

UNIT-V

Errors and Exceptions

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Two types of Error occurs in python.

- Syntax errors
- Logical errors (Exceptions)

Topic1: Syntax Errors

Syntax errors:

Syntax errors are detected when we have not followed the rules of the particular programming language while writing a program. These errors are also known as **parsing errors**.

Syntax errors are mistakes in the source code, such as spelling and punctuation errors, incorrect labels, and so on, which cause an error message to be generated by the compiler. These appear in a separate error window, with the error type and line number indicated so that it can be corrected in the edit window.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

```
amount = 10000

# check that You are eligible to
# purchase Dsa Self Paced or not

if(amount>2999)
    print("You are eligible to purchase Dsa Self Paced")
```

Output:

Output:

```
File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
        ^
SyntaxError: invalid syntax
```

logical errors(Exception):

- A logic error is an error in a program's source code that gives way to unanticipated and erroneous behavior.

- A logic error is classified as a type of runtime error that can result in a program producing an incorrect output. It can also cause the program to crash when running.
- Logic errors are not always easy to recognize immediately.

When in the runtime an error that occurs after passing the syntax test is called exception or logical type.

For example, when we divide any number by zero then the `ZeroDivisionError` exception is raised, or when we import a module that does not exist then `ImportError` is raised.

Ex:

```
marks = 10000
```

```
# perform division with 0
```

```
a = marks / 0
```

```
print(a)
```

Output:

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

Topic2:Exceptions

1. An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.
2. Whenever an exception occurs, the program stops the execution, and thus the further code is not executed.
3. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python has many built-in exceptions that enable our program to run without interruption and give the output. These exceptions are given below:

ZeroDivisionError: Occurs when a number is divided by zero.

NameError: It occurs when a name is not found. It may be local or global.

IndentationError: If incorrect indentation is given.

IOError: It occurs when Input Output operation fails.

EOFError: It occurs when the end of the file is reached, and yet operations are being performed.

Python Programming

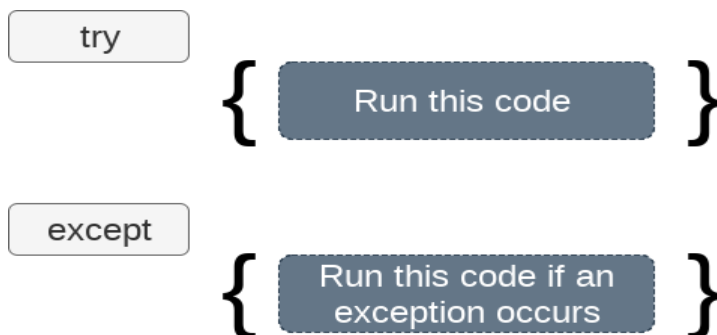
Topic 3: Exception handling in python

- The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc.
- Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files.
- Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.
- Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

The try-except statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the try block.

The try block must be followed with the except statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax:

try :

#statements in try block

except :

#executed when error in try block

For ex:

```
a=5
```

```
b=2
```

```
print(a/b)
```

```
print("Bye")
```

Output:

```
2.5
```

```
Bye
```

The above is normal execution with no error, but if we say when b=0, it is a critical and gives error, see below

```
a=5
```

```
b=0
```

```
print(a/b)
```

```
print("bye") #this has to be printed, but abnormal termination
```

Output:

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python3832/pyyy/ex2.py", line

3, in <module>

```
print(a/b)
```

ZeroDivisionError: division by zero

□ To overcome this we handle exceptions using except keyword

```
a=5
```

```
b=0
```

```
try:
```

```
print(a/b)
```

Python Programming

except Exception:

```
print("number can not be divided by zero")
```

```
print("bye")
```

Output:

number can not be divided by zero

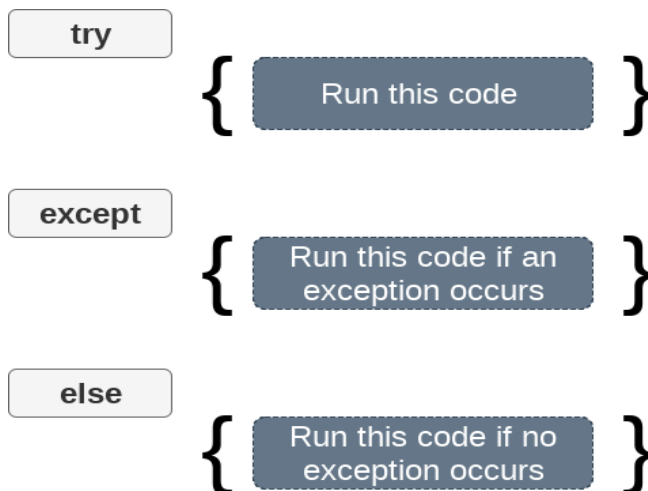
Bye

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```



Ex:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return except
ion class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

Enter a:10

Enter b:0

can't divide by zero

<class 'Exception'>

Topic 4: Raising Exceptions

- Each time an error is detected in a program, the Python interpreter raises (throws) an exception.
- Exception handlers are designed to execute when a specific exception is raised. Programmers can also forcefully raise exceptions in a program using the raise and assert statements.
- Once an exception is raised, no further statement in the current block of code is executed. So, raising an exception involves interrupting the normal flow execution of program and jumping to that part of the program (exception handler code) which is written to handle such exceptional situations.

