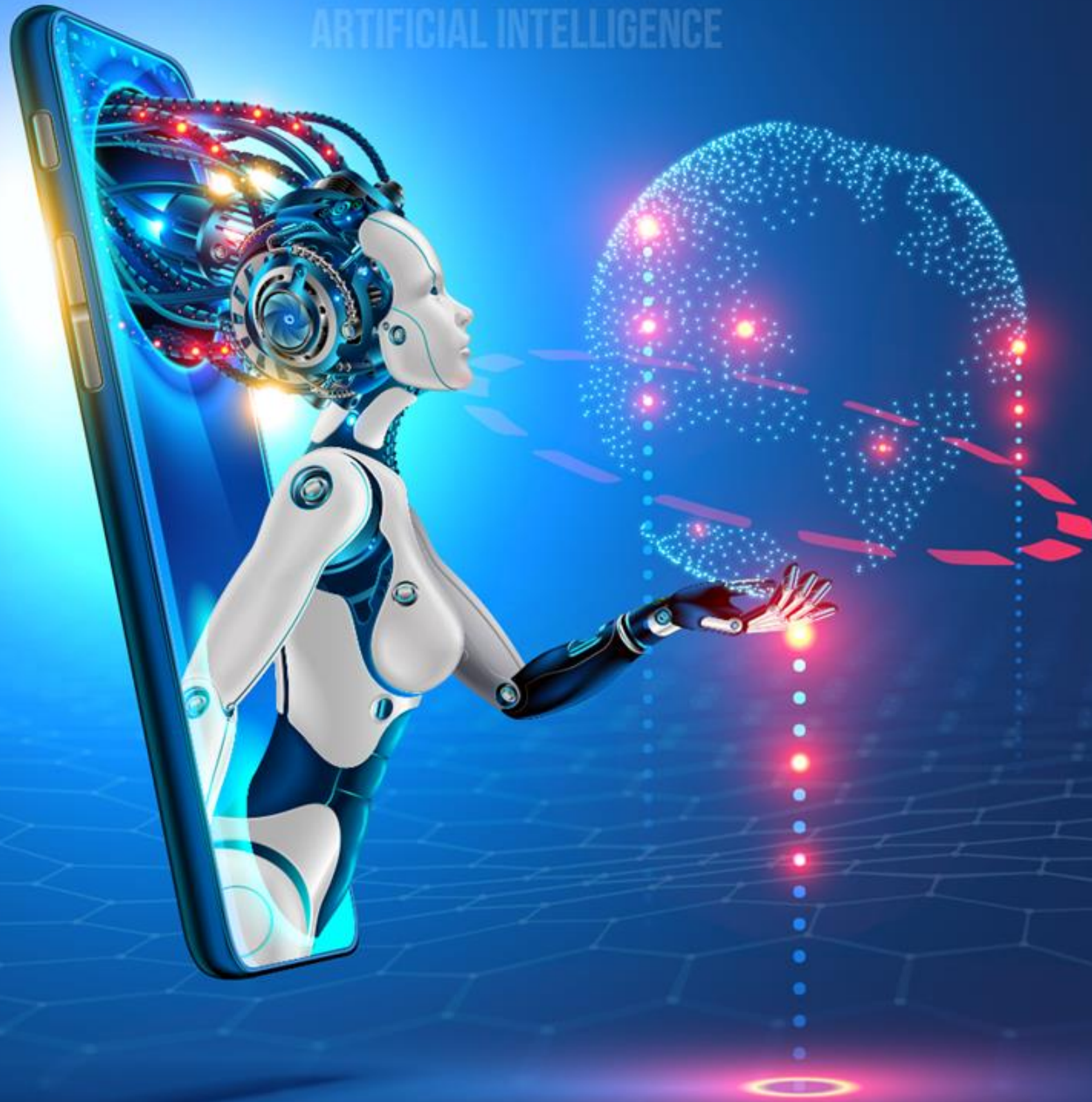


DATA AND
ARTIFICIAL INTELLIGENCE



Programming Basics and Data Analytics with Python



File Handling, Exception Handling, and Package Handling

Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Perform file handling operations using built-in functions
- 🕒 Explain directories in Python
- 🕒 Explain exception handling and clauses
- 🕒 Describe modules and packages and perform module import



File Handling

What Is File Handling?

The concept of file handling is to handle files on a computer using built-in functions that perform file operations.

File operations can be performed in the following order using Python:

1. Open a file
1. Read or write file
1. Close the file



File Handling Operations

File handling operations can be performed using built-in functions and a *file object* in Python.

