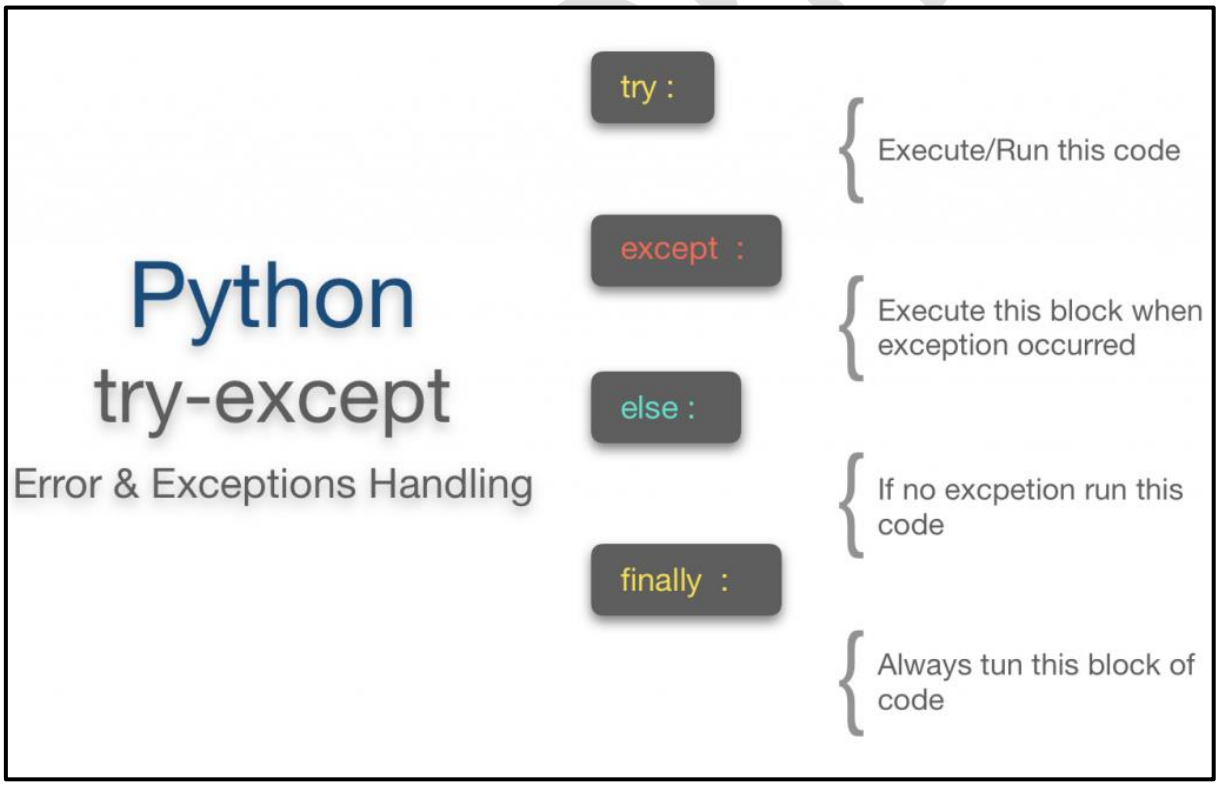
# Table of Contents

# Chapter 17

## Try Except

In Python, try and except blocks are used to handle errors and exceptions in code. The try block lets you test a block of code for errors. The except block lets you handle the error. The else block lets you execute code when there is no error. The finally block lets you execute code, regardless of the result of the try- and except blocks.

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block lets you execute code when there is no error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.




The basic Syntax for using try and except is as follows:
Syntax:

```
try:
   # code that might raise an exception
except ExceptionType:
```

## What are exceptions in Python?

Python has built-in exceptions which can output an error. If an error occurs while running the program, it's called an exception.
If an exception occurs, the type of exception is shown. Exceptions needs to be dealt with or the program will crash. To handle exceptions, the try-catch block is used.
Some exceptions you may have seen before are FileNotFoundError, ZeroDivisionError or ImportError but there are many more.
All exceptions in Python inherit from the class BaseException. If you open the Python interactive shell and type the following statement it will list all built-in exceptions:
>>> dir(builtins)

## Catching Exceptions in Python:

The try-except block can handle exceptions. This prevents abrupt exits of the program on error. In the example below we purposely raise an exception.
Ex.

```
try:
   1 / 0
except ZeroDivisionError:
   print('Divided by zero')

print('Should reach here')
# Divided by zero
# Should reach here
```

After the except block, the program continues. Without a try-except block, the last line wouldn't be reached as the program would crash.