Syntax:

raise exception-name[(optional argument)]

- To raise an exception, the raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type.

Ex:

try:

```
age = int(input("Enter the age:"))
```

```
if(age<18):
```

```
    raise ValueError
```

```
else:
```

```
    print("the age is valid")
```

```
except ValueError:
```

```
    print("The age is not valid")
```

Output:

Enter the age:17
The age is not valid

Topic 5: User-Defined Exceptions

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause.

However, we suggest you read this section after visiting the Python object and classes.

Ex:

```
class ErrorInCode(Exception):
```

```
    def __init__(self, data):
```

```
        self.data = data
```

```
    def __str__(self):
```

```
        return repr(self.data)
```

```
try:
```

```
    raise ErrorInCode(2000)
```

```
except ErrorInCode as ae:
```

```
    print("Received error:", ae.data)
```

Output:

Received error: 2000

User defined exceptions can be implemented by raising an exception explicitly, by using assert statement or by defining custom classes for user defined exceptions.

Using Assert statements to implement user defined exceptions in python:

We can use assert statement to implement constraints on values of our variable in python. When, the condition given in assert statement is not met, the program gives AssertionError in output.

The syntax for assert statement in python is

assert condition

where condition can be any conditional statement which evaluates to True or False.

Ex:

```
age= 10
print("Age is:")
print(age)
assert age>0
yearOfBirth= 2021-age
print("Year of Birth is:")
print(yearOfBirth)
```

Output:

```
Age is:
10
Year of Birth is:
2011
```

Topic 6: Defining Clean-up Actions, Redefined Cleanup Actions.

1. Clean up actions are those statements within a program that are always executed.
2. These statements are executed even if there is an error in the program.
3. If we have used exception handling in our program then also these statements get executed.
4. In Python, we use finally keyword to state the part of the code that is going to execute every time the program runs. That is every code line under finally is clean up action.

Ex:

```
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
```



```
except ZeroDivisionError:
```

```
    print("Sorry ! You are dividing by zero ")
```

```
else:
```

```
    print("Yeah ! Your answer is:", result)
```

```
finally:
```

```
    print("I'm finally clause, always raised !! ")
```

```
# Look at parameters and note the working of Program
```

```
divide(3, 2)
```

Output:

```
Yeah ! Your answer is : 1
```

```
I'm finally clause, always raised !!
```

Graphical User Interfaces

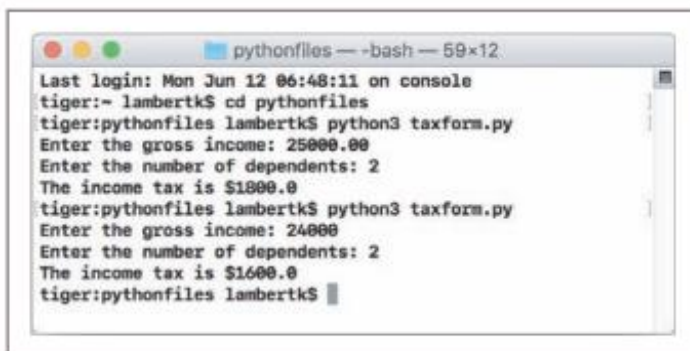
Most people do not judge a book by its cover. They are interested in its contents, not its appearance. However, users judge a software product by its user interface because they have no other way to access its functionality.

graphical user interface or GUI (or its close relative, the touchscreen interface). A GUI displays text as well as small images (called icons) that represent objects such as folders, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select some of these icons with a pointing device, such as a mouse, and move them around on the display.

Topic7: The Behavior of Terminal Based Programs and GUI - Based Programs

The Behavior of Terminal-Based Programs:

The terminal-based version of the program prompts the user for his gross income and number of dependents. After he enters his inputs, the program responds by computing and displaying his income tax. The program then terminates execution. A sample session with this program is shown in Figure



```
pythonfiles -- bash -- 59x12
Last login: Mon Jun 12 06:48:11 on console
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 25000.00
Enter the number of dependents: 2
The income tax is $1800.0
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 24000
Enter the number of dependents: 2
The income tax is $1600.0
tiger:pythonfiles lambertk$
```

Figure 8-1 A session with the terminal-based tax calculator program

This terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.

