Drag and Drop Processing With Python and QT6

Demonstrating the Principle of Operational Laziness

Mark G.

Nov. 8, 2025

This presentation will look at how to write a program in the `Python` programming language using a graphical user interface (GUI) library named `PySide6`. Our example program will accept a string of text, usually an HTTP link, and perform some sort of processing on said string. As part of the architecture of the program, to increase generality and usefulness, we use a shell (Bourne is the example) script which is used to hold the command or commands processing the string. The python GUI program will call this external program and capture its output. We will also take a brief look at `PyInstaller` for creating a distributable, single executable. Tools used are `VSCode` (open source edition) and `Qt Widgets Designer`.

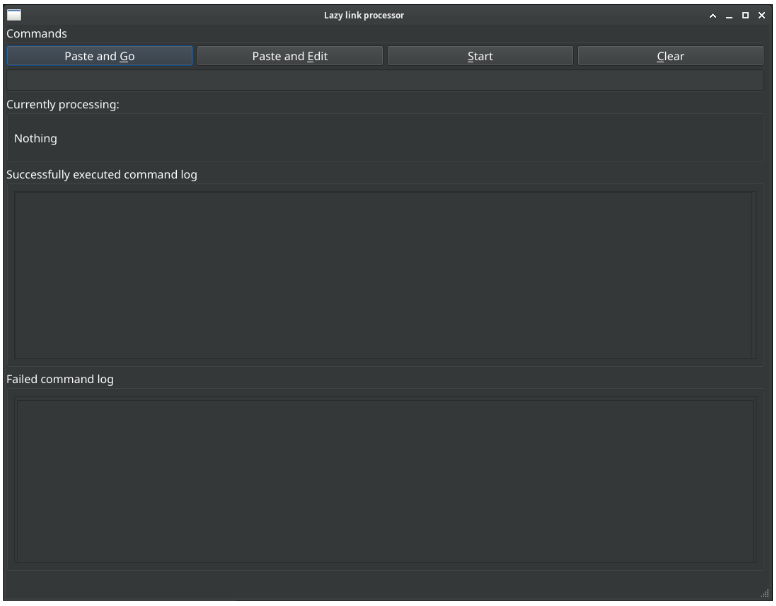

Figure 1: The software we created

Contents

List of Figures

1 Introduction and Requirements

Our goal is to make it easier to do mundane processing that follows this set of actions:

1. Copy some text or a link that you'd like to do something with.
2. Edit the text if needed.
3. Run a command on the text.
4. Observe the command's output.
5. Indicate success or failure.

As a general computer user it behooves me to be as lazy as possible. This goes double for programmers and system administrators. To this end, I will encapsulate the above numbered steps into a program, whose default action is to accept the text via drag and drop, then immediately run the command and capture/display the output. The program will also keep two temporary status lists indicating the success or failure of the command's action on the text.

From this point on, the steps "run the command and capture/display the output, indicate status" will be referred to as `processing`.

The user will have an option to paste-and-edit, allowing for changes to the text before processing. This use case requires a start button to begin processing after edits are done.

If drag and drop is inconvenient, there is a paste-and-go option which immediately copies the text from the paste buffer and starts processing.

2 Development Environment

We'll use two programs in our development efforts. The first is the open-source version of VSCode and the second is called Qt Widgets Designer.

2.1 Open Source VSCode

This software is installed using normal package manager idioms. We use the `pkg` command:

```
mv@think:~ % pkg install vscode
```

After its installation we can check out its information:

```
mv@think:~ % pkg info vscode
vscode-1.103.0
Name           : vscode
```

```
Version        : 1.103.0
Installed on   : Mon Aug 11 17:51:40 2025 PDT
Origin         : editors/vscode
Architecture   : FreeBSD:14:amd64
Prefix         : /usr/local
Categories     : editors
Licenses       : MIT
Maintainer     : tagattie@FreeBSD.org
WWW            : https://code.visualstudio.com/
Comment        : Visual Studio Code - Open Source ("Code - OSS")
...
Annotations    :
FreeBSD_version: 1403000
build_timestamp: 2025-08-11T06:58:54+0000
built_by       : poudriere-git-3.4.2
...
repository     : Poudriere
Flat size      : 390MiB

Description    :
VS Code is a type of tool that combines the simplicity of a code
editor with what developers need for their core edit-build-debug
cycle. It provides comprehensive editing and debugging support, an
extensibility model, and lightweight integration with existing tools.
```

Figure 2 shows a view of the VSCode interface.

