Python Desktop Application Development

by Bo Milanovich

Start Course

Bookmark Add to Channel Download Course

Python Desktop Application Development

Introduction

Hello, Everyone. My name is Bo Milanovich and welcome to the Python Desktop Application Development course. In this course, I'm going to teach you how to write cross platform desktop applications, using Python and its friend called Qt, and yes, I did say cute. In fact, my aim is to show you how simple it can be to write desktop applications and run on all major operating systems, and this is something not a lot of languages can brag about. But, before we begin, I would like to point out some Prerequisites. First, you should have general programming skills. In other words, I'm not going to explain what a variable is or what happens when you call a function. And secondly, I would like you to have a basic understanding of Python programming language, as well. In other words, you should know the Python syntax, how to declare a function, how to create an instance of a class, etc. That being said, keep in mind that this course is still aimed at novice programmers, but also programmers who do not have any experience writing Python Desktop Applications, or rather, writing desktop applications in any programming language. In fact, one of my goals for this course is to show you the basic work flow, the design, if you will, that takes place while working on a desktop application in almost any programming language.

This way, you'll be able to apply the knowledge gained from this course on other programming languages, as well. By the end of this course, I would expect that you'll be able to understand the principles of desktop application programming, and also, write your own desktop applications in Python, with the help of Qt. But, why Python? What is this Qt? Well, let's take a look at that in our next video.

Why Python?

This video is curiously entitled, Why Python? Normally, when I tell people that I use Python to write desktop applications, they ask me why, as in, why not just use C# or maybe Visual Basic? And, my answer is always the same, well, why not use Python? But, let's look at some of the specific reasons why I pick Python over other programming languages to write my desktop applications in. First off, Python is Cross-platform. What this means is that the code you write once can be run on any major operating system, be it Windows, Mac OS, or Linux. Secondly, it's all so very powerful. I've gotten into endless debates with other programmers about this. For some reason, most of them think that Python is nothing but a scripting language. A language is when you want to perform a certain task, such as move a large number of files or rename multiple folders at once. In fact, it came to the point that I would show them the application I wrote, and they would not believe me that I wrote it using Python, so that part about Python being used only as a scripting language, simply not true. While, of course, Python can be used as a scripting language, did you know that YouTube was written in Python? Large parts of other Google websites are also written in Python, and, after this course, you will see that you can even write desktop applications in Python. This is one of the main reasons why I pick Python over other languages. Another advantage of Python is that it is relatively easy to learn and fairly simple to use. In fact, you'll be happy to know that even our desktop applications will be very simple to write, in a true Pythonic way. Finally, and this is more of a point to note, after spending hours thinking about this, I have decided to use Python 3. 4 in this course. Trust me, I'm more than aware of the huge debate between Python 2 and Python 3, use cases in the Python community today, but I believe that Python 3 is the way to go, and all the libraries that we'll be using in this course are Python 3 compatible. I hope that my choice of Python version will make a make you a happy Python, developer, that is. Now, what do I mean when I say Qt? No, I don't mean this cute puppy, unfortunately, but, rather, I mean Qt, a framework that we'll be using to help us write our desktop application. It, in fact, is pronounced cute. Whether that's intentional or not, I'm going to let you decide. However, I would like to note that their slogan perfectly describes one of the main reasons why I prefer Qt over some other Gui framework, such as GIMP Toolkit, or otherwise

known as GTK. It says code less, create more, and deploy everywhere, but, again, let's look at some of the specific reasons, as well. Once again, Qt is Cross-platform. It is also very powerful. For example, it's not just used for drawing Gui elements on the screen. It can run database queries, provide network interfaces, even has its own implementation of the web kit rendering engine. These are all, what I kind of call them, micro-libraries within this framework, but, in this course, we'll be focusing on just two--the Qt core library and the Qt Gui library. Qt also calls native widget toolkits on all platforms*. Gees, what did I just say? Well, essentially, this means that Qt application on Windows will look like it's designed specifically for Windows. On Mac OS, it's going to look like an application design for Mac OS, and on Linux, it's going to look like an application design for whatever desktop environment you're using. However, note that Qt will attempt to use native toolkits on all platforms, but this is not always guaranteed, hence the asterisk at the end. One thing I should also mention is that we'll actually be using a Python binding for Qt. There are several Python bindings available, but the most notable ones are PyQt and PySide. Now, let me explain the concept of binding, a little bit. Qt was written in C++ programming language, and it was written for C++. Since these are all what we call bindings, even though we'll write code purely in Python, internally, PyQt will still execute its functions in C++. Should you care about this? No, not really. There is no reason for you to, at least, not at this point. Finally, for the purpose of this course, I have decided to use PyQt 4. There is PyQt 5 available that implements Qt 5, but Qt 5 has made some changes that slightly change the concept of Guid programming with Qt. One more thing I would like to point out, is that PyQt and PySide are very similar. In fact, they're so similar that you can often swap the import statements and change the import PyQt to import PySide. Their only major difference is the license. While they're both absolutely free to use, if you want to use PyQt for commercial products, you would have to pay the license fee. With PySide, this is not the case, and you can use PySide in whatever application you want. And, finally, I should mention at the end of this video is that you're free to use Python 2. 7 or PySide, or even PyQt 5, but note that you will have to do the code translations on the fly.

Tools

Before we begin diving into the code, we want to get the tools and libraries that we need that will make our lives as programmers a lot easier. First of all, I'm assuming that you already have Python installed, but just to be on the safe side, I'm going to show you what version of Python I'll be using in this course. In a search for Python in Google, and that's obviously going to show python. org as a first result, and, if I hover over this download's menu bar, I'm offered two versions of Python, 3. 4. 1 and 2. 7. 8. Now as mentioned in the previous video, I want the 3. 4. 1. Obviously, since I'm on

Windows, it's offering me a Windows version of it. If you are using Linux or Mac OS, just make sure to get the 3. 4. 1. Also, a very important note, is that you need to be using the 32-bit version of Python, so make sure the Python version you have installed for this course is 32-bit Python 3. 4. 1. And secondly, and probably more importantly for this course, we need to dial up PyQt framework or library, if you will. Let me Google PyQt. The first result that pops out is the Riverbank software. This is the company behind PyQt. If I hover over this software, go to PyQt, and select PyQt4 download, it's going to take me to the PyQt4 download page. Now, I'm using Windows and anyone else who was using also Windows, you probably want to download one of the Binary Packages available. These are installers for Windows. If you're using a different operating system, as you can see here, there's a Linux and MacOS source available, as well. One important note is that we need to download the proper version of PyQt4. As you can see, there are multiple versions right here, but remember, we're using a 32-bit of Python 3. 4, and we want PyQt4 that's based off Qt4, so this is exactly the link we need. Once you have that installed, we're going to actually be using an IDE throughout this course. This IDE is cross platform, as well, so you can install it in any operating system. The name of the IDE is PyCharm. Let me Google pycharm. PyCharm is an IDE made by the JetBrains, and this is the same company that makes IntelliJ Idea, a very popular IDE when it comes to Java development, and, if you're a web developer, you may have also heard of PhpStorm or WebStorm. They are also made by the JetBrains company. Now, if you go to the PyCharm website and hit the Get PyCharm Now, you're in for a good surprise. There is a Community Edition that is free to download and free to use, that we'll be using throughout this course.

