*Python Pro Programming Course*
*Module 5. Desktop Applications.*
*Introduction to PyQt5.*

# Theory

**Contents:**



## Basic Concepts

A **windowed application** is a program that uses the elements of a graphical interface (such as buttons, windows, and radio buttons) to interact with the user. With the help of input devices (keyboard, mouse, touchpad, etc.), the user can interact with the objects in the windowed application: move them, activate them, scroll them.

A **widget** is a special element of the user interface that displays information or lets the user interact with the operating system or application.

Examples of widgets are labels, buttons, and radio buttons.

**PyQt5** is a cross-platform library for creating applications. If a library is cross-platform, that means an application created in PyQt5 will open equally well in any operating system.

### Importing Modules

The PyQt5 library has many ready-to-use modules and functions. To import a module for a widget, use the familiar command:

```
from PyQt5.QtWidgets import widget1, widget2
```

Here are some of the widgets you can import:

| Widget | Designation |
|---|---|
| Application | QApplication |
| Application window | QWidget |
| Label | QLabel |
| Button | QPushButton |
| Radio button | QRadioButton |
| Group of buttons* | QButtonGroup |

* The buttons might be of different types.

### Creating an Application

As we begin development, we have to create the application itself. This is done by a special method, QApplication([]), which returns an object of type “application”:

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

Immediately after creating the application, we must write the command app.exec_() at the end of the program. This is a standard command that will leave the application open until the user clicks the exit (red X) button.

### Application Window

To create an application window, use the command *QWidget()*, which returns an object of type "window":

```
my_win = QWidget()
```

A window has a set of parameters that can be changed:

| Method | Designation |
|---|---|
| `my_win.setWindowTitle('Title')` | Set window title |
| `my_win.move(900, 70)` | Make window appear in the indicated spot (instead of in the center of the screen) |
| `my_win.resize(400, 200)` | Change the window's dimensions |
| `my_win.show()` | **Make the window visible** |
| `my_win.hide()` | Hide the window |

### Creating Widgets

To add a widget to a window, we must do several things in order. First, we must create the corresponding widget object:

| Widget | Method |
|---|---|

| | |
|---|---|
| Label | `title = QLabel('Hello, world!')` |
| Button | `button = QPushButton('Confirm')` |
| Radio Button | `radiobutton = QRadioButton('Answer Option 1')` |
| Group of Buttons | `buttongroup = QButtonGroup()` |
| Text Field | `line = QLineEdit('A hint for the user')` |

After the corresponding object is created, we can position it in the window (see next section).

**Positioning Widgets in the Window**

The application elements we create must be positioned in the window. Guiding lines are a convenient tool to use for this purpose. To use them, we must import the corresponding modules:

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import other widgets, QHBoxLayout, QVBoxLayout
```

where QHBoxLayout is a horizontal guiding line, and QVBoxLayout is a vertical guiding line.

For example, to position the application elements vertically, we need to create a vertical guiding line, add widgets to it (and, if we want, align them to the center, the left edge, etc.), and add the line to the application window:

| *Method* | *Designation* |
|---|---|
| `v_line = QVBoxLayout()` | **Creating a vertical line** we can use to align the elements of the application |
| `h_line = QHBoxLayout()` | **Creating a horizontal line** (same purpose) |

Adding an object to a line:

| *Method* | *Designation* |
|---|---|
| `v_line.addWidget(title, alignment = Qt.AlignCenter)` | Add a label to a vertical line. Align the label to the center. |

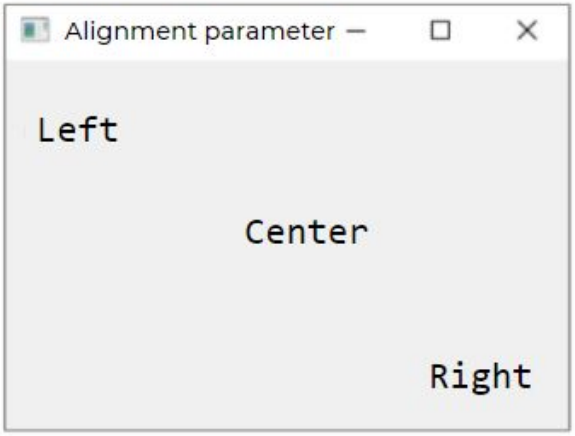
The *alignment* parameter in the *addWidget* method aligns the components along a guiding line.

This parameter can have the following attributes:

- ★ **AlignLeft** — horizontal alignment along the left edge
- ★ **AlignRight** — horizontal alignment along the right edge
- ★ **AlignCenter** — horizontal alignment to the center
- ★ **AlignBottom** — alignment along the bottom edge
- ★ **AlignTop** — alignment along the top edge

Example:

```
l = QHBoxLayout()
l.addWidget(label1, alignment = Qt.AlignLeft)
l.addWidget(label2, alignment = Qt.AlignRight)
l.addWidget(label3, alignment = Qt.AlignCenter)
```

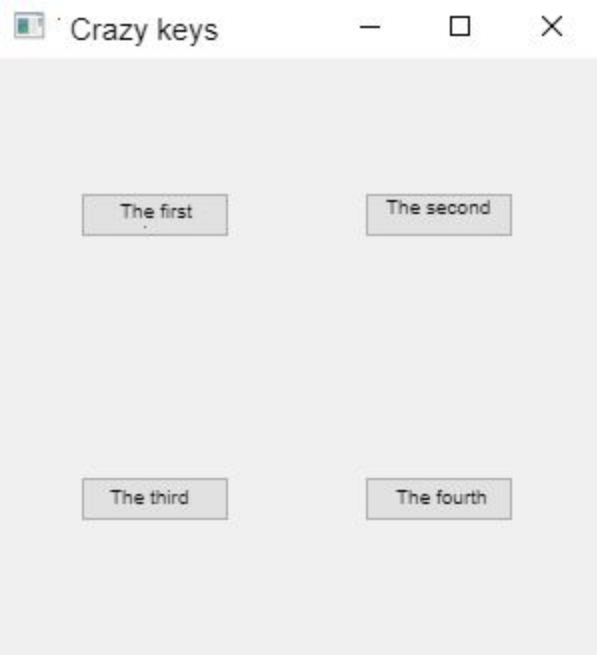



To position a guiding line in a window along with all of the attached widgets:

| *Method* | *Designation* |
|---|---|
| `my_win.setLayout(v_line)` | Add the resulting line and its objects to the application window. |

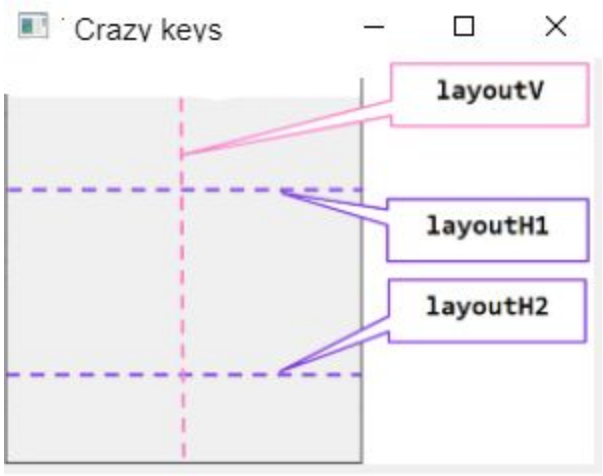
To create windows with a complicated design, we can use several guiding lines that are attached to one another as the window is created. To do that, we must create baselines, position objects on them, and then attach all the baselines (and the objects attached to them) to the main baseline. The main baseline is, in turn, positioned inside the window.

Let's examine an example where objects must be positioned as they appear in this illustration:




Skipping over the part where we import the modules, let's create the applications and windows and immediately move on to creating the objects:

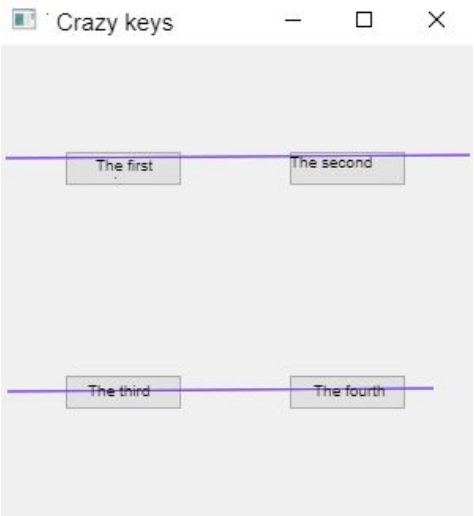
```
#Create button objects
button1 = QPushButton('First')
button2 = QPushButton('Second')
button3 = QPushButton('Third')
button4 = QPushButton('Fourth')

