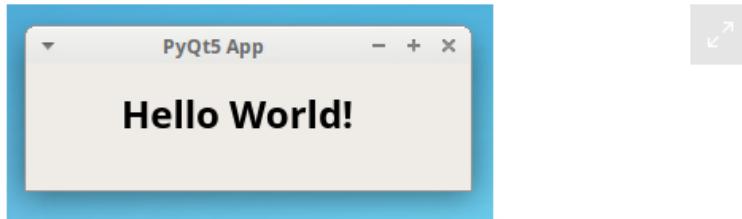
Note: A **paint event** is a request for painting the widgets that compose a GUI.

Finally, you start the application's event loop by calling `app.exec_()`. The call to `.exec_()` is wrapped in a call to `sys.exit()`, which allows you to cleanly exit Python and release memory resources when the application terminates. You can run `hello.py` with the following command:

Shell

```
$ python3 hello.py
```

When you [run this script](#), you should see a window like this:



Here, your application shows a window (based on `QWidget`) with the message Hello World! on it. To show the message, you use a `QLabel` that contains the message in HTML format.

Congrats! You've created your first PyQt GUI desktop application!

[ⓘ Remove ads](#)

Considering Code Styles

If you take a closer look at the code for your first application, then you'll notice that PyQt doesn't follow [PEP 8](#) coding style and naming conventions. PyQt is built on top of Qt, which is written in C++ and uses a camelCase naming style for functions, methods, and variables. That said, you'll need to decide what naming style you're going to use for your own PyQt GUI applications.

With regard to this issue, PEP 8 states that:

New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred. ([Source](#))

In addition, the [Zen of Python](#) says:

...practicality beats purity. ([Source](#))

If you want to write consistent code, then you might want to disregard PEP 8 naming style and stick to the PyQt naming style. This is a decision that you need to make. In this tutorial, you'll follow the PyQt naming style for consistency.

Learning the Basics of PyQt

You'll need to master the basic concepts of PyQt logic in order to efficiently use the library to develop GUI applications. Some of these concepts include:

- Widgets
- Layout managers
- Dialogs
- Main windows
- Applications
- Event loops
- Signals and slots

These elements will be the building blocks of your PyQt GUI applications. Most of them are represented as Python classes. `PyQt5.QtWidgets` is the module that provides all these classes. These elements are extremely important, so you'll cover them in the next few sections.

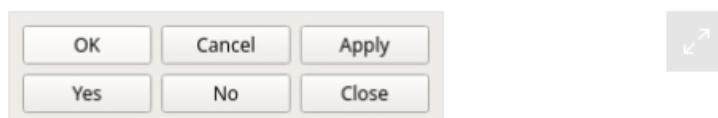
Widgets

`QWidget` is the base class for all user interface objects, or **widgets**. These are rectangular-shaped graphical components that you can place on your application's windows to build the GUI. Widgets contain a series of attributes and methods that allow you to model their appearance and behavior. They can also paint a representation of themselves on the screen.

Widgets also receive mouse clicks, keypresses, and other events from the user, the window system, and many other sources. Each time a widget catches an event, it emits a signal to announce its state change. PyQt5 has a rich and modern collection of widgets that serve several purposes. Some of the most common and useful widgets are:

- Buttons
- Labels
- Line edits
- Combo boxes
- Radio buttons

Let's take a closer look at each of these widgets. First up is the **button**. You can create a button by instantiating `QPushButton`, a class that provides a classical command button. Typical buttons are OK, Cancel, Apply, Yes, No, and Close. Here's how they look on a Linux system:



Buttons like these are perhaps the most commonly used widget in any GUI. When

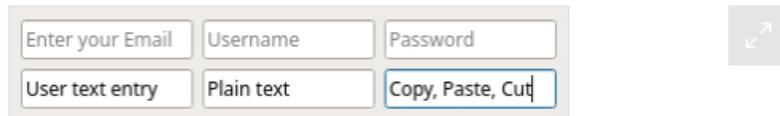
you click them, you can command the computer to perform actions. You can even perform actions in response to a user clicking a button.

Up next are **labels**, which you can create with [QLabel](#). Labels give you a way to display useful information in the form of text or images:



You can use labels like these to better explain the purpose or usage of your GUI. You can tweak their appearance in several ways, and they can even accept HTML text, as you saw earlier. Labels can also be used to specify a focus mnemonic key for another widget.

Another common widget is the **line edit**, a single-line text box that you can create with [QLineEdit](#). Line edits are useful when you need the user to enter or edit data in plain text format. Here's how they look on a Linux system:



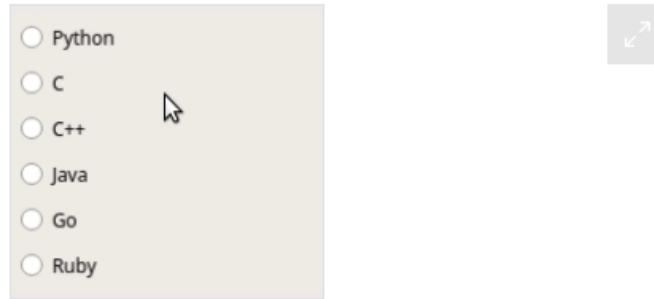
Line edits like these provide basic editing operations like copy, paste, undo, redo, drag, drop, and so on. In the above figure, you can also see that the objects on the first row show placeholder text to inform the user what kind of input is required.

Combo boxes are another useful widget that you can create with [QComboBox](#). A combo box will present your user with a list of options in a way that takes up a minimal amount of screen space. Here's an example of a dropdown list on a Linux system:



This combo box is **read-only**, which means the user can select one of several options but can't add their own. Combo boxes can also be **editable**, allowing the user to add new options. They can contain pixmaps, strings, or both.

The last widget you'll cover here is the **radio button**, which you can create with [QRadioButton](#). A `QRadioButton` object is an option button that can be switched on (checked) or off (unchecked). Radio buttons are useful when you need the user to select one of many options. In this case, all options are visible on the screen at the same time:



In this group of radio buttons, only one button can be checked at a given time. If the user selects another radio button, then the previously selected button will switch off automatically.

PyQt5 has a large collection of widgets. At the time of this writing, there are over forty available for you to use to create your application's GUI. Those you've covered so far are only a small sample, but they show you the power and flexibility of PyQt5. In the next section, you'll cover how to lay out different widgets to create modern and functional GUIs for your applications.

[ⓘ Remove ads](#)

Layout Managers

Now you know what widgets are and how to use them to build GUIs. But how can you arrange a set of widgets to create a GUI that is both coherent and functional? There are a variety of techniques that you can use to lay out the widgets on a form or window. For instance, you can use `.resize()` and `.move()` to give widgets absolute sizes and positions. However, this can have some drawbacks:

- You'll have to do a lot of manual calculations to determine the correct size and position of every single widget in your forms.
- You'll have to do some extra calculations to correctly respond to changes in form size (**resize event**).
- You'll have to redo all the calculations whenever you change the layout of your forms or add or remove widgets.

One alternative is to use `.resizeEvent()` to calculate widget size and position dynamically. However, the most effective alternative is might be to use [layout managers](#), which will both increase your productivity and improve your code's maintainability.

Layout managers are classes that allow you to size and position your widgets at the places you want them to be on the application's form. Layout managers automatically adapt to resize events and content changes. They also control the size of the widgets within them. This means that the widgets in a layout are automatically resized whenever the form is resized.

