Python Handbook

“ A comprehensive notebook documenting my Python learning journey, offering all the essentials in Python “

By
Ahmad Jawabreh

March, 2022

What prerequisites should be in place before commencing the course?

1- Code Editor Selection: Numerous options are available, with the most renowned being VS Code, or alternatively, you can opt for the PyCharm IDE.

2- Python Installation and Configuration: Begin by installing and setting up Python from the official website.

3- VS Code Python Extension: If you choose VS Code as your code editor, install the Python extension to enhance your development environment.

4- Prerequisite Programming Knowledge: Ensure you possess basic programming knowledge before diving into the Python course.

5- Command Line Proficiency: Familiarize yourself with command line operations, as it is a beneficial skill for Python development. Some Command Line notes are attached in this repository.

6- Please review the file named "notes.txt" in this repository before commencing the course.

1.0 Print Statement:

The print statement is commonly employed as a useful means to display information on the debugging console. However, it is crucial to emphasize that the use of print statements is restricted when working on production code.

So, how can we print some information on the debugging console?

Example: print(“ Hello World”)
By running the app you’ll be able to see the “ Hello World

2.0 Implicit Line Continuation (Optional Semicolon):

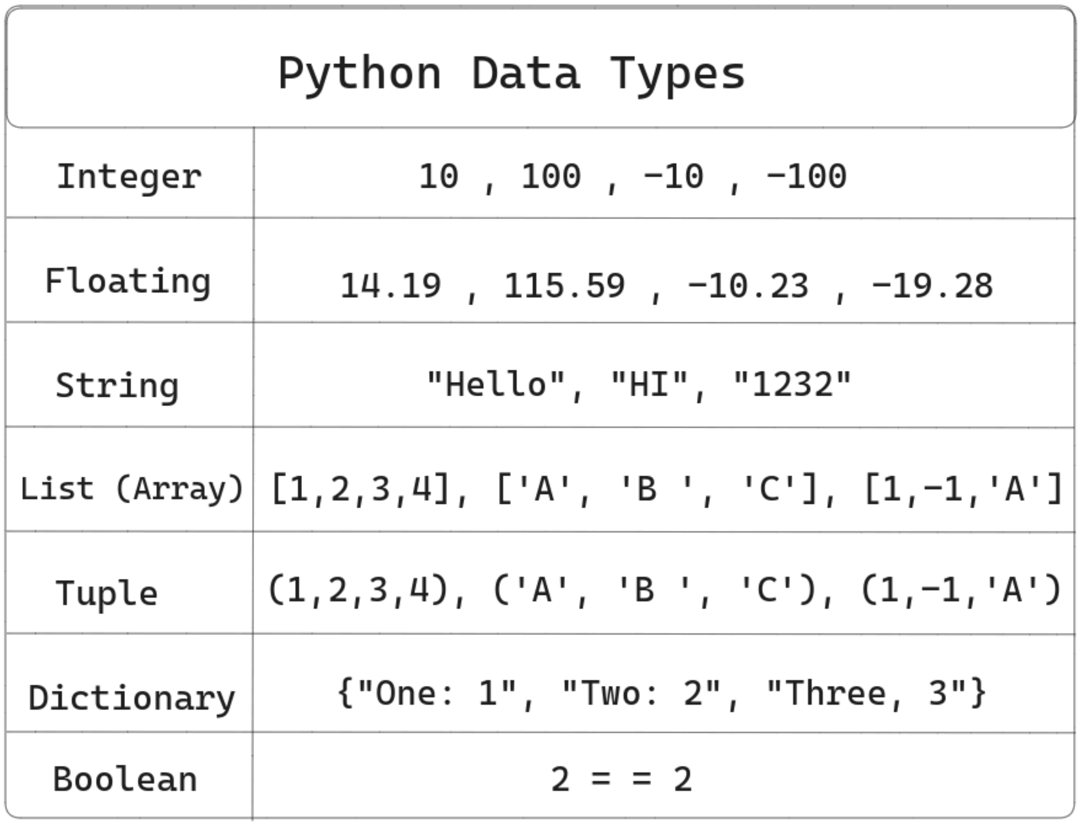
If you’ve noticed in the last example that we are not using semicolon at the end of the line as we used to do, because Python's avoidance of semicolons in its code is a characteristic of a programming language feature known as "optional semicolons" or "implicit line continuation." In Python, the end of a line generally marks the end of a statement, and the use of semicolons to terminate statements is optional. The interpreter relies on indentation to determine the structure of the code, making semicolons unnecessary for line termination in most cases. This approach enhances readability and contributes to Python's clean and concise syntax.

3.0 Comments:

In Python, the hash sign is employed to comment on a specific line. For instance: # This is a single-line comment If you are using VS Code, you can conveniently comment on a specific line using the shortcut CTRL+/.

Note: It's essential to note that Python does not have explicit syntax for multiple-line comments. While some use triple quotes """ HERE """ for this purpose, it's crucial to recognize that this construct is, in fact, a string not assigned to any variable, making it an unconventional method for commenting in Python.

4.0 Data Types:



Python Data Types	
Integer	10 , 100 , -10 , -100
Floating	14.19 , 115.59 , -10.23 , -19.28
String	"Hello", "HI", "1232"
List (Array)	[1,2,3,4], ['A', 'B ', 'C'], [1,-1,'A']
Tuple	(1,2,3,4), ('A', 'B ', 'C'), (1,-1,'A')
Dictionary	{"One: 1", "Two: 2", "Three, 3"}
Boolean	2 = = 2

figure 1: Data Types in Python

- int (Integer): Any positive or negative number without any decimal or fractional part is integer

- float (Floating): Any positive or negative number with any decimal or fractional part is integer

- str (String): String type for representing text, e.g., "hello", 'Python'

- List: Ordered, mutable sequence of elements that can hold any type of data

- Tuple: Ordered, immutable sequence that can hold any type of data

- dict (Dictionary): A collection of key-value pairs

- bool: Boolean type representing True or False

4.1 List VS Tuple:

- List: Lists are mutable, meaning you can modify their elements, add new elements, or remove existing ones after the list is created. Use lists when you have a collection of items that may need to be modified, such as adding or removing elements. Lists are suitable for sequences where the length may change during the program's execution.

- Tuple: Tuples are immutable, and once a tuple is created, you cannot change, add, or remove elements. Use tuples when you want to create an immutable and ordered collection. Tuples are useful for representing fixed collections of items, like coordinates or settings, where you don't want the values to be changed accidentally.

5.0 Variables:

Python uses the concept of dynamic typing or duck type which is used also by other programming languages such as PHP and Dart. In Python, variables are dynamically typed, which means their type is determined at runtime based on the value assigned to them. Unlike statically-typed languages where you explicitly declare the variable type, Python allows more flexibility by inferring the type based on the assigned value. This dynamic typing contributes to the language's flexibility and ease of use.

Example:

Variable assignment in Python (dynamic-typed):
myVariable = “Hello World”

VS

Variable assignment in Dart (statically-typed):
String myVariable = “Hello World”

5.1 Variables Rules:

5.1.1 Variables Naming Rules:

- Variable names can consist of letters (both lowercase and uppercase), digits, and the underscore character _.
- They cannot start with a digit.
- Variable names are case-sensitive, meaning myVar and myvar would be considered different variables.
- Avoid using Python reserved words (keywords) as variable names.

5.1.2 Variables Assignment Rules:

Variables are assigned using the assignment operator =. For example: my_variable = 10.

5.1.3 Data Types:

Python is dynamically typed, so you don't need to explicitly declare the data type of a variable. The interpreter infers the type based on the assigned value.

5.1.4 Convention for Variable Names:

It's a convention in Python to use lowercase names with underscores to represent variables (snake_case). For example: my_variable.

5.1.5 Avoid Single Character Names:

Unless it has a specific purpose (like loop counters), it's generally better to use descriptive names rather than single-character names.

5.1.6 Readability Counts:

Choose variable names that are descriptive and convey the purpose of the variable. This enhances the readability of your code.

Note: Python uses the concept of Variable Reassignment, where variables are mutable, meaning their values can be changed during the execution of the program.

6.0 Escape Character Sequences

- Backspace: using \b we can make a backspace in our string, where the character before the \b will be deleted

  Example: print “Hello \b World” OUTPUT: HelloWorld.

- Escape New Line: so if you want to write a string line in multiple lines in your code editor but you want to show it on the same lien when you print it you can use the backslash sign to do that

  Example: print(“ Hello \
                              World\
                              Baby”)
  OUTPUT: Hello World Baby

- Escape Backslash: if you want to write a backslash sign in you’re string you just need to put one more backslash before it

  Example: print(“ Hello \\ World”)
  OUTPUT: Hello \ World

- Escape Single and Double Quote Sign: if we want to write a single or double quote sign in our string we can use also the backslash sign

  Example: print(‘ Hello \‘World\’ ’)
  OUTPUT: Hello ‘World’

  Example: print(“ Hello \”World\” “)
  OUTPUT: Hello “World”

- Carriage Return: moves the cursor to the beginning of the line

  Example: print("Hello\rWorld")
  OUTPUT: World

  Example: print("123456\rAbcd")
  OUTPUT: abcd56

7.0 String Concatenation:

String concatenation in Python refers to the process of combining two or more strings into a single string. This can be done using the + operator.

