Python Refinement Course

LEVEL OF THIS COURSE: INTERMEDIATE TO ADVANCED

This Refinement Course will focus on enhancing your practical skills through advanced exercises and projects, covering areas such as algorithms, data structures, and more complex programming concepts. The course will be structured into modules, each focusing on a specific topic.

RECAP OF THIS COURSE

1. Advanced Data Structures
- Deep dive into lists, dictionaries, sets, and tuples.
- Introduction to collections module - Counter, defaultdict, OrderedDict.

2. Algorithms
- Sorting and Searching algorithms.
- Implementing basic algorithms from scratch.

3. File Operations and Exception Handling
- Advanced file handling techniques.
- Mastering try, except, else, finally blocks.

4. Functional Programming
- Lambda functions, map, filter, and reduce.
- Understanding closures and decorators.

5. Object-Oriented Programming
- Advanced concepts: Polymorphism, Abstraction.
- Design patterns and best practices.

6. Modules and Packages
- Creating and using modules.
- Understanding Python packages and pip.

7. Project Work
- Application of learned concepts in a small project.
- Code reviews and optimization techniques.

COURSE FORMAT
- Theoretical reading and practical exercises.
- Regular Check-ins to discuss progress, clarify doubts, and provide feedback.
- Project Phase: Implementing a project towards the end of the course.

1. ADVANCED DATA STRUCTURE

Collections Module

1. Study Material:

- Read about the `collections` module in Python.
- Focus on `Counter`, `defaultdict`, and `OrderedDict`.

2. Practical Tasks:

- Task 1: Create a `Counter` to count the occurrences of each character in a given string.
- Task 2: Use a `defaultdict` to group words in a list by their first letter.
- Task 3: Implement an `OrderedDict` to maintain the order of items added in a dictionary.

Guidelines:

- Try writing the code for these tasks yourself.
- Test your code with different inputs.
- Focus on understanding how these data structures can simplify certain tasks.

STUDY MATERIAL

In Python, "Collection Modules" typically refer to built-in modules that provide various data structures for organizing and managing collections of data.
These modules include some “Collectors”:
More info here: https://docs.python.org/3/tutorial/introduction.html

1. BUILD-IN COLLECTORS

The following are used to “collect” data and they have a different use in Python.
They are build-in Python collectors and you can use them without importing them.
Those include:

1.1. LIST

- Lists are ordered, mutable sequences that can hold a variety of data types.

Creation - Change Values - Slicing - Operations - Add Values - Count values - Nested List

```
lista = ['obj1', 'obj2', 'obj3']

#list has an index so you can find it's correspondent element, starting from 0
print (lista[0])  #output = obj1

#you can start counting from the end with negatives starting from -1
print (lista[-1])  #output = obj3

#you can change the value of an index
lista[1] = 'mucca'

#to slice [start ; end -1 ; jumps]
#start is starting index // end is ending index, not counted // jumps is skips (1 is default, if -1 reverses)
#[:n] automatically starts from 0 and finishes to n-1
#[n:] starts from n and finishes at the end (counted) and [::] copies the whole list
new_list = lista[:3]

#you can do operations like concatenation (list1 + list2)
liste = lista + [3,4,6,12]

#use list.append(value) to add a value at the end
lista.append('mucchina')

#use list.insert(index, value) to add a value at that index
lista.insert(0,'mucchetta')

print (len(lista))  #output = 3
#you can nest lists (or any other collector or type) inside lists
lista = [ [0,2,1], ['mu', 'pu'] ]
```

1.2. TUPLE

- Tuples are ordered, immutable sequences. Once created, you cannot modify their contents. Lists and Tuples are similar, but tuples cannot be changed and are used just to store data.

Creation - ~~Change Values~~ - Slicing - Operations - ~~Add Values~~ - Count Values - Nested Tuple

```
tupla = ('obj1', 'obj2', 'obj3')

#you can slice tuples in the same way as lists
#you cannot change values in a tuple
#you can do the same operations as for lists, but you still won't be able to change the values
tuplo = tupla + (3,4,6,12)

#you cannot add elements because tuples are immutable
#you can use len(tuple) to count its values
#you can nest a tuple inside a tuple
tuplo = ( (1,2,'f'), 'frr', False)
```

1.3. SET

- Sets are unordered collections of unique elements.

Creation - ~~Change Values~~ - ~~Slicing~~ - Operations - Add Values - Count Values - ~~Nested Set~~

```
#you create a set with { } without index, in fact sets are unordered.
#it means the order you add elements in a set may differ from the output
#please note you cannot have duplicates in sets
setz = {"mucca", False, 3, False}
print (setz)  #output = {False, 3, 'mucca'}

#since set doesn't have an index, you cannot slice them
#you cannot use + to combine two sets but you can use other methods like | , &
set1 = {"ok", 3}
set2 = {"nok", 1}
setz = set1 | set2
print(setz)

#you cannot use append or insert methods but you can use add(1 value) or update ({more values})
setz.add(5)
setz.update({6,9})

#you can use lent(set) to count its values
#you cannot nest a set inside another set
```

1.4. DICTIONARY

- Dictionaries are collections of key-value pairs that allow you to map keys to values. Similar to Sets, but Dictionaries have named keys and thus can modify their values.

Creation - Change Values - ~~Slicing~~ - Operations - Add Values - Count Values - Nested Dictionary

```
#you create a dictionary with { } like sets, but you must add keys (that can be any type)
dictz = {"key1": 'hola', "key2": 'ciao', "key3": 'hello'}

#you can change values, but in order to change its keys you have to create a new key and delete the other
dictz["key1"] =  'mucca'  #change value
dictz["key4"] = dictz.pop("key2")    #change key
print (dictz)

#since dictionary are unordered, you cannot use Slicing methods
#you cannot use + to combine two sets but you can use other methods like | , &
diz = {'1': "ok"}
dicto = {'2': "nok"}
dictz = diz | dicto
print (dictz)

#you add a new value just like the creation or change value
dictz["key5"] = 'puu'

#you can use lent(dictionary) to count its values
#you cannot nest a dictionary inside another dictionary
```

2. COLLECTIONS MODULE

'Collections' module has some interesting methods to use collectors, each with its own features.

2.2.1. COLLECTIONS - NAMEDTUPLE

- Similar to build-in tuple, a collections namedtuple is a Factory Function for creating tuple subclasses with named fields. It allows you to access elements by name.

Creation - Change Values - Slicing - Operations - ~~Add Values~~ - Count Values - Nest

```
#you create it with a name and some values. It may be used as a "vector"
import collections
point = collections.namedtuple("Name", ["x","y"])
p = point(1,2)
```

2.2.2. COLLECTIONS - DEQUE

- A deque is a double-ended queue that supports insertions and deletions from both ends. It's used to implement queues and stacks efficiently

Creation - Change Values - ~~Slicing~~ - Operations - Add Values - Count Values - Nest

```
#you create it like a NumPy array, but you can add or delete values from both sides
#Collections deque doesn't have duplicates
import collections
deck = collections.deque([0,5,"ok"])
```

2.2.3. COLLECTIONS - COUNTER

- A counter is a subclass of a dictionary designed for counting elements in an iterable. It's used for tasks like finding the most common element in a collection.

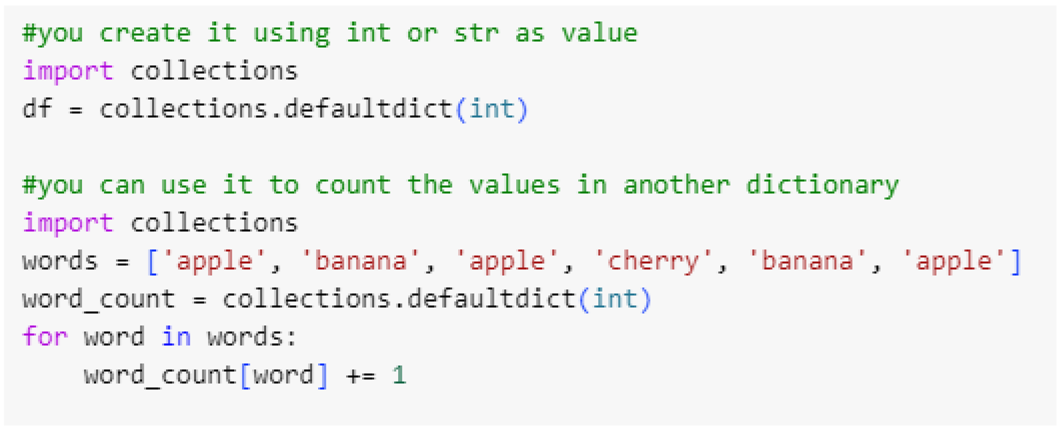
Creation - Change Values - ~~Slicing~~ - Operations - Add Values - Count Values - Nest

```
#you create it with a string
#you can print the number of elements in the [ ]
import collections
count = collections.Counter('string')
print(count['s'])    #output = 1
```

2.2.4. COLLECTIONS - DEFAULTDICT

- A defaultdict is a subclass of a dictionary that allows you to specify a default value for a missing key.

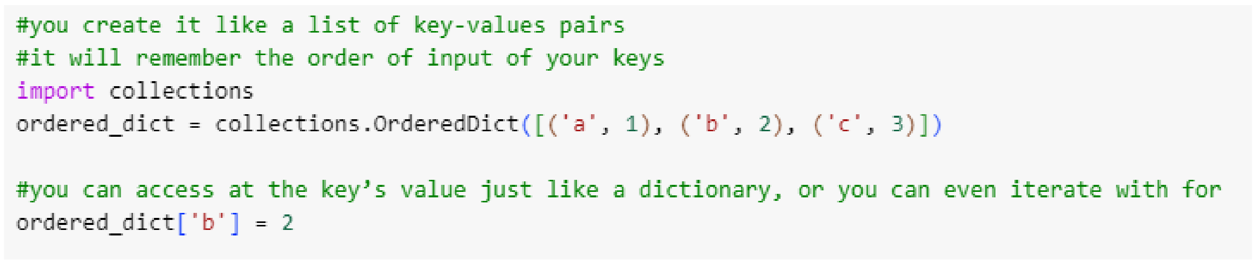
Creation - ~~Change Values~~ - ~~Slicing~~ - ~~Operations~~ - ~~Add Values~~ - Count Values - ~~Nest~~



```
#you create it using int or str as value
import collections
df = collections.defaultdict(int)

#you can use it to count the values in another dictionary
import collections
words = ['apple', 'banana', 'apple', 'cherry', 'banana', 'apple']
word_count = collections.defaultdict(int)
for word in words:
    word_count[word] += 1
```

2.2.5. COLLECTIONS - ORDEREDDICT

- An OrderedDict is a dictionary subclass that remembers the order in which key-value pairs were added to the dictionary, unlike a regular dict which does not guarantee any specific order.



```
#you create it like a list of key-values pairs
#it will remember the order of input of your keys
import collections
ordered_dict = collections.OrderedDict([('a', 1), ('b', 2), ('c', 3)])

#you can access at the key's value just like a dictionary, or you can even iterate with for
ordered_dict['b'] = 2
```

2.2.6. COLLECTIONS - USERDICT

- UserDict is a class for creating custom dictionary-like objects.
It's intended for subclassing and can be used as a base class for creating custom dictionary classes with customized behavior.

