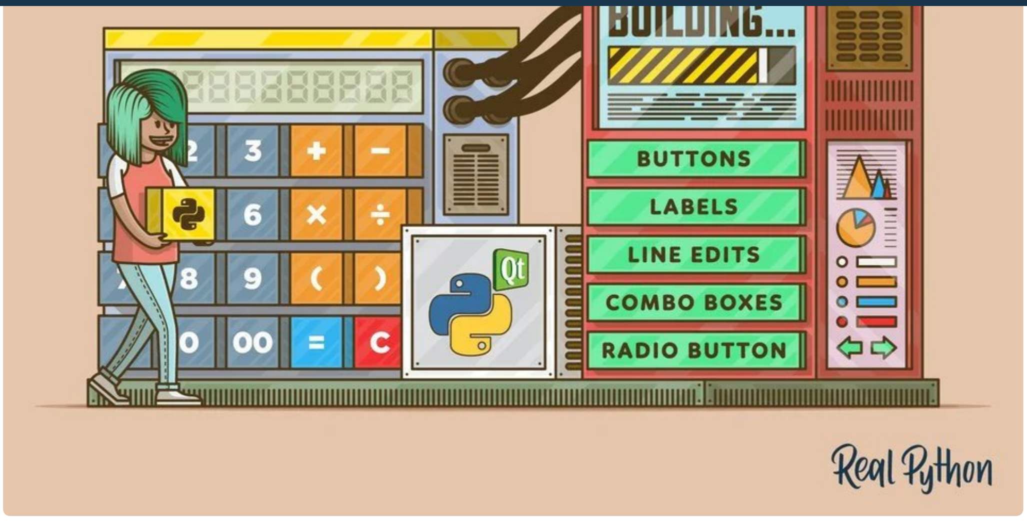


Python and PyQt: Building a GUI Desktop Calculator

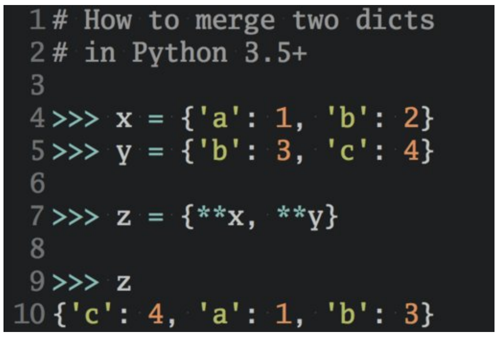
by Leodanis Pozo Ramos Aug 29, 2022 70 Comments

gui intermediate projects

Mark as Completed

Tweet Share Email

Table of Contents



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

over the software development market, there's still demand for traditional **graphical user interface (GUI)** desktop applications. If you're interested in building these kinds of applications in Python, then you'll find a wide variety of libraries to choose from. They include Tkinter, wxPython, PyQt, PySide, and a few others.

In this tutorial, you'll learn the basics of building GUI desktop applications with Python and **PyQt**.

In this tutorial, you'll learn how to:

- Create **graphical user interfaces** with Python and PyQt
- Connect the **user's events** on the app's GUI with the **app's logic**
- Organize a PyQt app using a proper **project layout**
- Create a **fully functional GUI application** with PyQt

For this tutorial, you'll create a calculator app with Python and PyQt. This short project will help you grasp the fundamentals and get you up and running with this GUI 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 that you'll use to build a calculator in Python with PyQt in this tutorial.

Getting to Know PyQt

PyQt is a Python binding for Qt, which is a set of C++ libraries and development tools providing platform-independent abstractions for graphical user interfaces (GUIs). Qt also provides tools for networking, threads, regular expressions, SQL databases, SVG, OpenGL, XML, and many other powerful features.

Developed by RiverBank Computing Ltd, PyQt's latest editions are:

1. PyQt5: An edition that's built against Qt 5.x only
2. PyQt6: An edition that's built against Qt 6.x only

In this tutorial, you'll use PyQt6, as this version is the future of the library. From now on, be sure to consider any mention of PyQt as a reference to PyQt6.

Note: If you want to dive deeper into the differences between these two versions of the library, then check out the PyQt6 documentation on the topic.

PyQt6 is based on Qt v6. Therefore, it provides classes and tools for GUI creation, XML handling, network communication, regular expressions, threads, SQL databases, web browsing, and other technologies available in Qt. PyQt6 implements binding for many Qt classes in a set of Python modules, which are organized in a top-level Python package called `PyQt6`. For PyQt6 to work, you need Python 3.6.1 or later.

PyQt6 is compatible with Windows, Unix, Linux, macOS, iOS, and Android. This is an attractive feature if you're looking for a GUI framework to develop multiplatform applications that have a native look and feel on each platform.

PyQt6 is available under two licenses:

1. The Riverbank Commercial License
2. The General Public License (GPL), version 3

Your PyQt6 license must be compatible with your Qt license. If you use the GPL license, then your code must also use a GPL-compatible license. If you want to use PyQt6 to create commercial applications, then you 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 the official Qt for Python. Its Python package is called PySide.
>
> PyQt and PySide are both built on top of Qt. Their APIs are quite similar because they reflect the Qt API. That's why porting PyQt code to PySide 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 PyQt6 vs PySide6.

If you need more information about PyQt6 licensing, then check out the license FAQs page on the project's official documentation.

Your Guide to the Python Programming Language and a Best Practices Handbook
python-guide.org
The Hitchhiker's Guide to Python

Installing PyQt

You have several options for installing PyQt on your system or development environment. The recommended option is to use to use binary wheels. Wheels are the standard way to install Python packages from the Python package index, PyPI.

In any case, you need to consider that wheels for PyQt6 are only available for Python 3.6.1 and later. There are wheels for Linux, macOS, and Windows (64-bit).

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

Another installation option is to build PyQt 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 check out what the library's documentation recommends in those cases.

Alternatively, you have the option of using package managers, such as APT on Linux or Homebrew on macOS, to install PyQt6. In the next few sections, you'll go through some of the options for installing PyQt6 from different sources and on different platforms.

Virtual Environment Installation With `pip`

Most of the time, you should create a Python virtual environment to install PyQt6 in an isolated way. To create a virtual environment and install PyQt6 in it, run the following on your command line:

Windows | Linux + macOS

Windows Command Prompt

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install pyqt6
```

Here, you first create a virtual environment using the `venv` module from the standard library. Then you activate it, and finally you install PyQt6 in it using `pip`. Note that you must have Python 3.6.1 or later for the install command to work correctly.

System-Wide Installation With `pip`

You'll rarely need to install PyQt directly on your system Python environment. If you ever need to do this kind of installation, then run the following command on your command line or in your terminal window without activating any virtual environment:

Shell

```
$ python -m pip install pyqt6
```

With this command, you'll install PyQt6 in your system Python environment directly. 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.

Even though this is a fast way to install PyQt6 and start using it right away, it's not the recommended approach. The recommended approach is to use a Python virtual environment, as you learned in the previous section.

Platform-Specific Installation

Several Linux distributions include binary packages for PyQt6 in their repositories. If this your case, then you can install the library using the distribution's package manager. On Ubuntu, for example, you can use the following command:

Shell

```
$ sudo apt install python3-pyqt6
```

With this command, you'll install PyQt6 and all of 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 the `sudo` command.

If you're a macOS user, then you can install PyQt6 using the Homebrew package manager. To do this, open a terminal and run the following command:

```shell
$ brew install pyqt6
```

After running this command, you'll have PyQt6 installed on your Homebrew Python environment, and it'll be ready for you to use.

If you use a package manager on Linux or macOS, then there's a chance you won't get the latest version of PyQt6. A `pip` installation will be better if you want to ensure that you have the latest release.

Creating Your First PyQt Application

Now that you have a working PyQt installation, you're ready to create your first GUI app. You'll create a `Hello, World!` application with Python and PyQt. Here are the steps that you'll follow:

1. Import `QApplication` and all the required widgets from `PyQt6.QtWidgets`.
2. Create an instance of `QApplication`.
3. Create 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 that you'll code in this section by clicking the link below:

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

To kick things off, start by creating a new file called `hello.py` in your current working directory:

Python

```python
# hello.py

"""Simple Hello, World example with PyQt6."""

import sys

# 1. Import QApplication and all the required widgets
from PyQt6.QtWidgets import QApplication, QLabel, QWidget
```

First, you import `sys`, which will allow you to handle the application's termination and exit status through the `exit()` function. Then you import `QApplication`, `QLabel`, and `QWidget` from `QtWidgets`, which is part of the `PyQt6` package. With these imports, you're done with step one.

To complete step two, you just need to create an instance of `QApplication`. Do this as you would create an instance of any Python class:

Python

```python
# hello.py
# ...