#Create 1 vertical and 2 horizontal lines
layoutV = QVBoxLayout()
layoutH1 = QHBoxLayout()
layoutH2 = QHBoxLayout()
```




Let's position the buttons along the horizontal guiding lines:

```
#Add a button_ object to the central line
layoutH1.addWidget(button1, alignment = Qt.AlignCenter)
layoutH1.addWidget(button2, alignment = Qt.AlignCenter)
layoutH2.addWidget(button3, alignment = Qt.AlignCenter)
layoutH2.addWidget(button4, alignment = Qt.AlignCenter)
```

The buttons button1 and button2 are positioned along the horizontal guiding line layoutH1. The buttons button3 and button4 are positioned along the horizontal guiding line layoutH2:




Next, we must attach the horizontal guiding lines to the vertical one:

```
layoutV.addLayout(layoutH1)
layoutV.addLayout(layoutH2)
window.setLayout(layoutV)
window.show()
```

**Creating Radio Buttons**

A **radio button** is an element of the interface that allows the user to pick one option (item) from a predetermined set (group).

To work with radio buttons, we must import the corresponding module (see Creating Widgets) and create radio button objects. It's important to note that a radio button object is a single radio button. If you're trying to create, for example, three answer options to a test question, you must create three radio button items:

```
# create radio button objects
radio_button_1 = QRadioButton('1')
radio_button_2 = QRadioButton('2')
radio_button_3 = QRadioButton('3')
```

It's often necessary for some button to already be selected when the program is launched. To do this, use the setChecked method, which will give the "selected" state to the indicated radio button:

```
#create radio button objects
radio_button_1 = QRadioButton('1')
radio_button_2 = QRadioButton('2')
radio_button_3 = QRadioButton('3')
# establish which radio button will be selected when the program is launched
radio_button_1.setChecked(True)
```

To unite the radio button objects and make it possible to choose only one of the answer options, you must put the radio buttons into a button group. For this, you must import the module QButtonGroup, create a “button group” object, and add the necessary radio buttons to the group. Take a look at the id field: here, you can give a unique number (identifier) to each button so that, later, you will be able to refer to this radio button at any point in the program:

```
# create a group of radio buttons and add in the radio button objects we've already created
button_group = QButtonGroup()
button_group.addButton(radio_button_1, id = 1)
button_group.addButton(radio_button_2, id = 2)
button_group.addButton(radio_button_3, id = 3)
```

Position the radio buttons on the correct guiding lines to see the result on the screen.

To learn which radio button is selected at any given time, use the following method, which returns the unique identifier (the number from the id field) of the corresponding button:

```
button_group_name.checkedId()
```

**Setting and Getting Widget Text**

To change the widget text while the program is executing, use the following method:

```
widget_object_name.setText("New text")
```

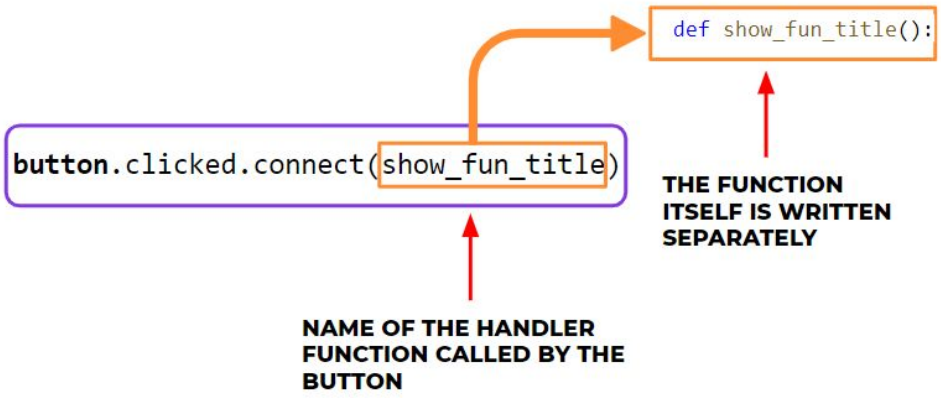
To get the widget text (for example, the text entered by a user into an entry field), use the following method:

```
s = widget_object_name.text()
```

### Handling a Button Click Event

Once a button has been created, it must be made active. If we launch the program and click the button right now, nothing will happen. When a control element (button) reacts to an event in the outside world (a mouse click), this is called event handling.

A program's actions in response to a button click are often described by a separate function. Let's imagine that the actions of the program are described by the function show_fun_title(). We'll call it a handler function. To launch the function show_fun_title() in response to a button click, we will use the command button.clicked.connect(show_fun_title). In other words, we are saying "when the button called 'button' is clicked, launch the function show_fun_title."




### Working with Fonts

To work with fonts, import the module QFont:

```
from PyQt5.QtGui import QFont
```

The size, weight and font of the text in a widget can be changed like this:

```
widget_object_name.setFont(QFont("Times", 36, QFont.Bold))
```

The color of the widget text can be changed as follows, where (0,0,0) is the RGB color code:

```
widget_object_name.setStyleSheet("color: rgb(0,0,0)")
```

**Restricting What Can be Entered Into a Text Field**

To work with text fields, import the module QLineEdit:

```
from PyQt5.QtWidgets import QApplication, QWidget,..., QLineEdit

line = QLineEdit("Hint")
```

Qt contains a QValidator class that checks the correctness of the data entered. This class cannot be used directly. To check your data, you will have to use the subclasses QIntValidator (integers) and QDoubleValidator (fractional numbers). The QLocale module is also helpful because it allows you to set the correct language for your data. This way, you can restrict the data entered as follows:

```
from PyQt5.QtWidgets import QApplication, QWidget,..., QLineEdit
from PyQt5.QtGui import QDoubleValidator, QIntValidator
from PyQt5.QtCore import QLocale

# set the country's language
loc = QLocale(QLocale.English, QLocale.UnitedStates)
validator = QDoubleValidator()
validator.setLocale(self.loc)

line = QLineEdit("Hint")
# set a restriction: test only numbers
line.setValidator(validator)
# set a range of acceptable values
line.setValidator(QIntValidator(0, 150))
```

**Timer**

To work with a timer and time, import the following modules:

```
from PyQt5.QtCore import QTimer, QTime
```

Next, create an object of the QTime class, which can be used to set the standard time. The first number in the parameters represents the hours, the second represents minutes and the third represents seconds:

```
from PyQt5.QtCore import QTimer, QTime
time = QTime(0, 0, 15)
```

The next step is to create an object of the Timer class. This instrument will launch a specified function at a specified time interval:

```
from PyQt5.QtCore import QTimer, QTime
time = QTime(0, 0, 15)

