

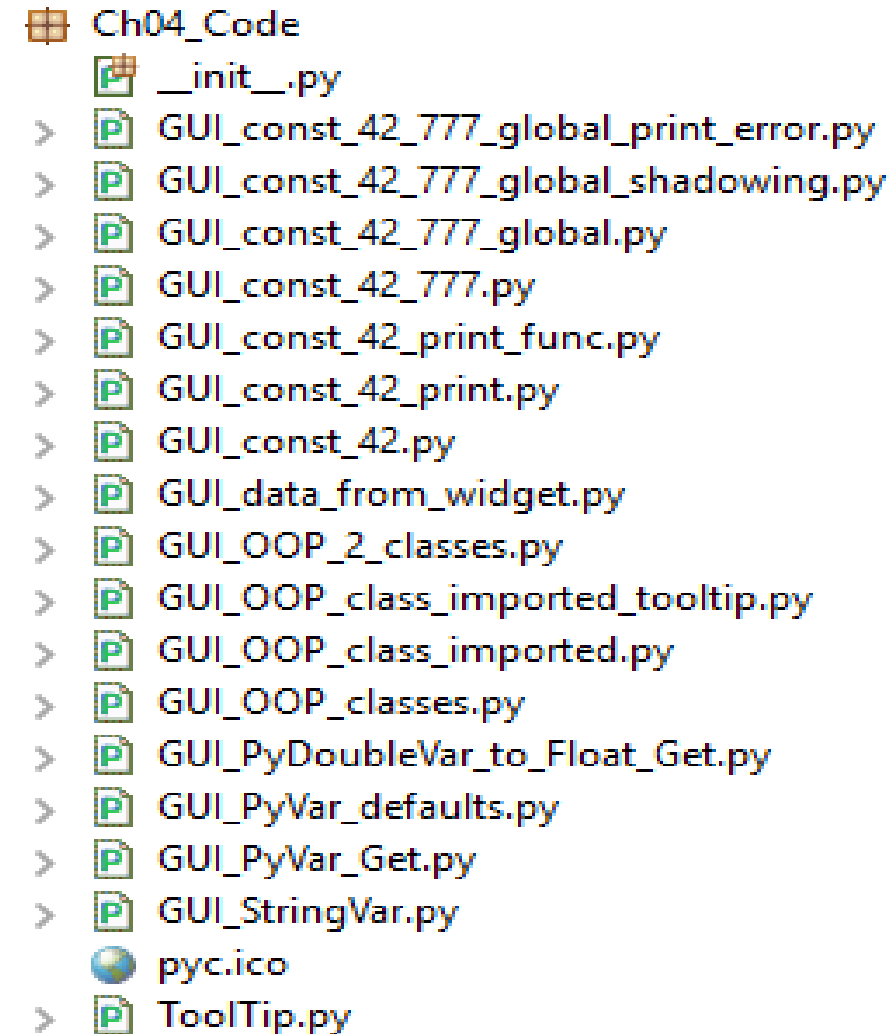
# **Chapter 4: GUI**

## **Data and Classes**

# Data and Classes

In this chapter, we will **save** our GUI **data into** tkinter **variables**. We will also start using object-oriented programming (OOP). This will lead us to **creating reusable OOP components**.

By the end of this chapter, you will know how to save data from the GUI into local tkinter variables. You will also learn **how to display tooltips** over widgets, which **give the user additional information**.



# Content

1. How to use `StringVar()`
2. How to get data from a widget
3. Using module-level global variables
4. How coding in classes can improve the GUI
5. Writing callback functions
6. Creating reusable GUI components

# 1\_How to use StringVar()

There are built-in programming types in tkinter that differ slightly from the Python types we are used to programming with. StringVar() is one such tkinter type.

## Getting ready

In this recipe, you will learn how to **save data** from the tkinter GUI **into variables** so we can use that data. We can set and get their values, which is very **similar** to how you would use the Java **getter/setter** methods.

Here are some of the types of code in tkinter:

<code>strVar = StringVar()</code>	Holds a string; the default value is an empty string ("")
<code>intVar = IntVar()</code>	Holds an integer; the default value is 0
<code>dbVar = DoubleVar()</code>	Holds a float; the default value is 0.0
<code>blVar = BooleanVar()</code>	Holds a Boolean, it returns 0 for False and 1 for True



Different languages call numbers with decimal points float or double. tkinter calls them DoubleVar, which is known in Python as the float data type. Depending on the level of precision, float and double data can be different. Here, we are translating **DoubleVar** of tkinter **into** a **Python float** type.

## How to do it...

We will create a DoubleVar of tkinter variable and add a float number literal to it using the + operator. After that, we will look at the resulting Python type.

Here are the steps to see the different tkinter data types:

1. Create a new Python module and name it `GUI_PyDoubleVar_to_Float_Get.py`.
2. At the top of the `GUI_PyDoubleVar_to_Float_Get.py` module, import tkinter:  

```
import tkinter as tk
```

3. Create an instance of the tkinter class:

```
win = tk.Tk()
```

4. Create a DoubleVar and give it a value:

```
doubleData = tk.DoubleVar()
```

```
print(doubleData.get()) doubleData.set(2.4)
```

```
print(type(doubleData))
```

```
add_doubles = 1.22222222222222222222222222222222 + doubleData.get()
```

```
print(add_doubles)
```

```
print(type(add_doubles))
```

5. The following screenshot shows the final GUI\_PyDoubleVar\_to\_Float\_Get.py code:



```
import tkinter as tk

# Create instance of tkinter
win = tk.Tk()

# Create DoubleVar
doubleData = tk.DoubleVar()
print(doubleData.get())           # default value
doubleData.set(2.4)
print(type(doubleData))

add_doubles = 1.222222222222222222222222 + doubleData.get()
print(add_doubles)
print(type(add_doubles))
```

<

 Console 

<terminated> GUI\_PyDoubleVar\_to\_Float\_Get.py [C:\Python37\python.exe]

0.0  
<class 'tkinter.DoubleVar'>  
3.6222222222222222  
<class 'float'>



We can do the same with tkinter with regards to strings.

We will create a new Python module as follows:

1. Create a new Python module and name it `GUI_StringVar.py`.

2. At the top of the `GUI_StringVar.py` module, import tkinter:

```
import tkinter as tk
```

3. Create an instance of the tkinter class:

```
win = tk.Tk()
```

4. Assign a StringVar of tkinter to the strData variable:

```
strData = tk.StringVar()
```

5. Set the strData variable:

```
strData.set('Hello StringVar')
```

6. Get the value of the strData variable and save it in varData:

```
varData = strData.get()
```

7. Print out the current value of strData:

```
print(varData)
```

8. The following screenshot shows the final GUI\_StringVar.py code and the output after running the code:

```
import tkinter as tk

# Create instance of tkinter
win = tk.Tk()

# Assign tkinter Variable to strData variable
strData = tk.StringVar()

# Set strData variable
strData.set('Hello StringVar')

# Get value of strData variable
varData = strData.get()

# Print out current value of strData
print(varData)
```

<

Console ✕

<terminated> GUI\_StringVar.py [C:\Python37\python.exe]