# 2. Create an instance of QApplication
app = QApplication([])
```

In this line of code, you create the instance of `QApplication`. You should create your `app` instance before you create any GUI object in PyQt.

Internally, the `QApplication` class deals with command-line arguments. That's why you need to pass in a list of command-line arguments to the class constructor. In this example, you use an empty list because your app won't be handling any command-line arguments.

Note: You'll often find that developers pass `sys.argv` to the constructor of `QApplication`. This object contains the list of command-line arguments passed into a Python script. If your application needs to accept command-line arguments, then you should use `sys.argv` to handle them. Otherwise, you can just use an empty list, like you did in the above example.

Step three involves creating the application's GUI. In this example, your GUI will be based on the `QWidget` class, which is the base class of all user interface objects in PyQt.

Here's how you can create the app's GUI:

Python

```python
# hello.py
# ...

# 3. Create your application's GUI
window = QWidget()
window.setWindowTitle("PyQt App")
window.setGeometry(100, 100, 280, 80)
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 that you'll need to create the application's window, or form. As its names suggests, `.setWindowTitle()` sets the window's title in your application. In this example, the app's window will show `PyQt App` as its title.

Note: More precisely, this step requires you to create the app's **top-level** or **main window**. The term **application's GUI** is a bit generic. Typically, an application's GUI consists of more than one window.

You can use `.setGeometry()` to define the window's size and screen position. The first two arguments are the `x` and `y` screen coordinates where the window will be placed. The third and fourth arguments are the window's `width` and `height`.

Every GUI application needs widgets, or graphical components that make the app’s GUI. In this example, you use a `QLabel` widget, `helloMsg`, to show the message `Hello, World!` on your application’s window.

`QLabel` objects can display HTML-formatted text, so you can use the HTML element `"<h1>Hello, World!</h1>"` to provide the desired text as an `h1` header. Finally, you use `.move()` to place `helloMsg` at the coordinates `(60, 15)` on the application’s window.

> **Note:** In PyQt, you can use any widget—a subclass of `QWidget`—as a top-level window. The only condition is that the target widget must not have a `parent` widget. When you use a widget as your top-level window, PyQt automatically provides it with a title bar and turns it into a normal window.
>
> The **parent-child relationship** between widgets has two complementary purposes. A widget with no `parent` is considered a main or **top-level window**. In contrast, a widget with an explicit `parent` is a **child widget**, and it’s shown within its parent.
>
> This relationship is also known as **ownership**, with parents owning their children. The PyQt ownership model ensures that if you delete a parent widget, such as your top-level window, then all of its child widgets will automatically be deleted as well.
>
> To avoid memory leaks, you should always make sure that any `QWidget` object has a parent, with the sole exception of your top-level windows.

You’re done with step three, so you can continue with the final two steps and get your PyQt GUI application ready to run:

Python

```python
# hello.py
# ...
# 4. Show your application's GUI
window.show()

# 5. Run your application's event loop
sys.exit(app.exec())
```

In this code snippet, you call `.show()` on `window`. The call to `.show()` schedules a **paint event**, which is a request to paint the widgets that compose a GUI. This event is then added to the application's event queue. You'll learn more about PyQt's event loop in a later section.

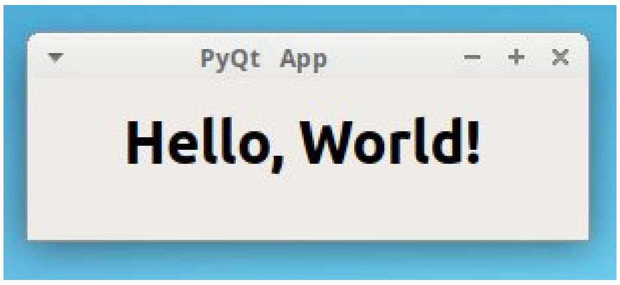
Finally, you start the application's event loop by calling `.exec()`. The call to `.exec()` is wrapped in a call to `sys.exit()`, which allows you to cleanly exit Python and release memory resources when the application terminates.

You can run your first PyQt app with the following command:

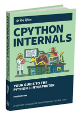
Shell

```shell
$ python hello.py
```

When you run this script, you'll see a window that'll look something like this:



Your application shows a window based on `QWidget`. The window displays the `Hello, World!` message. To show the message, it uses a `QLabel` widget. And with that, you've written your first GUI desktop application using PyQt and Python! Isn't that cool?



Considering Code Styles

If you check the code of your sample GUI application from the previous section, then you'll notice that PyQt's API doesn't follow PEP 8 coding style and naming conventions. PyQt is built around Qt, which is written in C++ and uses the camel case naming style for functions, methods, and variables. That said, when you start writing a PyQt project, you need to decide which naming style you'll use.

In this regard, 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 PyQt-related code, then you should stick to the framework's coding style. In this tutorial, you'll follow the PyQt coding style for consistency. You'll use camel case instead of the usual Python snake case.

Learning the Basics of PyQt

You'll need to master the basic components of PyQt if you want to proficiently use this library to develop your GUI applications. Some of these components include:

- Widgets
- Layout managers
- Dialogs
- Main windows

- Applications
- Event loops
- Signals and slots

These elements are the building blocks of any PyQt GUI application. Most of them are represented as Python classes that live in the `PyQt6.QtWidgets` module. These elements are extremely important. You'll learn more about them in the following few sections.

Widgets

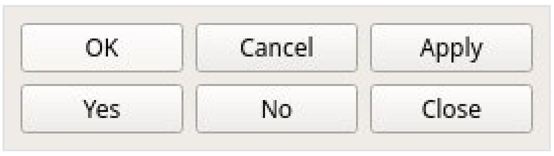
Widgets are rectangular graphical components that you can place on your application's windows to build the GUI. Widgets have several attributes and methods that allow you to tweak their appearance and behavior. They can also paint a representation of themselves on the screen.

Widgets also detect mouse clicks, keypresses, and other events from the user, the window system, and other sources. Each time a widget catches an event, it emits a signal to announce its state change. PyQt has a rich and modern collection of widgets. Each of those widgets serves a different purpose.

Some of the most common and useful PyQt widgets are:

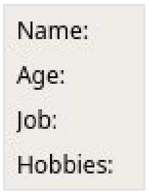
- Buttons
- Labels
- Line edits
- Combo boxes
- Radio buttons

First up is the **button**. You can create a button by instantiating `QPushButton`, a class that provides a classic 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 widgets in any GUI. When someone clicks them, your app commands the computer to perform actions. This is how you can execute computations when a user clicks a button.

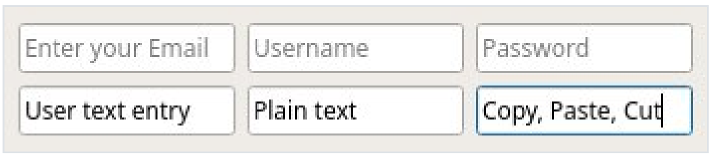
Up next are **labels**, which you can create with `QLabel`. Labels let you display useful information as text or images:



You'll use labels like these to explain how to use your app's GUI. You can tweak a label's appearance in several ways. A label can even accept HTML-formatted text, as you saw earlier. You can also use labels to specify a keyboard shortcut to move the cursor focus to a given widget on your GUI.

Another common widget is the **line edit**, also known as the **input box**. This widget allows you to enter a single line of text. You can create line edits with the `QLineEdit` class. Line edits are useful when you need to get the user's input as plain text.

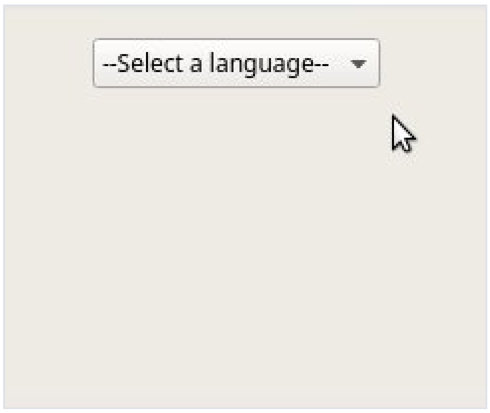
Here's how line edits look on a Linux system:



Line edits like these automatically 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.