Example:

```
str1 = "Hello"
str2 = "World"
mistake = str1 + str2
result = str1 + " " + str2
print(result)
```

OUTPUT:

```
HelloWorld
Hello World
```

In this example, the strings "Hello" and "World" are concatenated with a space in between, resulting in the output.

7.1 String Concatenation Examples:

Example:
Concatenate the variables with some words to produce the message in the following output

OUTPUT: "Hello, My Name Is Ahmad Jawabreh And I'm 21 Years Old and I Live in Palestine."

Variables:

```
Name = “Ahmad”
Last Name = “Jawabreh”
Age = 21
Live = Palestine
```

CODE:

```
print("Hello, My Name Is " + name + " " + last_name + " And I'm " + age + " Years Old and I Live in " + live)
```

8.0 Strings in Python

8.1 String Methods:

1-) len() :
returns the length of the string

Example:
```
myString = "I Love You"
print(len(myString))
```

OUTPUT:
```
10
```

2-) strip()
[strip, rsrip, lstrip]:

- strip: removes the whitespaces from the beginning and the end of the string, you can also specify any character instead of the whitespaces.
- rstrip: removes the whitespaces from the end (right) of the string, you can also specify any character instead of the whitespaces.
- lstrip: removes the whitespaces from the beginning (left) of the string, you can also specify any character instead of the whitespaces.

Example:
```
myString = "       I Love You          "
print(myString.strip())
```

OUTPUT:
```
I Love You
```

Example:
```
myString = "       I Love You"
print(myString.lstrip())
```

OUTPUT:
```
I Love You
```

Example:
```
myString = "I Love You      "
print(myString.rstrip())
```

OUTPUT:
```
I Love You
```

3-) title(): returns the string as title (First letter of every word is capital letter)
Example:
```
myString = "i love you"
print(myString.title())
```

OUTPUT:
```
I Love You
```

4-) capitalize(): capitalizes only the first character of the string.

Example:
```
myString = "i love you"
print(myString.capitalize())
```

OUTPUT:
```
I love you
```

5-) zfill(): used to pad a string representation of a number with zeros on the left.

Example:
```
number = "42"
width = 5
result = number.zfill(width)
print(result)
```

OUTPUT:
```
00042
```

6-) upper(): used to change the case of the characters to uppercase in a string.

Example:

```
text = "hello world"
result = text.upper()
print(result)
```

OUTPUT:

```
HELLO WORLD
```

7-) lower(): converts all the characters in a string to lowercase.

Example:

```
text = "Hello World"
result = text.lower()
print(result)
```

OUTPUT:

```
hello world
```

8-) split(), rsplit(): used to split a string into a list of substrings based on a specified delimiter.

Example:

```
text = "apple orange banana"
result = text.split()
print(result)
```

OUTPUT: `['apple', 'orange', 'banana']`

9-) center(): used to center a string within a specified width by make whitespace before and after the string

Example:

```
text = "Hello World"
result = text.center()
print(result)
```

OUTPUT: `    Hello World`

Example:

```
text = "hello"
width = 10
fillchar = '*'
result = text.center(width, fillchar)
print(result)
```

OUTPUT:
```
**hello***
```

10-) count(): used to count the occurrences of a specified substring within the given string

Example:

```
text = "hello world hello"
substring = "hello"
result = text.count(substring, 6, 18)
print(result)
```

OUTPUT:
```
1
```

11-) swapcase(): used to swap the case of each character in the given string. It converts uppercase characters to lowercase and vice versa.

Example:

```
text = "Hello World"
result = text.swapcase()
print(result)
```

OUTPUT:
```
hELLO wORLD
```

12-) endswith(): This method returns True if the string ends with the specified suffix; otherwise, it returns False.

string.endswith(prefix, start, end)

Example:

```
text = "Hello World"
result = text.endswith("World")
print(result)
```

OUTPUT:
```
True
```

13-) startswith(): This method returns True if the string starts with the specified prefix; otherwise, it returns False.

string.startswith(prefix, start, end)

Example:

```
text = "Hello World"
result = text.startswith("Hello")
print(result)
```

OUTPUT: True

14-) index(): used to find the index of the first occurrence of a specified substring within the given string. If the substring is not found, it raises a ValueError.

string.index(substring, start, end)

Example:

```
text = "Hello World"
substring = "lo"
result = text.index(substring)
print(result)
```

OUTPUT: 3

15-) find(): used to find the index of the first occurrence of a specified substring within the given string. If the substring is not found, it returns -1.

string.find(substring, start, end)

Example:

```
text = "Hello World"
substring = "lo"
result = text.find(substring)
print(result)
```

OUTPUT: 3

16-) rjust(), ljust(): rjust() and ljust() are string methods used for right-justifying and left-justifying a string within a specified width, respectively. These methods are often used for formatting text in a fixed-width field.
string.rjust(width, fillchar)
string.ljust(width, fillchar)

Example:
```
text = "hello"
width = 10
fillchar = '*'
result = text.ljust(width, fillchar)
print(result)
```

OUTPUT:
```
hello*****
```

17-) splitlines(): used to split a multi-line string into a list of lines. It identifies line breaks in the string and separates the text accordingly.
string.splitlines(keepends)
Example:
```
text = "Hello\nWorld\nPython"
result = text.splitlines()
print(result)
```

OUTPUT:
```
['Hello\n', 'World\n', 'Python']
```

18-) expandtabs(): used to replace tab characters (`\t`) in a string with a specified number of spaces.
string.expandtabs(tabsize)
Example:
```
text = "Hello\tWorld\tPython"
result = text.expandtabs(tabsize=4)
print(result)
```

OUTPUT:
```
Hello   World   Python
```

19-) istitle(): checks whether each word in a string starts with an uppercase letter and the rest of the characters are lowercase. It returns True if the string is in title case format; otherwise, it returns False.
string.istitle()
Example:
text = "Hello\tWorld\tPython"

result = text.expandtabs(tabsize=4)

print(result)

OUTPUT: True

20-) isspace(): used to check if a string consists only of whitespace characters. It returns True if all characters in the string are whitespace (spaces, tabs, and newlines), and False otherwise.
string.isspace()
Example:
text = "   \t\n"

result = text.isspace()

print(result)

OUTPUT: True

21-) islower(): used to check if all alphabetic characters in a string are lowercase
string.islower()
Example:
text = "Hello World"

result = text.islower()

print(result)

OUTPUT: False

22-) isupper(): used to check if all alphabetic characters in a string are lowercase
string.isupper()
Example:
text = "Hello World"

result = text.isupper()

print(result)

OUTPUT: True

23-) isidentifier(): used to check if a given string is a valid identifier. An identifier is a name given to entities like variables, functions, classes, etc., in a program. For a string to be considered a valid identifier.
string.isidentifier()
Example:

```
text = "variable_name"
result = text.isidentifier()
print(result)
```

OUTPUT: True

24-) isalpha(): used to check if all characters in a given string are alphabetic.
string.isalpha()
Example:

```
text = "Hello123"
result = text.isalpha()
print(result)
```

OUTPUT: True

25-) isalnum(): used to check if all characters in a given string are alphanumeric, It returns True if all characters are either letters (alphabetic) or numbers (digits), and False otherwise.
string.isalnum(),
Example:

```
text = "Hello123"
result = text.isalnum()
print(result)
```

OUTPUT: True

8.2 Strings Examples:

Example:
Create three variables: your name, age, and country, then print the following message and Concatenate the variables with the following tags and words to get the same message in the end. You must show the message as it is with all the tags in the same order and spacing.

OUTPUT: "Hello 'Ahmad', How Are You Doing \ """ Your Age Is "21"" + And Your Country Is: Palestine

CODE:

```
name = "Ahmad"
age = "21"
live = "Palestine"

print(" \"Hello \'"+name+"\', How Are You Doing \\ \"\"\" Your Age Is \"" +age+ "\"\" + And
Your Country Is: "+live)
```

Example:
Print the same message as before, but on more than one line. See the desired output

OUTPUT:

```
"Hello 'Osama', How You Doing \
""" Your Age Is "38"" +
And Your Country Is: Egypt
```

CODE:

```
print("\n \"Hello \'"+name+"\', How Are You Doing \\ \n \"\"\" Your Age Is \"" +age+ "\"\" +
\n And Your Country Is: "+live)
```

Example:
Remove the extra signs from the word and only the word remains. See the example to see the idea

OUTPUT:
```
name = "#@#@Jawabreh#@#@"
```

CODE:
```
name = "#@#@Jawabreh#@#@"
clean_name = name.strip("#@")
print(cleaned_name)
```

Example:
Create a variable that contains any number you want and its type is String, then put zeros before any number that is put as the value of the variable, provided that the width of the numbers does not exceed 4 numbers. For example, 20 is 0020, 199 is 0199, and 1200 is the same as 1200. See what is required in the following example.

