# How to create a Typing Test application on Python

## Intro

Some jobs, which deal with inputting text data to a computer, require fast typing skills. If you're not sure whether you type fast or slowly, there are special programs to measure your speed.

Despite the fact that lots of them are freely available on the web, this article is about building such kind of program by your own. If you would like to create your unique customized product, this guide is for you. Actual aim of the guide is educational: following it, you can train your programming skills, such as building Graphical User Interface (GUI)[[1]](#endnote-1), time counting, and applying the method of multithreading.

## Before we start

Writing this article, I expect that a reader has basic skills in coding in **Python**, including **Object Oriented Programming**. As for software, you need to have **PyCharm** installed on your computer.

## Features

What the program should consist of? What experience we want a user to have while taking the test? Let's figure everything out.

At least, your application should satisfy the following requirements:

* A user is able to see target[[2]](#endnote-2) text and retype it at the same time
* There is a timer to measure how long a user retypes a text
* Program gives feedback to a test taker, which consists of three variables: time spent on typing, typing speed and error rate.

Besides the minimal requirements, I suggest you to add some extra features for a user's convenience.

Optional requirements:

* The application is provided with GUI
* A user is able to choose
  + a text for retyping from a set of texts
  + session duration
* A session is finished when time is over or entered[[3]](#endnote-3) text is completely the same as its target text
* The application not only measures a user's typing speed, but also estimates whether it's high or low.

If your view of how the program should look like and behave is slight different from mine, you are free to customize your program requirements as you see them.

## Tools for development

We will build the application with Python programming language, PyCharm IDE. To lay out GUI, we will use a tool called Qt Designer and a special utility for converting from .ui format to .py. The utility comes along with source code files.

Libraries used in the project:

|  |  |
| --- | --- |
| **Library** | **Its Role in the Project** |
| PyQT5 | Working with GUI |
| time | Counting elapsed time |
| threads | Providing simultaneous operations |
| fuzzywuzzy | Evaluating entered text accuracy |

## Steps of development

### Create user-friendly GUI

Think how you see a user's interaction with your application.

First of all, GUI must make it possible to provide all that features which are described above.

As for visual aspect, it's a good idea to make it neat and attractive so that customers could enjoy using the application. Here are some considerations which could make your GUI more user-friendly:

* No extra widgets. There should be as few widgets as possible for appropriate work
* To provide higher user's efficiency, target text area and input field should be placed close to each other. While taking the test, a user has to switch his or her look between two texts. The eye track amplitude should be minimal; otherwise, test result can get worse
* Funny pictures, such as emojis[[4]](#endnote-4), amuse a test taker and cause positive emotions. In my project I've used images of emojis (in PNG format, not as characters) to encourage a user for their result.

|  |  |
| --- | --- |
| **Image** | **Meaning** |
| C:\Users\andreeva.d\Desktop\Privates\Хранилище\Devsoft\pics\Congrats.png | The user's result is excellent |
| C:\Users\andreeva.d\Desktop\Privates\Хранилище\Devsoft\pics\Not_Bad.png | The user's result is quite good |
| C:\Users\andreeva.d\Desktop\Privates\Хранилище\Devsoft\pics\Needs_improving.png | User's typing skills need improvement |

### Design your GUI

The simplest and the most time-saving way to lay out GUI is WYSIWYG (what-you-see-is-what-you-get)[[5]](#endnote-5) method. You will have to use third-party software, because PyCharm IDE doesn't provide this feature. The special tool for it is Qt Designer.

(Download software).

Create a project. There is a **Widget Panel** on the left side. Choose a widget from there which you want to add, drag it, and then drop on your GUI area.