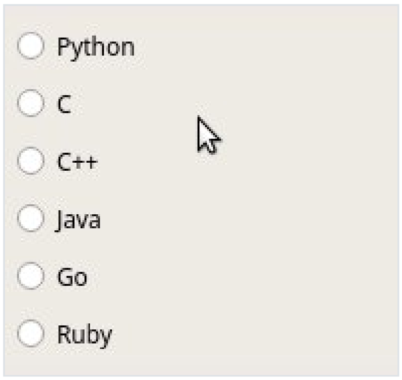
Combo boxes are another fundamental type of widget in GUI applications. You can create them by instantiating `QComboBox`. A combo box will present your user with a dropdown list of options in a way that takes up minimal screen space.

Here's an example of a combo box that provides a dropdown list of popular programming languages:



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

The last widget that you'll learn about is the **radio button**, which you can create with `QRadioButton`. A `QRadioButton` object is an option button that you can click to switch on. Radio buttons are useful when you need the user to select one of many options. All options in a radio button are visible on the screen at the same time:



In this radio buttons group, 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.

PyQt 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. Here, you've studied only a small sample. However, that's enough to show you the power and flexibility of PyQt. In the next section, you'll learn how to lay out different widgets to build modern and fully functional GUIs for your applications.

Layout Managers

Now that you know about widgets and how they're used to build GUIs, you need to know how to arrange a set of widgets so that your GUI is both coherent and functional. In PyQt, you'll find a few techniques for laying out the widgets on a form or window. For instance, you can use the `.resize()` and `.move()` methods to give widgets absolute sizes and positions.

However, this technique can have some drawbacks. You'll have to:

- Do many manual calculations to determine the correct size and position of every widget
- Do extra calculations to respond to window resize events
- Redo most of your calculations when the window's layout changes in any way

Another technique involves using `.resizeEvent()` to calculate the widget's size and position dynamically. In this case, you'll have similar headaches as with the previous technique.

The most effective and recommended technique is to use PyQt's layout managers. They'll increase your productivity, mitigate the risk of errors, and improve your code's maintainability.

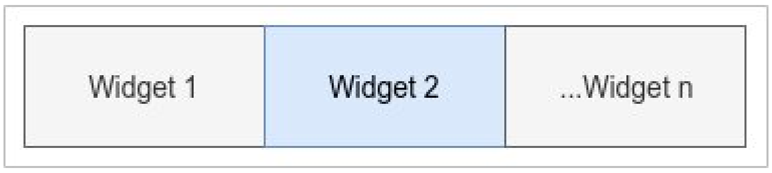
Layout managers are classes that allow you to size and position your widgets on the application's window or form. They automatically adapt to resize events and GUI changes, controlling the size and position of all their child widgets.

Note: If you develop internationalized applications, then you've probably seen translated text get cut off mid-sentence. This is likely to happen when the target natural language is more verbose than the original one. Layout managers can help you prevent this common issue by automatically adjusting the widget size to the available space. However, this feature can sometimes fail with particularly wordy natural languages.

PyQt provides four basic layout manager classes:

1. `QHBoxLayout`
2. `QVBoxLayout`
3. `QGridLayout`
4. `QFormLayout`

The first layout manager class, **`QHBoxLayout`**, arranges widgets horizontally from left to right, like with the hypothetical widgets in the following figure:



In the horizontal layout, the widgets will appear one next to the other, starting from the left. The code example below shows how to use `QHBoxLayout` to arrange three buttons horizontally:

Python

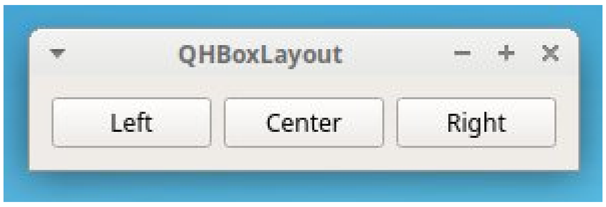
```python
# h_layout.py

"""Horizontal layout example."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
    QHBoxLayout,
    QPushButton,
    QWidget,
)

app = QApplication([])
window = QWidget()
```

```python
16 window.setWindowTitle("QHBoxLayout")
17
18 layout = QHBoxLayout()
19 layout.addWidget(QPushButton("Left"))
20 layout.addWidget(QPushButton("Center"))
21 layout.addWidget(QPushButton("Right"))
22 window.setLayout(layout)
23
24 window.show()
25 sys.exit(app.exec())
```

Here’s how this example creates a horizontal layout of buttons:

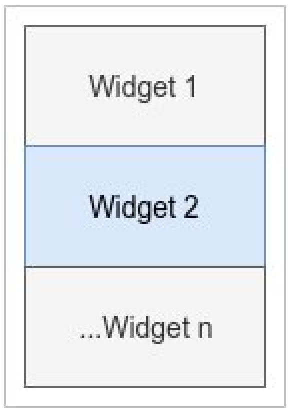
- **Line 18** creates a `QHBoxLayout` object called `layout`.
- **Lines 19 to 21** add three buttons to `layout` by calling the `.addWidget()` method.
- **Line 22** sets `layout` as your window’s layout with `.setLayout()`.

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



The above figure shows three buttons in a horizontal arrangement. 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, like in the following figure:



Each new widget will appear beneath the previous one. This layout allows you to to construct vertical layouts and organize your widgets from top to bottom on your GUI.

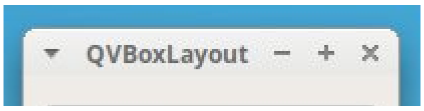
Here's how you can create a `QVBoxLayout` object containing three buttons:

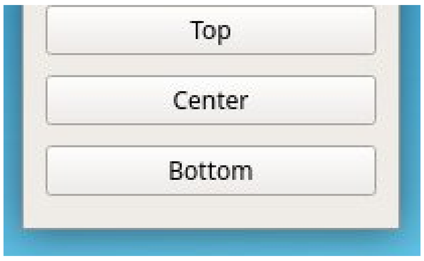
Python

```python
 1  # v_layout.py
 2
 3  """Vertical layout example."""
 4
 5  import sys
 6
 7  from PyQt6.QtWidgets import (
 8      QApplication,
 9      QPushButton,
10      QVBoxLayout,
11      QWidget,
12  )
13
14  app = QApplication([])
15  window = QWidget()
16  window.setWindowTitle("QVBoxLayout")
17
18  layout = QVBoxLayout()
19  layout.addWidget(QPushButton("Top"))
20  layout.addWidget(QPushButton("Center"))
21  layout.addWidget(QPushButton("Bottom"))
22  window.setLayout(layout)
23
24  window.show()
25  sys.exit(app.exec())
```

On line 18, you create an instance of `QVBoxLayout` called `layout`. In the next three lines, you add three buttons to `layout`. Finally, you use the `layout` object to arrange the widget in a vertical layout through the `.setLayout()` method on line 22.

When you run this sample application, you'll get a window that looks something like this:

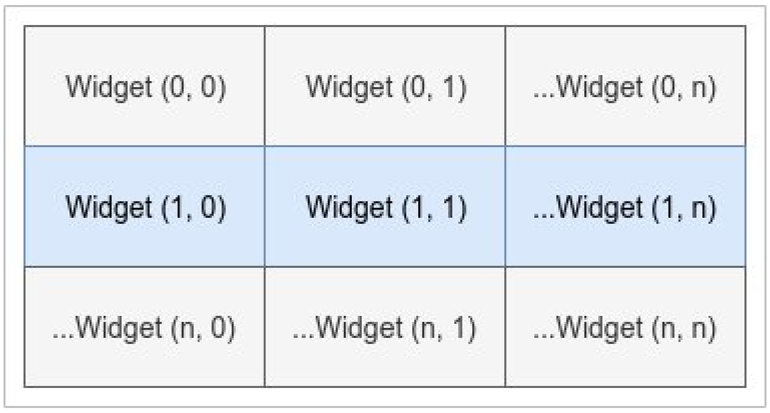




This figure shows three buttons in a vertical arrangement, one below the other. The buttons appear in the same order as you added them to your code, from top to bottom.

The third layout manager in your list is **`QGridLayout`**. This class arranges widgets in a grid of rows and columns. Every widget will have a relative position on the grid. You can define a widget's position with a pair of coordinates like `(row, column)`. Each coordinate must be an integer number. These pairs of coordinates define which cell on the grid a given widget will occupy.

