

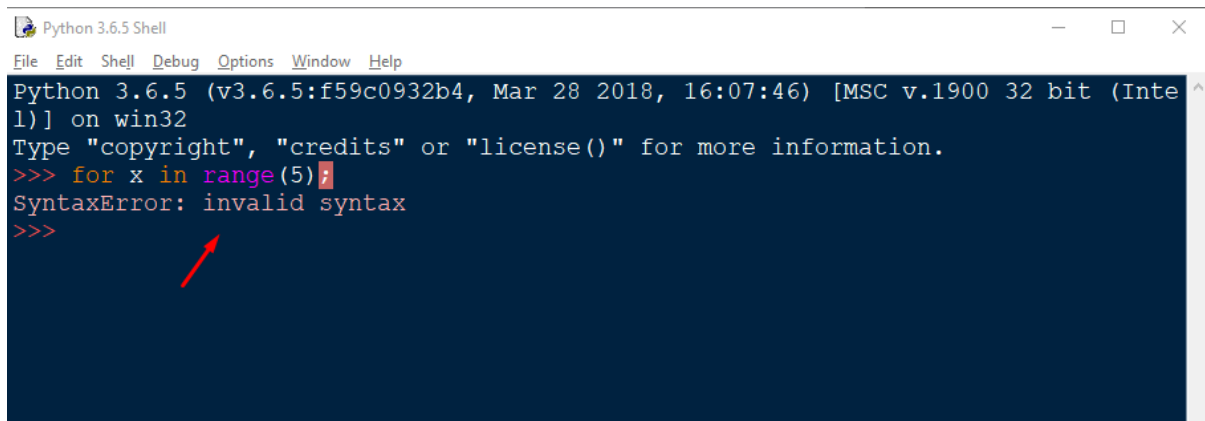
EXCEPTION HANDLING

Introduction:

So far in python while creating your programs you might have encountered a Number of errors which is not necessarily a bad thing. You may have observed it to be of two types.

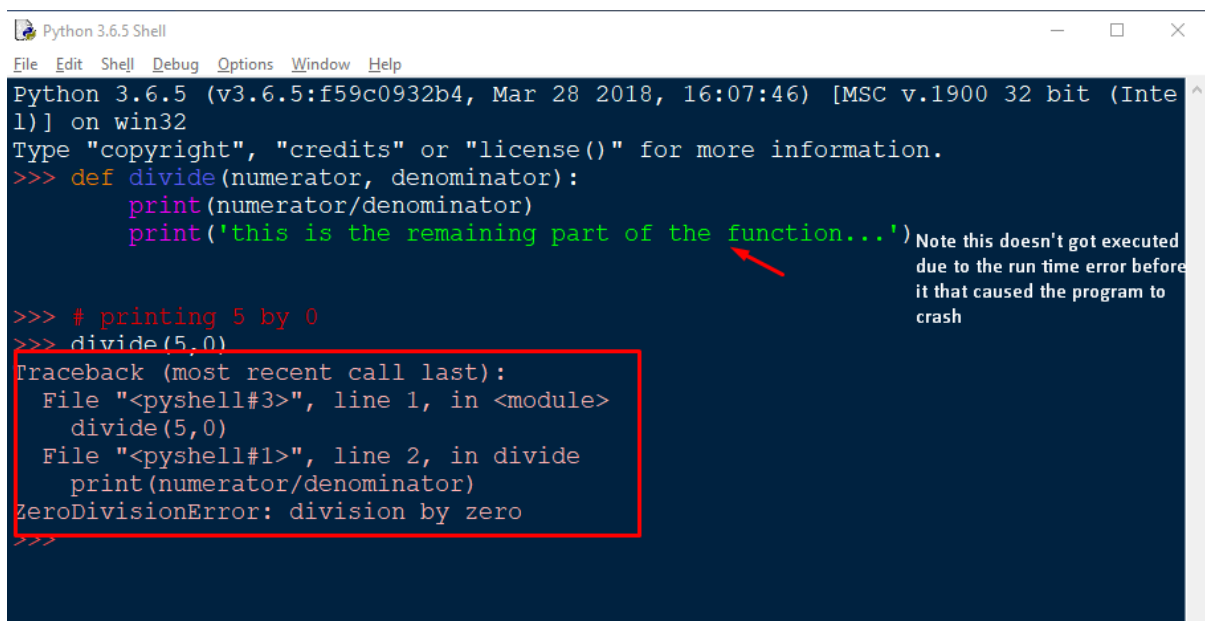
1. Errors due to incorrect syntax – Syntactical Errors
2. Errors during execution – Exceptions