OUTPUT:
```
# 0009
# 0015
# 0130
# 0950
# 1500
```

CODE:
```
num1 = "9"
num2 = "15"
num3 = "130"

print(num1.zfill(4))
print(num2.zfill(4))
print(num3.zfill(4))
```

Example:
Place @ signs before any String you are given, provided that the number of characters does not exceed 20. See the example to see the idea.

OUTPUT:
```
@@@@@@@@@@@@@@@Ahmad
@@@@@@@@@@@@Jawabreh
```

CODE:
```
first_name = "Ahmad"
surname = "Jawabreh"
print(first_name.rjust(20,'@'))
print(surname.rjust(20,'@'))
```

Example:
Convert uppercase letters to lowercase letters and vice versa. See the example to see the idea
```
first_name = "AhMaD"
surname = "jAwAbreh"
```

OUTPUT:
```
aHmAd
JaWaBREH
```

CODE:
```
first_name = "AhMaD"
surname = "jAwAbreh"
print(first_name.swapcase())
print(surname.swapcase())
```

Example:
Count how many times the word Love is repeated in the String that will be given to you.
```
msg = "I Love Python And Although Love Elzero Web School"
```

CODE:
```
msg = "I Love Python And Although Love Elzero Web School Love"
print(msg.count("Love"))
```

Example:
Print the index for the letter w in the word Jawabreh

CODE:

```
msg = "Jawabreh"
print(msg.index('w'))
```

Example:
Replace the following word “<3” with the word Love only once in the sentence that will be given to you
msg = "I <3 Python And Although <3 Elzero Web School"

CODE:

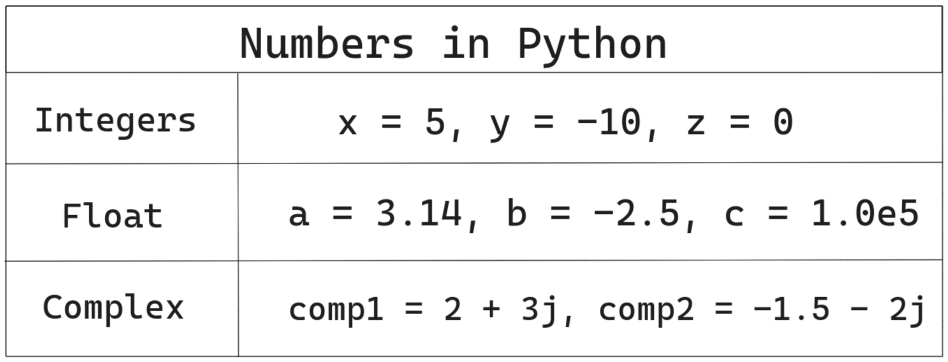
```
msg = "I <3 Python And Although <3 Elzero Web School"
print(msg.replace("<3", "Love", 1))
```

Example:
Create three variables containing your name, age, and country. Make sure that the age is an Integer and not a String, and print the following message using the new Format method “f.” See the example.
My Name Is Ahmad, And My Age Is 22, and My Country is Palestine

CODE:

```
name = "Ahmad"
age = 21
country = "Palestine"
output = f"My Name Is {name}, And My Age Is {age}, and My Country is {country}"
print(output)
```

9.0 Numbers in Python:



Numbers in Python	
Integers	x = 5, y = -10, z = 0
Float	a = 3.14, b = -2.5, c = 1.0e5
Complex	comp1 = 2 + 3j, comp2 = -1.5 - 2j

figure 2: Numbers in Python

- Integers are whole numbers without a fractional component.
- Floats are numbers with a decimal point or in exponential form.
- Complex numbers have a real and an imaginary part, expressed as a + bj, where a and b are real numbers, and j is the imaginary unit.
-

Example:
Convert the number 10 to a Floating Point Number with ten digits after the decimal point

CODE:

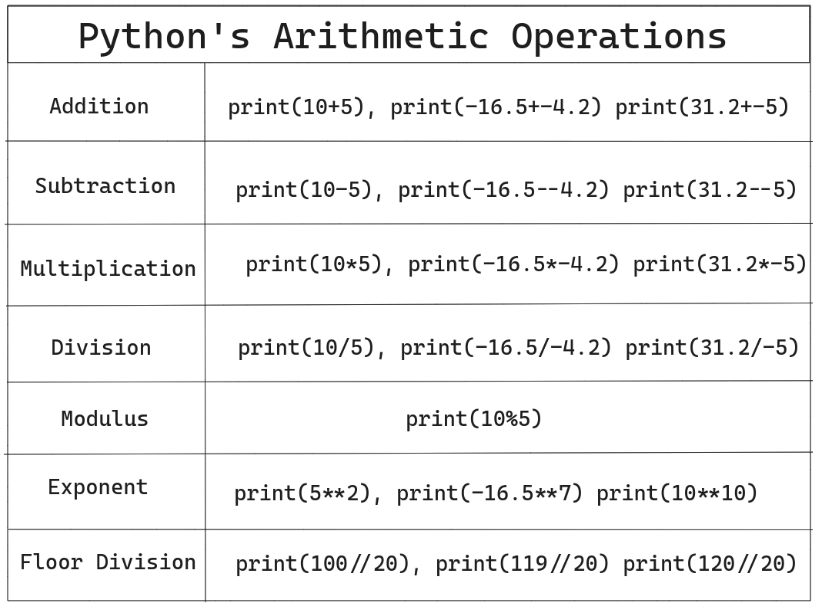
```
num = 10

formatted_float = format(float(num), ".10f")
print(formatted_float)
print(type(float(num)))
```

Example:
Convert the number 159.650 to Integer, then print it in the first line, then in the second line you print its type and confirm that it is Integer.

CODE:

```
num = 159.650
int_num = int(num)
print(int_num)
print(type(int_num))
```

10.0 Arithmetic Operations:



Python's Arithmetic Operations	
Addition	print(10+5), print(-16.5+-4.2) print(31.2+-5)
Subtraction	print(10-5), print(-16.5--4.2) print(31.2--5)
Multiplication	print(10*5), print(-16.5*-4.2) print(31.2*-5)
Division	print(10/5), print(-16.5/-4.2) print(31.2/-5)
Modulus	print(10%5)
Exponent	print(5**2), print(-16.5**7) print(10**10)
Floor Division	print(100//20), print(119//20) print(120//20)

figure 3: Arithmetic Operations

11.0 String Indexing and Slicing:

In Python, strings are zero-indexed, meaning the first character is at index 0, the second at index 1, and so on. You can use square brackets to access individual characters in a string. On the other hand, slicing allows you to extract a portion of a string. The syntax is string[start:stop], where start is the index of the first character you want, and stop is the index of the first character you don't want to include.

Indexing:

```
my_string = "Hello, World!"
print(my_string[0])  # Output: H
print(my_string[7])  # Output: W
print(my_string[-1])  # Output: !
```

Slicing:

```
my_string = "Hello, World!"
print(my_string[0:5])  # Output: Hello
print(my_string[7:])   # Output: World!
print(my_string[:5])   # Output: Hello
print(my_string[:-1])  # Output: Hello, World
```

11.1 String Indexing and Slicing Examples:

Example:
Create a variable (name) with the value “Jawabreh” inside it, then using Indexing + Slicing, bring the second letter in the first line, the third letter in the second line, and the last letter in the third line. You must bring the letter in a dynamic way, as the word can change.

CODE:

```
name = "Jawabreh"
print(name[0])
print(name[1])
print(name[-1])
```

Example:
Create a variable (name) with the value “Elzero” inside it, then using Indexing + Slicing, create a code to show the below output

OUTPUT:

```
"lze"
"Ezr"
"rzE"
```

CODE:

```
name = "Elzero"
print(name[1:4] )
print(name[0::2])
print(name[-2::-2])
```

12.0 Lists:

- List items are enclosed in square brackets.
- Lists are ordered.
- Lists are mutable, you can add, delete, and edit the list easily
- List items are not unique, which means you can repeat the same item more than once.
- Lists allows you to use different data types for the items.
- The index of the first item is 0.
- You are allowed to use back indexing (negative indexing) ex: -1, -2 etc..
- You are allowed to use steps in the lists.

12.1 List Methods:

1-) append(): append (add) new item to the list

Example:

```
myList =[1,2,3,4]
myList.append(5)
print(myList)
```

OUTPUT:

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

Note: we can append a list to the main list, and the list that has been appended will be identified as one item.

Example:

```
myList =[1,2,3,4]
aList = [5,6,7]
myList.append(aList)
print(myList)
```

OUTPUT: `[1, 2, 3, 4, [5 , 6, 7]]`

How can we access an item in our list?
Let’s assume that we want to print the item that has index 2

```
print(myList[2])
```

OUTPUT: `3`

2-) extend(): extend the list by appending new list in the main list, in the append() function it's appending the new list to the main one but the new list will be identified as one item, here we are extending the main list by appending the items of the new list in the main list.