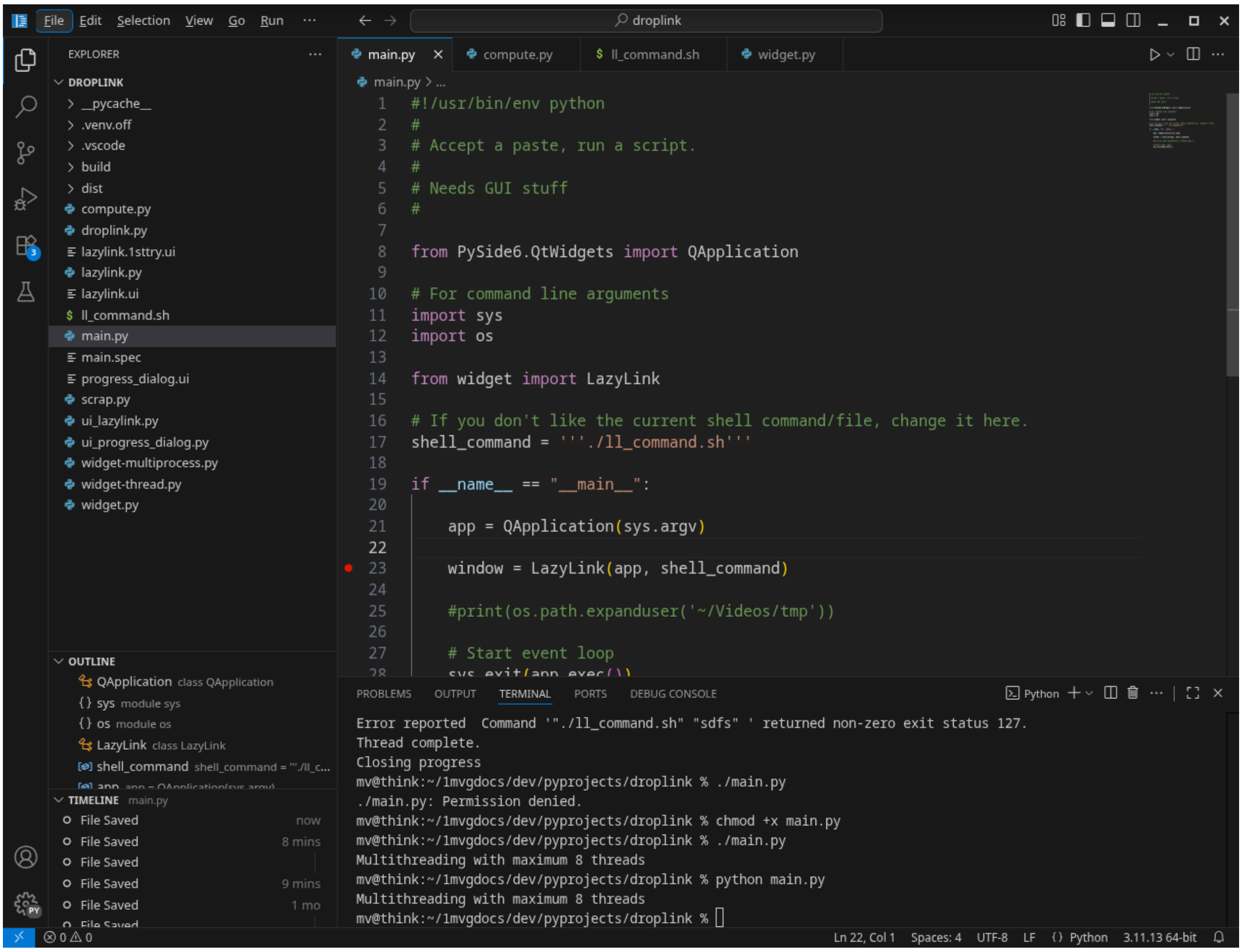

Figure 2: View of Code-OSS (open source VSCode)

2.2 Qt Widgets Designer

The second is Qt Widgets Designer version 6. The purpose of this tool is to allow us to graphically design the user interface and then save this design as a python module for use in our program.

This program is part of the `qt6-tools` package and is installed as a dependency of `py311-pyside6-tools` which itself is installed as a dependency of `py311-pyside6`.

Figure 3 shows a view of the Qt Widgets Designer interface.

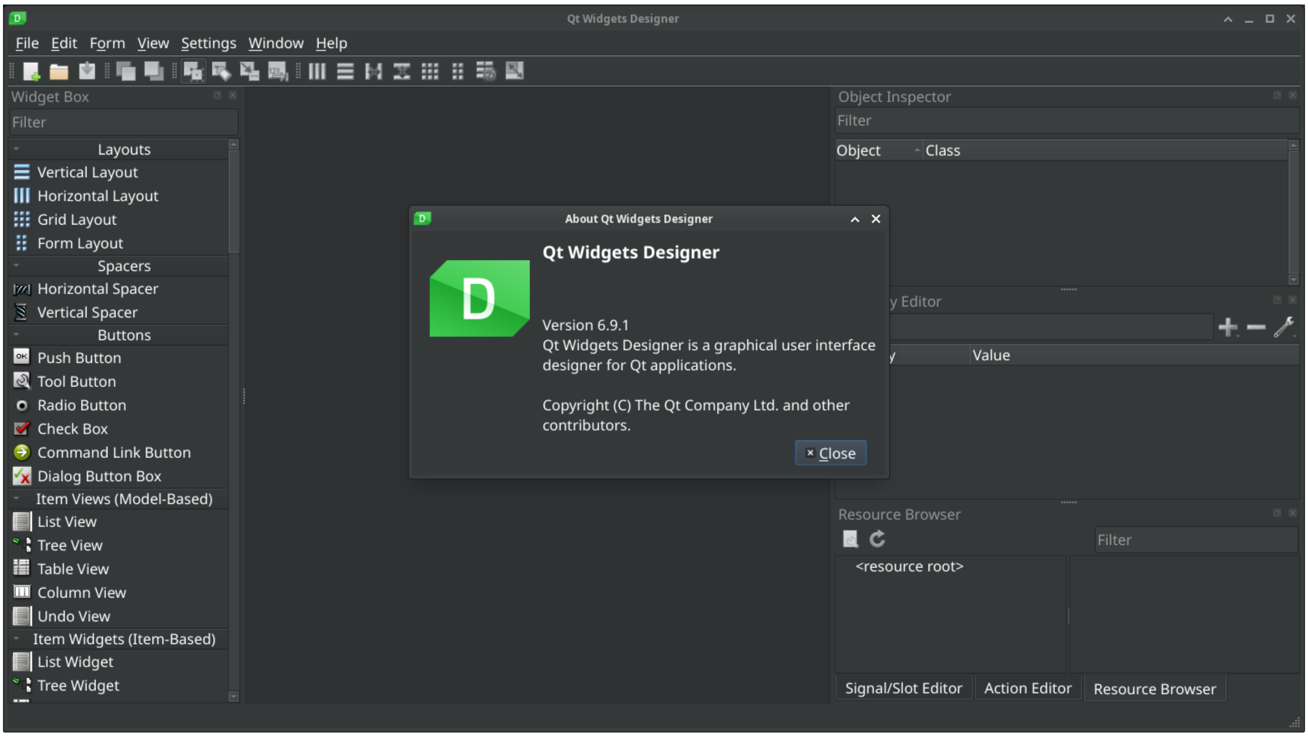

Figure 3: View of the QT Designer 6 Program

The details of installing this on a Linux or Windows system are left as an exercise to the student.

3 Languages and Libraries

We use the Python language for this project and some python libraries.

3.1 Python 3.11

This program was written using version 3.11 of python.

How do we refer to our version of python?

```
mv@think:~ % /usr/bin/env python3
env: python3: No such file or directory


mv@think:~ % /usr/bin/env python
Python 3.11.13 (main, Jul  6 2025, 04:01:02) Clang 19.1.7 on freebsd14
Type "help", "copyright", "credits" or "license" for more information.
>>>

mv@think:~ % /usr/bin/env python3.11
Python 3.11.13 (main, Jul  6 2025, 04:01:02) Clang 19.1.7 on freebsd14
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

So we can be fairly relaxed and use the `python` command directly which is linked to version 3.11.13. Alternatively, we can be more restrictive and use `python3.11` as our python command (which uses the same version). Note there is no `python3` command on this system.

The name of your `python` interpreter will depend on what operating system you are using, of course.

3.1.1 No Virtual Environment?

In this project, I don't use a python virtual environment. This is a FreeBSD specific issue, since `PySide6` wasn't easily install-able into a virtual environment using `pip`. Some needed '`wheels`' of module dependencies could not be created / installed due to missing platform (freebsd) support. This is annoying, and I would prefer a virtual environment, but the `PyInstaller` program allows us to make a single stand-alone executable, making the python environment requirements moot.

However, the `PySide6` software can be installed system-wide using the `pkg` system. A side benefit to installing the `py311-pyside6` package and `py311-pyside6-tools` is that many useful programs are installed as dependencies, in particular, `Qt Widgets Designer 6` and the Qt 6 libraries themselves.

3.2 `PySide6`

While there are a few graphical libraries to choose from, I chose `PySide6`, which uses `Qt` version 6. This is a huge software suite and is tremendously well-documented.

I installed it as follows:

```
# pkg install py311-pyside6 py311-pyside6-tools
```

These two packages install all the needed Qt 6 libraries and programs. Other operating systems will have similar installation mechanisms.

The project's home page is at Qt for Python[1]. Use it regularly and read the details of the widgets and classes you are using.

3.3 `subprocess` module

We'll use threads and create a completely separate process to run our computing workload. This avoids running into problems that the python interpreter has regarding locks (the GIL in particular). New processes, if they are running a python program, use a different instance of the interpreter than the one our main program uses.

To this end, we use the `subprocess` module's `Popen` methodology for executing our commands. It is flexible enough to allow us to run a shell command and to capture the process' output and exit code.

3.4 Bourne Shell

To enhance generality, the `lazylink` program uses a bourne shell command script named `ll_command.sh` (which stands for lazylink command). This is hard-coded in the `main.py` program, but you may replace it by changing the variable in `main.py`:

```
# If you want a different command/file name, change it here.
shell_command = '''./ll_command.sh'''
```

The shell command is combined with the pasted text as its first argument and passed to the `subprocess` module from within a worker thread.

4 Code Patterns

There are a number of code conventions, also know as patterns, used in this project. Generally, we are loosely using a model-view-controller (MVC) pattern. This is an older Object Oriented Design paradigm, but is worth mentioning.

The `model` aspect is implemented in the `compute.py` module, which, instead of using database functions typical in MVC patterns, it provides subprocess/compute functions.

[1] https://doc.qt.io/qtforpython-6/index.html

The `view` aspect consists of the python code generated by the Qt Widget Designer and encapsulates the graphical user interface functionality. Files: `ui_lazylink.py`, `ui_progress_dialog.py`

The `controller` aspect is implemented in the `widget.py` module, which contains all the code for combining the GUI button presses with functions and compute processing. Most of this code is comprised of methods attached to the main window's `widget`.

4.1 Program Execution and Threading

In general, all executed programs have a single main thread. This thread runs the event loop and it deals with button presses and menu actions. It does all the heavy lifting when it comes to updating the GUI screen and controller actions such as receiving and display the input and output via buttons or text boxes.

The programmer should add other threads to execute long running computations so that they won't interfere with the responsiveness or block the main thread. Threading is a common part of GUI programs.

Our program will use a thread pool (`QThreadPool`) to manage and run new threads as needed for our processing. The thread pool is created on our main window, in the class called `LazyLink` within `widget.py`.

The threads we create are commonly referred to as workers.

4.1.1 Workers

As part of Qt's architecture, it includes the `QRunnable` class which is basically a thread. It works in conjunction with thread pools. It is what we use as a base class for our `Worker` class. This can be seen in the `compute.py` module.

Our worker class is essentially a new thread that will communicate with the main thread via signals.

4.1.2 Signals

Qt includes a signaling mechanism which we'll use to tell our main program about the progress and status of our compute workers. We use the Qt `Signal` class for this.

The recommended pattern for signals is to combine them into a single class that our `Worker` class can use. To this end, in `compute.py` we create a `WorkerSignals` class and give it as many `Signal` properties as we need.

Our system uses the following signals:

1. `finished = Signal()`
2. `error = Signal(object) # Exception`

3. `result = Signal(object) # subprocess.CompletedProcess`
4. `output = Signal(str) # stdout`
5. `errors = Signal(str) # stderr`

The argument to `Signal` is the type of data we can pass when the signal is emitted and accepted by the connected function. The Qt architecture refers to the connected functions as `Slots`, but that's as far as I'll go in mentioning them in this presentation.

We don't actually use all of the signals in our program, but the examples I followed had most of these. We use `finished`, `error`, `output` and `result`.

The `errors` signal could have been used if we didn't absorb standard error (`stderr`) into standard output (`stdout`) in our subprocess calls.

Our worker's signals are connected in our main window's `start` method when we create a `worker` for use. They are each connected to a distinct main window method, since they represent different outcomes or data from the thread's subprocess.

NOTE: It is very important to never attempt to update the GUI using code running in a worker thread. The application will crash mysteriously and be very difficult to debug. This is why the signaling pattern implemented here is used.

The worker has this line to incorporate the signals it needs:

```
# In Worker's init
self.signals = WorkerSignals()
```

4.1.3 Other Signals

The above section 4.1.2 describes the signals we use for our threading communication. They are the same conceptually as the otherwise 'normal' signals emitted by button presses and other GUI elements.

However, when a button is created, it comes with many built-in properties and methods, with several signals being among them. The most used button signal is `clicked`, which we can `connect` to a function so as to do something useful.

These signal connections are typically setup in the main window initialization (in `widgets.py`).

5 Graphical User Interface

The program we use for designing the GUI (i.e. the `view`) is called Qt Widget Designer. We can run it from the command line (and send it to the background) via:

```
% designer6 &
```

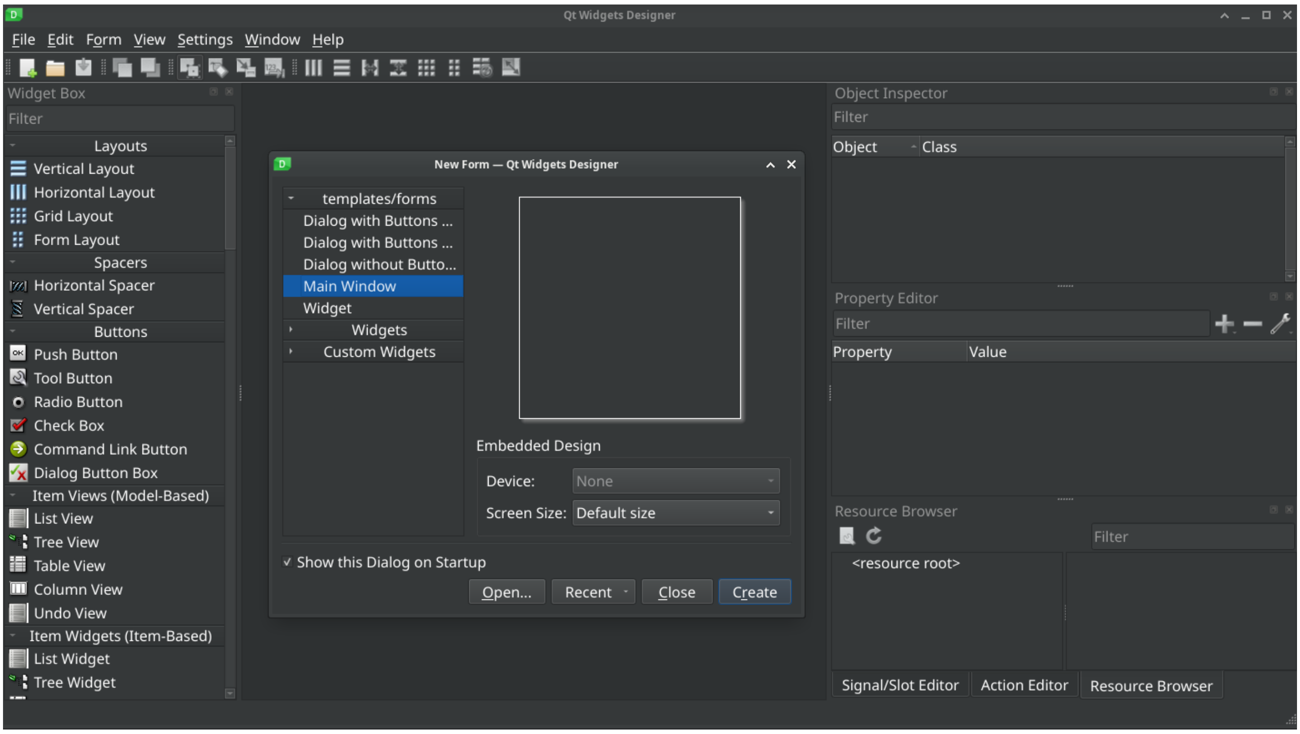

Figure 4: QT Designer 6 Main Window Creation

It is a complex program and has a bit of a learning curve, but there are many useful tutorials on the 'net to get you started.

Once you're happy with how your GUI looks and acts, you then save the GUI layout as python code. Use the `Form->View Python Code` command to display the generated python code for your GUI. Then use the `Save` command at the top of the python code view to save the code to your program's code directory.

Here are a few of my tips I learned from others:

1. Give your widgets names other than the default and use a naming convention. For example, all line edit boxes have names starting with `le_` and labels have names starting with `lb_` and so on. These conventions will make VSCode's auto-complete more useful when working with your widgets.
2. Never edit the generated python code for your GUI. Always make changes using the GUI editor and regenerate the python code.
3. Create a skeleton copy of your layout with the button / text box names and place it in a comment section in your main window widget for easy reference.

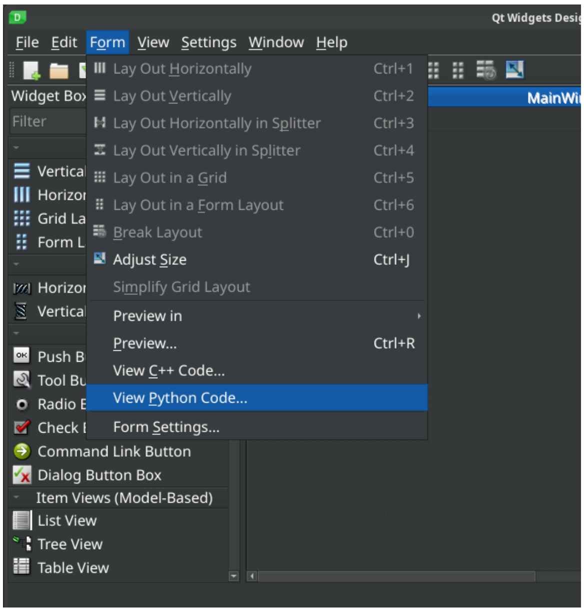

Figure 5: QT Designer 6 Form→ View Python Code

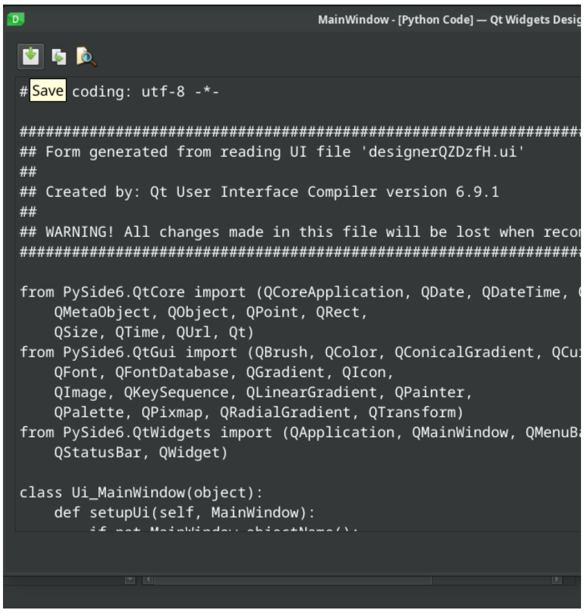

Figure 6: QT Designer 6 Save Python Code

The generated python code will have a main class that you must include as a base class with your `QMainWindow` widget.

My main window's class name is LazyLink and it has base classes of `QMainWindow` and

Ui_MainWindow from the generated file which I named ui_lazylink.py.

Here is a quick look at the top of the ui_lazylink.py code:

```
% cat ui_lazylink.py|more

# -*- coding: utf-8 -*-

################################################################################
## Form generated from reading UI file 'lazylinkoKFMNM.ui'
##
## Created by: Qt User Interface Compiler version 6.9.1
##
## WARNING! All changes made in this file will be lost when recompiling UI file!
################################################################################

from PySide6.QtCore import (QCoreApplication, QDate, QDateTime, QLocale,
    QMetaObject, QObject, QPoint, QRect,
    QSize, QTime, QUrl, Qt)
from PySide6.QtGui import (QBrush, QColor, QConicalGradient, QCursor,
    QFont, QFontDatabase, QGradient, QIcon,
    QImage, QKeySequence, QLinearGradient, QPainter,
    QPalette, QPixmap, QRadialGradient, QTransform)
from PySide6.QtWidgets import (QAbstractScrollArea, QApplication, QGridLayout, QGroupBox
    QHBoxLayout, QLabel, QLayout, QLineEdit,
    QMainWindow, QPushButton, QScrollArea, QSizePolicy,
    QStatusBar, QTextEdit, QVBoxLayout, QWidget)