In this example, MyDict is a custom dictionary class that inherits from UserDict, allowing you to define custom methods or behavior for your dictionary.

```python
from collections import UserDict
class MyDict(UserDict):
    def __init__(self, data=None):
        super().__init__()
        if data:
            self.update(data)
my_dict = MyDict({'a': 1, 'b': 2})
```

2.2.7. COLLECTIONS - USERLIST

- UserList is a class for creating custom list-like objects.
It's designed to be subclassed and used as a base class for creating custom list classes with specialized functionality.

```python
from collections import UserList
class MyList(UserList):
    def append_unique(self, item):
        if item not in self.data:
            self.data.append(item)
my_list = MyList([1, 2, 3])
my_list.append_unique(2)   #Won't add duplicate
```

2.2.7. COLLECTIONS - USERLIST

- UserString is a class for creating custom string-like objects.
It's intended for subclassing and can be used as a base class for creating custom string classes with specialized behavior.
In this example, MyString is a custom string class that inherits from UserString, allowing you to define custom methods like reverse to manipulate the string.
Here, MyList is a custom list class that inherits from UserList, allowing you to add custom methods like append_unique to manipulate the list.

```python
from collections import UserString
class MyString(UserString):
    def reverse(self):
        return self.data[::-1]
my_string = MyString("hello")
reversed_str = my_string.reverse()
```

3. OTHER COLLECTOR MODULES

These modules have some unique “Collectors” that are similar to those build-in, but with different methods and functionalities.
Those are:

3.1. NUMPY - ARRAY

- The `array` module comes up with numpy and provides arrays, which are more memory-efficient than standard lists when you need to store a large number of elements of the same data type.
With NumPy arrays you can create multidimensional arrays to store even more data.
Those arrays are very similar to lists and thus you can use almost every list method.

Creation - Change Values - Slicing - ~~Operations~~ - Add Values - Count Values - Nested Array

```
#you can create an array with a common list using numbers
#or you can create an array via the numpy module
import numpy
arr = numpy.array([7, 2, 5])

#you can change a value in the array like this:
arr[0] = 3
print(arr)

#you can slice them as a list
#you cannot do operations
#to add values you have to use numpy.append(array, new_value)
arr = numpy.append(arr, 3)

#you can use lent(array) to count its values
#with NumPy arrays you can nest arrays inside an array
```

3.2. HEAPQ

- The `heapq` module offers a collection of heap queue algorithms, which can be useful for tasks like finding the largest or smallest elements in a collection.
The heapq module in Python provides a collection of heap queue algorithms. A heap is a specialized tree-based data structure that satisfies the heap property, where the parent node is either greater than or less than its child nodes, depending on whether it's a max-heap or a min-heap.
The heapq module is a versatile and efficient tool for various scenarios where you need to maintain a collection of elements with the minimum or maximum element easily accessible.

BASIC FUNCTIONS:

- heapify(iterable)
This function transforms a regular list into a valid heap in-place.
It rearranges the elements so that the heap property is satisfied.

- heappush(heap, item)
Adds an element to the heap while maintaining the heap property.

- heappop(heap)
Removes and returns the smallest element from the heap.

- heappushpop(heap, item)
Pushes a new item onto the heap and then pops and returns the smallest item from the heap. It's more efficient than calling heappush() followed by heappop() separately.

- nlargest(n, iterable, key=None)
Returns the n largest elements from an iterable.
You can specify a key function to extract a comparison key from each element.

- nsmallest(n, iterable, key=None)
Returns the n smallest elements from an iterable.
You can specify a key function to extract a comparison key from each element.

3.3. QUEUE

- The queue module in Python provides implementations of various types of thread-safe queues, which can be used for inter-thread communication and synchronization. These queues are useful when multiple threads or processes need to exchange data in a safe and orderly manner.
The queue module is a valuable tool for handling synchronization and communication between threads or processes in Python. It ensures that data is shared safely and efficiently.
Here are the main classes provided by the queue module:

Queue:
This class implements a basic first-in-first-out (FIFO) queue.
It's a thread-safe data structure, which means that multiple threads can safely add and remove items from the queue without causing data corruption or race conditions.

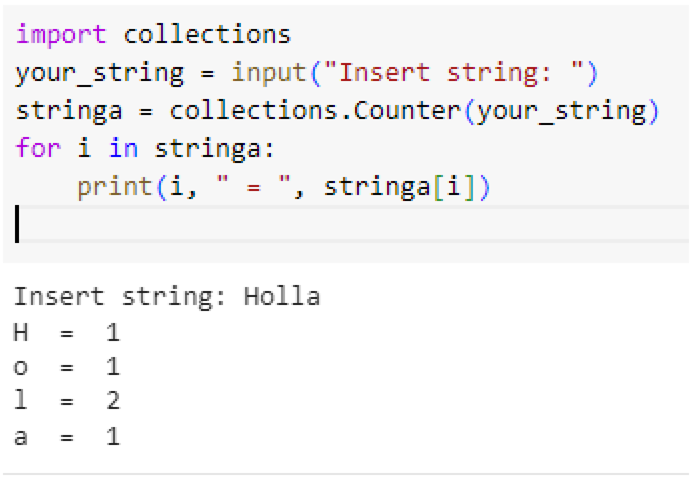
LifoQueue:
This class implements a last-in-first-out (LIFO) queue, which is essentially a stack.
It is also thread-safe.

PriorityQueue:
This class implements a priority queue, where items are assigned priorities, and the item with the highest priority is retrieved first.
The priority of items is determined by their comparison key (e.g., numeric values or custom comparison functions).
Like the others, it's also thread-safe.

PRACTICAL TASKS

- Task 1: Create a `Counter` to count the occurrences of each character in a given string.
- Task 2: Use a `defaultdict` to group words in a list by their first letter.
- Task 3: Implement an `OrderedDict` to maintain the order of items added in a dictionary.

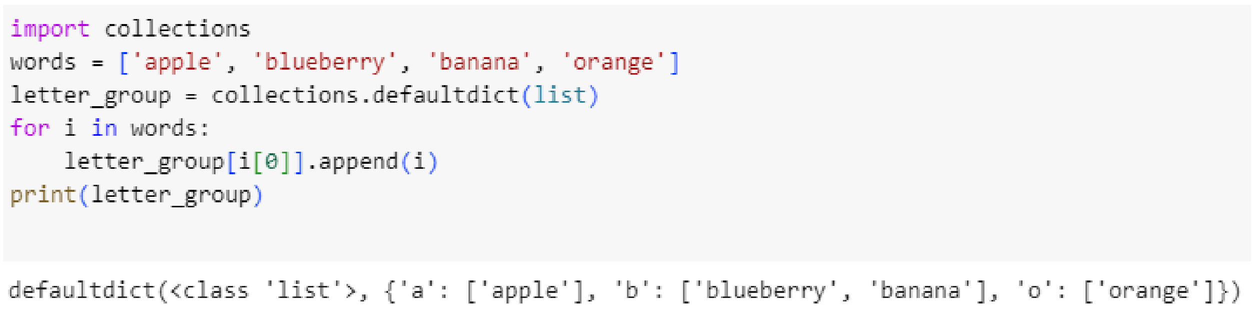
1. Counter



```python
import collections
your_string = input("Insert string: ")
stringa = collections.Counter(your_string)
for i in stringa:
    print(i, " = ", stringa[i])
```

```
Insert string: Holla
H  =  1
o  =  1
l  =  2
a  =  1
```

2. DefaultDict



```python
import collections
words = ['apple', 'blueberry', 'banana', 'orange']
letter_group = collections.defaultdict(list)
for i in words:
    letter_group[i[0]].append(i)
print(letter_group)
```

```
defaultdict(<class 'list'>, {'a': ['apple'], 'b': ['blueberry', 'banana'], 'o': ['orange']})
```

3. OrderedDict

```python
import collections
ordered_dict = collections.OrderedDict([('a',1),('b',2)])
print(ordered_dict)
```

```
OrderedDict([('a', 1), ('b', 2)])
```

2. ALGORITHMS

Sorting and Searching

1. Study Material:

- Learn about basic sorting algorithms: Bubble Sort, Merge Sort, Insertion Sort.
- Understand basic searching algorithms: Linear Search and Binary Search

2. Practical Tasks:

- Task 1: Implement Bubble Sort in Python. Use it to sort a list of numbers.
- Task 2: Implement Merge Sort in Python. Test it on a list of numbers.
- Task 3: Write a function for Linear Search that searches for an element in a list.
- Task 4: Write a function for Binary Search. Remember, Binary Search requires the list to be sorted.

Guidelines:

- Try writing the code for these algorithms yourself.
- Test your algorithms with different lists of numbers.
- Focus on understanding how each algorithm works and its time complexity

STUDY MATERIAL

1.0 WHAT'S AN ALGORITHM?

- In the context of computer science and programming, an algorithm is a set of instructions designed to perform a specific task or solve a particular problem. Think of it as a recipe in cooking: a series of steps that you need to follow to achieve a desired outcome.

Key characteristics of algorithms include:

Precision: Each step is clearly defined.
Finiteness: They must terminate after a finite number of steps.
Input: They take zero or more inputs.
Output: They produce one or more outputs.
Effectiveness: Each step must be basic enough to be carried out, in principle, by a person using only pencil and paper.
Algorithms are fundamental in computer programming. They are used for data processing, calculation, and automated reasoning.

2.0 BUBBLE SORT ALGORITHM

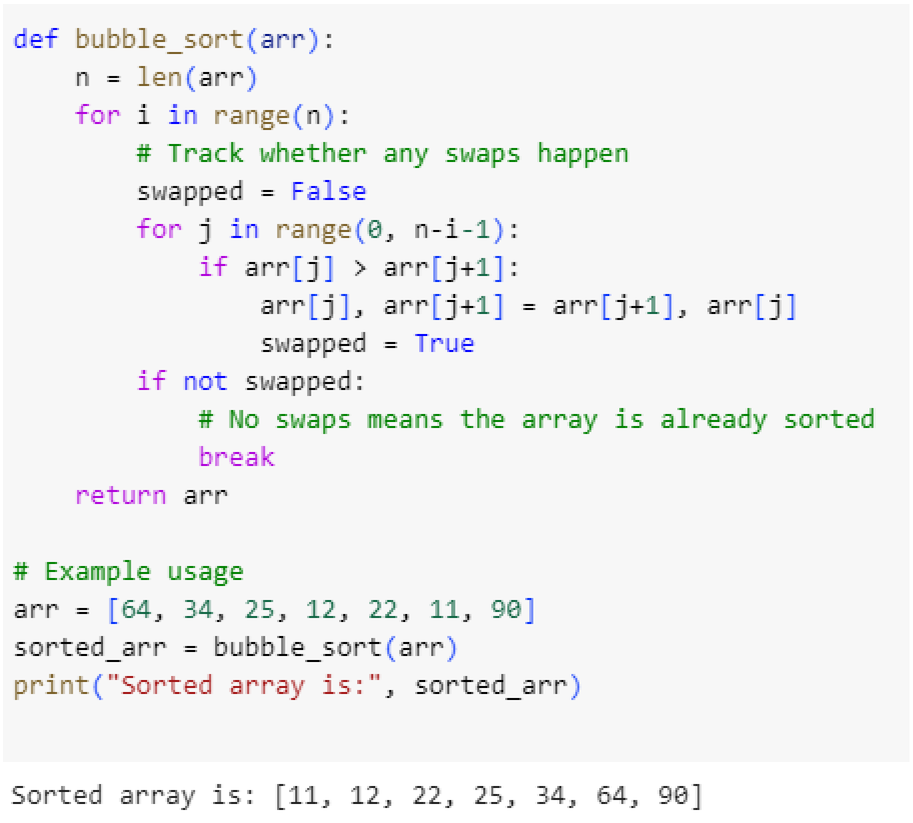
Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm gets its name because smaller elements "bubble" to the top of the list.

How Bubble Sort Works:
- Start at the beginning of the list.
- Compare the first two elements.
- If the first is greater than the second, swap them.
- Move to the next pair of elements and repeat the process.
- Continue until you reach the end of the list. At this point, the last element should be the highest number.
- Repeat the entire process for the remaining list (excluding the last element which is already sorted).
- Continue until no swaps are needed, meaning the list is sorted.

Time Complexity:
- Best Case: O(n) when the list is already sorted.
- Worst and Average Case: O(n^2) where n is the number of elements. This makes it inefficient on large lists.

