Python Modules

In this tutorial, we will explain how to construct and import custom Python modules. Additionally, we may import or integrate Python's built-in modules via various methods.

What is Modular Programming?

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Modularizing our code in a big application has a lot of benefits.

Simplification: A module often concentrates on one comparatively small area of the overall problem instead of the full task. We will have a more manageable design problem to think about if we are only concentrating on one module. Program development is now simpler and much less vulnerable to mistakes.

Flexibility: Modules are frequently used to establish conceptual separations between various problem areas. It is less likely that changes to one module would influence other portions of the program if modules are constructed in a fashion that reduces interconnectedness. (We might even be capable of editing a module despite being familiar with the program beyond it.) It increases the likelihood that a group of numerous developers will be able to collaborate on a big project.

Reusability: Functions created in a particular module may be readily accessed by different sections of the assignment (through a suitably established api). As a result, duplicate code is no longer necessary.

Scope: Modules often declare a distinct namespace to prevent identifier clashes in various parts of a program.

In Python, modularization of the code is encouraged through the use of functions, modules, and packages.

What are Modules in Python?

A document with definitions of functions and various statements written in Python is called a Python module.

In Python, we can define a module in one of 3 ways:

- Python itself allows for the creation of modules.

- Similar to the re (regular expression) module, a module can be primarily written in C programming language and then dynamically inserted at run-time.
- A built-in module, such as the itertools module, is inherently included in the interpreter.

A module is a file containing Python code, definitions of functions, statements, or classes. An example_module.py file is a module we will create and whose name is example_module.

We employ modules to divide complicated programs into smaller, more understandable pieces. Modules also allow for the reuse of code.

Rather than duplicating their definitions into several applications, we may define our most frequently used functions in a separate module and then import the complete module.

Let's construct a module. Save the file as example_module.py after entering the following.

Example:

```
# Here, we are creating a simple Python program to show how to create a module.
# defining a function in the module to reuse it
def square( number ):
    # here, the above function will square the number passed as the input
    result = number ** 2
    return result     # here, we are returning the result of the function
```

Here, a module called example_module contains the definition of the function square(). The function returns the square of a given number.

How to Import Modules in Python?

In Python, we may import functions from one module into our program, or as we say into, another module.

For this, we make use of the import Python keyword. In the Python window, we add the next to import keyword, the name of the module we need to import. We will import the module we defined earlier example_module.

Syntax:

```
import example_module
```

The functions that we defined in the example_module are not immediately imported into the present program. Only the name of the module, i.e., example_ module, is imported here.

We may use the dot operator to use the functions using the module name. For instance:

Example:

```
# here, we are calling the module square method and passing the value 4
result = example_module.square(  4  )
print("By using the module square of number is: ", result )
```

Output:

By using the module square of number is: 16

There are several standard modules for Python. The complete list of Python standard modules is available. The list can be seen using the help command.

Similar to how we imported our module, a user-defined module, we can use an import statement to import other standard modules.

Importing a module can be done in a variety of ways. Below is a list of them.

Python import Statement

Using the import Python keyword and the dot operator, we may import a standard module and can access the defined functions within it. Here's an illustration.

Code

```
# Here, we are creating a simple Python program to show how to import a standard module
# Here, we are import the math module which is a standard module
import math
print( "The value of euler's number is", math.e )
# here, we are printing the euler's number from the math module
```

Output:

The value of euler's number is 2.718281828459045

Importing and also Renaming

While importing a module, we can change its name too. Here is an example to show.

Code

```
# Here, we are creating a simple Python program to show how to import a module a nd rename it
# Here, we are import the math module and give a different name to it
import math as mt     # here, we are importing the math module as mt
```

```
4. print( "The value of euler's number is", mt.e )
5. # here, we are printing the euler's number from the math module
```

Output:

The value of euler's number is 2.718281828459045
The math module is now named mt in this program. In some situations, it might help us type faster in case of modules having long names.

Please take note that now the scope of our program does not include the term math. Thus, mt.pi is the proper implementation of the module, whereas math.pi is invalid.