Basic Code Structure

Basic Code Structure

Welcome back to Python Desktop Application Development course. My name is Bo Milanovich. In this module, we will look at some of the basics of Qt and how it integrates with Python. In our last video, in the previous module, I promised that we'll dig straight into the code, and I intend on keeping that promise. In this video, I'm going to talk about setting up a basic structure of our code, a code that must be present in every desktop application, at least when we're writing in Python and Qt. I've started off with PyCharm here, our IDE for the course, and with it, when you first start it, you greet it with the screen that looks like this. One of the first things you should do

is make sure that you have the correct interpreter selected in the PyCharm settings. To do that, you just hit Configure here, its Settings, and find the Project Interpreter setting. Make sure that you have 3. 4. 1 selected. PyCharm should automatically recognize all the interpreters you have installed on your computer. For example, in my case, that's 2. 7. 8 and 3. 4. 1. If for some reason it does not detect one, you can just simply add one by Clicking the Setting button and taking you through a kind of a add an interpreter wizard. Once you're done with that, you can hit OK, and don't worry about the fact that PyQt does not appear here. This is just for packages that were installed, using the pip. Now, I'm going to Create a New Project, and I'm going to name it Pluralsight. Also, make sure that you have the correct interpreter selected here, hit OK, and we are greeted with a screen that looks like this. In our new project, let's create a Python file. To do that, I'm going to Right-Click this folder here and select the Python file. I'm going to name this file structure. Now, PyCharm has this setting where it automatically inserts the name of the author, so let's get rid of that quickly, and another thing I'm going to do is, this is taking up too much space, so if I just Double-Click this file name, it's going to get rid of that for me, temporarily, at least. Now, in order for us to have a basic or skeletal structure of our code, we need to do two simple things. First, we need to import our PyQt libraries that we've installed earlier, and we also need to make some specific method calls that will enable our PyQt applications to run. Additionally, we'll have to import the Python's sys library, and that comes with this Python standard library and it'll explain why later, and that's it. I promised you it'll be easy, didn't I? So, let's do this thing. As mentioned at first, we need to import the PyQt modules we need. So if I just do a from PyQt4. QtCore import *, and also, from PyQt4. QtGui import *, finally, import sys, and as far as that's concerned, we're done. We have imported all the dependencies we need in order for our application or our Qt application to run successfully. One thing you may note here, particularly those of you who may be more experienced with Python, is the way I'm importing these two modules. Using this syntax from PyQt4. QtCore import, every- thing is often frowned upon. It's considered bad practice, due to potential name/space conflicts. And I don't completely agree. You'll often see code that says something like this, from PyQt4 import QtGui, and that's completely fine. However, you should know that all Qt classes begin with the letter Q, so name/space conflicts are not likely, and it'll save us a lot of key strokes in the future. Think about it this way. If you're a JavaScript developer, for example, imagine using jQuery, and then, having another library that also uses a dollar sign as a global object. Now if you're getting the red squiggly lines here, indicating that PyCharm is not recognizing the PyQt modules, you might need to verify that PyQt is installed properly. For example, by trying to import it in the interpreter, additionally, you may need to wait a little bit before PyCharm indexes all the third-party modules you have installed on your computer. With this, as I mentioned earlier, we've imported all the

necessary modules, so we need to start making those specific function calls. I'm going to make some spaces here, because usually those function calls will come at the end of our file, at least our main file, and I'm going to create a new instance of the Q application class. I'm going to say app = QApplication, and I'll explain line by line this later. Just bear with me for now, dialog = QDialog, dialog. show, and, app. exec_. Okay, so this probably doesn't make a lot of sense to you right now, but, like I said, I'm going to explain them. This is basically creating a new instance of the Q application. A Q application is a kind of a unique object. It's unique in a literal sense, as in you only have one Q application instance in your Qt application, even if your Qt application has multiple windows, etc. The dialog is a sign to a Qdialogue instance, and the Q dialog is kind of window. We'll look up more about that in our next video. I'm telling the dialog to show up on the screen, and finally, I'm executing our application. Notice here that there's an underscore at the end, because execute, per se, just with this, is a reserved keyword in Python, so the developers had to go around a little bit and add in an underscore. What happens if I run this application? Normally, you would run the application by clicking the button right here, but for the first time, you have to go to Run, and hit this Run. It's going to ask you what interpreter you want. You say structure, usually the name of the file, and you can see our first application showing up on the screen. If I Click the Close button, you can see that the Process has finished with exit code 0. Now, one thing I would like to mention before I go on, is that often, instead of this app. exec_ looking like this, we'll see something that looks like this, sys. exit(app. exec_). What this means essentially, you'll get the same results, and, if you don't believe me, I'll prove it. The only difference here is that with this line added or with this call added, the application will exit with a specific exit code. This may be important to you or probably is not, especially if you're a novice programmer, so you'll be just fine without this sys. exit call, and your application should look like this. You can see here that the QtCore is grayed out completely, because we're not actually using it anywhere in our application right now, but I just wanted to show you the import statements, because we will be definitely using it throughout our course. And, that's it. That's all you have to do in order to initialize an actual PyQt application. What's this? Seven lines of code? Of course, the obvious question you're asking right now, well, Bo, that's all fine and dandy, but where's my application? Don't worry. We'll do that in our next video.

QWidgets - UI Elements

Now that we've looked at the foundation of our application in the previous video, let's look at its building blocks, as well. Let's familiarize ourselves with the QWidget class. QWidget is the base class for all user interface elements. All those UI elements inherit from the QWidget class,

including the QDialog we saw in our previous video. All this talk about QWidgets, and you're probably very confused already, so let's look at some examples. As you can see here, we have several UI elements on this screen shot. Right here, we have the QGroupBox, which is this group box, the QDateEdit, which enables us to edit a date easily, QTime Edit. QLineEdit is just a standard text box. One of the most used elements is also QLabel, which is just static text, and there's also the QTextEdit. Some other elements you will likely use is the QPush button, which is OK or a Cancel, a Yes or No button, and all these inherit from QWidgets, or, if you will, we call all these collectively QWidgets. Think, UI element equals QWidget. Now this screen shot was taken straight from the Qt project documentation page, and it does not seem that their machine is running Windows or that this screen shot is very up-to-date, so don't worry. You will still have the native look and feel or whatever operating system you're working on. Two QWidgets that stand off from the crowd, they're QDialog and the QMain window Widgets or classes. We saw the Qdialog in our previous video, so what exactly is the difference between the two? Well, let's take PyCharm, for example. Imagine that this entire PyCharm window is one QMain window. We have all the usual stuff, the menu bar, the toolbar right here with some but coms, in this case, some buttons. We have the main wooded area, if you will, or the central area of our application, where all this stuff is usually happening, and at the bottom here, we have the status bar. Now, let me go to File and hit the Settings. What pops up right here is also another very rich interface, so that you can see you have buttons, you have a list here, another list here, a search box, a combo box. There's also, of course, the QCombo box class. All in all, it's a very rich interface, but I would consider this a QDialog. Throughout this course, we'll primarily be using QDialog and not QMain window. Let me close this for a second. Well, this may seem somewhat limiting at first, in reality, it really is not. Think of QDialog as the QMain window's younger cousin, which provides most methods as it's assembling, but not implementation for certain things, such as the menu bar, the toolbar, or the status bar, or other things that I've mentioned. Since we understand the basics of the foundation and the building blocks of our application, let's make our first application in our next video.

DEMO: QWidgets

In my last video, I mentioned that the QWidgets are essentially UI elements, or the elements that display on the screen, but you probably did not really like my demonstration of those UI elements, which consisted primarily of an ancient screen shot and me moving my mouse cursor around PyCharm. Since I felt bad for doing that, I've created this small application to demonstrate some of the QWidgets available to you as a developer. I'm sure this entire application looks like a very

familiar interface to you, at least, if you are in Windows, but to start off, we have the Q menu bar at the top here. We have the R sender File, Edit, and Help Q menus, and, if I Click File here, you see that I can specify a shortcut for New, in this case. I can add icons, as well, as I did for New, Open, and Exit. I can even create a checkable menu, so, if I go back here, you can see that Qt is Awesome! is unchecked, and since, Qt is awesome, I'm going to check back. If I hover over the Edit Q menu, we have the preferences and ASAP menu up here. Going a little bit south, we have here what's called a Q toolbar, and what this essentially is, is a shortcut to some of the actions found in the Q menu bar. We have here New, Open, and Exit. Remember that here I have many more. I have New, Open, Save, Qt is Awesome!, Exit and in Edit I have even more. These are kind of, you know, they're general shortcuts that you would find in other applications, as well. Lower than that, we have a cool, little guy called the Q tab Widget, and that provides us with two tabs, in this case, and in it, we can put whatever we want. In this case, I've put a demonstration of various QWidgets available to you, as I mentioned. So, first off, we have here, this is actually what is bordering all these radio buttons inside, is a QGroup box. You can probably see the borders, and you've definitely seen it in other applications. As it is known with radio buttons, you can only select one of those, but that's not the case with the QtCheckBox. As you can see here, we can select multiple checkboxes at once, and this, together, this last QCheckBox here has a tri-state property turned on. What this means is that you can have this little square come in, if there's a special selection you need to make. You can have it also be checked or be unchecked. On to the QLabel and QLineEdit, arguably the two most common Qt Widgets that you are going to use. This is a QLabel essentially. It's just a static text. This is a QLineEdit, something where you can type, for example, Bo or Pluralsight. So normally, when you're designing an application, you have a QLabel that's going to describe to your user what your application is expecting you to put in, such as username, in this case, but Qt also comes with a super-cool property called a placeholder, and, if you are an HTML developer, you may know this as a placeholder, as well. So, in an effort to save some space, Qt developers came with this cool idea, to have the kind of grayed-out text error label put in inside of the QLineEdit itself, and, if I just Click it, that text is going to go away. Now, what's interesting about this is that if you call a method to check what's the text contents of this QLineEdit, it's going to return an empty string. So this, your username right here, doesn't count as text. By moving even more down, we have our QComboBox. You've probably seen this, as well. It just allows you to select different items, and right here, we have another QtComboBox, the same class with a different property, and particularly, this one allows you to have editable items. For example, I can change this 6 to 0 and have it stay that way, you know, when you Click the Submit button or do something else. Here, we have the QDateEdit, and with these up and down buttons, we can change a date or we can just type it, as well, either way. Right below it, we