While the former category of errors occurred when you were writing the code itself indicated by a error message something like in the image show below.



A screenshot of a Python 3.6.5 Shell window. The window title is "Python 3.6.5 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The command prompt shows the Python version and build information: "Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32". It then displays the prompt "Type 'copyright', 'credits' or 'license()' for more information." followed by the user input ">>> for x in range(5){" and a red arrow pointing to the closing curly brace. The error message "SyntaxError: invalid syntax" is displayed, followed by ">>>".

The latter category, Exceptions occurs during the execution of program, when you are running your program. These cause your program to crash/stop abruptly like shown below in the image



A screenshot of a Python 3.6.5 Shell window. The window title is "Python 3.6.5 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The command prompt shows the Python version and build information: "Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32". It then displays the prompt "Type 'copyright', 'credits' or 'license()' for more information." followed by the user input ">>> def divide(numerator, denominator):". The function body contains "print(numerator/denominator)" and "print('this is the remaining part of the function...')". A red arrow points to the second print statement. The user then enters ">>> # printing 5 by 0" and ">>> divide(5,0)". A red box highlights the traceback (most recent call last): File "<pyshell#3>", line 1, in <module> divide(5,0) File "<pyshell#1>", line 2, in divide print(numerator/denominator) ZeroDivisionError: division by zero. To the right of the traceback, a note says "Note this doesn't got executed due to the run time error before it that caused the program to crash".

Here comes the exception handling to the rescue. By using exception handling you can handle these exceptions (run time errors) so that your program doesn't close abruptly/crashes and the remaining part of your code gets executed.

EXCEPTION:

These are the run time errors. Python has several built-in exceptions that forces the program to output the error when something goes wrong. When such an exception occurs, it causes the current process to stop (breaks the current flow of the program) and passes the exception to the calling process to handle. If it's not handled the program will crash.

If the exception is not handled in the entire program, an error message is flashed on the screen and the program is stopped at the very point where the exception occurred preventing the further part of the program to be executed ever.

EXCEPTION HANDLING

Exceptions in python can be handled using the **Try-Except** block.

The syntax is as follows:

```
try:
    .....
    Suspicious code that can raise exception
    .....
except:
    .....
    Handling the exception if it occurred
    .....
```

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def divide(numerator, denominator):
    try:
        print(numerator/denominator)
    except ZeroDivisionError:
        print('Invalid Input: You entered 0, Try again')
    print('This is remaining part of the function...')

>>> # divide 5 by 0
>>> divide(5, 0)
Invalid Input: You entered 0, Try again
This is remaining part of the function...
>>>
```

Handling Exception

Notice

1. The exception is handled,
2. Remaining part of the function is executed
3. program has not stopped abruptly

Try:

The try block is the block where suspicious code (a code that can cause exceptions) is placed. If you have such statements that may/may not cause an exception and you want to handle it, then such code/statements are placed inside the try block.

