```
"""
- File handling
- List Comprehension
- Lambda Functions
- Higher Order Functions
- Decorators
"""

# imports
import time
import math


### File Handling

"""
Modes:
"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist (Will overwrite entire file)

"a" - Append - Opens a file for appending, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists
"""

# Open File
f = open("file.txt", "r")


# Read File
print(f.read())


# Read file again
print(f.read())


# Go to starting of file
f.seek(0)
print(f.read())


# Read only one line at a time
f.seek(0)
print(f.readline())
```

```
# Read first few bytes
f.seek(0)
print(f.read(4))


# Read only first few bytes of every line
f.seek(0)
for line in f:
    print(line[:2])


# Read all line
f.seek(0)
print(f.readlines())


# Close File
f.close()
"""
Operating systems limit the number of open files any single process can have.

Too many open files can slow down program.

When writing to a file, often the contents are cached in memory and written to file
only when file is closed.
"""



# Alternate way to handle files using with
"""
The with statement allows a series of statements to execute inside a runtime context
that is controlled by an object serving as a context manager.

You dont need to explicitly close file when using with.
"""

# Read file
with open("file.txt", "r") as file:
    print(file.readlines())


# Write to file
with open("newfile.txt", "w") as file:
    file.write("Hello World!\n")


# Write multiple lines
lines = ["Welcome\n", "To\n", "IIIT-H\n"]
```

```
with open("newfile.txt", "a") as file:
    file.writelines(lines)


# Read a csv file
headers = None
data = []
with open("student.csv", "r") as csv:
    headers = csv.readline().split(",")
    for line in csv:
        data.append(line.split(","))

print(headers)
for row in data:
    print(row)



### List Comprehension

"""
result = []
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                    for itemN in iterableN:
                        if conditionN:
                            result.append(expression)


[ expression for item1 in iterable1 if condition1
             for item2 in iterable2 if condition2
             .
             .
             .
             for itemN in iterableN if conditionN ]
"""

# Traditional Python
a = [1, 2, 3, 4, 5]
b = []
for element in a:
    b.append(element ** 2)
print(b)

# LC
b = [element ** 2 for element in a]
```

```
print(b)

# Map
b = list(map(lambda x: x*x, a))
print(b)

# Convert to str using map
b = list(map(str, a))
print(b)

# LC with condition
c = [element ** 2 for element in a if element % 2 == 0]
print(c)

# filter
c = list(filter(lambda x: x % 2 == 0, map(lambda x: x*x, a)))
print(c)


# Extract column from csv data
# Traditional python
marks = []
for row in data:
    marks.append(int(row[3]))
print(marks)

# LC
marks = [int(row[3]) for row in data]
print(marks)


# Extract names of all stocks
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'MSFT', 'shares': 50, 'price': 45.67},
    {'name': 'HPE', 'shares': 75, 'price': 34.51},
    {'name': 'CAT', 'shares': 60, 'price': 67.89},
    {'name': 'IBM', 'shares': 200, 'price': 95.25}
]
names = [elem["name"] for elem in portfolio]
print(names)


# Flatten nested list
nest = [[1, 2, 3], [4, 5, 6, 7, 8], [9, 10]]
flat = [elem for row in nest for elem in row]
print(flat)
```

```
### Lambda Functions

"""
Short (probably one-liner) functions.

Syntax:
    lambda args: expression

args is a comma-separated list of arguments
expression is evaluated and returned
"""

# A simple function with one argument
def fun(x):
    return x + 1

print(fun(4))

# Lambda function
fun = lambda x: x + 1
print(fun(6))

# A function with two argument
def fun(x, y):
    return x + y

print(fun(4, 6))

# Lambda Function
fun = lambda x, y: x + y
print(fun(2, 0))


# Yet another funtion
def full_name(first, last):
    first = first.title()
    last = last.title()
    result = "Full Name: " + first + " " + last
    return result

print(full_name("karan", "bhatt"))

# Lambda Function
full_name = lambda first, last: "Full Name: " + first.title() + " " + last.title()
print(full_name("karan", "bhatt"))


# mean of some ints
```

```
mean = lambda data: sum(data) / len(data)
print(mean(marks))


# As a custom comparator
ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
print(sorted(ids)) # Lexicographic sort

sorted_ids = sorted(ids, key=lambda x: int(x[2:]))  # Integer sort
print(sorted_ids)



### Higher Order Functions

"""
Functions are said to be first-class objects,
meaning there is no difference between how you might handle a function
and any other kind of data.

Functions can be passed as arguments to other functions.
Function can return function.
Function can be assigned to a variable.
"""

# A simple function assigned to a variable
def fun(x):
    return x + 1

v = fun
print(v(1))


# Pass function as an argument to a function
def after(seconds, func):
    time.sleep(seconds)
    func()


def greeting():
    print('Hello World')


after(3, greeting)


# Yet another example
def apply_function_to_all(func, data):
    res = []
```

```
    for elem in data:
        res.append(func(elem))
    return res

print(apply_function_to_all(len, ["Welcome", "To", "IIIT-H"]))


# Just one more to go
# odd, even, filter
odd = lambda x: x % 2 == 1
even = lambda x: x % 2 == 0


def filter(func, data):
    res = []
    for i in data:
        if func(i):
            res.append(i)
    return res

a = list(range(20))
b = filter(odd, a)
print(b)
c = filter(even, a)
print(c)


# Lets store functions in a list
def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda: print('Hello', name))
    return funcs

a, b, c = make_greetings(['Person1', 'Person2', 'Person3'])
a()
b()
c()


# Pass arguments to the function passed as an argument
def add(x, y):
    print(x + y)

# after(3, add(10, 20))     # add() gets called immediately

# Solution --> pass argument to outer function
def after(seconds, func, x, y):
    time.sleep(seconds)
```

```
    func(x, y)

after(1, add, 10, 20)


# What function to call? Lets (apply some AI/ML model and then) decide!
def add_two_nums(a, b):
    return a+b


def add_three_nums(a, b, c):
    return a+b+c


def decide(nums_length):
    if nums_length == 2:
        return add_two_nums
    else:
        return add_three_nums


nums = [1,2,3]
result_func = decide(len(nums))
print(result_func(*nums))



### Decorators

"""
Definition: A decorator is a function that creates a wrapper around another function.

In simple words, it is just an extension to an already
existing function (but without modifying the function's code)

Syntax:
    @decorate
    def fun(x):
        pass
"""

# Example
def trace(func):
    def call(*args):
        print('Calling', func.__name__)
        return func(*args)
    return call

@trace
```

```
def square(x):
    return x ** 2

print(square(4))


# A decorator to time every function
def calculate_time(func):
    def wrap(*args, **kwargs):
        start = time.time()

        func(*args, **kwargs)

        end = time.time()
        print("Total time taken: ", end - start)

    return wrap


@calculate_time
def factorial(num):
    print(math.factorial(num))


factorial(10)


# Decorator chaining
def logging(func):
    def wrap(*args, **kwargs):
        print("Checkpoint 1")

        func(*args, **kwargs)

        print("Checkpoint 2")

    return wrap

@logging
@calculate_time
def exponent(b, e):
    print(b ** e)

exponent(1000, 50)

"""
# Author: Karan Bhatt (kbbhatt04)
# Date: 28/10/2023
"""
```