Some of the built-in functions used for file operations are:

- *open()*: opens a file
- *write()*: writes to a file
- *read()*: reads a file
- *close()*: closes a file

All attributes related to a *file* object are:

- *closed()*: checks if a file is closed
- *mode()*: returns the access mode
- *name()*: returns the file name
- *softspace()*: checks if any space is explicitly required with print

File Opening and Closing

File Opening and Closing

The ***open()*** function: This function is used to open an existing file in a specific *access mode*. It takes the *filename* and *access mode* as input parameters and returns the *file object* as an output.

Syntax:

```
file_object = open(fileName, accessMode)
```

#where *accessMode* can be:

- "r", for reading
- "r+", for both reading and writing
- "w", for writing
- "a", for appending

Example:

```
# Open "demo.txt" in the reading mode
```

```
file = open('demo.txt', "r")
```

```
# Print each line in the file one by one
```

```
for each in file:
```

```
    print (each)
```


File Opening and Closing

The ***close()*** method: This method is used to close an already opened file and free up the resources tied with the file.

Syntax:

```
file_object.close()
```

#where *file_object* is the file to be closed

Example:

```
# Open "test.txt" in the reading mode.
```

```
file = open(test.txt', 'r')
```

```
# Print the file name
```

```
print ("Name of the file: ", file.name)
```

```
# Close the file
```

```
file.close()
```

Reading and Writing Files

Reading and Writing Files

The ***read()*** method: This method is used to read text from a file opened in reading mode (***r***). It takes the *size* as an optional input parameter and returns a *string* as an output.

Syntax: *str = file_object.read(size)*

Example:

```
# Open "demo.txt" in the reading mode.  
file = open('demo.txt', 'r')
```

```
# Print the whole text as a string  
print (file.read())
```

```
# Print the 5 characters from the file  
print (file.read(5))
```

```
# Open "demo.txt" in the reading mode.  
file = open('demo.txt', 'r')  
  
# Print the whole text as a string  
print (file.read())  
  
print ("The first 5 characters from the file are:")  
print (file.read(5))
```

```
Welcome to Jupyter Notebook.  
This is a demo text file.  
Use this file to perform read and write operations.  
The first 5 characters from the file are:  
Welco
```

Reading and Writing Files

The ***readline()*** method: This method is used to read individual lines from a file opened in reading ***r*** mode. It reads a file until the newline, including the newline character (***\n***).

Syntax:

```
str = file_object.readline()

#where str is the string output
```

Example:

```
# Open "demo.txt" in the reading mode.
file = open(demo.txt', 'r')

# Print the whole text as a string
print (file.readline())
```


Reading and Writing Files

The ***seek()*** method: This method is used to change the current file cursor while reading files. It takes the *position* as an input and moves the cursor to that position.

The ***tell()*** method: This method returns the current file cursor location.

Syntax:

```
file_object.seek(position)
```

#where *position* is the cursor position
where cursor will be moved

```
file_object.tell()
```

Example:

```
# Open "demo.txt" in the reading mode.  
file = open(demo.txt', 'r')
```

```
# Print the current cursor position  
print (file.tell())
```

```
# Move the cursor to initial position  
file.seek(0)
```

Reading and Writing Files

The ***write()*** method: This method is used to write any string or text to a file opened in write (***w***), append (***a***), or exclusive creation (***x***) mode. It takes a *string* and *access mode* as the input parameters and returns the number of characters written to the file.

Syntax: `file_object.write(string, access_mode)`

Example:

```
# Create a file "test.txt" in the writing mode.  
file = open('test.txt', 'w')  
  
file.write("This is a test file.")  
  
file.close()
```

```
# Create a file "test.txt" in the writing mode.  
file = open('test.txt', 'w')  
  
# Write a string to the file  
file.write('This is a test file.')  
  
# Close the file  
file.close()
```

Reading and Writing Files

Example 1:

Writing in an existing file in *append mode* to append the existing content

```
#Open an existing file "test.txt" in append mode
file = open('test.txt', 'a')

# Write a string to the file
file.write("Second statement in the file.")

# Close the file
file.close()
```

examples.ipynb × test.txt

1 This is a test file.Second statement in the file.

Example 2:

Writing in an existing file in *write mode* to overwrite the existing content

```
#Open an existing file "test.txt" in write mode
file = open('test.txt', 'w')

# Write a string to overwrite any content
file.write("This is the new text.")

# Close the file
file.close()
```

examples.ipynb × test.txt ×

1 This is the new text.|

Directories in File Handling

Directories in File Handling

Directories are part of a hierarchical file system to store folders and files in an organized tree structure and act as a container for them.

File handling operations use a directory to list all the files contained in it.