Implementation in Python

Here is a simple implementation of Bubble Sort in Python:



```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Track whether any swaps happen
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            # No swaps means the array is already sorted
            break
    return arr

# Example usage
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array is:", sorted_arr)
```

```
Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

2.1 MERGE SORT ALGORITHM

Merge Sort is an efficient, stable, comparison-based, divide and conquer sorting algorithm. Most implementations produce a stable sort, meaning that the implementation preserves the input order of equal elements in the sorted output.

How Merge Sort Works:
- Divide: The list is divided into n sublists, each containing one element (a list of one element is considered sorted).
- Conquer: Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

The Merge Process:
- The merging of two sorted lists into a single sorted list is a crucial step in Merge Sort. It involves comparing the elements of the sublists and combining them in sorted order.

Time Complexity:
- Best, Worst, and Average Case: O(n log n), where n is the number of elements. This makes Merge Sort much more efficient for larger lists than simpler algorithms like Bubble Sort.

2.2 INSERTION SORT ALGORITHM

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one item at a time. It's much less efficient on large lists than more advanced algorithms like quicksort, heapsort, or merge sort, but has some advantages in certain scenarios.

How Insertion Sort Works:
- Iterative Process: Starting from the second element of the array (assuming the first element is the initial sorted portion), compare this element with the elements before it.
- Find the Correct Position: Move backwards through the sorted portion and compare the current element with each of these elements. If the current element is smaller, shift the larger elements one position up to make space for the current element.
- Insert the Element: Insert the current element into its correct position within the sorted portion.
- Repeat: Continue this process for each element until the entire array is sorted.

Characteristics:
- Adaptive: Efficient for (partially) sorted arrays.
- Stable: Does not change the relative order of elements with equal keys.
- In-Place: Only requires a constant amount O(1) of additional memory space.

Time Complexity:
- Best Case: O(n) when the array is already sorted.
- Worst and Average Case: O(n^2) where n is the number of elements.

3 LINEAR SEARCH ALGORITHM

Linear Search, also known as Sequential Search, is the most straightforward searching algorithm. In this algorithm, each element of the list is checked in sequence until the desired element is found or the list ends.

How Linear Search Works:
- Start from the First Element: Examine the first element of the list.
- Check Each Element: Compare each element with the value you're searching for.
- Found or Not: If you find the element, return its index (or a confirmation). If you reach the end of the list without finding the element, the item is not in the list.

Characteristics:
- Simplicity: Very straightforward and easy to implement.
- No Need for Sorting: Works on unsorted and sorted lists.
- Performance: Inefficient for large lists, as it can potentially require checking each element.

Time Complexity:
- Best Case: O(1) if the element is found at the first position.
- Worst Case: O(n) where n is the number of elements in the list (when the element is not present or is at the last position).

3.1 BINARY SEARCH ALGORITHM

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed the possible locations to just one.

How Binary Search Works:
- Compare the Middle Item: Find the middle item of the array. If it's equal to the target value, the search is complete.
- Halve the Search Space: If the target value is less than the middle item, repeat the search on the left half of the array; if it's more, search the right half.
- Iterative or Recursive: This process can be implemented iteratively (using loops) or recursively (function calling itself with new bounds).
- Repeat: Continue halving the search space until the value is found or the search space is empty.

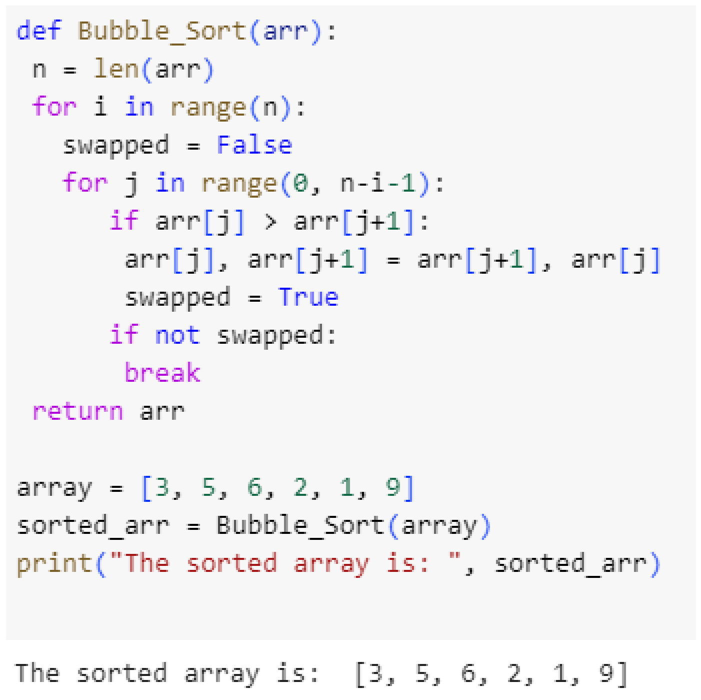
Characteristics:
- Efficiency: Much faster than linear search, especially for large lists.
- Requirement: The list must be sorted to apply binary search.

Time Complexity:
- Best Case: O(1) if the element is at the middle.
- Worst and Average Case: O(log n) where n is the number of elements.

PRACTICAL TASKS

- Task 1: Implement Bubble Sort in Python. Use it to sort a list of numbers.
- Task 2: Implement Merge Sort in Python. Test it on a list of numbers.
- Task 3: Write a function for Linear Search that searches for an element in a list.
- Task 4: Write a function for Binary Search. Remember, Binary Search requires the list to be sorted.

1. Bubble Sort Algorithm



```
def Bubble_Sort(arr):
 n = len(arr)
 for i in range(n):
   swapped = False
   for j in range(0, n-i-1):
      if arr[j] > arr[j+1]:
       arr[j], arr[j+1] = arr[j+1], arr[j]
       swapped = True
      if not swapped:
       break
 return arr

array = [3, 5, 6, 2, 1, 9]
sorted_arr = Bubble_Sort(array)
print("The sorted array is: ", sorted_arr)
```

```
The sorted array is:  [3, 5, 6, 2, 1, 9]
```

2. Merge Sort Algorithm

```
def merge_sort(arr):
    if len(arr) > 1:
        # Finding the mid of the array
        mid = len(arr) // 2
        # Dividing the array into two halves
        L = arr[:mid]
        R = arr[mid:]
        # Sorting the first half
        merge_sort(L)
        # Sorting the second half
        merge_sort(R)
        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1


        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr

# Example usage
arr = [12, 11, 13, 5, 6, 7]
print("Given array is", arr)
arr = merge_sort(arr)
print("Sorted array is", arr)
```

```
Given array is [12, 11, 13, 5, 6, 7]
Sorted array is [5, 6, 7, 11, 12, 13]
```

3. Linear Search Algorithm

```python
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i  # Returning the index where the element is found
    return -1  # Element is not found

# Example usage
arr = [2, 4, 5, 7, 9, 11, 15]
print(arr)
x = 9

result = linear_search(arr, x)
if result != -1:
    print(f"Element {x} is present at index {result}.")
else:
    print(f"Element {x} is not in the array.")
```

```
[2, 4, 5, 7, 9, 11, 15]
Element 9 is present at index 4.
```

4. Binary Search Algorithm

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        # Check if element is present at mid
        if arr[mid] == x:
            return mid
        # If element is smaller than mid, it can only be present in left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        # Element is not present in array
        return -1
# Example usage
arr = [2, 3, 4, 10, 40]
print(arr)
x = 10
result = binary_search(arr, 0, len(arr)-1, x)
if result != -1:
    print(f"Element {x} is present at index {result}")
else:
    print("Element is not present in array")
```

```
[2, 3, 4, 10, 40]
Element 10 is present at index 3
```

3. FILE OPERATIONS AND EXCEPTION HANDLING

Sorting and Searching

1. Study Material:

- Learn about file operations in Python: reading, writing, appending, and working with file contexts (with statement).
- Understand different types of file formats you can work with, such as text files, CSV, and JSON.
- Get familiar with exception handling: try, except, else, finally.

2. Practical Tasks:

- Task 1: Write a Python program to read a text file and print its contents.
- Task 2: Create a Python script to write data to a CSV file and then read from it.
- Task 3: Write a Python program to handle multiple exceptions (FileNotFoundError, ValueError, etc.).
- Task 4: Implement a Python script that uses a finally block to close a file, regardless of whether an exception occurs.

Guidelines:

- Focus on handling different types of file operations and understanding their use cases.
- Practice writing clean and efficient exception handling blocks.
- Test your code with different scenarios, including error scenarios to see how your exception handling works.

STUDY MATERIAL

1.1 OPENING A FILE

Use the open() function to open a file.
Syntax: file_object = open(file_name, mode)
The mode can be 'r' for reading, 'w' for writing, 'a' for appending, 'r+' for both reading and writing, etc.
Example: `f = open('example.txt', 'r')`

1.2 READING FROM A FILE

Methods: read(), readline(), readlines()
read() reads the entire file.
readline() reads the next line.
readlines() reads all the lines and returns them as each line a list item.
Example: `content = f.read()`

1.3 WRITING TO A FILE

Methods: write(), writelines()
write() is used to write a string to a file.
writelines() can write a list of strings.
In write mode, if the file doesn’t exist, Python will create it.
Example: `f.write('Hello, Python!')`

1.4 APPENDING TO A FILE

Opening a file in append mode ('a') allows you to add new data to the end of the file without deleting the existing data.

Example: f.write('Appending new line')

1.5 CLOSING A FILE

Always close the file after completing your operations on it using close().

Example: f.close()

2.1 USING “WITH” STATEMENT FOR FILES

The with statement automatically takes care of closing the file once it leaves the with block, even in cases of error.

Example:

```
with open('example.txt', 'r') as f:
    content = f.read()
```

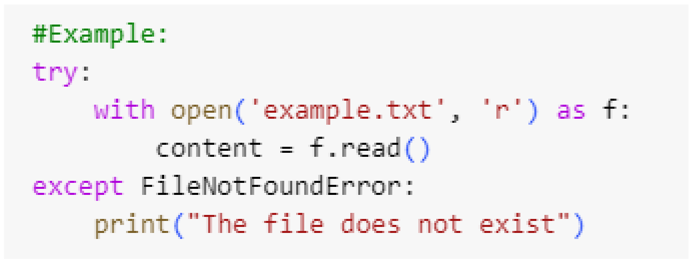
3.1 FILE FORMATS

Text Files (.txt): Used for reading and writing plain text.
CSV Files (.csv): Comma-Separated Values, used for spreadsheets and databases.
JSON Files (.json): JavaScript Object Notation, used for storing and exchanging data.

4.1 EXCEPTION HANDLING IN FILE OPERATIONS

While performing file operations, several exceptions can occur, such as FileNotFoundError if the file does not exist or IOError if the file is unreadable. Using try and except blocks can help in handling these exceptions gracefully.

Example:

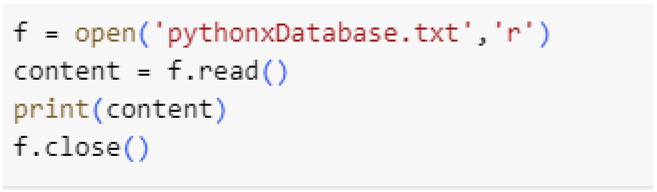


```
#Example:
try:
    with open('example.txt', 'r') as f:
        content = f.read()
except FileNotFoundError:
    print("The file does not exist")
```

PRACTICAL TASKS

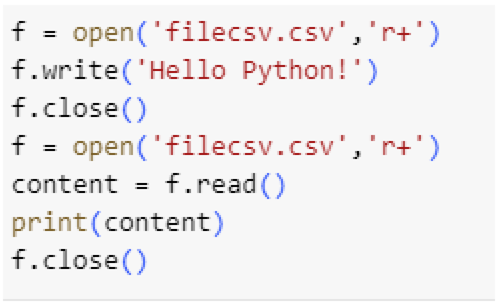
- Task 1: Write a Python program to read a text file and print its contents.
- Task 2: Create a Python script to write data to a CSV file and then read from it.
- Task 3: Write a Python program to handle multiple exceptions (FileNotFoundError, ValueError, etc.).
- Task 4: Implement a Python script that uses a finally block to close a file, regardless of whether an exception occurs.

