Exception Handling in Python

As human beings, we commit several errors. A software developer is also a human being and hence prone to commit errors either in the design of the software or in writing the code. The errors in the software are called bugs' and the process of removing them is called 'debugging'. Let's learn about different types of errors that can occur in a program.

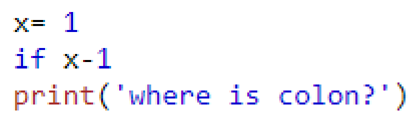
Errors in a Python Program

In general, we can classify errors in a program into one of these three types:

- Compile-time errors
- Runtime errors
- Logical errors
- Compile-Time Errors

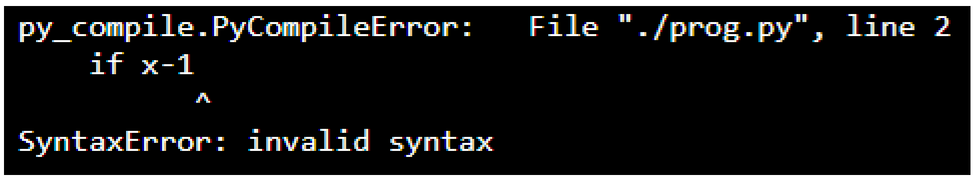
These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a colon in the statements like if, while, for, def, etc. will result in compile-time error. Such errors are detected by the Python compiler and the line number along with error description is displayed by the Python compiler. Let's see a program to understand this better. In this program, we have forgotten to write colon in the if statement, after the condition. This will raise SyntaxError.

A Python program to understand the compile-time error.



```python
x= 1
if x-1
print('where is colon?')
```

Output:



```
py_compile.PyCompileError:   File "./prog.py", line 2
    if x-1
         ^
SyntaxError: invalid syntax
```

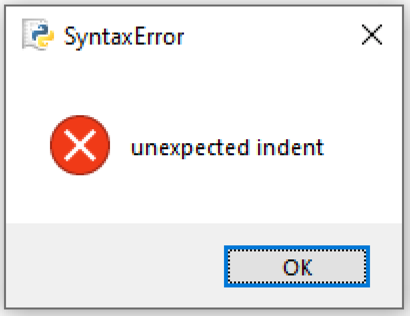
We know that Python statements are written in blocks using proper indentation. The default number of spaces used for indentation is 4. All the statements belonging to the same block should use the same number of spaces before them. If there is any deviation in the spaces, then we can see an IndentationError raised by the Python compiler. Consider the following program where we are using an unequal number of spaces for the print statements into the if statement.

A Python program to demonstrate compile-time error.

```
x = 10
if x%2==0:
        print(x, 'is divisible by 2')
            print(x, 'is even number')
```

Output:

```
x = 10
if x%2==0:
        print(x, 'is divisible by 2')
            print(x, 'is even number')
```


SyntaxError

unexpected indent

The Python compiler displays error message and the line number in case of compile errors. By checking the program source code and rewriting the statements properly, can eliminate the compile-time errors.

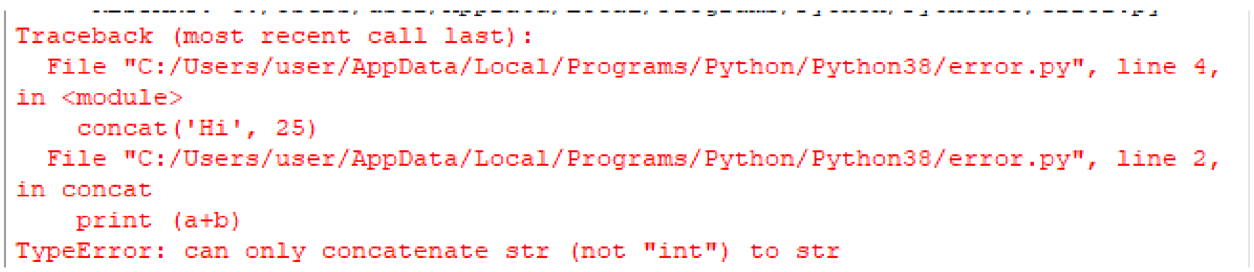
Runtime Errors

When PVM cannot execute the byte code, it flags runtime errors. For example, insufficient memory to store something or inability of the PVM to execute some statement om under runtime errors. Runtime errors are not detected by the Python compiler. They are detected by the PVM, only at runtime. The following program explains this further.

A Python program to understand runtime errors.

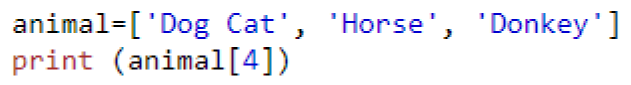
```
def concat(a, b):
    print (a+b)

concat('Hi', 25)
```

Output:



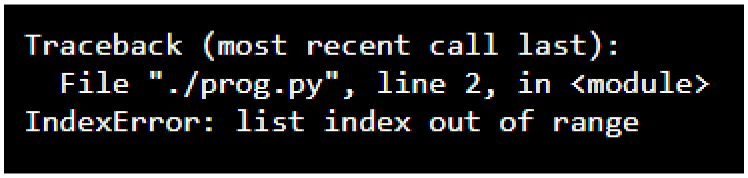
```
Traceback (most recent call last):
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/error.py", line 4,
in <module>
    concat('Hi', 25)
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/error.py", line 2,
in concat
    print (a+b)
TypeError: can only concatenate str (not "int") to str
```

In the program above, we have written a function by the name 'concat' that accepts 2 arguments and b. It adds them using the operator and displays the result. At the time of calling this function, if we pass two strings, they will be concatenated or joined. In this case, '+' acts like a concatenation operator. On the other hand, if we pass 2 numbers, then they are added and the result is displayed. In this case, it acts as an additional operator. Blut, in the above example, we are passing one string and one number. Since the data types are not the same, PVM shows TypeError. In Python, the compiler will not check the data types. Type checking is done by PVM during runtime. Program 4 is also an example for runtime error. In this program, we are creating a list with 4 elements. The indexes (or position numbers) of these elements will be from 0 to 3. When we refer to the index 4 which is not in the list, there will be an IndexError during runtime.
A Python program to demonstrate runtime error.



```
animal=['Dog Cat', 'Horse', 'Donkey']
print (animal[4])
```