Note: If you develop international applications, then you may have seen how translated labels can be cut short. This is particularly likely when the target

language is more verbose than the original language. Layout managers can help you avoid this common pitfall. However, this feature can be a bit tricky and can sometimes fail with particularly wordy languages.

PyQt provides four basic layout manager classes:

1. [QHBoxLayout](#)
2. [QVBoxLayout](#)
3. [QGridLayout](#)
4. [QFormLayout](#)

The first layout manager class is **QHBoxLayout**, which arranges widgets horizontally from left to right:



The widgets will appear one next to the other, starting from the left.

This code example shows you how to use QHBoxLayout to arrange buttons horizontally:

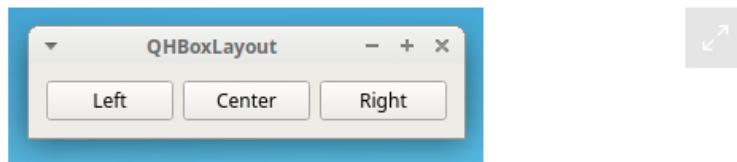
Python

```
1 # Filename: h_layout.py
2
3 """Horizontal layout example."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QHBoxLayout
9 from PyQt5.QtWidgets import QPushButton
10 from PyQt5.QtWidgets import QWidget
11
12 app = QApplication(sys.argv)
13 window = QWidget()
14 window.setWindowTitle('QHBoxLayout')
15 layout = QHBoxLayout()
16 layout.addWidget(QPushButton('Left'))
17 layout.addWidget(QPushButton('Center'))
18 layout.addWidget(QPushButton('Right'))
19 window.setLayout(layout)
20 window.show()
21 sys.exit(app.exec_())
```

The highlighted lines do the magic here:

- **Line 15** creates a QHBoxLayout object called `layout`.
- **Lines 16 to 18** add three buttons to `layout` with `.addWidget()`
- **Line 19** sets `layout` as your window's layout with `.setLayout()`.

When you run `python3 h_layout.py` from your command line, you'll get the following output:



In the above figure, you added three buttons in a horizontal arrangement. Notice that the buttons are shown from left to right in the same order as you added them in your code.

The next layout manager class is `QVBoxLayout`, which arranges widgets vertically, from top to bottom:



Each new widget will appear beneath the previous one. You can use this class to construct vertical box layout objects and organize your widget from top to bottom.

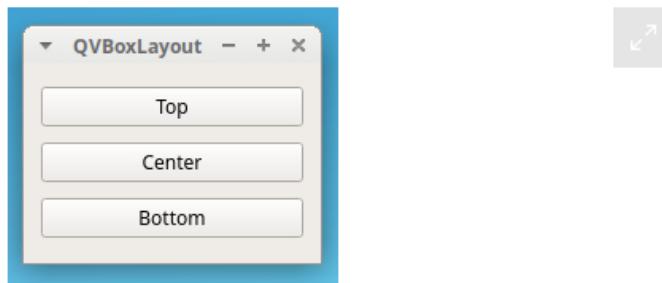
Here's how you can create and use a `QVBoxLayout` object:

```
Python
1 # Filename: v_layout.py
2
3 """Vertical layout example."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QPushButton
9 from PyQt5.QtWidgets import QVBoxLayout
10 from PyQt5.QtWidgets import QWidget
11
12 app = QApplication(sys.argv)
13 window = QWidget()
14 window.setWindowTitle('QVBoxLayout')
15 layout = QVBoxLayout()
16 layout.addWidget(QPushButton('Top'))
17 layout.addWidget(QPushButton('Center'))
18 layout.addWidget(QPushButton('Bottom'))
19 window.setLayout(layout)
20 window.show()
21 sys.exit(app.exec_())
```

In line 15, you create an instance of `QVBoxLayout`. In the next three lines, you add

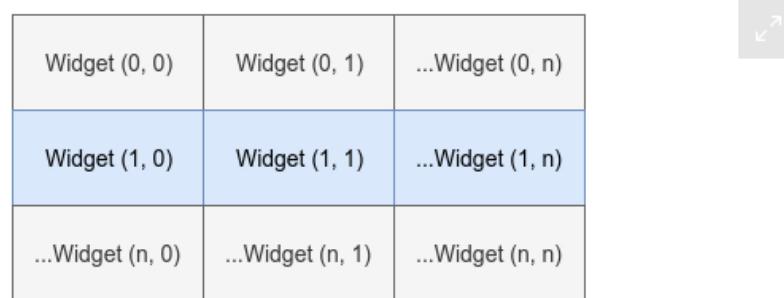
three buttons to layout. Finally, you use layout to arrange the widget in a vertical layout.

When you run this application, you'll get an output like this:



This application shows three buttons in a vertical layout, one below the other. The buttons appear in the same order as you added them in your code, from top to bottom.

The third layout manager class is **QGridLayout**, which arranges widgets into a grid of rows and columns. Every widget will have a relative position on the grid. You can define a widget's position by passing it a pair of coordinates in the form of (row, column). These coordinates should be valid `int` numbers. They define which cell of the grid you're going to place the widget on. The grid layout works as follows:



`QGridLayout` takes the space made available to it by its parent, divides it up into rows and columns, and puts each widget into its own cell.

Here's how to use `QGridLayout` in your GUI:

```
Python
1 # Filename: g_layout.py
2
3 """Grid layout example."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QGridLayout
9 from PyQt5.QtWidgets import QPushButton
10 from PyQt5.QtWidgets import QWidget
11
12 app = QApplication(sys.argv)
13 window = QWidget()
14 window.setWindowTitle('QGridLayout')
15 layout = QGridLayout()
16 layout.addWidget(QPushButton('Top'), 0, 0, 1, 1)
17 layout.addWidget(QPushButton('Center'), 1, 0, 1, 1)
18 layout.addWidget(QPushButton('Bottom'), 2, 0, 1, 1)
19 window.setLayout(layout)
20 window.show()
21 app.exec_()
```

```

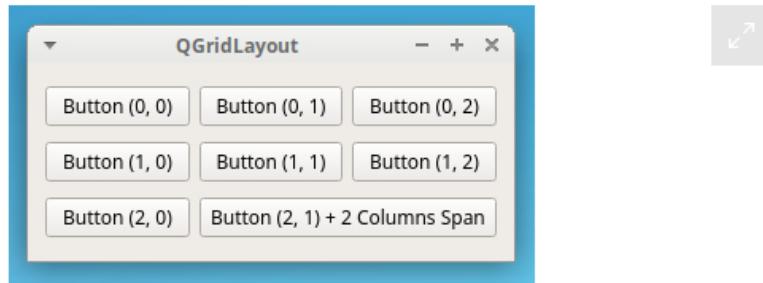
10 layout.addWidget(QPushButton('Button (0, 0)'), 0, 0)
11 layout.addWidget(QPushButton('Button (0, 1)'), 0, 1)
12 layout.addWidget(QPushButton('Button (0, 2)'), 0, 2)
13 layout.addWidget(QPushButton('Button (1, 0)'), 1, 0)
14 layout.addWidget(QPushButton('Button (1, 1)'), 1, 1)
15 layout.addWidget(QPushButton('Button (1, 2)'), 1, 2)
16 layout.addWidget(QPushButton('Button (2, 0)'), 2, 0)
17 layout.addWidget(QPushButton('Button (2, 1) + 2 Columns Span'), 2, 1)
18 window.setLayout(layout)
19 window.show()
20 sys.exit(app.exec_())

```

In this example, you create an application that uses a `QGridLayout` object to organize its widgets. Notice that, in this case, the second and third arguments you pass to `.addWidget()` are `int` arguments that define the position of each widget.