have another QDateEdit with some different properties. You notice, right off the bat, that we have a different date format here, and, if I Click this down arrow right here, a nice calendar pops up that allows you to easily select the date. Another interesting QWidget I found is the QFontComboBox. Clicking this arrow here allows me to see all the fonts I have installed on my system, along with a preview of those phones, something that you might find useful, when working with your application. Over to the right side and before I actually move over, I would like to note that this is also QWidget. It's called the QVerticalLine. There is, of course, a horizontal line, as well. This is a QSlider, and you may have seen this around, such as when adjusting the volume on your favorite media player. Of course, this is a QHorizontalSlider or QSlider with a horizontal property. There is a vertical slider, as well. Now, these two guys are kind of unique to Qt, from what I know, at least. I've never seen them anywhere else, in other UI toolkits. This is a QDial, which allows you to set some value from 0 to 100, for example, and this is a QLCDNumber. Why does this exist exactly? I'm honestly not really sure. It's just there, I guess. It's all a class in its own. What you'll probably use more often in QLCDNumber is this QProgressBar and that's displaying, you know, the loading state. You've seen that definitely around. Finally, we have here the QSpinBox, which allows you to set some arbitrary numerical values, and we also have the QDoubleSpinBox. It' actually a separate class that allows you to put some floating point numbers or a double number. And, at the end, we have a Button Box and a QPushButton. This is actually of a class QButtonBox. Now, seemingly, these two are identical. We have the OK, Cancel, and Apply buttons here, and OK, Cancel, and Apply buttons there, too. So, what's exactly the difference? Well, this Button Box is a class where you can select properties, such as, I want the OK, Cancel, Apply buttons to appear, but I also want a board, Yes, No, etc. Those buttons will magically appear on the screen. But the more important property of a Button Box, when it comes to comparing it to QPushButton, is that on Windows, this is the order of the buttons when, for example, a setting style pops up. You have OK on the left, Cancel in the middle, and Apply on the right. On different operating systems, the order is not always the same. Sometimes it may have Cancel all the way on the left, OK all the way on the right, and to the left of OK would be an Apply button. If you use Button Box, Qt will be able to figure out auto- matically what operating system you're running on and is going to adjust the positions of the buttons accordingly. This is definitely something to keep in mind, if you're designing a cross platform application. Now, I'm going to another tab where I have more QWidgets, as well. This is the Q plain text edit box. This is behaving like a notepad. There is no way for you to increase the font size, as a user, or make text bold. The QTextEdit does allow rich text formatting, as you can see here, change the size, italics, underline, and bold text. Now, what's interesting about this one is that it takes any HTML or CSS. In fact, behind the scenes, this is actually HTML. On the left here, we have the QListWidget, with

some list items, useful very much for selecting. Similarly to the QListWidget, we have the QTreeWidget, with some sub-items here, and the QTableWidget is right below, as you can see here. We have three rows and three columns, and what's interesting about this particular QWidget is that you can format the self independently of each other. What I usually do is I put the last column as a status or something, and when there's an Error, I color it red, Done is green, and Waiting is usually yellow. Finally, we have the QWeb view, and this is actually loading the live website right now, and it looks like we have some sad news here, but just to show you that this is actually a live website, I'm going to Click here settings, go to, for example, register, and it seems to work just fine. Click, go back to the home page, and scroll down. It works perfectly fine. So, there you have it. In our next videos, we're going to start implementing some of these QWidgets, but my point of this video was to show you that, yes, Qt is very powerful and, it can have all the UI elements that a normal Windows application does, and some extra, like QDial and QLCDNumber.

Hello World Application

As it always goes, let's build our first "Hello World! " application, of course. Our application will have a label, or rather a QLabel, with the text, "Hello World!, a text-headed box, or as we call it, the QLineEdit, and a button with a text close. Of course, that's a QPush button. These are arguably the three most common UI elements used, and that you'll probably encounter throughout this course, and when writing your application. This video will also allow me to show you the fact that the UI elements rendered on my operating system, Windows 8, in this case, look like, well, Windows UI elements. I will also introduce the concept of layouts, but only its basics. We'll look at the layouts in more detail later on. So, let's start working. Remember that in our previous structure of Py video, where I've introduced kind of the demonstration, we only had a large QDialog to pop up on the screen, with the title of Python. There was nothing really on it. So, let's add some stuff to it, the QWidgets that I mentioned earlier, and, in order to do that, we need to override some of the QDialog's methods. Since QDialog is class, we need to write our own class, and I'm going to name my class HelloWorld, and I'm going to inherit from QDialog. In this video, I'm just going to override its constructor method, but the first thing you should do when we're overriding any constructor method, when it comes to QWidget, is call its parent or QDialog's constructor method. The way to do that is simply by typing QDialog. _init_ and passing self as an argument. Now, this falls under the basic structure of the code, as well. Normally, what you would have here, if you're overriding even a QMain window, as well, you would put the QMain window here and the QMain window there. Essentially, what I'm trying to say is that this also the

foundation, the same foundation we presented in the structure of the Py video, but a little bit extended, since we need to override the init method, the constructor method. In some cases, you may find a different way to override this constructor method by using the super keyword, and that will look something like this, super, in this case, we'll actually pass "HelloWorld! " as the first argument, and call init. That's another way to do it. I prefer this first way. As you can see, it's much shorter, and I think this is also con- sidered a newer way to override the constructor methods of a base class, which, in our case, is QDialog. Now, let's add that layout that I mentioned, so I'll just do QVBoxLayout. Layouts are essentially, well, what its name says. It's a layout on how your QWidget should be displayed on the screen. There is a vertical box layout. There's a horizontal box layout, and there is also the grid layout, as well. Think of this as if there were only a vertical box layout, for example. There's horizontal line of elements here, then another one here, then a large one here, etc. A bunch of elements are lined vertically. Similarly, the horizontal box layout, the h box layout, will be a bunch of elements aligned horizontally, and the grid layout will be, for example, you'd have an element in row one, column one, row two, column six, etc. Grid layout is most likely you will use with the more complex application. Now, we're going to add those QWidgets that I mentioned, so the first one is the label, and the QLabel class constructor takes its text as an argument, so I'm just going to type "Hello World" here. The next one would be line_edit. QLineEdit constructor method also takes an argument with text, but we want our line_edit to be blank. Finally, I'm going to add a button, which is the QPushButton, and again, you can add whatever text you want here, which, in our case, is going to be "Close. " When Clicking this button, the application should shut down. Now, what I need to do is add these QWidgets to the layout. I'm going to say layout. addWidget label, and I'm going to copy/paste the same code two more times, add the line_edit, as well, and finally, the button. And, the last thing I need to do is instruct the QDialog constructor method to use this layout to display the elements on the screen. So, I'll just say, self. setLayout layout. Finally, one important thing to remember to do is rename this QDialog. We don't want our empty, blank, useless QDialog here. We want our new "HelloWorld! " dialog to pop up. If I run this file, watch what happens. We have a newer, much smaller dialog, and you can see that the elements are arranged vertically. There's our label. We have the text of "Hello World! " There's our QLineEdit, which we can type any text we want, also, "Hello World! ", for example, or Pluralsight, and there's our Close button, as well. However, our application doesn't really do anything right now. It just displays these kind of useless Widgets. As you can see, if I Click this Close button, nothing happens. So let's add some interactivity to our application.