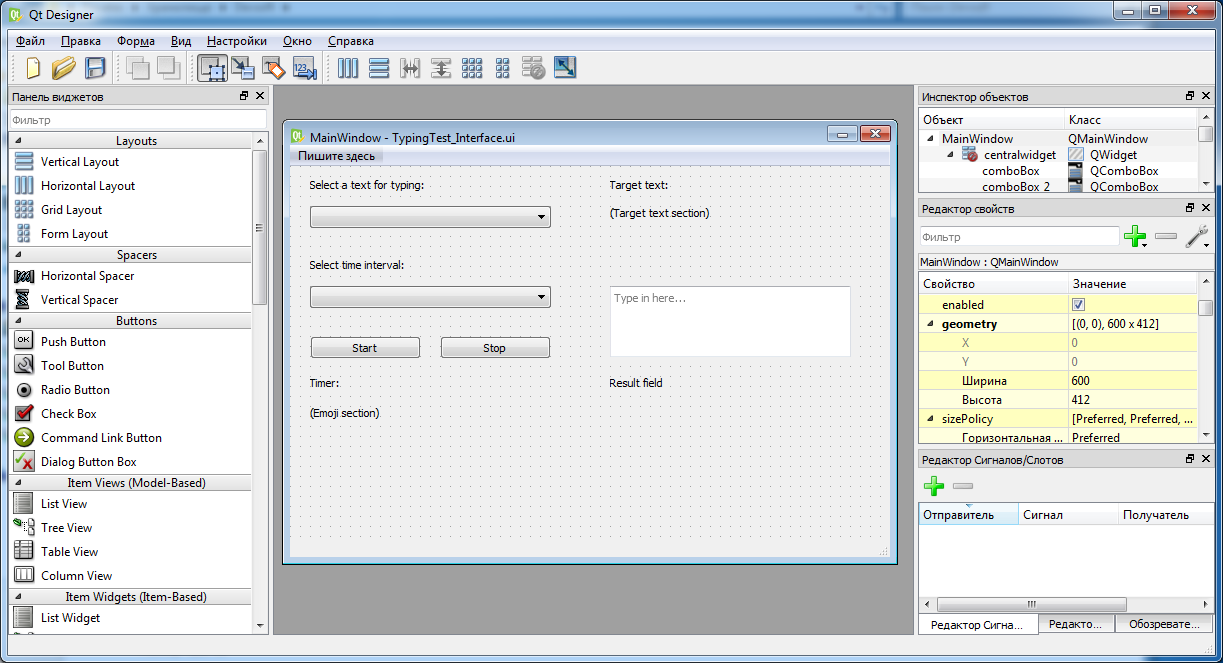


Fig.1. Qt Designer interface

The following is a list of widgets that I used in my GUI.

|  |  |  |  |
| --- | --- | --- | --- |
| **Widget's keyword** | **Quantity** | **Reader-friendly name** | **Its Role in GUI** |
| **label** | 7 | label | to provide a user with some text information. There should be at least labels for displaying a target text, live timer and a session's result |
| **comboBox** | 2 | drop-down list | to give user a choice of texts and time intervals |
| **pushButton** | 2 | button | to start a session or to interrupt it before a timer stop |
| **textEdit** | 1 | input field | for user's text entering |

On the right side there is **Property Editor**. When a widget is selected, you can set a value to a property from the list. For example, after selecting a label or a button and setting a message to the **text** property, the message will be displayed on that widget.

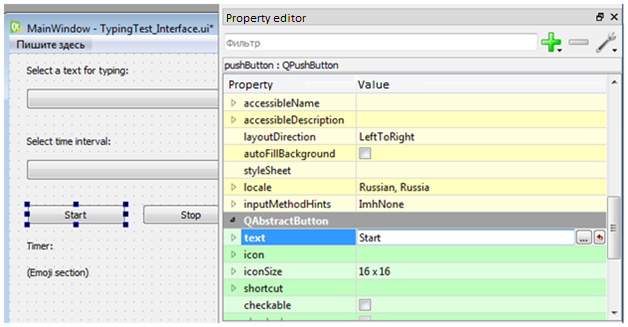


Fig.2. Setting a caption to a button

As you can see on the Figure 3, I've added the hint *Type in here…* to navigate a user. When they start typing, the message disappears. If you want to add a similar hint, set it to **placeholderText** property of textEdit widget.

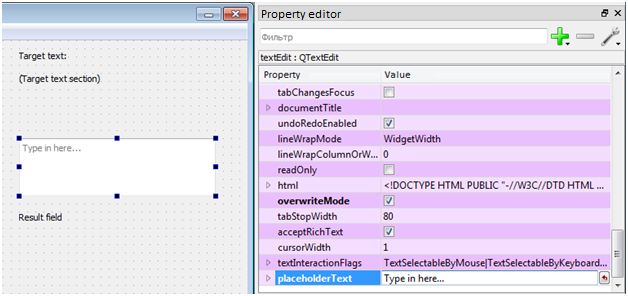


Fig.3. Setting hint to an input field

As for drop-down lists' items, you can only add them later, when writing your code, but not in Qt Designer.