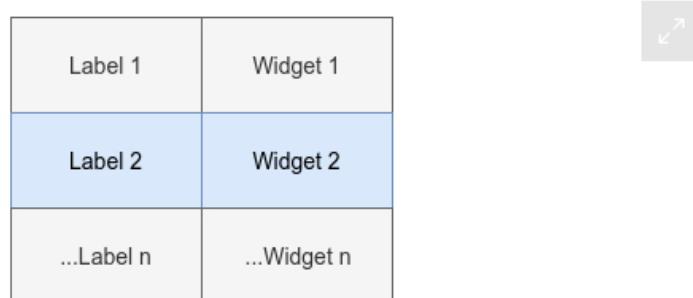
In line 23, you add two more arguments to `.addWidget()`. These arguments are called `rowSpan` and `columnSpan`, and they're the fourth and fifth arguments passed to the function. You can use them to make a widget occupy more than one row or column like you did with `QPushButton('Button (2, 1) + 2 Columns Span')` here.

If you run this code from your command line, then you'll get a window like this:



You can see your widgets arranged in a grid of rows and columns. The last widget occupies more than one cell, as you specified in line 23.

The last layout manager class is `QFormLayout`, which arranges widgets in a two-column layout. The first column usually displays messages in labels. The second column generally contains widgets like `QLineEdit`, `QComboBox`, `QSpinBox`, and so on. These allow the user to enter or edit data regarding the information in the first column. The following diagram shows how form layouts work in practice:



The left column consists of labels, and the right column consists of field widgets. If you're dealing with a [database application](#), then this kind of layout can be an attractive option for increased productivity when you're creating your forms.

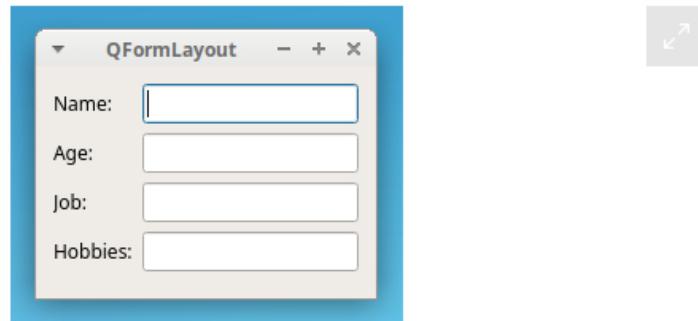
The following example shows you how to create an application that uses a `QFormLayout` object to arrange its widgets:

Python

```
1 # Filename: f_layout.py
2
3 """Form layout example."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QFormLayout
9 from PyQt5.QtWidgets import QLineEdit
10 from PyQt5.QtWidgets import QWidget
11
12 app = QApplication(sys.argv)
13 window = QWidget()
14 window.setWindowTitle('QFormLayout')
15 layout = QFormLayout()
16 layout.addRow('Name:', QLineEdit())
17 layout.addRow('Age:', QLineEdit())
18 layout.addRow('Job:', QLineEdit())
19 layout.addRow('Hobbies:', QLineEdit())
20 window.setLayout(layout)
21 window.show()
22 sys.exit(app.exec_())
```

Lines 15 to 20 do the hard work in this example. Notice that `QFormLayout` has a convenient method called `.addRow()`. You can use this method to add a two-widget row to the layout. The first argument of `.addRow()` should be a label, and the second argument should be any other widget that allows the user to enter or edit data.

If you run this code, then you'll get the following output:



The above figure shows a GUI that uses a form layout. The first column contains labels to ask the user for some information. The second column shows widgets that allow the user to enter or edit the information you asked from them.

[ⓘ Remove ads](#)

Dialogs

With PyQt, you can develop two types of GUI desktop applications. Depending on the class you use to create the main form or window, you'll have one of the following:

1. **A Main Window-Style application:** The application's main window inherits from QMainWindow.
2. **A Dialog-Style application:** The application's main window inherits from QDialog.

You'll start with Dialog-Style applications first. In the next section, you'll cover Main Window-Style applications.

To develop a Dialog-Style application, you need to create a GUI class that inherits from QDialog, which is the base class of all dialog windows. A **dialog window** is always a top-level window that you can use as the main window for your Dialog-Style application.

Note: Dialog windows are also commonly used in Main Window-Style applications for brief communication and interaction with the user.

When dialog windows are used to communicate with the user, they may be:

- **Modal dialogs:** block input to any other visible windows in the same application. You can display a modal dialog by calling .exec_().
- **Modeless dialogs:** operate independently of other windows in the same application. You can display a modeless dialog by using .show().

Dialog windows can also provide a return value and have default buttons (for example, OK and Cancel).

A dialog is always a top-level widget. If it has a parent, then its default location is centered on top of the parent's top-level widget. This kind of dialog will also share the parent's taskbar entry. If you don't set a parent for a given dialog, then the dialog will get its own entry in the system's taskbar.

Here's an example of how you'd use QDialog to develop a Dialog-Style application:

Python

```
1 # Filename: dialog.py
2
3 """Dialog-Style application."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QDialog
9 from PyQt5.QtWidgets import QDialogButtonBox
10 from PyQt5.QtWidgets import QFormLayout
11 from PyQt5.QtWidgets import QLineEdit
12 from PyQt5.QtWidgets import QVBoxLayout
13
```

```

14 class Dialog(QDialog):
15     """Dialog."""
16     def __init__(self, parent=None):
17         """Initializer."""
18         super().__init__(parent)
19         self.setWindowTitle('QDialog')
20         dlgLayout = QVBoxLayout()
21         formLayout = QFormLayout()
22         formLayout.addRow('Name:', QLineEdit())
23         formLayout.addRow('Age:', QLineEdit())
24         formLayout.addRow('Job:', QLineEdit())
25         formLayout.addRow('Hobbies:', QLineEdit())
26         dlgLayout.addLayout(formLayout)
27         btns = QDialogButtonBox()
28         btns.setStandardButtons(
29             QDialogButtonBox.Cancel | QDialogButtonBox.Ok)
30         dlgLayout.addWidget(btns)
31         self.setLayout(dlgLayout)
32
33 if __name__ == '__main__':
34     app = QApplication(sys.argv)
35     dlg = Dialog()
36     dlg.show()
37     sys.exit(app.exec_())

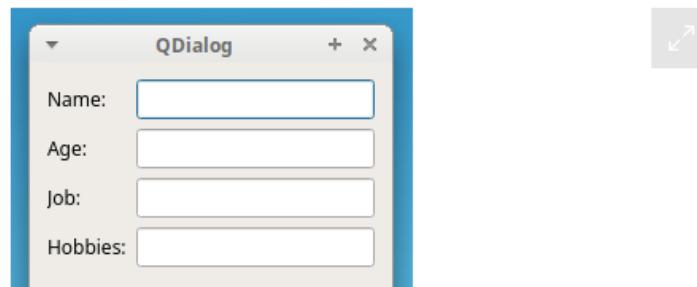
```

This application is a bit more elaborate. Here's what's going on:

- **Line 14** creates a full class `Dialog` for the GUI, which inherits from `QDialog`.
- **Line 20** assigns a `QVBoxLayout` object to `dlgLaout`.
- **Line 21** assigns a `QVFormLayout` object to `formLayout`.
- **Line 22 to 25** add widgets to `formLayout`.
- **Line 26** uses `dlgLayout` to arrange all the widgets on the form.
- **Line 27** provides a convenient object to place the dialog buttons.
- **Lines 28 and 29** add two standard buttons: `ok` and `cancel`.
- **Lines 33 to 37** wrap the boilerplate code in an `if __name__ == '__main__':` idiom. This is considered a best practice for Pythonistas.