The grid layout will look something like this:



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

Here's how to create a grid layout arrangement in your GUI:

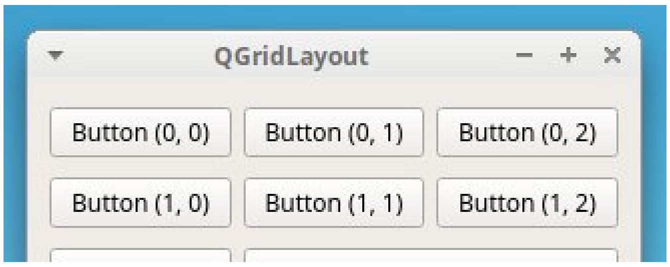
Python

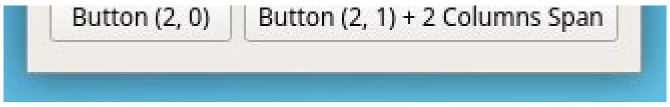
```python
# g_layout.py

"""Grid layout example."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
 9       QGridLayout,
10       QPushButton,
11       QWidget,
12   )
13
14   app = QApplication([])
15   window = QWidget()
16   window.setWindowTitle("QGridLayout")
17
18   layout = QGridLayout()
19   layout.addWidget(QPushButton("Button (0, 0)"), 0, 0)
20   layout.addWidget(QPushButton("Button (0, 1)"), 0, 1)
21   layout.addWidget(QPushButton("Button (0, 2)"), 0, 2)
22   layout.addWidget(QPushButton("Button (1, 0)"), 1, 0)
23   layout.addWidget(QPushButton("Button (1, 1)"), 1, 1)
24   layout.addWidget(QPushButton("Button (1, 2)"), 1, 2)
25   layout.addWidget(QPushButton("Button (2, 0)"), 2, 0)
26   layout.addWidget(
27       QPushButton("Button (2, 1) + 2 Columns Span"), 2, 1, 1,
28   )
29   window.setLayout(layout)
30
31   window.show()
32   sys.exit(app.exec())
```

In this example, you create an application that uses a `QGridLayout` object to organize its widgets on the screen. Note that, in this case, the second and third arguments that you pass to `.addWidget()` are integer numbers defining each widget's position on the grid.

On lines 26 to 28, you pass two more arguments to `.addWidget()`. These arguments are `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 in the example.

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

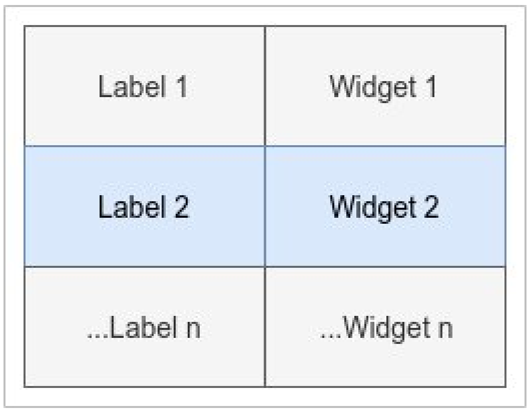




In this figure, you can see your widgets arranged in a grid of rows and columns. The last widget occupies two columns, as you specified on lines 26 to 28.

The last layout manager that you'll learn about is **`QFormLayout`**. This class 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, while the right column consists of input widgets. If you're developing a database application, then this kind of layout can be a useful tool that'll increase your productivity when creating input forms.

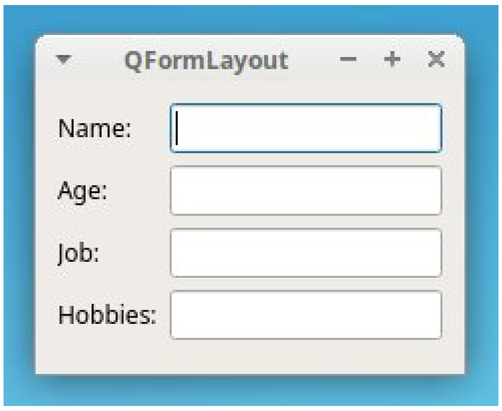
The following example shows how to create an application that uses a `QFormLayout` object to arrange its widgets:

Python

```python
# f_layout.py

"""Form layout example."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
 9      QFormLayout,
10      QLineEdit,
11      QWidget,
12  )
13
14  app = QApplication([])
15  window = QWidget()
16  window.setWindowTitle("QFormLayout")
17
18  layout = QFormLayout()
19  layout.addRow("Name:", QLineEdit())
20  layout.addRow("Age:", QLineEdit())
21  layout.addRow("Job:", QLineEdit())
22  layout.addRow("Hobbies:", QLineEdit())
23  window.setLayout(layout)
24
25  window.show()
26  sys.exit(app.exec())
```

Lines 18 to 23 do the hard work in this example. `QFormLayout` has a convenient method called `.addRow()`. You can use this method to add a two-widget row to the layout. The first argument to `.addRow()` should be a label or a string. Then, the second argument can be any widget that allows the user to enter or edit data. In this specific example, you've used line edits.

If you run this code, then you'll get a window that looks something like this:



The above figure shows a window 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 required information.

Dialogs

With PyQt, you can develop two types of GUI desktop applications. Depending on the class that 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 learn about 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 a stand-alone window that you can use as the main window for your application.

Note: Dialog windows are commonly used in main window–style applications for brief communication and interaction with the user.

When you use dialog windows to communicate with the user, those dialogs can be:

- **Modal:** Blocks input to any other visible windows in the same application. You can display a modal dialog by calling its `.exec()` method.
- **Modeless:** Operates independently from other windows in the same application. You can display a modeless dialog by using its `.show()` method.

Dialog windows can also provide a return value and have default buttons, such as `Ok` and `Cancel`.

A dialog is always an independent window. If a dialog has a `parent`, then it'll display centered on top of the parent widget. Dialogs with a parent will share the parent's task bar entry. If you don't set `parent` for a given dialog, then the dialog will get its own entry in the system's task bar.

Here's an example of how you'd use `QDialog` to develop a dialog-style application:

Python

```python
# dialog.py

"""Dialog-style application."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
    QDialog,
    QDialogButtonBox,
    QFormLayout,
    QLineEdit,
    QVBoxLayout,
)

class Window(QDialog):
    def __init__(self):
        super().__init__(parent=None)
        self.setWindowTitle("QDialog")
        dialogLayout = QVBoxLayout()
        formLayout = QFormLayout()
        formLayout.addRow("Name:", QLineEdit())
        formLayout.addRow("Age:", QLineEdit())
        formLayout.addRow("Job:", QLineEdit())
        formLayout.addRow("Hobbies:", QLineEdit())
        dialogLayout.addLayout(formLayout)
        buttons = QDialogButtonBox()
        buttons.setStandardButtons(
            QDialogButtonBox.StandardButton.Cancel
            | QDialogButtonBox.StandardButton.Ok
        )
        dialogLayout.addWidget(buttons)
        self.setLayout(dialogLayout)

```

```python
35 if __name__ == "__main__":
36     app = QApplication([])
37     window = Window()
38     window.show()
39     sys.exit(app.exec())
```

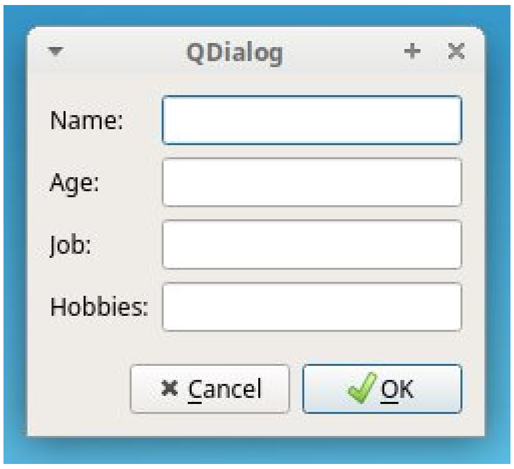
This application is a bit more elaborate. Here's what this code does:

- **Line 16** defines a `Window` class for the app's GUI by inheriting from `QDialog`.
- **Line 18** calls the parent class's `.__init__()` method using `super()`. This call allows you to properly initialize instances of this class. In this example, the `parent` argument is set to `None` because this dialog will be your main window.
- **Line 19** sets the window's title.
- **Line 20** assigns a `QVBoxLayout` object to `dialogLayout`.
- **Line 21** assigns a `QFormLayout` object to `formLayout`.
- **Lines 22 to 25** add widgets to `formLayout`.
- **Line 26** calls `.addLayout()` on `dialogLayout`. This call embeds the form layout into the global dialog layout.
- **Line 27** defines a button box, which provides a convenient space to display the dialog's buttons.
- **Lines 28 to 31** add two standard buttons, `Ok` and `Cancel`, to the dialog.
- **Line 32** adds the button box to the dialog by calling `.addWidget()`.

The `if __name__ == "__main__":` construct wraps up the app's main code. This kind of conditional statement is common in Python apps. It ensures that the indented code will only run if the containing file is executed as a program rather than imported as a module. For more about this construct, check out What Does if **name** == "**main**" Do in Python?.