Syntax: *files = os.listdir(path)*

Example:

List all files from "my_directory"

```
import os
```

```
files = os.listdir("my_directory/")
```

Print all the file names

```
for name in files:
```

```
    print (name)
```

```
import os
```

```
files = os.listdir("DEMOS/")
```

```
for name in files:
```

```
    print (name)
```

```
demo3.ipynb
```

```
.ipynb_checkpoints
```

```
demo2.ipynb
```

```
demo1.ipynb
```

Demonstrate File Handling Operations



Problem Statement: Write a program to demonstrate file handling operations using built-in methods.

Steps to Perform:

1. Create a text file using open method with write mode and write text to it
2. Read text from the test.txt file
3. Add more text to the test.txt file without overwriting it
4. Open the file again to read the text line-by-line

ASSISTED PRACTICE

Errors and Exceptions

Errors and Exceptions

Syntax Error: This is the most common type of error and occurs due to an incorrect syntax in the code.

Type Error: This type of error occurs due to an invalid data type of some function or operation.

```
fileObject = open('demo1/test.txt', 'r')
```

```
fileObject.read('This is a test file.')
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-3-a8c562fa1b85> in <module>
```

```
1 fileObject = open('demo1/test.txt', 'r')
```

```
2
```

```
----> 3 fileObject.read('This is a test file.')
```

```
TypeError: argument should be integer or None, not 'str'
```

```
print 'this is a demo'
```

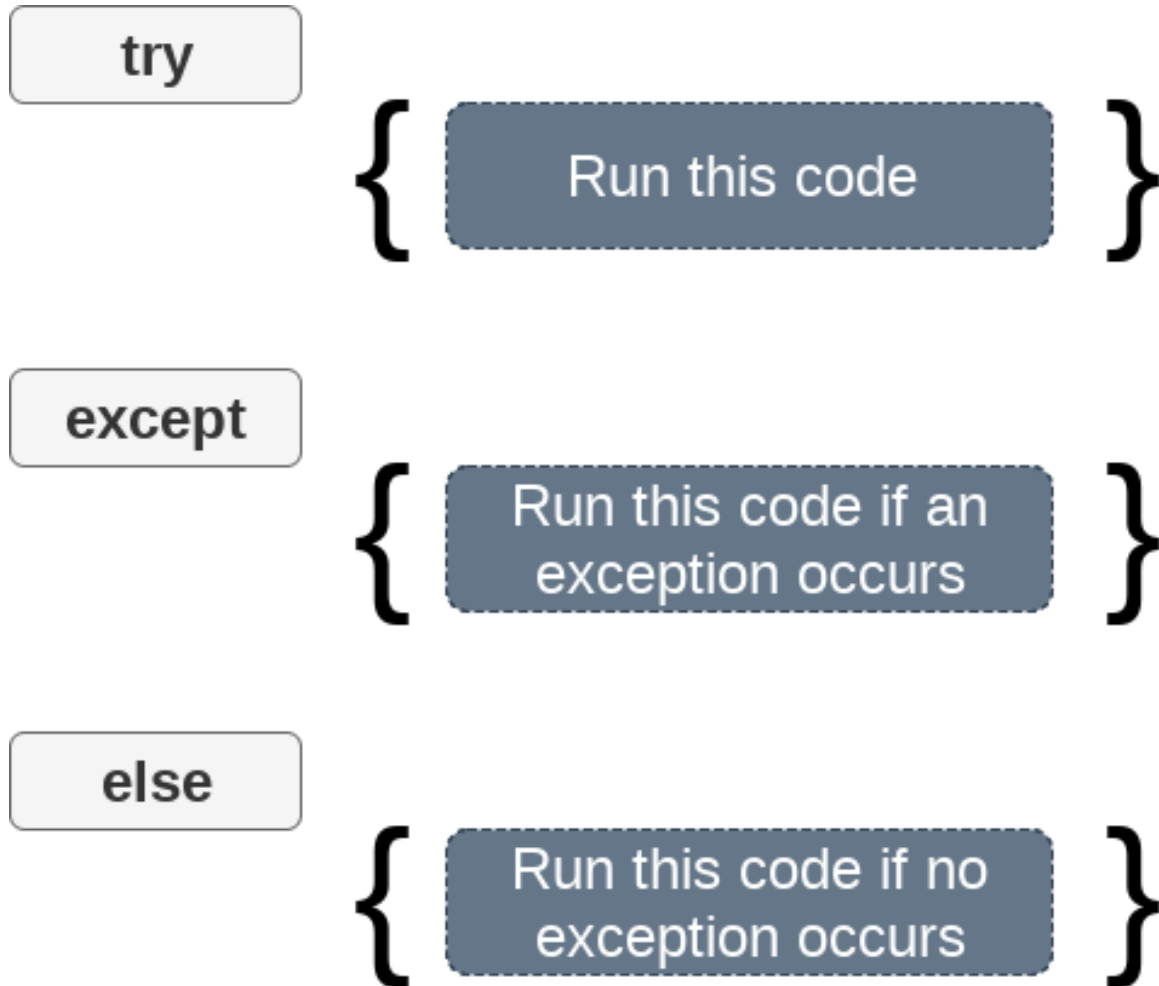
```
File "<ipython-input-4-778720b5ad18>", line 1
```

```
print 'this is a demo'
```

```
^
```

```
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('this is a demo')?
```


Errors and Exceptions



- **Exceptions** are the errors raised during the execution time and result in error messages.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Exceptions arise when the parser knows what to do with a piece of code but is unable to perform the action.