# create an object of type Timer
timer = QTimer()
# connect it to a function
timer.timeout.connect(timerEvent)
# set the intervals at which the function is launched
timer.start(1000)
```

The function connected to the timer can contain instructions for what the timer should do every time this function is called. This must involve a change in the time, and the instructions must also indicate under what conditions the timer should stop:

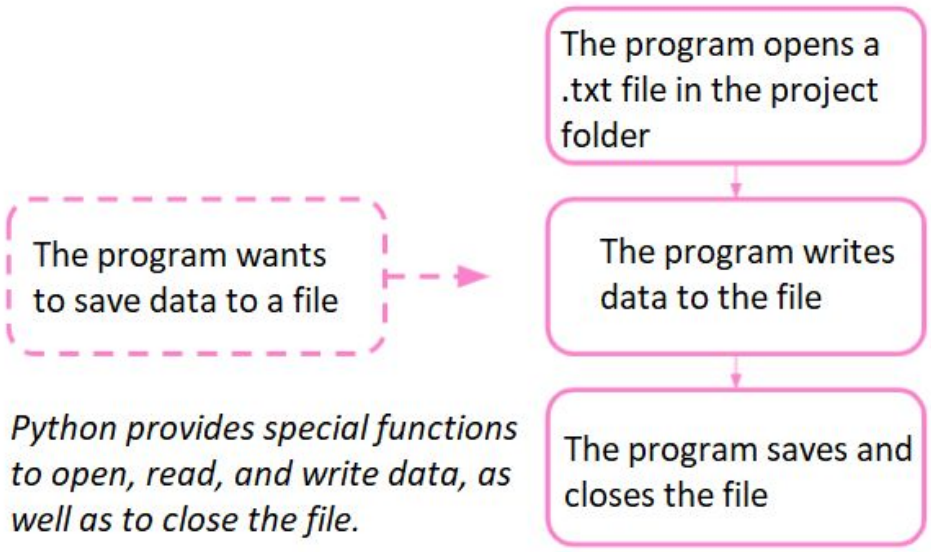
```
from PyQt5.QtCore import QTimer, QTime

def timerEvent(self):
    # subtract one second of time every time the function is called
    time = time.addSecs(-1)
    # stopping conditions - when the time is exactly 00:00:00
    if time.toString("hh:mm:ss") == "00:00:00":
       self.timer.stop()
```

```
time = QTime(0, 0, 15)
# create an object of type Timer
timer = QTimer()
# connect it to a function
timer.timeout.connect(timerEvent)
# set function launch intervals of 1 second
timer.start(1000)
```

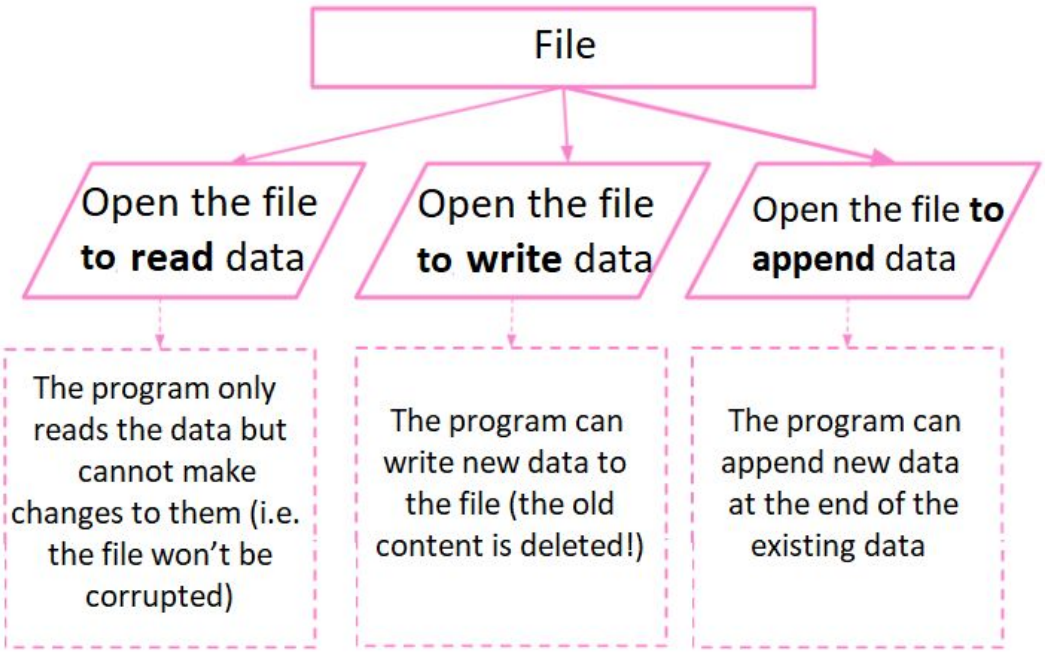
**Entering and Getting Data (.txt format)**

Data can be entered and extracted using text files. A text file can be created using the "Notebook" text editor or the VS Code development environment. The file must be in the same folder as the program. The program might use the file in different ways: as a source of information, a place to store data, or both.

The program's interaction with the text file can be mapped out as follows:




A file can have several modes of access:




Using various access attributes, we can open a file for various purposes:

| *Function Designation* | *Function in Python* |
|---|---|
| Open the file to read | `file = open("notes.txt", "r")` |
| Open the file to write<br>(old contents will be deleted!) | `file = open("notes.txt", "w")` |
| Open file to append (information added to the end of the file) | `file = open("notes.txt", "a")` |

To read the data from a file, we will need the following functions:

| *Function Designation* | *Function in Python* |
|---|---|
| Open file to read | `file = open("notes.txt", "r")` |
| Read data from file | `data = file.read()` |
| Read data from file<br>(in parts) | `data = file.read(1024)` |

| | |
|---|---|
| Close file when finished working | `file.close()` |

Here's an example. The file poem.txt contains a poem:

*files_program.py*