Note: If you look at line 26 in the code block above, then you'll notice that layout managers can be nested inside one another. You can nest layouts by calling `.addLayout()` on the container layout and passing in the nested layout as an argument to this method.

The code block above displays the following window:





This is the GUI that you created using QFormLayout for the widgets and QVBoxLayout for the general application's layout (dlgLayout in line 20).

Main Windows

Most of the time, your GUI applications will be **Main Window-Style**. This means that they'll have a [menu bar](#), some toolbars, a status bar, and a central widget that will be the GUI's main element. It's also common that your apps will have several dialog windows to accomplish secondary actions that depend on user input.

You'll use the class `QMainWindow` to develop Main Window-Style applications. You need to inherit from `QMainWindow` to create your main GUI class. An instance of a class that derives from `QMainWindow` is considered to be a **main window**.

`QMainWindow` provides a framework for building your application's GUI. The class has its own built-in layout, which you can use to place the following:

- **One menu bar** is at the top of the window. The [menu bar](#) holds the application's main menu.
- **Several toolbars** are on the sides of the window. [Toolbars](#) are suitable for holding [tool buttons](#) and other kinds of widgets such as `QComboBox`, `QSpinBox`, and more.
- **One central widget** is in the center of the window. The central widget can be of any type, or it can be a composite widget.
- **Several dock widgets** are around the central widget. [Dock widgets](#) are small, movable windows.
- **One status bar** is at the bottom of the window. The [status bar](#) shows information on the application's general status.

You can't create a main window without first setting a central widget. You *must* have a central widget, even if it's just a placeholder. When this is the case, you can use a `QWidget` object as your central widget. You can set the main window's central widget with `.setCentralWidget()`. The main window's layout will allow you to have only one central widget, but it can be a single or a composite widget.

The following code example shows you how to use `QMainWindow` to create a Main Window-Style application:

Python

```
1 # Filename: main_window.py
2
3 """Main Window-Style application."""
4
5 import sys
6
```

```

7  from PyQt5.QtWidgets import QApplication
8  from PyQt5.QtWidgets import QLabel
9  from PyQt5.QtWidgets import QMainWindow
10 from PyQt5.QtWidgets import QStatusBar
11 from PyQt5.QtWidgets import QToolBar
12
13 class Window(QMainWindow):
14     """Main Window."""
15     def __init__(self, parent=None):
16         """Initializer."""
17         super().__init__(parent)
18         self.setWindowTitle('QMainWindow')
19         self.setCentralWidget(QLabel("I'm the Central Widget"))
20         self._createMenu()
21         self._createToolBar()
22         self._createStatusBar()
23
24     def _createMenu(self):
25         self.menu = self.menuBar().addMenu("&Menu")
26         self.menu.addAction('&Exit', self.close)
27
28     def _createToolBar(self):
29         tools = QToolBar()
30         self.addToolBar(tools)
31         tools.addAction('Exit', self.close)
32
33     def _createStatusBar(self):
34         status = QStatusBar()
35         status.showMessage("I'm the Status Bar")
36         self.setStatusBar(status)
37
38 if __name__ == '__main__':
39     app = QApplication(sys.argv)
40     win = Window()
41     win.show()
42     sys.exit(app.exec_())

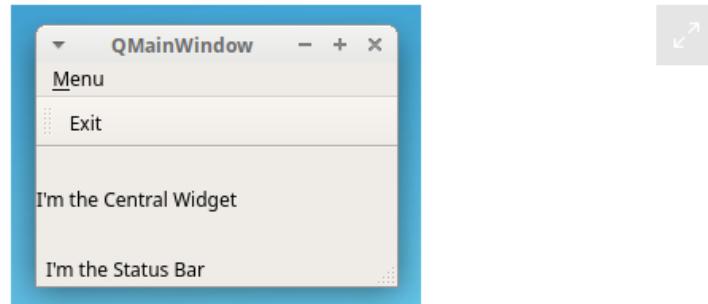
```

Here's how this code works:

- **Line 13** creates a class `Window` that inherits from `QMainWindow`.
- **Line 18** sets the window's title.
- **Line 19** sets a `QLabel` as the central widget.
- **Lines 20 to 22** call private methods in the lines that follow in order to create different GUI elements:
 - **Lines 24 to 26** create the main menu.
 - **Lines 28 to 31** create the toolbar.
 - **Lines 33 to 36** create the status bar.

Note: When you implement different GUI components in their own method, you're making your code more readable and more maintainable. This is not a requirement, however, so you're free to organize your code in the way you like best.

When you run the above code, you'll see a window like the following:



You can see that your Main Window-Style application has the following components:

- One main menu called `Menu`
- One toolbar with a functional `Exit` tool button
- One central widget (a `QLabel` object)
- One status bar at the bottom of the window

So far, you've covered some of the more important graphical components of PyQt5's set of widgets. In the next two sections, you'll cover some other important concepts related to PyQt GUI applications.

[ⓘ Remove ads](#)

Applications

The most basic class you'll use when developing PyQt GUI applications is `QApplication`. This class is at the core of any PyQt application. It manages the application's control flow as well as its main settings. In PyQt, any instance of `QApplication` is considered to be an **application**. Every PyQt GUI application must have one `QApplication` object. Some of the application's responsibilities include:

- Handling initialization and finalization
- Providing the event loop and event handling
- Handling most of the system-wide and application-wide settings
- Providing access to global information, such as the application's directory, screen size, and so on
- Parsing common command-line arguments
- Defining the application's look and feel
- Providing localization capabilities

These are just some of the core responsibilities of `QApplication`. As you can see, this is a fundamental class when it comes to developing PyQt GUI applications!

One of the most important responsibilities of `QApplication` is to provide the event loop and the entire event handling mechanism. Let's take a closer look at the event loop now.

Event Loops

GUI applications are **event-driven**. This means that functions and methods are executed in response to user actions like clicking on a button, selecting an item from a combo box, entering or updating the text in a text edit, pressing a key on the keyboard, and so on. These user actions are generally called **events**.

Events are commonly handled by an **event loop** (also called the **main loop**). An event loop is an infinite loop in which all events from the user, the window system, and any other sources are processed and dispatched. The event loop waits for an event to occur and then dispatches it to perform some task. The event loop continues to work until the application is terminated.

Event loops are used by all GUI applications. The event loop is kind of an infinite loop that waits for the occurrence of events. If an event happens, then the loop checks if the event is a Terminate event. In that case, the loop is terminated and the application exits. Otherwise, the event is sent to the application's event queue for further processing, and the loop starts again.

In PyQt, you can run the application's event loop by calling `.exec_()` on the `QApplication` object.

Note: PyQt was first developed to target Python 2, which already has an exec keyword. In earlier versions, an underscore was added to the end .exec_() to help avoid name conflicts.

PyQt5 targets Python 3, which doesn't have an `exec` keyword. Still, the library provides two methods that start the event loop:

1. .exec_()
 2. .exec()

This means that you can remove `.exec_()` from your code, and you can safely use `.exec()` instead.