Output:

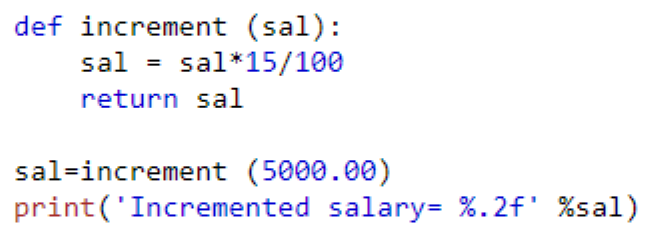


```
Traceback (most recent call last):
  File "./prog.py", line 2, in <module>
IndexError: list index out of range
```

In case of runtime error, the PVM displays the line number and the type of error. Most of the runtime errors can be eliminated by following the message given by PVM. For example, in the previous program, restricting the list index below 4 is the solution to eliminate the runtime error. But some runtime errors cannot be eliminated. In that case, we should handle those errors using the 'exception handling mechanism' of Python.

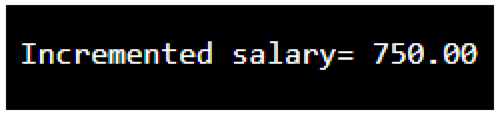
Logical Errors

These errors depict flaws in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Python compiler or PVM. The programmer is solely responsible for them. In the following program, the programmer wants to calculate the increment salary of an employee, but he gets the wrong output, since he uses the wrong formula.
A Python program to increment the salary of an employee by 15%.



```
def increment (sal):
    sal = sal*15/100
    return sal

sal=increment (5000.00)
print('Incremented salary= %.2f' %sal)
```

Output:



```
Incremented salary= 750.00
```

By comparing the output of a program with manually calculated results, a programmer can guess the presence of a logical error. In the program above, we are using the following formula to calculate the incremental salary:

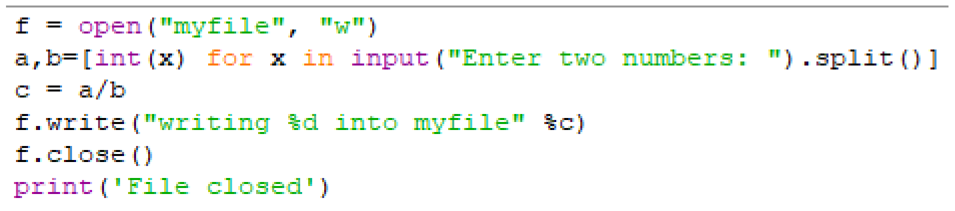
sal=sal*15/100

This is wrong since this formula calculates only the increment but it is not adding it to the original salary. So, the correct formula would be:

sal = sal + sal*15/100

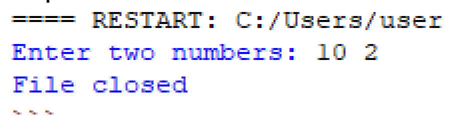
Compile time errors and logical errors can be eliminated by the programmer by modifying the program source code. In case of runtime errors, when the programmer knows which type of error occurs, he has to handle them using an exception handling mechanism. The runtime errors which can be handled by the programmer are called exceptions. Before discussing exception handling mechanisms, we will first understand what type of harm and exception can cause. The program below will help us in this regard.

A Python program to understand the effect of an exception.



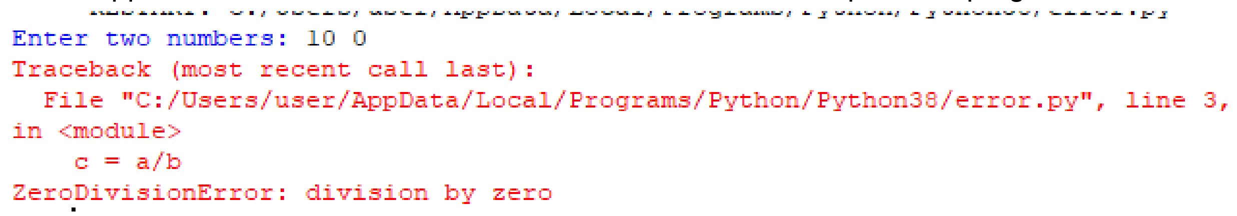
```
f = open("myfile", "w")
a,b=[int(x) for x in input("Enter two numbers: ").split()]
c = a/b
f.write("writing %d into myfile" %c)
f.close()
print('File closed')
```

Output:



```
==== RESTART: C:/Users/user
Enter two numbers: 10 2
File closed
```

When we execute Program 6, it opens a file by the name "myfile" using the open() method, and then writes the result of a/b into that file using the write() method. Finally, the file is closed using the close() method. This program runs well if we give 'a' and b' values from the keyboard as: 10 and 2. Since a/b value is 5, this value (i.e. 5) is stored into the file. After that, the file is closed. What happens if we enter 10 and 0 as values for 'a' and 'b' in the previous program. Let's see



```
Enter two numbers: 10 0
Traceback (most recent call last):
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/error.py", line 3,
in <module>
    c = a/b
ZeroDivisionError: division by zero
```

Since a/b represents 10/0 that gives infinity which is a huge quantity that cannot be stored into any variable, we are getting an error ZeroDivisionError'. When this error occurred, PVM was simply displaying the error message and immediately terminating the program in line number 8. Due to this abnormal termination, the subsequent statements in the program are not executed. Hence, f.close() is not executed and the file which is opened in the beginning of the program is not closed. This leads to loss of entire data that is already present in the file. A file that is opened in any mode should be closed properly. This ensures safety for the data present in the file.

So, when there is an error in a program, due to its sudden termination, the following things can be suspected:

- The important data in the files or databases used in the program may be lost.
- The software may be corrupted.
- The program abruptly terminates giving an error message to the user making the user losing trust in the software.

Hence, it is the duty of the programmer to handle the errors. Please understand that we cannot handle all errors. We can handle only some types of errors which are called exceptions.