Events

Before we begin working on our interactive application, and just to be sure, by interactive, I mean there's an interaction between the user and the application. We need to understand one very important concept, Event. When I say Event, you may be thinking of something like concerts, or a picnic, or a parade, or something fun. Well, it's not exactly what I mean, but I promise we'll try to make Events, when it comes to programming languages, at least half fun. Events are actually crucial to any interactive appli- cation, and most programming languages, such as Java, C#, or JavaScript, incorporate them in some way. Without getting into the computer science textbook definition of what Events are or are not, I'm just going to say this. An Event is when something happens. This actually sounds a lot like something taken from the dictionary. Well, that's what it is, but what exactly do I mean when I say, something happened? I'm moving my mouse now. An Event has occurred. I just Clicked something, maybe. An Event happened. Did I type something on my keyboard? There is another Event or actually multiple Events in a row. When it comes to wording, we usually say that an Event is emitted, fired, or broadcast. You can take your pick. I normally say fired or emitted. Also, you will often hear me of referring to Events as signals, and this is actually coming straight from the Qt syntax, as you'll see soon. Now, we actually need to capture those Events and do something with them. In Qt, if an Event is not caught or handled, as we actually say, it is being disregarded. This handler can be any Python callable, and by that, I mean any Python function or method. You can think of a handler as an action you want to take when a particular Event occurs. One note: A handler is sometimes called an Event Listener, as well. Those two terms are equivalent. Here, we have an example of an Event that is not being handled, and, as such, is being auto- matically disregarded by the garbage collection in Python and Qt. Right here, we have an Event that, when fired, prints the message "Hello" to the Python console. Finally, we can also attach Events to our own methods, as I mentioned earlier, and do whatever we want to do. One good example would be asking the user if he or she wants to save the changes when exiting the application. Let's make a definition. A handler, which is a function, is attached to an Event. So you've seen in our previous video that when I Click the Close button, nothing happened. Be aware that an Event, specifically the clicked Event, has been fired, but there was nothing to handle it, so it was automatically disregarded, as you can see in our first example here. But if you were paying really close attention in the last video, you also noted that we didn't really specify any Events anywhere in our code, yet, I'm saying that an Event has been fired. Well, that's because Qt actually comes with some built-in Events, such as clicked, text change, for example, which fires when you change a text in the QLineEdit, and remember that's our text box. What's also interesting about this text change Event, is that it also passes a

parameter to whatever handler is attached to it. There are many other Events available. Overall, I'd say that every UI element has some sort of an Event that it emits. One more important thing I feel I should say, is that an Event is sometimes referred to as a signal, and the handler is sometimes referred to as a sloth. I know, ugh! So many different names, but the signal sloth name actually comes straight from the Qt syntax. Now, you're probably confused more than ever, but trust me. Even though Events are a very powerful concept, they're actually really simple to understand. As I said earlier, it is our job to capture or handle those Events and do something with them, and that's exactly what we're going to do in our next video.

DEMO: Events

We have here our little "Hello World! " application, which, in the meantime, I've renamed from structure. py to HelloWorld. py, and just to remind you what it looks like, I'm going to run it. Recall that if I Click this Close button, nothing happens, at least nothing visible happens. As I mentioned in the previous video, an Event has been fired, but there was nothing to handle it. Let's fix that. In order to attach an Event to handler, we need to connect the two, and for that, we use the following syntax, and to say, button. clicked. connect (self. close). Now, let me explain this line a little bit. Button is our QPushButton, with a text of Close, as we have defined earlier, and the QPushButton just happens to emit the Clicked event, which, as you may imagine, happens when the user clicks that button. The connect is what actually attaches our Event to a handler, connects the two. The handler, in our case, is this self. close call. This self. close is a built-in method in QDialog and some other QWidgets, as well, that is simply going to close the window. One important note: Even though this self. close is a function, we're not passing it as such. We're just passing it as an argument. What this means translated, is that you should not add parenthesis at the end of the handler, even when we write our own handlers, as you'll see later. Now, what we need to do is, well, we don't' really need to do anything anymore. In a true Pythonic way, this one line of code is what does it for us. So let's test it. If I hit the Close button right now, the application closes, and, as you can see here, the process has been finished. Now let's do something more exciting. Remember that we have our HelloWorld QLabel and an empty QLineEdit. Well, what if we wanted for the text that we type in QLineEdit to immediately appear in our QLabel as we type it. We're in luck. There is a very simple way to implement this, too. If I just do line_Edit. textChanged. connect(label. setText), this is going to work. But before I test it, let me explain this a little bit. Line_edit is our QLineEdit. The textChanged in this case is the event that is emitted every time the text has been changed in our QLineEdit. Connect is the same as above, and the label, being our QLabel, has a set text method that's going to set the text of the label, as its name

says. Now the setText does take one argument, which is a text of the label. So, if I say label. setText HelloWorld, it's going to set the text of the label to HelloWorld, but notice here that we're not passing any arguments. Let's see what happens if I test this. And, if I type here, Pluralsight!, you can see that the text and QLabel appears jut as we type it right here. What happens behind the scenes is that every time a text has been changed in this QLineEdit, an event has been emitted, and we have connected that event to our QLabel. Hello Bo! It works through. Anything will work, and, of course, our close button still works. But how exactly, does the label know what text to update itself to? We're not passing any parameters. Well, I've already mentioned earlier that events can carry parameters or arguments with them, but to explain that a little bit better, let's refactor our code. Let's create our own method called changeTextLabel. This method is going to take text as an argument, and all we're going to do in this method is update our label, and to do that, I'm going to first need you make this label variable an instance variable, so we can access this for modern methods, as well. And, right here, I'm going to call it sameMethodsetText and set it to this text from the parameter. This obviously needs to be changed to self. chgTextLabel. Notice here again that I'm not passing any arguments here. What happens if we test the application now? Hello, Pluralsight! You can see that we have the same results, even with the code refactored. As you can see, the textChanged event actually sends a parameter to our handler, in this case, the changeTextLabel. But, how did I know this? Well, from the documentation, the QtDocumentation specifies what event sends what parameters and what types of parameters, as well, such as integer, string, boolean, etc. Now, I also want to make you aware of an older syntax for attaching a handler to an event. You'll still find this syntax being used in some older PyQt or Pyside applications, so, older syntax looks something like this. Let's get rid of that real quick. So, if we wanted to accomplish the same result as we had with our line_edit here, the older syntax would look something like self. connect(line_edit, SIGNAL("textChanged")) We need to pass the Q string, which is kind of a drop in replacement for the Python string, as an argument, and finally, we need to pass our handler, which, in our case, is self. changeTextLabel. Yeah, you can probably see why there's a new style syntax here. You really should not be using this syntax, in any of your code. This is just something to make you aware that there is this older syntax so that you're not confused, if you happen to encounter it in some older code. Well, now that we know signals or events, there's really just one more thing keeping us apart from creating our first real application, and that's layouts. You saw that I have been using them from the beginning, but why? Let's answer that question in our next video.

Layouts Are Your Friends

I've touched briefly on the topic of layouts before, but never really explained what they did and why they're useful. Layouts, like events, are a simple, yet powerful concept whose primary focus is not just in the UI design, but the user experience, as well. Many programming languages incorporate layouts in one way or the other, and, for example, if you're an Android developer, you'll definitely know what I'm talking about. But let me demonstrate some layouts that Qt ships with. First off, we have the Vertical layout or the vertical BoxLayout. This layout has its QWidgets, remember those are the UI elements, aligned in a vertical order. We can see here that our elements are aligned one below the other. Recall also, that our Hello World! application has this kind of a layout set. Contrary to the vertical layout, we also have the Horizontal layout, or the horizontal BoxLayout. In this mockup, you can see that the elements are aligned next to each other, in a horizontal fashion. We also have the Grid layout, which, as you probably assume, aligns its elements in a Grid, or a table of rows and columns. So, we see here that we have our two elements, the QLabel and QLineEdit, on the first row, but our QPushButton is on the second row. Similarly, notice that our QLabel is in the first column, but the QLineEdit and QPushButton are in the second column. With Grid layout, you can have as many rows and columns as you'd like. And, also, the Grid layout is likely the one you will use the most when writing somewhat complex applications. I should also mention that there is another one, the QFormLayout, and although it is very similar to the Grid layout and its behavior, the Form layout can contain two columns, and only two columns. Normally, you would use the Form layout when you want your user to fill out some sort of a form. So those are the four basic layouts that the Qt ships with, but I still haven't said anything about their usefulness, so far, and it may seem odd to you that we have to write many extra lines of code, just so we can add the QWidgets to our application on the screen. Well, I didn't talk about that, because the best way to demonstrate their usefulness would be when I'm demonstrating the layouts themselves, which is what I'm going to do in our next video.