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        if not MainWindow.objectName():
            MainWindow.setObjectName(u"MainWindow")
...
```

See the widgets.py file for the inclusion of this class in our program.

6 Our LazyLink Software

The software is slightly complicated, but there are common patterns that are used for Qt GUI programs that we have followed.

I used many video tutorials and made extensive use of the PySide6 documentation when writing this software. One cannot do without the documentation[2] at Qt for

[2] https://doc.qt.io/qtforpython-6/index.html

Python.

It is frequently necessary to look up the classes and base classes to find out the properties and methods that are provided for buttons, text boxes, dialogs and so on.

6.1 Main Program

The project uses the common file name of `main.py` to contain the main program functionality: It has a common template structure when using `PySide6` as our graphical library. Mostly, this code is written once and rarely changed.

Below we see the use of the `QApplication` class to instantiate our program in a variable named `app`. This `app` will partner with our main window widget and will then become our event loop when we run `app.exec()`.

```
#!/usr/bin/env python
# Main program for the lazylink program.
# Accept a paste, run a script.
#
# Uses Qt 6 GUI stuff.
#

from PySide6.QtWidgets import QApplication

# For command line arguments
import sys
import os

from widget import LazyLink

# If you want a different command/file name, change it here.
shell_command = '''./ll_command.sh'''

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = LazyLink(app, shell_command)

    # Start the app.exec event loop, report its status via sys.exit
    sys.exit(app.exec())
```

This program can be executed as follows:

```
mv@think:~/.../pyprojects/droplink % python main.py
Multithreading with maximum 8 threads
```

On the other hand, since we have a bang-path (`#!`) statement shown as `#!/usr/bin/env` `python` as the first line in `main.py`, we can try and run the program as `./main.py`:

The example below shows us changing the `main.py` file to be executable (`chmod +x`) after we receive a permission error.

```
mv@think:~/.../pyprojects/droplink % ./main.py
./main.py: Permission denied.

mv@think:~/.../pyprojects/droplink % chmod +x main.py
mv@think:~/.../pyprojects/droplink % ./main.py
Multithreading with maximum 8 threads
```

This is how we can execute the program as we iterate through debugging and feature building.

At the end of the day, we'll use `PyInstaller` (see section 7) to create a file named `lazylink`, and that will be our project's executable file name that we can distribute.

6.2 Widget Code

The code in `widget.py` controls the flow of our program and deals with button presses, worker creation and execution, and the display of processing output and errors.

It has two main purposes, the first to prepare and display the GUI and the second to declare all the functions (methods) that we'll use to perform the program's operations.

```
"""
fullLayout

    gb_commandsBox
        vCommandsLayout
            hButtonsLayout
                pb_pasteGoButton   (paste activate startButton  for processing)
                pb_pasteEditButton (paste activate pastedLink for edit)
                pb_startButton
                pb_clearButton
            le_pastedLink (default empty string)

    gb_currentlyProcessing
        lb_processingLinkValue (default "Nothing")

    gb_successLink "Successfully retrieved:"
      scrollArea, widget
        te_successLog (default empty string) multiline textbox, no edit, scroll

    gb__failedLink "Failed to retrieve:"
      scrollArea, widget
```

```
        te_failureLog (default empty string) multiline textbox, no edit, scroll

"""

import os
import sys
import time

from PySide6.QtCore import QThreadPool

from PySide6.QtWidgets import QMainWindow, QPushButton, QDialog, QDialogButtonBox
from PySide6.QtGui import QGuiApplication

from ui_lazylink import Ui_MainWindow
from ui_progress_dialog import Ui_dw_progressDialog
from compute import Worker

class LazyLinkProgress(QDialog, Ui_dw_progressDialog):
    def __init__(self, parent=None):
        super(LazyLinkProgress, self).__init__(parent=parent)

        self.setupUi(self)
        self.parent = parent

        self.buttonBox.rejected.connect(self.close_dialog) # Close?
        self.buttonBox.button(QDialogButtonBox.StandardButton.Reset).clicked.connect(self.reset_c

    def close_dialog(self):
        print("Closing progress")
        self.te_progressDialog.setText("")

    def reset_contents(self):
        print("Reset progress")
        self.te_progressDialog.setText("")


class LazyLink(QMainWindow, Ui_MainWindow):
    """
    Since this is our main program, we use QMainWindow coupled
    with our custom GUI stuff we created with Qt Widget Designer.

    We have to remember the names that were used in the designer
    for things like buttons, text boxes and so on.

    Important call to self.setupUi from Ui_MainWindow
    should be noted.
    """
    def __init__(self, app, shell_command='./ll_command.sh', parent=None):
        super(LazyLink, self).__init__(parent=parent)
```

```
    self.app = app
    self.shell_command = shell_command

    # Some properties are specified via the UI import.
    # For example, the "paste and go" button has its 'default'
    # property set, and has the '&G' keyboard shortcut
    # set within the button text.  It could be argued
    # that those settings should be stored here for documentation
    # purposes.

    self.setupUi(self)

    self.setWindowTitle("Lazy link processor")

    # We'll use worker threads (QRunnable) in our pool (see start method).
    self.threadpool = QThreadPool()
    thread_count = self.threadpool.maxThreadCount()
    # Realistically, our design only uses one thread at a time.
    print(f"Multithreading with maximum {thread_count} threads")

    # The .clicked.connect lines below allow us to link
    # what happens when a user clicks a button to the code
    # we want to run for that button.
    self.pb_pasteGoButton.clicked.connect(self.paste_and_go)
    self.pb_pasteGoButton.setCheckable(False)

    self.pb_pasteEditButton.clicked.connect(self.paste_and_edit)

    self.pb_startButton.clicked.connect(self.start)
    self.pb_startButton.setCheckable(False)

    self.pb_clearButton.clicked.connect(self.clear)

    self.accepting_drop = False

    # We get our progress dialog ready.  We'll show it later in .start.
    self.progress_dialog = LazyLinkProgress(parent=self)