Errors and Exceptions

Exceptions come in different types and these types are printed as part of the error message.

```
10 * (1/0)
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-3-0b280f36835c> in <module>  
----> 1 10 * (1/0)  
  
ZeroDivisionError: division by zero
```

```
4 + spam*3
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-4-c98bb92cdcac> in <module>  
----> 1 4 + spam*3  
  
NameError: name 'spam' is not defined
```

The error message provides details like the type of exception, what caused it, and the line where the exception was found.



Handling Exceptions

Exception handling can be done by using *try*, *except*, and *else* blocks.

Syntax:

try:

Enter Suspicious code here

except Exception1:

If there is Exception1, then execute this block.

else:

If there is no exception then execute this block.

1. Enter the suspicious code in the *try* block.
1. Include an *except* statement.
1. Add an *else* block to execute the code when there is no exception.

Handling Exceptions

Example 1:

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

Error: can't find file or read data

Example 2:

```
try:
    file = open("testfile", "w")
    file.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")
    file.close()
```

Written content in the file successfully



Clauses in Exception Handling

except clause (no exceptions): The *except* clause can be used even if there are no exceptions to be defined. This *try-except* statement identifies all exceptions that occur but does not show the root cause of the problem.

Syntax:

try:

Code snippet

except:

If there is any exception, then execute this block.

```
try:
```

```
    print (1/0)
```

```
except:
```

```
    print ("You can't divide a number by zero.")
```

```
You can't divide a number by zero.
```

Clauses in Exception Handling

except clause (multiple exceptions): An *except* clause may handle multiple exceptions by defining the exceptions in a parenthesized tuple.

Syntax:

try:

Code snippet

.....

except (RuntimeError, TypeError, NameError):

If there is any exception, then execute this block.

```
try:
```

```
    y = int(input('Please enter a number\n'))
```

```
    print(y)
```

```
except (ValueError, TypeError) as e:
```

```
    print(type(e), '::', e)
```

```
Please enter a number
```

```
9.0
```

```
<class 'ValueError'> :: invalid literal for int() with base 10: '9.0'
```

Clauses in Exception Handling

finally Clause: A *finally* clause is optional and can be used with *try* block to define clean-up actions that must be executed under all circumstances. It is executed before leaving the *try* block, whether an exception has occurred or not.

Syntax:

try:

Code block....

Due to any exception, this may be skipped.

finally:

This would always be executed.

```
try:
    f = open("testfile", "a")
    f.write("This is my test file for exception handling!!")
finally:
    print ("Error: can't find file or read data")
```

Error: can't find file or read data

Demonstrate Exception Handling with Clauses



Problem Statement: Write a program to demonstrate various types of errors and exception handling.

