Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
   You do your operations here
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
```

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The **finally:** block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Note − You can provide except clause(s), or a finally clause, but not both. You cannot use *else* clause as well along with a finally clause.

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

```
try:
   You do your operations here
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception −

Live Demo

```
#!/usr/bin/python3

# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError as Argument:
      print ("The argument does not contain numbers\n", Argument)
```

```
# Call above function here.

temp_convert("xyz")
```

Exception Errors

Some of the common exception errors are:

IOError

If the file cannot be opened.

ImportError

If python cannot find the module

ValueError

Raised when a built-in operation or function receives an argument that has the

right type but an inappropriate value

KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C or Delete)

EOFError

Raised when one of the built-in functions (input() or raw_input()) hits an

end-of-file condition (EOF) without reading any data

Example

Let's have a look at some examples using exceptions.

```
except IOError:
    print('An error occured trying to read the file.')

except ValueError:
    print('Non-numeric data found in the file.')

except ImportError:
    print "NO module found"

except EOFError:
    print('Why did you do an EOF on me?')

except KeyboardInterrupt:
    print('You cancelled the operation.')

except:
    print('An error occured.')
```

MULTIPLE EXCEPTIONS:

```
try:
   # do something
   pass
```

```
except ValueError:
    # handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass

except:
    # handle all other exceptions
    pass
```

Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword raise.

We can also optionally pass in value to the exception to clarify why that exception was raised.

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
```

```
MemoryError: This is an argument


>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
...
Enter a positive integer: -2
That is not a positive number!
```

User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass


class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass


class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
```

```
    pass

# our main program
# user guesses a number until he/she gets it right

# you need to guess this number
number = 10

while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly.")
```

Working of open() function

We use **open ()** function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: **open(filename, mode)**. There are three kinds of mode, that Python provides and how files can be opened:

- “ **r** “, for reading.
- “ **w** “, for writing.
- “ **a** “, for appending.
- “ **r+** “, for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “ **r** ” by default.

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

```
# a file named "python", will be opened with the reading mode.
file = open(‘pyt.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)
```

reading:

```
f = open("demofile.txt", "r")
print(f.read())
```

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Writing:

o write to an existing file, you must add a parameter to the `open()` function:

`"a"` - Append - will append to the end of the file

`"w"` - Write - will overwrite any existing content

Example

Open the file "demofile.txt" and append content to the file:

```
f = open("demofile.txt", "a")
f.write("Now the file has one more line!")
f.close()
```

Deleting file:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

With:

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

JSON:

The built-in json package has the magic code that transforms your

```
import json
with open('data.json', 'w') as outfile:
    json.dump(data, outfile)
```

To get ***utf8****-encoded* file as opposed to ***ascii****-encoded*

```
import json
with open('data.txt', 'w') as f:
  json.dump(data, f, ensure_ascii=False)
```

Read:

```
import json


with open('path_to_file/person.json') as f:
  data = json.load(f)


# Output: {'name': 'Bob', 'languages': ['English', 'Fench']}
print(data)
```