If you provide **absolute positioning** and a user changes a size of the window, it will distort your widgets' layout. That's why it's important to either provide **relative positioning** or **forbid resizing** by fixing the size. I've chosen the second alternative. To do so, set the same values to **minimumSize** and **maximumSize** properties of the **MainWindow** object.

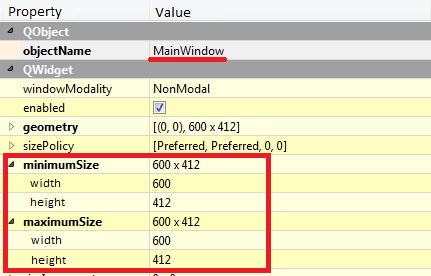


Fig.4. Fixing the window's size

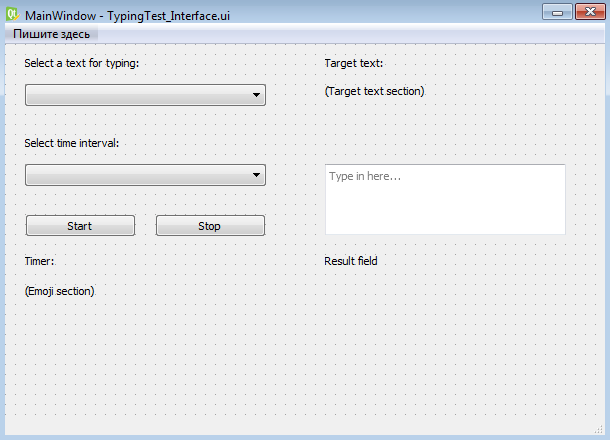


Fig.5. The final appearance of designed GUI

Save what you've designed with .ui extension. Instructions how to make your GUI work, I give in [*Writing code*](#_Writing_code) section.

### Choose texts

Now it's time to gather a text collection for your application. There are some factors that you have to take into account when selecting a text for the collection:

* Text's topic

It's a good idea to make a text amuse a test taker. If a text is boring, it's highly likely that he or she will never take it again. It also shouldn't contain any specific words such as terms or proper names. It should be as easy as possible.

* Text's characters

Your text shouldn't contain any specific characters that a user cannot find on a keyboard, such as dash (—), degree symbol (°) and so on. If it contains, a user can hardly achieve 100% result.

* Text's length

A number of characters in a text is also important. Texts should be as equal as possible in length. They should be not too long so that it was possible to retype it from beginning to end within allotted time interval. Considerations about how much time we should allot for a session, read in [*Define session duration*](#_Define_session_duration) section.

* A number of texts in the collection

There should be as many as possible. I have only three in my project, but you can find more.

Where can you find them? Texts that gathered into my collection are inspirational quotations of famous people. Such kind of texts is easy to perceive and usually short—it's exactly what we need! I scrolled various lists of quotations and picked up those which are good at its idea and about 150 characters long. I chose this number as an ideal text length: it's the number of characters which an average user is able to type per 60 seconds.

It's handy to store texts in an .xlsx file. Then your script will address to this file to get a text or its caption.

My texts' collection looks this way:

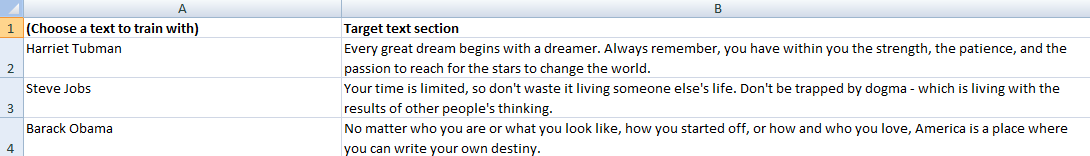


Fig.6. The texts' collection used in author's project

Choosing text this way was a bit time-consuming activity, so you may create your own texts with an interesting plot (if you have writing skills, of course).