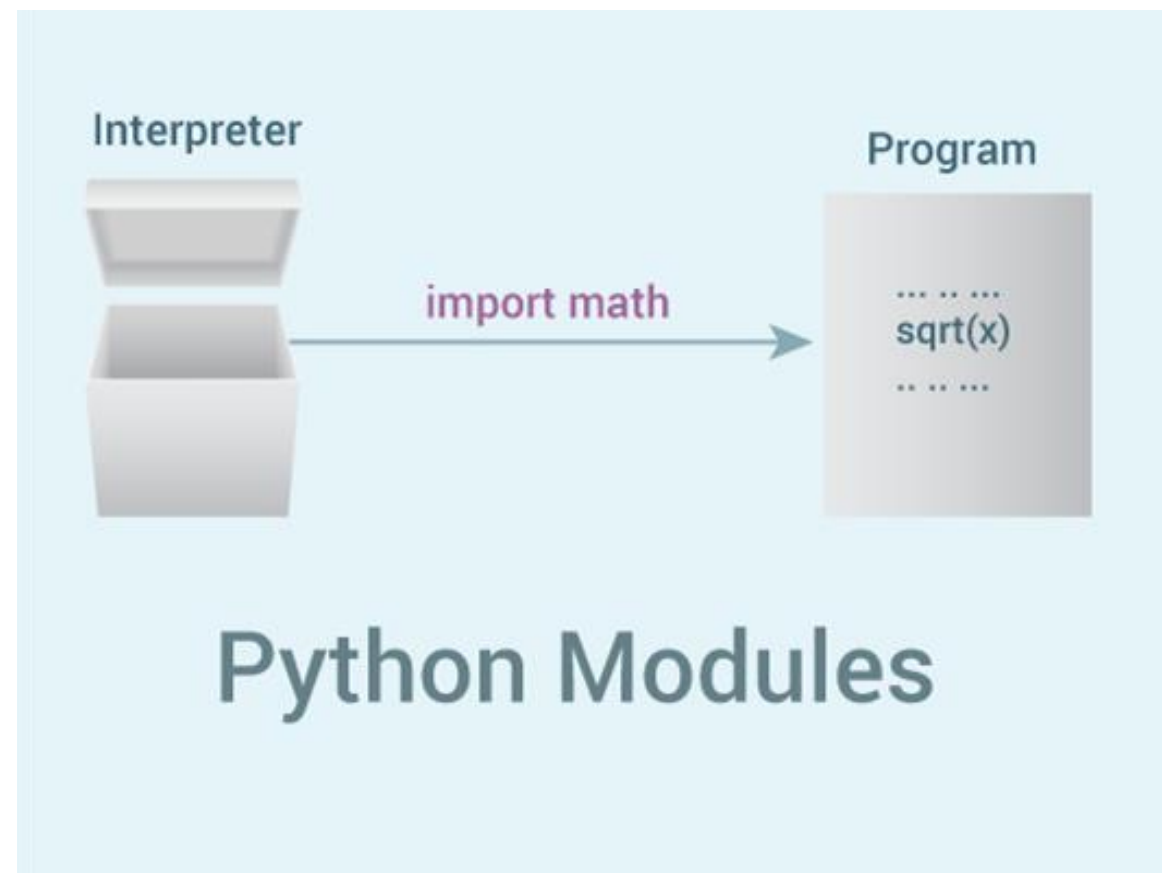
Steps to Perform:

1. Create a function to check even numbers
2. Create a list of integers
3. Print numbers from the list

ASSISTED PRACTICE

Modules and Packages

Python Modules: Overview



- A **module** is a file containing Python code that defines functions, classes, and variables.
- It also contains statements and runnable code.
- A module helps logically organize the code by grouping related code into a module.
- The file name is the module name with the suffix **.py** appended and the module's name is the value of the global variable **__name__**.

Python Modules: Overview

test.py is called a module and its module name would be **test**.

Example:

```
# Python Module test
```

```
def add(a, b):
```

```
    """Add two numbers and return the result"""
```

```
    result = a + b
```

```
    return result
```

```
# Python Module test
```

```
def add(x, y):
```

```
    """Adds two numbers and return the result"""
```

```
    sum = x + y
```

```
    return sum
```



Converting .ipynb files to .py files

Convert a default **.ipynb** file into a **.py** file using the following steps:

The screenshot illustrates the process of converting a Jupyter Notebook (.ipynb) to a Python script (.py) in JupyterLab. The interface is divided into three main panels: the left sidebar, the center Launcher, and the right Editor.

- Step 1:** A red box highlights the '+' icon in the top-left corner of the sidebar, used to create a new file.
- Step 2:** A red box highlights the 'Python 3' icon in the Launcher panel, indicating the kernel to be used for the new file.
- Step 3:** A red box highlights the 'Untitled.ipynb' file in the sidebar, which is the default notebook created.
- Step 4:** A red box highlights the 'Rename' option in the context menu that appears when right-clicking on 'Untitled.ipynb'.
- Step 5:** A red box highlights the 'Untitled.py' tab in the top-right corner, showing the new Python file has been created.
- Step 6:** A red box highlights the JSON metadata in the Editor panel, which contains the notebook's structure. A red arrow points from this box to the text 'Delete all content', indicating that this metadata should be removed to leave only the Python code.

```
{  
  "cells": [],  
  "metadata": {},  
  "nbformat": 4,  
  "nbformat_minor": 4  
}
```

Import Modules

A **module** can import other modules for their definitions and functions by executing the **import** statement.

Syntax: `import test` → Module Name

All functions of an imported module can be accessed by using the **dot (.)** operator after the module name.

Syntax: `test.add`

→ Function Name

→ Module Name → Dot (.) Operator

Import Modules

- **Import Module (Renaming):** A module can be imported by renaming it.