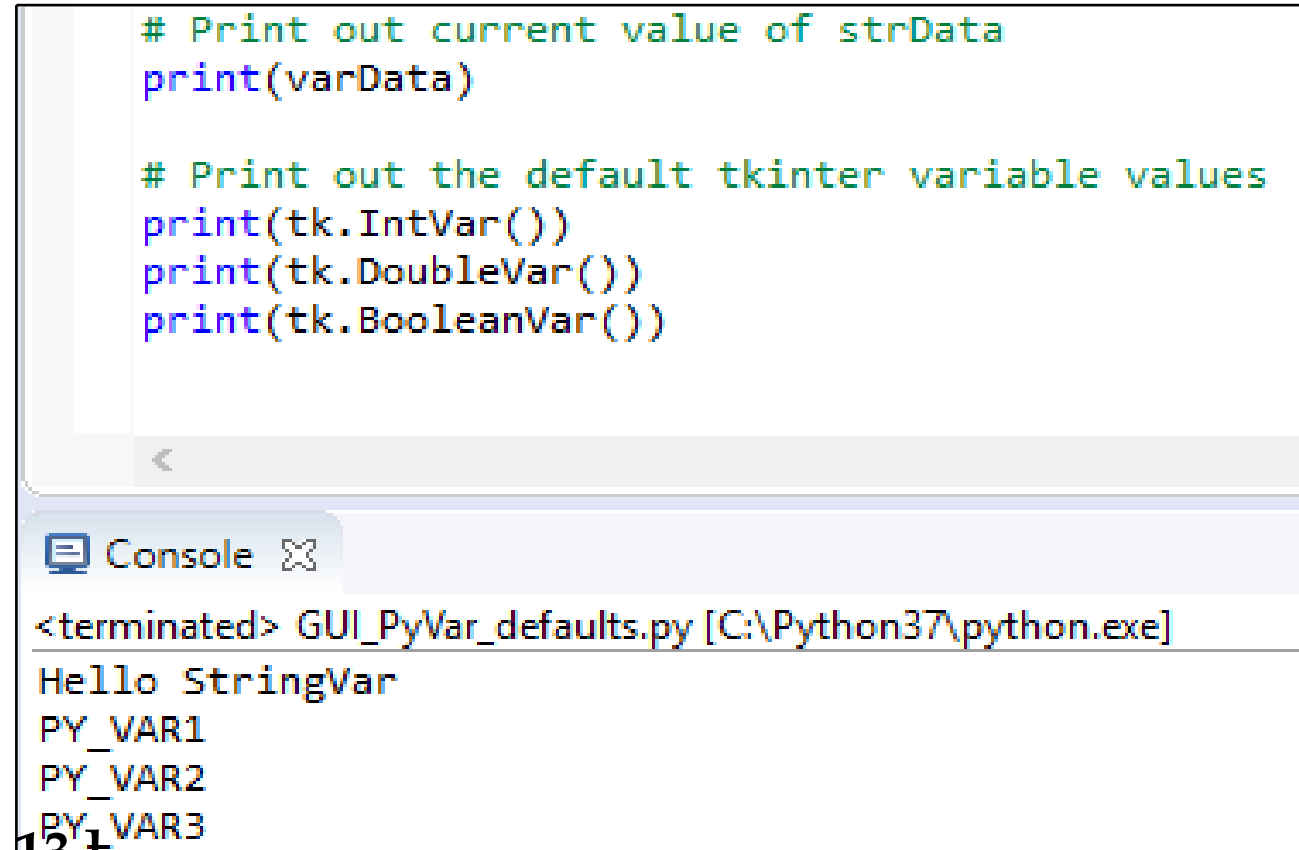
Hello StringVar

Next, we will print the default values of, IntVar, DoubleVar, and BooleanVar types of tkinter:

1. Open GUI\_StringVar.py and save the module as GUI\_PyVar\_defaults.py.
2. Add the following lines of code toward the bottom of this module:

```
print(tk.IntVar())  
  
print(tk.DoubleVar())  
  
print(tk.BooleanVar())    [ 12 ]
```

3. The following screenshot shows the final GUI\_PyVar\_defaults.py code and the output after running the GUI\_PyVar\_defaults.py code file:



```
# Print out current value of strData
print(varData)

# Print out the default tkinter variable values
print(tk.IntVar())
print(tk.DoubleVar())
print(tk.BooleanVar())
```

Console

<terminated> GUI\_PyVar\_defaults.py [C:\Python37\python.exe]

Hello StringVar  
PY\_VAR1  
PY\_VAR2  
PY\_VAR3

The steps to print the default tkinter variable value are as follows:

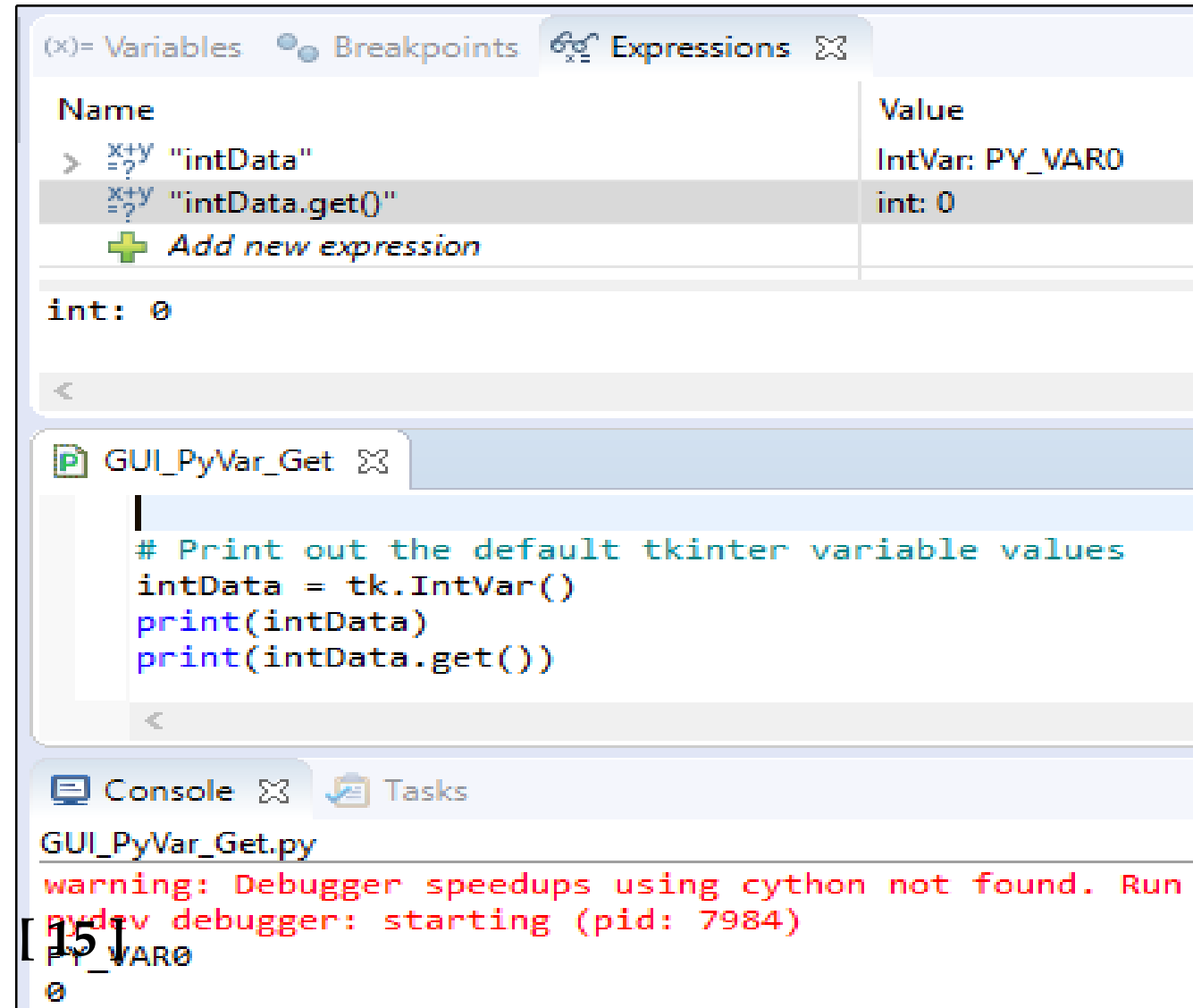
1. Create a new Python module and name it `GUI_PyVar_Get.py`.
2. Type the following code into the module:

```
import tkinter as tk
# Create instance of tkinter win = tk.Tk()
# Print out the default tkinter variable values intData = tk.IntVar()

print(intData) print(intData.get())
# Set a breakpoint here to see the values in the debugger

print()
```

