Chapter – 8

Arrays and Strings

Arrays :-

Array are a collection of items stored at contiguous memory location. In python arrays are available through the 'array' module though lists are more commonly used. However for numerical operations the 'numpy' library provides a more powerful array object.

Features of Arrays :—

- Fixed size :— The size of an array is defined at the time of creation.
- Homogeneous elements :— Arrays typically store elements of the same type.
- Efficient memory usage :— Arrays use less memory and are more efficient for numerical operations compared to lists.

Creating and Initialization Arrays :—

using the 'array' module

```
import array
# creating an integer array
int_array = array.array('i', [1,2,3,4])
```

print(int_array)

using the 'numpy' library

```
Import numpy as np
# Creating a numpy array
np_array = np.array([1,2,3,4])
print(np_array)
```

Indexing and Transversal:-

```
# Accessing elements.
print(int_array[0])
print(int_array[2])

# Transversing an array
for elements in int_array:
    print(element)
```

* Manipulation:-

```
# Adding element using array
int_array.append(5)
print(int_array)

# Remove element using array
int_array.remove(2)
print(int_array)

# Adding elements (using numpy)
np_array = np.append(np_array, [5,6])
print(np_array)
```

Strings

Strings in python are sequence of character. They are immutable, meaning once a string is created, it cannot be changed.

Creating And Assignment Strings.

```
# Creating a String
my_string = "Hello, world!"
print(my_string)

# Assigning a String
another_string = 'python is fun'
print(another_string)
```

Indexing and Slicing

```
# Accessing character by index
print(my_string[0])

# Slicing String.
print(my_string[0:5])
print(my_string[7:])
print(my_string[::-1])
```

Built in function for strings:-

# len() - returns the length of the string
print(len(my_string))

# Str.upper() - convert all character to uppercase
print(my_string.upper())

# Str.lower() - convert all character ~~with~~ to lowercase
print(my_string.lower())

# Str.replace → replace a substring with another
print(my_string.replace('world', 'python'))

# Str.split() - split the string into a list
print(my_string.split(','))

# Str.strip() → remove leading and trailing whitespace.
whitespace_string = " Hello, world! "
print(whitespace_string.strip())

# Str.find() - returns the index of the first occurence
print(my_string.find('world'))

# Str.join() - join elements of a list into single string.
list_of_string = ['python', 'is', 'fun']
print(''.join(list_of_string))

Summary

- Arrays are fixed size, homogeneous collections of elements.
- Can be created using the 'array' module or 'numpy' library.
- Support efficient indexing, traversal and manipulation.
- Strings are immutable sequence of characters.
- Can be created and assigned using quotes.
- Support indexing, slicing and variety of built-in function for manipulation and transformation.

Chapter - 9
Functions

Functions are reusable blocks of code that perform a specific task. They help in organising code, reducing redundancy, and improving readability and maintainability.

Need for functions:-

- Reusability:- write once, use many times.
- Modularity:- Breakdown complex problem into simple pieces.
- Maintainability:- Easier to update and manage code.
- Readability:- Improve the clarity of the code.

Types of functions:-

1. Built - in functions:- functions provided by python. (eg:- print(), len(), range())
2. User - defined function:- function defined by user to perform specific task.
3. Anonymous functions:- functions defined using the 'lambda' keywords.

Defining and calling functions

Defining a function

```
def function_name(parameters):
    """Docstring (optional): Describe the function"""
    # function body
    return value  # optional
```

Ex-

```
def greet(name):
    """Greet a person by name"""
    return f"Hello, {name}!"
print(greet("Alice"))
```

Calling a function

```
result = function_name(arguments)
```

Ex-

```
print(greet("Sagar"))
```

* Function Arguments.

1. Positional Arguments:— Arguments passed in a specific order.
2. Keyword Arguments:— Argument passed by explicitly naming the parameter.