For an event to trigger a response action, you need to connect the event with the action you want to be executed. In PyQt5, you can establish that connection by using the signals and slots mechanism. You'll cover these in the next section.

Signals and Slots

PyQt widgets act as **event-catchers**. This means that every widget can catch a specific number of events, like mouse clicks, keypresses, and so on. In response to these events, widgets always emit a **signal**, which is a kind of message that announces a change in its state.

The signal on its own doesn't perform any action. If you want a signal to trigger an action, then you need to connect it to a **slot**. This is the function or method that will perform an action whenever the connecting signal is emitted. You can use any

Python callable (or **callback**) as a slot.

If a signal is connected to a slot, then the slot is called whenever the signal is emitted. If a signal isn't connected to any slot, then nothing happens and the signal is ignored. Here are some of the most useful features of this mechanism:

- A signal can be connected to one or many slots.
- A signal may also be connected to another signal.
- A slot may be connected to one or many signals.

You can use the following syntax to connect a signal to a slot:

Python

```
widget.signal.connect(slot_function)
```

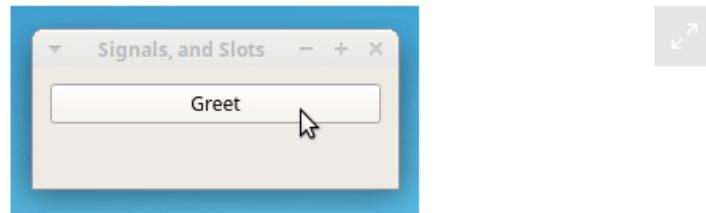
This will connect `slot_function` to `widget.signal`. Whenever `signal` is emitted, `slot_function()` will be called.

This code shows you how to use the signals and slots mechanism:

Python

```
1 # Filename: signals_slots.py
2
3 """Signals and slots example."""
4
5 import sys
6
7 from PyQt5.QtWidgets import QApplication
8 from PyQt5.QtWidgets import QLabel
9 from PyQt5.QtWidgets import QPushButton
10 from PyQt5.QtWidgets import QVBoxLayout
11 from PyQt5.QtWidgets import QWidget
12
13 def greeting():
14     """Slot function."""
15     if msg.text():
16         msg.setText("")
17     else:
18         msg.setText("Hello World!")
19
20 app = QApplication(sys.argv)
21 window = QWidget()
22 window.setWindowTitle('Signals and slots')
23 layout = QVBoxLayout()
24
25 btn = QPushButton('Greet')
26 btn.clicked.connect(greeting) # Connect clicked to greeting()
27
28 layout.addWidget(btn)
29 msg = QLabel('')
30 layout.addWidget(msg)
31 window.setLayout(layout)
32 window.show()
33 sys.exit(app.exec_())
```

In line 13, you create `greeting()`, which you'll use as a slot. Then in line 26, you connect the button's `clicked` signal to `greeting()`. This way, whenever the user clicks on the button, `greeting()` is called and `msg` will alternate between the message `Hello World!` and an empty string:



When you click on Greet, the `Hello World!` message will appear and disappear on your screen.

Note: Every widget has its own set of predefined signals. You can check them out on the widget's documentation page.

If your slot function needs to receive extra arguments, then you can pass them in by using `functools.partial`. For example, you can modify `greeting()` as follows:

Python

```
def greeting(who):
    """Slot function."""
    if msg.text():
        msg.setText('')
    else:
        msg.setText(f'Hello {who}')
```

Now, `greeting()` needs to receive an argument called `who`. If you want to connect this new version of `greeting()` to the `btn.clicked` signal, then you can do something like this:

Python

```
btn.clicked.connect(functools.partial(greeting, 'World!'))
```

For this code to work, you need to import `functools` first. The call to `functools.partial()` returns an object that behaves similarly to calling `greeting()` with `who='World!'`. Now, when the user clicks on the button, the message '`Hello World!`' will be shown in the label as well as before.

Note: You can also use `lambda` to connect a signal to a slot that requires extra arguments. For a practice exercise, try to code the above example using `lambda` instead of `functools.partial()`.

The signals and slots mechanism is what you'll use to give life to your PyQt5 GUI applications. This mechanism will allow you to turn user events into concrete actions. You can dive deeper into the signals and slots mechanism by taking a look

at the [PyQt5 documentation](#).

Now you've finished covering the most important concepts of PyQt5. With this knowledge and the library's documentation at hand, you're ready to start developing your own GUI applications. In the next section, you'll build your first fully-functional GUI application.

Let's go for it!

 Remove ads

Creating a Calculator With Python and PyQt

In this section, you're going to develop a calculator using the [Model-View-Controller \(MVC\)](#) design pattern. This pattern has three layers of code, each with different roles:

1. **The model** takes care of your app's [business logic](#). It contains the core functionality and data. For your calculator, the model will handle the calculations.
2. **The view** implements your app's GUI. It hosts all the widgets the end-user would need to interact with the application. The view also receives user actions and events. For your calculator, the view will be the window you'll see on your screen.
3. **The controller** connects the model and the view to make the application work. User events (or requests) are sent to the controller, which puts the model to work. When the model delivers the requested result (or data) in the right format, the controller forwards it to the view. For your calculator, the controller will receive user events from the GUI, ask the model to perform calculations, and update the GUI with the result.

Here's a step-by-step MVC pattern for a GUI desktop application:

1. The user performs an action or request (event) on the view (GUI).
2. The view notifies the controller about the user's action.
3. The controller gets the user's request and queries the model for a response.
4. The model processes the controller query, performs the required operations, and returns an answer or result.
5. The controller receives the model's answer and updates the view accordingly.
6. The user finally sees the requested result on the view.

You'll use this MVC design pattern to build your calculator.

Creating the Skeleton

You'll start by implementing a basic skeleton for your application, called `pycalc.py`. You can find this script and the rest of the source code at the link below:

Download Code: Click here to download the code you'll use to build a calculator in Python with PyQt in this tutorial.

If you'd prefer to code the project on your own, then go ahead and create `pycalc.py` in your current working directory. Then, open the file in your code editor and type the following code:

Python

```
#!/usr/bin/env python3

# Filename: pycalc.py

"""PyCalc is a simple calculator built using Python and PyQt5."""

import sys

# Import QApplication and the required widgets from PyQt5.QtWidgets
from PyQt5.QtWidgets import QApplication
from PyQt5.QtWidgets import QMainWindow
from PyQt5.QtWidgets import QWidget

__version__ = '0.1'
__author__ = 'Leodanis Pozo Ramos'

# Create a subclass of QMainWindow to setup the calculator's GUI
class PyCalcUi(QMainWindow):
    """PyCalc's View (GUI)."""
    def __init__(self):
        """View initializer."""
        super().__init__()
        # Set some main window's properties
        self.setWindowTitle('PyCalc')
        self.setFixedSize(235, 235)
        # Set the central widget
        self._centralWidget = QWidget(self)
        self.setCentralWidget(self._centralWidget)

    # Client code
    def main():
        """Main function."""
        # Create an instance of QApplication
        pycalc = QApplication(sys.argv)
        # Show the calculator's GUI
        view = PyCalcUi()
        view.show()
        # Execute the calculator's main loop
        sys.exit(pycalc.exec_())

    if __name__ == '__main__':
        main()
```