Example:
```
myList =[1,2,3,4]
aList = [5,6,7]
myList.extend(aList)
print(myList)
```

OUTPUT:
```
[1, 2, 3, 4, 5 , 6, 7]
```

We can extend append more than one list items in the main list

Example:
```
myList =[1,2,3,4]
aList = [5,6,7]
bList = [8, 9, 10]
myList.extend(aList)
myList.extend(bList)
print(myList)
```

OUTPUT: [1, 2, 3, 4, 5 , 6, 7, 8, 9, 10]

3-) remove(): remove an item from the list

Example:
```
myList =[1,2,3,4]
myList.remove(2)
print(myList)
```

OUTPUT: [1, 3, 4]

Note: if the value that we want to delete is repeated in the list so the function will delete only the first occurrence of the value.

4-) sort(): order the list items

Example:

```
myList =[1, 5, 3, 3, 2]
myList.sort()
print(myList)
```

OUTPUT: [1, 2, 3, 4, 5]

We can also make the function ordering the items reversed

Example:

```
myList =[1, 5, 3, 4, 2]
myList.sort(reverse = True)
print(myList)
```

OUTPUT: [5, 4, 3, 2, 1]

5-) reverse(): reverse the list items upside down

Example:

```
myList =[1, 5, 3, 4, 2]
myList.sort(reverse = True)
print(myList)
```

OUTPUT: [2, 4, 3, 5, 1]

6-) clear(): delete all the items of the list

Example:

```
myList =[1, 5, 3, 4, 2]
myList.clear()
print(myList)
```

OUTPUT: []

7-) copy(): copy the list items

Example:

```
myList =[1, 5, 3, 4, 2]
aList = myList.copy()
print(aList)
```

OUTPUT: [1, 5, 3, 4, 2]

8-) count(): count how many times the item is repeated
Example:
myList =[1, 5, 1, 4, 1]
print(myList.count(1))

OUTPUT: 3

9-) index(): get the index of an item by providing it's value
Example:
myList =[1, 5, 1, 4, 1]
print(myList.index(1))

OUTPUT: 0

Note: if the value is repeated in the list, you will get the index of the first occurrence of the value.

10-) pop(): remove and return the last item from a list
Example:
myList =[1, 5, 1, 4, 1]
print(myList.pop(1))

OUTPUT: 5

11-) insert(): insert (add) new item to the list by specifying the index where you want to insert the item

Example:
myList =[1, 5, 1, 4, 1]
print(myList.insert(1, 9))

OUTPUT: [1, ,9, 5, 1, 4, 1]

Example:
myList =[1, 5, 1, 4, 1]
print(myList.insert(-1, 9))

OUTPUT: [1, 5, 1, 4, 1, 9]

But what is the difference between insert() and append() if both of them are adding new item/s to the list ?

append(): insert new value to the end of the list
insert(): insert new value to anywhere in the list

12.2 List Examples:

Example:
Make a list containing the names of your friends, with at least 5 names. In the first and second lines, you are required to print the name of the first friend in the list in two ways, then in the third and fourth lines, you print the name of the last friend in the list in two ways.

CODE:

```
# We Will use index and negative index to do it
friends = ["Ahmad", "Anas", "Mohammad", "Hussain", "Abd"]
print(friends[0])
print(friends[-len(friends)+1])
print(friends[-1])
print(friends[len(friends)-2])
```

OUTPUT:

```
Ahmad
Anas
Abd
Hussain
```

Example:
From the previous list, print the names that contain an odd index on the first line, and in the second line, print the names that contain an even index.

CODE:

```
friends = ["Osama", "Ahmed", "Sayed", "Ali", "Mahmoud"]
print(friends[1::2])
print(friends[::2])
```

Example:
Print the List of names Number 2, 3, and 4 on the first line, then the last name and the one before it on the second line, knowing that the code must work if we change the number of items in the list.

CODE:
```
friends = ["Osama", "Ahmed", "Sayed", "Ali", "Mahmoud"]
print(friends[1:4])
print(friends[-3:-1])
```

Example:
Update the last two names in the list to “Elzero”

CODE:
```
friends = ["Osama", "Ahmed", "Sayed", "Ali", "Mahmoud"]
friends[-2:] = ["Elzero", "Elzero"]
print(friends)
```

Example:
Add a name from your friends to the list at the beginning of the list first, then add another name at the end of the list

CODE:
```
friends = ["Osama", "Ahmed", "Sayed", "Ali", "Mahmoud"]

friends.append("Amer")
#OR
friends.insert(len(friends),"Amer")

friends.insert(0,"Musa")
print(friends)
```

Example:
Delete the first two names from the list, then end on another line and delete the last two names from the list

CODE:

```
friends = ["Nasser", "Osama", "Ahmed", "Sayed", "Salem"]

del friends[:2]
print(friends)

del friends[-1]
print(friends)
```

Example:
Create two more lists with more friends, then add them to the first list to come up with a final list with all your friends

CODE:

```
friends = ["Ahmed", "Sayed"]
employees = ["Samah", "Eman"]
school = ["Ramy", "Shady"]

friends.extend(employees)
friends.extend(school)
print(friends)
```

Example:
Arrange the names in the list on the first line from A to Z and on the second line from Z to A

CODE:

```
friends = ["Ahmed", "Sayed", "Samah", "Eman", "Ramy", "Shady"]

a_to_z = sorted(friends)
z_to_a = sorted(friends, reverse=True)

print(a_to_z)
print(z_to_a)
```

Example:
Count the number of friends in the list

<u>CODE:</u>

```
friends = ["Ahmed", "Sayed", "Samah", "Eman", "Ramy", "Shady"]

print(len(friends))
```

Example:
Make a list containing the famous programming languages and within it a submenu containing the names of famous frameworks. Then, in the first line, print the name of the first framework in the submenu, and in the second line, the name of the last framework in the submenu, taking into account that the submenu can increase, but it is always the last item. In the main menu

<u>CODE:</u>

```
technologies = ["Html", "CSS", "JS", "Python", ["Django", "Flask", "Web"]]

frameworks = technologies[-1]
print((frameworks[0]))
print((frameworks[-1]))
```

13.0 Tuples:

- Tuple items are enclosed in parentheses
- You can remove the parentheses if you want
- Tuples are ordered, so you can use the indexing to access any item
- Tuples are immutable, you can’t add or delete items
- Tuple items are not unique
- Tuples may contain different data types at the same time
- Operators that is used in String and Lists are available in the Tuples
- If you have only one item in you tuple, use comma after the item so the compiler can understand that this is tuple

13.1 Tuples Concatenation :

Example:
```
a = (1, 2, 3, 4, 5)
b = (6, 7, 8, 9, 10)
c = a + b
print(c)
```
OUTPUT: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Example:
```
a = (1, 2, 3, 4, 5)
b = (6, 7, 8, 9, 10)
c = a + (“Hello”, “World”) +b
print(c)
```
OUTPUT: (1, 2, 3, 4, 5, ”Hello”, “World”, 6, 7, 8, 9, 10)

13.2 Repetition :

If you want to repeat same Tuple more than one time you can multiply it with the number of the repetitions you want

Example:

```
a = (1, 2)
print(a*3)\
```

OUTPUT: (1, 2, 1, 2, 1, 2)

Note: same idea works with Tuples, Lists and Strings.

13.3 Tuples Methods :

1-) count(): counts the repetition of a specific item (element)

Example:

```
a = (1, 2, 2, 2, 5)
print(a.count(3))
```

OUTPUT: 3

2-) index(): get the index of a specific element by providing it's value

Example:

```
a = (1, 2, 2, 2, 5)
print(a.count(1))
```

OUTPUT: 0

3-) index(): get the index of a specific element by providing it's value

Example:

```
a = (1, 2, 2, 2, 5)
print(a.count(1))
```

OUTPUT: 0

13.4 Tuples Examples :

Example:
Create a tuple consisting of one element, and let it be your name without using brackets (), then print the only element in the tuple on the first line, and on the second line, print the type to make sure that the type is tuple.

CODE:
```
myName = "Ahmad",
print(myName)
print(type(myName))
```

Example:
Create a tuple containing the names of 3 of your friends and the first name is Osama. Use your experience and what you learned previously to change the first name from Osama to Elzero. Print the content of the tuple on the first line. Print the type to make sure it is a tuple and not another data type, on the line Third, print the number of elements within the tuple

CODE:
```
names = ("Osama", "Ahmad", "Jawabreh")

names = ("Elzero",) + names[1:]
print(names)
print(type(names))
print(len(names))
```

Example:
Create a tuple containing the numbers 1 to 3. Create a tuple containing the letters A to C. Concatenate them into a new tuple and print its content in the first line. In the second line, count the number of elements contained within the new tuple.