Note: On line 26 in the above example, you'll note that layout managers can be nested inside one another. You can nest layouts by calling `.addLayout()` on the container layout with the nested layout as an argument.

The above code example will show a window that looks something like this:



This figure shows the GUI that you've created using a `QFormLayout` object to arrange the widgets and a `QVBoxLayout` layout for the application's global layout.

Main Windows

Most of the time, your GUI applications will be **main window–style** apps. This means that they'll have a menu bar, some toolbars, a status bar, and a central widget that'll be the GUI's main element. It's also common for your apps to have several dialogs to accomplish secondary actions that depend on a user's input.

You'll inherit from `QMainWindow` to develop main window–style applications. An instance of a class that derives from `QMainWindow` is considered the app's **main window** and should be unique.

`QMainWindow` provides a framework for building your application's GUI quickly. This class has its own built-in layout, which accepts the following graphical components:

Component	Position on Window	Description
One menu bar	Top	Holds the application’s main menu
One or more toolbars	Sides	Hold tool buttons and other widgets, such as `QComboBox`, `QSpinBox`, and more
One central widget	Center	Holds the window’s central widget, which can be of any type, including a composite widget
One or more dock widgets	Around the central widget	Are small, movable, and hidable windows
One status bar	Bottom	Holds the app’s status bar, which shows status information

You can’t create a main window without a central widget. You need 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 window’s central widget with the `.setCentralWidget()` method. 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

```python
# main_window.py

"""Main window-style application."""

```

```
 5  import sys
 6
 7  from PyQt6.QtWidgets import (
 8      QApplication,
 9      QLabel,
10      QMainWindow,
11      QStatusBar,
12      QToolBar,
13  )
14
15  class Window(QMainWindow):
16      def __init__(self):
17          super().__init__(parent=None)
18          self.setWindowTitle("QMainWindow")
19          self.setCentralWidget(QLabel("I'm the Central Widge
20          self._createMenu()
21          self._createToolBar()
22          self._createStatusBar()
23
24      def _createMenu(self):
25          menu = self.menuBar().addMenu("&Menu")
26          menu.addAction("&Exit", self.close)
27
28      def _createToolBar(self):
29          tools = QToolBar()
30          tools.addAction("Exit", self.close)
31          self.addToolBar(tools)
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([])
40      window = Window()
41      window.show()
42      sys.exit(app.exec())
```

Here's how this code works:

- **Line 15** creates a class, `Window`, that inherits from `QMainWindow`.
- **Line 16** defines the class initializer.
- **Line 17** calls the base class's initializer. Again, the `parent` argument is set to `None` because this is your app's main

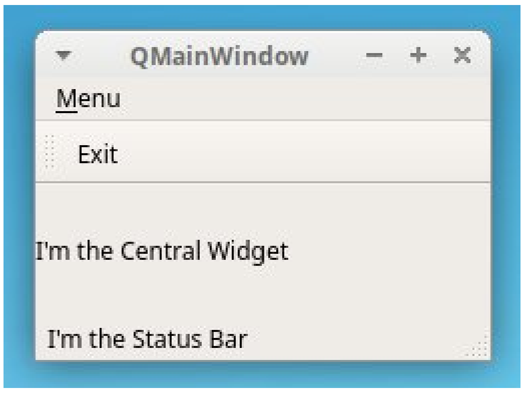
window, so it must not have a parent.

- **Line 18** sets the window's title.
- **Line 19** sets a `QLabel` as the window's central widget.
- **Lines 20 to 22** call non-public methods to create different GUI elements:
    - **Lines 24 to 26** create the main menubar with a drop-down menu called *Menu*. This menu will have a menu option to exit the app.
    - **Lines 28 to 31** create the toolbar, which will have a toolbar button to exit the app.
    - **Lines 33 to 36** create the app's status bar.

When you implement GUI components using their own methods, like you did with the menu bar, toolbar, and status bar in this example, you're making your code more readable and more maintainable.

Note: If you're running this example on macOS, then you may have issues with the app's main menu. macOS hides certain menu options, like *Exit*. Remember that macOS shows the *Exit* or *Quit* option under the app's entry on the top of the screen.

When you run the above sample application, you'll get a window like the following:



As you can confirm, your main window–style application has the following components:

- One main menu generically called *Menu*

- One toolbar with an *Exit* tool button
- One central widget consisting of a `QLabel` object with a text message
- One status bar at the window's bottom

That's it! You've learned how to build a main window–style application with Python and PyQt. Up to this point, you've learned about some of the more important graphical components in PyQt's set of widgets. In the next few sections, you'll study other important concepts related to building GUI applications with PyQt.

Applications

`QApplication` is the most foundational class that you'll use when developing PyQt GUI applications. This class is the core component of any PyQt application. It manages the application's control flow as well as its main settings.

In PyQt, any instance of `QApplication` is an **application**. Every PyQt GUI application must have one `QApplication` instance. Some of the responsibilities of this class include:

- Handling the app's **initialization** and **finalization**
- Providing the **event loop** and event handling
- Handling most 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`. So, 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. In the following section, you’ll take a closer look at what the event loop is and how it works.

Event Loops

GUI applications are **event-driven**. This means that functions and methods are called 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 commonly known as **events**.

Events are handled by an **event loop**, also known as a **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.

All GUI applications have an event loop. When an event happens, then the loop checks if it’s a **terminate event.** In that case, the loop finishes, and the application exits. Otherwise, the event is sent to the application’s event queue for further processing, and the loop iterates again. In PyQt6, you can run the app’s event loop by calling `.exec()` on the `QApplication` object.

For an event to trigger an action, you need to connect the event with the action that you want to execute. In PyQt, you can establish that connection with the signals and slots mechanism, which you’ll explore in the next section.

Signals and Slots

PyQt widgets act as **event-catchers**. This means that every widget can catch specific events, like mouse clicks, keypresses, and so on. In response to these events, a widget emits 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'll perform an action whenever its associated signal is emitted. You can use any Python **callable** 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. Some of the most relevant features of signals and slots include the following:

- 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 and a slot:

Python

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

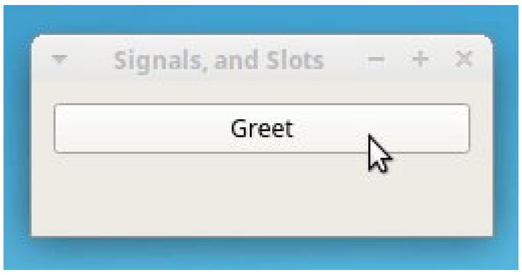
This will connect `slot_function` to `widget.signal`. From now on, whenever `.signal` is emitted, `slot_function()` will be called.

The code below shows how to use the signals and slots mechanism in a PyQt application:

Python

```python
# signals_slots.py

"""Signals and slots example."""

import sys

from PyQt6.QtWidgets import (
    QApplication,
    QLabel,
    QPushButton,
    QVBoxLayout,
    QWidget,
13 )
14
15 def greet():
16     if msgLabel.text():
17         msgLabel.setText("")
18     else:
19         msgLabel.setText("Hello, World!")
20
21 app = QApplication([])
22 window = QWidget()
23 window.setWindowTitle("Signals and slots")
24 layout = QVBoxLayout()
25
26 button = QPushButton("Greet")
27 button.clicked.connect(greet)
28
29 layout.addWidget(button)
30 msgLabel = QLabel("")
31 layout.addWidget(msgLabel)
32 window.setLayout(layout)
33 window.show()
34 sys.exit(app.exec())
```

On line 15, you create `greet()`, which you'll use as a slot. Then in line 27, you connect the button's `.clicked` signal to `greeting()`. This way, whenever the user clicks the *Greet* button, the `greet()` slot is called and the label object's text alternates between `Hello, World!` and an empty string:



When you click the *Greet* button, the `Hello, World!` message appears and disappears on your application's main window.

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

If your slot function needs to receive extra arguments, then you can pass them using `functools.partial()`. For example, you can modify `greet()` to take an argument, like in the following code:

Python

```python
# signals_slots.py
# ...

def greet(name):
    if msg.text():
        msg.setText("")
    else:
        msg.setText(f"Hello, {name}")

# ...
```

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

Python

```python
# signals_slots.py

"""Signals and slots example."""

import sys
from functools import partial

# ...

button = QPushButton("Greet")
button.clicked.connect(partial(greeting, "World!"))

# ...
```

For this code to work, you need to import `partial()` from `functools` first. The call to `partial()` returns a function object that behaves similarly to `greet()` when called with `name="World!"`. Now, when the user clicks on the button, the message `Hello, World!` will appear in the label just like before.