    """
        failbar = self.scrollArea_2.verticalScrollBar()
        failbar.rangeChanged.connect(self.resize_scroll)

        successbar = self.scrollArea.verticalScrollBar()
        successbar.rangeChanged.connect(self.resize_scroll)
    """
    self.show()

def dragEnterEvent(self, event):
```

```
        self.accepting_drop = False
        if event.mimeData().hasFormat("text/plain"):
            event.acceptProposedAction()
            self.accepting_drop = True


    def dropEvent(self, event):
        print(f"Dropped: {event.mimeData().text()}")
        print(event.mimeData().formats())
        if self.accepting_drop:
            self.le_pastedLink.setText(event.mimeData().text())
            event.acceptProposedAction()
            self.accepting_drop = False
            self.start(True)


    def resize_scroll(self, min, maxi):
        print("Range change", min, maxi)
        self.scrollArea_2.verticalScrollBar().setValue(maxi)


    def paste_and_edit(self, data):
        # Take the paste buffer (clipboard) and allow for edits
        print("Paste and Edit!")
        paste_text = QGuiApplication.clipboard().text()
        self.le_pastedLink.setText(paste_text)
        self.pb_startButton.setDisabled(False)

    def paste_and_go(self, data):
        # Take the paste buffer (clipboard) and process it immediately
        print("Paste and Go!")
        self.pb_pasteGoButton.setDisabled(True)
        self.pb_startButton.setDisabled(True)
        paste_text = QGuiApplication.clipboard().text()
        self.le_pastedLink.setText(paste_text)

        # Our start method where we create workers and
        # begin processing.
        self.start(data)

    # Threading is implemented in this method.
    def start(self, data):
        # Take the submitted text and send it for processing
        print("Start!")
        do_something = False
        command_arg = ""
        if self.le_pastedLink.text().strip() != "":
            do_something = True
            command_arg = self.le_pastedLink.text().strip()
```

```
    elif self.lb_processingLinkValue.text() != "" and self.lb_processingLinkValue.text() != "
        do_something = True
        command_arg = self.lb_processingLinkValue.text().strip()
        self.le_pastedLink.setText(command_arg)


    if do_something:
        self.pb_pasteGoButton.setDisabled(True)
        self.pb_startButton.setDisabled(True)
        self.lb_processingLinkValue.setText(command_arg)
        #self.te_failureLog.append("")

        self.progress_dialog.show()

        self.command = f'''"{self.shell_command}" "{command_arg}" '''
        #self.command = f"""sleep 4"""
        #self.command = f"""exit 4"""

        worker = Worker(self.command)
        worker.signals.error.connect(self.error_callback)
        worker.signals.result.connect(self.callback)
        worker.signals.finished.connect(self.thread_complete)
        worker.signals.output.connect(self.progress_output)
        worker.signals.errors.connect(self.progress_errors)

        # Call our subprocess indirectly here using the worker's
        # run method.
        self.threadpool.start(worker)

    else:
        print("Nothing to do.")
        self.pb_pasteGoButton.setDisabled(False)
        self.pb_startButton.setDisabled(False)

def thread_complete(self):
    print("Thread complete.")
    self.pb_pasteGoButton.setDisabled(False)
    self.pb_startButton.setDisabled(False)

def clear(self, data):
    # Clear the link editing buffer and logs
    print("Clear!")
    self.le_pastedLink.setText("")
    self.te_failureLog.setText("")
    self.te_successLog.setText("")

def error_callback(self, error=None):
    print("Error reported ", error)
    try:
```

```
            text = self.lb_processingLinkValue.text().strip()
            self.te_failureLog.append(text)

        except Exception as err:
            self.te_failureLog.append(self.lb_processingLinkValue.text())
            #self.te_failureLog.append("===")

        self.lb_processingLinkValue.setText("")
        self.pb_pasteGoButton.setDisabled(False)
        self.pb_startButton.setDisabled(False)


    def callback(self, result):
        print("Result reported ", result)
        self.te_successLog.append(self.lb_processingLinkValue.text().strip())
        self.lb_processingLinkValue.setText("")

        self.pb_pasteGoButton.setDisabled(False)
        self.pb_startButton.setDisabled(False)

    def progress_errors(self, line):
        #print("Errors reported")
        self.progress_dialog.te_progressDialog.append(line.strip())
        print(line.strip())

    def progress_output(self, line):
        #print("Output reported")
        self.progress_dialog.te_progressDialog.append(line.strip())
        #print(line.strip())
```

6.3 Pasting Text and Drag and Drop

The Qt framework has excellent built-in support for drag and drop / copy and paste functionality. The code is present in the `widgets.py` file.

We'll break out the bits here for clarity. We need to include `QGuiApplication` which has clipboard methods we can use to extract any text copied to the paste buffer.

```
from PySide6.QtGui import QGuiApplication
```

Example of extracting paste buffer text from the clipboard.

```
    paste_text = QGuiApplication.clipboard().text()
```

The above lines are in the `paste_and_go` method and the `paste_and_edit` method.

For drag and drop, we simply include some methods, as explained by the documentation, in our main window widget:

```
    def dragEnterEvent(self, event):
        self.accepting_drop = False
        if event.mimeData().hasFormat("text/plain"):
            event.acceptProposedAction()
            self.accepting_drop = True


    def dropEvent(self, event):
        print(f"Dropped: {event.mimeData().text()}")
        print(event.mimeData().formats())
        if self.accepting_drop:
            self.le_pastedLink.setText(event.mimeData().text())
            event.acceptProposedAction()
            self.accepting_drop = False
            self.start(True)
```

Drag and drop uses an `event` to pass the text into the program, which we accept and extract the text from if it has the proper mime type. Then we just run our `start` method after putting the text into our pasted link (`le_pastedLink`) field.

Refer to PySide6.QtGui.QDropEvent - Qt for Python[3] for more information.

To be honest, I copied and pasted most of this so my understanding is shallow.

6.4 Compute Code

The `compute.py` module contains two classes used to implement our threading functionality.

The `WorkerSignals` class defines our set of five Qt communication channels using `Signal` as described in section 4.1.2.

```
"""
Contains the computing functionality of the program.

This is the threading version.

"""

import subprocess

from PySide6.QtCore import QRunnable, QThreadPool, Slot, Signal
from PySide6.QtCore import QObject
```

[3] https://doc.qt.io/qtforpython-6/PySide6/QtGui/QDropEvent.html

```
class WorkerSignals(QObject):
    """
    Source: https://www.pythonguis.com/tutorials/multithreading-pyside6-applications-qthreadpool/
    Useful/typical signals emitted from a running worker thread.

    finished
        No arguments

    error
        One argument: Exception object thrown

    errors
        One argument: string from process stderr (not used since we
        pipe stderr to stdout)

    output
        One argument: string from process stdout

    result
        One argument: usually the subprocess.CompletedProcess object returned from processing.
    """

    finished = Signal()
    error = Signal(object)   # Exception
    result = Signal(object)  # subprocess.CompletedProcess

    output = Signal(str)    # stdout
    errors = Signal(str)    # stderr
```

The worker class is run via the thread pool we create in our `QMainWindow` widget. Then the widget's `start` method instantiates the worker and runs it. The `Worker` class uses the signals defined in our `WorkerSignals` class (above).