DEMO: Layouts

In the last video, I showed you what layouts are, but just as some mockups and images on the screen, so in this video, we're going to see them in action. I have here our Hello World! application from before, which, as you might remember, uses the vertical BoxLayout. But just to remind ourselves, let's run the application, and you can see that the elements are aligned in a vertical order. We have our QLabel at the top, our QLineEdit in the middle, and finally, our QPushButton at the bottom. Now, if I would only change this letter v to a letter h, I would make this a horizontal BoxLayout. Let's run the application We still have the same elements, as you might suspect, but this time, they're aligned horizontally. We have our QLabel on the left, our QLineEdit in the

middle, and our QPushButton on the right. Similarly, we can make a QGridLayout, as well, but this is going to require some setting up. We're namely going to instruct our layout what elements we want to display in what row and column or, in other words, to simplify. We want to tell the GridLayout where we want our elements positioned within that grid. Now, the rows and columns in the GridLayout are zero-based, so, if I wanted our label here to be in the first row in the first column, I would say comma 0, 0. The second parameter, but the first 0, denotes the row, and this third parameter, but the second 0, denotes the column. I want our line_edit to be next to the QLabel, meaning in the same row, but in the second column, so I'll just add 0, for the row number one, and I'll add 1 here, for the column number two. As you may imagine, this works kind of like the same as a list in Python. The 0 index is the first element in the list. Finally, I want our QPushButton to be to the bottom right of the screen. That means we want it to be in the second column and the second row, as well. I'll just say, comma 1, 1. Let's run the application and voila. It's grid-aligned. We have our QLabel to the left in our first row and first column. We have our QLineEdit in the first row, but the second column, and finally, our QPushButton in the second row and in the second column, as well. I also promised you in the last video that I'll demonstrate the usefulness of the layouts, as well. Looking at the code here, you're probably wondering why our layouts are cluttering it. I mean for every widget that we want to display on the screen, we have to call this addWidget on our layout. This seems very silly and redundant, and definitely not Pythonic. Well, there are some tools that will help us automatically manage our layouts, so that we don't have to care about them almost at all, but that's slightly more advanced, and they'll be in the part two of this course. For now, I'm just going to demonstrate the usefulness of the layouts. I have here the application that looks the same as our Hello World! application, but don't be deceived. Behind the scenes, there is a slightly different code, the difference being that in this application on the screen, I have no layouts defined, and, as you can see here, I have exited from our Hello World! application. Now, watch what happens when I resize this application. Remember, this one does not have any layouts defined. I increase its size, or modify it, or do anything with it. You can see that the resize has no effect on the UI elements whatsoever. In fact, let me make this smaller, and you can see that the UI elements are clicked. This is definitely not something you want your user to experience, as well. Now, I'm going to close this application, and run our application with the QGridLayout, and watch what happens when I resize this one. You can see that the elements are kind of reflowing and realigning themselves. In other words, they're obeying the resize events that fires when we resize our window. Now, if I want to make this one smaller than the elements themselves, watch what happens. I can't. I can't make it any smaller than the QGridLayout tells me to. Qt is smart enough to figure out, okay, this is really the smallest you want to go. So, this may seem like an insignificant problem, but it's not. Trust me. You're listening

to someone who has made the same mistake before. I completely thought of layouts as irrelevant. Long story, short, I spent days, days, refactoring the code, so that I can add layouts to it. Moral of the story is, use layouts. They're your friends. Well, the moment is here. Now, there's nothing stopping us from creating our first real application, and this is what we're going to do in our next module.

Summary

Let's recap what we've learned in this module. First, we'll look at the Basic structure of the code, the code that must be present in every Qt application, in order for it to run. We then defined what QWidgets are, and remember, they are the UI elements on the screen, and I also showed you some of the QWidgets that exist in Qt. We also saw what events or signals are and how they're related to their close cousins, the handlers, or the sloths. Remember, the event is fired, or synonymously, a signal is emitted, and then, it's handled by the handler or the sloth. Lots of naming there, so, hopefully, this makes sense. An event is the same as a signal, which fires or emits, and a handler is the same as a sloth. These names are just synonymous. We saw the first hand communication between the two, when we close our application by Clicking our Close QPushButton, or how our QLabel changes its text, when we edit the text in our QLineEdit. Finally, we've seen how simple, yet powerful, layouts are and how they can help us build user-friendly applications. Coming up in our next module is designing a real application from scratch, using the knowledge we gained so far, but also learning new things along the way.

Our Real App

Our First Real Application

Welcome back. My name is Bo Milanovich and you're watching Python Desktop Application Development course on Pluralsight. By now, you should have a basic understanding of some of the core concepts, when it comes to desktop programming, with Python and Qt. In this module, I intend to put what we learned so far to use, but also show you some new things that will make your application more user-friendly. Speaking of the application, in this module, we're going to be writing an actual application from scratch, and I thought it might be a good idea to write a simple file downloader application, which we'll just download files from the Web, and which will also

allow me to demonstrate some very useful features that Qt ships with. So let's talk a little bit more about our new file downloader.

PyDownloader - What Is It?

Every time before I start working on a new application, I like to have a notepad opened on my computer and a pencil and piece of paper on my desk. One of the first things you should do is ask yourself, what do I want my application to do? What do I want to accomplish with it? Anything that crosses your mind, you can write it down in the Requirements section of your notepad, just as I kind of did here. Let's talk more about the Requirements of our Py Downloader application, and for the record, yes, I do realize that Py downloader is not a very creative name. The basic Requirement of our application is to be able to Download files from the Web. The application will actually download the file from the URL, the user specifies in a QLineEdit box, that is going to be in our application. The QProgressBar will also be present in our application, and it will show the user the progress, the percentage of the file that has been downloaded so far. We also would like to Allow the user to browse their computer, so that they can specify the location where they want to save their file, and this will be a dialog that is the same as the one you're probably seeing, at least, if you're using Windows, when you go to, for example, file menu, and then, Click on Save As. When the download completes, we should notify the user, by showing a message on the screen that says, well, download completed, or something similar. Finally, we should try to predict some errors and exceptions that may show up, while using the application, for example, when a file has not been found in the server, when the server returns the 404 message, and, also, in that case, display an error message. The next thing I do is create an application mockup, a sketch, or a blueprint, if you will, that gives us the basic idea of what the application will look like. This is where a piece of paper and a pencil come in handy, but since I obviously can't draw on my screen using a pencil, or at least, I haven't tried, I use an online mockup tool. So, this is what our application is going to look like. You see that we have the URL and the File save location QLineEdit, our QProgressBar, and our QPushButton, which will call a method that will download the file. And, don't worry, this mockup is a draft. I mean, we can modify it slightly along the way, as we see fit, while we're actually working on our application. Finally, let's look at our workflow. The first thing we should do is design the UI, and, following that, we'll write some code that will allow for some basic functionality of our application, things, such as actually downloading the file from the location specified, and, in the end, we'll implement some extended functionality, such as exception handling, the file-saved dialog, and the success or failure message box. While working on our application, feel free to take baby steps. I most certainly will, and by that I mean, we'll

often test our application to see if everything we've done so far works just fine. So let's start with step one. Let's design our Basic UI in the next video.