Note: You can also use a `lambda` function to connect a signal to a slot that requires extra arguments. As an 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 PyQt GUI applications. This mechanism will allow you to turn user events into concrete actions. You can dive deeper into signals and slots by checking out the PyQt6 documentation on the topic.

Now you know the basics of several important concepts of PyQt. 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.

Creating a Calculator App With Python and PyQt

In this section, you'll develop a calculator GUI app using the Model-View-Controller (MVC) design pattern. This pattern has three layers of code, with each one having different roles:

1. The **model** takes care of your app's business logic. It contains the core functionality and data. In your calculator app, the model will handle the input values and the calculations.

2. The **view** implements your app's GUI. It hosts all the widgets that the end user would need to interact with the application. The view also receives a user's actions and events. For your example, the view will be the calculator window on your screen.

3. The **controller** connects the model and the view to make the application work. Users' 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. In your calculator app, the controller will receive the target math expressions from the GUI, ask the model to perform calculations, and update the GUI with the result.

Here's a step-by-step description of how your GUI calculator app will work:

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's query, performs the **required computations**, and returns the **result**.
5. The controller receives the model's response and **updates the view** accordingly.
6. The user finally sees the requested **result on the view**.

You'll use this MVC design to build your calculator app with Python and PyQt.

Creating the Skeleton for Your PyQt Calculator App

To kick things off, you'll start by implementing a minimal skeleton for your application in a file called `pycalc.py`. You can get this file and the rest of the source code for your calculator app by clicking the link below:

Download Code: Click here to download the code that 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. Open the file in your favorite code editor or IDE and type the following code:

Python

```python
 1  # pycalc.py
 2
 3  """PyCalc is a simple calculator built with Python and PyQt
 4
 5  import sys
 6
 7  from PyQt6.QtWidgets import QApplication, QMainWindow, QWic
 8
 9  WINDOW_SIZE = 235
10
11  class PyCalcWindow(QMainWindow):
12      """PyCalc's main window (GUI or view)."""
13
14      def __init__(self):
15          super().__init__()
16          self.setWindowTitle("PyCalc")
17          self.setFixedSize(WINDOW_SIZE, WINDOW_SIZE)
18          centralWidget = QWidget(self)
19          self.setCentralWidget(centralWidget)
20
21  def main():
22      """PyCalc's main function."""
23      pycalcApp = QApplication([])
24      pycalcWindow = PyCalcWindow()
25      pycalcWindow.show()
26      sys.exit(pycalcApp.exec())
27
28  if __name__ == "__main__":
29      main()
```

This script implements all the boilerplate code that you'll need to run a basic GUI application. You'll use this skeleton to build your calculator app.

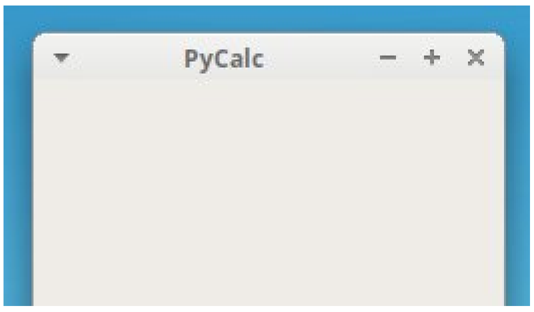
Here's how this code works:

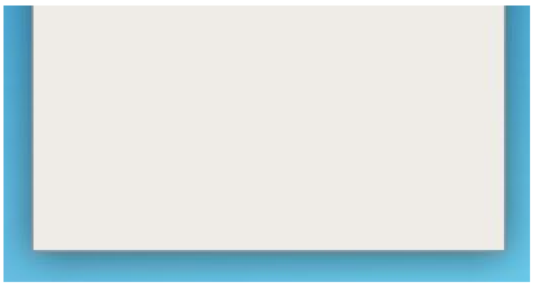
- **Line 5** imports `sys`. This module provides the `exit()` function, which you'll use to cleanly terminate the app.
- **Line 7** imports the required classes from `PyQt6.QtWidgets`.
- **Line 9** creates a Python constant to hold a fixed window size in

pixels for your calculator app.

- **Line 11** creates the `PyCalcWindow` class to provide the app's GUI. Note that this class inherits from `QMainWindow`.
- **Line 14** defines the class initializer.
- **Line 15** calls `.__init__()` on the super class for initialization purposes.
- **Line 16** sets the window's title to `"PyCalc"`.
- **Line 17** uses `.setFixedSize()` to give the window a fixed size. This ensures that the user won't be able to resize the window during the app's execution.
- **Lines 18 and 19** create a `QWidget` object and set it as the window's central widget. This object will be the parent of all the required GUI components in your calculator app.
- **Line 21** defines your calculator's main function. Having a `main()` function like this is a best practice in Python. This function provides the application's entry point. Inside `main()`, your program does the following:
    - **Line 23** creates a `QApplication` object named `pycalcApp`.
    - **Line 24** creates an instance of the app's window, `pycalcWindow`.
    - **Line 25** shows the GUI by calling `.show()` on the window object.
    - **Line 26** runs the application's event loop with `.exec()`.

Finally, line 29 calls `main()` to execute your calculator app. When you run the above script, the following window appears on your screen:





That's it! You've successfully built a fully functional app skeleton for your GUI calculator app. Now you're ready to continue building the project.

Completing the App's View

The GUI that you have at this point doesn't really look like a calculator. You need to finish this GUI by adding a display to show the target math operation and a keyboard of buttons representing numbers and basic math operators. You'll also add buttons representing other required symbols and actions, like clearing the display.

First, you need to update your imports like in the code below:

Python

```python
# pycalc.py

import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QGridLayout,
    QLineEdit,
    QMainWindow,
    QPushButton,
    QVBoxLayout,
    QWidget,
)

# ...
```

You'll use a `QVBoxLayout` layout manager for the calculator's global layout. To arrange the buttons, you'll use a `QGridLayout` object. The

QLineEdit class will work as the calculator's display and QPushButton will provide the required buttons.

Now you can update the initializer for PyCalcWindow:

Python

```python
# pycalc.py
# ...

class PyCalcWindow(QMainWindow):
    """PyCalc's main window (GUI or view)."""

    def __init__(self):
        super().__init__()
        self.setWindowTitle("PyCalc")
        self.setFixedSize(WINDOW_SIZE, WINDOW_SIZE)
        self.generalLayout = QVBoxLayout()
        centralWidget = QWidget(self)
        centralWidget.setLayout(self.generalLayout)
        self.setCentralWidget(centralWidget)
        self._createDisplay()
        self._createButtons()

# ...
```

You've added the highlighted lines of code. You'll use .generalLayout as the app's general layout. In this layout, you'll place the display at the top and the keyboard buttons in a grid layout at the bottom.

The calls to ._createDisplay() and ._createButtons() won't work at this point, because you haven't implemented those methods yet. To fix this issue, you'll start by coding ._createDisplay().

Get back to your code editor and update pycalc.py like in the following code:

Python

```python
# pycalc.py
# ...

WINDOW_SIZE = 235
DISPLAY_HEIGHT = 35
```

```
class PyCalcWindow(QMainWindow):
    # ...

    def _createDisplay(self):
        self.display = QLineEdit()
        self.display.setFixedHeight(DISPLAY_HEIGHT)
        self.display.setAlignment(Qt.AlignmentFlag.AlignRight)
        self.display.setReadOnly(True)
        self.generalLayout.addWidget(self.display)

# ...
```

In this code snippet, you first define a new constant to hold the display height in pixels. Then you define `._createDisplay()` inside `PyCalcWindow`.

To create the calculator's display, you use a `QLineEdit` widget. Then you set a fixed height of thirty-five pixels for your display using the `DISPLAY_HEIGHT` constant. The display will have its text left-aligned. Finally, the display will be read-only to prevent direct editing by the user. The last line of code adds the display to the calculator's general layout.

Next up, you'll implement the `._createButtons()` method to create the required buttons for your calculator's keyboard. These buttons will live in a grid layout, so you need a way to represent their coordinates on the grid. Each coordinate pair will consist of a row and a column. To represent a coordinate pair, you'll use a list of lists. Each nested list will represent a row.

Now go ahead and update the `pycalc.py` file with the following code:

Python

```python
# pycalc.py
# ...

WINDOW_SIZE = 235
DISPLAY_HEIGHT = 35
BUTTON_SIZE = 40
```

```
# ...
```

In this piece of code, you define a new constant called `BUTTON_SIZE`. You'll use this constant to provide the size of your calculator's buttons. In this specific example, all the buttons will have a square shape with forty pixels per side.