CODE:
```
nums = (1, 2, 3)
ltrs = ('A', 'B', 'C')
my_tuple = nums+ltrs
print(my_tuple)
print(len(my_tuple))
```

Example:
Create a tuple containing 4 elements of any data type you want. Destruct the tuple and make an Assign for the first value of the variable a, the second value of the variable b, and the fourth value of the variable c. Print the variables a, b, c each one on a different line. Make sure that you have Destruct with just one line

CODE:

```
nums = (1, 2, 3, 4)
# tuple destruction
a, b, _, c = nums
# print the values
print(a)
print(b)
print(c)
```

14.0 Sets:

- Set items are enclosed with curly braces
- Set items are not ordered and not indexed
- Set indexing and slicing are not allowed in the sets
- Set has only immutable data types (string, numbers, tuples) dict and list are not allowed
- Set items are unique, so you are not allowed to repeat the same value more than once

14.1 Set Methods:

1-) clear(): delete all the items of the set (make it empty)

Example:

a = {1,2,3,4,5}
print(a.clear())

OUTPUT: set()

2-) union(): combine elements from two or more sets and create a new set that contains all unique elements from the input sets.

Example:

a = {1,2,3,4,5}
b = {6,7,8}
c = {9,10}
print(a.union(b, c))

OUTPUT: {1,2,3,4,5,6,7,8,9,10}

3-) add(): add an element to a set

Example:

a = {1,2,3,4,5}
print(a.add(6))

OUTPUT: {1,2,3,4,5,6}

4-) copy(): create a shallow copy of a set.

Example:

a = {1,2,3,4,5}
b = a.copy()
print(a)
print(b)

OUTPUT: {1,2,3,4,5} {1,2,3,4,5}

5-) remove(): remove specific element from a set
Example:
a = {1,2,3,4,5}
print(a.remove(1))

OUTPUT: {2,3,4,5}
Note: if the element is not available an error will be printed

---

6-) discard(): remove specific element from a set
Example:
a = {1,2,3,4,5}
print(a.remove(1))

OUTPUT: {2,3,4,5}
Note: if the element is not available there will be no errors.

7-) pop(): remove random element from a set
Example:
a = {1,2,3,4,5}
print(a.pop())

OUTPUT: {2,3,4,5}

8-) update(): update a set with the elements of another iterable or with multiple elements
Example:
a = {1,2,3,4,5}
b = {1,6,7}
print(a.update(b))

OUTPUT: {1,2,3,4,5,6,7}

9-) difference(): get the difference between two set
Example:
a = {1,2,3,4,5}
b = {1,2,3}
print(a.difference(b))

OUTPUT: {4,5}

10-) difference_update(): remove the common elements between two sets.

Example:
a = {1,2,3,4,5}
b = {1,2,3}
a.difference_update(b)
print(a)

OUTPUT: {4,5}

11-) intersection(): get the common elements between two or more sets.
Example:
a = {1,2,3,4,5}
b = {1,2,3}
print(a.intersection(b))

OUTPUT: {1,2,3}

12-) intersection_update(): update a set with the intersection of itself and another iterable.

Example:
a = {1,2,3,4,5}
b = {1,2,3}
a.intersection_update(b)
print(a)

OUTPUT: {4,5}

13-) symmetric_difference(): the symmetric difference between two sets is the set of elements that are in either of the sets, but not in both.

Example:
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7}
print(a.symmetric_difference(b))

OUTPUT: {1,2,6,7}

14-) symmetric_difference_update(): to update a set with the symmetric difference of itself and another iterable.

Example:
```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7}
a.symmetric_difference(b)
print(a)
```

OUTPUT: {1,2,6,7}

15-)issuperset(): used to check if a set is a superset of another set or any iterable. A superset is a set that contains all the elements of another set, and possibly more

Example:
```
a = {1, 2, 3, 4, 5}
b = {1,2,3}
print(a.issuperset(b))
```

OUTPUT: True

16-)issubset(): method is used to check if a set is a subset of another set or any iterable. A subset is a set that contains all the elements of another set

Example:
```
a = {1, 2, 3, 4, 5}
b = {1,2,3}
print(a.issubset(b))
```

OUTPUT: False

17-) isdisjoint(): method is used to check whether two sets are disjoint, meaning that they have no elements in common

Example:
```
a = {1, 2, 3, 4, 5}
b = {1,2,3}
print(a.isdisjoint(b))
```

OUTPUT: Fals

14.2 Set Examples:

Example:
Create a List containing the following numbers 1, 2, 3, 3, 4, 5, 1. Create a variable named unique_list and then store only the unique values from the previous List. In the first line, print the content of unique_list and make sure it contains the unique numbers. Only in the second line, print the unique_list type and make sure that the data type is a list. In the third line, print the unique_list without the last element.

CODE:
```
my_list = [1, 2, 3, 3, 4, 5, 1]

# the set items are unique but the list items are not
# so we can convert the list to set then to list to get to unique items

unique_set = set(my_list)
unique_list = list(unique_set)

print(unique_list)
print(type(unique_list))
print(unique_list[0:-1])
```

Example:
Create a new Set containing the numbers 1, 2, 3, then create a new Set containing the letters A, B, C, combine the first and second Set in three different ways and print each method on a line

CODE:
```
nums = {1, 2, 3}
ltrs = {'A', 'B', 'C'}

union = nums.union(ltrs)
nums.update(ltrs)
combined = nums | ltrs

print(union)
print(nums)
print(combined)
```

Example:
Create a Set containing elements 1, 2, 3 , create a second Set containing 1, 2, 3, 4, 5, 6 ,check whether all the contents of the first Set are present in the second or not

CODE:

```
set_one = {1, 2, 3}
set_two = {1, 2, 3, 4, 5, 6}

result = set_one.issubset(set_two)
print(result)
```

15.0 Dictionaries:

- Dictionary values are enclosed with curly braces
- Dictionary contains KEY : VALUE
- Dictionary key must be immutable (number, string, tuple), lists are not allowed
- Dictionary value can be any data type
- Dictionary key must be unique
- Dictionary is not ordered, you can access the elements by it’s key
- To print the dictionary (keys and values) we can use, print(dictname)
- To print all of the dictionary keys, print(dictionary.keys())
- To print all of the dictionary values, print(dictionary.values())
- To access element by it’s key, (dictname[key])

15.1 Two Dimensional Dictionaries:

a two-dimensional dictionary is essentially a dictionary of dictionaries. You can create it using a nested dictionary.

Example:
```
two_dimensional_dict = {
   'row1': {'column1': 1, 'column2': 2, 'column3': 3},
   'row2': {'column1': 4, 'column2': 5, 'column3': 6},
   'row3': {'column1': 7, 'column2': 8, 'column3': 9}
}

# Accessing elements
print(two_dimensional_dict['row1']['column2'])
```

15.2 Dictionary Methods:

1-)clear(): clear or remove all of the items

Example:
```
users = {“Name” : “Ahmad”}
print(users.clear())
```

OUTPUT: {}

2-)update(): add new item to the dictionary

Example:
```
users = {“Name” : “Ahmad”}
users.update({“Last Name” : “Jawabreh”})
print(users)
```

OUTPUT: {“Name” : “Ahmad”, “Last Name” : “Jawabreh”}

OR

```
users = {“Name” : “Ahmad”}
users[“Last Name”] = “Jawabreh”
```

3-) copy(): copy the items into new dictionary

Example:
```
users = {“Name” : “Ahmad”}
members = users.copy()
print(members)
```

OUTPUT: {“Name” : “Ahmad”}

4-) setdefault(): allows you to set the default value for a key in a dictionary. If the key is present in the dictionary, it returns the corresponding value. If the key is not present, it inserts the key with the specified default value.

Example:
```
my_dict = {'a': 1, 'b': 2, 'c': 3}
value = my_dict.setdefault('d', 0)
print(my_dict)
print(value)
```

OUTPUT:
```
{'a': 1, 'b': 2, 'c': 3, 'd': 0}
0
```

5-) popitem(): removes and returns the last key-value pair from the dictionary as a tuple

Example:
```
my_dict = {'a': 1, 'b': 2, 'c': 3}
key, value = my_dict.popitem()
print(f"Popped item: {key}: {value}")
print(f"Updated dictionary: {my_dict}")
```

OUTPUT:
```
Popped item: c: 3
Updated dictionary: {'a': 1, 'b': 2}
```

6-) items(): returns a view object that displays a list of dictionary's key-value tuple pairs

Example:

```
users = {“Name” : “Ahmad”}
print(users.items())
```

OUTPUT:

```
dict_items([(‘name’, ‘Ahmad’])
```

7-) fromkeys(): creates a new dictionary with keys from a given iterable (such as a list or tuple) and assigns a specified value to all keys.