### Define session duration

In my project there are 60, 45, and 30 seconds time intervals, which correspond to an easy, medium, and a hard level.

Here are a couple of tips how to choose intervals' length:

1. These intervals shouldn't be too long so that it couldn't bore a user
2. An interval's duration should be enough for a user to retype the whole text.

Define assessment criteria

Metrics for typing speed are **CPM** and **WPM**.

CPM stands for Characters Per Minute. The formula for calculating this metric is essentially the same as the formula for movement speed in physics:

In our project, we may consider *Distance* as number of characters that a user has typed before a session is over. As for *Time*, it's elapsed time from the beginning of a session to the moment when either session's time is over or a user has completed the whole text. So, now we have the specific formula for calculating typing speed:

(1) The formula for CPM metric

WPM stands for Words Per Minute. It's a conventional unit of measure; in English-speaking countries, WPM is used much more often than CPM.

As words have different length, a word is considered to be a five characters long string. For example, the phrase *I eat* counts as one word, but the single word *strawberry* counts as two. Therefore, to convert a user's result to the conventional representation, we divide an entered string's length (measured in characters) by five.

(2) The formula for WPM metric

According to statistics, an average speed range is 150-250 CPM (30-45 WPM). This range is for making a judgment if a user's typing speed is high or low.

Now let's create three grades for differentiating a user's result:

* *Excellent* level: above average (above 250 CPM (45 WPM))
* *Not bad* level: in average (from 150 to 250 CPM (30-45 WPM))
* *Typing skills need improving* level: below average (below 150 CPM (30 WPM))

When giving your judgment, speak with them carefully. Your message shouldn't be negative. If a result isn't high, the message should inspire a user to improve their skills, but not stating the fact that they are a bad typer.

## Writing code

We have designed ideas how our application should work. Now it’s time to put ideas into reality.

A python GUI application usually contains two files with .py extension:

* Interface configuration file
* Main file that describes widgets' behavior

Your **GUI file** that you designed earlier has **.ui extension**. To connect it to your script, you have to **convert it into .py**. A utility that you may use for converting goes along with other source files of my project.

It's almost impossible to create a perfect interface from the first attempt, so you will have to repeat this operation many times.

|  |
| --- |
| **Note** |
| If you edit your interface file, you will have to convert it once again |

There is a template for your convenience. Open this file and start writing your code there.

*Please, pay attention: this is not the whole script, just template. To run it, insert code which you can find below in this article or write your own if you wish. The whole author's script you can find in the distributive package.*

from fuzzywuzzy import fuzz  
from TypingTest\_Interface import \*  
from PyQt5 import QtWidgets  
import sys  
import threading  
import time

class Functions:  
  
 def read\_excel\_cells(self, fname, n, nsheet):  
 import xlrd  
 workbook = xlrd.open\_workbook(fname)  
 sheet = workbook.sheet\_by\_index(nsheet)  
 cells = []  
  
 for c in range(sheet.nrows):  
 cell = str(sheet.cell\_value(c, n))  
 cells.append(cell)  
  
 return cells  
  
 def read\_excel\_cells\_couple(self, fname, n1, n2):  
 import xlrd  
 workbook = xlrd.open\_workbook(fname)  
 sheet = workbook.sheet\_by\_index(0)  
 cells = []  
 for c in range(sheet.nrows):  
 cell = []  
 cell1 = str(sheet.cell\_value(c, n1))  
 cell2 = str(sheet.cell\_value(c, n2))  
 cell.append(cell1)  
 cell.append(cell2)  
 cells.append(cell)  
  
 return cells

class MainWindow(QtWidgets.QMainWindow):  
  
 # Variables section  
  
 # Initialization function  
 def \_\_init\_\_(self, parent=None):  
 QtWidgets.QWidget.\_\_init\_\_(self, parent)  
 self.ui = Ui\_MainWindow()  
 self.ui.setupUi(self)  
  
 # Other functions  
  
  
# Entry point of the program  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QtWidgets.QApplication(sys.argv)  
 myapp = MainWindow()  
 myapp.show()  
  
 # Display a message if an error occurs  
 sys.\_excepthook = sys.excepthook  
  
 def exception\_hook(exctype, value, traceback):  
 print(exctype, value, traceback)  
 sys.\_excepthook(exctype, value, traceback)  
 sys.exit(1)  
  
 sys.excepthook = exception\_hook  
 sys.exit(app.exec\_())

The first thing you can do is to add items to drop-down lists. As you remember, **comboBox** objects were created with no options.