Python from...import Statement

We can import specific names from a module without importing the module as a whole. Here is an example.

Code

```
1. # Here, we are creating a simple Python program to show how to import specific
2. # objects from a module
3. # Here, we are import euler's number from the math module using the from keywor
   d
4. from math import e
5. # here, the e value represents the euler's number
6. print( "The value of euler's number is", e )
```

Output:

The value of euler's number is 2.718281828459045
Only the e constant from the math module was imported in this case.

We avoid using the dot (.) operator in these scenarios. As follows, we may import many attributes at the same time:

Code

```
1. # Here, we are creating a simple Python program to show how to import multiple
2. # objects from a module
3. from math import e, tau
4. print( "The value of tau constant is: ", tau )
5. print( "The value of the euler's number is: ", e )
```

Output:

The value of tau constant is: 6.283185307179586

The value of the euler's number is: 2.718281828459045

Import all Names - From import * Statement

To import all the objects from a module within the present namespace, use the * symbol and the from and import keyword.

Syntax:

1. **from** name_of_module **import** *

There are benefits and drawbacks to using the symbol *. It is not advised to use * unless we are certain of our particular requirements from the module; otherwise, do so.

Here is an example of the same.

Code

```
# Here, we are importing the complete math module using *
from math import *
# Here, we are accessing functions of math module without using the dot operator
print( "Calculating square root: ", sqrt(25) )
# here, we are getting the sqrt method and finding the square root of 25
print( "Calculating tangent of an angle: ", tan(pi/6) )
# here pi is also imported from the math module
```

Output:

Calculating square root: 5.0
Calculating tangent of an angle: 0.5773502691896257

Locating Path of Modules

The interpreter searches numerous places when importing a module in the Python program. Several directories are searched if the built-in module is not present. The list of directories can be accessed using sys.path. The Python interpreter looks for the module in the way described below:

The module is initially looked for in the current working directory. Python then explores every directory in the shell parameter PYTHONPATH if the module cannot be located in the current directory. A list of folders makes up the environment variable known as PYTHONPATH. Python examines the installation-dependent set of folders set up when Python is downloaded if that also fails.

Here is an example to print the path.

Code

```
# Here, we are importing the sys module
import sys
# Here, we are printing the path using sys.path
print("Path of the sys module in the system is:", sys.path)
```

Output:

Path of the sys module in the system is:
['/home/pyodide', '/home/pyodide/lib/Python310.zip', '/lib/Python3.10', '/lib/Python3.10/lib-dynload', '', '/lib/Python3.10/site-packages']

The dir() Built-in Function

We may use the dir() method to identify names declared within a module.

For instance, we have the following names in the standard module str. To print the names, we will use the dir() method in the following way:

Code

```
# Here, we are creating a simple Python program to print the directory of a module

print( "List of functions:\n ", dir( str ), end=", " )
```

Output:

List of functions:
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Namespaces and Scoping

Objects are represented by names or identifiers called variables. A namespace is a dictionary containing the names of variables (keys) and the objects that go with them (values).

Both local and global namespace variables can be accessed by a Python statement. When two variables with the same name are local and global, the local variable takes the role of the global

variable. There is a separate local namespace for every function. The scoping rule for class methods is the same as for regular functions. Python determines if parameters are local or global based on reasonable predictions. Any variable that is allocated a value in a method is regarded as being local.

Therefore, we must use the global statement before we may provide a value to a global variable inside of a function. Python is informed that Var_Name is a global variable by the line global Var_Name. Python stops looking for the variable inside the local namespace.

We declare the variable Number, for instance, within the global namespace. Since we provide a Number a value inside the function, Python considers a Number to be a local variable. UnboundLocalError will be the outcome if we try to access the value of the local variable without or before declaring it global.

