Unit 5 Python

Errors and Exceptions

There are two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Errors

Syntax errors, also known as parsing errors, are the most common kind of errors.

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. in the example, the error is detected at the function print(), since a colon (':') is missing before it.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions.*

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions.

Handling an exception

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a ***try:*** block. After the ***try:*** block, include an ***except:*** statement, followed by a block of code which handles the problem.

```
try:
   a = (12/0)
except ZeroDivisonError:
   print('You cannot divide by zero')
except KeyboardInterrupt:
   print('Ctrl + c pressed')
except:                    # If there is any other exception, then execute this block.
   print('An exception occurred')
```

You can also use the same except statement to handle multiple exceptions as follows:

```
try:
   print(12/0)
except ZeroDivisonError, KeyboardInterrupt:
   print('An exception occurred')
```

The finally Clause

You can use a ***finally:*** block along with a ***try:*** block. The ***finally:*** block is a place to put any code that must execute, whether the ***try:*** block raised an exception or not.

```
try:
    file = open('demo.txt', 'r')
    print(file.read())
except IOError:
    print('Input output Error')
finally:
    file.close()
```

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
try:
   raise NameError('HiThere')
except NameError:
  print('An exception flew by!')
  raise

An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

Exception Chaining

Exception chaining happens automatically when an exception is raised inside an except or finally section. Exception chaining can be disabled by using from None idiom:

```
try:
  open('database.sqlite')
except IOError:
  raise RuntimeError from None

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Python Standard Exceptions

Exception	**Description**
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object
SystemExit	Raised by the sys.exit() function
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit
ArithmeticError	Base class for all errors that occur for numeric calculation
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type
FloatingPointError	Raised when a floating point calculation fails
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types
AssertionError	Raised in case of failure of the Assert statement
AttributeError	Raised in case of failure of attribute reference or assignment
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached
ImportError	Raised when an import statement fails
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c
LookupError	Base class for all lookup errors
IndexError	Raised when an index is not found in a sequence
KeyError	Raised when the specified key is not found in the dictionary
NameError	Raised when an identifier is not found in the local or global namespace
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it
EnvironmentError	Base class for all exceptions that occur outside the Python environment
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist
OSError	Raised for operating system-related errors
SyntaxError	Raised when there is an error in Python syntax
IndentationError	Raised when indentation is not specified properly
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type

Exception	Description
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified
RuntimeError	Raised when a generated error does not fall into any category
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented

User Defined Exceptions

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. However, sometimes you may need to create custom exceptions that serves your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived form this class.

Example: User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

```
class ValueTooSmallError(Exception):
  """Raised when the input value is too small"""
  pass

class ValueTooLargeError(Exception):
  """Raised when the input value is too large"""
  pass


number = 10

while True:
  try:
     i_num = int(input("Enter a number: "))
     if(i_num < number):
        raise ValueTooSmallError
     elif(i_num > number):
        raise ValueTooLargeError
     else:
        break
  except ValueTooSmallError:
     print("This value is too small, try again!")
     print()
  except ValueTooLargeError:
```

```
    print("This value is too large, try again!")
    print()

print("Congratulations! You guessed it correctly.")
```

Output:

```
Enter a number: 12
This value is too large, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

Defining Clean-up Actions

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
try:
  raise KeyboardInterrupt
finally:
  print('Goodbye, world!')

Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

If a finally clause is present, the finally clause will execute as the last task before the try statement completes. The finally clause runs whether or not the try statement produces an exception.