Python Programming

- To obtain results for a different set of input data, the user must run the program again.

At that point, all of the inputs must be re-entered.

Each of these effects poses a problem for users that can be solved by converting the interface to a GUI.

GUI-Based Programs:

- The GUI-based version of the program displays a window that contains various components, also called widgets.
- Some of these components look like text, while others provide visual cues as to their use.

Below Figure shows snapshots of a sample session with this version of the program. The snapshot on the left shows the interface at program start-up, whereas the snapshot on the right shows the interface after the user has entered inputs and clicked the Compute button. This program was run on a Macintosh; on a Windows- or Linuxbased PC, the windows look slightly different.

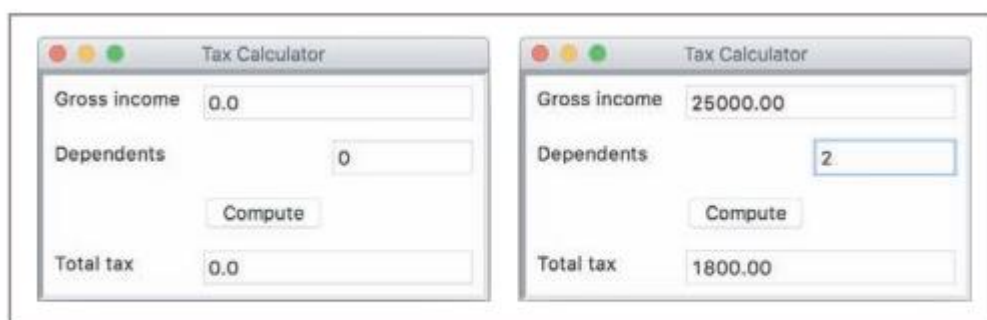


Figure 8-2 A GUI-based tax calculator program

The window in Figure 8-2 contains the following components:

- A title bar at the top of the window. This bar contains the title of the program, “Tax Calculator.” It also contains three colored disks. Each disk is a command button.

- A set of labels along the left side of the window. These are text elements that describe the inputs and outputs. For example, “Gross income” is one label.
- A set of entry fields along the right side of the window. These are boxes within which the program can output text or receive it as input from the user. The first two entry fields will be used for inputs, while the last field will be used for the output. At program start-up, the fields contain default values, as shown in the window on the left side of Figure 8-2.
- A single command button labeled Compute. When the user uses the mouse to press this button, the program responds by using the data in the two input fields to compute the income tax. This result is then displayed in the output field. Sample input data and the corresponding output are shown in the window on the right side of Figure 8-2.
- The user can also alter the size of the window by holding the mouse on its lower-right corner and dragging in any direction.

Topic 8: Coding Simple GUI-Based Programs

In this section, we show some examples of simple GUI-based programs in Python. Python’s standard **tkinter** module includes classes for windows and numerous types of window components, but its use can be challenging for beginners.

open-source module called breezypythongui, while occasionally relying upon some of the simpler resources of tkinter. You will find the code, documentation, and installation instructions for the breezypythongui module at <http://home.wlu.edu/~lambertk/bre>

A Simple “Hello World” Program

Our first demo program defines a class for a main window that displays a greeting. Figure 8-3 shows a screenshot of the window.

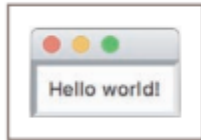


Figure 8-3 Displaying a label with text in a window

A Template for All GUI Programs:

```
from breezypythongui import EasyFrame
```

```
Other imports
```

```
class ApplicationName(EasyFrame):
```

```
    The __init__ method definition
```

```
    Definitions of event handling methods
```

```
def main():
```

```
    ApplicationName().mainloop()
```

```
if __name__ == "__main__":
```

```
    main()
```

The Syntax of Class and Method Definitions:

Note that the syntax of class and method definitions is a bit like the syntax of function definitions. Each definition has a one-line header that begins with a keyword (class or def), followed by a body of code indented one level in the text.

A class header contains the name of the class, conventionally capitalized in Python, followed by a parenthesized list of one or more parent classes

A method header looks very much like a function header, but a method always has at least one parameter, in the first position, named self.

```
def someMethod(self):
```

the method call

```
anObject.someMethod()
```

Topic 9: Other Useful GUI Resources

1:

Using Nested Frames to Organize Components

Suppose that a GUI requires a row of three command buttons beneath two columns of labels and text fields, as shown in Figure 8-14.