There are two drop-down lists on my GUI:

* **comboBox\_1**: for choosing a text to retype
* **comboBox\_2**: for choosing session duration.

To fill a drop-down list with options' names, follow these steps:

Step 1: Pass its list of strings to the **addItems** method (I will start demonstrating with the second object.)

## Initialization function

...

time\_intervals = ['Easy (60s)', 'Medium (45s)', 'Hard (30s)']

self.ui.comboBox\_2.addItems(time\_intervals)

|  |
| --- |
| **Note** |
| A comment header at the top of each example indicates a place where you should insert your code. It starts with ## (double hash sign); ... (three dots) indicate code's ellipsis. |

After attaching options' list to an object it's time to define what each of these options means. Each one must activate its own piece of code.

Step 2: Create a new function's name, attach it to your drop-down list, and then define the function. Describe there what happens in each case using **if-statement**.

## Initialization function

...

self.ui.comboBox\_2.activated[str].connect(self.onTimeActivated)

# Attaching the onTimeActivated function to comboBox\_2

# comboBox\_2 passes the string argument. It tells the

function which option a user has selected

## Other functions

...

def onTimeActivated(self, text):

# Describing what each option means  
 if text == 'Easy (60s)':  
 self.time\_range = 60  
 elif text == 'Medium (45s)':  
 self.time\_range = 45  
 elif text == 'Hard (30s)':  
 self.time\_range = 30

Repeat these two steps with the first drop-down list. As it serves for choosing a text, a list of options should consist of texts' headers in this case. Headers, as well as texts themselves, are stored in the Excel file, so read it from there before putting into the drop-down list.

## Variables section

...

# Reading texts' headers from the Excel file

headers = Functions().read\_excel\_cells(text\_filename, 0, 0)

## Initialization function

def \_\_init\_\_(self, parent=None):

...  
 self.ui.comboBox.addItems(self.headers)  
 self.ui.comboBox.activated[str].connect(self.onHeaderActivated)

The onHeaderActivated function displays selected text on the label.

## Other functions

...

def onHeaderActivated(self, text):  
 self.selected\_header = text  
 index = self.headers.index(self.selected\_header)  
 self.target\_text = self.text\_collection[index][1]  
 self.ui.label\_2.setText(self.target\_text)  
 self.ui.label\_2.setWordWrap(True)

Ok, both drop-down lists are now ready-to-use. Let's move on.

Next widget that needs defining is pushButton. There should be two of them on the form: a *Start* and a *Stop* buttons. As the names imply, the first one is for starting a session and another one is for terminating a session prematurely.

Attaching functions to buttons works the same way as in case of drop-down lists: you invent a function's name, pass it as an argument to a button's connect method, and then define the function's content.

## Initialization function

...

# Attaching functions to push-buttons

self.ui.pushButton.clicked.connect(self.preStart)  
 self.ui.pushButton\_2.clicked.connect(self.stopSessionF)

## Other functions

...

def preStart(self):  
  
 # Turning the input field available,  
 # clearing it and some labels before each session's start  
 self.ui.textEdit.setEnabled(True)  
 self.ui.label\_3.setText("")  
 self.ui.label\_7.setText("")  
 self.ui.textEdit.setPlainText("")  
  
 # Start a session  
 TH = threading.Thread(target=self.Session())  
 TH.start()

def stopSessionF(self): # The function interrupts a session execution  
 self.stopSession = True

The preStart function is used only for clearing the window's text and launching the Start function in parallel **thread**. The description of what threading is, read in the next section.

### Threading

Long-running tasks, such as timer's work, require **multithreading**. It allows us to launch two processes at the same time. Otherwise the GUI will be frozen while a timer counts seconds.

When a user pushes the *Start* button, the program launches a parallel branch of code execution. Those two parallel branches are called **threads**. The new thread is executed as long as a session lasts, while the main thread keeps the interface displayed on screen.

### Session execution

The Start function, which is the new thread, manages timer's work and calculates a user's result.