Basic UI Design

I've created a new file called PyDownloader about the Py, and it's still in the same PyCharm project we created early on in the course, and the file right now contains the blueprint code that we also defined back at the beginning of this course. As mentioned in the previous video, this one will design the interface, and our application's going to be using a vertical BoxLayout, so let's add that, now it is QVBoxLayout, and recall that we need two QLineEdits, a QProgressBar, and a QPushButton. First one we'll call url, the second one, we'll save_location. The ProgressBar will just be progress, and finally, our QPushButton will be called download, and it'll have a text of "Download", as well. Of course, we need to add all these to the layout, and, in interest of saving some time, I'm just going to copy/paste the code. First, make sure it's all indented properly, and this is fixed. Finally, of course, we need to set this layout to our QDialog. Nothing new really here, but let's run our application. The first time you run a new file in the existing PyCharm project, you have to Right-Click here and select Run. Later on, we'll be able to use this standard Run button, I guess. So, if we run the application, we see that the interface is functioning, but it's also very confusing. There's no description of what these line_edits are. The ProgressBar looks kind of like an error on the screen, honestly, and the title of the application is still python. In fact, the only thing that looks decent in this entire application is our QPushButton that says, Download. So, let's make the entire application look more than decent. First, let's change the window title from python to something more meaningful, so to do that, I'll just call self. setWindowTitle, and I'm going to put "PyDownloader" in the window tile. Note that it's usually a good idea to store this in a variable, but this will do for now. Then, let's add the placeholder text to our QLineEdits here. One important thing is that you should add the placeholder text before adding the Widget to the layout, so we're just going to insert some code here, and I'll say, setPlaceholderText, which, as its name says, PlaceholderText, and takes a string as an argument, in this case, our PlaceholderText. The first one will have the url, and our save_location will have the PlaceholderText of "File save location". Let's run the application to make sure that everything is working so far, and notice here that now we can Click this regular Run button, and you see that the window tile is now PyDownloader, and the PlaceholderText is here, but is it? Notice that the URL PlaceholderText is missing, but that's just because we currently have the URL QLineEdit selected. If I select something else, it will show up, as you can see right here on the screen. This happens because of something called focus. Our URL QLineEdit, in this case, has the focus, which translated means,

has this blinking cursor on it. So we can easily fix this by setting the focus manually on the entire application. If I close this, I can just add at the end, self. setFocus. Let's run the application once again just to make sure, and you see that both the URL and File save location PlaceholderText is there. But one thing that's still looks like we use our QProgressBar, which is the only thing left, kind of still looks like nothing, so let's make it look like something. If I actually make our ProgressBar have the value of zero, and to do that, I'll just call this setValue and pass an integer as a parameter, in this case, 0. It will make it look like something. Baby steps, so run the application. You see that 0% appears on the screen. However, this still kind of looks ugly. This 0% is off-setting. Everything's vertically aligned, but this 0% is to the right of the ProgressBar, so let's move this 0% to the middle of the ProgressBar. In order to do that, I'll just call progress. setAlignment, and I'll pass the argument Qt. AlignHCenter. Now, what's interesting about this Qt static class is, not only that it has a very super-extensive functionality, but also that it's actually the first time we use something from our Qt core module that we've been importing the whole time. As you can see, it's no longer grayed-out, meaning that something is using it. In this case, that's our Qt static class. Let's run the application again, and you can see that the 0% is now in the middle, and the QProgressBar is stretched out. Now, we have our UI in place, and even in this first basic UI video, I'm sure that you've learned a lot of stuff. We've definitely covered a lot. We haven't called the setValue before. We haven't definitely called the setAlignment. We haven't set the PlaceholderText manually, and all that stuff, but still, our PyDownloader doesn't do anything yet, so let's fix that, as well.

Functionality

Our PyDownloader, with just a functioning UI, is in front of me here, so let's add some functionality to it. In this video, we'll add basic functionality, such as the ability to download a file from the Web and save it to the specified location on the user's hard drive. What we're going to do first is create a new method called Download. I'm going to scroll down, and we'll just put pass for now. This method will be our event handler, or a sloth, for our signal, whatever you prefer calling it, so we need to connect our download QPushButton to this method. We learned how to do that in our events video, so. clicked. connect(self. download), and, of course, our button is not a button but that download. This method will actually be responsible for downloading the file and, also, saving to our hard drive. Remember, both the URL and the location of hard drive are two things the user has specified, so we need to obtain these from our QLineEdit. In order for us to be able to do that, we need to make our QLineEdits and instance-wide variable. We'll just do self. url, as well as self. save_location, and, remember that we need to do this everywhere where

the URL and the save_location variables are used. Then we can go back to our download method and create a new local variable called url, and in order to actually grab the text of our QLineEdit, we need to call the text method. This text method actually returns a string, so our URL will be set to or whatever text the user has typed in our QLineEdit, our URL QLineEdit, and we'll do the same for our save_location, as well. And, now, to actually download the file from the Web and save it to our hard drive, we first need to import Python's URL library, specifically the urllib. request part, so just go back to the top of the file here and import it. One thing I should note is that this urllib. request is not present in Python 2, but there are some alternative methods, to get the same result that we're going to get, as well. From this urllib. request, we're going to use the urlretrieve method, to actually download a file and save it to the disk, for a temp time. What's handy about this method is that it provides an optional parameter where you can put your report hook, and by this, I mean a function that urlretrieve will report to and tell it things, such as, oh, this is the file size, this is where I am right now, etc. It's very useful indeed, so to implement it, let me first create such a function, and let's call it report, and this report function must take three arguments. It's a chunk number, the maximum size of chunks that are read in, and a total size of the download. It will be called once the start, at the start of the reporting, and after each chunk of data is read from the server, but we really don't have to worry about that part. All we need to care about right now is creating our report function or method that takes three arguments, blocknum, blocksize, and totalsize. So let's put pass here for now, as well, and go back to our download method and actually report to this one. In order to do that, we'll just call urllib. request. urlretrieve. The first argument it takes is the url, and we have this as url. The second one is our save_location, that's this. The third one is the report hook, and, in this case, it's our self. report method. In this report method, we'll actually do the calculation as to what percentage of the file has been downloaded so far, and update our ProgressBar accordingly. First, I need to create a new variable called readsofar, and this will be blocknumber, the chunk number, and multiply with the blocksize, and we need to create an if statement. This if statement will check and make sure that the total size is greater than zero, because sometimes when you're downloading a file from the server, the server doesn't necessarily report the total size of the file. In here, we'll do a little bit of math to calculate the percentage, and they'll be just readsofar, times 100, divided by total size. Now, we can update our ProgressBar, but before we do that, we need to make our ProgressBar also an instance-wide variable, so we'll do self. progress. setValue everywhere in our contructor method, and in here, we're going to call self. progress. setValue, and this is the same setValue method that we've called earlier in our last video, when we were setting the value, the initial value to zero. Now recall that the set value takes an integer as a parameter, so this is why we're converting our percentage points to an integer right here. Finally, let's test our application and see if it works. The first thing

it's asking for is a URL of the file, and I have here a kind of a URL that is more of a test our server speed URL. Basically, it's a 10MB file with nothing really in it, but it's good for demonstration purposes, at least. Hopefully, the folks at reliable service don't mind, although I would recommend that you find a different file to download, and, of course, it can be any file. Now, I'm going to put the same location as D:\test. bin, for example, and this is on my D drive, obviously, and let's see what happens when I Click the Download button. Yah, our application is working, and it just downloaded a file, so let's see if that file is really present in my D drive. And, as you can see right here, we have the grade results. The test. bin file is really present in my D drive, so that means that our application is functioning perfectly. I hope that you're as excited as I am, because our application works, but let's add some more functionality to the application, to make it kind of more user-friendly.

Extended Functionality - Part 1

Let's add some extended functionality to our application. We're going to add some messages that inform the user of the success of the download or its failure, and we'll also enable the user to explore their hard drive using a Save As dialog and be able to specify the file name that way, instead of typing it manually in our QLineEdit. Let's do the easiest part first, and that's add the information message when the download finishes. Qt is kind enough to provide us with a very simple function to display a message, so if I scroll down to where our download method is, and once the file is downloaded, I can just do QMessageBox. information, pass some arguments that I'll discuss in just a second. (typing) Now let's analyze this function call a little bit. The self parameter here is the parent QWidget. Now, we really haven't mentioned parent context before, and it's not really relevant for an introductory-level course. Just be sure to put self here for now. The second argument "Information", in this case, is the title of the MessageBox, and finally, the third argument is the contents of the message itself, which, in our case, includes some static text. While we are at it, it would probably be a good idea for us to reset the application to its default state, meaning let's reset the ProgressBar back to zero, and remove the text on the QLineEdits. It's very easy to do that. Let's call self. progress. setValue(0), just like in our constructor method, self. url. setText. This is a method that is used to set the text for a QLineEdit, and we'll do the same for our save_location, and with this, we've kind of reset our application to its starting point. Before we test the application, let's create some exception handling, as well. From looking at the code, it seems that this line, is most likely the one that will cause an exception. For example, when a server returns a 404 error, meaning it can't find a file. We'll wrap this with a try and except block, like so, and this is a broad exception clause, and in our exception, we want to also display a

message on the screen, but, in this case, we want it to be an error message or a warning message. We'll just say, QMessageBox. warning, pass very similar parameters to below, kind of in this case, should be "Warning", and let's say, "The download failed". We also need to return here so that the rest of the code in this method is not executed. Alternatively, we could also put this code block within the try clause, but I kind of prefer keeping my try and except blocks short and sweet. Now, let's test the application and see if it works the way we told it to. If I put in the same URL as in the previous video, and I'm going to still put the D:\test. bin, hit the Download, and wait a little bit. You can see that we have our information message pop up. We have the information as the title. The download is complete as our message text, and notice that even though we haven't written any code for our OK button or our information icon here, then Qt handles it automatically, knowing that it's an information message. If I close this dialog, you see that the application has reset itself to kind of default state, but let me put the D:\test. bin in here again and put some random URL here, hit Download. The exception has been raised, and a warning message pops up with a warning triangle here, and the download fail message contents. Of course, you could add some more validation in error handling, such as not allow the user to proceed if any of the QAlignEdits here are empty, but those are out of scope for this video.