Example:

```
keys = ['a', 'b', 'c']
default_value = 0
new_dict = dict.fromkeys(keys, default_value)
print(new_dict)
```

OUTPUT:

```
{'a': 0, 'b': 0, 'c': 0}
```

14.3 Dictionar Examples:

Example:

Create a Dictionary containing three programming skills with the percentage of your level in them Without using the loop, print each skill on a line with the level written in percentage next to it Add a new skill to the Dictionary with its percentage and print it on the fifth line

CODE:

```
programming_skills = {
   "HTML": 90,
   "CSS": 80,
   "Python": 30
}

print(f"HTML Progress Is {programming_skills['HTML']}%")
print(f"CSS Progress Is {programming_skills['CSS']}%")
print(f"Python Progress Is {programming_skills['Python']}%")

programming_skills["AI"] = 20
print(f"AI Progress Is {programming_skills['AI']}%")
```

16.0 Booleans:

Boolean values are the two constant objects - True / False

Example:
```
name = “Ahmad”
print(name == “Hussam”)
```
OUTPUT: False

Example:
```
result = 10 + 10
print(result == 20)
```
OUTPUT: True

16.1 Boolean Operator:

<u>and</u> : ‘and’ operator is used for boolean operations. It returns True if both operands are True, and False otherwise.

Example:
```
x = 10
y = 20
print (x==10 and y==20)
```
OUTPUT: True

Example:
```
x = 20
y = 20
print (x==10 and y==20)
```
OUTPUT: False

<u>or</u> : ‘or’ operator is another boolean operator. It returns True if at least one of the operands is True, and False if both operands are False.

Example:
```
x = 20
y = 20
print (x==10 or y==20)
```
OUTPUT: True

17.0 Assignment Operators:

1-) +=
assignment operator that is used to add the right operand to the left operand and then assign the result to the left operand.

Example:
x = 20
y = 20
x+=y
print (x)
OUTPUT: 40

2-) -=
assignment operator that subtracts the right operand from the left operand and then assigns the result to the left operand

Example:
x = 20
y = 20
x-=y
print (x)
OUTPUT: 0

3-) *=
assignment operator that multiplies the right operand by the left operand and then assigns the result to the left operand

Example:
x = 20
y = 20
x*=y
print (x)
OUTPUT: 400

4-) /=
assignment operator that divides the left operand by the right operand and then assigns the result to the left operand.
Example:
```
x = 20
y = 20
x/=y
print (x)
```
OUTPUT: 1

5-) **=
assignment operator used for exponentiation. It raises the left operand to the power of the right operand and then assigns the result to the left operand.

Example:
```
x = 4
y = 4
x**=y
print (x)
```
OUTPUT: 256

6-) %=
assignment operator used for the modulo operation. It calculates the remainder when the left operand is divided by the right operand and assigns the result to the left operand.

Example:
```
x = 4
y = 4
x%=y
print (x)
```
OUTPUT: 0

18.0 Comparison Operators (Relational Operators):

1-) ==
Check if the values on both sides are equal.

Example:
```
a = 5
b = 5
result = a == b
print(result)
```
OUTPUT: True

2-) !=
Check if the values on both sides are not equal.

Example:
```
x = 10
y = 20
result = x != y
print(result)
```
OUTPUT: True

3-) <
Check if the value on the left is less than the value on the right.

Example:
```
x = 10
y = 20
result = x < y
print(result)
```
OUTPUT: False

4-) >
Check if the value on the left is less than the value on the right.

Example:
```
x = 10
y = 20
result = x > y
print(result)
```
OUTPUT: True

5-) >=
Checks if the value on the left is greater than or equal to the value on the right.

Example:
```
e = 12
f = 10
result = e >= f
print(result)
```
OUTPUT: True

6-) <=
Checks if the value on the left is less than or equal to the value on the right.

Example:
```
c = 7
d = 7
result = c <= d
print(result)
```
OUTPUT: True

19.0 Data Type Conversion

1-) str(): to convert the data type to String

Example:
```
c = 7
str(c)
print(type(c))
```
OUTPUT: <class 'str'>

Example:
```
c = (7, 8 , 9)
c = str(c)
print(type(c))
```
OUTPUT: <class 'str'>

2-) tuple(): to convert the data type to tuple

Example:
```
c = “ahmad”
tuple(c)
print(type(c))
```
OUTPUT: <class ‘tuple’>

Example:
```
c = (7, 8 , 9)
c = tuple(c)
print(type(c))
```
OUTPUT: <class ‘tuple’>

3-) list(): to convert the data type to list

Example:
```
c = “ahmad”
list(c)
print(type(c))
```
OUTPUT: <class ‘list’>

4-) set(): to convert the data type to set

Example:
c = “ahmad”
set(c)
print(type(c))
OUTPUT: <class ‘set’>

Example:
c = (7, 8 , 9)
c = set(c)
print(type(c))
OUTPUT: <class ‘set’>

5-) dict(): to convert the data type to dictionary

In the conversion from any data type to dictionary it will be kinda different than the other data types, because the dictionary contains key and value which is not available in the other data types.

- Conversion from tuple to dictionary, it is not possible if the tuple is not containing nested tuples, see the example below

  Example:
  c = (1, 2, 3, 4, 5)
  dict(c)
  print(type(c))
  OUTPUT: ERROR

  BUT

  Example:
  c = ((‘A’, 1), (‘B’, 2))
  set(c)
  print(type(c))
  OUTPUT: <class ‘dict’>

- Conversion from list to dictionary, it is not possible if the tuple is not containing nested lists, see the example below

  Example:
  ```
  c = [1, 2]
  dict(c)
  print(type(c))
  ```
  OUTPUT: ERROR

  BUT

  Example:
  ```
  c = [['A', 1], ['B', 2]]
  set(c)
  print(type(c))
  ```
  OUTPUT: <class 'dict'>

- Conversion from set to dictionary, it is not possible if the tuple is not containing nested sets, see the example below

  Example:
  ```
  c = {1, 2}
  dict(c)
  print(type(c))
  ```
  OUTPUT: ERROR

  BUT

  Example:
  ```
  c = {{'A', 1}, {'B', 2}}
  set(c)
  print(type(c))
  ```
  OUTPUT: <class 'dict'>

20.0 User Input:

In Python, user input is typically obtained using the input() function. This function allows the program to pause and wait for the user to enter some data through the keyboard.

Example:

```
print("Enter Your Personal Information")

first_name = input("Enter Your First Name : ")
surname = input("Enter Your Surname : ")
age = input("Enter Your Age : ")

print(f"Your full name is {first_name} ss{surname}, and you'r {age} years old ")
```

OUTPUT:

```
Enter Your Personal Information
Enter Your First Name : Ahmad
Enter Your Surname : Jawabreh
Enter Your Age : 21
Your full name is Ahmad Jawabreh, and you're 21 years old
```

20.1 User Input Examples

Example:
Create a variable called name that contains the Input of the person's name. Make sure that any spaces before and after the person's name will be removed. Make sure that the person's name will have the first letter Capital and all other letters Small. Print the name with a welcome message.

CODE:

```
name = input("Please Enter Your Name : ").strip().title()
print(f"Hello {name}, Happy To See You Here")
```

Example:
Create a variable called age that contains the input for the person’s age. Make sure that the input will be an Integer and not a String if the age is younger than 16. Print a message expressing that the site contains articles that are not suitable for an age under 16 if the age is 16 or older. Print a welcome letter that includes the age and that the location is appropriate for the person.

CODE:
```
age = int(input("Please Enter You Age: "))

if age < 16:
   print("Your Age Is Under 16, Some Articles Is Not Suitable For You")
elif age >= 16:
   print(f"Hello Your Age Is {age}, All Articles Is Suitable For You")
```

Example:
Create two variables called first_name and second_name that contain an Input containing the person's first and second names. Make sure that any spaces before and after the first and second names will be removed. Make sure that the first and second names have the first letter in them Capital and the rest is Small Print a welcome message with the first and first names in it. A letter of the second name only.

CODE:
```
first_name = input("Please Enter Your First Name: ").strip().title()
second_name = input("Please Enter Your Second Name: ").strip().title()

print(f"Hello {first_name} {second_name}")
```

Example:
Create a variable called email that contains the Input of the person's email. Make sure to remove the spaces before and after the email. Make sure that all letters will be lowercase. Print a message on the first line that contains only the person's name before the @ sign with a conversion. The first letter is a capital letter. Print a message on the second line that contains the site on which the email is located only, without the Domain. On the third line, print the Top Level Domain after the "Dot."

CODE:
```
email = input("Enter Your Email Address Here: ").strip().lower()

username, rest = email.split('@')
domain, tld = rest.split('.')

formatted_username = username.capitalize()

print(f"Your username is ({formatted_username}), your domain is ({domain}), your TLD
is ({tld})")
```

21.0 Control Flow in Python (if, Else, Elif)

In Python, conditional statements like if, else, and elif are used to control the flow of a program based on certain conditions. The if statement is used to check a condition and execute a block of code if the condition is true. The else statement is used to specify a block of code that should be executed if the condition in the if statement is false. The elif statement (short for "else if") allows you to check multiple conditions sequentially.