In the above example we catch the specific exception ZeroDivisionError. You can handle any exception like this:
Ex.

```
try:
   open("fantasy.txt")
except:
   print('Something went wrong')

print('Should reach here')
```

You can write different logic for each type of exception that happens:
Ex.

```
try:
   # your code here
except FileNotFoundError:
   # handle exception
except IsADirectoryError:
   # handle exception
except:
   # all other types of exceptions

print('Should reach here')
```

Let's drive deeper:

try-except:
Lets take do a real world example of the try-except block.
The program asks for numeric user input. Instead the user types characters in the input box. The program normally would crash. But with a try-except block it can be handled properly.
The try except statement prevents the program from crashing and properly deals with it.

Ex.

```
try:
   x = input("Enter number: ")
   x = x + 1
   print(x)
except:
   print("Invalid input")
```

Entering invalid input, makes the program continue normally.

Ex.
The try-except block works for function calls too:
Ex.

```
def fail():
   1 / 0

try:
   fail()
except:
   print('Exception occured')

print('Program continues')

# Exception occured
# Program continues
```

try finally:
A try-except block can have the finally clause (optionally). The finally clause is always executed.
So the general idea is:
Syntax:

```
try:
   <do something>
except Exception:
   <handle the error>
finally:
   <cleanup>
```

Ex. If you open a file you'll want to close it, you can do so in the finally clause.

```
try:
   f = open("test.txt")
except:
   print('Could not open file')
finally:
   f.close()

print('Program continue')
```

## try else:

The else clause is executed if and only if no exception is raised. This is different from the finally clause that's always executed.

Ex.

```
try:
   x = 1
except:
   print('Failed to set x')
else:
   print('No exception occured')
finally:
   print('We always do this')

# No exception occured
# We always do this
```

You can catch many types of exceptions this way, where the else clause is executed only if no exception happens.

## Raise Exception:

Exceptions are raised when an error occurs. But in Python you can also force an exception to occur with the keyword raise.

Any type of exception can be raised:

Ex.

```
>>> raise MemoryError("Out of memory")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError: Out of memory
```

Ex.

```
>>> raise ValueError("Wrong value")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Wrong value
```

Built-in exceptions:

A list of Python's Built-in Exceptions is shown below. This list shows the Exception and why it is thrown (raised).

| Exception | Cause of Error |
|---|---|
| AssertionError | if `assert` statement fails. |
| AttributeError | if attribute assignment or reference fails. |
| EOFError | if the `input()` functions hits end-of-file condition. |
| FloatingPointError | if a floating point operation fails. |
| GeneratorExit | Raise if a generator's `close()` method is called. |
| ImportError | if the imported module is not found. |
| IndexError | if index of a sequence is out of range. |
| KeyError | if a key is not found in a dictionary. |
| KeyboardInterrupt | if the user hits interrupt key (Ctrl+c or delete). |
| MemoryError | if an operation runs out of memory. |
| NameError | if a variable is not found in local or global scope. |
| NotImplementedError | by abstract methods. |
| OSError | if system operation causes system related error. |
| OverflowError | if result of an arithmetic operation is too large to be represented. |
| ReferenceError | if a weak reference proxy is used to access a garbage collected referent. |
| RuntimeError | if an error does not fall under any other category. |
| StopIteration | by `next()` function to indicate that there is no further item to be returned by iterator. |
| SyntaxError | by parser if syntax error is encountered. |
| IndentationError | if there is incorrect indentation. |
| TabError | if indentation consists of inconsistent tabs and spaces. |
| SystemError | if interpreter detects internal error. |
| SystemExit | by `sys.exit()` function. |

| | |
|---|---|
| TypeError | if a function or operation is applied to an object of incorrect type. |
| UnboundLocalError | if a reference is made to a local variable in a function or method, but no value has been bound to that variable. |
| UnicodeError | if a Unicode-related encoding or decoding error occurs. |
| UnicodeEncodeError | if a Unicode-related error occurs during encoding. |
| UnicodeDecodeError | if a Unicode-related error occurs during decoding. |
| UnicodeTranslateError | if a Unicode-related error occurs during translating. |
| ValueError | if a function gets argument of correct type but improper value. |
| ZeroDivisionError | if second operand of division or modulo operation is zero. |