3. Run the code, optionally setting a breakpoint in your IDE in the final `print()` statement:



The screenshot displays a Python IDE interface with three main panels:

- Expressions Window:** Located at the top, it shows a table of expressions and their values.

Name	Value
<code>&gt; x+y</code> "intData"	IntVar: PY_VAR0
<code>&gt; x+y</code> "intData.get()"	int: 0
<a href="#">+ Add new expression</a>	
- Code Editor:** The middle panel shows the source code for `GUI_PyVar_Get.py`.

```
# Print out the default tkinter variable values
intData = tk.IntVar()
print(intData)
print(intData.get())
```
- Console Window:** The bottom panel shows the output of the program.

```
GUI_PyVar_Get.py
warning: Debugger speedups using cython not found. Run
pydev debugger: starting (pid: 7984)
PY_VAR0
0
```

## 2\_ How to get data from a widget

When the user enters data, we want to do something with it in our code. This recipe shows how to capture data in a variable. In the previous recipe, we created several tkinter class variables. They were standalone. Now, we are connecting them to our GUI, using the data we get from the GUI, and storing them in Python variables.

### Getting ready

We will continue using the Python GUI we were building in Chapter 3, Look and Feel Customization. We'll reuse and enhance the code from `GUI_progressbar.py` from that chapter.



## How to do it...

We will assign a value from our GUI to a Python variable:

1. Open `GUI_progressbar.py` from Chapter 3, Look and Feel Customization, and save the module as `GUI_data_from_widget.py`.
2. Add the following code toward the bottom of our module. Just above the main event loop, add `strData`:

```
strData = spin.get() print("Spinbox  
value: " + strData)      [ 17 ]
```

3. Add code to place the cursor into the name entry:

```
name_entered.focus()
```



4. Start the GUI:

```
win.mainloop()
```

5. Running the code gives us the following result:

```
# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')
```

<

 Console 

<terminated> GUI\_data\_from\_widget.py [C:\Python37\python.exe]  
Spinbox value: 1

We will retrieve the current value of the Spinbox control:

1. We create our Spinbox widget using the following code, hard-coding the available values into it:

```
# Adding a Spinbox widget using a set of values
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5,
               bd=8, command=_spin)
spin.grid(column=0, row=2)
```

2. We can also move the hard-coding of the data out of the creation of the Spinbox class instance and set it later:

```
# Adding a Spinbox widget assigning values after  
creation spin = Spinbox(mighty, width=5, bd=8,  
command=_spin) spin['values'] = (1, 2, 4, 42, 100)  
spin.grid(column=0, row=2)
```

#### NOTE:

In order to get the values out of our GUI written using tkinter, we use the `get()` method of tkinter

## 3\_ Using module-level global variables

Encapsulation is a major strength in any programming language, enabling us to program using OOP.

Because as we add more and more functionality to our GUI, we want to avoid naming conflicts that could result in bugs in our code.

### Getting ready

We can declare module-level globals in any module just above and outside functions.

We then have to use the global Python keyword to refer to them. If we forget to use `global` in functions, we will accidentally create new local variables.

## How to do it...

Add the following code to the GUI we used in the previous recipe, How to get data from a widget, creating a module-level global variable. We use the all-uppercase convention for constants:

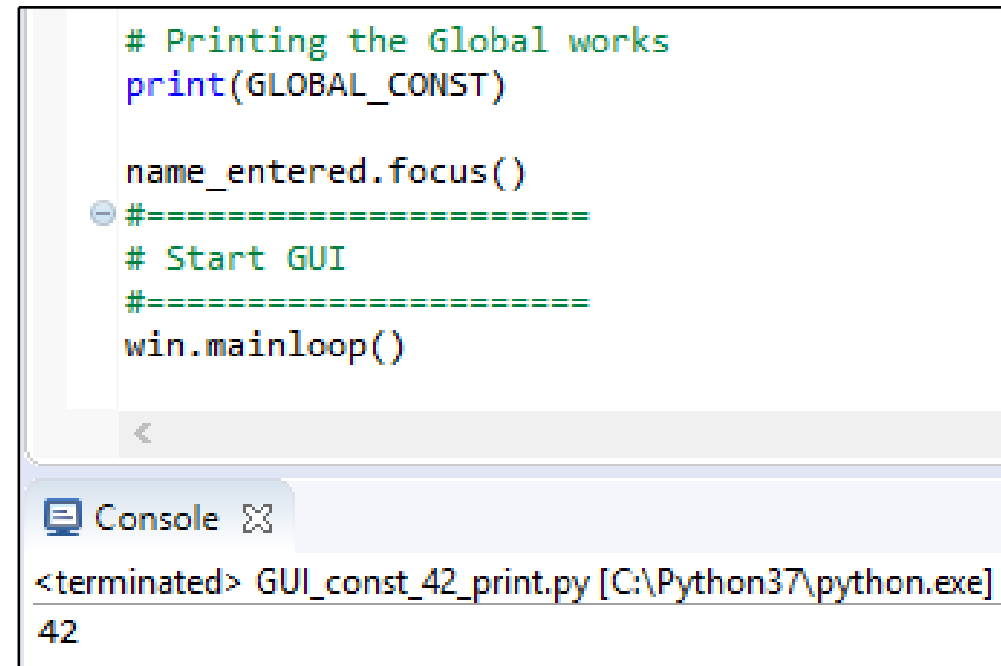
You can find more information in PEP 8 -- Style Guide for Python Code at

<https://www.python.org/dev/peps/pep-0008/#constants>.