```
try:
    statement1 inside the try block
    statement2 inside the try block
```

Except:

Except [ExceptionName]

The Except block expects the user to supply the exception name. A **Try block can have multiple except blocks** (one or more). Each except block is meant to handle the specific exception (which is mentioned by the user) for instance,

```
except Exception1:
    If there is exception 1, then execute this block to handle
except Exception2:
    If there is exception 2, then execute this block to handle
except:
    This block handles all types of exceptions
```

The Handling mechanism

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement
- If an exception occurs which does not match the exception named in the except clause, it is passed on to the following/next except clause/outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.
- if no specific type of exception is mentioned after the except, then that except clause is executed for all types of exception

ELSE block:

- It's an **optional** clause (You may/may not include it in your exception handling).
- If included, it **must follow all except clauses** in other words must be present after the last except clause
- It **is executed** if the **try block doesn't raises any exception**. If some exception is raised, the code/lines of code inside this block will never be executed.
- If you want to perform some operation if the exception is not raised But don't want to handle any accidental exception raised by such code you place it inside the else block.

```
try:
    operation_that_can_throw_ioerror()
except IOError:
    handle_the_exception_somewhat()
else:
    # we don't want to catch the IOError if it's raised
    another_operation_that_can_throw_ioerror()
finally:
    something_we_always_need_to_do()
```

If you just

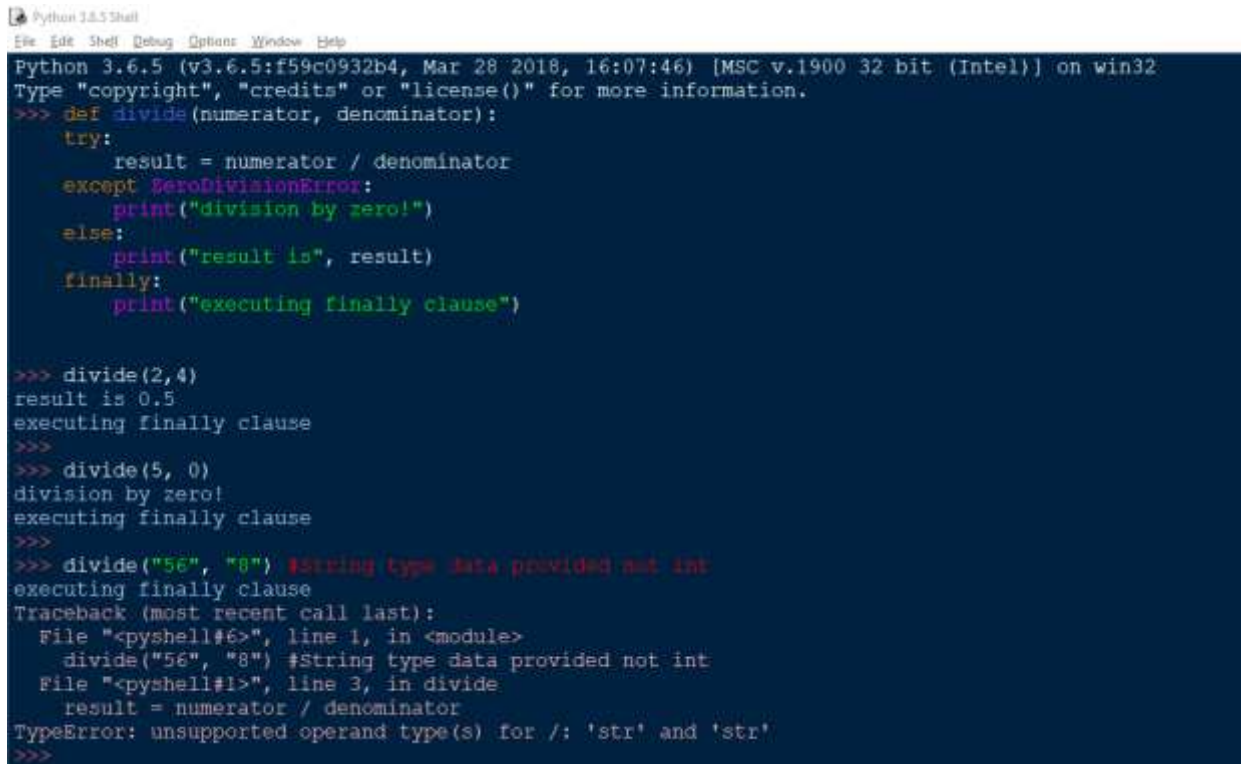
put `another_operation_that_can_throw_ioerror()` after `operation_that_can_throw_ioerror`, the except would catch the second call's errors. And if you put it after the whole try block, it'll always be run, and not until after the `finally`.

You will read about finally block after this section. For now you may note that this block will always be executed regardless of the exception is raised or not. The else lets you make sure

1. the second operation's only run if there's no exception,
2. it's run before the finally block, and
3. any IOErrors it raises (the code inside it, raises) aren't caught here

FINALLY block:

- It's an **optional** clause
- This block will always be executed regardless of the exception is raised or not before leaving the try block
- It is intended to define clean-up actions that must be executed under all circumstances.



```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def divide(numerator, denominator):
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

>>> divide(2,4)
result is 0.5
executing finally clause
>>>
>>> divide(5, 0)
division by zero!
executing finally clause
>>>
>>> divide("56", "8") #String type data provided not int
executing finally clause
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    divide("56", "8") #String type data provided not int
  File "<pyshell#1>", line 3, in divide
    result = numerator / denominator
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>>
```

Raising Exceptions:

The raise statement allows you to manually raise the specified exception to occur.

Syntax:

```
raise Exception_to_be_raised()
OR
raise Exception_to_be_raised
```

It accepts only one argument, that specifies the type of exception to raise

The argument must be either an exception instance or an exception class

If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments

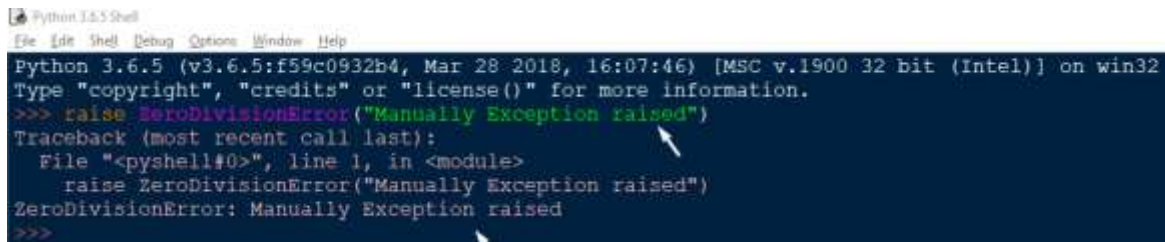
```
def MyFunction( age ):

    if age < 20:

        raise "Invalid age!", age

        # The code below to this would not be executed

        # if we raise the exception
```



```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> raise ZeroDivisionError("Manually Exception raised")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise ZeroDivisionError("Manually Exception raised")
ZeroDivisionError: Manually Exception raised
>>>
```

User- Defined Exceptions:

- Python also allows you to create your own exceptions i.e. user-defined exceptions. You can do so deriving classes from the standard built-in exceptions
- Exceptions should typically be derived from the Exception class, either directly or indirectly
- This is useful when you need to display more specific information when an exception is caught

```
class MyException(RuntimeError):

    def __init__(self, arg1, message):

        self.args = arg

        self.message = message
```

```
try:

    raise MyException("My First exception!")

except MyException as e:

    print e.args
```

Python 3.6.5 Shell

File Edit Shell Debug Options Window Help

Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

```
>>> class MyException(ZeroDivisionError):  
    def __init__(self, arg1, message):  
        self.arg1 = arg1  
        self.message = message
```

Defining Custom Exception
Derived from ZeroDivisionError

```
>>> raise MyException(10, "My Exception info message")
```

Raising User-defined exception

Traceback (most recent call last):

```
File "<pyshell#2>", line 1, in <module>  
    raise MyException(10, "My Exception info message")  
MyException: (10, 'My Exception info message')
```

```
>>>
```

```
>>> try:  
    raise MyException(5, "This is the exception info message")  
except MyException as e:  
    print (e.message, e.arg1)
```

Handling the User-defined exception

This is the exception info message 5

```
>>>
```