3. Default argument :- parameter with default value.

4. Arbitary Arguments :- Accepts variable number of argument using '*args" and '**kwargs'.

Example

```
def example_func(a, b, c=3, *args, **kwargs):
    print(a,b,c)
    print(args)
    print(kwargs)

example_func(1,2)
example_func(1,2,4,5,6, e=7, f=8)
```

~~Example :-~~

~~def example_func(a,b,c=3 *args*~~

Return and yield :-

- Return :- Exits the function and optionally passes back a value to the caller.

```
def add(a,b):
    return(a+b)
print(add(2,3))
```

• yield :- used in generators to return a value and pause the function, maintaining its state.

```
def generate_numbers():
    for i in range(5):
        yield i

gen = generate_numbers()
print(next(gen))
print(next(gen))
```

* 'None' Keyword
None is used to signify the absence of a value.

```
def do_nothing():
    pass

result = do_nothing()
print(result)
```

* Scope of variables:

• Local Scope :- variable defined within a function.

• Global Scope :- variable defined outside all function.

• Non local Scope :- variable defined in the nearest

enclosing scope (excluding global scope)

Example :-

```
x = "global"
def outer ()
    x = "outer"
    def inner ():
        nonlocal x
        x = "inner"
        print ("Inner:", x
    inner()
    print ("outer:", x)
outer()
print ("Global:", x)
```

* Recursion

A function calling itself to solve a smaller instance of the same problem.

```
def factorial(n):
    if n == 1
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
```

* Anonymous functions:-
defined using the 'lambda' keyword.

square = lambda x: x**2
print(square(5))

# lambda function with map
nums = [1,2,3,4]
squared_nums = list(map(lambda x: x**2, nums))
print(squared_nums)

* ~~Examples:-~~

Summary:-

- Function are reusable blocks of code that perform specified task.
- Arguments can be positional, keyword, default or arbitary
- Return exits the function and returns a value, while yield is used in generators.
- Scope defines the visibility of variables (local, global, nonlocal)
- Recursion involves a function calling itself.
- Anonymous function are defined using the 'lambda' keywords.

Python module is a file that contain built in function classes, its and variable and

Ch-10

Modules and packages

Why modules?

* Modules are used to break down large programs into smaller, manageable and organised files.

They help in:

- Code Reusablity:— Functions and classes can be reused in different Programs.
- Maintainability: Easier to update and manage code.
- Namespace management:- Avoids name conflicts by encapsulating function, classes and variables.
- Code Organization:— makes the codebase more structured and easier to navigate.

* module creation:—
A module creation a python file ('.py') that contains function, classes, and variables.

Example file-name - mymodule

→ (1) def greet(name):
    return f"Hello, {name}!"

→ usage — import mymodule in print(mymodule.greet("Alice"))

(11) def add(a,b):
    return a+b.

Importing modules:-
You can import module using 'import' statement.

```
import mymodule
print (mymodule.greet ("Alice"))
print (mymodule.add (2,3))
```

* you can also import specific functions or variable from a module:-

```
from mymodule import greet, add
print (greet ("bob"))
print (add (4,5))
```

Or import all names from a module. (not recommended) due to potential name conflicts.

```
from mymodule import *
print (greet ("Satyam"))
print (add (6,7))
```

* module Namespace:
Each module has its own namespace, which helps in avoiding name conflicts.

1. Normal import — import module_name
import math (Print math.sqrt(25)) //5
Adv - clear where each function comes from.
- Prevent name conflicts

2. From import
from module_name import specific_function_or_class
(only import specific() we can use directly

from math import sqrt
print(sqrt(36))
//no need to write sqrt

3. from import *
Syn - from module_name import *
from math import *
~~print(sqrt(49))~~
print(factorial(5))

```
# mymodule.py
x = 10
def show()
    print(x)
# another_module.py
x = 20
def display():
    print(x)
# main.py
import mymodule
import another_module
mymodule.show()
another_module.display()
```

* A namespace is a container where names are mapped to objects. In python, each module creates its own namespace, which means that variables function and classes defined in one module do not interface with these defined in another module. This helps avoid name conflicts, especially when multiple modules are imported into a single script.

* Package : Basics
A package is a collection of related modules stored in a directory that includes a special '__init__.py' file.