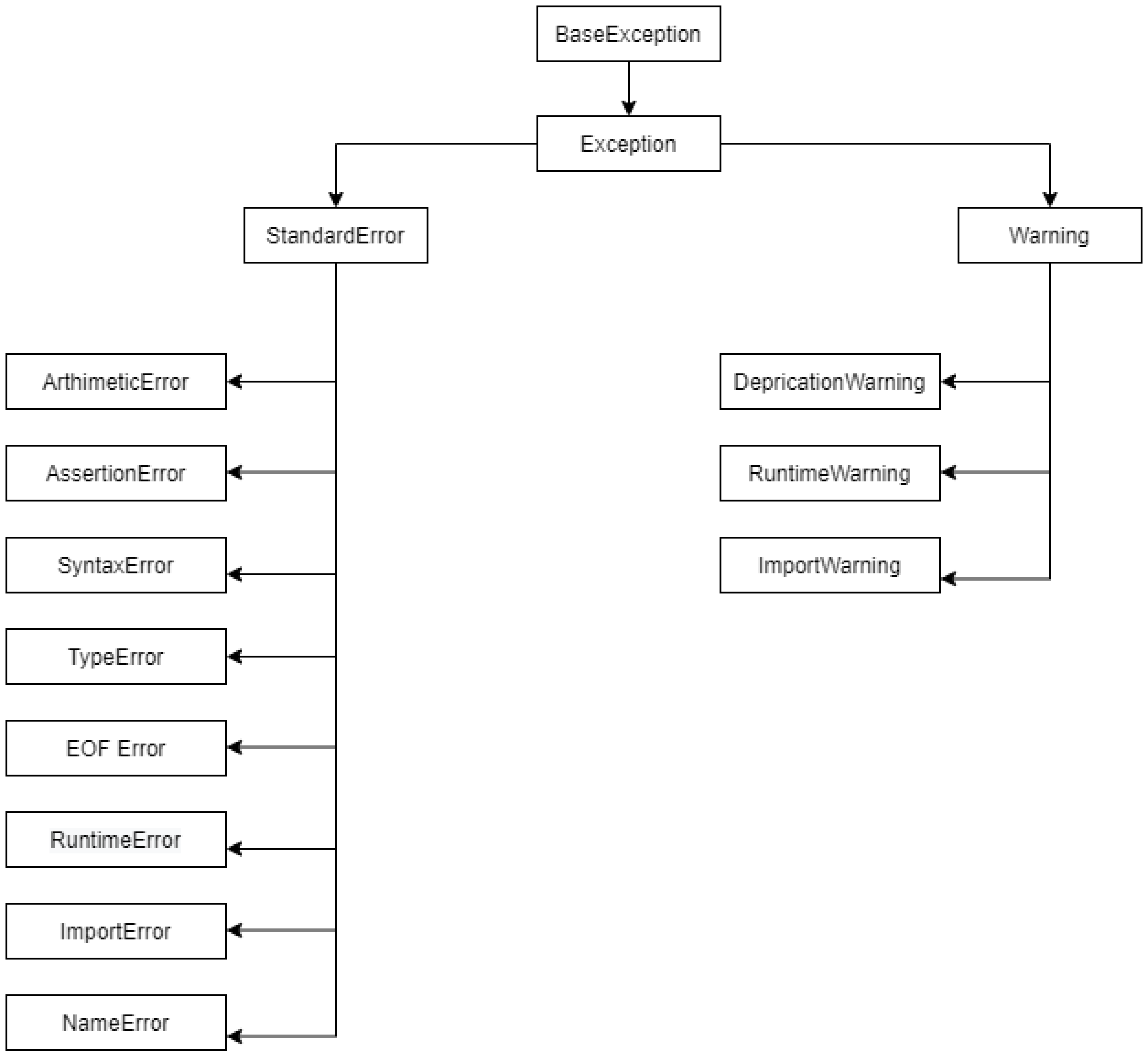
Exceptions

An exception is a runtime error which can be handled by the programmer. That means if the programmer can guess an error in the program and he can do something to eliminate the harm caused by that error, then it is called an 'exception'. If the programmer cannot do anything in case of an error, then it is called an 'error' and not an exception.

All exceptions are represented as classes in Python. The exceptions which are already available in Python are called built-in' exceptions. The base class for all built-in exceptions is 'BaseException' class. From the BaseException class, the sub class 'Exception' is derived. From the Exception class, the subclasses ‘StandardError' and 'Warning' are derived.

All errors (or exceptions) are defined as sub classes of StandardError. An error should be compulsorily handled otherwise the program will not execute. Similarly, all warnings are derived as subclasses from the 'Warning' class. A warning represents a caution and even though it is not handled, the program will execute. So, warnings can be neglected but errors cannot be neglected.

Just like the exceptions which are already available in Python language, a programmer can also create his own exceptions, called 'user-defined' exceptions. When the programmer wants to create his own exception class, he should derive his class from the Exception class and not from BaseException' class.



Exception Handling

The purpose of handling errors is to make the program robust. The word 'robust' means 'strong". A robust program does not terminate in the middle. Also, when there is an error in the program, it will display an appropriate message to the user and continue execution. Designing such programs is needed in any software development. For this the errors. When the errors can be handled,purpose, the programmer should handle them and they are called exceptions.

To handle exceptions, the programmer should perform the following three steps:

Step 1: The programmer should observe the statements in his program where there may be a possibility of exceptions. Such statements should be written inside a ‘try’ block. A
try block looks like as follows:

```
try:
     statements
```

The greatness of try block is that even if some exception arises inside it, the program will not be terminated. When PVM understands that there is an exception, it jumps into an 'except' block.

Step 2: The programmer should write the 'except' block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this except block looks like as follows:

```
except exceptionname:
          statements
```

The statements written inside an except block are called 'handlers' since they handle the situation when the exception occurs.