With this initial setup, you can code the `._createButtons()` method. You'll use a list of lists to hold the keys or buttons and their position on the calculator keyboard. A `QGridLayout` will allow you to arrange the buttons on the calculator's window:

Python

```python
# pycalc.py
# ...

class PyCalcWindow(QMainWindow):
    # ...

    def _createButtons(self):
        self.buttonMap = {}
        buttonsLayout = QGridLayout()
        keyBoard = [
            ["7", "8", "9", "/", "C"],
            ["4", "5", "6", "*", "("],
            ["1", "2", "3", "-", ")"],
            ["0", "00", ".", "+", "="],
        ]

        for row, keys in enumerate(keyBoard):
            for col, key in enumerate(keys):
                self.buttonMap[key] = QPushButton(key)
                self.buttonMap[key].setFixedSize(BUTTON_SIZE, B
                buttonsLayout.addWidget(self.buttonMap[key], ro

        self.generalLayout.addLayout(buttonsLayout)

# ...
```

You first create the empty dictionary `self.buttonMap` to hold the calculator buttons. Then, you create a list of lists to store the key labels. Each row or nested list will represent a row in the grid layout, while the index of each key label will represent the corresponding column on the layout.

Then you define two for loops. The outer loop iterates over the rows and the inner loop iterates over the columns. Inside the inner loop, you create the buttons and add them to both self.buttonMap and buttonsLayout. Every button will have a fixed size of 40x40 pixels, which you set with .setFixedSize() and the BUTTON_SIZE constant.

Finally, you embed the grid layout into the calculator's general layout by calling .addLayout() on the .generalLayout object.

Note: When it comes to widget size, you'll rarely find measurement units in the PyQt documentation. The measurement unit is assumed to be **pixels**, except when you're working with QPrinter class, which uses **points**.

Now your calculator's GUI will show the display and the buttons gracefully. However, you have no way to update the information shown on the display. You'll fix this by adding a few extra methods to PyCalcWindow:

Method	Description
.setDisplayText()	Sets and updates the display's text
.displayText()	Gets the current display's text
.clearDisplay()	Clears the display's text

These methods will provides the GUI's public interface and complete the view class for your Python calculator app.

Here's a possible implementation:

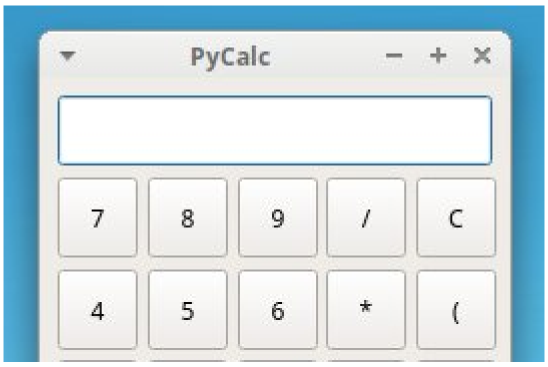
Python

```python
# pycalc.py
# ...

class PyCalcWindow(QMainWindow):
    # ...

    def setDisplayText(self, text):
        """Set the display's text."""
        self.display.setText(text)
        self.display.setFocus()

    def displayText(self):
        """Get the display's text."""
        return self.display.text()

    def clearDisplay(self):
        """Clear the display."""
        self.setDisplayText("")

# ...
```

Here's a breakdown of what each method does:

- **`.setDisplayText()`** uses `.setText()` to set and update the display's text. It also uses `.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 the equal sign (`=`) on the calculator's keyboard, the app 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 that the user can introduce a new math expression. This method will be triggered every time the user presses the *C* button on the calculator's board.

Now your calculator's GUI is ready for use! When you run the application, you'll get a window like the following:



You’ve completed the calculator’s GUI, which looks pretty sleek! However, if you try to do some calculations, then the calculator won’t respond as expected. That’s because you haven’t implemented the model and the controller components. In the next section, you’ll write the calculator’s model.

Implementing the Calculator’s Model

In the MVC pattern, the model is the layer of code that takes care of the business logic. In your calculator app, the business logic is all about basic math calculations. So, your model will evaluate the math expressions that your users introduced in the calculator’s GUI.

The calculator’s model also needs to handle errors. To this end, you’ll define the following global constant:

Python

```python
# pycalc.py
# ...

ERROR_MSG = "ERROR"
WINDOW_SIZE = 235
# ...
```

This `ERROR_MSG` constant is the message that the user will see on the calculator’s display if they introduce an invalid math expression.

With the above change, you’re ready to code your app’s model, which will be a single function in this example:

Python

```python
# pycalc.py
# ...

class PyCalcWindow(QMainWindow):
    # ...

def evaluateExpression(expression):
    """Evaluate an expression (Model)."""
    try:
        result = str(eval(expression, {}, {}))
    except Exception:
        result = ERROR_MSG
    return result

# ...
```

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

- The `try` … `except` block doesn't catch a specific exception, so it's using a practice that's discouraged in Python.
- The function uses `eval()`, which can lead to some serious security issues.

You're free to rework the function to make it more reliable and secure. In this tutorial, you'll use the function as is to keep the focus on implementing the GUI.

Creating the Controller Class for Your Calculator

In this section, you're going to code the calculator's controller class. This class will connect the view to the model that you just coded. You'll use the controller class to make the calculator perform actions in response to user events.

Your controller class needs to perform three main tasks:

3. Access the GUI's public interface.
4. Handle the creation of math expressions.
5. Connect all the buttons' `.clicked` signals with the appropriate slots.

To perform all these actions, you'll code a new `PyCalc` class in a moment. Go ahead and update `pycalc.py` with the following code:

```python
# pytcalc.py

import sys
from functools import partial
# ...

def evaluateExpression(expression):
    # ...

class PyCalc:
    """PyCalc's controller class."""

    def __init__(self, model, view):
        self._evaluate = model
        self._view = view
        self._connectSignalsAndSlots()

    def _calculateResult(self):
        result = self._evaluate(expression=self._view.displayTe
        self._view.setDisplayText(result)

    def _buildExpression(self, subExpression):
        if self._view.displayText() == ERROR_MSG:
            self._view.clearDisplay()
        expression = self._view.displayText() + subExpression
        self._view.setDisplayText(expression)

    def _connectSignalsAndSlots(self):
        for keySymbol, button in self._view.buttonMap.items():
            if keySymbol not in {"=", "C"}:
                button.clicked.connect(
                    partial(self._buildExpression, keySymbol)
                )
        self._view.buttonMap["="].clicked.connect(self._calcula
        self._view.display.returnPressed.connect(self._calculat
        self._view.buttonMap["C"].clicked.connect(self._view.cl
# ...
```

At the top of `pycalc.py`, you import `partial()` from `functools`. You'll use this function to connect signals with methods that need to take extra arguments.

Inside `PyCalc`, you define the class initializer, which takes two arguments: the app's model and its view. Then you store these arguments in appropriate instance attributes. Finally, you call `._connectSignalsAndSlots()` to make all the required connections of signals and slots.

In `._calculateResult()`, you use `._evaluate()` to evaluate the math expression that the user has just typed into the calculator's display. Then you call `.setDisplayText()` on the calculator's view to update the display text with the computation result.

As its name suggests, the `._buildExpression()` method takes care of building the target math expression. To do this, the method concatenates the initial display value with every new value that the user enters on the calculator's keyboard.

Finally, the `._connectSignalsAndSlots()` method connects all the buttons' `.clicked` signals with the appropriate slots method in the controller class.