* module path :- when we import a module python searches for it in a specific order, using a list called sys.path

Search path :-
1. current directory (where script run)
2. Environment variable PYTHONPATH (if set)
3. standard library directories
4. site package (installed third party)

Creating a package

Directory Structure :-
mypackage /
__init__.py
module_1.py
module_2.py

* view module search path

```
import sys
for path in sys.path:
    print(path)
```

o/p → /home/user/myprojects directory
/usr/lib/python3.11 Standard Library
/usr/lib/Python 3.11/site-Pack

'__init__.py' can be an empty file or can execute initialization code for the package.

Examples :

```
# module_1.py
def func1():
    return "This is function 1"

# module_2.py
def func2():
    return "This is function 2"
# __init__.py
from .module1 import func1
from .module2 import func2
```

* Importing a package :-

```
import mypackage
print(mypackage.func1())
print(mypackage.func2())
```

Permanently set -
Linux
export PYTHONPATH="/path/to/my/module:$PYTHONPATH"
win set PYTHONPATH=C:/path/to/my/module

DATE
PAGE NO

path setting :- (Temporarily)
To import a module from a different directory you can modify the 'sys.path' list.

import sys
sys.path.append('/path/to/your/module')

import mymodule.

ages
(Third party)

* Commonly used Modules.

• math module :-
provide mathematical functions.

import math
print(math.sqrt(16))
print(math.pi)
print(math.sin(math.radians(90)))

* Random module :- provide functions to generate random numbers.
import random
print(random.random())
print(random.randint(1,10))
print(random.choice(['apple','banana','cherry']))

Emoji module:-
provides functions to work with emojis, you need to install it using 'pip install emoji'.

```
import emoji
print emoji.emojize ("python is fun: thumbs_up:")
print emoji.demojize ("python is fun fun 👍")
```

Summary

- Modules: Help organise code into manageable and reusable files.
- Create a module by saving code in a '.py' file.
- Import modules using 'import' or 'from module' import'.
- Each module has its own namespace.

* packages: Collection of related modules.

- Creates a package by organising modules in a directory with an '__init__.py' file.
- import package to use the functionalities of contained modules.
- Common modules:
- math: provides mathematical function and constant.
- Random: provides random number generation.
- Emoji provides functions to handle emojis in string.

Ch-11

Numpy and Pandas

Numpy (Numerical python) is a powerful library for numerical computing in python. It provides support for large multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Numpy is the foundation for many scientific computing libraries in python, including scipy, pandas, and more!

- Installation:- ~~You~~ can install Numpy using pip!

```
pip install numpy
```

Numpy Arithmetic functions:-
Numpy provides a variety of arithmetic operations that can be applied element-wise to arrays:

```
import numpy as np
# Creating arrays
a = np.array([1,2,3,4])
# Sum
print(np.sum(e))
# min and max
print(np.min(e))
print(np.max(e))
```

* pandas:

- pandas is a powerful, fast and easy to use open source data analysis and data manipulation library built on top of Numpy. It provides data structure like Series and dataframe that are essential for data science and machine learning tasks.

- Installation :-

you can install pandas using pip.

```
pip install pandas
```

Series:

A Series is a one-dimensional labeled array capable of holding any data type.

```
import pandas as pd
import numpy as np

# creating series
s = pd.Series([1,3,5,np.nan,6,8])
print(s)
```

* Dataframe

A data frame is a two dimensional labeled data structure with columns of potentially different types.

```
# Creating a Data frame
data = {
    'A' = [1,2,3,4],
    'B' = [5,6,7,8],
    'C' = [9,10,11,12]
}
df = Pd. Dataframe (data)
print (df)
```

Creating Dataframes :-

You can create dataframes from various data structure like lists, dictionaries, and arrays.

```
# from list of lists
data = [[1,2], [3,4], [5,6]]
df1 = Pd. dataframe (data, columns = ['column1',
    'column2'])
print (df1)
# from dictionary
data = {'column 1': [1,2,3], 'column2': [4,5,6]}
df2 = pd. Dataframe (data)
print (df2)

# from numpy array
data = np. array ([[1,2,3], [4,5,6]])
df3 = Pd. Dataframe (data, columns = ['A', 'B', 'C'])
print (df3)
```