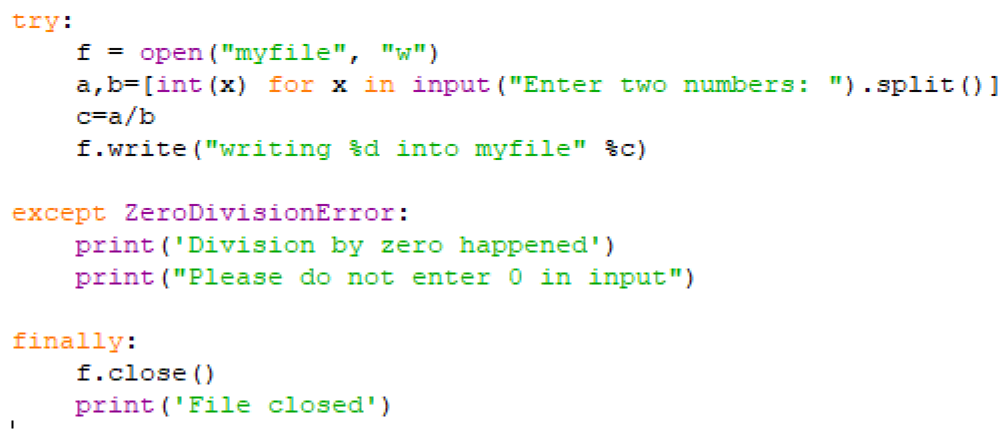
Step 3: Lastly, the programmer should perform clean up actions like closing the files and terminating any other processes which are running. The programmer should write this code in the finally block. Finally block looks like as follows:

```
finally:
      statements
```

The specialty of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running processes are properly terminated. So, the data in the files will not be corrupted and the user is at the safe-side.

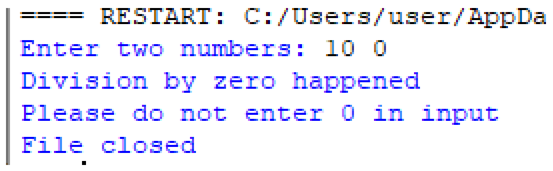
Performing the above 3 tasks is called 'exception handling'. Remember, in exception handling, the programmer is not preventing the exception, as in many cases it is not possible. But the programmer is avoiding any damage that may happen to data anc software. Let's rewrite the previous program to handle the ZeroDivisionError exception using try except and finally blocks..

A Python program to handle the ZeroDivisionError exception. an exception handling example



```
try:
    f = open("myfile", "w")
    a,b=[int(x) for x in input("Enter two numbers: ").split()]
    c=a/b
    f.write("writing %d into myfile" %c)

except ZeroDivisionError:
    print('Division by zero happened')
    print("Please do not enter 0 in input")

finally:
    f.close()
    print('File closed')
```

Output:



```
==== RESTART: C:/Users/user/AppDa
Enter two numbers: 10 0
Division by zero happened
Please do not enter 0 in input
File closed
```

From the preceding output, we can understand that the finally' block is executed and the file is closed in both the cases, i.e. when there is no exception and when the exception occurred. In the previous discussion, we used try-catch-finally to handle the exception. However, the complete exception handling syntax will be in the following format:

```
try:
   statements

except Exception1:
   handler1

except Exception2:
   handler2

else:
   statements
```

```
finally:
    statements
```

The 'try' block contains the statements where there may be one or more exceptions. The subsequent 'except' blocks handle these exceptions. When Exception 1' occurs, 'handler' statements are executed. When 'Exception2' occurs, 'handler2' statements are executed and so forth. If no exception is raised, the statements inside the 'else' block are executed.Even if the exception occurs or does not occur, the code inside finally' block is always executed. The following points are noteworthy:

- A single try block can be followed by several except blocks.
- Multiple except blocks can be used to handle multiple exceptions.
- We cannot write except blocks without a try block.
- We can write a try block without any except blocks.
- Else block and finally blocks are not compulsory.
- When there is no exception, the else block is executed after the try block.
- Finally block is always executed.

Types of Exceptions

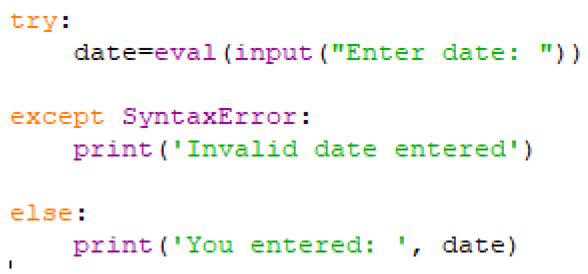
There are several exceptions available as part of Python language that are called built-in exceptions. In the same way, the programmer can also create his own exceptions called user-defined exceptions. The table below summarizes some important built-in exceptions in Python. Most of the exception class names end with the word 'Error'.

Exception Class Name	Description
Exception	Represents any type of exception. All exceptions are subclasses of this class.
ArithmeticError	Represents the base class for arithmetic errors like OverflowError, ZeroDivisionError, FloatingPointError.
AssertionError	Raised when an assert statement gives error.
AttributeError	Raised when an attribute reference or assignment fails.
EOFError	Raised when input() function reaches end without reading any data.
Floating PointError	Raised when a floating point operation fails.
GeneratorExit	Raised when generator's close() method is called

IOError	Raised when an input or output operation failed. It raises when the file opened is not found or when the writing data disk is full.
ImportError	Raised when an import statement fails to find the module being imported.
IndexError	Raised when a sequence index or subscript is out of range
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys.
KeyboardInterrupt	Raised when the user hits the interrupt key (normally Control-C or Delete).
NameError	Raised when an identifier is not found locally or globally.
NotimplementedError	Derived from "RuntimeError. In user defined base classes.abstract methods should raise this exception when they require derived classes to override the method.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise Memory Error)
RuntimeError	Raised when an error is detected that doesn't fall in any of the other categories.
Stopiteration	Raised by an iterator's next() method to signal that there are no more elements.
SyntaxError	Raised when the compiler encounters a syntax error. Import or exec statements and input() and eval() functions may raise this exception.
IndentationError	Raised when indentation is not specified properly.
SystemExit	Raised by the sys.exit() function. When it is not handled, the Python interpreter exits.
TypeError	Raised when an operation or function is applied to an object of inappropriate datatype.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
ValueError	Raised when a built-in operation or function receives an argument that has the right data type but wrong value.
ZeroDivisionError	Raised when the denominator is zero in a division or modulus operation.