Extended Functionality - Part 2

Finally, we need to implement the ability for the user to browse their computer and save the file to a destination on the hard drive. For that, we need a slight UI change, just to add the browse button. I'll go back up, and I'm going to add another button here, another QPushButton, with a text of "Browse", and this QPushButton, the Browse button will be located right underneath our save_location QLineEdit. Finally, we need to, of course, connect our button, the clicked event that it emits, to a handler, and obviously, we need to write this method, as well. Let's just go up on the top. In this method, we will have our save_file as dialog pop up. We'll have a new variable called save_file, and assign into a nifty and very useful method of the QFileDialog static class called get save_file name, so let's do that, getSaveFileName, and let's pass on parameters first. (typing) Okay, that looks kind of complicated, but when you think about it, really it's not. This self right here is also that parent QWidget we mentioned in the previous video. The caption, in this case, is the title of the saved dialog, which, in our case, is "SaveFile As". Now the directory, the starting directory that the save_file dialog should open when it first pops up. The dot here denotes current working directory. In our case, that's going to mean whatever directory our PyDownloader file is located in. Finally, the filter has to deal with the extensions, but since we're saving a random file from the Web, it can be any file, so that's why you have "All Files (*. *)" But in

PyQt, this save_file will not contain a string, an absolute path, or location to the file that the user is trying to save. In PySide, however, the save_file will be a to pull with the zero element being the string, the absolute path, and the second element being a boolean, so, just something you should note. If you're using PyQt, the save_file will be a string that's an absolute path to the file the user has selected, and in PySide, it's going to be a to pull. In order to make our application work without rewriting it a whole lot, what we're just going to do is update our save_location QLineEdit with this absolute path here. So, I'm just going to do a self. save_location. setText, again, (save_file). Let's test our application. Now, we see our Browse button here, and, if I Click it, this dialog pops up. There is a save file as in the title of our dialog, or as our caption, and just to prove it to you that this indeed is a save file dialog, if I select one of the earlier codes, such as structure, and hit Save, it's going to ask me if I want to override or replace the existing file. Since I really don't want to do that, I'm going to go scroll here, and go to my D drive, and create a test2. bin, Save it, and we can see here that the QLineEdit has updated with the text contents, the full path, the absolute path, to the file I have selected in the save file dialog. But, ummm, the text is definitely there, so it kind of seems to be working, but the slashes are forward slashes, and we know that Windows uses back slashes as folder separators, but the Mac OS and Linux both use forward slashes, and we want our application to be just as beautiful in all operating systems. So, what do we do? Well, luckily, Qt has a function that will easily allow us to overcome this minor problem. I go back to the code. I'll just replace this save_file with something very similar. I'm going to call a method called toNativeSeparators and pass save_file. This is a static class, and this toNativeSeparators name of the method kind of explains itself. Let's convert whatever you pass me toNativeSeparators, which on Windows would be a backslash and on Linux and Mac OS would be a forward slash. Now, let's access the application again. I'm going to put a valid URL from before here. I'm going to browse my drive to say test2. bin. If I Hit Save, notice here that we have the backslash now, Hit the Download, and our application is downloading, and our information message pops up. Let's close the application. Everything is reset back to its default state. Everything works like a charm. Whew, that was a lot, but was it really? If you look at it, there's only about, what, 70 lines of code, and that's including all the blank spaces. Even though this is a very basic and simple application, this is still impressive, in my opinion, but I would still like to congratulate you, because our application is complete. Now, let's jump back a little bit and sum up what we've learned so far in this module.

Summary

We have definitely learned a lot in this module I'd say. Not only did we use the knowledge we gained in the previous module and put it to practice, but we also learned a lot of new things, such as the Save File Dialog, and, by the way, of course, there's an open file dialog in Qt, as well, but also, how to make our application look as native as possible in all platforms, with the use of toNativeSeparators, for example. We saw how the QProgressBar gets updated. Overall, I'd say we learned a lot, but, even so, we barely even scratched the surface of Qt. I'm sure that you realize that even though we built a very simple application, we'll still explore lots of the main parts of the Qt and desktop development, in general. Our application has 70 lines of code, but will you be intrigued, if I tell you that you can get rid of a large part of that code, that boring part about dealing with the layouts and adding Widgets manually, and have something do it automatically for you? Well, you definitely can, and I'll give you a hint. Watch the conclusion video for this course for some more details.

Let's Relax!

Introduction

Welcome to Let's Relax module of the Python Desktop Application Development course on Pluralsight. My name is Bo Milanovich. Yes, this title is intentional. In the previous module, we saw how we can build a functioning application in less than 70 lines of code, but I'm sure that, at one point, you raised your eyebrows. For example, when I was explaining the setText method on QLineEdit, I'm guessing some of you had a reaction along the lines, oh, wow, Bo, you're telling us that now? Well, in this module, we'll sail through. I'm going to go back to what I call advanced basics. In short, I'll explain some commonly-used methods on some commonly-used QWidgets, such as the ones we've seen already, like QLineEdit and QLabel, but also, other ones that we haven't seen in action, just yet, QComboBox, for one, which is a dropdown selector and QCheckBox, which is a checkbox. So let's sit back, relax, and start off with our good old friend, the QLineEdit.

QLineEdit

I've created a new file called QWidgetDemos, and this is just the same old basic structure of our code, but you can see here that I've added the QLineEdit and one QPushButton with a text of "Close" that's going to close the application. You also see that I'm using the vertical BoxLayout,

and I'm also manually setting the focus on the entire application. Now, we've already seen the setText, the setPlaceholderText, and the text methods, but let's remind ourselves what they do before we move on. To set the text of a QLineEdit, we use the setText method and pass a string as an argument, like so, line_edit. setText, let's say ("Hello Pluralsight"), and if I run the application, remember the first time you run a new file in the same project, you have to Right-Click here, Hit Run QWidgetDemo, and we can see that this Hello Pluralsight is displayed in our QLineEdit. To grab the text currently displayed in the QLineEdit, we just call a text method, and, in fact, I'm going to print this manually-set Hello Pluralsight message to the Python console below. So, let's create a new variable called text that will store the text we have in our QLineEdit and print it to the console. Now, if I run the application, we can see that Hello Pluralsight appears in our QLineEdit, but also down below in the Python console, as well. Finally, if we want to set the placeholder text, and, remember, that's the grayish text, we use the setPlaceholderText and also pass the string as an argument, so I'm just going to get rid of this setText call for now, and replace it with setPlaceholderText, and let's put something meaningful, such as, ("Enter username"). Now, remember, that setPlaceholderText is not the actual text of the QLineEdit. It's just something like a descriptor. Let's see what happens when I set this PlaceholderText and I try to print the text to the console. If I run the application, you see that this grayish Enter username does show up. However, down below in our Python console, there is nothing. That's because this is not text. This is just a descriptor. Okay, well, we've see all of those in the previous module, now let's explore some other cool things that QLineEdit ships with. There is a handy method available to us, if we want to select all texts in the QLineEdit, so while I need to replace this PlaceholderText with the setText, and, also, probably change this back to Hello Pluralsight, but in order for this method to work, one of the things I need to do is uncomment out this setFocus, because otherwise, we'll be stealing focus from our QLineEdit and setting it on our application. It is very simple to select the entire text in the QLineEdit. We just call line_edit. selectAll, very Pythonic. We run the application. You can see that the entire text has been selected. We can also set the text as Read Only and, therefore, disallow editing. Let's first get rid of this selectAll, and here, I'm just going to say line_edit. setReadOnly, and this method takes a boolean as an argument. Obviously, if we pass True, it's going to set this QLineEdit to a Read Only module. We run the application and see that I can't really type anything here, and I'm pressing my keyboard keys right now, and there's nothing I can do. Finally, and this is a nifty one, we can also make our QLineEdit behave like a password field, where it will replace the characters we type with the asterisks. Get rid of this ReadOnly, and let's actually get rid of this setText, as well, and all of these. The way to accomplish this is by setting the echo mode of our QLineEdit, and this is very simple, as well, Line_edit. setEchoMode (QLineEdit. Password). Now, let's run the application, and I'm going to type some text, Hello

Pluralsight. You can see that this looks like a password field. There. Those are just some useful methods we have in the QLineEdit. There are many more, but they're out of the scope of this video.