Formatting Data:
You can format dataframes by setting index, renaming columns, and applying functions.

```
# Setting an index
df.set_index('A', inplace = True
print(df)
# Renaming columns
df.rename(columns={'B':'column 2', 'C':'column 3'},
          inplace=True)
print(df)

# Applying functions
df['column 3'] = df['column 3'].apply(lambda x:
    x*2)
print(df)
```

* Fundamental dataframe operations:
Basic operations on dataframe include selection, filtering, grouping, merging and aggregation.

```
# Selecting a column
print(df['column 2'])

# Filtering rows
Print(df[df['column 2']>5])
```

# Grouping
grouped = df.groupby('column2').sum()
print(grouped)

# merging
df1 = Pd.Dataframe({'key': ['A','B','C'], value1: [1,2,3]})
df2 = Pd.Dataframe({'key': ['A','B','C'], 'value2': [4,5,6]})
merged = Pd.merge(df1, df2, on='key', how='inner')
print(merged)

Aggregation:

print(df.agg({'column2': ['sum','mean'], 'column3': ['min','max']})

Summary

- Numpy
- provides powerful tools for numerical computing with support for multinationsional arrays and various mathematical function.
- Key function include arithmetic operations, array manipulation and statistical calculations.

- pandas
- essential for data manipulation and analysis, offering structure like Series and dataframe.
- Support data creation, formatting and fundamentals operation such as selection, filtering, grouping, merging etc

Ch-12

Files

Files are used to store data permanently. In python, file operations allow you to read from, write to and manipulate files. Python provides built-in function and methods to handle files efficiently.

Features:

- persistence:- Data stored in files persist beyond the termination of the program.
- Large data handling:- Suitable for storing large amount of data that do not fit into the memory.
- Easy sharing:- Files can be easily shared b/w programs or systems.

csv : Comma Seperated values
JSON : Javascript object Notation
PDF : portable Document Format

* File operations :-

1. opening file:- using the open() function.

2. closing files: using the close() method.

3. Reading from files: using methods like read(), readline(), readlines()

4. writing to files:- using methods like write(), writelines().

5. file methods:- various methods for file manipulation.

opening files

To open a file, use the open() function. It requires at least one argument, the filename, and has optional mode and encoding arguments.

```
file = open('example.txt', mode='r', encoding='utf-8')
```

Common modes include:

- "r" : Read (default mode)
- "w" : write (create a new files or truncate an existing file)
- "a" : Append (writes to the end of the file)

" 'b' ": Binary mode/used with other modes for binary files, eg - 'rb', 'wb')

" 't' ": Read and write (eg 'rt', 'wt')

Closing files:

Always close file after use to free up system resources.

file.close()

Alternating use of 'with' statement to handle files, which automatically closing the files:

with open 'example.txt', mode='r', encoding='utf-8' as file:
    content = file.read()

* Writing to files:

use write() or writelines to write data to a file.

with open('example.txt', mode='w', encoding='utf-8') as file:
    file.write("Hello world!\n")

lines = ['line 1\n', 'line 2\n', 'line 3\n']
with open('example.txt', mode='a', encoding='utf-8') as file:
    file.writelines(lines)

Reading from files:-

use read(), readline() or readlines() to read data from files.

```
# Reading the entire file
with open('example.txt', mode='r', encoding='utf-8') as file:
    content = file.read()
    print(content)

# Reading line by line
with open('example.txt', mode='r', encoding='utf-8') as file:
    line = file.readline()
    while line:
        print(line.strip())
        line = file.readline()

# Reading all line in list

with open('example.txt', mode='r', encoding='utf-8') as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())
```

ile:

* File Methods:

Common file methods include:

- read(size): Reads up to size bytes from the files.
- readline(size): Reads a single line from the file.

- readlines() : Read all lines into list.
- write(String) : write a String to the file.
- writelines(list) : write a list of strings to the file.
- seek(offset, whence) : moves the file pointer to a specific position.
- tell() : Returns the current positions of the file pointer.