```
class Worker(QRunnable):
    """
    Requires an argument to pass to the subprocess.Popen (command).

    The command argument will be called directly via subprocess.Popen
    as a shell command.  It's output will be captured
    Be aware that this can introduce risk in the form of arbitrary command
    execution.
    """

    def __init__(self, command):
        super().__init__()

        self.command = command
```

```
        self.signals = WorkerSignals()

    @Slot()
    def run(self):
        """ Run our process """
        print("Now in worker's run method.")
        print(f"Run command as subprocess: {self.command}")

        try:
            with subprocess.Popen(self.command,
                                  shell=True,
                                  bufsize=1,
                                  universal_newlines=True,
                                  stdout=subprocess.PIPE,
                                  stderr=subprocess.STDOUT,
                                  text=True) as subproc:
                for line in subproc.stdout:
                    # send the process' stdout to our signal
                    # processer by using emit on our output signal.
                    # This signal processor is a method on our LazyLink widget
                    # called progress_output in widget.py.
                    self.signals.output.emit(line.strip())

            # The subprocess command receives an error code, check it for
            # non zero, indicating an error.  Re-raise a CalledProcessError.
            if subproc.returncode != 0:
                raise subprocess.CalledProcessError(subproc.returncode,
                                                    subproc.args)


        except subprocess.CalledProcessError as err:
            print("Caught CalledProcessError")
            print("stderr: ", err.stderr)
            self.signals.error.emit(err)

        except subprocess.TimeoutExpired as err:
            print("Caught timeout")
            self.signals.error.emit(err)

        except Exception as e:
            print("Caught unexpected exception")
            self.signals.error.emit(e)

        else:
            print("Process completed with result")
            print(subproc.args)
            #print(f"Output: {result.stdout}\nErrors: {result.stderr}")
            self.signals.result.emit(subproc.args)  # Return the result of the processing
```

```
        finally:
            self.signals.finished.emit()  # Done
```

6.5 Progress Dialog

We also used the Qt Widget Designer to generate a pop-up dialog window for actively displaying the output of our processing command. It is named `LazyLinkProgress` and is found in `widgets.py`. This class has `QDialog` and `Ui_dw_progressDialog` as its base classes and has the following structure:

```
...
from ui_progress_dialog import Ui_dw_progressDialog


class LazyLinkProgress(QDialog, Ui_dw_progressDialog):
    def __init__(self, parent=None):
        super(LazyLinkProgress, self).__init__(parent=parent)

        self.setupUi(self)
        self.parent = parent

        self.buttonBox.rejected.connect(self.close_dialog) # Close
        self.buttonBox.button(
          QDialogButtonBox.StandardButton.Reset
          ).clicked.connect(self.reset_contents) # Reset

    def close_dialog(self):
        print("Closing progress")
        self.te_progressDialog.setText("")

    def reset_contents(self):
        print("Reset progress")
        self.te_progressDialog.setText("")
```

We attach this dialog to our main window and show it when we start running our worker thread. Its text edit box gets filled with the subprocess' standard output via the main window's `progress_output` method. The text edit box is named `te_progressDialog` and is referenced as follows:

```
self.progress_dialog.te_progressDialog.append(line.strip())
```

`self` is the main window in this context. `line` comes from the `output` signal emitted by the executing thread.

We have a Reset button to clear the contents and a standard Close button.

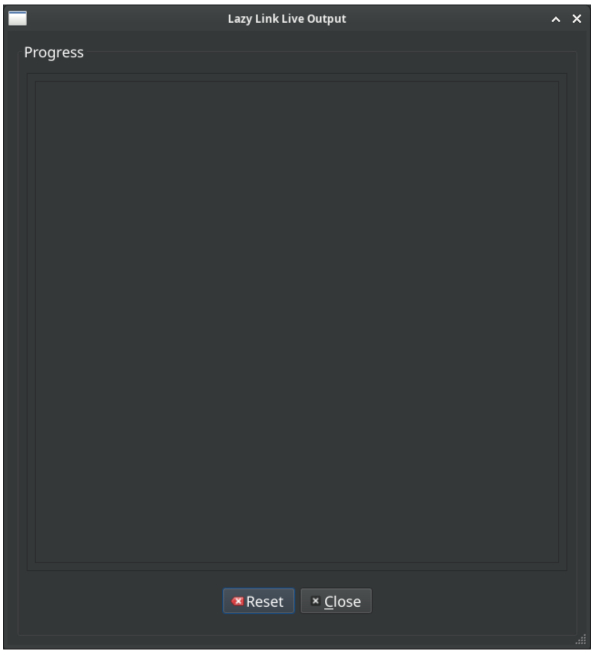

Figure 7: Lazy Link Progress Dialog

6.6 Shell Code

We use an external command file named ll_command.sh to execute as our subprocess.

```
#!/bin/sh +x
# Run this command as our lazy link program
# Totally not secure.
#
# Pass one argument only as the regular parameter, i.e. the URL
# Pass two arguments: the first is a path to prefer for file operations,
#     i.e. the working directory for the command if it accepts one.
#     Does not change directory to this path.
#     The second is the regular parameter as described in the one argument
#     case.
#
sleep 1
# Number of arguments
echo $#
echo "PID: " $$

if [ 2 -eq $# ]
```

```
then
    PREFERRED_DIR="$1"
    shift
else
    PREFERRED_DIR="./"
fi
#PREFERRED_DIR=shift()


echo "Preferred directory: " $PREFERRED_DIR
echo "Remaining arguments: " $@


# Our arbitrary command:
echo "$@"
```

7 PyInstaller Tool

To avoid having to install a bunch of dependencies and python libraries on systems where the `lazylink` program might be useful, we can bundle up all the code into a single executable. This is where `PyInstaller` comes in.

This tool has comprehensive instructions: PyInstaller Manual - PyInstaller 6.16.0 documentation[4] It can be installed system-wide using `pip` (note again, we are not using a virtual environment or the `pip -U` form for a user installation):

```
# pip install pyinstaller
```

For the first run of `pyinstaller` we execute this command to create an initial executable as well as a '`spec`' file. We are in the program's code directory, of course.