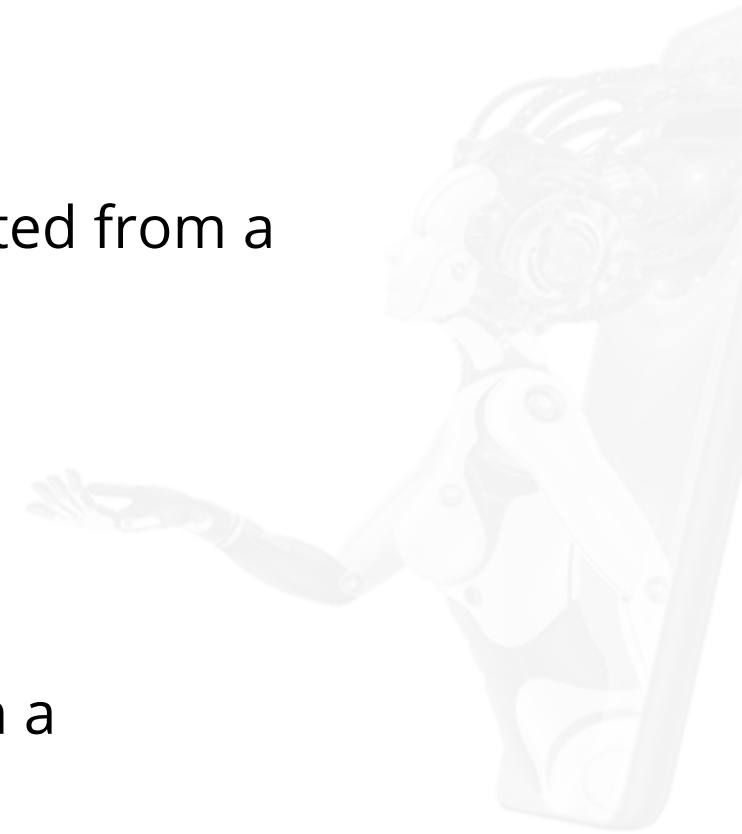
Syntax: *import module1 as m1*

- **Python from...import statement:** Specific names or methods can be imported from a module without importing the module as a whole.

Syntax: *from module1 import def1*

- **Python from...import * statement:** All names (definitions) or methods from a module can be imported using the **asterisk (*)**.

Syntax: *from module1 import **



Import Modules: Examples

Example 1:

```
import math
```

```
print("The value of mathematical constant Pi is: ", math.pi)
```

```
The value of mathematical constant Pi is:  3.141592653589793
```

Example 2:

```
import math as m
```

```
print("The value of mathematical constant e is: ", math.e)
```

```
The value of mathematical constant e is:  2.718281828459045
```

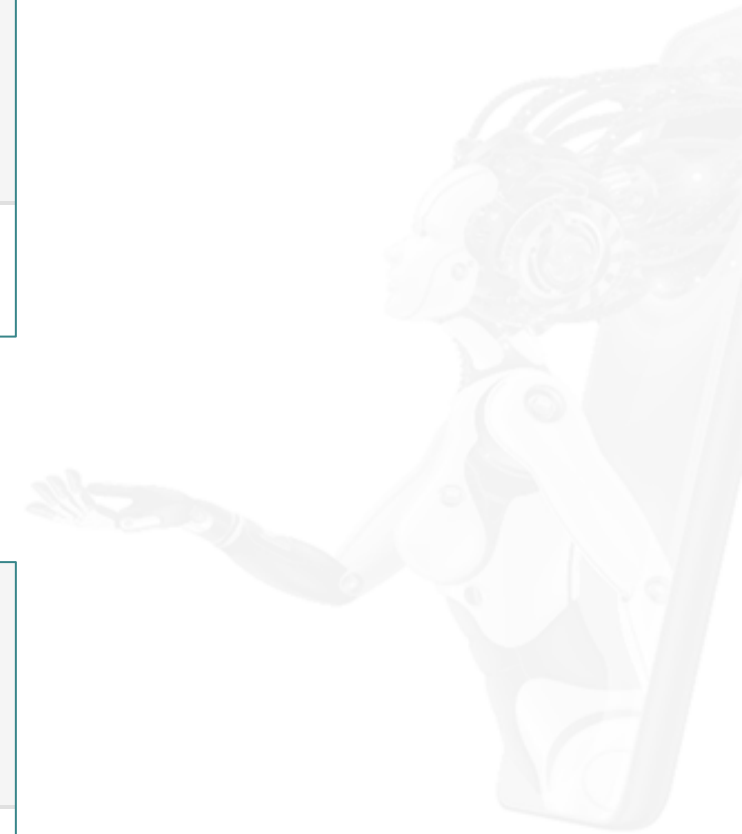

Import Modules: Examples

Example 3:

```
from math import factorial  
  
print("The factorial of 6 is: ", factorial(6))  
  
The factorial of 6 is: 720
```

Example 4:

```
from math import *  
  
print("Square root of 4 is: ", sqrt(4))  
  
Square root of 4 is: 2.0
```



The Dir() Function

The **dir()** is a built-in function of Python to find out names that are defined inside a module. It returns a sorted list of strings containing the names of all modules, variables, and functions.