Working with files using Dataframes (pandas):

pandas make it easy to read from and write to various file format including CSV, EXCEL, JSON and more.

Reading file into Dataframe:

```
import pandas as pd

# Reading as csv file
def = pd.read_csv('data.csv')
print( df.read())
# Reading an Excel file
    df = Pd.read_excel('data.xlsx', sheet_name='
        Sheet1')
```

print(df.head())

# Reading a JSON file
df = pd.read_json('data.json')
print(df.read())

* Writing Dataframes to files

writing to a CSV file

df.to_csv('output.csv', index = false)

# writing to an excel file
df.to_excel('output.xlsx', sheetname = 'Sheet1', index = false)

# writing to a JSON file
df.to_JSON('output.json, orient = 'records')

Summary
- Files are essential for storing and managing data persistently.
- File operation include opening, ~~reading~~ reading, writing, and closing file.
- pandas provide convenient methods to handle file operations with dataframe, making it easier to work with structured data in various format.

Ch-13

Errors and Exception Handling:

Errors:
Errors in python can be broadly classified into two categories:

1. Syntax Errors: Occur when the parser encounter a syntax error.

2. Exception: Error detected during execution that are not unconditionally fatal.

Example of Syntax error:

```
print("Hello World"        # missing the closing parenthesis.
```

Example of an exception:

```
X = 10/0                   # Division by Zero.
```

* Exceptions:
Exceptions are raised when the interpreter encounters an error during execution. Python provides several built-in exception, and you can also define your own.

try:
result = 10/0
except ZeroDivisionError:
print("...")

num = int("abc")
except ValueError:
print

num = [1,2,3]
except IndexError:
print(num[5])
except IndexError:
print("...")

Built in Exception:

Some common built in exceptions include:

- IndexError: Raised when a sequence subscript is out of range.
- KeyError: Raised when a dictionary key is not found.
- ValueError: Raised when a function receives an argument of the right type but inappropriate value.
- TypeError: Raised when an appropriate is applied to an object of inappropriate type.
- ZeroDivisionError: Raised when the second argument of a division or module operation is zero.
- FileNotFoundError: Raised when an attempt to open a file that does not exist fails.

Example:

```
try:
    lst = [1,2,3]
    print(lst[5])
except IndexError as e:
    print(f"I caught an IndexError: {e}")
```

User-defined Exception

You can create custom exception by defining a new class that inherit from the exception base class.

Example:

```
class customError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)
try:
    raise customError("This is a custom error")
except customError as e:
    print(f"Caught a custom error: {e}")
```

Catching Exception

To handle exception, use the "try---except" block you can also use "else" and "finally" for additional control.

"try" Block: The code that might raise an exception is placed inside the try block.

"except" Block: If an exception occurs, the code inside the "except" block runs, handling the error.

"else" Block:: Runs if no exception are raised.

"finally" Block: Always runs, regardless of whether an exception occurred.

Example

```
try:
    x = int(input("enter a number"))
    y = 10/x
except zerodivisionError as e:
        print(f"error: division by zero is not allowed {e}")
except value Error as e:
        print(f"Error: Invalid input. {e}")
else:
    print(f"Result: {y}")
Finally:
    print("This block is always executed")
```

* Raising Exception

To raise exceptions, use the 'raise' statement,

ex-

```
def check_positive(number):
    if number <= 0:
        raise ValueError("Number must be +ve")
    return number
try:
    check_positive(-10)
except value Error as e:
    print(f"exception raised: {e}")
```

Summary

- Errors can be syntax error or exceptions.
- Exceptions are runtime errors that can be handled by "try ... except" blocks.
- Built-in exceptions are provided by python for common error cases.
- user-defined exceptions allow for custom error handling.
- Catching exceptions involve using "try", "except", "else" and "finally" blocks.
- Raising exceptions is done using the "raise" statement to indicate that an error has occured.

proper error and exception handling ensures that your programs can handle unexpected situations gracefully and continue functioning or fail safely.

Vinay Kumar
16/08/24