1. Read a text file



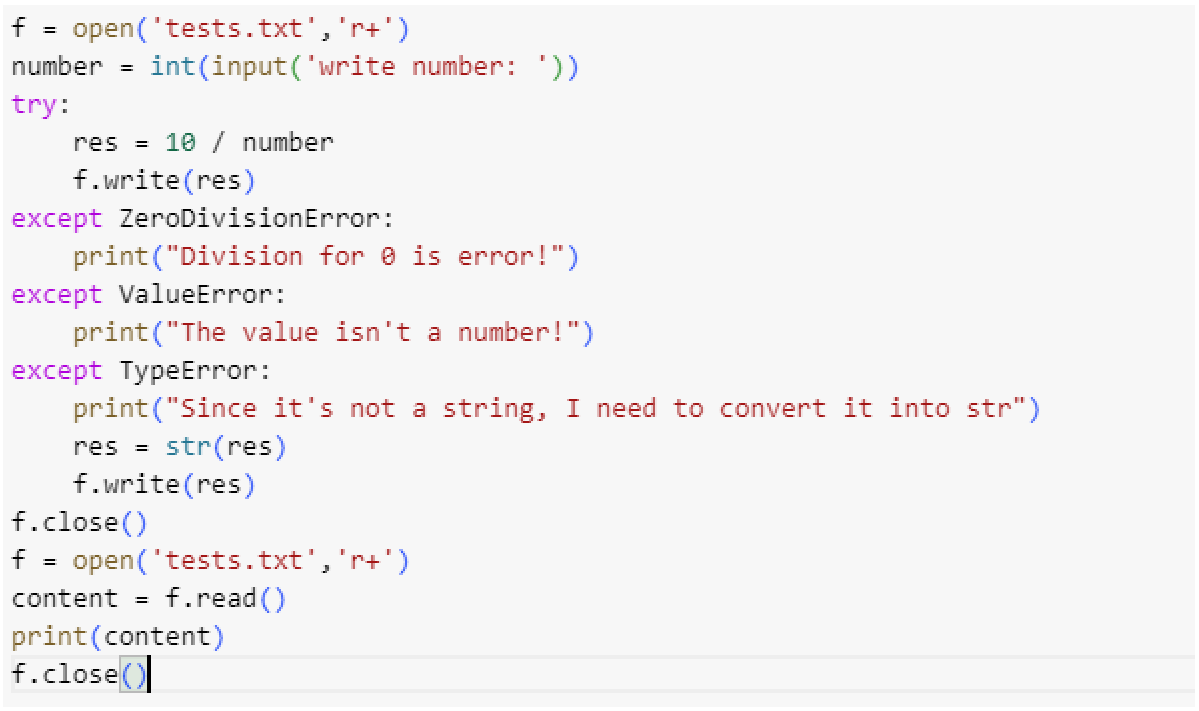
```python
f = open('pythonxDatabase.txt','r')
content = f.read()
print(content)
f.close()
```

2. Write and read a CSV file



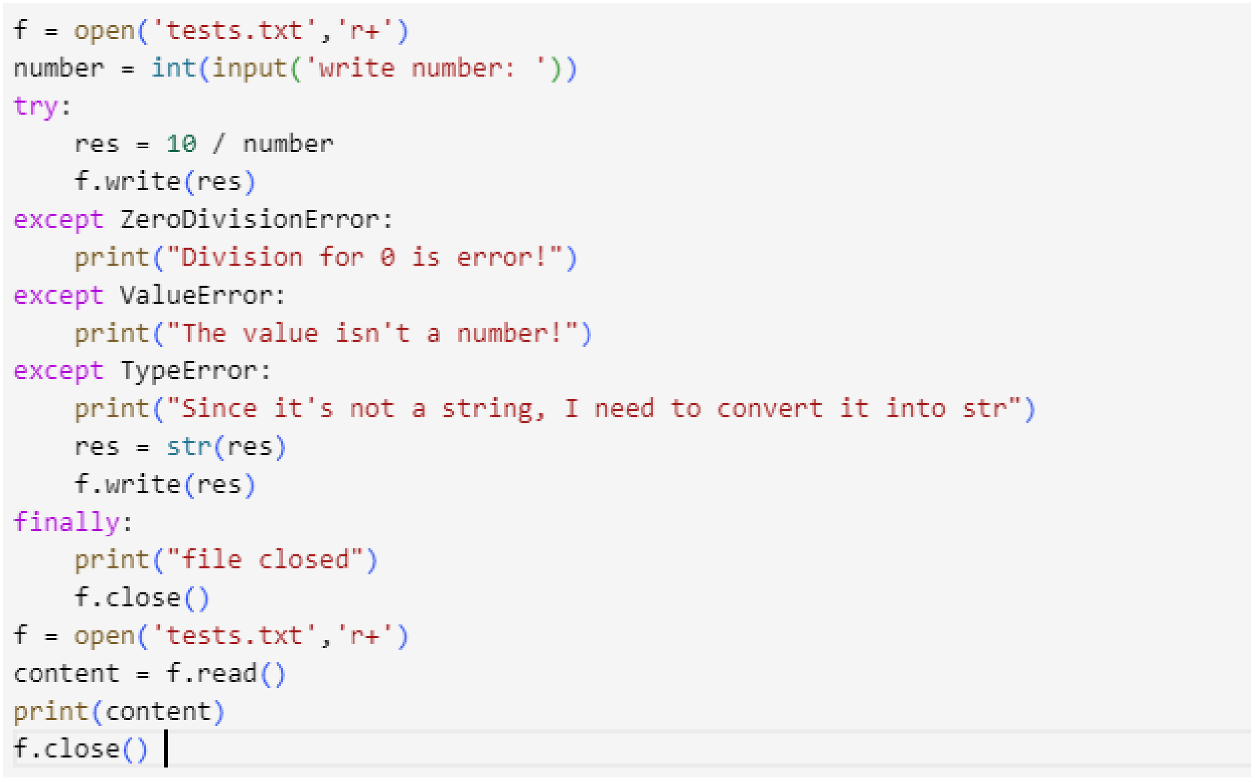
```python
f = open('filecsv.csv','r+')
f.write('Hello Python!')
f.close()
f = open('filecsv.csv','r+')
content = f.read()
print(content)
f.close()
```

3. Handle multiple exceptions while writing and reading files



```python
f = open('tests.txt','r+')
number = int(input('write number: '))
try:
    res = 10 / number
    f.write(res)
except ZeroDivisionError:
    print("Division for 0 is error!")
except ValueError:
    print("The value isn't a number!")
except TypeError:
    print("Since it's not a string, I need to convert it into str")
    res = str(res)
    f.write(res)
f.close()
f = open('tests.txt','r+')
content = f.read()
print(content)
f.close()
```

4. Use finally to close a file



```
f = open('tests.txt','r+')
number = int(input('write number: '))
try:
    res = 10 / number
    f.write(res)
except ZeroDivisionError:
    print("Division for 0 is error!")
except ValueError:
    print("The value isn't a number!")
except TypeError:
    print("Since it's not a string, I need to convert it into str")
    res = str(res)
    f.write(res)
finally:
    print("file closed")
    f.close()
f = open('tests.txt','r+')
content = f.read()
print(content)
f.close()
```

4. FUNCTIONAL PROGRAMMING

Lambda Functions

1. Study Material:

- Learn about lambda functions, also known as anonymous functions, and their uses.
- Understand higher-order functions like `map()`, `filter()`, and `reduce()`.
- Explore the concepts of closures and decorators in Python.

2. Practical Tasks:

- Task 1: Write a lambda function to compute the square of a number.
- Task 2: Use `map()` to apply a function over a list of numbers.
- Task 3: Use `filter()` to filter out elements from a list based on a condition.
- Task 4: Use `reduce()` to reduce a list of numbers into a single cumulative value.
- Task 5: Create a simple decorator that logs the execution time of a function.

Guidelines:

- Try to write concise and efficient lambda functions.
- Understand how `map()`, `filter()`, and `reduce()` can replace traditional loops in many cases.
- Get a good grasp of closures and decorators, as they are powerful tools in Python.

STUDY MATERIAL

1.0 WHAT'S FUNCTIONAL PROGRAMMING?

- Functional programming is a programming paradigm where you primarily construct and structure your code using functions. Python supports functional programming and provides several features to facilitate this style.

2.0 LAMBDA FUNCTIONS

- A lambda function in Python is a small anonymous function, meaning it doesn't need to be named. It can have any number of arguments, but can only have one expression. The power of lambda functions is best shown when you use them as an anonymous function inside another function.

Key Characteristics of Lambda Functions:
1. Syntax: The basic syntax of a lambda function is `lambda arguments: expression`.
2. Simplicity: Used for creating small, one-time, anonymous function objects.
3. Usage: Commonly used with functions like `map()`, `filter()`, and `reduce()`.

Example of a Lambda Function:
Let's say you want a function to compute the square of a number. A traditional function in Python would look like this:

```python
def square(num):
    return num * num
```

Using a lambda function, you can achieve the same with a single line:

```
square = lambda num: num * num
```

You can then use this `square` function exactly like any other function:

```
print(square(5))
```

```
25
```

Lambda functions are a concise way to represent small functions. They are particularly useful when you need a small function for a short period and do not want to formally define it with the `def` keyword.

2.1 Deeper Dive into Lambda Functions

2.1.1. Technical Definition
- A lambda function is a small anonymous function defined by the keyword `lambda`.
- It follows the syntax: `lambda arguments: expression`.
- The `expression` is executed and the result is returned when lambda is called.

2.1.2 Anonymity
- Lambda functions are anonymous, which means they are not bound to a name at runtime. This feature makes them useful as throw-away functions or for passing simple functions as arguments to higher-order functions.

2.1.3. Arguments
- Lambda functions can take any number of arguments, including zero.
- Syntax Examples:
  - No arguments: `lambda: True`
  - One argument: `lambda x: x*2`
  - Multiple arguments: `lambda x, y: x + y`

4. Single Expression
- Lambda function can only have a single expression, which is evaluated and returned.
- They cannot contain statements or annotations, and they don't support type hints.

5. Use Cases
- Often used when a function is required for a short duration
- Commonly used with higher-order functions like `map()`, `filter()`, and `reduce()`.
- Useful in GUI frameworks for defining callback functions.

6. Lambda vs Regular Functions
   - Lambda functions are less expressive but can be less readable
   - Regular functions can be more descriptive (with names and comments).

7. Scoping
   - They follow the same scoping rules as regular functions. Lambda functions have access to variables from the scope in which they are created (lexical scoping).

8. Return Behavior
   - Implicitly return the result of the expression. There is no need for a `return` statement.

Lambda functions are not always the best choice, especially for complex logic, as they sacrifice readability for brevity. However, for simple functionalities, they can provide an elegant and efficient way to write concise code.

3.0 MAP() FUNCTIONS

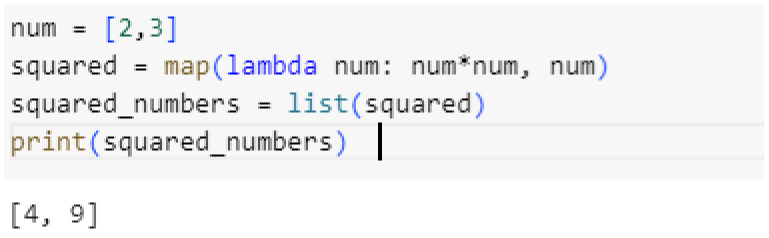
The `map()` function in Python applies a given function to each item of an iterable (like a list) and returns a map object (which is an iterator). It's often used for transforming data.

1. Syntax:
map(function, iterable, ...)
- `function`: The function to execute for each item.
- `function` is typically a lambda function that takes an item from `iterable` (like a list) and returns some form of transformation of that item.
- `iterable`: A sequence, collection, or an iterator object.

2. Outputting Results from map():
The `map()` function returns a map object, which is an iterator. To actually view the results, you need to convert this iterator into a list or iterate through it.