#### Timer's work

In this application, timer is *while* loop, one iteration of which lasts one second. Python package **time** allows measuring elapsed time and retarding a script's execution. The last feature is useful as we display remaining time for a user, and it should be updated once a second.

Before timer starts, we need to calculate the moment when timer should stop. session\_duration variable is time interval in seconds which a user selected earlier. time.time() command takes current time measured in seconds as well; we may call it an *initial point*. The *final point* of a session is a sum of these two parameters.

stop\_time = time.time() + self.session\_duration  
  
# TIMER  
while True:  
 # calculating remained time  
 remained\_time = round(stop\_time - time.time())

While the loop is being executed, we calculate remained\_time variable at each iteration. It informs a user how much time is remained.

The loop is executed forever until it's broken by one of three termination conditions:

1. Time is over

Current time exceeds stop\_time value

1. A test taker completed the whole text

Text entered by a user is completely the same as its target text

1. A test taker pushed the *Stop* button

A user has decided to interrupt a test session.

# TIMER  
while True:  
 # calculating remained time  
 remained\_time = round(stop\_time - time.time())  
  
 # displays actual time on the screen  
 self.ui.label\_5.setText(f'Timer: {remained\_time}')  
 self.ui.label\_5.setWordWrap(True)  
  
 time.sleep(1) # retards loop's execution  
  
 # The timer stops (and the session is terminated)  
 # if one of three conditions is true  
 if time.time() > stop\_time: # 1st  
 remained\_time = 0  
 self.ui.label\_5.setText(f'Timer: {remained\_time}')  
 break  
 elif entered\_text == self.target\_text or \  
 self.stopSession: # 2nd & 3rd  
 break  
 # Updating text which a user has entered to monitor the progress  
 entered\_text = self.ui.textEdit.toPlainText()  
# \end\ TIMER

#### Calculating session's result

There are two metrics that influence a result: typing speed and error rate. Speed is calculated by the formula which is described in *Assessment criteria* section (formula No. 2). It's a key metric in estimating a user's typing skills.

Depending on which level a user's result belongs to, we display a corresponding emoji. The special widget for displaying images in Python is **pixmap**. The widget is to be set on a label.

## loading image

...

pixmap = QtGui.QPixmap("path/filename").scaled(Height, Width)

self.ui.label.setPixmap(pixmap)

**Height** and **Width** are numeric values here.

As for error rate, there is special metric called *Levenshtein distance*. It's a numeric metric for measuring the difference between two strings. In our project it shows how much entered text deviates from its original. The **fuzzywuzzy** Python library allows us to calculate this metric with only one string of code. Its value in our case depends on two factors: completeness and number of misprints.

## Building an executable

We together have already built the application. Revise your script; add some pieces of code that you have missed, if any. Test for possible bugs. Highly likely that it will take many iterations to reach the desired work.

If you'd like to share your application with other people, it's a good idea to build an **executable file**. This file type is able to provide users running your application on their PCs no matter whether they have a Python IDE installed or not. There are several utilities for achieving this goal, and **pyinstaller** is one of them.

|  |
| --- |
| To build an EXE file, open command prompt (CMD) and go to your application’s directory. From there, type **pyinstaller --onefile --windowed <*your script’s name*>** |

The command will bundle all source files and dependencies into a single stand-alone executable. Don’t forget to test the final result to ensure if everything is ok.

That’s it. Enjoy the result! It's interesting to test your own typing skills ☺

## Glossary

1. **Graphical User Interface**, or **GUI**, is a visual form of an application that contains input fields, buttons, and other widgets. It makes a user's interaction with the application easier. [↑](#endnote-ref-1)
2. **Target text** is a text chosen by a user for retyping. It is displayed on a screen while a session is running. [↑](#endnote-ref-2)
3. **Entered text** is what a user succeeds to enter before a session is over. [↑](#endnote-ref-3)
4. **Emojis** are pictograms (picture-like symbols) that are widely used in text messaging nowadays. An emoji shows an emotion: joy, anger, or another one from a wide range. [↑](#endnote-ref-4)
5. **What You See Is What You Get**, or **WYSIWYG**, is an approach in which software allows a user to edit content in a form that resembles its final appearance. [↑](#endnote-ref-5)