Python Unit 4

Exception

In Python, all exceptions must be instances of a class that derives from BaseException.

In a try statement with an except clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which it is derived)

Python Try exception

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.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
These exceptions can be handled using the `try` statement:

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.
Without the try block, the program will crash and raise an error:

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources. For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

```python
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

The `raise` keyword is used to raise an exception.
You can define what kind of error to raise, and the text to print to the user.

Raise a TypeError if x is not an integer:

```
 x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

This can be useful to close objects and clean up resources:

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

```
try:
        a = 10/0
        print (a)
except ArithmeticError:
                print ("This statement is raising an arithmetic exception.")
else:
        print ("Success.")
```

Output : This statement is raising an arithmetic exception.

```
try:
        a = [1, 2, 3]
        print (a[3])
except LookupError:
        print ("Index out of bound error.")
else:
        print ("Success")
```

Output : Index out of bound error.

```
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Python built in Exception

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indendation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete

LookupError	Raised when errors raised cant be found
MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the method
OSError	Raised when a system related operation causes an error
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific expections
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces

SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined
UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a unicode problem occurs
UnicodeEncodeError	Raised when a unicode encoding problem occurs
UnicodeDecodeError	Raised when a unicode decoding problem occurs
UnicodeTranslateError	Raised when a unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

File Wildcards

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files.

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

File Access Modes:

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
2. **Read and Write ('r+') :** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
3. **Write Only ('w') :** Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
4. **Write and Read ('w+')** : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only ('a')** : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File

It is done using the open() function. No module is required to be imported for this function.

```
File_object = open(r"File_Name","Access_Mode")
```

Example :
```
file1 = open("MyFile.txt","a")
file2 = open("D:\Text\MyFile2.txt","w+")
```

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

File_object.close()

```
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file

write() : Inserts the string str1 in a single line in the text file.`File_object.write(str1)`
writelines() : For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

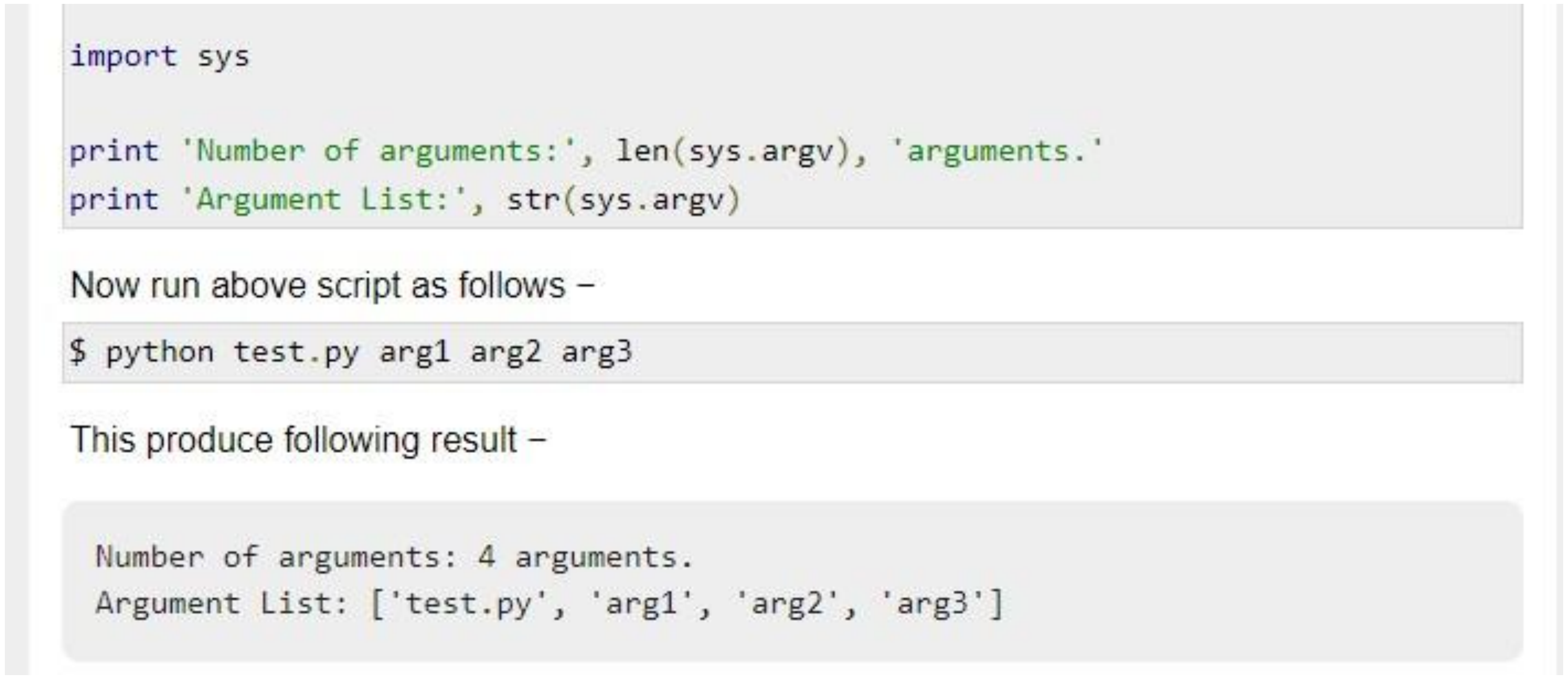
Command Line arguments

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes −

- sys.argv is the list of command-line arguments.
- len(sys.argv) is the number of command-line arguments.

  Here sys.argv[0] is the program ie. script name.

Example



```
import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'
print 'Argument List:', str(sys.argv)
```

Now run above script as follows −

```
$ python test.py arg1 arg2 arg3
```

This produce following result −

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

Creating Virtual Environments

The module used to create and manage virtual environments is called `venv`. `venv` will usually install the most recent version of Python that you have available. If you have multiple versions of Python on your system, you can select a specific Python version by running `python3` or whichever version you want.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python3 -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn’t exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you’ve created a virtual environment, you may activate it.

On Windows, run:

```
tutorial-env\Scripts\activate.bat
```

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

Managing Packages with pip

You can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the Python Package Index, <https://pypi.org>. You can browse the Python Package Index by going to it in your web browser.
`pip` has a number of subcommands: "install", "uninstall", "freeze", etc. (Consult the Installing Python Modules guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
```

You can also install a specific version of a package by giving the package name followed by == and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
```

`pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`pip show` will display information about a particular package:

`pip list` will display all of the packages installed in the virtual environment:

Verifying the Connector Installation,

https://dev.mysql.com/doc/connector-python/en/connector-python-verification.html