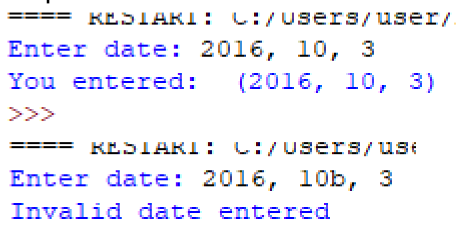
We will now see how to work with built-in exceptions. In the program below, we are trying to handle SyntaxError that is raised by the eval() function. The eval() function accepts input in the form of a list, tuple or dictionary and evaluates the input properly. In this program. we are entering date in the form of year, month and date separating them by comman c.g. 2016, 10, 3. When a group of values are entered separating them by commas, they are understood as a tuple by the eval() function. While entering these values any letter is by mistake typed along with the value, there will be SyntaxError raised by the eval function. This error is caught in the except block and a message Invalid date entered is displayed.

A Python program to handle syntax errors given by eval() function.



```
try:
    date=eval(input("Enter date: "))

except SyntaxError:
    print('Invalid date entered')

else:
    print('You entered: ', date)
```

Output:



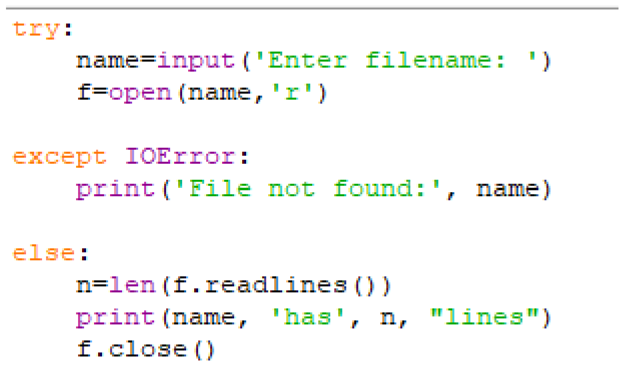
```
==== RESTART: C:/Users/user/
Enter date: 2016, 10, 3
You entered:  (2016, 10, 3)
>>>
==== RESTART: C:/Users/use
Enter date: 2016, 10b, 3
Invalid date entered
```

In the next program, we will accept a file name from the keyboard and then open it using the open() function. If the file is not found, then IOError is raised. Then the 'except' block will display a message: ‘File not found’.If file is found, then all the lines of the file are read using readlines() method as:

n=len(f.readlines())

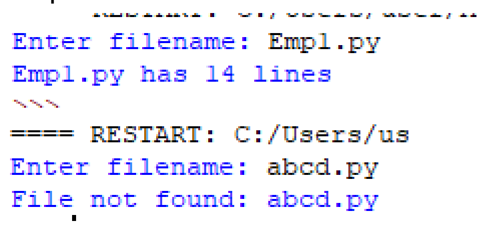
Here, f.readLines() will read the lines available in the file. Then len() function will return their number. This is stored into ‘n’.

A Python program to handle IOError produced by opent function.



```
try:
    name=input('Enter filename: ')
    f=open(name,'r')

except IOError:
    print('File not found:', name)

else:
    n=len(f.readlines())
    print(name, 'has', n, "lines")
    f.close()
```

Output:



```
Enter filename: Empl.py
Empl.py has 14 lines
>>>
==== RESTART: C:/Users/us
Enter filename: abcd.py
File not found: abcd.py
```

The Except Block

The 'except' block is useful to catch an exception that is raised in the try block. When there is an exception in the try block, then only the except block is executed. It is written in various formats.

1. To catch the exception which is raised in the try block, we can write except block with the Exceptionclass name as:

   except Exceptionclass:
2. We can catch the exception as an object that contains some description about the exception.

   except Exceptionclass as obj:

3. To catch multiple exceptions, we can write multiple catch blocks. The other way is to use a single except block and write all the exceptions as a tuple inside parentheses as:

   except (Exceptionclass1, Exceptionclass2,-):

4. To catch any type of exception where we are not bothered about which type of exception it is, we can write except block without mentioning any Exceptionclass name as:

   except:

In the previous program, we are catching two exceptions using two except blocks. The same can be written using a single except block as:

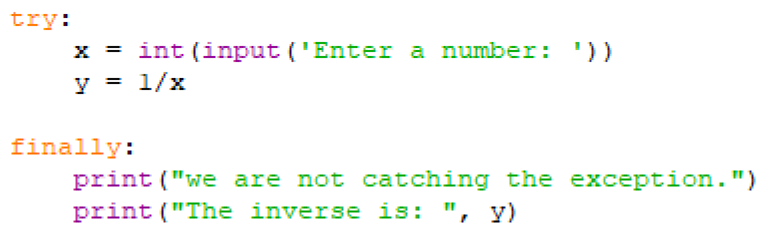
```
except(TypeError,ZeroDivisionError):
        print('Either TypeError or ZeroDivisionError occurred.')
```

The other way is not writing any exception name in the except block. This will catch any type of exception, but the programmer cannot determine specifically which exception has occurred. For example,

except:
	print('Some exception occurred.')

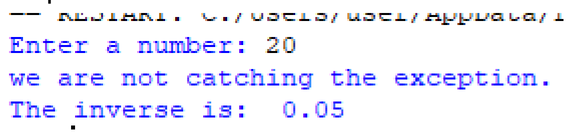
In the program below, we are finding the inverse of a given number. In this program, we are using try block without except block. When we want to use the try block alone, we need to follow it with a finally block. Since we are not using the except block, it is not possible to catch the exception.