Syntax: *names = dir(module1)*

Example:

```
import math

names = dir(math)

print(names)
```

['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

Python Packages



- A package is a hierarchical file structure that defines a single Python application environment consisting of modules, subpackages, and sub-subpackages.
- Packages structure Python's module namespace by using *dotted module names*.
- Similar modules of a project are placed in one package and different modules in different packages, making it easier to manage a project.
- A directory must contain a file named `__init__.py` for Python to consider it as a package.

Python Package: Example

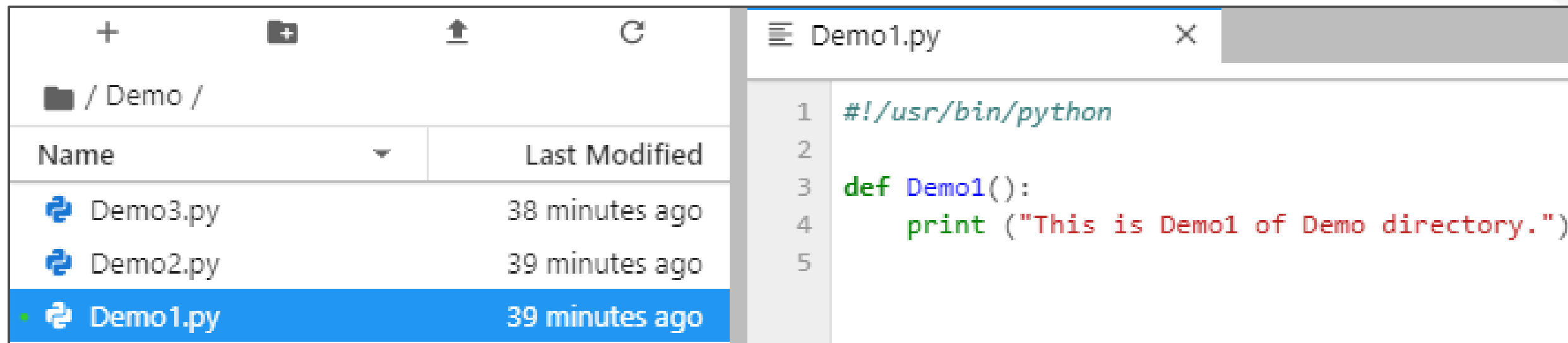
Example:

#Demo1.py is a file available in a Demo directory with the following line of source code –

```
#!/usr/bin/Python
```

```
def Demo1():  
    print ("This is Demo1 of Demo directory.")
```

#Similarly, Demo2.py has Demo2() function and Demo3.py file has Demo3() function.



The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a directory named '/ Demo /' containing three files: Demo3.py, Demo2.py, and Demo1.py. Demo1.py is selected and highlighted in blue. The code editor shows the contents of Demo1.py, which is a Python script with a shebang line and a function definition.

Name	Last Modified
Demo3.py	38 minutes ago
Demo2.py	39 minutes ago
Demo1.py	39 minutes ago

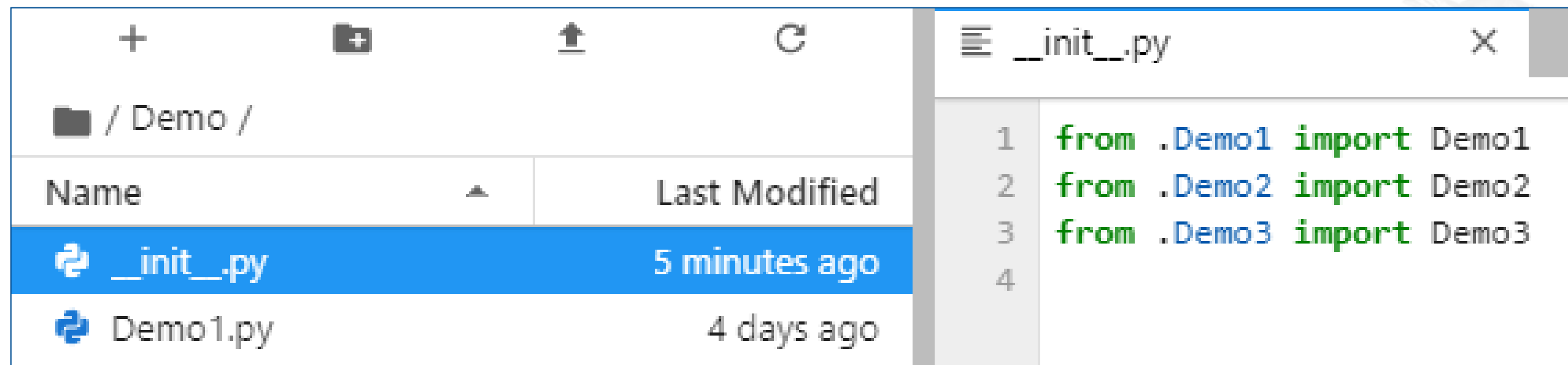
```
1  #!/usr/bin/python  
2  
3  def Demo1():  
4      print ("This is Demo1 of Demo directory.")  
5
```