Here are two ways to output the results:

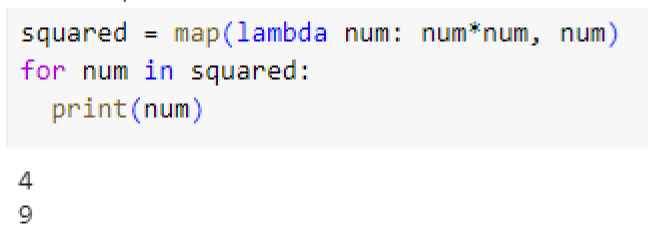
A. Convert to List:
   - Directly convert the map object to a list.
   - Example:



```python
num = [2,3]
squared = map(lambda num: num*num, num)
squared_numbers = list(squared)
print(squared_numbers)
```

```
[4, 9]
```

B. Iterate Through the Map Object:
  - Use a loop to iterate through the map object and print each item.
  - Example:

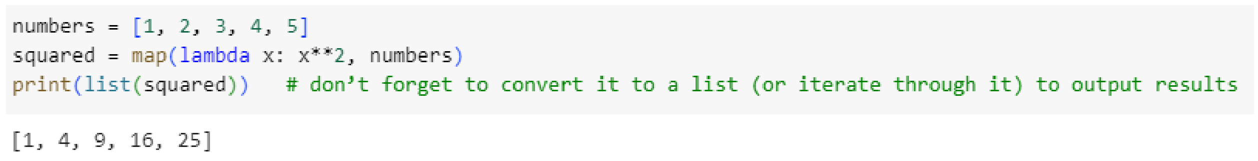


```
squared = map(lambda num: num*num, num)
for num in squared:
  print(num)
```

```
4
9
```

It's important to note that map objects can only be iterated once. After you've gone through them, they'll be empty. So, if you need to use the results multiple times, it's better to convert them to a list.

Example:
- Consider a lambda function used within a `map()` to square each number in a list:



```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared))   # don’t forget to convert it to a list (or iterate through it) to output results
```

```
[1, 4, 9, 16, 25]
```

- The `lambda x: x**2` part is the lambda function.
- `x` is the argument to the lambda function.
- `x**2` is the expression that gets evaluated and returned.
- `map()` applies this lambda function to each element of `numbers`.

3.1 FILTER() FUNCTIONS

The `filter()` function is used to create an iterator from elements of an iterable for which a function returns `True`. In simple terms, `filter()` filters the given iterable with the help of a function that tests each element in the iterable to be true or not.

1. Syntax:
filter(function, iterable)
- `function`: A function that tests whether each element of an iterable is true or not. This function needs to return a boolean value (either `True` or `False`).
- `iterable`: The iterable which is to be filtered.

2. How filter() Works:
1. `filter()` takes a function and an iterable.
2. It applies the function to every element of the iterable.
3. If the function returns `True`, the element is included in the result.

3. Using filter() with Lambda Functions:

`filter()` is commonly used with lambda functions because we often use `filter()` for small operations where we don't want to define a separate function.

Example:
Let's filter a list of numbers to get only even numbers using `filter()` and a lambda function:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
```

4. Outputting Results from filter():
Similar to `map()`, the `filter()` function returns a filter object, which is an iterator. To view the results, you can convert it into a list or iterate through it.

3.2 REDUCE() FUNCTIONS

Unlike `map()` and `filter()`, `reduce()` isn't a built-in function but is available in the `functools` module.
`reduce()` is a function that applies a given function to the items of an iterable, cumulatively, so as to reduce the iterable to a single value. This makes `reduce()` particularly useful for performing computations where you need to carry forward a value as you iterate.

1. Syntax:
```
from functools import reduce
reduce(function, iterable[, initializer])
```
- `function`: This is a function that takes two arguments, cumulatively applies these to the items of the iterable from left to right, reducing it to a single value.
- `iterable`: The iterable whose items are to be reduced.
- `initializer` (optional): A value with which to start the reduction. If provided, it's placed before the items of the sequence in the calculation, serving as a default if the iterable is empty.

2. How reduce() Works:
1. The `function` first applies to the first two items from the iterable and generates a partial result.
2. This partial result is then combined with the next item in a similar manner, and this process continues until there are no more items left.

Example:
Let's say we want to find the product of all numbers in a list:

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
```

This will calculate ((((1*2)*3)*4)*5), giving us the product of all numbers in the list.

3. Outputting Results from reduce():
The result from `reduce()` is not an iterator but the final cumulative value. So, you can simply print or use the result directly:

```
print(product)  # Output: 120
```

4.1 CLOSURE

A closure in Python refers to a situation where a function remembers the environment in which it was created. This is possible because functions in Python are first-class citizens, meaning they can be passed around and used as objects.

1. Key Aspects of Closures:
1. **Nested Functions**: In Python, we have the ability to define a function inside another function. The inner function is accessible only within the scope of the outer function (not outside of it).

2. Preserving the Inner Function’s Environment:
When the outer function returns the inner function, the inner function remembers the variables and their values from its enclosing scope, even if the outer function's execution is complete. This is the essence of a closure.

3. When to Use Closures:
- Use closures to avoid the use of global values and provide some form of data hiding.
- They can also provide an object-oriented solution to the problem.

4.1.1 Understanding the Closure Code with an example

```python
def outer_function(text):
    def inner_function():
        print(text)
    return inner_function

my_func = outer_function('Hello, World!')
my_func()  # Output: Hello, World!
```

```
Hello, World!
```

In this example, `inner_function` is a closure that remembers the value of `text` from the scope of `outer_function`.
Let’s break it down:

1. Defining outer_function:
   - outer_function is a regular function that takes one argument, `text`.
   - inside `outer_function`, we define another function, `inner_function`.

2. Defining inner_function:
   - inner_function is defined within the scope of outer_function.
   - it doesn't take any arguments itself, but it accesses the `text` variable, which is defined in the enclosing scope of `outer_function`.
   - inner_function simply prints the value of `text`.

3. Returning inner_function:
   - At the end of outer_function, we return inner_function without parentheses. This is crucial because we're returning the function itself, not the result of calling the function.

4. Creating `my_func:
   - When we call outer_function('Hello, World!'), it executes and returns the inner_function.
   - We assign this returned function to my_func. Now, my_func is essentially inner_function.

5. Calling my_func:
   - When we call my_func(), we're actually calling inner_function that was returned by outer_function.
   - Even though the execution context (scope) of outer_function has finished, inner_function remembers the value of `text` due to the closure. Hence, it prints "Hello, World!".

4.2 DECORATORS

Decorators are a very powerful and useful tool in Python, allowing you to modify the behavior of a function or class. Decorators allow you to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

1. Key Aspects of Decorators:
1. Functions as Arguments:
Since functions are first-class citizens in Python, you can pass functions as arguments to other functions.

2. Enhancing Functions:
A decorator takes in a function, adds some functionality, and returns it.

3. Syntax:
 - Decorators use the `@` symbol before the name of the decorator function.
 - Place it above the definition of the function to be decorated.

Example of a Decorator:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

This example wraps the `say_hello` function inside the `wrapper` function, adding messages before and after its execution.

4. When to Use Decorators:
- Logging, enforcing access control and authentication, instrumentation, and adding functionalities in frameworks (like Flask or Django) are common use cases.
- Decorators are extensively used in web frameworks and for modifying the behavior of class methods.

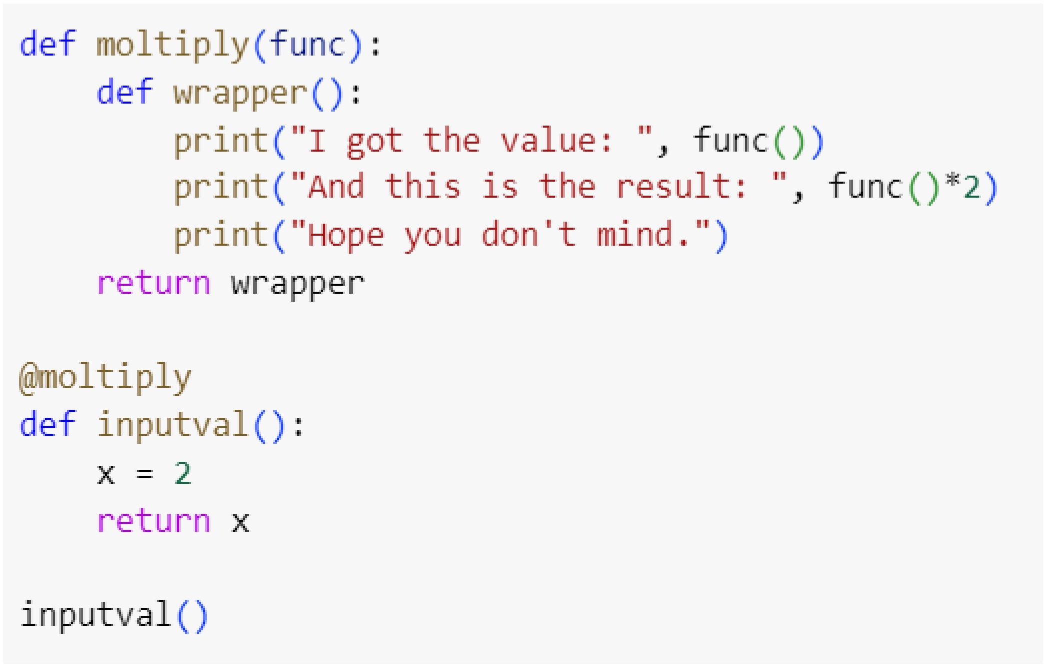
4.2.1 Deep Understanding of Decorators

Decorators in Python are used to modify or extend the behavior of functions or methods without permanently modifying them. They are represented by the `@` symbol and are placed above the function definition.

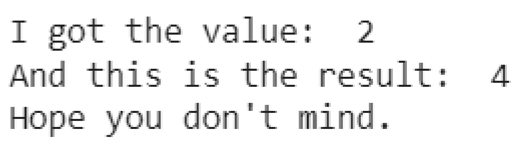
- Decorators can take arguments and can decorate functions with any number of arguments.
- They can also be used to modify the return value of a function.
- Python provides support for class decorators and function decorators.
- The key to understanding decorators is to focus on how functions are objects in Python and how they can be passed around and manipulated.

Closures and decorators are advanced concepts and are particularly useful for high-level abstraction and meta-programming.

Other example:



```
def moltiply(func):
    def wrapper():
        print("I got the value: ", func())
        print("And this is the result: ", func()*2)
        print("Hope you don't mind.")
    return wrapper

@moltiply
def inputval():
    x = 2
    return x

inputval()
```



```
I got the value:  2
And this is the result:  4
Hope you don't mind.
```

4.2.2 Handling Multiple Arguments in Decorators

When you create a decorator that needs to handle functions with more than one argument, you can use `*args` and `**kwargs` in the inner wrapper function. This allows the decorator to accept an arbitrary number of positional and keyword arguments.