```
% pyinstaller --onefile --windowed --name='lazylink' main.py
```

```
156 INFO: PyInstaller: 6.16.0, contrib hooks: 2025.8
156 INFO: Python: 3.11.13
167 INFO: Platform: FreeBSD-14.3-RELEASE-p5-amd64-64bit-ELF
167 INFO: Python environment: /usr/local
168 INFO: wrote /home/mv/1mvgdocs/dev/pyprojects/droplink/lazylink.spec
174 INFO: Module search paths (PYTHONPATH):
['/usr/local/lib/python311.zip',
 '/usr/local/lib/python3.11',
 '/usr/local/lib/python3.11/lib-dynload',
 '/usr/local/lib/python3.11/site-packages',
 '/home/mv/1mvgdocs/dev/pyprojects/droplink']
<frozen importlib._bootstrap>:241: RuntimeWarning: Your system is avx2
 capable but pygame was not built with support for it. The performance of
 some of your blits could be adversely affected. Consider enabling compile
 time detection with environment variables like PYGAME_DETECT_AVX2=1 if
 you are compiling without cross compilation.
pygame 2.6.1 (SDL 2.32.8, Python 3.11.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

[4] https://pyinstaller.org/en/stable/

```
712 INFO: checking Analysis
...
15074 INFO: checking PKG
15074 INFO: Building PKG because PKG-00.toc is non existent
15074 INFO: Building PKG (CArchive) lazylink.pkg
30712 INFO: Building PKG (CArchive) lazylink.pkg completed successfully.
30718 INFO: Bootloader /usr/local/lib/python3.11/site-packages/PyInstaller/bootloader/FreeBSD-64bit/run
30718 INFO: checking EXE
30718 INFO: Building EXE because EXE-00.toc is non existent
30718 INFO: Building EXE from EXE-00.toc
30718 INFO: Copying bootloader EXE to /home/mv/1mvgdocs/dev/pyprojects/droplink/dist/lazylink
30719 INFO: Appending PKG archive to EXE
30740 INFO: Building EXE from EXE-00.toc completed successfully.
30747 INFO: Build complete! The results are available in: /home/mv/1mvgdocs/dev/pyprojects/droplink/dist
```

The runnable program is placed in a `./dist` folder and is usually named `main`, but since we specified the `--name` option, it is called `lazylink` instead. The created spec file is also named `lazylink.spec`.

```
% ls -la dist/
total 136355
drwxr-xr-x  2 mv mv        6 Nov  7 17:29 .
drwxr-xr-x  7 mv mv       24 Nov  7 17:29 ..
-rwxr-xr-x  1 mv mv 46534614 Nov  7 17:29 lazylink
-rwxr-xr-x  1 mv mv      873 Nov  7 17:15 ll_command.sh
```

NOTE: I had to manually copy the shell script `ll_command.sh` into the `dist/` folder as `pyinstaller` isn't aware of its use in the program.

The `lazylink.spec` file can now be used with `pyinstaller` to make regeneration of the single executable easier:

```
% pyinstaller lazylink.spec
```

We can also edit the `lazylink.spec` file to change the name of the executable file from the default of `lazylink` to something else, such as `lazylink.v1` if desired.

```
% cat lazylink.spec

# -*- mode: python ; coding: utf-8 -*-


a = Analysis(
    ['main.py'],
    pathex=[],
    binaries=[],
    datas=[],
    hiddenimports=[],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=[],
    noarchive=False,
```

```
    optimize=0,
)
pyz = PYZ(a.pure)

exe = EXE(
    pyz,
    a.scripts,
    a.binaries,
    a.datas,
    [],
    name='lazylink',
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=False,
    disable_windowed_traceback=False,
    argv_emulation=False,
    target_arch=None,
    codesign_identity=None,
    entitlements_file=None,
)
```

To use the `lazylink` program and its `ll_command.sh` companion, use your favourite copy command (I use `scp`) to move the program and its command to your other FreeBSD systems.

I copied the large 46MB `lazylink` file to my home folder's bin directory so it can be found in the `PATH`.

NOTE: The `lazylink` program will look for the `ll_command.sh` script in your current working directory, so placing `ll_command.sh` in your `PATH` won't work.

This is a feature. Really.

One last thing, the executable `pyinstaller` creates is platform specific, so if you want a `Windows` executable, you have to use `pyinstaller` on a `Windows` system. Same for Linux and MacOS.

8 Conclusion

There is no end to laziness.

Well, actually, I did have to learn a bunch of stuff to realize my laziness, so there is that.

9 Resources

Here is a list of videos that I used to get started and learn how to use the tools, languages and libraries used in this project.

The channels that these videos belong to contain many other useful tutorials and reference material. The ones chosen below are good examples I found particularly useful.

1. GUI Frameworks for Python

   https://www.youtube.com/watch?v=pCXcQU-aMYs

   From the description of the video: "Aug 27, 2025 In this video I give an overview of the most common GUI frameworks for Python"

   The author describes tkinter, wxPython, PyQT and PySide. I chose PySide mostly because of this useful video.

2. PySide6 Crash Course: GUI Development in Python with Qt6

   https://www.youtube.com/watch?v=9_NGCpM2r7s

   From the description: "Aug 11, 2025 In this video, we do a crash course on PySide6, which is a modern framework for developing graphical user interface (GUI) applications in Python with Qt6. What you learn in this video can easily also be applied to PyQt6."

   I used the above video as a way to get a fairly complete overview of the Qt development process. It also introduces Qt Widget Designer, which is what I choose to use to create my GUI visually. The video focuses on creating the GUI programatically, however, so it is a good reference, but I needed further examples using the designer.

3. PySide6 Widgets Tutorial - Ep07 - Signals and slots

   https://www.youtube.com/watch?v=iU1wbOriwIw

   From the description: "Nov. 8, 2022 Use signals and slots to respond to things happening in your Qt Widgets GUI applications - Pyside6 Widgets tutorial"

   Provides insight into linking signals such as button presses (and similar) with a function (slot) to perform an action.

4. Creating GUI threads with Python PySide 6 and QThreadPool

   https://www.youtube.com/watch?v=Vh0y8ZrlX4w

   From the description: "Mar 31, 2025 ... The aim in this video is to show how you can use QThreadPool to ensure the application continues to be responsive when performing other operations. This is not the only option but some way of using an alternative thread or process is needed for most GUI applications."

This video taught me the basics of using the Worker class and thread pools.

5. Using Qt Designer files in PySide6 or PyQt6

   https://www.youtube.com/watch?v=MfCOJltTSCk

   From the description: “Dec 19, 2022 This video will show you the correct way to use Qt Designer UI files in either PySide6 or PyQt6.”

   This video showed me how to use Qt Widget Designer in a workflow for visually creating my GUI and gave me insight into how the generated code fits into my main window. Note that QWidget was used in his example instead of QMainWindow in our program.