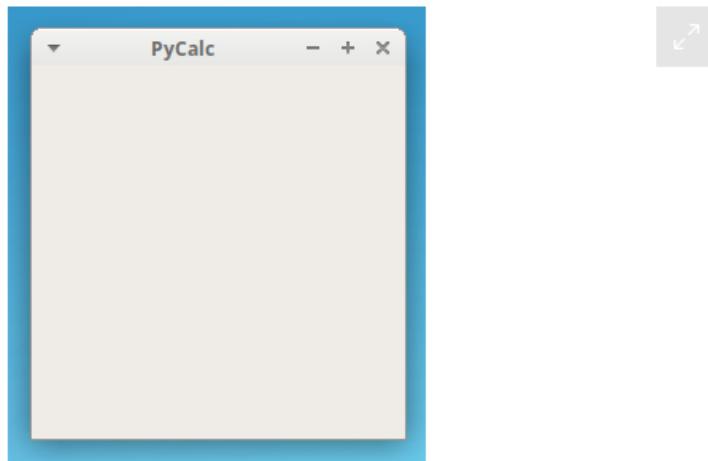
This script implements all the code you'll need to run a basic GUI application. You'll use this skeleton to build your calculator. Here's how it works:

- **Lines 10 to 12** import the required modules and classes from

PyQt5.QtWidgets.

- **Line 18** creates the GUI with the class PyCalcUi. Note that this class inherits from QMainWindow.
- **Line 24** sets the window's title to PyCalc.
- **Line 25** uses .setFixedSize() to give the window a fixed size. This ensures that the user won't be able to resize the window.
- **Line 27** creates a QWidget object to play the role of a central widget. Remember that since your GUI class inherits from QMainWindow, you need a central widget. This object will be the parent for the rest of the GUI component.
- **Line 31** defines your calculator's main function, which is considered a best practice. This function will be the entry point to the application. Inside main(), your program does the following:
 - **Line 34** creates a QApplication object pycalc.
 - **Line 37** shows the GUI with view.show().
 - **Line 39** runs the application's event loop with pycalc.exec_().

When you run the script, the following window will appear on your screen:



This is your GUI application skeleton.

Completing the View

The GUI you have at this point doesn't really look like a calculator. Let's finish the GUI by adding the display and buttons for the numbers. You'll also add buttons for basic math operations and for clearing the display.

First, you'll need to add the following imports to the top of your file:

Python

```
from PyQt5.QtCore import Qt  
from PyQt5.QtWidgets import QGridLayout
```

```
from PyQt5.QtWidgets import QLineEdit
from PyQt5.QtWidgets import QPushButton
from PyQt5.QtWidgets import QVBoxLayout
```

You're going to use a QVBoxLayout for the calculator's general layout. You'll also use a QGridLayout object to arrange the buttons. Finally, you import QLineEdit for the display and QPushButton for the buttons. There should now be eight import statements at the top of your file.

Now you can update the initializer for PyCalcUi:

Python

```
# Create a subclass of QMainWindow to setup the calculator's GUI
class PyCalcUi(QMainWindow):
    """PyCalc's View (GUI)."""
    def __init__(self):
        """View initializer."""
        super().__init__()
        # Set some main window's properties
        self.setWindowTitle('PyCalc')
        self.setFixedSize(235, 235)
        # Set the central widget and the general layout
        self.generalLayout = QVBoxLayout()
        self._centralWidget = QWidget(self)
        self.setCentralWidget(self._centralWidget)
        self._centralWidget.setLayout(self.generalLayout)
        # Create the display and the buttons
        self._createDisplay()
        self._createButtons()
```

Here, you've added the highlighted lines of code. You'll use a QVBoxLayout to place the display at the top and the buttons in a grid layout at the bottom.

The calls to `_createDisplay()` and `_createButtons()` won't work, because you haven't yet implemented those methods. Let's fix that by coding `_createDisplay()`:

Python

```
class PyCalcUi(QMainWindow):
    # Snip
    def _createDisplay(self):
        """Create the display."""
        # Create the display widget
        self.display = QLineEdit()
        # Set some display's properties
        self.display.setFixedHeight(35)
        self.display.setAlignment(Qt.AlignRight)
        self.display.setReadOnly(True)
        # Add the display to the general layout
        self.generalLayout.addWidget(self.display)
```

To create the display widget, you use a QLineEdit object. Then you set the following display properties:

- The display has a fixed height of 35 pixels.
- The display shows the text as left-aligned.
- The display is set to read-only to avoid direct editing.

The last line adds the display to the calculator's general layout with `generalLayout.addWidget()`.

Next, you'll implement `._createButtons()` to create buttons for your calculator. You'll use a dictionary to hold each button's text and position on the grid. You'll also use `QGridLayout` to arrange the buttons on the calculator's window. The final code will look like this:

Python

```
class PyCalcUi(QMainWindow):
    # Snip
    def _createButtons(self):
        """Create the buttons."""
        self.buttons = {}
        buttonsLayout = QGridLayout()
        # Button text | position on the QGridLayout
        buttons = {'7': (0, 0),
                   '8': (0, 1),
                   '9': (0, 2),
                   '/': (0, 3),
                   'C': (0, 4),
                   '4': (1, 0),
                   '5': (1, 1),
                   '6': (1, 2),
                   '*': (1, 3),
                   '(': (1, 4),
                   '1': (2, 0),
                   '2': (2, 1),
                   '3': (2, 2),
                   '-': (2, 3),
                   ')': (2, 4),
                   '0': (3, 0),
                   '00': (3, 1),
                   '.': (3, 2),
                   '+': (3, 3),
                   '=': (3, 4),
        }
        # Create the buttons and add them to the grid layout
        for btnText, pos in buttons.items():
            self.buttons[btnText] = QPushButton(btnText)
            self.buttons[btnText].setFixedSize(40, 40)
            buttonsLayout.addWidget(self.buttons[btnText], pos[0], pos[1])
        # Add buttonsLayout to the general layout
        self.generalLayout.addLayout(buttonsLayout)
```

You first create an empty dictionary `self.buttons` to hold the calculator buttons. Then, you create a temporary dictionary to store their labels and relative positions on the grid layout (`buttonsLayout`). Inside the `for` loop, you create the buttons and add them to both `self.buttons` and `buttonsLayout`. Every button will have a fixed

size of 40x40 pixels, which you set with `.setFixedSize(40, 40)`.

Note: When it comes to widget size, you'll rarely find measurement units in the PyQt documentation. It's assumed that the measurement unit is **pixels** (except for `QPrinter`, which uses **points**).

Now, the calculator's GUI (or view) can show the display and the buttons. But there's still no way to update the information shown in the display. You can fix this by adding a few extra methods:

1. `.setDisplayText()` to set and update the display's text
2. `.displayText()` to get the current display's text
3. `.clearDisplay()` to clear the display's text

These methods will form the GUI public interface and complete the view class for your Python calculator. Here's a possible implementation:

Python

```
class PyCalcUi(QMainWindow):  
    # Snip  
    def setDisplayText(self, text):  
        """Set display's text."""  
        self.display.setText(text)  
        self.display.setFocus()  
  
    def displayText(self):  
        """Get display's text."""  
        return self.display.text()  
  
    def clearDisplay(self):  
        """Clear the display."""  
        self.setDisplayText('')
```

Here's what each function does:

- `.setDisplayText()` uses `.setText()` to set and update the display's text, and `.setFocus()` to set the cursor's focus on the display.
- `.displayText()` is a getter method that returns the display's current text. When the user clicks on the equals sign (=), the program will use the return value of `.displayText()` as the math expression to be evaluated.
- `.clearDisplay()` sets the display's text to an empty string ('') so the user can introduce a new math expression.