Example:

```
# Example: Grade Classification
score = int(input("Enter your score: "))

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
else:
    grade = 'C'
print(f"Your grade is: {grade}")
```

22.1 Nested If

In Python, nested if statements refer to placing one or more if statements inside another. This allows for more complex decision-making based on multiple conditions. Here's an example of nested if statements:

Example:

```
# Example: Nested If Statements for Grade Classification
score = int(input("Enter your score: "))
if score >= 60:
    print("You passed.")
        if score >= 90:
        print("Your grade is A.")
    elif score >= 80:
        print("Your grade is B.")
    else:
        print("Your grade is D.")
else:
    print("Sorry, you failed.")
```

In the above example, there is an outer if statement checking whether the score is greater than or equal to 60. If it is, it prints "You passed" and then enters the nested if-elif-else block to determine the specific grade based on the score. If the score is less than 60, it prints "Sorry, you failed."

Nested if statements are useful when you need to consider additional conditions within a specific branch of an outer condition. However, it's essential to maintain clarity and readability in your code to avoid confusion.

22.2 or in control flow statements

In Python, the or keyword is a logical operator used in conditional statements to combine multiple conditions. It returns True if at least one of the conditions is true.

Example:

```
# Example: Using the 'or' logical operator
age = int(input("Enter your age: "))

if age < 18 or age >= 65:
    print("You qualify for special benefits.")
else:
    print("You do not qualify for special benefits.")
```

In this example, the or operator is used to check if the entered age is either less than 18 or greater than or equal to 65. If either condition is true, the program prints "You qualify for special benefits." Otherwise, it prints "You do not qualify for special benefits."

The or operator provides a way to create more flexible conditions by allowing a block of code to be executed if at least one of the specified conditions evaluates to True.

22.3 Ternary Conditional Operator

In Python, the ternary conditional operator, also known as the conditional expression, provides a concise way to write a simple conditional statement in a single line. The syntax of the ternary operator is:
value_if_true if condition else value_if_false

Here's an example to illustrate the use of the ternary conditional operator:

Example:

```
# Example: Ternary Conditional Operator
age = int(input("Enter your age: "))
result = "You qualify for special benefits" if age < 18 or age >= 65 else "You do not
qualify for special benefits"
print(result)
```

In this example, the ternary conditional operator is used to assign the value of result based on the condition. If the age is less than 18 or greater than or equal to 65, it assigns the string "You qualify for special benefits"; otherwise, it assigns the string "You do not qualify for special benefits." The result is then printed.

22.4 Control Flow Small Project

Question:

Create a python code that takes the users age in years, and convert it to month, weeks or days according to the user choice.

CODE:

```
print("\n\n\t Age Conversion Application\n\n")

age = int(input("Please enter your age : "))

print("\nPlease choose the unit to convert")
print("Month (M) \t Weeks(W) \t Days(D) \n")
unit = input("Unit : ").strip().upper()

months = age*12
weeks = months*4
days = age*356

if unit == "M":
   print(f"Your age in months is : {months}")

elif unit == "W":
   print(f"Your age in weeks is : {weeks}")
else:
   print("Please enter a valid unit")
```

22.5 Control Flow Examples

Question:
Create a python code that takes the users age in years, and convert it to month, weeks Create two variables named num1, num2, and they are Input, which contains the first and second numbers. Create a third variable, named operation, which is Input, which contains the type of arithmetic operation. Make sure that the number variables are Integer and there are no spaces before or after them. Make sure that the variable is for the operation. Arithmetic There are no spaces before or after it. Fetch the first and second numbers and the arithmetic operation from the input, then perform the operation based on the numbers and the arithmetic operation, whether it is + – * / %

CODE:
```
num1 = int(input("Enter first number here: ").strip())
num2 = int(input("Enter second number here: ").strip())
operation = input('"+" Or "-" Or "*" Or "/" Or "%": ').strip()
if operation == "+":
   result = num1+num2
elif operation == "-":
   result = num1+num2
elif operation == "*":
   result = num1*num2
elif operation == "/":
   result = num1/num2
else:
   print("Please Enter A Valid Operation")
print(f"{num1} {operation} {num2} = {result}")
```

Question:
Create a variable that contains the person’s age and its value is 17. Create a short conditional function, If Condition, in just one line using the Ternary Conditional Operator. If the age is greater than 16, a message appears that the application is suitable for you, otherwise the application is not suitable for you.

CODE:
```
age = 17
message = "The application is suitable for you." if age > 16 else "The application is not
suitable for you."
print(message)
```

Question:
You have a list that contains the names of some countries. You have the value of discounts stored in a variable called discount. Make an input containing the name of the person's country and store it in a variable called country. Make sure that the name of the country has the first letter in it, Capital, and there are no spaces before or after it. Check If the country is present in the list of countries, if it is present, print a message stating that you have a specific discount, then calculate the discount. If it is not present in the country, print a message stating that there are no discounts and print the amount as it is. Try entering an existing country once and for all. Not present once to confirm the solution

CODE:

```
country = input("Enter Your Country Name: ").strip().title()
countries = ["Palestine", "Jordan", "Lebanon", "Yemen"]

price = 100
discount = 30
discount_price = price - discount

if country in countries:
   print(f"Your Country Eligible For Discount And The Price After Discount Is
{discount_price}")
else:
   print(f"Your Country Not Eligible For Discount And The Price Is {price}")
```

23.0 Membership Operators

In Python, membership operators are used to test whether a value belongs to a sequence, such as a string, list, or tuple. There are two membership operators: in and not in. Here's a brief explanation of each:

1-) ‘in’ Operator : It returns True if a specified value is found in the sequence.

Example:

```
# Membership operator 'in'
fruits = ['apple', 'banana', 'orange']
print('banana' in fruits)  # Output: True
print('grape' in fruits)   # Output: False
```

2-) ‘not in Operator: It returns True if a specified value is not found in the sequence.

Example:

```
# Membership operator 'not in'
colors = ['red', 'green', 'blue']
print('yellow' not in colors)  # Output: True
print('red' not in colors)     # Output: False
```

Membership operators are particularly useful when you need to check whether an element is present or absent in a collection, allowing you to make decisions based on the membership status of a value within a sequence.

24.0 Small Project 1

Question:

Utilizing your understanding of control flow, data types, their methods, and membership operators, generate a list of administrators and populate it with three admin names. Subsequently, prompt the user to input their username. If the provided name is not found in the list, inform the user that they are not recognized as an admin. Conversely, if the entered username matches an admin, inquire whether the user wishes to update their username or delete their account. In the event of an update, request the new username and make the necessary modification. If the user opts for account deletion, proceed to remove it from the list.

CODE:

```
print("\n\n\t Admin Portal\n\n")

admins = ["Ahmad", "Hassan", "Osama"]

username = input("Please enter your username : ").strip()

if username in admins:
   option = input("Update username or delete account ? ").strip().lower()

   if option == "update":
      new_username = input("\n Enter you new username please : ")
      admins[admins.index(username)] = new_username
      print("\n [INFO] Username updated successfully [INFO] \n")

   elif option == "delete":
      admins.remove(username)
      print("\n [INFO] Account deleted successfully [INFO] \n")

   else:
      print("\n [INFO] Please enter a valid option [INFO] \n")

else:
   print("\n [INFO] You are not an admin [INFO] \n")
```

25.0 Loops:

Loops in Python are essential constructs for repetitive execution of code. The for loop is particularly useful for iterating over sequences like lists, strings, or ranges, simplifying the process of performing the same operation on each element. On the other hand, the while loop continues to execute as long as a specified condition holds true, providing flexibility for dynamic scenarios. Python's loop structures facilitate efficient handling of iterative tasks, allowing developers to create concise and readable code. Whether it's processing elements in a collection or continuously executing a block of code until a condition changes, Python's loop mechanisms contribute to the language's versatility and ease of use.

25.1 While Loop:

In Python, the while loop is a control flow statement that repeatedly executes a block of code as long as a specified condition remains true.

Example:

```
# Example: Using a 'while' loop to count from 1 to 5
count = 1

while count <= 5:
   print(count)
   count += 1
```

25.1.1 While Loop Examples

Question:
Create a python code to print all the list items and it’s index

CODE:

```
mylist = ['a', 'b', 'c', 'd', 'f', 'g', 'h', 'i', 'j', 'k']
counter = 0
while counter < len(mylist):
   print(f"index {counter} value {mylist[counter]}")
   # adding +1 to the counter to step one by one
   counter +=1
```

Question:
Create a simple bookmarks management app, where you will be asking the user to enter the websites he wants to bookmark, maximum bookmarks is 5.

CODE:

```
print("\n\nSimple Bookmark Management App\n\n")

bookmarks_list = [ ]
max_bookmarks = 5

while max_bookmarks > len(bookmarks_list):
   bookmark = input("Enter the website without https:// : ")
   bookmarks_list.append(f"https://{bookmark.strip().lower()}")
else:
   choice = input("Bookmarks Is Full")
```

Question:
Create a simple ATM login where you will be asking the user to enter his pin, if the PIN is correct so he will login if the PIN is not correct you will give him three chances.