Python Package: Example

#Create one more file **`__init__.py`** in Demo directory to be considered as a package and add explicit import statements in the file as follows:

```
from .Demo1 import Demo1
from .Demo2 import Demo2
from .Demo3 import Demo3
```

#Now you will have all of these classes available, when you import the Demo package.



The screenshot displays a file explorer on the left and a code editor on the right. The file explorer shows the directory structure of a Python package named 'Demo'. It contains a file named `__init__.py` which was modified '5 minutes ago' and a file named `Demo1.py` which was modified '4 days ago'. The code editor shows the content of `__init__.py`, which contains three import statements: `from .Demo1 import Demo1`, `from .Demo2 import Demo2`, and `from .Demo3 import Demo3`.

Name	Last Modified
<code>__init__.py</code>	5 minutes ago
<code>Demo1.py</code>	4 days ago

```
1 from .Demo1 import Demo1
2 from .Demo2 import Demo2
3 from .Demo3 import Demo3
4
```

Python Package: Example

#Now import the *Demo* package and check the result

```
import Demo
```

```
Demo.Demo1()
```

```
Demo.Demo2()
```

```
Demo.Demo3()
```

```
import Demo
```

```
Demo.Demo1()
```

```
Demo.Demo2()
```

```
Demo.Demo3()
```

```
This is Demo1 of Demo directory.  
This is Demo2 of Demo directory.  
This is Demo3 of Demo directory.
```


Module and Package Handling



Problem Statement: Write a program to demonstrate package handling by creating a package and importing it to use the package functions.

Steps to Perform:

1. Create a folder named *Packages*
2. Create a file *Pkg1.py* in Packages directory with the source code for addition function
3. Create another file *Pkg2.py* in Packages directory with the source code for multiplication function
4. Create one more file *__init__.py* in Packages directory to be considered as a package and add explicit import statements in the file
5. Create a demo file outside the Packages directory and add import statement and package functions

ASSISTED PRACTICE

DATA AND ARTIFICIAL INTELLIGENCE



Knowledge Check

Knowledge Check

1

What is the use of the `readline()` method?

- a. Reads individual lines from a file opened in *reading* `r` mode
- b. Reads all the lines from a file opened in *reading* `r` mode
- c. Reads individual lines from a file opened in *writing* `w` mode
- d. None of the above



Knowledge Check

1

What is the use of the `readline()` method?

- a. Reads individual lines from a file opened in *reading* `r` mode
- b. Reads all the lines from a file opened in *reading* `r` mode
- c. Reads individual lines from a file opened in *writing* `w` mode
- d. None of the above



The correct answer is **a**

The `readline()` method is used to read individual lines from a file opened in *reading* `r` mode.

Knowledge Check

2

SyntaxError and TypeError are the only types of errors in Python.

- a. True
- b. False



Knowledge
Check

2

SyntaxError and TypeError are the only types of errors in Python.

- a. True
- b. False



The correct answer is **b**

There are various types of errors in Python such as IOError, IndentationError, ValueError, and RuntimeError. The most common errors are SyntaxError and TypeError.

Knowledge Check

3

Which of the following statements is true about importing a module?

- a. A module can be imported by renaming it.
- b. Specific names or methods can be imported from a module.
- c. A module can be imported as a whole.
- d. All of the above



Knowledge Check

3

Which of the following statements is true about importing a module?

- a. A module can be imported by renaming it.
- b. Specific names or methods can be imported from a module.
- c. A module can be imported as a whole.
- d. All of the above



The correct answer is **d**

A module can either be imported as a whole or some specific names can be imported from the module. A module can also be imported by renaming it.

Key Takeaways

- *open()*, *write()*, *read()*, and *close()* are the major file handling operations.
- Directories are a part of a hierarchical file system to store folders and files in an organized tree structure.
- Exceptions are errors that occur during code execution and can be handled by *try-except-finally* blocks.
- A module is a file containing a Python code that defines functions, classes, and variables.
- A package is a hierarchical file directory structure consisting of modules, subpackages, and sub-subpackages.



Student Data Handling

Duration: 30 mins

Problem Statement:

Create a module that can be imported and used by all departments of a school to read, write, and modify student files. Use exception handling in your programming logic to handle IOErrors.