Now your calculator's GUI is ready! When you run the application, you'll see a window like this one:





You've completed the calculator's GUI interface. However, if you try to do some calculations, then you'll notice that the calculator doesn't do anything just yet. That's because you haven't implemented the model or the controller. Next, you'll add a basic controller class to start giving life to your calculator.

[ⓘ Remove ads](#)

Creating a Basic Controller

In this section, you're going to code the calculator's controller class. This class will connect the view to the model. You'll use the controller class to make the calculator perform actions in response to user events. You'll start with the following import:

Python

```
from functools import partial
```

At the top of `pycalc.py`, you import `partial()` to connect signals with methods that need to take extra arguments.

Your controller class needs to perform three main tasks:

1. Access the GUI's public interface
2. Handle the creation of math expressions
3. Connect button `clicked` signals with the appropriate slots

This will ensure that your calculator is working correctly. Here's how you code the controller class:

Python

```
# Create a Controller class to connect the GUI and the model
class PyCalcCtrl:
    """PyCalc Controller class."""
    def __init__(self, view):
        """Controller initializer."""
        self._view = view
        # Connect signals and slots
        self._connectSignals()

    def _buildExpression(self, sub_exp):
        """Build expression."""
        expression = self._view.displayText() + sub_exp
        self._view.setDisplayText(expression)
```

```

def _connectSignals(self):
    """Connect signals and slots."""
    for btnText, btn in self._view.buttons.items():
        if btnText not in {'=', 'C'}:
            btn.clicked.connect(partial(self._buildExpression, btnText))

    self._view.buttons['C'].clicked.connect(self._view.clearDisplay)

```

The first thing you do is give `PyCalcCtrl` an instance of the view `PyCalcUi`. You'll use this instance to gain full access to the view's public interface. Next, you create `_buildExpression()` to handle the creation of math expressions. This method also updates the calculator's display in response to user input.

Finally, you use `_connectSignals()` to connect the **printable** buttons with `_buildExpression()`. This allows your users to create math expressions by clicking on the calculator's buttons. In the last line, you connect the clear button (c) to `_view.clearDisplay()`. This method will clear up the text on the display.

For this new controller class to work, you need to update `main()`:

Python

```

# Client code
def main():
    """Main function."""
    # Create an instance of QApplication
    pycalc = QApplication(sys.argv)
    # Show the calculator's GUI
    view = PyCalcUi()
    view.show()
    # Create instances of the model and the controller
    PyCalcCtrl(view=view)
    # Execute calculator's main loop
    sys.exit(pycalc.exec_())

```

This code creates an instance of `PyCalcCtrl(view=view)` with the `view` passed in as an argument. This will initialize the controller and connect the signals and slots to give your calculator some functionality.

If you run the application, then you'll see something like the following:



As you can see, the calculator already has some useful functionalities! Now, you can build math expressions by clicking on the buttons. Notice that the equals sign (=) doesn't work yet. To fix this, you need to implement the calculator's model.

Implementing the Model

The model is the layer of code that takes care of the business logic. In this case, the business logic is all about basic math calculations. Your model will evaluate the math expressions introduced by your users. Since the model needs to handle errors, you're going to define the following global constant:

Python

```
ERROR_MSG = 'ERROR'
```

This is the message the user will see if they introduce an invalid math expression.

Your model will be a single function:

Python

```
# Create a Model to handle the calculator's operation
def evaluateExpression(expression):
    """Evaluate an expression."""
    try:
        result = str(eval(expression, {}, {}))
    except Exception:
        result = ERROR_MSG

    return result
```

Here, you use `eval()` to evaluate a string as an expression. If this is successful, then you'll return the result. Otherwise, you'll return the error message. Note that this function isn't perfect. It has a couple of important issues:

- The `try...except` block doesn't catch any specific exception, which is not a best practice in Python.
- The function is based on the use of `eval()`, which can lead to some serious [security issues](#). The general advice is to only use `eval()` on trusted input.

You're free to rework the function to make it more reliable and secure. For this tutorial, you'll use the function as-is.

[ⓘ Remove ads](#)

Completing the Controller

Once you've completed the calculator's model, you can finish the controller. The final version of `PyCalcCtrl` will include logic to process the calculations and to make sure the equals sign (=) works correctly:

Python

```
# Create a Controller class to connect the GUI and the model
class PyCalcCtrl:
    """PyCalc's Controller."""
    def __init__(self, model, view):
        """Controller initializer."""
        self._evaluate = model
        self._view = view
        # Connect signals and slots
        self._connectSignals()

    def _calculateResult(self):
        """Evaluate expressions."""
        result = self._evaluate(expression=self._view.displayText())
        self._view.setDisplayText(result)

    def _buildExpression(self, sub_exp):
        """Build expression."""
        if self._view.displayText() == ERROR_MSG:
            self._view.clearDisplay()

        expression = self._view.displayText() + sub_exp
        self._view.setDisplayText(expression)

    def _connectSignals(self):
        """Connect signals and slots."""
        for btnText, btn in self._view.buttons.items():
            if btnText not in {'=', 'C'}:
                btn.clicked.connect(partial(self._buildExpression, btnText))

        self._view.buttons['='].clicked.connect(self._calculateResult)
        self._view.display.returnPressed.connect(self._calculateResult)
        self._view.buttons['C'].clicked.connect(self._view.clearDisplay)
```

The new lines of code are highlighted. First, you add a new parameter to the `init` function. Now the class receives instances from both the model and the view. Then in `_calculateResult()`, you take the display's content, evaluate it as a math expression, and finally show the result in the display.

You also add an `if statement` to `_buildExpression()` to check if an error has occurred. If so, then you clear the display and start over with a new expression. Finally, you add two more connections inside `_connectSignals()`. The first enables the equals sign (`=`). The second ensures that when the user hits `Enter ↵`, the calculator will process the expression as expected.

For all this code to work, you need to update `main()`:

Python

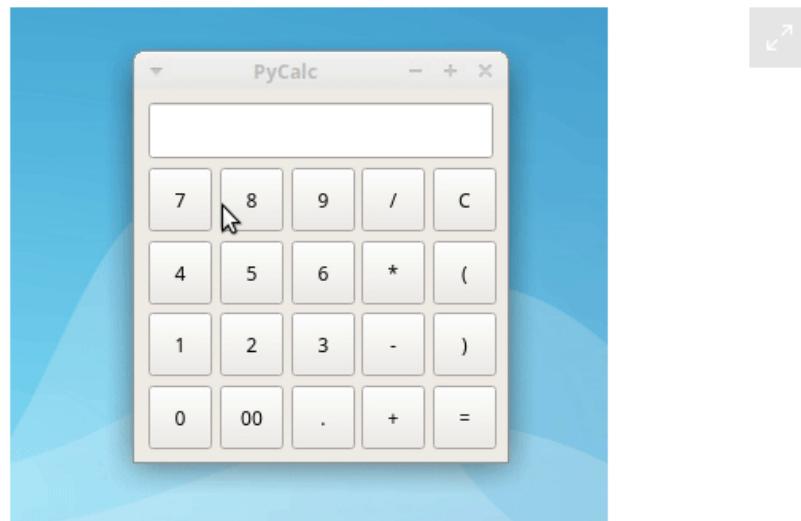
```
# Client code
def main():
    """Main function."""
    # Create an instance of `QApplication`
    pycalc = QApplication(sys.argv)
```

```
# Show the calculator's GUI
view = PyCalcUi()
view.show()
# Create instances of the model and the controller
model = evaluateExpression
PyCalcCtrl(model=model, view=view)
# Execute calculator's main loop
sys.exit(pycalc.exec_())
```

Here, your `model` holds a reference to `evaluateExpression()`. In addition, `PyCalcCtrl()` now receives two arguments: the `model` and the `view`.