1. Basic Structure of a Decorator with Multiple Arguments:

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        # Do something before the original function is called
        result = func(*args, **kwargs)
        # Do something after the original function is called
        return result
    return wrapper
```

In this structure:
- `func` is the function being decorated.
- `*args` is a tuple of positional arguments.
- `**kwargs` is a dictionary of keyword arguments.

Example of a Decorator with some arguments:
- Let's create a decorator that prints the arguments passed to a function:

```python
def print_args(func):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return func(*args, **kwargs)
    return wrapper

@print_args
def my_function(a, b, c=3):
    return a + b + c

print(my_function(1, 2, c=6))
```

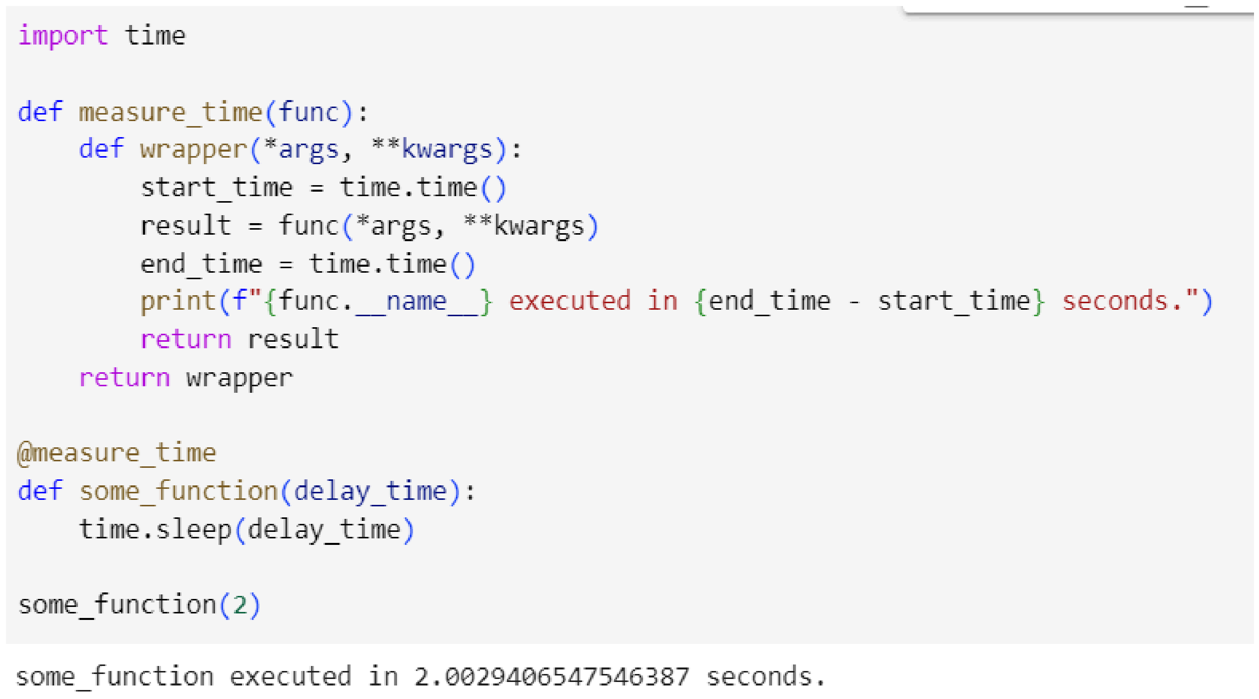
```
Arguments: (1, 2) {'c': 6}
9
```

In this example, `print_args` is a decorator that prints out the arguments (`args` and `kwargs`) passed to `my_function`. The `my_function` can accept any number of positional and keyword arguments.

4.2.3 Advanced Use Cases

Decorators can be used for a variety of advanced use cases, such as:
- Logging function calls.
- Enforcing access control and authentication.
- Caching the results of function calls.
- Measuring execution time.

1. Creating a Decorator to Measure Execution Time
As a practical example, let's create a decorator that measures the execution time of a function:



```python
import time

def measure_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} executed in {end_time - start_time} seconds.")
        return result
    return wrapper

@measure_time
def some_function(delay_time):
    time.sleep(delay_time)

some_function(2)
```

```
some_function executed in 2.0029406547546387 seconds.
```

In this example, the `measure_time` decorator calculates the time taken to execute the `some_function` function.

Decorators are a powerful feature in Python that can greatly enhance and modify the behavior of functions.

PRACTICAL TASKS

- Task 1: Write a lambda function to compute the square of a number.
- Task 2: Use `map()` to apply a function over a list of numbers.
- Task 3: Use `filter()` to filter out elements from a list based on a condition.
- Task 4: Use `reduce()` to reduce a list of numbers into a single cumulative value.
- Task 5: Create a simple decorator that logs the execution time of a function.

1. Lambda Function for square of a number

```python
square = lambda x : x*x
number = int(input("Insert your number: "))
print(square(number))
```

```
Insert your number: 4
16
```

2. Map Function over a list of numbers

```python
numbers = [1,4,5,6]
frac = map(lambda x: x/2, numbers)
print(list(frac))
```

```
[0.5, 2.0, 2.5, 3.0]
```

3. Filter Function over a list of numbers

```python
numbers = [2, 5, 6, 7]
evens = filter(lambda x: x%2==0, numbers)
print(list(evens))
```

```
[2, 6]
```

4. Reduce Function over a list of numbers

```
import functools
numbers = [3, 6, 7, 8]
prods = functools.reduce(lambda x,y: x*y, numbers)
print(prods)
```

```
1008
```

5. Decorators to log the execution time of a function

```
import time
def calc_time(func):
  def wrapper(*args, **kwargs):
    start_t = time.time()
    func(*args, **kwargs)
    end_t = time.time()
    print(f"{func.__name__} launched in: {end_t - start_t} seconds")
  return wrapper

@calc_time
def my_func(delay_time):
    time.sleep(delay_time)

my_func(2)
```

```
my_func launched in: 2.000751495361328 seconds
```

5. OBJECT-ORIENTED PROGRAMMING (OOP)

OOP Concepts

1. Study Material:

- Understand the basic principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction.
- Learn how to define classes and create objects in Python.
- Explore class methods, instance methods, static methods, and property decorators.

2. Practical Tasks:

- Task 1: Define a Python class named `Rectangle`. It should have properties like length and width. Define methods to calculate the area and perimeter of the rectangle.
- Task 2: Create a class `Circle` with the property radius and methods to calculate the area and circumference.
- Task 3: Implement Inheritance by creating a class `Square` that inherits from `Rectangle` and uses its methods.
- Task 4: Demonstrate the use of class methods and static methods in any of the above classes.
- Task 5: Explore the concept of polymorphism by creating a function that can take either a Circle or a Rectangle object and display the area.

Guidelines:

- Focus on how classes and objects are defined and used in Python.
- Practice writing methods and understand the difference between instance, class, and static methods.
- Implement inheritance and understand how it can be used to create a hierarchy of classes.

STUDY MATERIAL

1.0 WHAT'S OBJECT-ORIENTED PROGRAMMING (OOP)?

- Object-Oriented Programming is a programming paradigm that uses "objects" – data structures consisting of data fields and methods – and their interactions to design applications and computer programs.

This module will strengthen your understanding of OOP concepts in Python. OOP is a critical skill in software development, enabling you to write more modular, scalable, and maintainable code.

2.0 INTRODUCTION TO CLASSES AND OBJECTS

- In Python, a class is a blueprint for creating objects. Objects have member variables and have behavior associated with them. In python, a class is created by the keyword class.

Defining a Class

A class is defined in Python using the class keyword, followed by the class name and a colon. Inside the class, methods (functions) and attributes (variables) are defined.

```python
class MyClass:
    def __init__(self, attribute):
        self.attribute = attribute
    def my_method(self):
        return f"Value of attribute is {self.attribute}"
```

__init__ method is a special method called a **constructor**, used for initializing an instance of the class. self refers to the current instance of the class.
my_method is an example of a method in the class, which can perform operations using the class attributes.

Creating an Object

An object is an instance of a class. Once a class is defined, you can create objects with the defined properties and methods.

```python
my_object = MyClass("Hello")
print(my_object.my_method())
```

```
Value of attribute is Hello
```

3.0 ENCAPSULATION

- Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also restricts direct access to some of the object's components, which is a means of preventing accidental interference and misuse of the methods and data.

Key Points of Encapsulation:
- Data Hiding:
Private variables and methods can be created by prefixing them with double underscores __. This prevents them from being accessed directly from outside the class.

- Getter and Setter Methods: These are used to access or modify private variables. They provide a controlled way of accessing and changing the data.

Example of Encapsulation in Python:

```
class EncapsulatedObject:
    def __init__(self):
        self.public_data = "Public"
        self.__private_data = "Private"

    def get_private_data(self):
        return self.__private_data

    def set_private_data(self, value):
        self.__private_data = value

# Creating an object
obj = EncapsulatedObject()

# Accessing public data
print(obj.public_data)
# Accessing private data
print(obj.get_private_data())

# Modifying private data
obj.set_private_data("New Value")
print(obj.get_private_data())
```

```
Public
Private
New Value
```

4.0 INHERITANCE

- Inheritance allows us to define a class that inherits all the methods and properties from another class. The new class is called a derived (or child) class, and the class from which it inherits is called the base (or parent) class.

Key Points of Inheritance:
- Reusability: Inheritance supports the concept of “reusability”, i.e., we can add more features to an existing class without modifying it.
- Extensibility: It is easy to extend the base class logic in a derived class.

- Hierarchies: Create hierarchical classifications.

Example of Inheritance in Python:

```python
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

# Derived class
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Derived class
class Cat(Animal):
    def speak(self):
        return "Meow!"

dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())
print(cat.speak())
```

```
Woof!
Meow!
```

In this example, Dog and Cat classes inherit from the Animal class and override the speak method.

4.1 Encapsulation and Inheritance in Practice

In practice, encapsulation is used to hide the internal representation, or state, of an object from the outside. This is known as data hiding. Inheritance, on the other hand, is used to model a relationship between two classes, where one extends the functionality of the other.

These concepts are crucial for creating well-structured and maintainable code in object-oriented programming.

5.0 POLYMORPHISM

- Polymorphism in OOP refers to the ability of different classes to be treated as instances of the same class through inheritance. It allows functions to use objects of different types at different times, depending on the type of object it is at the moment.

Key Points of Polymorphism:

- Same Interface, Different Implementation: Different classes can have methods with the same name but different functionalities.
- Common Use in Inheritance: Typically used with inheritance where a child class can override a method of the parent class.
- "Many Forms": The word 'polymorphism' means having many forms. In the context of OOP, it means the same function name can be used for different types.

Example of Polymorphism in Python:

```python
class Bird:
    def fly(self):
        print("Some birds can fly.")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow can fly.")

class Ostrich(Bird):
    def fly(self):
        print("Ostrich cannot fly.")

# Polymorphism in action
for bird in [Sparrow(), Ostrich()]:
    bird.fly()
```

```
Sparrow can fly.
Ostrich cannot fly.
```

In this example, the fly method is polymorphic. Different bird types (Sparrow and Ostrich) implement it differently, but the interface is consistent.

6.0 ABSTRACTION

- Abstraction in OOP is the concept of hiding the complex reality while exposing only the necessary parts. It is more about creating a simple model that represents the more complex parts of a problem, which in the context of programming means creating simple classes to represent complex systems.

Key Points of Abstraction:
- Abstract Classes: These are classes that contain one or more abstract methods. An abstract method is a method that is declared but contains no implementation. Abstract classes cannot be instantiated.

- Enforces a Contract: By defining an abstract base class, you define a common API for a set of subclasses. This ensures a certain level of conformity in subclasses.

- Use of abc Module: In Python, abstract classes are achieved through the module abc (Abstract Base Classes).

Example of Abstraction in Python:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Cannot create an instance of Shape
# shape = Shape()  # This will raise an error

# Can create an instance of Rectangle
rect = Rectangle(3, 4)
print(rect.area())
```

```
12
```

In this example, Shape is an abstract class that defines a contract (an area method), and Rectangle implements this method.

6.1 Polymorphism and Abstraction in Practice

- Polymorphism is used when multiple classes are related by inheritance and they share a common structure or method names but implement them differently.
Abstraction aims at reducing complexity by hiding the technical complexity and showing only the essential features of the object.

6.2 WHEN TO USE ABSTRACTION

- Abstraction becomes particularly useful in scenarios where you have a set of related classes with shared behavior, but each class has its own specific way of implementing that behavior. The key is when you have multiple objects that should follow the same interface or structure but have different internal implementations.

Abstraction is most useful when:
- You're Designing a Large or Complex System:
In a complex system, abstraction helps manage complexity by hiding the detailed implementation and exposing only the necessary parts.