For example:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

>>> divide(2, 1)
```

```
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
```

As you can see, the finally clause is executed in any event. In real world applications, the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines.

Graphical User Interfaces(GUI)

Introduction

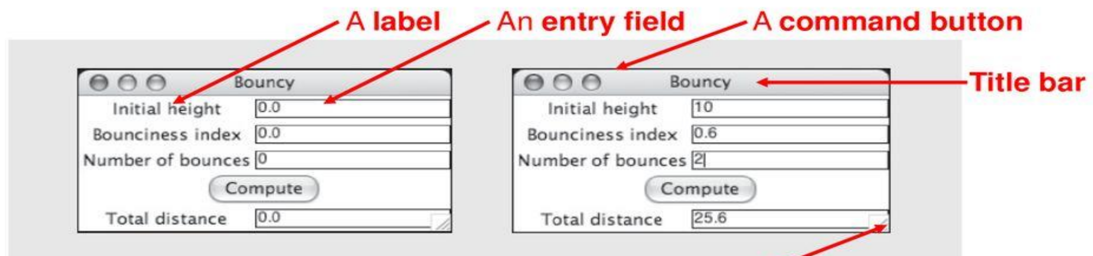
Most modern computer software employs a graphical user interface or GUI. A GUI displays text as well as small images (called icons) that represent objects such as directories, files of different types, command buttons, and dropdown menus. In addition to entering text at keyboard, the user of a GUI can select an icon with pointing device, such as mouse, and move that icon around on the display.

The Behavior of Terminal-Based Programs and GUI-Based Programs

• Two different versions of the bouncy program from a user's point of view:

The GUI-Based Version

- Uses a window that contains various components
  - Called **window objects** or **widgets**



[FIGURE 9.2] A GUI-based **bouncy** program

- Solves problems of terminal-based version

The Terminal-Based Version

```
Welcome to the bouncy program!

1 Compute the total distance
2 Quit the program

Enter a number: 1

Enter the initial height: 10
Enter the bounciness index: .6
Enter the number of bounces: 2

The total distance is 25.6

1 Compute a distance
2 Quit the program

Enter a number: 2
```

[FIGURE 9.1] A session with the terminal-based **bouncy** program

Fundamentals of Python: From First Programs Through Data Structures 5

• Both programs perform exactly the same function – However, their behavior, or look and feel, from a user's perspective are quite different.
• Problems: – User is constrained to reply to a definite sequence of prompts for inputs
• Once an input is entered, there is no way to change it – To obtain results for a different set of input data, user must wait for command menu to be displayed again
• At that point, the same command and all of the other inputs must be re-entered – User can enter an unrecognized command

The GUI-Based Version

• Uses a window that contains various components – Called window objects or widgets
• Solves problems of terminal-based version: Title bar, A label, An entry field, A command button.
• Event-Driven Programming **:** User-generated events (e.g., mouse clicks) trigger operations in program to respond by pulling in inputs, processing them, and displaying results.

Coding phase:

- Define a new class to represent the main window
- Instantiate the classes of window objects needed for this application (e.g., labels, command buttons).
- Position these components in the window.
- Instantiate the data model and provide for the display of any default data in the window objects .

- Register controller methods with each window object in which a relevant event might occur.
- Define these controller methods.
- Define a main that launches the GUI.

Coding Simple GUI-Based Programs

• There are many libraries and toolkits of GUI components available to the Python programmer – tkinter includes classes for windows and numerous types of window objects.

Create GUI applications

First, we will import Tkinter package and create a window and set its title:

```
from tkinter import *
window = Tk()
window.title("Welcome to tkinter")
window.mainloop()
```

The last line which calls mainloop function, this function calls the endless loop of the window, so the window will wait for any user interaction till we close it.
If you forget to call the mainloop function, nothing will appear to the user.

Create a label

To add a label to our previous example, we will create a label using the label class like this:

```
lbl = Label(window, text="Hello")
```

Then we will set its position on the form using the grid function and give it the location like this:

```
lbl.grid(column=0, row=0)
```

So the complete code will be like this:

```
from tkinter import *
window = Tk()
lbl = Label(window, text="Hello")
lbl.grid(column=0, row=0)
window.mainloop()
```

You can set the label font so you can make it bigger and maybe bold. You can also change the font style. To do so, you can pass the font parameter like this:

```
lbl = Label(window, text="Hello", font=("Arial Bold", 50))
```

Setting window size

We can set the default window size using geometry function like this:

```
window.geometry('350x200')
```

The above line sets the window width to 350 pixels and the height to 200 pixels.

Adding a button

Let’s start by adding the button to the window, the button is created and added to the window the same as the label:

```
btn = Button(window, text="Click Me")
```

```
btn.grid(column=1, row=0)
```

Get input using Entry class (Tkinter textbox)

You can create a textbox using Tkinter Entry class like this:

```
txt = Entry(window,width=10)
```

Then you can add it to the window using grid function as usual.