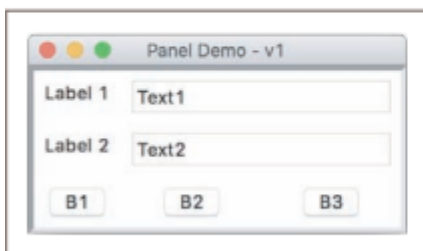


Figure 8-14 Widgets in uneven columns

2:

Multi-Line Text Areas

Although text fields are useful for entering and displaying single lines of text, some applications need to display larger chunks of text with multiple lines. For instance, the message box introduced earlier displays a multi-line message in a scrolling text area. In a manner similar to the editing window of a word processor, a text area widget allows the program to output and the user to input and edit multiple lines of text.

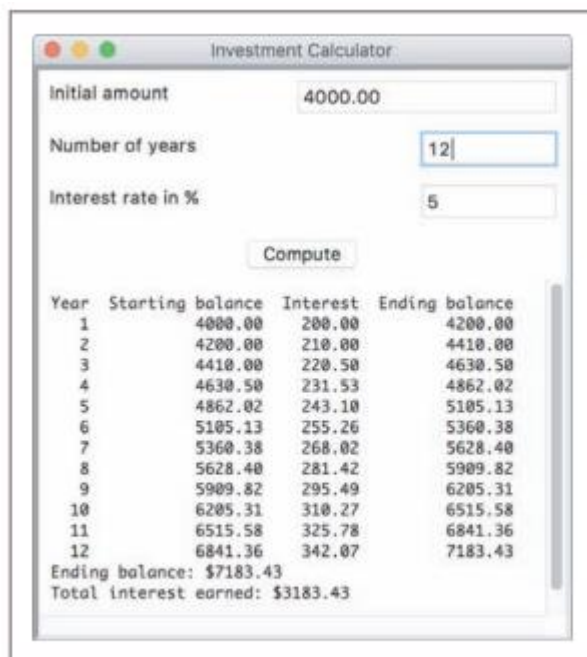


Figure 8-16 Displaying data in a multi-line text area

3:

Obtaining Input with Prompter Boxes

You have seen the advantages of displaying fields for multiple inputs in the same window: you can enter them in any order and change just one or two of them to explore “what if” situations in data processing. However, occasionally you might want to guide the user rigidly through a sequence of inputs, in the manner of terminal-based programs. For example, at start-up a program might prompt the user for a username and then for a password, after launching the main window of the application. GUI applications use a popup dialog called a **prompter box** for this purpose. Figure 8-19 shows a prompter box requesting a username.



Figure 8-19 Using a prompter box

4:

Check Buttons

A **check button** consists of a label and a box that a user can select or deselect with the mouse. Check buttons often represent a group of several options, any number of which may be selected at the same time. The application program can either respond immediately when a check button is manipulated, or examine the state of the button at a later point in time.

As a simple example, let's assume that a restaurant serves chicken dinners with a standard set of sides. These include French fries, green beans, and applesauce. A customer can omit any of the sides from her order, and vegetarians will want to omit the chicken. The user selects these options via check buttons and clicks the **Place order** button to place her order. A message box then pops up with a summary of the order. Figure 8-20 shows the user interface for the program (**checkbuttondemo.py**).

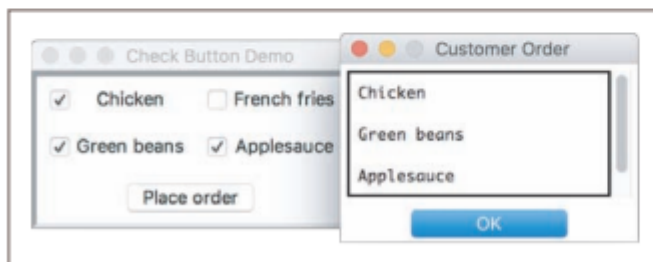


Figure 8-20 Using check buttons

5:Radio Buttons

Check buttons allow a user to select multiple options in any combination. When the user must be restricted to one selection only, the set of options can be presented as a group of radio buttons. Like a check button, a radio button consists of a label and a control widget. One of the buttons is normally selected by default at program start-up. When the user selects a different button in the same group, the previously selected button automatically deselects.

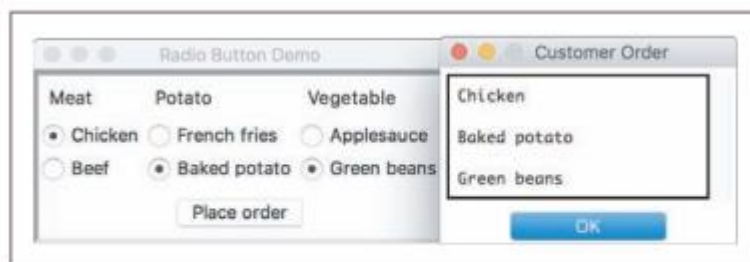


Figure 8-21 Using radio buttons