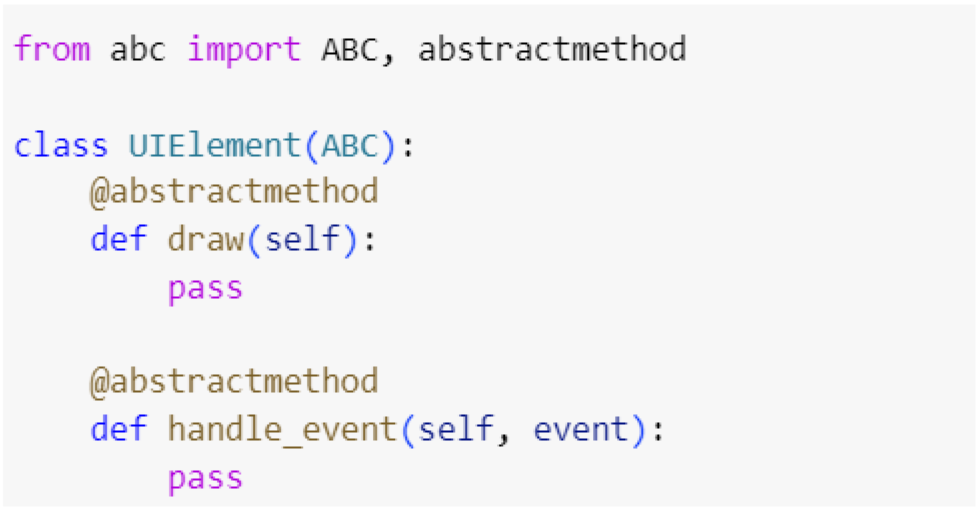
- There's a Need for a Common Interface for Different Implementations:
When you have different objects that need to share a common interface, abstraction ensures that each object adheres to a certain contract.

- Promoting Code Reusability and Flexibility:
Abstract classes allow you to define a blueprint for other classes, promoting reusability.

Real-World Example: A GUI Toolkit

- Imagine you're designing a GUI (Graphical User Interface) toolkit. In this toolkit, you need to handle different types of UI elements like buttons, text boxes, and checkboxes. Each of these elements shares some common behaviors (like being drawn on the screen, responding to user input) but also have unique behaviors and attributes.

Abstract Base Class for UI Elements



```python
from abc import ABC, abstractmethod

class UIElement(ABC):
    @abstractmethod
    def draw(self):
        pass

    @abstractmethod
    def handle_event(self, event):
        pass
```

Derived Classes for Specific UI Elements

```python
class Button(UIElement):
    def draw(self):
        print("Drawing a button")

    def handle_event(self, event):
        print(f"Button clicked with event {event}")

class TextBox(UIElement):
    def draw(self):
        print("Drawing a text box")
    def handle_event(self, event):
        print(f"Text box edited with event {event}")
```

Usage

```python
elements = [Button(), TextBox()]

for element in elements:
    element.draw()
    element.handle_event("click")
```

```
Drawing a button
Button clicked with event click
Drawing a text box
Text box edited with event click
```

- In this scenario, the UIElement abstract class is useful as it defines a common interface (draw and handle_event) that all UI elements must follow. Each specific UI element (like Button or TextBox) then implements these methods in its own way. This design allows you to write code that is agnostic to the specific types of UI elements, making it easier to add new types of UI elements in the future without modifying existing code.

Key Takeaway
- Abstraction is not always necessary for simple, straightforward problems. However, in larger and more complex systems, especially where you have multiple objects that should follow the same interface but have different internal workings, abstraction becomes invaluable. It allows you to write more modular, flexible, and maintainable code.

7.0 INSTANCE METHODS

- Instance methods are the most common type of methods in Python classes. They take a specific instance of the class as their first argument, traditionally named self.

Key Points:
- Access to Instance Attributes: They can modify object instance state by accessing attributes attached to self.
- Calling: Called on an instance of the class.

Example:

```python
class MyClass:
    def __init__(self, value):
        self.value = value

    def instance_method(self):
        return f"Instance method called, value is: {self.value}"
```

8.0 CLASS METHODS

Class methods take a class as their first argument, traditionally named cls. They can't modify object instance state but can modify class state that applies across all instances of the class.

Key Points:
- Decorator: Defined with the @classmethod decorator.
- Access to Class Attributes: Can modify class state that applies to all instances.

Example:

```python
class MyClass:
    count = 0

    @classmethod
    def increment_count(cls):
        cls.count += 1

MyClass.increment_count()
print(MyClass.count)
```

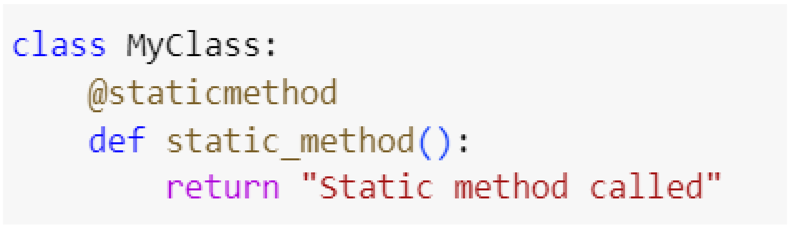
```
1
```

9.0 STATIC METHODS

- Static methods don't take a class or instance reference as their first argument. They behave like regular functions but belong to the class's namespace.

Key Points:
- Decorator: Defined with the @staticmethod decorator.
- No Access to Instance or Class Attributes: They work like regular functions but are enclosed in the class's scope.

Example:



```python
class MyClass:
    @staticmethod
    def static_method():
        return "Static method called"
```

10.0 Property Decorators

- Property decorators allow you to use getters and setters in order to add logic around getting and setting a value.

Key Points:
- Decorator: Use @property to define a getter, and @<property_name>.setter to define a setter.
- Encapsulation: Useful for encapsulating private attributes by providing getters and setters.

Example:

```
class MyClass:
    def __init__(self, value):
        self._value = value
    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        if value < 0:
            raise ValueError("Value cannot be negative")
        self._value = value

obj = MyClass(5)
print(obj.value)
obj.value = 10
print(obj.value)
```

```
5
10
```

In this example, value is a property object, and we define how to get and set its value with the @property decorator and the setter.
Please note both @property and @value.setter have a “value” function, where in this case the value.setter checks if the value is negative to raise an error before passing it to the property decorator.

11.0 Combining These Concepts

- All these methods offer different ways to interact with class and instance data. They allow you to organize your class's functionality in a way that's both easy to use and maintain. Understanding when and how to use these different types of methods will help you design more effective and encapsulated classes in Python.

PRACTICAL TASKS

- Task 1: Define a Python class named `Rectangle`. It should have properties like length and width. Define methods to calculate the area and perimeter of the rectangle.
- Task 2: Create a class `Circle` with the property radius and methods to calculate the area and circumference.
- Task 3: Implement Inheritance by creating a class `Square` that inherits from `Rectangle` and uses its methods.
- Task 4: Demonstrate the use of class methods and static methods in any of the above classes.
- Task 5: Explore the concept of polymorphism by creating a function that can take either a Circle or a Rectangle object and display the area.

1. Class Rectangle

```python
class Rectangle():
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return f"The area of the rectangle is {self.length*self.width}"

    def perimeter(self):
        return f"The perimeter of the rectangle is {self.length*2+self.width*2}"

wid = int(input("Enter width: "))
leng = int(input("Enter length: "))
my_rect = Rectangle(leng, wid)
print(my_rect.perimeter())
print(my_rect.area())
```

```
Enter width: 8
Enter length: 7
The perimeter of the rectangle is 30
The area of the rectangle is 56
```

2. Class Circle

```python
class Circle():
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return f"The area of the circle is {3.14*pow(self.radius,2)}"

    def circonference(self):
        return f"The circonference of the rectangle is {2*self.radius*3.14}"

radi = int(input("Enter radius: "))
my_circle = Circle(radi)
print(my_circle.circonference())
print(my_circle.area())
```

```
Enter radius: 4
The circonference of the rectangle is 25.12
The area of the circle is 50.24
```

3. Inheritance Class Square

```python
class Rectangle():
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return f"The area of the rectangle is {self.length*self.width}"

    def perimeter(self):
        return f"The perimeter of the rectangle is {self.length*2+self.width*2}"

class Square(Rectangle):
    def area(self):
        return f"The area of the square is {self.length*self.width}"

    def perimeter(self):
        return f"The perimeter of the square is {self.length*2+self.width*2}"


wid = int(input("Enter width: "))
leng = int(input("Enter length: "))
if wid==leng:
    print("This is a square")
    my_square = Square(leng,wid)
    print(my_square.perimeter())
    print(my_square.area())
else:
    print("This is a rectangle")
    my_rect = Rectangle(leng, wid)
    print(my_rect.perimeter())
    print(my_rect.area())
```

```
Enter width: 3
Enter length: 4
This is a rectangle
The perimeter of the rectangle is 14
The area of the rectangle is 12
```

4. Class and Static methods

```python
class Square:
    def __init__(self, side):
        self.side = side
    @staticmethod
    def static_method():
        return f"This is a static square"

side = int(input("Enter side: "))
my_square = Square(side)
print(Square.static_method())
```

```
Enter side: 5
This is a static square
```

5. Polymorphism Circle or Rectangle or Square

```python
from abc import ABC, abstractmethod

class UIType(ABC):
    @abstractmethod
    def area(self):
        pass
    def perimeter(self):
        pass
    def circonference(self):
        pass


class Circle(UIType):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return f"The area of the circle is {3.14*pow(self.radius,2)}"

    def circonference(self):
        return f"The circonference of the rectangle is {2*self.radius*3.14}"

class Rectangle(UIType):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return f"The area of the rectangle is {self.length*self.width}"

    def perimeter(self):
        return f"The perimeter of the rectangle is {self.length*2+self.width*2}"
```

```python
class Square(Rectangle):
    def area(self):
        return f"The area of the square is {self.length*self.width}"

    def perimeter(self):
        return f"The perimeter of the square is {self.length*2+self.width*2}"


quest = str(input("Is it a circle? Y / n: "))
if quest == "Y":
    radi = int(input("Enter radius: "))
    my_circle = Circle(radi)
    print(my_circle.circonference())
    print(my_circle.area())
else:
    wid = int(input("Enter width: "))
    leng = int(input("Enter length: "))
    if wid==leng:
        print("This is a square")
        my_square = Square(leng,wid)
        print(my_square.perimeter())
        print(my_square.area())
    else:
        print("This is a rectangle")
        my_rect = Rectangle(leng, wid)
        print(my_rect.perimeter())
        print(my_rect.area())
```

```
Is it a circle? Y / n: Y
Enter radius: 5
The circonference of the rectangle is 31.400000000000002
The area of the circle is 78.5
```

6. MODULES AND PACKAGES

Modules and Packages

1. Study Material:

- Modules: Learn what Python modules are, how to create them, and how to use them in your Python programs.
- Packages: Understand what Python packages are, how they differ from modules, and how to create your own packages.

2. Practical Tasks:

- Task 1: Create a simple Python module with a few functions and classes, then import and use them in a different Python file.
- Task 2: Organize multiple Python scripts into a package, use __init__.py, and understand how to import from this package.
- Task 3: Explore the use of built-in modules like math, datetime, and others.
- Task 4: (Optional) Publish a simple package to PyPI (Python Package Index) for community use.

Guidelines:

- Focus on understanding the namespace and scope when working with modules and packages.
- Learn the difference between module-level functions, class definitions, and variable declarations.
- Understand the use of __init__.py files in Python packages.

STUDY

1.0 WHAT ARE MODULES AND PACKAGES?

- Modules and packages are key to effective programming in Python, especially as your projects grow larger and more complex. This module will give you a solid foundation in working with Python's modular programming capabilities, which is essential for building larger and more complex applications.

- A Python module is simply a Python file that contains a collection of functions, classes, variables, and runnable code, or a mixture of these. Modules are a way to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

Key Points:

- Reusability: Modules can be reused across Python scripts.
- Namespace: Each module has its own namespace, which helps avoid collisions between identifiers.
- Standard Library: Python comes with a library of standard modules.

2.0 CREATING A PYTHON MODULE

- Creating a module is as simple as creating a Python file. For example, let's create a module named my_module.py.