```
from tkinter import *
window = Tk()
window.geometry('350x200')
lbl1 = Label(window, text="enter the first value")
lbl1.grid(column=0, row=0)
lbl2 = Label(window, text="enter the second value")
lbl2.grid(column=0, row=1)
txt1 = Entry(window,width=10)
txt1.grid(column=1, row=0)
txt2 = Entry(window,width=10)
txt2.grid(column=1, row=1)
txt3 = Entry(window,width=10)
txt3.grid(column=2, row=1)
def clicked():
   res=int(txt1.get())+int(txt2.get())
   txt3.insert(0,"result is {}".format(res))

btn = Button(window, text="Click Me", command=clicked)
btn.grid(column=2, row=0)
window.mainloop()
```

The above code creates a window with 2 labels, 3 text fields and reads two numbers from the user and finally displays the result in the third text field.

Add a combobox

To add a combobox widget, you can use the Combobox class from ttk library like this:

```
from tkinter import *

combo = Combobox(window)
```

```
from tkinter import *
window = Tk()
window.geometry('350x200')
combo = Combobox(window)
combo['values']= (1, 2, 3, 4, 5, "Text")
combo.current(1) #set the selected item
combo.grid(column=0, row=0)
window.mainloop()
```

To get the selected item, you can use the get function like this:

```
combo.get()
```

Add a Checkbutton (Tkinter checkbox)
To create a checkbutton, you can use Checkbutton class like this:

```
chk = Checkbutton(window, text='Choose')
```

the following program creates a check box.

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
chk_state = BooleanVar()
chk_state.set(True) #set check state
chk = Checkbutton(window, text='Choose', var=chk_state)
chk.grid(column=0, row=0)
window.mainloop()
```

Here we create a variable of type BooleanVar which is not a standard Python variable, it's a Tkinter variable, and then we pass it to the Checkbutton class to set the check state as the highlighted line in the above example. You can set the Boolean value to false to make it unchecked.

Add radio buttons widgets
To add radio buttons, simply you can use RadioButton class like this:

```
rad1 = Radiobutton(window,text='First', value=1)
```

Note that you should set the value for every radio button with a different value, otherwise, they won't work.

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
window.geometry('350x200')
rad1 = Radiobutton(window,text='First', value=1)
rad2 = Radiobutton(window,text='Second', value=2)
rad3 = Radiobutton(window,text='Third', value=3)
rad1.grid(column=0, row=0)
rad2.grid(column=1, row=0)
rad3.grid(column=2, row=0)
window.mainloop()
```

Create a MessageBox
To show a message box using Tkinter, you can use messagebox library like this:

```
from tkinter  import  *
window = Tk()
window.geometry('350x200')
def clicked():
```

```
    messagebox.showinfo('Message title', 'Message content')
btn = Button(window,text='Click here', command=clicked)
btn.grid(column=0,row=0)
window.mainloop()
```

You can show a warning message or error message the same way. The only thing that needs to be changed is the message function.

Show warning and error messages

You can show a warning message or error message the same way. The only thing that needs to be changed is the message function

```
messagebox.showwarning('Message title', 'Message content')  #shows warning message
messagebox.showerror('Message title', 'Message content')
```

You can choose the appropriate message style according to your needs. Just replace the showinfo function line from the previous line and run it.

Show askquestion dialogs

To show a yes no message box to the user, you can use one of the following messagebox functions:

```
from tkinter import messagebox
res = messagebox.askquestion('Message title','Message content')
res = messagebox.askyesno('Message title','Message content')
res = messagebox.askyesnocancel('Message title','Message content')
res = messagebox.askokcancel('Message title','Message content')
res = messagebox.askretrycancel('Message title','Message content')
```

- If you click OK or yes or retry, it will return True value, but if you choose no or cancel, it will return False.
- The only function that returns one of three values is askyesnocancel function, it returns True or False or None.

Add a SpinBox (numbers widget)

To create a Spinbox widget, you can use Spinbox class like this:

```
spin = Spinbox(window, from_=0, to=100)
```

Here we create a Spinbox widget and we pass the from_ and to parameters to specify the numbers range for the Spinbox.

Also, you can specify the width of the widget using the width parameter:

```
spin = Spinbox(window, from_=0, to=100, width=5)
```