1. Open GUI\_data\_from\_widget.py and save the module as GUI\_const\_42\_print.py.
2. Add the constant variable at the top and the print statement at the bottom of the module:

```
GLOBAL_CONST = 42 # ...  
print(GLOBAL_CONST)
```

3. Running the code results in a printout of the global.



The screenshot shows a Python IDE window with a code editor and a console. The code editor contains the following Python code:

```
# Printing the Global works  
print(GLOBAL_CONST)  
  
name_entered.focus()  
#=====  
# Start GUI  
#=====  
win.mainloop()
```

Below the code editor is a console window titled "Console". It shows the output of the program:

```
<terminated> GUI_const_42_print.py [C:\Python37\python.exe]  
42
```



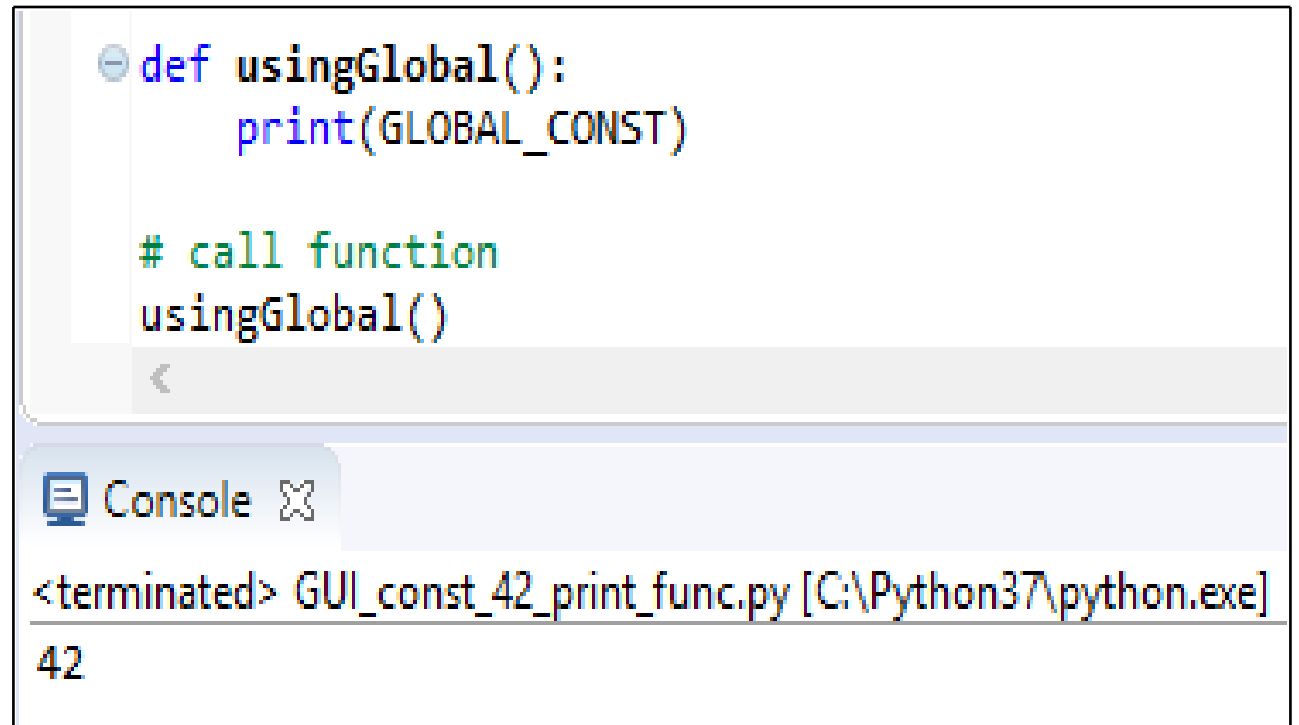
Add the usingGlobal function toward the bottom of the module:

1. Open GUI\_const\_42\_print.py and save the module as GUI\_const\_42\_print\_func.py.

2. Add the function and then call it:

```
def usingGlobal():  
    print(GLOBAL_CONST)  
# call the function  
usingGlobal()
```

3. The following screenshot shows the final GUI\_const\_42\_print\_func.py code and the output after running the code:

A screenshot of a Python IDE. The top pane shows a Python script with a function definition and a call to that function. The bottom pane, titled 'Console', shows the output of the script.

```
def usingGlobal():  
    print(GLOBAL_CONST)  
  
# call function  
usingGlobal()
```

<terminated> GUI\_const\_42\_print\_func.py [C:\Python37\python.exe]  
42

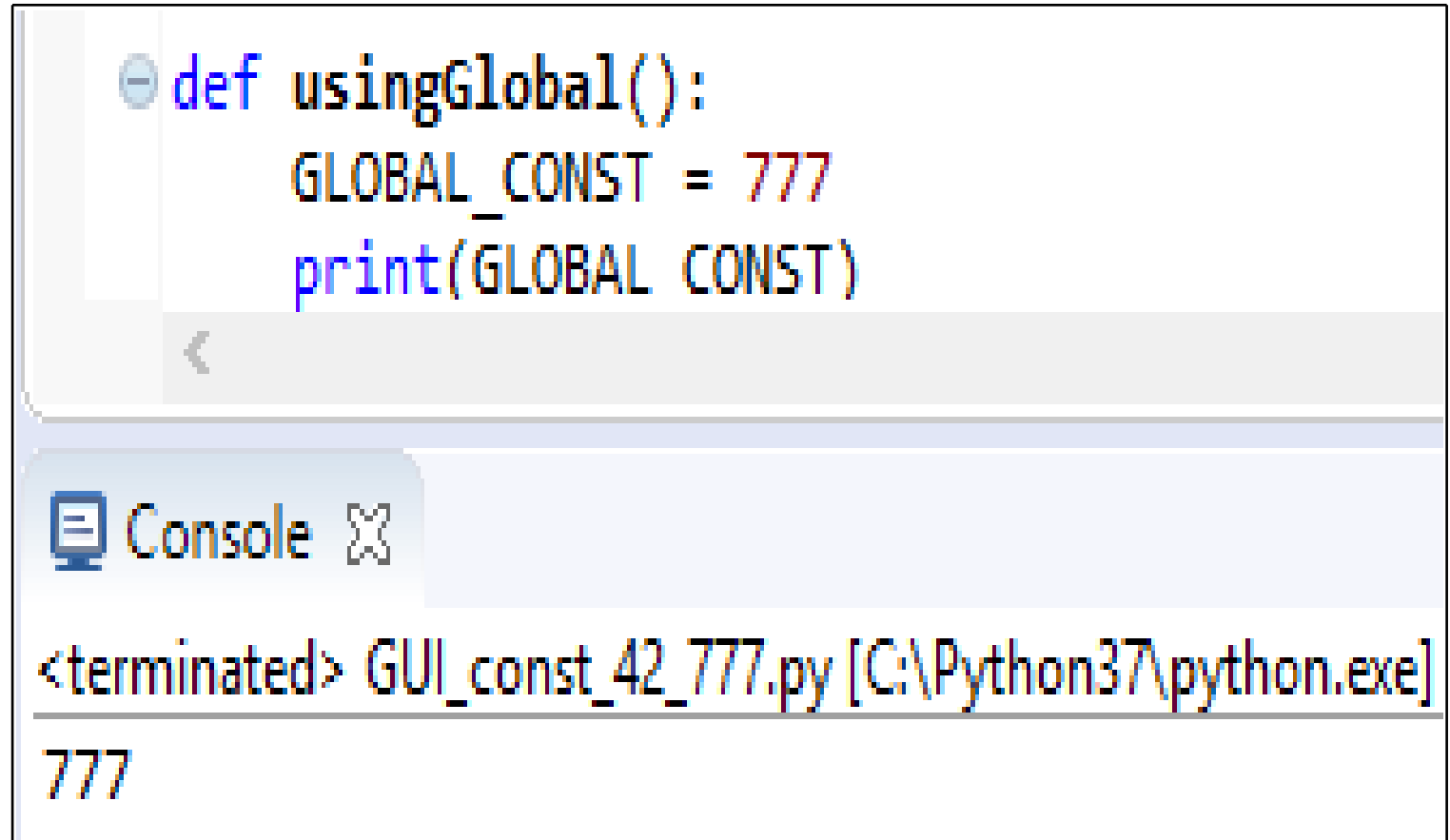
In the preceding code snippet, we use the module-level global. It is easy to make a mistake by shadowing the global, as demonstrated in the following code:

1. Open `GUI_const_42_print_func.py`  
and save the module as `GUI_const_42_777.py`.

2. Add the declaration of the constant within the function:

```
def usingGlobal():  
    GLOBAL_CONST = 777  
    print(GLOBAL_CONST) 27 ]
```

3. The following screenshot shows the final `GUI_const_42_777.py` code and the output after running the code:



The screenshot displays a Python IDE interface. The top pane shows the source code for `GUI_const_42_777.py`:

```
def usingGlobal():  
    GLOBAL_CONST = 777  
    print(GLOBAL_CONST)
```

The bottom pane, titled "Console", shows the output of the program:



```
<terminated> GUI_const_42_777.py [C:\Python37\python.exe]  
777
```

If we try to print out the value of the global variable, without using the global keyword, we get an error:

1. Open GUI\_const\_42\_777.py and save the module as GUI\_const\_42\_777\_global\_print\_error.py.
2. Comment out global and try to print:

```
def usingGlobal():  
    # global GLOBAL_CONST  
    print(GLOBAL_CONST)  
    GLOBAL_CONST = 777  
    print(GLOBAL_CONST)
```

### 3. Run the code and observe the output:

 Console 

```
<terminated> GUI_const_42_777_global_print_error.py [C:\Python37\python.exe]
```

```
Traceback (most recent call last):
```

```
File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd Edition Python GUI  
usingGlobal()
```

```
File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd Edition Python GUI  
print(GLOBAL_CONST)
```

```
UnboundLocalError: local variable 'GLOBAL_CONST' referenced before assignment
```

When we qualify our local variable with the global keyword, we can print out the value of the global variable and overwrite this value locally:

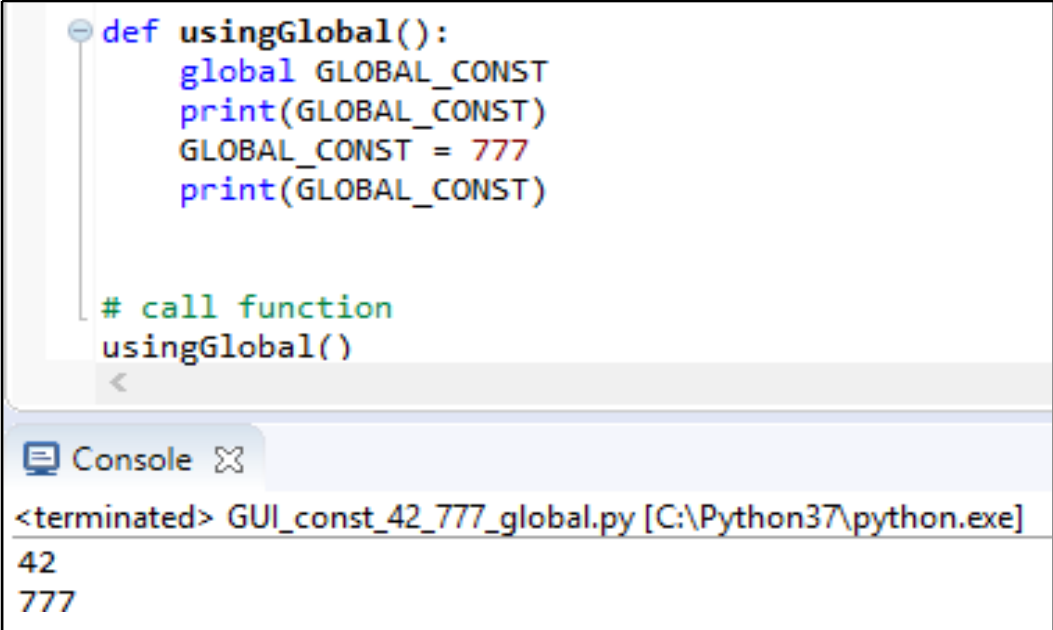
1. Open GUI\_const\_42\_777\_global.py.
2. Add the following code:

def usingGlobal():

```
global GLOBAL_CONST
print(GLOVAL_CONST)
GLOBAL_CONST = 777
print(GLOBAL_CONST)
```

3. Run the code and observe the output:

[ 31 ]



The screenshot shows a Python IDE window with a code editor and a console. The code editor contains the following Python code:

```
def usingGlobal():
    global GLOBAL_CONST
    print(GLOBAL_CONST)
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)

# call function
usingGlobal()
```

The console output shows the execution of the code, displaying the value 42 followed by 777:

```
<terminated> GUI_const_42_777_global.py [C:\Python37\python.exe]
42
777
```

We might believe that the value of the global variable is local to our function.

1. Open GUI\_const\_42\_777\_global.py and save as GUI\_const\_42\_777\_global\_shadowing.py.
2. Add `print('GLOBAL_CONST:', GLOBAL_CONST)` below the function.
3. Run the code and observe the output:

```
# call function
usingGlobal()

# call the global from outside the function
print('GLOBAL_CONST:', GLOBAL_CONST)
```

Console

<terminated> GUI\_const\_42\_777\_global\_shadowing.py [C:\Python37\python.exe]

42  
777  
GLOBAL\_CONST: 777



## 4\_ How coding in classes can improve the GUI

So far, we have been coding in a **procedural style**. When our **code** gets **larger** and larger, we **need** to advance to **coding in OOP**.

Why?

Because, OOP allows us to move code around by using methods. Once we use classes, we no longer have to physically place the code above the code that calls it. This gives us great flexibility in organizing our code. We can write the related code next to the other code and no longer have to worry that the code will not run because the code does not sit above the code that calls it.

If the methods we call have not been created by that time, we get a runtime error.

## Getting ready

We will turn our entire procedural code into OOP very simply. We just turn it into a class, indent all the existing code, and prepend `self` to all variables.

It is very easy.