QLabel

Same old code, but here, I have a QLabel, instead of the QLineEdit, and we've used QLabels before, but let's familiarize ourselves with it a bit more. In our earlier videos, when we were creating an instance of our QLabel, we'd set its text by passing the string to its constructor method, or, in other words, we'd put the text we want to be displayed in the parenthesis, right here, and that's fine. But there's another way to do that, and it can be done with, yes, you guessed it, this setText method. So let's get rid of this text here, and, instead, just call label. setText, type, let's say, ("Hello World") this time. If I run the application, we have our QLabel with the Hello World text in it. Of course, if we wanted to grab the current text in the QLabel, we'll just call the text method, same as with QLineEdit. But, what's interesting about this QLabel version of setText, is that it can take HTML for its text formatting. Yes, Qt actually really likes HTML and CSS, as well, which I'm sure will make lots of people happy. So, if we want to make our text bold, we just add the HTML bold text. Let's run the application, and we can see that our Hello World is now bold. Now, there are other ways to make the text bold, but we'll stick to HTML for now. QLabel does not really offer that many methods, but it is actually very versatile. As you saw, it can contain both plain and rich text, formatted text, but it can also contain a picture or even a movie, a file, a clip, whatever, but those are slightly more advanced functions.

QCheckBox

Now, we're getting into some new stuff here and in this video, we'll discuss the QCheckBox, which, as you probably assume, is a checkbox. The QCheckBox doesn't really come with that many methods, but it comes with all the stuff you need. I already have it created here, and, as you might assume, it also has the setText method that is going to set its text, same as with QLabel and QLineEdit, but let's run the application and see what checkbox looks like. Now obvoiusly this is just a standard-issue checkbox that you've probably seen a billion times on your computer screen. So, let's play with it a little bit. By default, as you've just seen, the QCheckBox is unchecked, so to set it as checked, we can just call checkbox. setChecked, and then, we pass a boolean as an argument. Obviously, it can be True or False. So, if we set it to True and then run the application, we can see that our checkbox is now checked. Now, obviously, we want to have

the ability to see if the checkbox is checked or not, and let's have some fun here. When you check or uncheck the checkbox, lots of checks there, it's going to fire an event called stage changed. Let's create a handler for that event, in which we'll see whether our checkbox is checked or not. Obviously, first, we need to get rid of this, but we also need to make our QCheckBox and instance wide variable, and here's a neat little trick. In PyCharm, if you hold ALT, the key on your keyboard, and you just keep Clicking your cursor, it's going to allow you to type in multiple places at once, so I'm just going to do this. Obviously, I had my cursor here, as well, get rid of that. Now, let's create a handler, which is going to be our own custom method, and we'll call this method selected. In this method, we're just going to see if our checkbox is checked or not, so, if I call if self. checkbox. isChecked, this is the method that we use to make sure and see if our checkbox is checked or not, and, if it is, it's going to return True, so this if will evaluate, and, if not, it's going to return a False. In this case, let's just print ("YAY"), and otherwise, we'll print "No" with a sad face. Now let's connect the stage changed event to this handler, self. checkbox. stateChanged. connect(self. selected), and, if we run our application now, watch what happens with the Python console, as I'm checking this and unchecking it. You can see that Yay shows up, and when I uncheck it, No shows up, as well. So, here, we've see the event that the QCheckBox emits or fires, every time you uncheck it or check it, every time you click it, and we've also seen the method on how we can see and make sure whether our checkbox is checked or not.

QComboBox

Another URL element that you will likely use often, is the QComboBox, which is a dropdown box which is a dropdown box that you've probably seen many times. You can see that I've created an instance of it already, but, also notice that I have this selected method at the bottom, which is going to be our event handler, and we'll get to that in a second. However, right now, our QComboBox is empty, so let's add some items to it. We can add items to a ComboBox by calling the addItem method, like so, addItem, and we pass a string as an argument. Let's call this "Item 1". Now, let's add two more items, so I'm just going to copy/paste this code, and this "Item 2" and "Item 3", and, if we run the application, we see our dropdown, and we see our three items are there, just as we told them to be. Now, there is a shorter way for us to add items, and this is especially useful, if you have lots of items to add to QComboBox. Instead of using addItem method, we can use addItems, with an "s" at the end, and this method takes a list of strings as an argument, so we can pass "Item 1", "Item 2", and finally, "Item 3. " If I run the application now, you see that we have the same results as the ones we had before. Obviously, we also want to see what is currently selected, and in that respect, we can check both for the text that's currently

selected, but also the index of that current selection, or the position of the item that is selected in the ComboBox. As such, any time we change the selection in our ComboBox, there is going to be an event called current index changed that is fired. So let's connect that event to our handler here at the bottom. I'll just say self. combobox. currentIndexChanged. connect(self. selected), get rid of the parenthesis at the end, and, in our selected method, we're going to check and see what is the currently selected text, and, also, what is the currently selected index. Let's create two variables. The first one will be called current_text, and to obtain it, we just call self. combobox. currentText, and we'll also have the current_index to get that, as you might assume, we get call the current_Index. Now, since this returns an integer, I'm going to wrap this with a string, so that I can convert it to a string, because we're going to print it both in one line. Let's print that to the Python console. We'll say (current_text + " at the index " + current_index). Now, if I run the application, nothing happens. But why? Well, it doesn't happen because the current_index has not been changed yet. The application has just initialized. If I change the current_index and select, for example, Item 2, we see that it says Item 2 at index 1. Indices are zero-based, just like lists, so actually the first element in the dropdown, which, in this case, would be Item 1, is at index number 0. Item 3 respectively will an index number 2.

Summary

I hope that you enjoyed this little trip down the memory lane. In this module, we kind of went back and revisited some of the methods that commonly-used QWidgets or UI elements have. And, if you look at it, they all make sense. To set the text on a QLabel, call a setText method. To add an item to a QComboBox, call an addItem method. This is also one of the main reasons why I like Python and Qt. It just all seems to make perfect sense, and I hope it also makes sense to you.

What's Next?

What's Next?

Well, you're here! Congratulations once again for completing this course. I hope that you liked it, and I also hope that you saw how powerful both Python and Qt are, even when it comes to desktop application development. So now what? Well, the first thing you should do is go out, celebrate, have some fun, but, of course, I'd strongly recommend that you check out Part Two of this course. In it, we'll learn some more great things about Qt, such as the Qt Designer, an

amazing tool that comes installed with PyQt and will make our lives a whole lot easier. We'll design the Gui, the interface in it, and then, just write the logic behind it. Here we have the application I demonstrated or I used in one of the videos in this course. Other things we'll learn is how to implement icons in our application, or how to perform some intensive tasks, such as indicing all the files in a folder, and how to make our application more beautiful, by just using CSS. Overall, practice makes perfect. Once again, I hope you enjoyed this course, and that you'll check out Part Two of this course, as well. Thank you.

Course author

Bo Milanovich

Originally from Serbia, Bo - Bogdan - started his own software development company at a very young age. He later moved to the United States to pursue studies. As he is finishing up his...

Course info

Level	Beginner
Rating	★★★★½ (499)
My rating	★★★★★
Duration	1h 43m
Released	4 Sep 2014

Share course