Running the Calculator

Now that you've finished the code, it's time for a test! If you run the application, then you'll see something like this:



To use PyCalc, enter a valid math expression with your mouse. Then, press `Enter ↵` or click on the equals sign (=) to see the result on the calculator's display.

Congrats! You've developed your first fully-functional GUI desktop application with Python and PyQt!

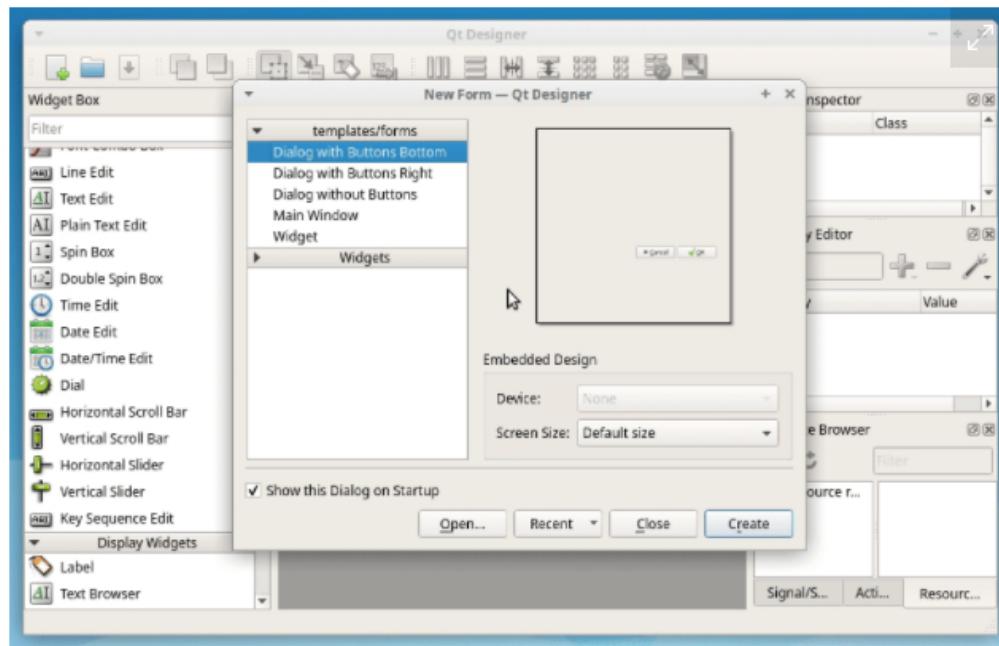
Additional Tools

PyQt5 offers quite a useful set of additional tools to help you build solid, modern, and full-featured GUI applications. Here are some of the most remarkable tools you can use:

- [Qt Designer](#)
- The set of [internationalization tools](#)
- The PyQt5 [resource system](#)

[Qt Designer](#) is the Qt tool for designing and building graphical user interfaces. You can use it to design widgets, dialogs, or complete main windows by using on-screen forms and a drag-and-drop mechanism. The following figure shows some of the Qt

Designer's features:



Qt Designer uses XML `.ui` files to store your GUI designs. You can load them with `QUiLoader`. PyQt includes a module called `uic` to help with this. You can also convert the `.ui` file content into Python code with a command-line tool called `pyuic5`.

PyQt5 also provides a comprehensive set of tools for the **Internationalization** of apps into local languages. `pylupdate5` creates and updates translation (`.ts`) files. Then, `Qt Linguist` updates the generated `.ts` files with translations of the strings. It also releases the `.ts` files as `.qm` files. These `.qm` files are compact binary equivalents that can be used directly by the application.

Finally, you can use the PyQt5 **resource system**, which is a facility for embedding resources such as icons and translation files. To use this tool, you need to generate a `.qrc` file. This is an XML file that's used to specify which resource files are to be embedded. Once you have this file ready, you can use `pyrcc5` to generate a Python module that contains the embedded resources.

Remove ads

Conclusion

Graphical User Interface (GUI) desktop applications still hold a substantial share of the software development market. Python offers a handful of frameworks and libraries that can help you develop modern and robust GUI applications.

In this tutorial, you learned how to use `PyQt`, which is arguably one of the most popular and solid libraries for GUI desktop application development in Python. Now you know how to effectively use both Python and PyQt to build modern GUI desktop applications.

You're now able to:

- Create Graphical User Interfaces with Python and PyQt

- Create graphical user interfaces with `PyQt`
- Connect user events to concrete actions in your application
- Create fully-functional GUI desktop applications to solve real-world problems

Now you can use Python and PyQt to give life to your desktop GUI applications! You can get the source code for the calculator project and all code examples at the link below:

Download Code: [Click here to download the code you'll use](#) to build a calculator in Python with PyQt in this tutorial.

Further Reading

If you want to dive deeper into PyQt and its related tools, then you can take a look at some of these resources:

- [PyQt5 Documentation](#)
- [PyQt4 Documentation](#)
- [Qt v5 Documentation](#)
- [PyQt Wiki](#)
- [Rapid GUI Programming with Python and Qt](#)
- [Qt Designer Manual](#)
- [Qt for Python \(PySide2\) documentation](#)

Although the PyQt5 Documentation is the first resource listed here, some important parts are still missing or incomplete. Fortunately, you can use the [PyQt4 Documentation](#) to fill in the blanks. The [Class Reference](#) is particularly useful for gaining a complete understanding of widgets and classes.

If you're using the PyQt4 documentation as a reference for PyQt5 classes, then bear in mind that the classes will be slightly different and may behave differently, too. Another option would be to use the original [Qt v5 Documentation](#) and its [Class Reference](#) instead. In this case, you may need some background in C++ to properly understand the code samples.

[Mark as Completed](#)



Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email Address

[Send Me Python Tricks »](#)

About Leodanis Pozo Ramos



Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

[» More about Leodanis](#)

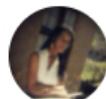
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



Aldren



Geir Arne



Jaya



Joanna



Mike

Master Real-World Python Skills
With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

[Level Up Your Python Skills »](#)

What Do You Think?

[Tweet](#) [Share](#) [Email](#)

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: [gui](#) [intermediate](#) [projects](#)

— FREE Email Series —

Python Tricks

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email...

[Get Python Tricks »](#)



No spam. Unsubscribe any time.

All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#) [community](#)

Table of Contents

- [Understanding PyQt](#)
- [Installing PyQt](#)
- [Creating Your First PyQt Application](#)
- [Considering Code Styles](#)
- [Learning the Basics of PyQt](#)
- [Creating a Calculator With Python and PyQt](#)
- [Additional Tools](#)
- [Conclusion](#)
- [Further Reading](#)

[Mark as Completed](#)



[Tweet](#)

[Share](#)

[Email](#)

[Remove ads](#)

© 2012–2022 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) ·
[Twitter](#) · [Facebook](#) · [Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy Policy](#) ·
[Energy Policy](#)

· [Advertise](#) · [Contact](#)

Happy Pythoning!

Help