A program to understand the usage of try with finally blocks.



```
try:
    x = int(input('Enter a number: '))
    y = 1/x

finally:
    print("we are not catching the exception.")
    print("The inverse is: ", y)
```

Output:



```
== RESTART: C:/Users/user/AppData/L
Enter a number: 20
we are not catching the exception.
The inverse is:  0.05
```

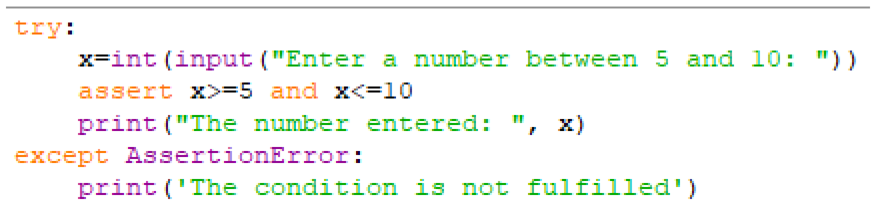
The assert Statement

The assert statement is useful to ensure that a given condition is True. If it is not true, it raises AssertionError. The syntax is as follows:

assert condition, message

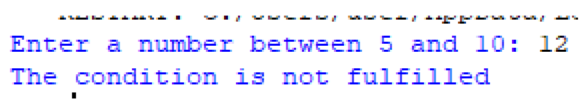
If the condition is False, then the exception by the name AssertionError is raised along with the 'message' written in the assert statement. If 'message' is not given in the assert statement, and the condition is False, then also AssertionError is raised without message. In the program below, we are using assert statements without a message. If the condition mentioned in the assert statement is False, thenAssertionError is raised.

A Python program using the assert statement and catching AssertionError.



```
try:
    x=int(input("Enter a number between 5 and 10: "))
    assert x>=5 and x<=10
    print("The number entered: ", x)
except AssertionError:
    print('The condition is not fulfilled')
```

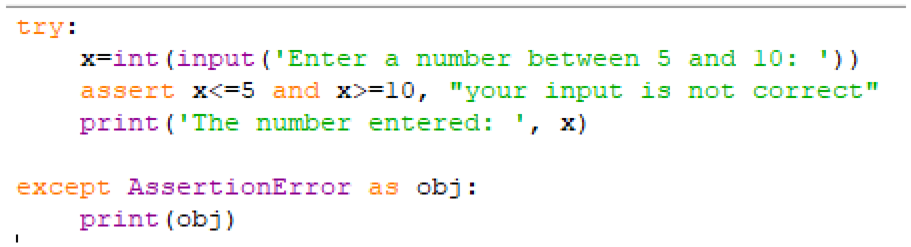
Output:



```
Enter a number between 5 and 10: 12
The condition is not fulfilled
```

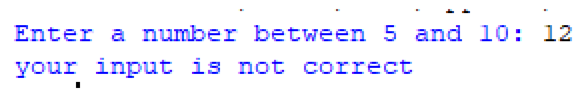
The same program can be rewritten using a message after the condition in the assen statement. When the condition is False, the message is passed to AssertionError object ‘obj' that can be displayed in the except block as shown in the program below.

A Python program to use the assert statement with a message.



```
try:
    x=int(input('Enter a number between 5 and 10: '))
    assert x<=5 and x>=10, "your input is not correct"
    print('The number entered: ', x)

except AssertionError as obj:
    print(obj)
```

Output:



```
Enter a number between 5 and 10: 12
your input is not correct
```

User-Defined Exceptions

Like the built-in exceptions of Python, the programmer can also create his own exceptions which are called 'User- defined exceptions' or 'Custom exceptions: We know Python offers many exceptions which will raise in different contexts. For example, when a number is divided by zero, the ZeroDivisionError is raised. Similarly, when the datatype is not correct, TypeError is raised.

But, there may be some situations where none of the exceptions in Python are useful for the programmer. In that case, the programmer has to create his own exception and raise it. For example, let's take a bank where customers have accounts. Each account is characterized by customer name and balance amount. The rule of the bank is that every customer should keep a minimum Rs. 2000.00 as balance amount in his account. The programmer now is given a task to check the accounts to know if every customer is maintaining a minimum balance of Rs: 2000.00 or not. If the balance amount is below s 2000.00, then the programmer wants to raise an exception saying "Balance amount is less in the account of so and so person. This will be helpful to the bank authorities to find out the customer. So, the programmer wants an exception that is raised when the balance amount in an account is less than Rs 2000.00. Since there is no such exception available in Python, the programmer has to create his own exception. For this purpose, he has to follow these steps:

1. Since all exceptions are classes, the programmer is supposed to create his own exception as a class. Also, he should make his class as a subclass to the in-built ‘Exception' class.
   class MyException (Exception):

```
        def__init__(self, arg):
            self.msg=arg
```

Here, 'MyException' class is the subclass of the ' Exception' class. This class has a constructor where a variable 'msg' is defined. This mag' receives a message passed from outside through 'arg'.

2. The programmer can write his code; maybe it represents a group of statements or a function. When the programmer suspects the possibility of exception, he should raise his own exception using 'raise' statement as:

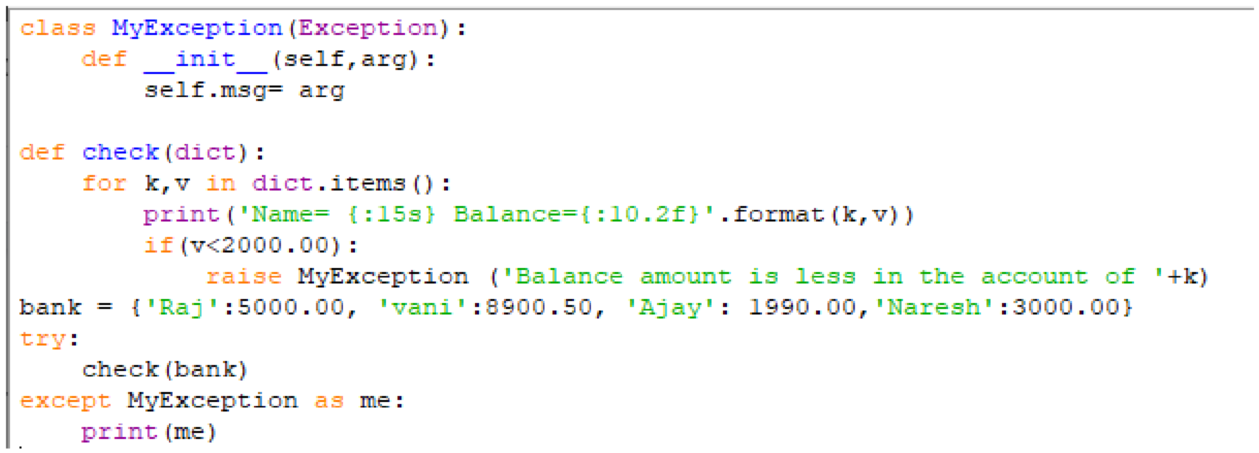
   ```
   raise MyException('message')
   ```

   Here, raise statement is raising MyException class object that contains the given 'message'.