Code

```
1. Number = 204
2. def AddNumber():  # here, we are defining a function with the name Add Number
3.     # Here, we are accessing the global namespace
4.     global Number
5.     Number = Number + 200
6. print("The number is:", Number)
7. # here, we are printing the number after performing the addition
8. AddNumber()    # here, we are calling the function
9. print("The number is:", Number)
```

Output:

The number is: 204
The number is: 404

Python Exceptions

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception. We will see what an exception is. Also, we will see the difference between a syntax error and an exception in this tutorial. Following that, we will learn about trying and except blocks and how to raise exceptions and make assertions. After that, we will see the Python exceptions list.

What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur. Consider the following scenario:

Code

```
1. #Python code after removing the syntax error
2. string = "Python Exceptions"
3.
4. for s in string:
5.     if (s != o:
6.         print( s )
```

Output:

if (s != o:
^
SyntaxError: invalid syntax

The arrow in the output shows where the interpreter encountered a syntactic error. There was one unclosed bracket in this case. Close it and rerun the program:

Code

```
#Python code after removing the syntax error
string = "Python Exceptions"

for s in string:
    if (s != o):
        print( s )
```

Output:

2 string = "Python Exceptions"
4 for s in string:
----> 5 if (s != o):
6 print(s)

NameError: name 'o' is not defined
We encountered an exception error after executing this code. When syntactically valid Python code produces an error, this is the kind of error that arises. The output's last line specified the name of the exception error code encountered. Instead of displaying just "exception error", Python displays information about the sort of exception error that occurred. It was a NameError in this situation. Python includes several built-in exceptions. However, Python offers the facility to construct custom exceptions.

Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

Code

```
# Python code to catch an exception and handle it using try and except code blocks

a = ["Python", "Exceptions", "try and except"]
try:
    #looping through the elements of the array a, choosing a range that goes beyond t
    he length of the array
    for i in range( 4 ):
        print( "The index and element from the array is", i, a[i] )
#if an error occurs in the try block, then except block will be executed by the Python
    interpreter
except:
```

```
10.     print ("Index out of range")
```

Output:

The index and element from the array is 0 Python
The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range

The code blocks that potentially produce an error are inserted inside the try clause in the preceding example. The value of i greater than 2 attempts to access the list's item beyond its length, which is not present, resulting in an exception. The except clause then catches this exception and executes code without stopping it.

How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

Code

```
1. #Python code to show how to raise an exception in Python
2. num = [3, 4, 5, 7]
3. if len(num) > 3:
4.     raise Exception( f"Length of the given list must be less than or equal to 3 but is {len
   (num)}" )
```

Output:

1 num = [3, 4, 5, 7]
2 if len(num) > 3:
----> 3 raise Exception(f"Length of the given list must be less than or equal to 3 but is {len(num)}")

Exception: Length of the given list must be less than or equal to 3 but is 4

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

The assert Statement

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

The syntax for the assert clause is –

1. **assert** Expressions[, Argument]

Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

Code

```
1. #Python program to show how to use assert keyword
2. # defining a function
3. def square_root( Number ):
4.     assert ( Number < 0), "Give a positive integer"
5.     return Number**(1/2)
6.
7. #Calling function and passing the values
8. print( square_root( 36 ) )
9. print( square_root( -36 ) )
```

Output:

7 #Calling function and passing the values
----> 8 print(square_root(36))
9 print(square_root(-36))

Input In [23], in square_root(Number)
3 def square_root(Number):
----> 4 assert (Number < 0), "Give a positive integer"
*5 return Number**(1/2)*

AssertionError: Give a positive integer

Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

Here is an instance of a try clause with an else clause.

Code

```
# Python program to show how to use else clause with try and except clauses

# Defining a function which returns reciprocal of a number
def reciprocal( num1 ):
    try:
        reci = 1 / num1
    except ZeroDivisionError:
        print( "We cannot divide by zero" )
    else:
        print ( reci )
# Calling the function and passing values
reciprocal( 4 )
reciprocal( 0 )
```

Output:

0.25
We cannot divide by zero

Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Here is an example of finally keyword with try-except clauses:

Code


```
# Python code to show the use of finally clause

# Raising an exception in try block
```

```
try:
    div = 4 // 0
    print( div )
# this block will handle the exception raised
except ZeroDivisionError:
    print( "Atepting to divide by zero" )
# this will always be executed no matter exception is raised or not
finally:
    print( 'This is code of finally clause' )
```

Output:

Atepting to divide by zero
This is code of finally clause

User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

Here is an illustration of a RuntimeError. In this case, a class that derives from RuntimeError is produced. Once an exception is detected, we can use this to display additional detailed information.