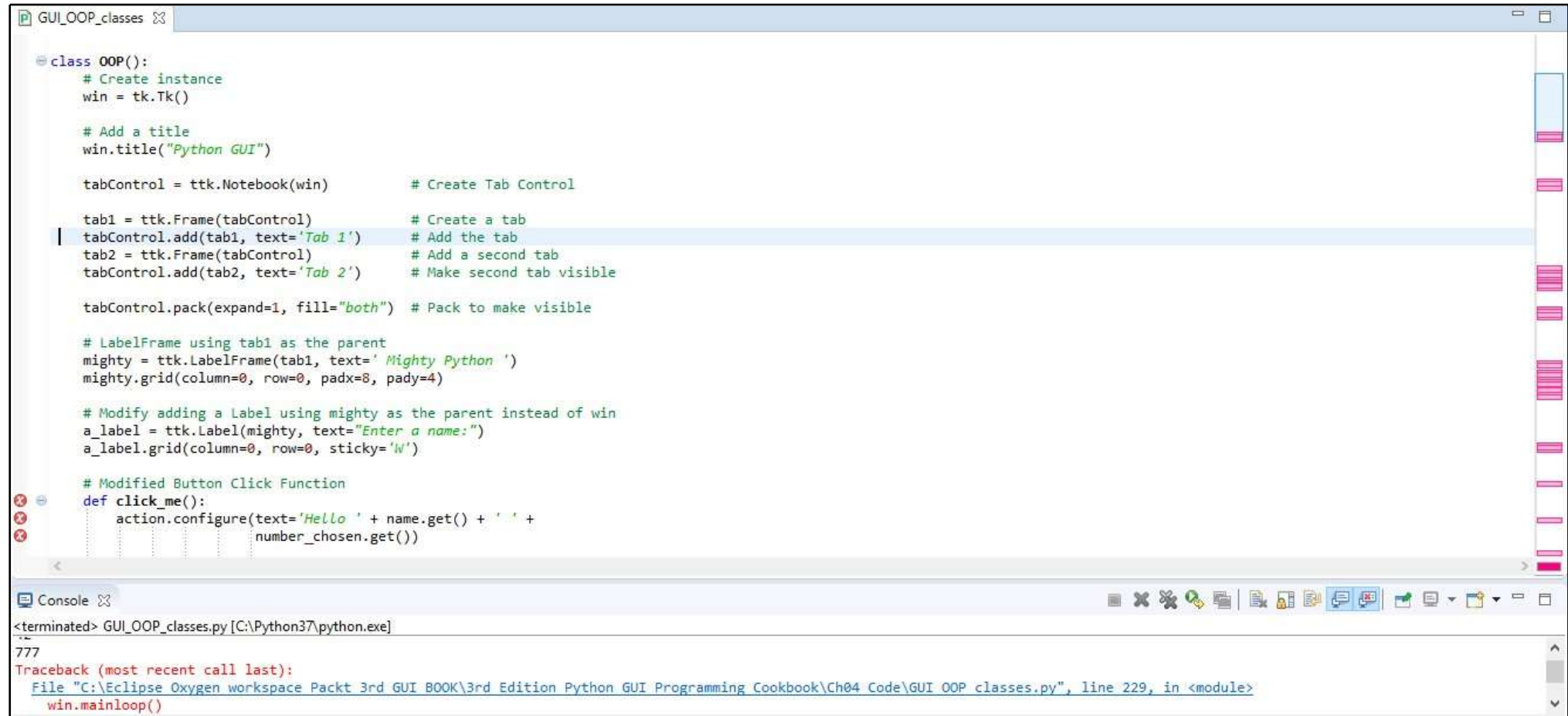
While at first it might feel a little bit annoying having to prepend everything with the `self` keyword, making our code more verbose (*hey, we are wasting so much paper...*), in the end it is worth it.

## **How to do it...**

Note that in the Eclipse IDE, the PyDev editor hints at coding problems by highlighting them in red on the right-hand side portion of the code editor.

1. Open `GUI_const_42_777_global.py`  
and save the module as `GUI_OOP_classes.py`.
2. Highlight the entire code below the imports and indent it by four spaces.
3. Add `class OOP():` above the indented code.

### 3.Look at all of the red errors in the code editor on the right-hand side:



```
class OOP():
    # Create instance
    win = tk.Tk()

    # Add a title
    win.title("Python GUI")

    tabControl = ttk.Notebook(win)      # Create Tab Control

    tab1 = ttk.Frame(tabControl)        # Create a tab
    tabControl.add(tab1, text='Tab 1')  # Add the tab
    tab2 = ttk.Frame(tabControl)        # Add a second tab
    tabControl.add(tab2, text='Tab 2')  # Make second tab visible

    tabControl.pack(expand=1, fill="both") # Pack to make visible

    # LabelFrame using tab1 as the parent
    mighty = ttk.LabelFrame(tab1, text='Mighty Python ')
    mighty.grid(column=0, row=0, padx=8, pady=4)

    # Modify adding a Label using mighty as the parent instead of win
    a_label = ttk.Label(mighty, text="Enter a name:")
    a_label.grid(column=0, row=0, sticky='W')

    # Modified Button Click Function
    def click_me():
        action.configure(text='Hello ' + name.get() + ' ' +
                           number_chosen.get())
```

Console

```
<terminated> GUI_OOP_classes.py [C:\Python37\python.exe]
777
Traceback (most recent call last):
  File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd Edition Python GUI Programming Cookbook\Ch04 Code\GUI OOP classes.py", line 229, in <module>
    win.mainloop()
```

We have to prepend all the variables with the self keyword and also bind the functions to the class by using self, which officially and technically turns the functions into methods.

Let's prefix everything with `self` to fix all of the red so we can run our code again:

1. Open `GUI_OOP_classes.py` and save the module as `GUI_OOP_2_classes.py`.
2. Add the `self` keyword wherever it is needed, for example, `click_me(self)`.
3. Run the code and observe it:

```
# Modified Button Click Function
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get() + ' ' +
                             self.number_chosen.get())
```

Once we do this for all of the errors highlighted in red, we can run our Python code again. The `click_me` function is now bound to the class and has officially become a method. → no error

Now let's add our ToolTip class from Chapter 3, *Look and Feel Customization*, into this Python module:

1. Open GUI\_OOP\_2\_classes.py.
2. Add the ToolTip class from GUI\_tooltip.py to the top of the following module's import statements:

```
class ToolTip(object):
```

```
    def __init__(self, widget, tip_text=None): self.widget = widget
```

```
    ...
```

```
class OOP():
```

```
    def __init__(self): self.win = tk.Tk()
```

```
        ToolTip(self.win, 'Hello GUI')
```

```
        # <-- use the ToolTip class here
```

```
    ...
```

## **How it works...**

We are translating our procedural code into object-oriented code. First, we indented the entire code and defined the code to be part of a class, which we named OOP. In order to make this work, we have to use the self keyword for both variables and methods.

Here is a brief comparison of our **previous code** with the **new OOP code** using a class:

**# Our procedural code looked like this:**

**# Button Click Function**

**def click\_me():**

**action.configure(text='Hello ' + name.get() + ' ' + number\_chosen.get())**

**# Adding a Textbox Entry widget**

**name = tk.StringVar()**

**name\_entered = ttk.Entry(mighty, width=12, textvariable=name)**

**name\_entered.grid(column=0, row=1, sticky='W')**



### # Adding a Button

```
action = ttk.Button(mighty, text="Click  
Me!", command=click_me)  
action.grid(column=2, row=1)
```

```
ttk.Label(mighty, text="Choose a number:").grid(column=1, row=0)
```

```
number = tk.StringVar()  
number_chosen = ttk.Combobox(mighty, width=12, textvariable=number, state='readonly')  
number_chosen['values'] = (1, 2, 4, 42, 100)  
number_chosen.grid(column=1, row=1)  
number_chosen.current(0)  
# ...
```

```
***** The new OOP code looks like this:
***** class OOP():
```

```
def __init__(self):      # Initializer method
    # Create instance
    self.win = tk.Tk()    # notice the self keyword
    ToolTip(self.win, 'Hello GUI')
    # Add a title
    self.win.title("Python GUI")
    self.create_widgets()

# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get() + ' ' + self.number_chosen.get())
    # ... more callback methods
```

```
def create_widgets(self):
    # Create Tab Control
    tabControl = ttk.Notebook(self.win)
    tab1 = ttk.Frame(tabControl)
    tabControl.add(tab1, text='Tab 1')    # Create a tab
    tab2 = ttk.Frame(tabControl)         # Add the tab
    tabControl.add(tab2, text='Tab 2')    # Create second tab
    # Pack to make visible                # Add second tab
    tabControl.pack(expand=1, fill="both")


    # Adding a Textbox Entry widget - using self
    self.name = tk.StringVar()
    name_entered = ttk.Entry(mighty, width=12, textvariable=self.name)
    name_entered.grid(column=0, row=1, sticky='W')
```

```
# Adding a Button - using self
self.action = ttk.Button(mighty, text="Click Me!", command=self.click_me)
self.action.grid(column=2, row=1) # ...
#=====
# Start GUI
#=====
oop = OOP()    # create an instance of the class
               # use instance variable to call mainloop via oop.win
oop.win.mainloop()
```

We moved all the widget-creation code into one rather long method, **create\_widgets**, which we call in the initializer of the class.