Example: my_module.py

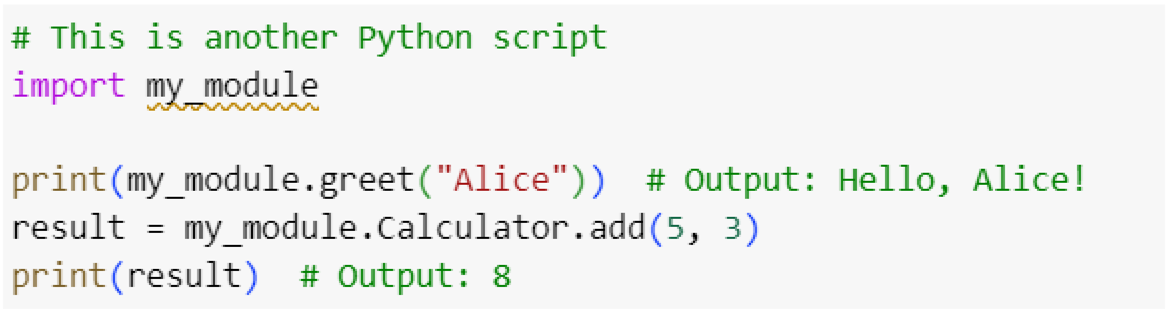
```
# This is my_module.py
def greet(name):
    return f"Hello, {name}!"

class Calculator:
    @staticmethod
    def add(x, y):
        return x + y
```

In this file, we have a function greet and a class Calculator with a static method “add”.

2.1 USING A PYTHON MODULE

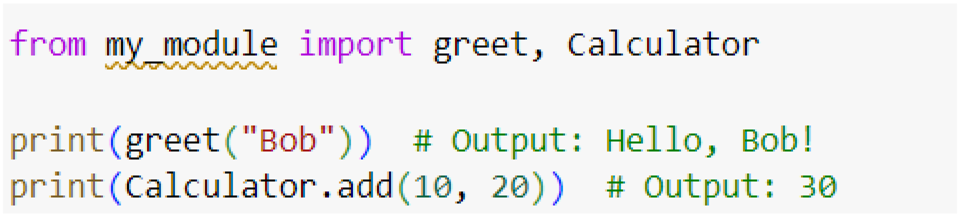
- To use the module, you simply import it into your Python script. This can be done with the import statement.

Example: Using my_module.py



```
# This is another Python script
import my_module

print(my_module.greet("Alice"))  # Output: Hello, Alice!
result = my_module.Calculator.add(5, 3)
print(result)  # Output: 8
```

You can also import specific attributes from a module using the from keyword.



```
from my_module import greet, Calculator

print(greet("Bob"))  # Output: Hello, Bob!
print(Calculator.add(10, 20))  # Output: 30
```

2.2 PYTHON MODULE SEARCH PATH

When you import a module, Python searches for the module in the following locations:
- The directory from which the input script was run, or the current directory if interactive.
- The list of directories contained in the PYTHONPATH environment variable, if set.
- The list of standard locations that Python searches by default, such as /usr/lib/python.

Tips for Module Creation

- Keep your modules small and focused on a single purpose.
- Use descriptive names for module files.
- Remember that any Python file with a .py extension can be a module.

3.0 WHAT ARE PYTHON PACKAGES?

- A Python package is a way of organizing related Python modules into a file hierarchy with a distinct namespace. Essentially, a package is a directory that contains a special file named `__init__.py` along with one or more module files and possibly sub-packages.

Key Points:
1. Directory Structure
- A package is represented by a directory, with each sub-directory being a sub-package if it contains an `__init__.py` file.
2. __init__.py
- This file can be empty, but its presence tells Python that the directory is a Python package, not just a folder.
3. Namespace Management
- Packages allow you to use the same function or class name in different modules without conflict.

3.1 CREATING A PYTHON PACKAGE

- To create a package, you need to organize your module files into a directory structure and add __init__.py files.

Example: Creating a Simple Package
- Let's create a simple package named `mypackage` with two modules: `module1.py` and `module2.py`.

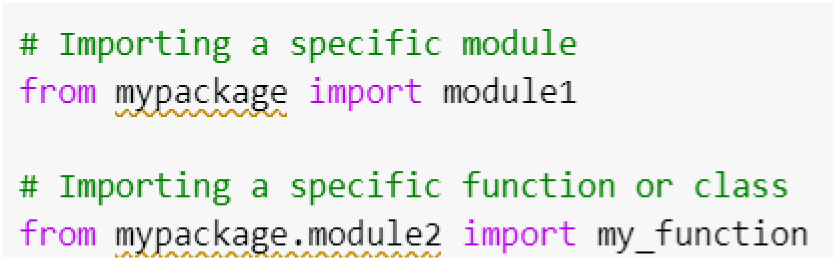
```
mypackage/
│   __init__.py
│   module1.py
│   module2.py
```

- __init__.py = This file makes `mypackage` a Python package. It can be empty or can contain valid Python code.
- module1.py = This is a regular Python module file.
- module2.py = Another Python module file.

3.2 USING A PYTHON PACKAGE

- To use the package, you import modules from the package using dot (`.`) notation.

Example: Importing from `mypackage`



```
# Importing a specific module
from mypackage import module1

# Importing a specific function or class
from mypackage.module2 import my_function
```

3.3 ORGANIZING LARGER PACKAGES

- For larger packages, you might have a more nested structure with sub-packages. Each sub-package should also have its own __init__.py file.

Example: Nested Package Structure

```
mypackage/
│   __init__.py
│   module1.py
│
└───subpackage1/
│   │   __init__.py
│   │   submodule1.py
│
└───subpackage2/
    │   __init__.py
    │   submodule2.py
```

To use `submodule1` from `subpackage1`, you would do:

```
from mypackage.subpackage1 import submodule1
```

Tips for Working with Packages

- Use packages to organize modules that provide related functionality or belong to the same domain.
- Keep the directory structure and import statements clean and intuitive.
- Use relative imports (`from . import some_module`) within the package for better modularity.

Now, you can try creating your own Python package following this structure. Remember, packages are incredibly useful for creating well-structured, scalable, and maintainable Python applications.

PRACTICAL TASKS

- Task 1: Create a simple Python module with a few functions and classes, then import and use them in a different Python file.
- Task 2: Organize multiple Python scripts into a package, use __init__.py, and understand how to import from this package.
- Task 3: Explore the use of built-in modules like math, datetime, and others.
- Task 4: (Optional) Publish a simple package to PyPI (Python Package Index) for community use.

1. Create and import a module of my creation

My firstmodule.py:

```python
def rectangles_area(width, length):
    return width*length
```

Using my firstmodule:

```python
import firstmodule

print(firstmodule.rectangles_area(4,3))
```

2. Create a simple Package with __init__.py

Folder/Package:

```
mypackage/
  __init__.py
  firstmodule.py
```

Python:

```python
from mypackage import firstmodule.py
print(firstmodule.rectangles_area(4,3))
```

3. Other built-in modules

Math Module:

```python
import math
radius = int(input("insert radius: "))
print(f"Area of circle is: {math.pi * radius ** 2}")
```

```
insert radius: 5
Area of circle is: 78.53981633974483
```

Datetime Module:

```python
import datetime
print(f"The current time is: {datetime.datetime.now()}")
```

```
The current time is: 2024-02-02 11:40:12.323141
```

7. CREATE A GITHUB PROJECT

Let's design a Python project that encompasses various aspects we've covered, such as basic programming constructs, OOP, modules, and packages.

Project Idea: Simple Contact Book Application

Let's create a simple "Contact Book" application. This application will allow users to add, view, delete, and search for contacts. It's a great project to demonstrate basic Python skills, OOP concepts, and how to structure a Python package.

7.1 PROJECT STRUCTURE

1. Package Structure:

```
contactbook/
|   __init__.py
|   main.py
|   contact.py
|   contact_manager.py
```

2. Module Descriptions:

-contact.py = Contains a `Contact` class representing a contact with attributes like name, phone number, and email.

- contact_manager.py = Contains a `ContactManager` class that manages a list of contacts. It includes methods to add, view, delete, and search for contacts.
- main.py = The main script where the application logic and user interface are handled.

3. Step-by-Step Development:

1. Develop the `Contact` Class (contact.py):
   - This class represents a single contact with attributes like `name`, `phone`, and `email`.
   - Include a method to display the contact's information.

2. Develop the `ContactManager` Class (contact_manager.py):
   - This class manages a collection of `Contact` objects.
   - Implement methods for adding, removing, searching, and listing contacts.

3. Create the Application Logic (main.py):
   - Implement the user interface in the console.
   - Allow users to interact with the contact book (add, view, delete, search for contacts).

4. Testing:
   - Test each component to ensure it works as expected.
   - Perform end-to-end testing of the application.

5. Documentation:
  - Add comments and document your code.
  - Write a `README.md` file explaining what the project is, how to run it, and its features.

3. Example Code Snippet:
Here's a snippet to start with for your `Contact` class:

contact.py:

```
class Contact:
    def __init__(self, name, phone, email):
        self.name = name
        self.phone = phone
        self.email = email

    def display_contact(self):
        return f"Name: {self.name}, Phone: {self.phone}, Email: {self.email}"
```

4. Next Steps
- Develop the `ContactManager` and `main.py` based on this structure.
- Remember to handle exceptions and edge cases.
- Once the project is complete and tested, push it to a GitHub repository.

5. Publishing to GitHub
1. Create a GitHub Repository.
2. Initialize your project folder as a Git repository.
3. Add your project files to the repository.
4. Commit your changes and push them to GitHub.

This project is a great way to showcase your Python skills. It covers a wide range of topics and provides a practical application that can be further expanded.

PYTHON FILES

Folder: contactbook
1) __init__.py

2) Contact.py

```python
class Contact:
    def __init__(self, name, phone, email):
        self.name = name
        self.phone = phone
        self.email = email

    def display_contact(self):
        return f"Name: {self.name}, Phone: {self.phone}, Email: {self.email}"
```

3) Contact_manager.py

```python
from contact import Contact

class ContactManager:
    def __init__(self):
        self.contacts = []

    def add_contact(self, name, phone, email):
        new_contact = Contact(name, phone, email)
        self.contacts.append(new_contact)

    def remove_contact(self, name):
        for contact in self.contacts:
            if contact.name == name:
                self.contacts.remove(contact)
                return f"Contact {name} removed."
        return f"Contact {name} not found."

    def search_contact(self, name):
        for contact in self.contacts:
            if contact.name == name:
                return contact.display_contact()
        return f"Contact {name} not found."

    def list_contacts(self):
        if not self.contacts:
            return "Contact list is empty."
        return '\n'.join([contact.display_contact() for contact in self.contacts])

# Example Usage
# manager = ContactManager()
# manager.add_contact("John Doe", "123456789", "john@example.com")
# print(manager.list_contacts())
```

4) main.py

```python
from contact_manager import ContactManager

def main_menu():
    print("Contact Book Application")
    print("1. Add a new contact")
    print("2. View all contacts")
    print("3. Search for a contact")
    print("4. Delete a contact")
    print("5. Exit")
    choice = input("Enter choice: ")
    return choice

def main():
    manager = ContactManager()

    while True:
        user_choice = main_menu()

        if user_choice == '1':
            name = input("Enter name: ")
            phone = input("Enter phone number: ")
            email = input("Enter email: ")
            manager.add_contact(name, phone, email)

        elif user_choice == '2':
            print(manager.list_contacts())

        elif user_choice == '3':
            name = input("Enter the name to search: ")
            print(manager.search_contact(name))

        elif user_choice == '4':
            name = input("Enter the name to delete: ")
            print(manager.remove_contact(name))

        elif user_choice == '5':
            print("Exiting the application.")
            break

        else:
            print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    main()
```

8. ACKNOWLEDGEMENTS

-This course was created with the help of GPT 4 - PYTHON MASTER COACH.
Most of the theory was provided by GPT, while most of the code was provided by me.
Please feel free to contact me for anything.

Andrea Gobbetti

You can find this course on kobo ebook:
https://www.kobo.com/it/it/ebook/python-refinement-course