That's it! Your controller class is ready. However, for all this code to work as a real calculator, you need to update the app's `main()` function like in the code below:

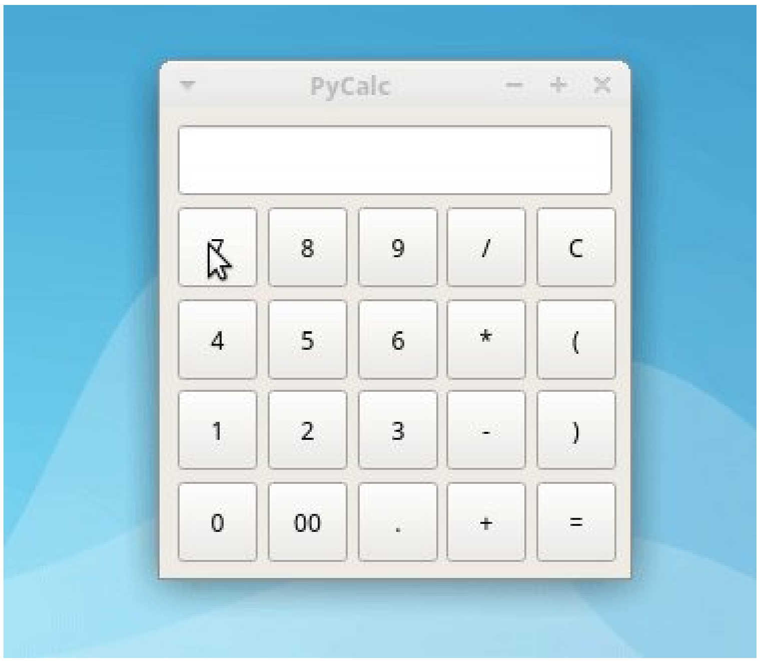
Python

```python
# pytcalc.py
# ...

def main():
    """PyCalc's main function."""
    pycalcApp = QApplication([])
    pycalcWindow = PyCalcWindow()
    pycalcWindow.show()
    PyCalc(model=evaluateExpression, view=pycalcWindow)
    sys.exit(pycalcApp.exec())
```

This piece of code creates a new instance of `PyCalc`. The `model` argument to the `PyCalc` class constructor holds a reference to the `evaluateExpression()` function, while the `view` argument holds a reference to the `pycalcWindow` object, which provides the app's GUI. Now your PyQt calculator application is ready to run.

Running the Calculator

Now that you've finished writing your calculator app with Python and PyQt, it's time for a live test! If you run the application from your command line, then you'll get something like this:

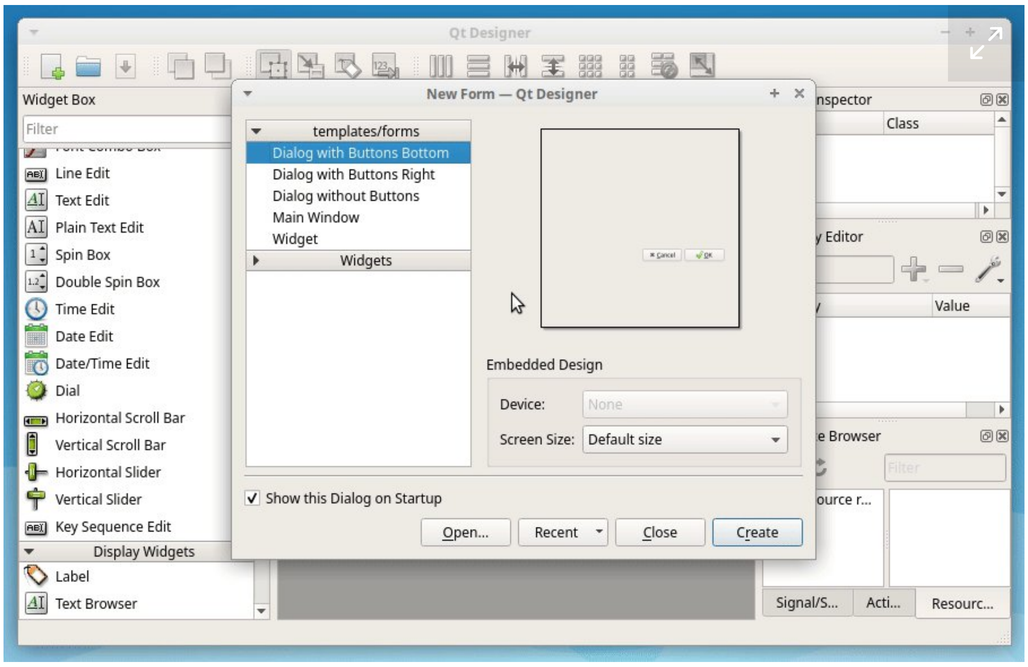


To use PyCalc, enter a valid math expression with your mouse. Then, press `Enter ↩` or click the equal sign (`=`) button to compute and show the expression result on the calculator's display. That's it! You've developed your first fully functional GUI desktop application with Python and PyQt!

Additional Tools

PyQt6 offers a useful set of additional tools that can help you build solid, modern, and full-featured GUI applications. Some of the most remarkable tools related to PyQt include Qt Designer and the internationalization kit.

Qt Designer allows you to design and build graphical user interfaces using a drag-and-drop interface. You can use this tool to design widgets, dialogs, and main windows by using on-screen forms and a drag-and-drop mechanism. The following animation shows some of Qt Designer's features:



Device: None

Screen Size: Default size

✓ Show this Dialog on Startup

Qt Designer uses XML `.ui` files to store your GUI designs. PyQt includes a module called uic to help with `.ui` files. You can also convert the `.ui` file content into Python code with a command-line tool called `pyuic6`.

Note: To dive deeper into Qt Designer and better understand how to use this tool to create graphical user interfaces, check out Qt Designer and Python: Build Your GUI Applications Faster.

PyQt6 also provides a comprehensive set of tools for the **internationalization** of apps into local languages. The `pylupdate6` command-line tool creates and updates translation (`.ts`) files, which can contain translations of interface strings. If you prefer GUI tools, then you can use Qt Linguist to create and update `.ts` files with translations of interface strings.

Conclusion

Graphical user interface (GUI) 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 one of the most popular and solid libraries for GUI application development in Python. Now you know how to effectively use PyQt to build modern GUI applications.

In this tutorial, you've learned how to:

- Build **graphical user interfaces** with Python and PyQt
- Connect the **user's events** with the **app's logic**
- Organize a PyQt application using a proper **project layout**
- Create a **real-world GUI application** with PyQt

Now you can use your Python and PyQt knowledge to give life to your own desktop GUI applications. Isn't that cool?

You can get the source code of your calculator app project and all its associated resources by clicking the link below:

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

Further Reading

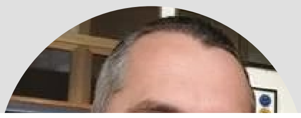
To dive deeper into PyQt and its ecosystem, check out some of the following resources:

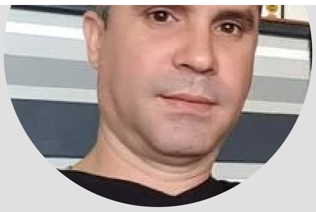
- PyQt6’s documentation
- PyQt5’s documentation
- PyQt4’s documentation
- Qt v6’s documentation
- The PyQt wiki
- The Rapid GUI Programming with Python and Qt book
- The Qt Designer manual
- Qt for Python’s documentation

Although the PyQt6 Documentation is the first resource listed here, some important parts of it are still missing or incomplete. Fortunately, you can use the Qt documentation to fill in the blanks.

Mark as Completed

About **Leodanis Pozo Ramos**

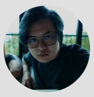
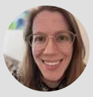




Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» 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		Kate		Mike

What Do You Think?

Rate this article:

Tweet Share in Share Email

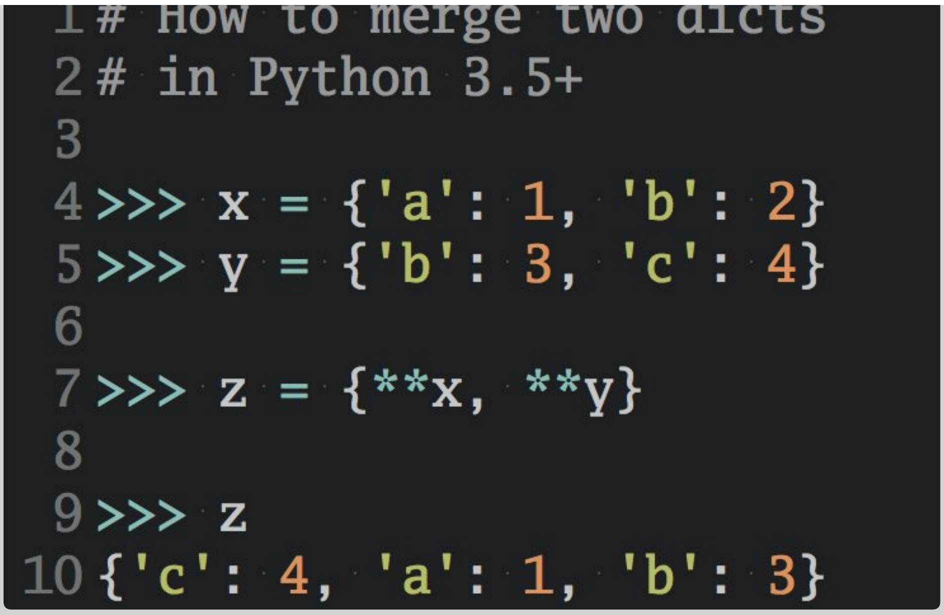
What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: gui intermediate projects



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

Mark as Completed

Tweet Share Email