```
file = open("poem.txt", "r")
data = file.read(1)
print(data)
file.close()
```

*poem.txt*

I love the pomp of Nature's fading dyes,
The forests, garmented in gold and purple,
The rush of noisy wind, and the pale skies
Half-hidden by the clouds in darkling billows,
And the rare sun-ray and the early frost,
And threats of grizzled Winter, heard and lost

Launching the code on the left causes the contents of the file to be displayed on the screen as follows:

*poem.txt*

I love the pomp of Nature's fading dyes,
The forests, garmented in gold and purple,
The rush of noisy wind, and the pale skies
Half-hidden by the clouds in darkling billows,
And the rare sun-ray and the early frost,
And threats of grizzled Winter, heard and lost

To write information into a file, we need the following functions:

| *Function Designation* | *Function in Python* |
|---|---|
| Open file for writing | `file = open("notes.txt", "w")` |
| Write data into the file | `file.write("Information")` |

| | |
|---|---|
| Close file | `file.close()` |

For example:

files_program.py

```
#let's see the result
file = open("poem.txt", "a")
file.write("\nR.Frost")
file.close()
file = open("poem.txt", "r")
data = file.read()
print(data)
```

poem.txt

```
Whose woods these are I think I
know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with
snow.
My little horse must think it queer
To stop without a farmhouse near
```

When this code is executed, the contents of the file poem.txt will be deleted and replaced with the given line. If we want to add the new information to the file, we must change the mode in which we access the file:

files_program.py

```
file = open("poem.txt", "a")
file.write("\nR.Frost")
file.close()
#let's see the result
file = open("poem.txt", "r")
data = file.read()
print(data)
```

To add new data to the existing data, you need to adjust the access attribute!

The code for opening/closing a file and reading/writing information takes up a lot of space.

For example:

```
file = open("quotes.txt", "r")
data = file.read()
file.close()
print(data)
author = input( "Who was the author?" )
file = open("quotes.txt", "a")
file.write("("+author+")"+"\n")
file.close()
```

There is an alternative command for **opening and automatically closing** a file. The code in the program above could be written like this:

```
with open("quotes.txt", "r") as file:
    for line in file:
        print(line)
author = input("Who was the author?")
with open("quotes.txt", "a") as file:
    file.write("("+author+")"+"\n")
```

Use the command "Open the file to read data"

Once the end of the block is reached, the file is closed automatically.

Notice how the "for" cycle makes it convenient to read the file line by line.