3. The programmer can insert the code inside a 'try' block and catch the exception using 'except' block as:

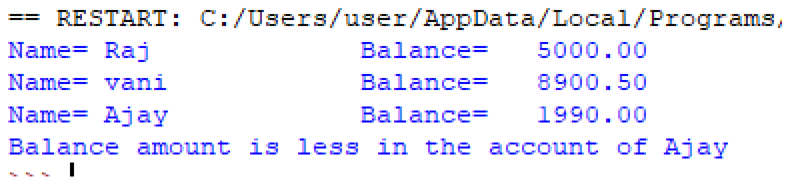
   ```
   try:
      code
   except MyException as me:
      print(me)
   ```

Here, the object 'me' contains the message given in the raise statement. All these steps are shown in the program below. In this program, we are passing a dictionary with names and balances to the check() function. As dictionary elements, name is the key and balance is the value. The check() function displays these details and checks whether the balance is less than 2000.00. If it is so, then it raises MyException with a message: Balance amount is less in the account of so and so person. This message is passed to 'except block where it is displayed.
A Python program to create our own exception and raise it when needed. create our own class as subclass to Exception class



```python
class MyException(Exception):
    def __init__(self,arg):
        self.msg= arg

def check(dict):
    for k,v in dict.items():
        print('Name= {:15s} Balance={:10.2f}'.format(k,v))
        if(v<2000.00):
            raise MyException ('Balance amount is less in the account of '+k)
bank = {'Raj':5000.00, 'vani':8900.50, 'Ajay': 1990.00,'Naresh':3000.00}
try:
    check(bank)
except MyException as me:
    print(me)
```

Output:



```
== RESTART: C:/Users/user/AppData/Local/Programs/
Name= Raj                 Balance=    5000.00
Name= vani                Balance=    8900.50
Name= Ajay                Balance=    1990.00
Balance amount is less in the account of Ajay
>>>
```

Logging the Exceptions

It is a good idea to store all the error messages raised by a program into a file. The file which stores the messages, especially of errors or exceptions is called a log' file and this technique is called ‘logging'. When we store the messages into a log file, we can open the file and read it or take a print out of the file later. This helps the programmers to understand how many errors are there, the names of those errors and where they are occurring in the program. This information will enable them to pinpoint the errors and also rectify them easily. So, logging helps in debugging the programs.
Python provides a module ‘logging' that is useful to create a log file that can store all error messages that may occur while executing a program.

There may be different levels of error messages. For example, an error that crashes the system should be given more importance than an error that merely displays a warning message. So, depending on the seriousness of the error, they are classified into 6 levels in logging' module, as shown in the table below

Level	Numeric Value	Description
CRITICAL	50	Represents a very serious error that needs high attention.
ERROR	40	Represents a serious error.

WARNING	30	Represents a warning message, some caution is needed.
INFO	20	Represents a message with some important information.
DEBUG	10	Represents a message with debugging information.
NOTSET	0	Represents that the level is not set.

As we know, by default, the error messages that occur at the time of executing a program are displayed on the user's monitor. Only the messages which are equal to or above the level of a WARNING are displayed. That means WARNINGS, ERRORS and CRITICAL ERRORS are displayed. It is possible that we can set this default behavior as we need.

To understand different levels of logging messages, we are going to write a Python program. In this program, first we have to create a file for logging (storing) the messages. This is done using basicConfig() method of logging module as:

logging.basicConfig(filename='mylog.txt', level=logging.ERROR)

Here, the log file name is given as 'mylog.txt'. The level is set to ERROR. Hence the messages whose level will be at ERROR or above, (i.e. ERROR or CRITICAL) will only be stored into the log file. Once, this is done, we can add the messages to the 'mylog.txt' file as:

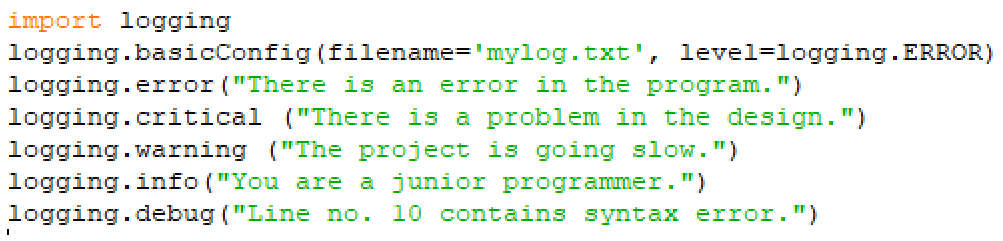
logging methodname ('message')

The methodnames can be critical(), error(), warning(), info() and debug(). For example, we want to add a critical message, we should use critical() method as:

logging.critical('System crash - Immediate attention required')

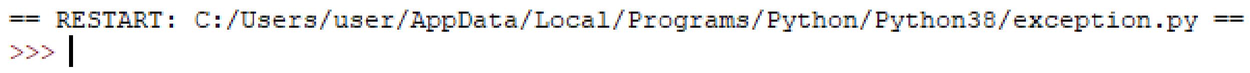
Now, this error message is stored into the log file, i.e. 'mylog.txt'. Observe the program below.

A Python program that creates a log file with errors and critical messages.



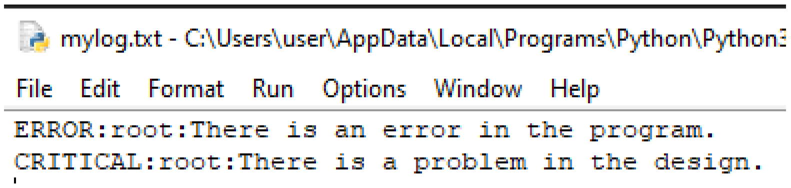
```
import logging
logging.basicConfig(filename='mylog.txt', level=logging.ERROR)
logging.error("There is an error in the program.")
logging.critical ("There is a problem in the design.")
logging.warning ("The project is going slow.")
logging.info("You are a junior programmer.")
logging.debug("Line no. 10 contains syntax error.")
```

Output:



```
== RESTART: C:/Users/user/AppData/Local/Programs/Python/Python38/exception.py ==
>>> |
```

When the above program is executed, we can see a file created by the name 'mylog.txt' in our current directory. Open the file to see the following messages:


mylog.txt - C:\Users\user\AppData\Local\Programs\Python\Python3
File Edit Format Run Options Window Help

```
ERROR:root:There is an error in the program.
CRITICAL:root:There is a problem in the design.
```

In the program above, we imported logging module as:

import logging

Since only the module is imported, we have to refer to its methods using this module name, as: logging basicConfig(), logging.error(), etc. To avoid writing the module name before the methods, we can change the import statement as:

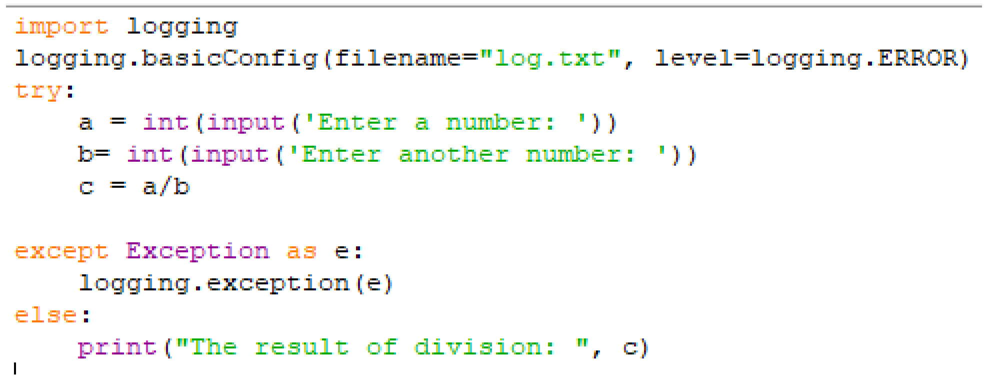
from logging import Here, we are importing all methods (means all) from the logging module. That means, instead of importing the logging module, we are importing its methods. Hence, we can refer to the methods directly, without using the module name. We can refer to them simply as: basicConfig), error(), etc.

Logging can be used to store all the exception messages which occur in a program. For this purpose, we should use the exception() method to send messages to the log file. But this exception() method should be always used inside the 'except' block only. For example, to store the exception message which is in 'e' into the log file, we can write the 'except' block as:

```
except Exception as e:
        logging.exception(e)
```

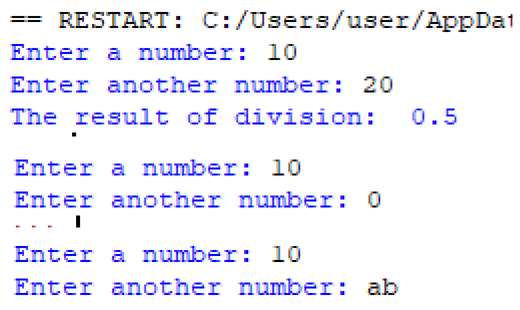
In the program below, we are accepting two numbers 'a' and 'b' from the user and then finding the result of their division. When an exception occurs inside the 'try' block, the 'except' block will catch it and the exception message will be stored into the object 'e'. This message is then written into the log file 'log.txt'.

A Python program to store the messages released by any exception into a log file



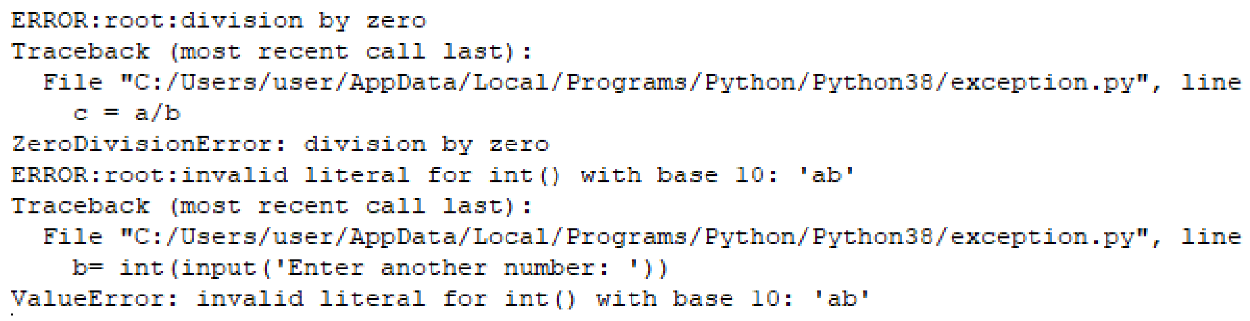
```python
import logging
logging.basicConfig(filename="log.txt", level=logging.ERROR)
try:
    a = int(input('Enter a number: '))
    b= int(input('Enter another number: '))
    c = a/b

except Exception as e:
    logging.exception(e)
else:
    print("The result of division: ", c)
```

Output:



```
== RESTART: C:/Users/user/AppDa
Enter a number: 10
Enter another number: 20
The result of division:  0.5

Enter a number: 10
Enter another number: 0

Enter a number: 10
Enter another number: ab
```

Please observe that the program is executed 3 times with different inputs. First time, the values supplied are 10 and 20. With these values, the program executed well and there are no exceptions. Second time, the values supplied are 10 and 0. In this case, there is a possibility for ZeroDivisionError. The message related to this exception will be stored into our log file. In the third time execution, the values entered are 10 and 'ab'. In this case, there is a possibility for ValueError. The message of this exception will also be added to our log file. Now, we can open the log file 'log.txt' and see the following messages:



```
ERROR:root:division by zero
Traceback (most recent call last):
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/exception.py", line
    c = a/b
ZeroDivisionError: division by zero
ERROR:root:invalid literal for int() with base 10: 'ab'
Traceback (most recent call last):
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/exception.py", line
    b= int(input('Enter another number: '))
ValueError: invalid literal for int() with base 10: 'ab'
```

.