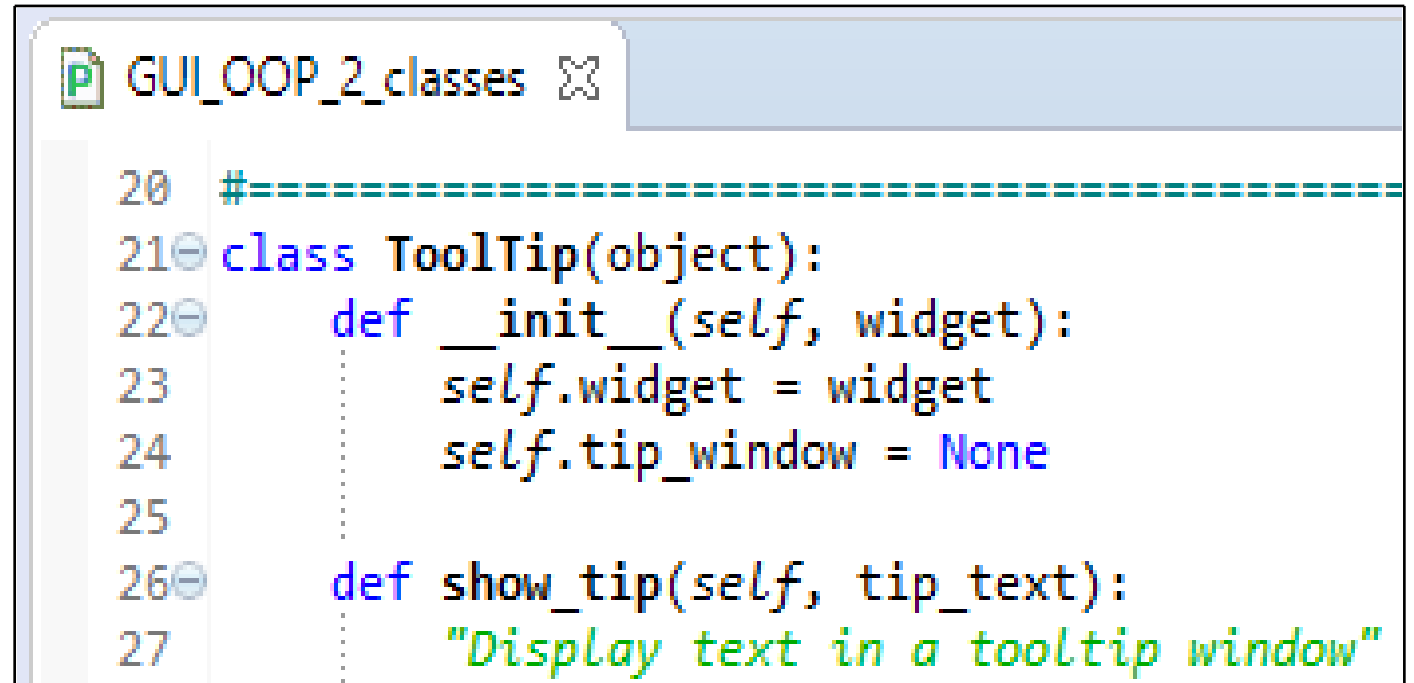
Instead, in addition to a real constructor, Python provides us with an initializer, **\_\_init\_\_(self)**. We are strongly encouraged to use this initializer. We can use it to pass in arguments to our class, initializing variables we wish to use inside our class instance.

In the end, we added the ToolTip class to the top of our module just below the import statements.

-  In Python, several classes can exist within the same Python module and the module name does not have to be the same as the class name.

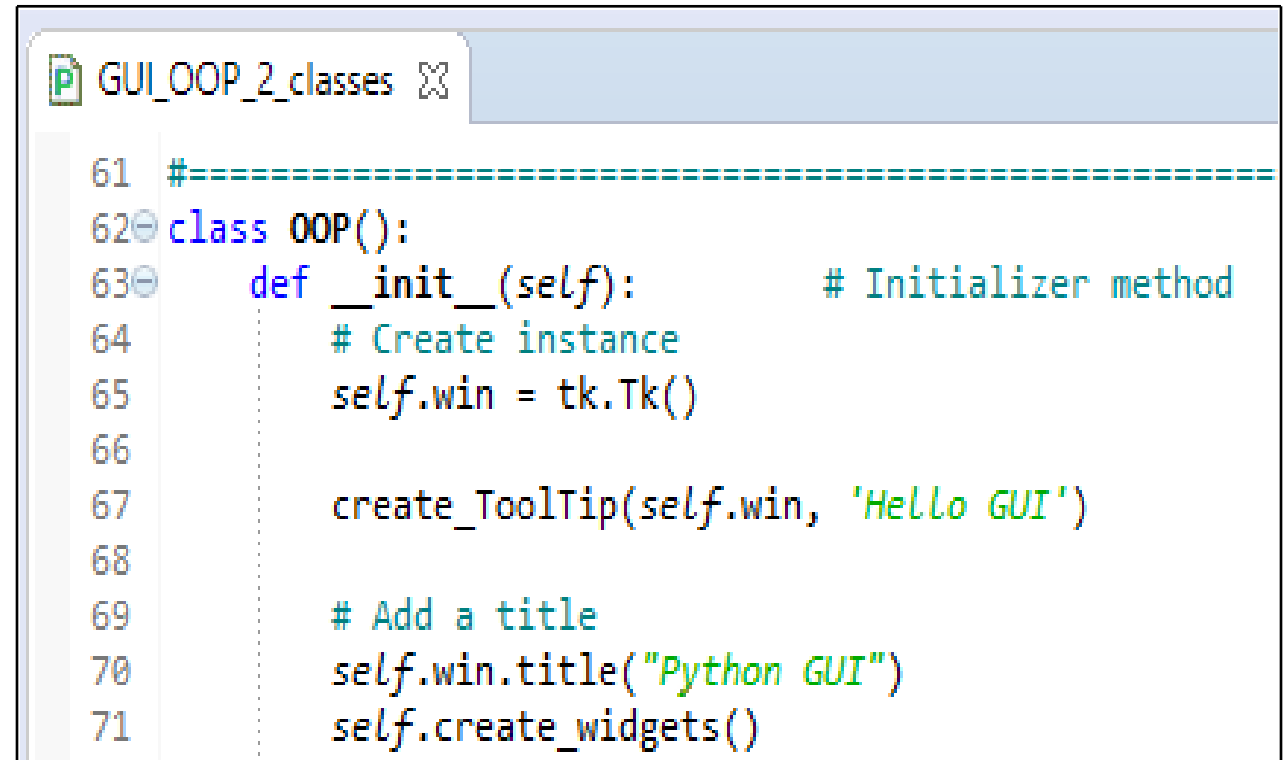
Here, in this recipe, we can see that more than one class can live in the same Python module.