## User-defined Exceptions:

Python has many standard types of exceptions, but they may not always serve your purpose.
Your program can have your own type of exceptions.
To create a user-defined exception, you have to create a class that inherits from Exception.

Ex.

```
class LunchError(Exception):
   pass

raise LunchError("Programmer went to lunch")
```

Output:

```
Traceback (most recent call last):
  File “userDefTE.py”, line 5, in
   raise LunchError(“Programmer went to lunch”)
main.LunchError: Programmer went to lunch
```

# You made a user-defined exception named LunchError in the above code. You can raise this new exception if an error occurs.

Your program can have many user-defined exceptions. The program below throws exceptions based on a new projects money:

Ex.

```
class NoMoneyException(Exception):
   pass

class OutOfBudget(Exception):
   pass

balance = int(input("Enter a balance: "))
if balance < 1000:
   raise NoMoneyException
elif balance > 10000:
   raise OutOfBudget
```

Output:

```
Enter a balance: 500
  Traceback (most recent call last):
    File “userDefTE.py”, line 10, in
      raise NoMoneyException
  main.NoMoneyException

Enter a balance: 100000
  Traceback (most recent call last):
    File “userDefTEs.py”, line 12, in
      raise OutOfBudget
  main.OutOfBudget
```

## BuiltIn Exception:

| Sr.No. | Exception Name | Description |
|---|---|---|
| 1 | Exception | Base class for all exceptions |
| 2 | StopIteration | Raised when the next() method of an iterator does not point to any object. |
| 3 | SystemExit | Raised by the sys.exit() function. |
| 4 | StandardError | Base class for all built-in exceptions except StopIteration and SystemExit. |
| 5 | ArithmeticError | Base class for all errors that occur for numeric calculation. |
| 6 | OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| 7 | FloatingPointError | Raised when a floating point calculation fails. |
| 8 | ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |
| 9 | AssertionError | Raised in case of failure of the Assert statement. |
| 10 | AttributeError | Raised in case of failure of attribute reference or assignment. |
| 11 | EOFError | Raised when there is no input from either the raw_input() raw_input() or input() input() function function and the end of file is reached. |
| 12 | ImportError | Raised when an import statement fails. |
| 13 | KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| 14 | LookupError | Base class for all lookup errors. |
| 15 | IndexError | Raised when an index is not found in a sequence. |
| 16 | KeyError | Raised when the specified key is not found in the dictionary. |
| 17 | NameError | Raised when an identifier is not found in the local or global namespace. |
| 18 | UnboundLocalError | Raised when trying to access a local variable ariable in a function function or method but no value has been assigned to it. |
| 19 | EnvironmentError | Base class for all exceptions that occur outside the Python environment. |
| 20 | IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| 21 | IOError | Raised for operating system-related errors. |
| 22 | SyntaxError | Raised when there is an error in Python syntax. |
| 23 | IndentationError | Raised when indentation is not specified properly. |
| 24 | SystemError | Raised when the interpreter interpreter finds an internal internal problem, problem, but when this error is encountered encountered the Python interpreter does not exit. |
| 25 | SystemExit | Raised when Python interpreter interpreter is quit by using the sys.exit() sys.exit() function. function. If not handled handled in the code, causes the interpreter to exit. |
| 26 | TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| 27 | ValueError | Raised when the built-in built-in function function for a data type has the valid type of arguments, arguments, but the arguments have invalid values specified. |
| 28 | RuntimeError | Raised when a generated error does not fall into any category. |
| 29 | NotImplementedError | Raised when an abstract abstract method that needs to be implemented implemented in an inherited inherited class is not actually implemented. |