We raise a user-defined exception in the try block and then handle the exception in the except block. An example of the class EmptyError is created using the variable var.

Code

```
class EmptyError( RuntimeError ):
  def __init__(self, argument):
    self.arguments = argument
Once the preceding class has been created, the following is how to raise an exceptio
n:
Code
var = " "
try:
    raise EmptyError( "The variable is empty" )
except (EmptyError, var):
    print( var.arguments )
```

Output:

2 try:

----> 3 raise EmptyError("The variable is empty")
4 except (EmptyError, var):

EmptyError: The variable is empty

Python Exceptions List

Here is the complete list of Python in-built exceptions.

Sr.No.	Name of the Exception	Description of the Exception
1	**Exception**	All exceptions of Python have a base class.
2	**StopIteration**	If the next() method returns null for an iterator, this exception is raised.
3	**SystemExit**	The sys.exit() procedure raises this value.
4	**StandardError**	Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions.
5	**ArithmeticError**	All mathematical computation errors belong to this base class.
6	**OverflowError**	This exception is raised when a computation surpasses the numeric data type's maximum limit.
7	**FloatingPointError**	If a floating-point operation fails, this exception is raised.

8	**ZeroDivisionError**	For all numeric data types, its value is raised whenever a number is attempted to be divided by zero.
9	**AssertionError**	If the Assert statement fails, this exception is raised.
10	**AttributeError**	This exception is raised if a variable reference or assigning a value fails.
11	**EOFError**	When the endpoint of the file is approached, and the interpreter didn't get any input value by raw_input() or input() functions, this exception is raised.
12	**ImportError**	This exception is raised if using the import keyword to import a module fails.
13	**KeyboardInterrupt**	If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised.
14	**LookupError**	LookupErrorBase is the base class for all search errors.
15	**IndexError**	This exception is raised when the index attempted to be accessed is not found.

16	**KeyError**	When the given key is not found in the dictionary to be found in, this exception is raised.
17	**NameError**	This exception is raised when a variable isn't located in either local or global namespace.
18	**UnboundLocalError**	This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value.
19	**EnvironmentError**	All exceptions that arise beyond the Python environment have this base class.
20	**IOError**	If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised.
22	**SyntaxError**	This exception is raised whenever a syntax error occurs in our program.
23	**IndentationError**	This exception was raised when we made an improper indentation.
24	**SystemExit**	This exception is raised when the sys.exit() method is used to terminate the Python

		interpreter. The parser exits if the situation is not addressed within the code.
25	**TypeError**	This exception is raised whenever a data type-incompatible action or function is tried to be executed.
26	**ValueError**	This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values.
27	**RuntimeError**	This exception is raised when an error that occurred during the program's execution cannot be classified.
28	**NotImplementedError**	If an abstract function that the user must define in an inherited class is not defined, this exception is raised.

Summary

We learned about different methods to raise, catch, and handle Python exceptions after learning the distinction between syntax errors and exceptions. We learned about these clauses in this tutorial:

- We can throw an exception at any line of code using the raise keyword.
- Using the assert keyword, we may check to see if a specific condition is fulfilled and raise an exception if it is not.
- All statements are carried out in the try clause until an exception is found.
- The try clause's exception(s) are detected and handled using the except function.

- If no exceptions are thrown in the try code block, we can write code to be executed in the else code block.

Here is the syntax of try, except, else, and finally clauses.

Syntax:

```
try:
    # Code block
    # These statements are those which can probably have some error

except:
    # This block is optional.
    # If the try block encounters an exception, this block will handle it.

else:
    # If there is no exception, this code block will be executed by the Python interpreter

finally:
    # Python interpreter will always execute this code.
```