Cool stuff, indeed! Here are two screenshots of the two classes residing in the same module:

A screenshot of a Python IDE window titled 'GUI\_OOP\_2\_classes'. The window shows a code editor with the following Python code:

```
20 #=====
21 class ToolTip(object):
22     def __init__(self, widget):
23         self.widget = widget
24         self.tip_window = None
25
26     def show_tip(self, tip_text):
27         "Display text in a tooltip window"
```

Both the ToolTip class and the OOP class reside within the same Python module, `GUI_OOP_2_classes.py`:



```
61 #=====
62 class OOP():
63     def __init__(self):          # Initializer method
64         # Create instance
65         self.win = tk.Tk()
66
67         create_ToolTip(self.win, 'Hello GUI')
68
69         # Add a title
70         self.win.title("Python GUI")
71         self.create_widgets()
```

In this recipe, we advanced our procedural code into OOP code.

## 5\_ Writing callback functions

The event handler is a callback function (or method, if we use classes). The callback method is also sitting there passively, like our GUI, waiting to be invoked. Once our GUI's button is clicked, it will invoke the callback.



## **Getting ready**

The Python interpreter runs through all the code in a module once, finding any syntax errors and pointing them out. You cannot run your Python code if you do not have the syntax right. This includes indentation (if not resulting in a syntax error, incorrect indentation usually results in a bug).

At runtime, many GUI events can be generated, and it is usually callback functions that add functionality to GUI widgets.

## How to do it...

Here is the callback for the `Spinbox` widget:

1. Open `GUI_OOP_2_classes.py`.
2. Observe the `_spin(self)` method in the code:

```
37     # Spinbox callback
38     def _spin(self):
39         value = self.spin.get()
40         print(value)
41         self.scrol.insert(tk.INSERT, value + '\n')

124
125     # Adding a Spinbox widget
126     self.spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=9, command=self._spin)
127     self.spin.grid(column=0, row=2)
```

## How it works...



We create a **callback method** in the OOP class that gets called when we **select a value** from the **Spinbox** widget because we bind the method to the widget via the command **argument** (*command=self.\_spin*). We use a leading underscore to hint at the fact that this method is meant to be respected like a private Java method.

## 6\_ Creating reusable GUI components

We will create reusable GUI components using Python. In this recipe, we will keep it simple by moving our ToolTip class into its own module. Then, we will import and use it to display tooltips over several widgets of our GUI.

### Getting ready

We are building our code from Chapter 3, *Look and Feel Customization*: GUI\_tooltip.py. We will start by breaking out our ToolTip class into a separate Python module.

## How to do it...

We will create a new Python module and place the ToolTip class code into it and then import this module into our primary module:

1. Open GUI\_OOP\_2\_classes.py and save the module as GUI\_OOP\_class\_imported\_tooltip.py.
2. Break out the ToolTip code from GUI\_tooltip.py into a new Python module and name the module ToolTip.py.
3. Import the ToolTip class into GUI\_OOP\_class\_imported\_tooltip.py:

```
from Ch04_Code.ToolTip import ToolTip
```

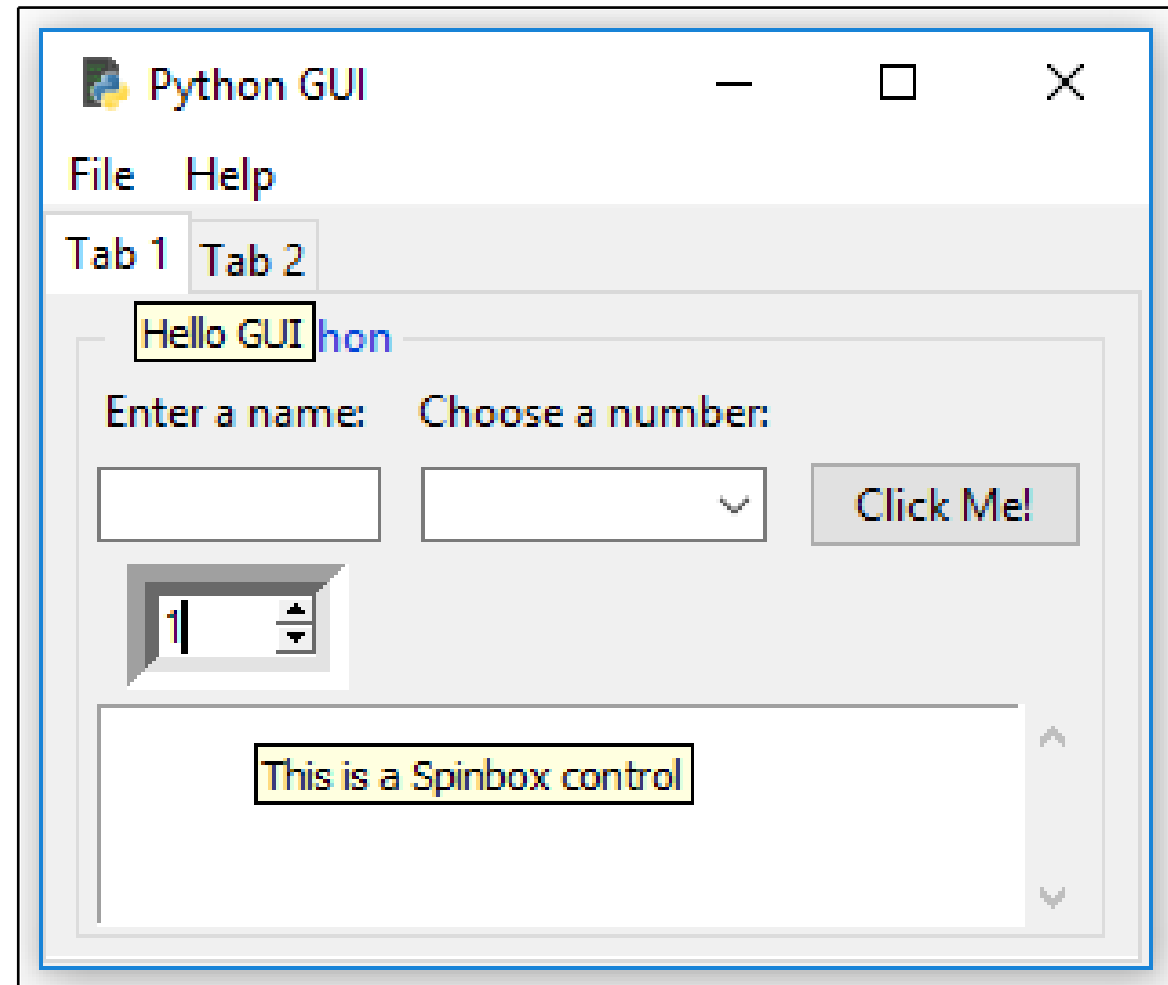
4. Add the following code to `GUI_OOP_class_imported_tooltip.py`:

```
ToolTip(self.win, 'Hello GUI')
# Add a ToolTip to the Spinbox
ToolTip(self.spin, 'This is a Spinbox control')

# Add tooltips to more widgets
ToolTip(self.name_entered, 'This is an Entry control')
ToolTip(self.action, 'This is a Button control')
ToolTip(self.scrol, 'This is a ScrolledText control')

# Tab 2
ToolTip(curRad, 'This is a Radiobutton control')
```

5. Run the code and hover the mouse over the different widgets:



This also works on the second tab:

