PES UNIVERSITY

(*Established under Karnataka Act No. 16 of 2023*)
100 ft. Ring Road, Bengaluru – 560085, Karnataka, India

Programming with Python

Unit III – Error Handling, File I/O, Modules, and Advanced Python Concepts

Prepared by,

Niteesh K R,
Assistant Professor,
Department of Computer Applications,
PES University,
Bengaluru

Contents

1 Error Handling **3**

1.1 Introduction to Errors 4

1.1.1 Types of Errors 5

1.2 Runtime Errors 8

1.2.1 NameError 9

1.2.2 TypeError 10

1.2.3 IndexError 12

1.2.4 AttributeError 13

1.2.5 ArithmeticError 14

1.2.6 KeyError 16

1.2.7 ValueError 17

1.2.8 KeyboardInterrupt 18

1.2.9 ImportError and ModuleNotFoundError 18

1.3 Error Handling in Python 19

1.3.1 Raising Errors 20

1.3.2 Handling Errors 20

1.3.3 try-except blocks 20

1.3.4 try-except-else blocks 22

1.3.5 try-except-finally 24

1.3.6 Points to Remember 25

1.4 Custom Errors 26

2 File Handling and I/O **27**

2.1 Functions for File Handling 28

2.2 File I/O – Text Files 29

2.2.1 Reading contents of a text file 29

2.2.2 Writing contents into text file 30

2.2.3 Copying contents of one file to another 32

2.3 File I/O – CSV Files 34
2.3.1 Reading CSV Files 35
2.3.2 Writing CSV Files 38
2.3.3 CSV File Handling using pandas 39
2.4 File I/O – JSON Files 41
2.4.1 Functions for JSON handling 42
2.5 File I/O – XML Files 44
2.5.1 Element Tree 47
2.5.2 DOM 49
2.5.3 BeautifulSoup 50
2.6 Object Serialisation – pickle 52

3 Modules and Packages **55**
3.1 Modules and Packages 56
3.1.1 Importing – At same level 57
3.1.2 Absolute Import 58
3.1.3 Relative Import 59
3.2 Standard Library 60
3.2.1 os 60
3.2.2 sys 61
3.2.3 datetime 61
3.2.4 collections 62

Chapter 1

Error Handling

1.1 Introduction to Errors

- **Error** is a problem caused because of incorrect syntax or logic
- Some errors occur while running the program and they don't allow further execution
- Some errors occur even before the program is executed
- Errors in Python are often fatal
  - Fatal errors cause the program to crash
  - As soon as an error is encountered in a Python program, it will not be executed further
- Since errors are fatal in Python, they need to be handled in a way that will not affect the flow of the program
- Handling errors *gracefully* so that they do not crash the programs is called as **Error Handling**
- Errors are also called as Exceptions
- Every error will show an error message upon interpreting the program. Error message consists of:
  - The cell and the line number at which the error occurred
  - The type of error (SyntaxError, NameError, ValueError etc.)
  - Error message describing why the error occurred
  - Highlighting the part of the line causing the error
- All errors in Python are objects of different error classes and are **raised**
- Class hierarchy of errors is illustrated in Figure 1.1

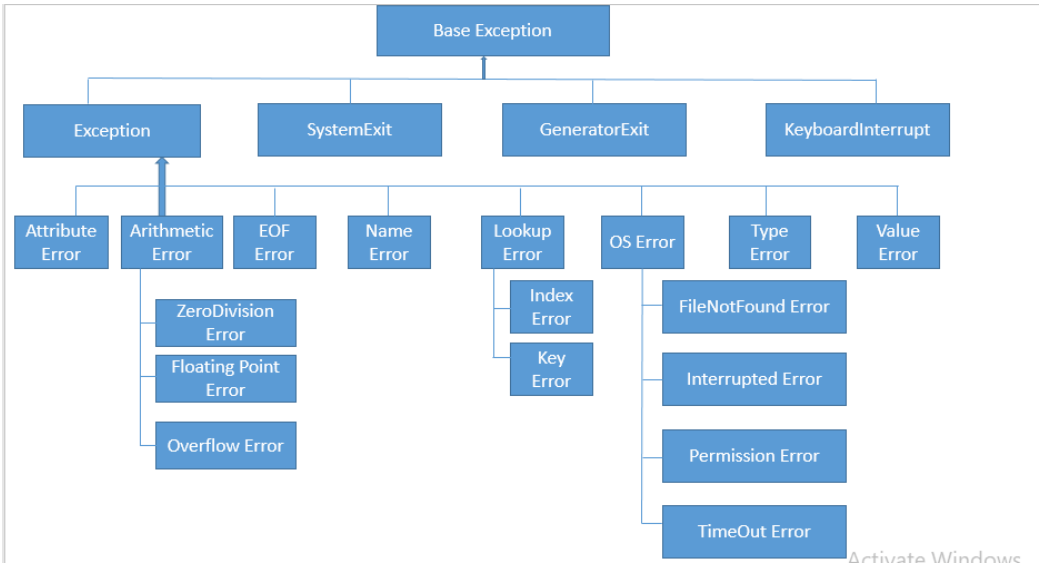

Figure 1.1: Error Classes in Python (Source: chercher.tech)

1.1.1 Types of Errors

- Errors are of 3 types
  - Syntax Errors
  - Runtime Errors
  - Logical Errors

Syntax Errors

- Syntax Errors are caused if the syntax rules of Python are not followed
- Syntax Errors do not allow the program to execute
- Caused while interpreting the program
- Caused if the interpreter is not able to parse the program written
- Reasons
  - Incorrect use of operators
  - Additional or missing indentation

  - Incorrect keyword usage

- Syntax Errors do not allow the programs to run and the interpreter will display a message stating the same
- Any syntax error will raise a **SyntaxError** in Python
- Example: Incorrect keyword usage (Def used instead of *def*), as shown in Figure 1.2

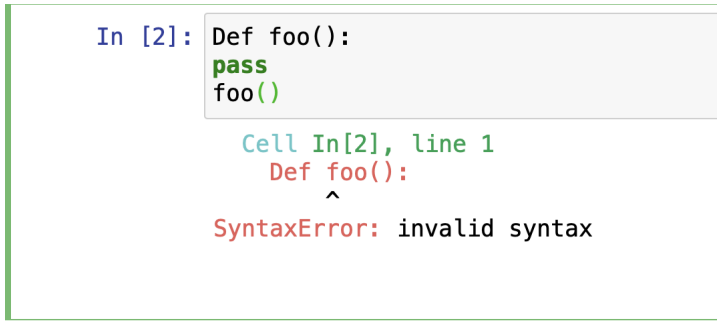

Figure 1.2: Example: SyntaxError

- Note that the IndentationError in Line 2 is skipped as the interpreter raised an error at Line 1
- Example: Missing colon (:) Figure 1.3

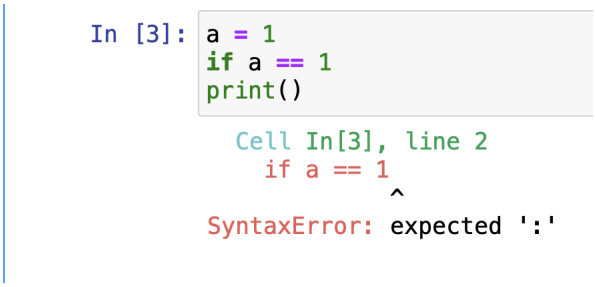

Figure 1.3: Example: SyntaxError due to missing :

Runtime Errors

- Runtime errors are caused after the program interpretation
- They are raised during runtime due to various reasons
  - Usage of an unassigned object
  - Performing any operation with invalid types
  - Mismatched number of arguments in a function call
  - Passing an argument of invalid type to a function
  - Dividing a number by 0
- It is important to handle runtime errors
- Most runtime errors are fatal
- Example: Trying to convert an alphabet into an integer using int(), illustrated in Figure 1.4

```
a = int('a')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 a = int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Figure 1.4: Runtime Error: Converting alphabet to int

Logical Errors

- Logical Errors are caused due to incorrect logic
- Logical Errors are not raised by Python
- Programs with logical errors are executed without any problems but, they produce erroneous/incorrect outputs
- To find logical errors, the programs are to be traced manually

- Example: Find the sum of the first 10 naturals. Here, the range must have been *(1, n + 1)*, not *(1, n)*, illustrated in Figure 1.5

```
In [6]: s = 0
        for i in range(1, 10):
            s += i
        print(s)

45
```

Figure 1.5: Logical Error Example

1.2 Runtime Errors

Some errors covered

- NameError
- TypeError
- IndexError
- AttributeError
- ArithmeticError
  - ZeroDivisionError
  - FloatingPointError
  - OverflowError
- KeyError
- ValueError

- KeyboardInterrupt
- Built-in Exceptions can be viewed using
  print(dir(locals()['__builtins__'])) as shown in Figure 1.6

```
print(dir(locals()['__builtins__']))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedEr
ror', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarn
ing', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'False',
'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorEx
it', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'Interrup
tedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryErro
r', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'No
tImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionErr
or', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeE
rror', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarni
ng', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'Unbound
LocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateE
rror', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IP
YTHON__', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name_
_', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'boo
l', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'compl
ex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate', 'e
val', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getatt
r', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issub
class', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'nex
t', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reverse
d', 'round', 'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

Figure 1.6: Built-in Errors

1.2.1 NameError

- Occurs when
  - A variable or a function is used before assigning or defining
  - Using a variable or a function outside of scope
- Example: NameError (Function used before defining), illustrated in Figure 1.7
- Example: NameError (Variable used before assigning), illustrated in Figure 1.8

```
abc()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[61], line 1
----> 1 abc()

NameError: name 'abc' is not defined
```

Figure 1.7: Function used before defining

```
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 a

NameError: name 'a' is not defined
```

Figure 1.8: Variable used before assigning

- Example: NameError (Accessing a function outside the scope), illustrated in Figure 1.9. Here, the random package is not imported.

```
seed(1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 seed(1)

NameError: name 'seed' is not defined
```

Figure 1.9: Accessing a function outside the scope

1.2.2 TypeError

- Occurs when
  - An operation is performed on invalid operand types
  - Performing an arithmetic or logical operation on incompatible types (Adding int to str, Concatenating str with int)

  - Calling a function with invalid type of arguments
  - Calling a function with less or more number of required parameters than defined

- Example: TypeError (Adding int with str), illustrated in Figure 1.10

```
1 + 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 1 + 'a'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figure 1.10: Adding int with str

- Example: TypeError (Concatenating str with int), illustrated in Figure 1.11

```
'a' + 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 'a' + 1

TypeError: can only concatenate str (not "int") to str
```

Figure 1.11: Concatenating str with int

Note that Figure 1.10 is trying to perform addition while in Figure 1.11, it is concatenation. This is because of the first operand.

- Example: TypeError (Calling a function with incorrect number of parameters), illustrated in Figure 1.12

```
def foo(a, b):
    return a + b
foo(10)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 3
      1 def foo(a, b):
      2     return a + b
----> 3 foo(10)

TypeError: foo() missing 1 required positional argument: 'b'
```

Figure 1.12: Calling a function with incorrect number of parameters

- Example: TypeError (Passing arguments of wrong type to a function), illustrated in Figure 1.13

```
def foo(a, b):
    return a + b
foo(10, 'a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 3
      1 def foo(a, b):
      2     return a + b
----> 3 foo(10, 'a')

Cell In[25], line 2, in foo(a, b)
      1 def foo(a, b):
----> 2     return a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figure 1.13: Passing arguments of wrong type to a function

1.2.3 IndexError

- Occurs when
  - Trying to access a positive index greater than or equal to the length of the iterable

Example, trying to access L[5] in a list of 5 items. Valid index values are 0, 1, 2, 3, and 4.

  - Trying to access an element that does not exist in an iterable
  - Trying to access a negative index that does not exist

- Raised only by indexable iterables (Lists, Tuples, Strings)
- Example: IndexError (Trying to access a positive index greater than or equal to the length of the iterable), illustrated in Figure 1.14

```
l = [1, 2, 3, 4, 5]
l[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[30], line 2
      1 l = [1, 2, 3, 4, 5]
----> 2 l[5]

IndexError: list index out of range
```

Figure 1.14: Trying to access a positive index greater than or equal to the length of the iterable

- Example: IndexError (String index out of range), illustrated in Figure 1.15

```
l = 'abcde'
l[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[31], line 2
      1 l = 'abcde'
----> 2 l[5]

IndexError: string index out of range
```

Figure 1.15: String index out of range

- Example: IndexError (Accessing a negative index that does not exist), illustrated in Figure 1.16

1.2.4 AttributeError

- Occurs when

```
l = [1, 2, 3]
l[-4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[34], line 2
      1 l = [1, 2, 3]
----> 2 l[-4]

IndexError: list index out of range
```

Figure 1.16: Accessing a negative index that does not exist

  - Trying to access an attribute (Data member or member function) that does not exist in a class
  - Trying to access a private attribute using the object

- Example: AttributeError (Accessing a Data Member that does not exist in the class), illustrated in Figure 1.17

```
class A:
    pass
a = A()
a.attr
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[35], line 4
      2     pass
      3 a = A()
----> 4 a.attr

AttributeError: 'A' object has no attribute 'attr'
```

Figure 1.17: Accessing a Data Member that does not exist in the class

- Example: AttributeError (Accessing a private data member), illustrated in Figure 1.18

1.2.5 ArithmeticError

Three sub-classes

- ZeroDivisionError – Raised when trying to divide any number by 0

  Example: ZeroDivisionError, illustrated in Figure 1.19

```
class A:
    def __init__(self, a): self.__a = a
a = A(100)
a.__a
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[36], line 4
      2     def __init__(self, a): self.__a = a
      3 a = A(100)
----> 4 a.__a

AttributeError: 'A' object has no attribute '__a'
```

Figure 1.18: Accessing a private data member

```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 1
----> 1 1/0

ZeroDivisionError: division by zero
```

Figure 1.19: ZeroDivisionError

- OverflowError – Raised when the result of an arithmetic operation is too large to be represented

  Example: OverflowError, illustrated in Figure 1.20

```
def pi():
    pi = 0
    for k in range(350):
        pi += (4./(8.*k+1.) - 2./(8.*k+4.) - 1./(8.*k+5.) - 1./(8.*k+6.)) / 16.**k
    return pi
print(pi())
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
Cell In[2], line 6
      4         pi += (4./(8.*k+1.) - 2./(8.*k+4.) - 1./(8.*k+5.) - 1./(8.*k+6.)) / 16.**k
      5     return pi
----> 6 print(pi())

Cell In[2], line 4, in pi()
      2 pi = 0
      3 for k in range(350):
----> 4     pi += (4./(8.*k+1.) - 2./(8.*k+4.) - 1./(8.*k+5.) - 1./(8.*k+6.)) / 16.**k
      5 return pi

OverflowError: (34, 'Result too large')
```

Figure 1.20: OverflowError

- FloatingPointError – Occurs while representing a real number. Not currently used in the recent version of Python

1.2.6 KeyError

- May occur with dictionaries and dictionary-like objects
- Occurs when
  - Trying to access a key that does not exist in a dictionary
  - Only when trying to access the value of a dictionary key using dictionary[key], not with the dict.get(key) function
  - dict.get(key) internally handles KeyError
- Example: KeyError, illustrated in Figure 1.21

```
d = {1:1}
d[2]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[6], line 2
      1 d = {1:1}
----> 2 d[2]

KeyError: 2
```

Figure 1.21: KeyError

1.2.7 ValueError

- Occurs when
  - An operation or function receives an argument that has the right type but an inappropriate value
  - Example: ValueError (Converting an alphabetic string to int), illustrated in Figure 1.22

```
int('a')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Figure 1.22: ValueError

  - Example: ValueError (Converting a user-input string to int), illustrated in Figure 1.23

```
n = int(input())
```

```
a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[8], line 1
----> 1 n = int(input())

ValueError: invalid literal for int() with base 10: 'a'
```

Figure 1.23: ValueError

1.2.8 KeyboardInterrupt

- Occurs when the user hits the interrupt key (normally Control-C or Delete)
- Example: KeyboardInterrupt, illustrated in Figure 1.24

```
n = int(input())
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[9], line 1
----> 1 n = int(input())
```

Figure 1.24: KeyboardInterrupt

1.2.9 ImportError and ModuleNotFoundError

- **ImportError** occurs when the import statement has trouble trying to load a module. Also raised when the "from list" in *from ... import* has a name that cannot be found
- Example: ImportError (Trying to import a module that is not available in a package), illustrated in Figure 1.25

```
from xml import a
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[13], line 1
----> 1 from xml import a

ImportError: cannot import name 'a' from 'xml' (/Users/niteesh/miniconda3/lib/python3.10/xml/__init__.py)
```

Figure 1.25: ImportError

- **ModuleNotFoundError** occurs when a module cannot be located. It is a subclass of ImportError
- Example: ModuleNotFoundError (Trying to import a module that is not available), illustrated in Figure 1.26

```
from a import b
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[10], line 1
----> 1 from a import b

ModuleNotFoundError: No module named 'a'
```

Figure 1.26: ModuleNotFoundError

1.3 Error Handling in Python

- Errors are to be handled gracefully
- Errors must not crash programs
- The mechanism to handle errors such that the programs are not crashed is **Exception/Error Handling**
- In Python, error handling is done using *try-except* blocks
  - Optional *else* and *finally* blocks can be used as well
- When any error occurs, Python *raises* an error
  - Raising an error is similar to instantiating an object of one of the error classes
- Two major components:

  - Raising errors
  - Handling errors

- Major keywords:
  - *try*
  - *except*
  - *finally*
  - *raise*

1.3.1 Raising Errors

- Raising an error is notifying that an error has occurred
- *raise* keyword is used to raise/cause/trigger an error explicitly
- Errors are raised implicitly as well
- Once an error is raised, it needs to be handled in order to not halt the execution of the program abruptly

1.3.2 Handling Errors

- Once an error is raised, the program looks for a way of handling that error
- Handling an error is executing a set of instructions when an error is raised
- *except* blocks will be searched for when any error is raised
- *except* blocks contain statements that are to be executed when any error occurs

1.3.3 try-except blocks

- *try* block holds the code that is suspected to be raising errors
- *except* block holds the code that is to be executed when an error is raised
- An except block can be made to handle

- Generic Errors
  - Generic Error handling blocks can be used to handle any type of errors
  - Example: Handling a ValueError using generic except block, illustrated in Figure 1.27

```
try:
    n = int(input())
    print(n)
except:
    print('Some Error Happened')
```

```
a
Some Error Happened
```

Figure 1.27: Generic except block

  - Generic except block can handle any error because, it treats any raised error as an object of class *Exception*
  - Since Exception is the base class for all errors in Python, any error can be treated as an object of Exception
  - Example: Generic except block made to handle any Exception, illustrated in Figure 1.28. The *except* block is made to handle any error as an object of Exception and using *as* keyword, the error is further referred in the *except* block as *e*. To view the specific error class, *type(e)* is used. To view the actual error message, *print(e)* is used.

```
try:
    n = int(input())
    print(n)
except Exception as e:
    print(type(e))
    print('Some Error Happened')
    print(e)
```

```
a
<class 'ValueError'>
Some Error Happened
invalid literal for int() with base 10: 'a'
```

Figure 1.28: Generic except block for Exception

- Specific Errors
  - An except block can also be made to handle a specific error, instead of a generic error, Exception
  - A specific error block can handle only one type of error
  - Example: Handling ValueError using *except ValueError*, illustrated in Figure 1.29

```
try:
    n = int(input())
    print(n)
except ValueError as e:
    print(type(e))
    print(e)
except:
    print('Some Error Happened')
```

```
a
<class 'ValueError'>
invalid literal for int() with base 10: 'a'
```

Figure 1.29: Specific except block – ValueError

  - It is always a good practice to have multiple except blocks
  - Have multiple except blocks for different types of errors that may occur and have one generic except block at the end to handle errors that are not handled by other except blocks
  - Since Python uses the nearest except block in case of errors, it is not ideal to have a generic except block as the first block

1.3.4 try-except-else blocks

- Python error handling can have an optional *else* block
- The suite under *else* block will be executed only when the *try* block executes without any errors
- A *try* block can have a maximum of one *else* block

- Example is illustrated in Figure 1.30

```
try:
    n = int(input())
    print(n)
except ValueError as e:
    print(type(e))
    print(e)
except:
    print('Some Error Happened')
else:
    print('No Error Happened')
```

```
5
5
No Error Happened
```

Figure 1.30: try-except-else (No error)

- The *else* block will be ignored if the *try* block raises an error as illustrated in Figure 1.31

```
try:
    n = int(input())
    print(n)
except ValueError as e:
    print(type(e))
    print(e)
except:
    print('Some Error Happened')
else:
    print('No Error Happened')
```

```
a
<class 'ValueError'>
invalid literal for int() with base 10: 'a'
```

Figure 1.31: try-except-else (With error)

1.3.5 try-except-finally

- Python error handling also offers an optional *finally* block
- The suite under *finally* will be executed irrespective of the *try* block raising errors
- If the *try* block raises an error, the corresponding *except* block will be executed and then, *finally* block will be executed
- If the *try* block executes without raising errors, no *except* blocks will be executed. Instead, the *finally* block gets executed
- *finally* block can be used for cleanup tasks
- One *try* block can have a maximum of one *finally* block
- Examples are illustrated in Figures 1.32 and 1.33

```
try:
    n = int(input())
    print(n)
except ValueError as e:
    print(type(e))
    print(e)
except:
    print('Some Error Happened')
finally:
    print('finally block executed')
```

```
5
5
finally block executed
```

Figure 1.32: try-except-finally (No error)

```
try:
    n = int(input())
    print(n)
except ValueError as e:
    print(type(e))
    print(e)
except:
    print('Some Error Happened')
finally:
    print('finally block executed')
```

```
a
<class 'ValueError'>
invalid literal for int() with base 10: 'a'
finally block executed
```

Figure 1.33: try-except-finally (With error)

1.3.6 Points to Remember

- A *try* block must have at least one *except* block
- A SyntaxError will be raised if a *try* block does not have an *except* block
- A *try* block can have multiple *except* blocks
- A *try* block can have one optional *else* block
- A *try* block can have one optional *finally* block

1.4 Custom Errors

- Since all errors are classes, custom errors can also be created and raised explicitly
- To create a custom error, inherit the basic error class, *Exception* into the error class
- Have a constructor function with one parameter, a string, which will be the error message
- Call the parent class constructor with the message string
- Raise the error explicitly with *raise ErrorName* and pass the error message
- Example is illustrated in Figure 1.34

```
class InvalidAgeError(Exception):
    def __init__(s, m, a):
        print('Given Age: '+str(a))
        super().__init__(m)

try:
    age = int(input())
    if age < 18 or age > 100:
        raise InvalidAgeError('Error: Age must be between 18 and 100', age)
    else:
        print('Valid age')
except ValueError as e:
    print(e)
except InvalidAgeError as e:
    print(e)
```

```
17
Given Age: 17
Error: Age must be between 18 and 100
```

Figure 1.34: Custom Error

Chapter 2

File Handling and I/O

2.1 Functions for File Handling

- Python provides various built-in functions for file handling
- A file can be opened using *open(str FilePath, str AccessMode)* function
- The open() function takes a string parameter with the absolute or relative file path and another string parameter for access mode (Read, Write, Append, Read+, Write+, Append+)
- If no AccessMode is specified, the file is opened in Read-only mode
- The open() function returns a file object called *_io.TextIOWrapper*
- The file has a pointer called *seek*, which is similar to the cursor and performs read and write operations from where it exists

Access Mode	**Parameter Used**	**Explanation**
Read	'r'	Opens the file in read-only mode. File seek will be at the beginning of the file. Raises FileNotFoundError if the file doesn't exist
Write	'w'	Opens the file in write-only mode. The file seek will be at the beginning of the file. Creates a new file if the file doesn't exist
Append	'a'	Opens the file in append-only mode. The file seek will be at the end of the file. Raises FileNotFoundError if the file doesn't exist
Read + Write	'r+'	Opens the file in read and write mode. The file seek will be at the beginning of the file. Raises FileNotFoundError if the file doesn't exist
Write + Read	'w+'	Opens the file in write and read mode. The file seek will be at the beginning of the file. Creates a new file if the file doesn't exist
Append + Read	'a+'	Opens the file in append and read mode. The file seek will be at the end of the file. Raises FileNotFoundError if the file doesn't exist

Table 2.1: Access Modes for open()

- File object has various attributes. Major attributes include:
  - closed: True if the file is closed
  - mode: String, mode in which the file is opened
- Major functions of the file object:
  - close(): Closes the file, sets closed = True
  - read(): Returns a string with the contents of the file. Read operation happens from the current position of the seek. The file seek will move to the end of the file upon completion of reading.
  - write(String): Writes the given String into the file starting from the position of the file seek. Only possible if the file is opened in write ('w') or read + write ('r+') or write + read ('w+') modes
  - seek(pos): Moves the file seek to the given pos
  - tell(): Returns an integer with the current position of the file seek

2.2 File I/O – Text Files

- Text files or Plaintext files are files with an extension of *.txt*
- Files can be opened using open()

2.2.1 Reading contents of a text file

- The file must exist and contain some text
- Open the file in 'r' or 'r+' or 'w+', store it in a variable
- Use read() function to get the contents of the file as a string
- If attempted to read again, the read() returns an empty string since the file seek is at the end of the file
- Move the file seek back to 0 to read again
- Close the file using close()

- Example is illustrated in Figure 2.1

```
f = open('a.txt')
content = f.read()
print(content)
# Re-reading
content = f.read()
print('Re-reading without setting seek to 0')
print(content)
print('After setting seek to 0')
f.seek(0)
content = f.read()
print(content)
```

```
This is a text file.
This is Line 2.
Re-reading without setting seek to 0

After setting seek to 0
This is a text file.
This is Line 2.
```

Figure 2.1: Reading a Text File

2.2.2 Writing contents into text file

- Open a file in 'w' or 'w+'
- The file seek will be at 0, anything written using write() will be overwritten on the existing contents of the file
- Use write(String)
- Close the file using close()
- View the changes

- Illustrated in Figure 2.2. Figure 2.3 has the resultant file

```
f = open('b.txt', 'w')
# b.txt is empty
f.write('Some Content')
f.close()
```

Figure 2.2: Writing a Text File

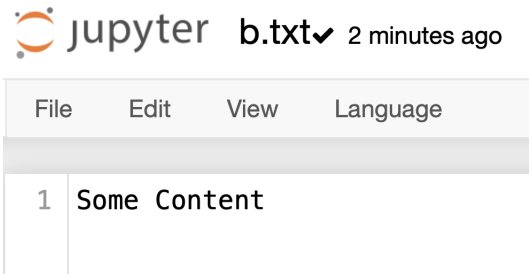

Figure 2.3: Resultant File

- The same file has been overwritten as shown in 2.4. The resultant file is in Figure 2.5

```
f = open('b.txt', 'w')
# b.txt is not empty
f.write('Some New Content for Overwriting')
f.close()
```

Figure 2.4: Overwriting a Text File

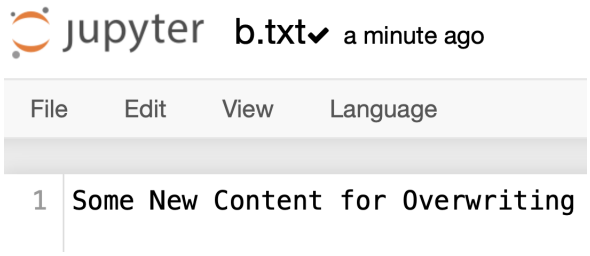

Figure 2.5: Overwritten File

2.2.3 Copying contents of one file to another

- Open source file in read mode
- Open destination file in write mode
- Read the contents of source file
- Write into the destination file
- Close both the files
- Illustrated in Figure 2.6. Destination file is in Figure 2.7

```
src = open('source.txt')
print('Contents of the source file: ')
content = src.read()
print(content)
dest = open('dest.txt', 'w')
dest.write(content)
print('Copy done')
src.close()
dest.close()
```

```
Contents of the source file:
Contents of the File
Copy done
```

Figure 2.6: Copying from one File to another

Figure 2.7: Destination File

2.3 File I/O – CSV Files

- CSV Files have data separated by comma
- Default delimiter for the file is comma, although, any other delimiter such as pipe (|) or semicolon (;) can be used
- Sample CSV file is shown in Figure 2.8

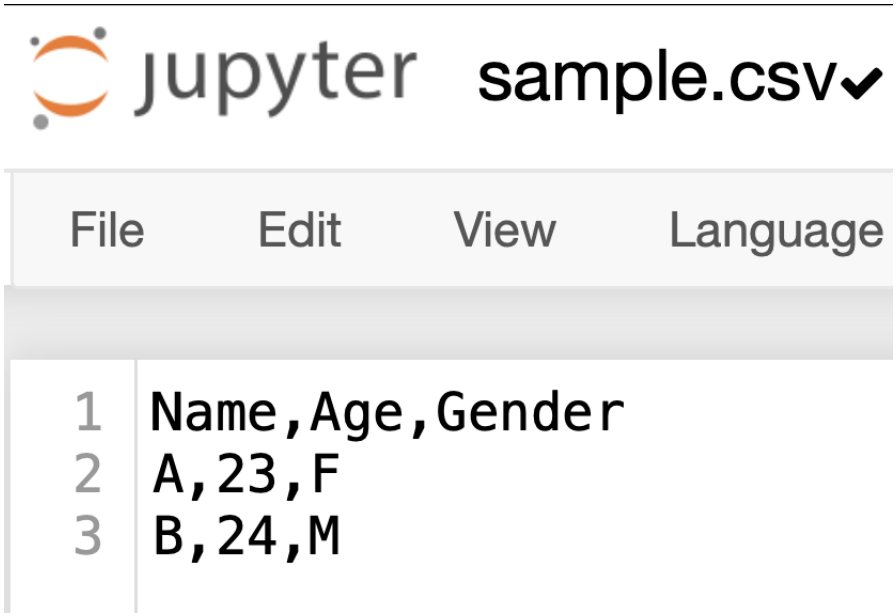

Figure 2.8: Sample CSV File

- CSV files are handled using a built-in csv module

2.3.1 Reading CSV Files

- CSV Files can be read using *reader* and *DictReader* classes of csv module
- Both reader and DictReader objects are dependent on the file, meaning, if the file is closed, the contents of the objects will become inaccessible
- Both reader and DictReader are iterables
- Upon running a for loop on any of these objects, the file seek moves to the end of the file
- Attempting to rerun the for loop will print nothing
- ValueError is raised if the file is closed and the for loop is run on the reader or DictReader objects

reader

- A reader object can be created using csv.reader()
- csv.reader() accepts one mandatory parameter (An open file in read mode) and an optional parameter, delimiter (Default value of delimiter is comma)
- The reader object is an iterable with each row of the CSV file as a list of strings
- The file cursor moves to the end of the file when a for loop is run on the reader object
- A ValueError is raised when the for loop is run after the file is closed
- Ideal to typecast the reader object into a list so that the object can be used later

- Example is illustrated in Figure 2.9

```
import csv
```

```
f = open('sample.csv')
r = csv.reader(f)
for i in r:
    print(i)
```

```
['Name', 'Age', 'Gender']
['A', '23', 'F']
['B', '24', 'M']
```

Figure 2.9: Reading CSV using reader

- Example of a CSV file with delimiter as | is illustrated in 2.10

```
f = open('sample.csv')
r = csv.reader(f, delimiter = '|')
for i in r:
    print(i)
```

```
['Name', 'Age', 'Gender']
['A', '23', 'F']
['B', '24', 'M']
```

Figure 2.10: Reading CSV file with delimiter | using reader

DictReader

- DictReader (DR) is an iterable with each row of the CSV file as dictionary
- The first row items become the keys of the dictionary, while the subsequent row values become the values of the dictionaries of the DR
- Default delimiter: Comma. Can be anything, specified with delimiter parameter
- Ideal to typecast it to a list to reuse the object

- Example of DR is illustrated in Figure 2.11

```
f = open('sample.csv')
dr = list(csv.DictReader(f, delimiter = '|'))
for i in dr:
    print(i)
```

```
{'Name': 'A', 'Age': '23', 'Gender': 'F'}
{'Name': 'B', 'Age': '24', 'Gender': 'M'}
```

Figure 2.11: Reading CSV File using DictReader

2.3.2 Writing CSV Files

- CSV files can be written using writer and DictWriter classes of csv module
- A file must be open in write mode to perform writing

writer

- csv.writer takes a file as parameter
- The file must be opened in write mode
- Functions of writer:
  - writerow(List): Writes the given List as a row of the CSV file
  - writerows(IterOfLists): Writes the lists of the given iterable as individual rows of the CSV file
- An example of creating a CSV file using writer is illustrated in Figure 2.12

```
l = [['Name', 'Age', 'Gender'], ['A', 23, 'F'], ['B', 23, 'M']]
with open('a.csv', 'w') as f:
    w = csv.writer(f)
    for row in l:
        w.writerow(row)
```

Figure 2.12: Writing CSV File using writer

DictWriter

- csv.DictWriter takes an open file in write mode as the first parameter
- The second parameter is the fieldnames, a List of all the field names of the CSV file
- Functions of DictWriter:
  - writeheader(): Writes the fieldnames into the CSV file

  - writerow(Dictionary): Writes the given Dictionary as the rows of the CSV file
  - writerows(IterOfDicts): Writes the dictionaries of the given iterable as individual rows of the CSV file

- An example of creating a CSV file using DictWriter is illustrated in Figure 2.13

```
l = [{'Name':'A', 'Age':23, 'Gender':'F'},
    {'Name': 'B', 'Age':24, 'Gender':'M'}]
fn = ['Name', 'Age', 'Gender']
with open('b.csv', 'w') as f:
    dw = csv.DictWriter(f, fieldnames = fn)
    dw.writeheader() # Writes the fieldnames
    dw.writerows(l)
```

Figure 2.13: Writing CSV File using DictWriter

2.3.3 CSV File Handling using pandas

- pandas is a library for data manipulation and analysis used in Python
- pandas has a class *DataFrame*, which can be used to read CSV files and manipulate the same
- read_csv(FilePath) returns a DataFrame object of the given File. An example is shown in Figure 2.14

```
df = pd.read_csv('c.csv')
df
```

	Name	Hire Date	Salary	Sick Days Remaining
0	GC	03/15/14	50000	10
1	JC	06/01/15	65000	8
2	EI	05/12/14	45000	10
3	TJ	11/01/13	70000	3

Figure 2.14: Reading CSV File using pandas

- Some major operations:
  - To display all the columns of a DataFrame, df, use *df.columns*
  - Each column in a DataFrame is an object of pandas.Series
  - To write a DataFrame into a CSV file, use pandas.to_csv(df)
  - To print an individual row of the DataFrame, use iloc. df.iloc[0] returns the first row of the DataFrame
  - Overriding the name of the columns: Using names = List of new attributes, the names of the columns can be modified as shown in Figure 2.15

```
df = pd.read_csv('c.csv',
     names = ['Initials', 'DOJ', 'Basic', 'Leaves'])
df
```

	Initials	DOJ	Basic	Leaves
0	Name	Hire Date	Salary	Sick Days Remaining
1	GC	03/15/14	50000	10
2	JC	06/01/15	65000	8
3	EI	05/12/14	45000	10
4	TJ	11/01/13	70000	3

Figure 2.15: Overriding field names in pandas

  - Setting an index column to an existing column from the file as shown in 2.16

```
df1 = pandas.read_csv('hr.csv', index_col = 'Name')
df1
```

Name	Hire Date	Salary	Sick Days Remaining
GC	03/15/14	50000	10
JC	06/01/15	65000	8
EI	05/12/14	45000	10
TJ	11/01/13	70000	3

Figure 2.16: Using a different index column

2.4 File I/O – JSON Files

- JSON stands for JavaScript Object Notation
- JSON is a standard for storing intermediary data as key-value pairs (KVPs)
- JSON objects can be stored as files with *.json* as an extension
- Every JSON file can have one object or an array of multiple objects
- A JSON object can have multiple key-value pairs between a pair of curly braces
- A JSON array has multiple JSON objects separated by a comma
- Example, {*"Name": "Tony Stark", "Alias": "Iron Man"*} is a JSON object
- Example, *[{"Name": "Tony Stark", "Alias": "Iron Man"}, {"Name": "Steven Rogers", "Alias": "Captain America"}]* is an array with 2 objects
- JSON keys or values can have strings within double quotes only
- A JSON object exists as a string in Python
- In Python, JSON file handling is done using a module called *json*
- Different JSON and corresponding Python types are tabulated in Table 2.2

Python Type	JSON Type
Dictionary	Object
List	Array
Tuple	Array
int/float	Number
str	String
True/False	true/false
None	null

Table 2.2: JSON Datatypes and Python Types

2.4.1 Functions for JSON handling

- **json** module has 4 functions for JSON handling
  - **json.load**: Loads the contents of a JSON file and return a Python object (As per Table 2.2)
  - **json.loads**: Creates a Python object (As per Table 2.2) by accepting a JSON string
  - **json.dump**: Creates a JSON file with the given Python object
  - **json.dumps**: Creates a JSON string of the given Python object
- Details are tabulated in Table 2.3

Function	Parameters	Return	What it does	Example
json.load(file)	file (.json file opened in read mode)	Python Object (JSON Array becomes list, Object becomes Dictionary)	Returns a Python dictionary or list by reading the given JSON file	Fig 2.17
json.loads(jsonstr)	jsonstr (A JSON String)	Python Object (JSON Array becomes list, Object becomes Dictionary)	Returns a Python dictionary or list by reading the given JSON string	Fig 2.18
json.dump(obj, file)	obj (A Python list or dictionary), file (.json file opened in write mode)	None	Writes the given obj into the file as a JSON object or array	Fig 2.19
json.dumps(obj)	obj (A Python list or dictionary)	str	Creates a JSON-formatted string of the given obj	Fig 2.20

Table 2.3: JSON Handling – Functions of **json** module

```
# json.load
with open('q.json') as f:
    obj = json.load(f)
obj
```

```
[{'id': 101, 'name': 'ABC'},
 {'id': 102, 'name': 'DEF'},
 {'id': 103, 'name': 'GHI'}]
```

Figure 2.17: json.load

```
# json.loads
json_string = '{"name":"ABC", "id": 100}'
obj = json.loads(json_string)
obj
```

```
{'name': 'ABC', 'id': 100}
```

Figure 2.18: json.loads

```
# json.dump
obj = [{'Name': 'Tony Stark', 'Alias': 'Iron Man'},
       {'Name': 'Steven Rogers', 'Alias': 'Captain America'}]
with open('marvel.json', 'w') as f:
    json.dump(obj, f)


print('Reading the file created: \n')
print(open('marvel.json').read())
```

```
Reading the file created:

[{"Name": "Tony Stark", "Alias": "Iron Man"}, {"Name": "Steven Rogers",
"Alias": "Captain America"}]
```

Figure 2.19: json.dump

```
# json.dumps
obj = [{'Name': 'Tony Stark', 'Alias': 'Iron Man'},
       {'Name': 'Steven Rogers', 'Alias': 'Captain America'}]
json_str = json.dumps(obj)
json_str
```

```
'[{"Name": "Tony Stark", "Alias": "Iron Man"}, {"Name": "Steven Rogers", "Alias": "Captain
America"}]'
```

Figure 2.20: json.dumps

2.5 File I/O – XML Files

- XML files hold non-relational data
- Unlike JSON, XML files hold data using tags and text
- Keys become tags, values become text
- A sample XML file can be seen in Figure 2.21
- In Figure 2.21, the first line is the XML version header
- XML files store data using user-defined tag names

```xml
<?xml version = "1.0"?>
<characters>
    <character>
        <name>Tony Stark</name>
        <alias>Iron Man</alias>
        <item>Arc Reactor Core</item>
    </character>
    <character>
        <name>Steven Rogers</name>
        <alias>Captain America</alias>
        <item>Strontium Shield</item>
    </character>
</characters>
```

Figure 2.21: Sample XML File

XML Files can be handled using 3 techniques

- Element Tree
- DOM
- BeautifulSoup

2.5.1 Element Tree

- An XML file can be written as an element tree where all the tags become elements, the data becomes the leaf node
- Python has xml.etree module which can be used to handle XML files using Element Tree
- Classes
  - Element: Element is a node in the tree, and has a tag (Name of the element). Created using Element(name). Can be appended to another element using append (Eg., parent.append(child))
  - SubElement: An element that's attached to a parent node. Created using SubElement(parentNode, name)
  - Both Element and SubElement classes have attributes tag (Name of the element in the tree, later the name of the tag in XML file) and text (Data node of the element, later the data text of the tag in XML file)
- An Element Tree representation of the XML file in Figure 2.21 is shown in Figure 2.22
- Steps in creating an XML file using ElementTree:
  - Create a list of dictionaries, using which, the XML file is to be created
  - Create a root element using Element(name)
  - Create sub-elements of root for every dictionary in the list
  - For every KVP in the dictionary, create a sub-element of the sub-element of the root node using SubElement(childOfRoot, key)
  - Set the text of the created sub-element to the value of the KVP
  - Create an ElementTree of the root
  - Write the tree into a file after opening in write-bytes mode
  - Example code is in Figure 2.23

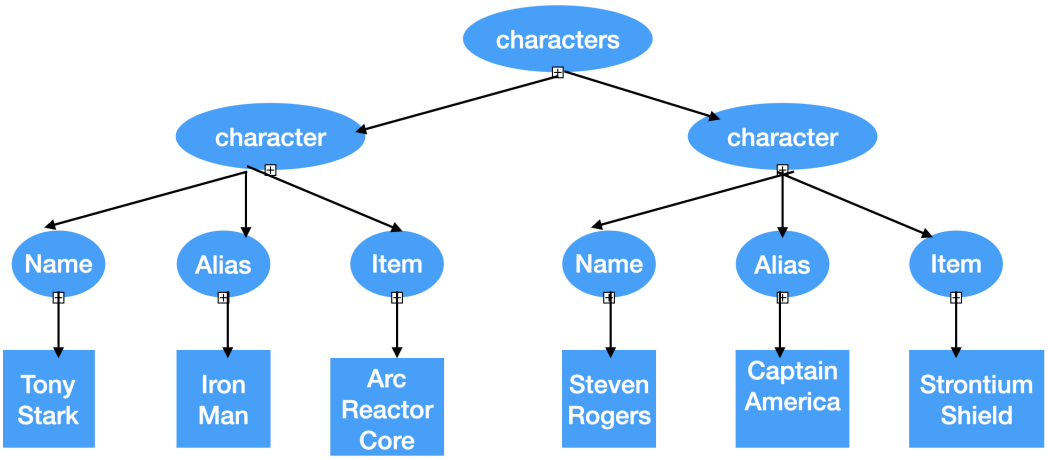

Figure 2.22: Sample ElementTree

```
l = [{'Name': 'Tony Stark', 'Alias': 'Iron Man', 'Item': 'Arc Reactor Core'},
     {'Name': 'Steven Rogers', 'Alias': 'Captain America', 'Item': 'Strontium Shield'}]
root = xml.Element('characters')
for d in l:
    ch = xml.SubElement(root, 'character')
    for key in d:
        ele = xml.SubElement(ch, key)
        ele.text = str(d[key])
tree = xml.ElementTree(root)
with open('mcu.xml', 'wb') as f:
    tree.write(f)
print('Resultant file: \n')
print(open('mcu.xml').read())
```

```
Resultant file:

<characters><character><Name>Tony Stark</Name><Alias>Iron Man</Alias><Item>Arc Reactor Core
</Item></character><character><Name>Steven Rogers</Name><Alias>Captain America</Alias><Item
>Strontium Shield</Item></character></characters>
```

Figure 2.23: Creating XML – ElementTree

```
# Parsing
tree = xml.ElementTree(file='mcu.xml')
root = tree.getroot()
l = []
for student in root.findall('character'):
    sd = {}
    for data in student:
        sd[data.tag] = data.text
    l.append(sd)
for i in l: print(i)
```

```
{'Name': 'Tony Stark', 'Alias': 'Iron Man', 'Item': 'Arc Reactor Core'}
{'Name': 'Steven Rogers', 'Alias': 'Captain America', 'Item': 'Strontium Shield'}
```

Figure 2.24: Reading XML – ElementTree

2.5.2 DOM

- DOM stands for Document Object Model
- In Python, a minimal DOM implementation using xml.dom.minidom
- Document becomes a part of the XML tree, the root of the XML tree is the child of the document itself
- Steps:
  - Create a **minidom.Document object**
  - Create the root element using **document.createElement(rootName)**
  - Append root to the document object using **document.appendChild(root)**
  - For every dictionary in the list, create an element and append it to the root
  - For every KVP in the dictionary, create an element and append it to the previously created child of the root
  - Create a text node with the value and append it to the element created
  - Create the XML string using doc.toprettyxml() with indent and write into the file
  - The code to create an XML file using DOM is shown in Figure 2.25

```
l = [{'Name': 'Tony Stark', 'Alias': 'Iron Man', 'Item': 'Arc Reactor Core'},
     {'Name': 'Steven Rogers', 'Alias': 'Captain America', 'Item': 'Strontium Shield'}]
doc = md.Document()
root = doc.createElement('characters')
doc.appendChild(root)
for d in l:
    ch = doc.createElement('character')
    root.appendChild(ch)
    for key in d:
        ele = doc.createElement(key)
        ele_text = doc.createTextNode(d[key])
        ele.appendChild(ele_text)
        ch.appendChild(ele)
xml_str = doc.toprettyxml(indent = '\t')
with open('mcudom.xml', 'w') as f:
    f.write(xml_str)
print('File: \n')
print(open('mcudom.xml').read())
```

```
File:

<?xml version="1.0" ?>
<characters>
        <character>
                <Name>Tony Stark</Name>
                <Alias>Iron Man</Alias>
                <Item>Arc Reactor Core</Item>
        </character>
        <character>
                <Name>Steven Rogers</Name>
                <Alias>Captain America</Alias>
                <Item>Strontium Shield</Item>
        </character>
</characters>
```

Figure 2.25: XML Creation – DOM

2.5.3 BeautifulSoup

- BeautifulSoup is widely used for web-scraping and to remove unnecessary data to retain meaningful data on websites
- Python provides lxml, a XML parser which is used by BeautifulSoup
- BeautifulSoup can be implemented using bs4 package (Needs to be installed first)
- Steps:
  - Create an element tree using etree of lxml. Follow the steps of ElementTree creation until the creation of the tree using the root
  - Create XML string using **tostring** function of etree, use optional *pretty_print = True*

- Create s BeautifulSoup object using the XML string created and format using XML formatted
- Convert the BeautifulSoup object to string and write into an open XML file
- The program is shown in Figure 2.26

```
l = [{'Name': 'Tony Stark', 'Alias': 'Iron Man', 'Item': 'Arc Reactor Core'},
     {'Name': 'Steven Rogers', 'Alias': 'Captain America', 'Item': 'Strontium Shield'}]
root = etree.Element('characters')
for d in l:
    ch = etree.SubElement(root, 'character')
    for key in d:
        ele = etree.SubElement(ch, key)
        ele.text = str(d[key])
xml_string = etree.tostring(root, pretty_print = True)
soup = BS(xml_string, 'xml')
with open('mcubs.xml', 'w') as f:
    f.write(str(soup))
print('File: \n')
print(open('mcubs.xml').read())
```

```
File:

<?xml version="1.0" encoding="utf-8"?>
<characters>
<character>
<Name>Tony Stark</Name>
<Alias>Iron Man</Alias>
<Item>Arc Reactor Core</Item>
</character>
<character>
<Name>Steven Rogers</Name>
<Alias>Captain America</Alias>
<Item>Strontium Shield</Item>
</character>
</characters>
```

Figure 2.26: XML Creation – BeautifulSoup

2.6 Object Serialisation – pickle

- Serialisation (Pickling) – The process of converting a Python object into a stream of bytes and then into a file
- Deserialisation (Depickling/Unpickling) – The process of converting a file into a stream of bytes and then into a Python object
- In Python, serialisation and deserialisation can be done using **pickle** module
- Objects that can be pickled in Python:
  - Classes (At the top level of the module)
  - Objects of the user-defined classes
  - Lists, Tuples, Sets, Dictionaries, Strings (As long as they contain only pickleable objects)
  - None, True, False
  - User-defined functions (Defined using **def** only) (**Lambda functions can't be pickled**)
  - int, float, complex
  - Built-in functions (At the top level of the module)
- Uses and Applications:
  - Persistence – Objects can be created once, and saved for reuse later. Very useful in Machine Learning
  - Inter-process Communication – Output of one file can be pickled and can be used as an input to another file by unpickling it

- Functions of pickle (Explained in Table 2.4):
  - pickle.dump()
  - pickle.dumps()
  - pickle.load()
  - pickle.loads()

Function	Parameters	Return	What it does	Example Fig
dump(obj, file)	obj (Any pickleable object) and file (A file opened in write-bytes)	None	Creates a pickle file after serialising the given object	2.27
dumps(obj)	obj (Any pickleable object)	bytes	Returns a bytes object after serialising the obj	2.28
load(file)	file (A file opened in read-bytes)	Python Object	Reads a file and returns a Python object after deserialisation	2.29
loads(sd)	sd (Serialised data, type bytes)	Python Object	Deserialises the given serialised bytes and returns a Python object	2.30

Table 2.4: Functions of pickle

```
# pickle.dump()
l = [1, 2, 3]
with open('l1.pkl', 'wb') as f:
    pickle.dump(l, f)
print('Pickled file contents: \n')
print(open('l1.pkl', 'rb').read())
```

```
Pickled file contents:

b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.'
```

Figure 2.27: pickle.dump()

```
# pickle.dumps()
l = [1, 2, 3]
sd = pickle.dumps(l)
print(sd)
```

```
b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.'
```

Figure 2.28: pickle.dumps()

```
# pickle.load()
with open('l1.pkl', 'rb') as f:
    l = pickle.load(f)
l
```

```
[1, 2, 3]
```

Figure 2.29: pickle.load()

```
# pickle.loads()
sd = open('l1.pkl', 'rb').read()
l = pickle.loads(sd)
l
```

```
[1, 2, 3]
```

Figure 2.30: pickle.loads()

Chapter 3

Modules and Packages

3.1 Modules and Packages

- **Module** – A single Python file with .py as the extension
- **Package** – A directory with multiple .py files
- A module can have any number of classes, variables, functions
- A package can have sub-packages and any number of Python scripts
- Contents of the modules can be imported into other files
- Each package has a __init__.py file, which can be empty or can be used to initialise the contents of the package
- Uses:
  - Code Reusability
  - Code Organisation

3.1.1 Importing – At same level

- Assume a directory structure as shown in Figure 3.1

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

Figure 3.1: Directory Structure – eCommerce

- To import the module *database* into products.py: ***import database*** (Contents of the module can be accessed using "." For example, to access the Database class in the module, use ***database.Database***)
- To import all the contents of the module *database* into products.py: ***from database import **** (No need to use the name of the module, the contents will be imported into the current module)
- To import Database class from *database* module to product.py: ***from database import Database***
- To import Database class and initialise() from *database* module to products.py: ***from database import Database, initialise***
- To import the *database* module and give it an alias name: ***import database as db*** (Name *db* will be used instead of *database*)

3.1.2 Absolute Import

- Providing the complete path to import modules or the contents of the modules in a package
- Consider a directory structure as shown in Figure 3.2

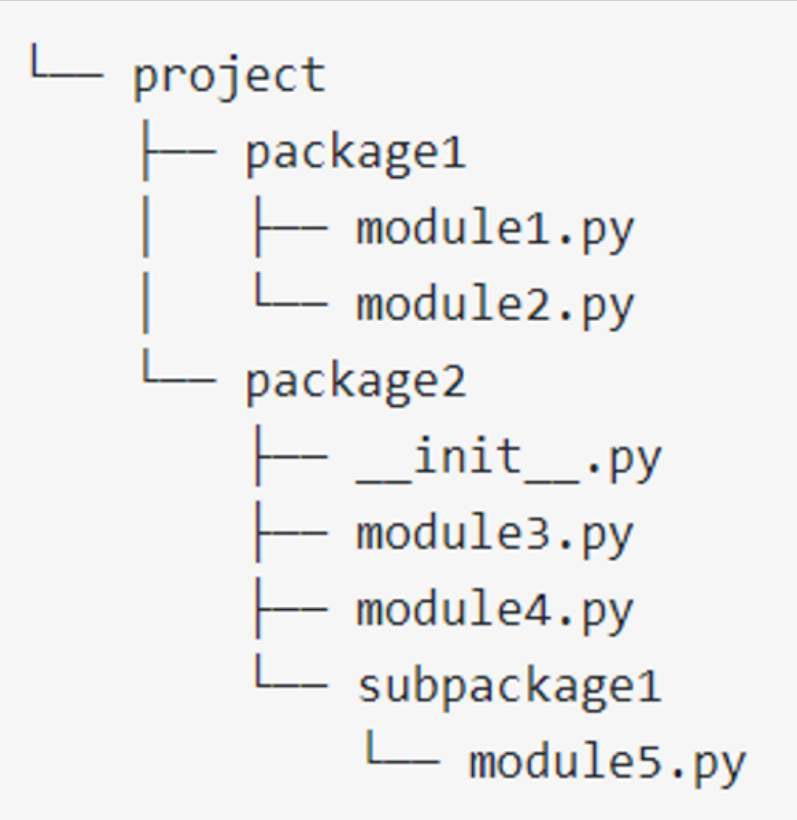

Figure 3.2: Sample Directory – Absolute Import

- When package2 is imported using **import package2**, the contents of the package2 becomes available and can be accessed using dot notation
- Upon importing the package, the __init__.py file gets executed
- To import contents of module3.py into module2.py, ***from package2.module3 import ****

- To import the entire module of module3.py into module2.py, ***import package2.module3***. Contents of module3 can be accessed using dot notation
- To import contents of module5.py into module2.py,

  from package2.subpackage1.module3 import *
- Absolute import statements are lengthy but, give an idea of the path at which the imported module is located
- Absolute import statements remain valid in all the modules of the folder until the structure of the directory is modified

3.1.3 Relative Import

- Relative import statements can be used to avoid writing the full path
- Modules and contents are accessed with relative path
- Single dot (.) can be used to access contents of the same directory
- Double dot (..) can be used to access the contents of the parent directory
- Consider the directory structure in Figure 3.2
- To import the contents of module4 into module5, ***from ..module4 import ****
- Relative imports are only valid in one directory as the position of the resource will be different from one directory to another, i.e., the relative import statements need to be modified in different modules in different directories

3.2 Standard Library

- Python has a set of modules that are standard library modules
- Some packages and modules are installed by default with every installation or distribution of Python
- These modules and packages deal with the working of Python and can be repeatedly used in different programs
- Some standard library modules:
  - **os** – For handling OS-related functionalities
  - **sys** – For handling system and environment-related functionalities
  - **random** – For generating pseudo-random objects
  - **datetime** – For handling date and time and their formats
  - **math** – For mathematical functions and operations
  - File handling **(json, csv, xml modules)**
  - **re** – For Regular Expression handling and matching

3.2.1 os

- Provides functions for interacting with the operating system
- Various functions of os module:
  - **os.listdir()** – Returns a List of the contents of the current directory
  - **os.getcwd()** – Returns the path of the present/current working directory as a str
  - **os.mkdir(dirname)** – Creates an empty directory, where dirname is a str
  - **os.open(), os.read(), os.write()** – To open, read, and write the files
  - **os.kill(pid)** – Kills a process based on the integer pid given
  - **os.getpid()** – Returns the pid of the current process
  - **os.rmdir(dirname)** – Deletes the given directory, dirname

3.2.2 sys

- Provides functions to handle the working environment and the system
- **sys.version** – Provides the version of Python
- **sys.platform** – Returns the underlying OS platform
- **sys.argv** – A List of strings with the command line arguments provided, the first object will be the name of the file
- **sys.stdin** – Returns the standard input and its address
- **sys.stdout** – Returns the standard output and its address
- **sys.stderr** – Returns the standard error output and its address
- **sys.module** – Provides a list of all available modules
- Various functions:
  - **sys.exit(code)** – Exits with the given code
  - **sys.getsizeof(obj)** – Returns the size of an object in bytes

3.2.3 datetime

- Used to handle dates, times, and formats
- Some functions and classes:
  - **Class datetime** – Represents a specific point in time with both date and time components (year, month, day, hour, minute, second, microsecond). Instances of this class are immutable
  - **Class date** – Represents a date with time as 00:00:00
  - **Class time** – Represents time (HH:MM:SS) without any date
  - **Class timedelta** – An object with difference in time, in terms of days, hours, seconds
  - date objects can be added and subtracted with the number of days (Using timedelta object) using + and - respectively

  - time objects can be added and subtracted with the number of hours or minutes
  - Two dates can be subtracted to find the date difference
  - **utcnow()** – Returns datetime object with current UTC time
  - **Timezone Management** – Timezones can be managed using datetime
  - **Date and time formatting** – date, time, and datetime objects can be formatted in any preferable format

3.2.4 collections

- To use additional data structures and collections than lists, tuples, sets, dictionaries, strings etc.
- Classes:
  - **namedutple** – tuple subclasses with named fields, a tuple-like object with named fields
  - **deque** – A double-ended queue-like structure. A list-like object with append and pop from both ends. Can be restricted for pop and append
  - **ChainMap** – Dictionary-like class for creating a single view of multiple mappings
  - **Counter** – Dictionary subclass, used for counting counting hashable objects
  - **OrderedDict** – Ordered dictionary in which the insertion order is maintained, unlike dictionaries
  - **defaultdict** – Provides default values to missing keys to avoid KeyErrors
  - **UserDict** – Wrapper of dictionary class
  - **UserList** – Wrapper of list class
  - **UserString** – Wrapper of str class