CODE:

```
print("\n\nSimple ATM App\n\n")

real_pin = "9173"
chances = 3

while chances > 0:
   input_pin = input("Please enter your card PIN: ").strip()

   if input_pin == real_pin:
      print("Correct PIN")
      break
   else:
      chances -= 1
      print(f"Wrong PIN, please try again. {chances} chances left")
```

25.2 For Loop:

In Python, a "for" loop is a versatile construct used to iterate over a sequence of elements, such as a list, tuple, string, or range. The syntax of a "for" loop is concise and readable, typically consisting of the "for" keyword, a variable that takes on each element in the sequence, the "in" keyword, and the iterable object.

During each iteration, the block of code within the loop is executed, allowing for the repeated execution of a specific set of instructions. The "for" loop is instrumental in simplifying repetitive tasks, enhancing code efficiency, and providing a clean and structured approach to handling iterable data structures in Python.

Example:

```
my_name = "Ahmad"

for name in my_name:
   print(name)
else:
   print("loop is done")
```

In Python, both "while" and "for" loops are fundamental control flow structures that facilitate repetitive tasks in different ways. The "while" loop continues iterating as long as a specified condition remains true, making it suitable for scenarios where the number of iterations is uncertain.

On the other hand, the "for" loop is designed for iterating over a sequence of elements, like lists or ranges, and is particularly useful when the number of iterations is known in advance.

The "for" loop simplifies the process of working with iterable objects, offering a concise and readable syntax. In contrast, the "while" loop provides flexibility when the loop's termination depends on dynamic conditions.

Both loops contribute to the efficiency and clarity of Python code by allowing developers to streamline repetitive operations based on specific requirements.

25.3 Loops Examples:

Question:
Create a variable called num, which is an Input that accepts a specific number from the person. Make sure that the number is of type Int and that it is greater than 0. If it is not greater than 0, print a message stating that. After entering the number, print the number that is smaller than it directly, then smaller and smaller until you reach the number 1. You must not Print the number itself and do not print the number 0. If the numbers contain the number 6, do not print it from among the numbers. After finishing printing the numbers, print a message stating that the printing of all the numbers was completed successfully, and write the number of numbers that were printed.

CODE:
```
num = int(input("Enter a number: "))
while num > 0:
   num-= 1
   print(num)
   if num == 1:
      break
else:
   print("The entered number is equals or less than zero")
```

Question:
Create a list with five of your friends, then make sure that at least two of them have names written in lowercase letters, and the rest are the first letter of the name, capital. Use While Loop to print all the names, ignoring the names that begin with lowercase letters. Print the number of names that were ignored, and it must be in a programmatic manner, not Manual

CODE:

```
friends = ["Mohamed", "Shady", "ahmed", "eman", "Sherif"]

ignored_names_count = 0
index = 0

while index < len(friends):
    current_name = friends[index]
    if current_name[0].isupper():
        print(current_name)
    else:
        ignored_names_count += 1

    index += 1

print(f"\nNumber of ignored names: {ignored_names_count}")
```

Question:

Write only one line to print the entire content of the List. Do not modify anything in the code

CODE:

```
friends = ["Mohamed", "Shady", "ahmed", "eman", "Sherif"]
for name in friends: print(name)
```

Question:

Create an empty List that will later contain the names of your friends and put it in a variable called my_friends. Specify the maximum number of friends to add to the list is 4. Do an Input request asking you for the name of the person you want to add to the list. If the person's full name consists of uppercase letters, reject the person and show it. A message stating that the name is invalid. If the name is all lowercase letters, convert only the first letter to a capital letter, then add it to the list. After adding the name, a message is printed with it stating that you have converted the first letter to a capital letter. The person's name must be printed with the special message, in addition, if the person writes A name and the first letter in which it is capitalized. Add it directly and print a message stating the addition. If the name has been added and there is still another place in the list, show Input to add another name until the list is filled with each addition. Write a message stating how many names remain in the list before it is full. Every name that is added must remove the spaces before and after it, if any

CODE:

```
my_friends = []
max_friends = 4

while len(my_friends) < 4:

   input_friend = input("Please Enter A Friend Name: ").strip()

   if input_friend.isupper():
      print("Entered name is not valid (upper case)")

   elif input_friend.islower():
      input_friend = input_friend.title()
      my_friends.append(input_friend)
      print(f"Name has been added but we changed the name format to {input_friend}
(title format)")
      print(f"Remaining places: {max_friends - len(my_friends)}")

   elif input_friend.istitle():
      input_friend = input_friend.title()
      my_friends.append(input_friend)
      print(f"Name {input_friend} has been added")
      print(f"Remaining places: {max_friends - len(my_friends)}")

   else:
      print("Please Enter A Valid Name")

else:
   print("The List is full")
```

Question:
We have a list of numbers. We want to loop them to extract the numbers that are divisible by the number 5 and return an integer. We want to print the order of the numbers automatically before each number in them. We want to print the numbers from largest to smallest. Print a message stating that the loop has completed successfully.
my_nums = [15, 81, 5, 17, 20, 21, 13]

CODE (First Way):

```
my_nums = [15, 81, 5, 17, 20, 21, 13]
sorted_nums = []

for num in my_nums:
    if num % 5 == 0:
        sorted_nums.append(num)

sorted_nums.sort(reverse=True)

for i, num in enumerate(sorted_nums, start=1):
    print(f"{i} => {num}")
print("All Numbers Printed")
```

CODE (Second Way):

```
my_nums = [15, 81, 5, 17, 20, 21, 13]

divisible_by_5 = sorted((num for num in my_nums if num % 5 == 0), reverse=True)

for i, num in enumerate(divisible_by_5, start=1):
    print(f"{i} => {num}")

print("All Numbers Printed")
```

Question:
Make a loop that prints the numbers from 1 to 20. If the number is less than 10, put a zero before it. Exclude the numbers 6, 8, and 12. Print a message stating that the loop has completed successfully.

CODE:

```
for number in range(1, 21):
    if number not in [6, 8, 12]:
        print(str(number).zfill(2))
```

26.0 Functions:

```
def functionName():
    # your function body

# call the function
functionName()
```

There are types of functions in python, one of them that's performing some task one you call it which is the basic meaning of a function, see the below example:

Example:
```
# define the function
def calculateGrade():
    a = 70
    if a >= 90:
        print("Grade is A")
    elif a >= 80:
        print("Grade is B")
    elif a >= 70:
        print("Grade is C")
    elif a >= 60:
        print("Grade is D")
    else:
        print("Failed")

# invoke the function
calculateGrade()
```

In the above example the function is doing a specific task that's already specified in the function body.

On the other hand we have types of functions that are returning value so in this case we can use the returned value to print or other uses according to the specific use case.

Example:
```
def calculateGrade():
   score 70
   if score >= 90:
      return "A"
   elif score >= 80:
      return "B"
   elif score >= 70:
      return "C"
   elif score >= 60:
      return "D"
   else:
      return "Failed"

# Example usage
grade = calculateGrade()
print(f"Grade is {grade}")
```

Another type is the functions that's accepting argument to perform a specific task, check the example below

Example:
```
   def calculateGrade(score):
   if score >= 90:
      return "A"
   elif score >= 80:
      return "B"
   else:
      return "Failed"

a = 70
grade = calculateGrade(a)
print(f"Grade is {grade}")
```

26.1 Functions packing and unpacking arguments:

Suppose you are building a function that calculates the total cost of a shopping cart with variable prices for different items, you have no idea how many items are in the shopping cart which means you have no idea how many items you value you have to calculate, so how are you gonna do that? Here the packing and unpacking becomes useful where you will be using it when you don't know how many items there will be in your function.

Example:

```
def calculate_total_cost(*items):
    total_cost = sum(items)
    return total_cost

# Example usage:
item_prices = [20, 30, 15, 25]

# Packing arguments using the * operator
total = calculate_total_cost(*item_prices)

print(f"The total cost is: ${total}")
```

26.2 Functions packing and unpacking keyword arguments:

Packing involves passing multiple keyword arguments to a function using the **kwargs syntax, which packs them into a dictionary. Here's an example:

Example:

```
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Packing keyword arguments
display_info(name="John", age=25, city="New York")
```

In this example, the display_info function takes any number of keyword arguments and prints them. The **kwargs syntax allows you to pass an arbitrary number of keyword arguments, which are then packed into the kwargs dictionary.

Example:
```
def greet_person(name, age, city):
    print(f"Hello {name}! You are {age} years old and live in {city}.")

# Creating a dictionary of keyword arguments
person_info = {"name": "Alice", "age": 30, "city": "London"}

# Unpacking and passing keyword arguments
greet_person(**person_info)
```

In this example, the greet_person function takes individual keyword arguments (name, age, city). The dictionary person_info is unpacked using the ** syntax, and its key-value pairs are passed as individual keyword arguments to the function.

Packing and unpacking keyword arguments provide flexibility in handling variable sets of parameters in functions. Packing is useful when defining functions that can accept any number of keyword arguments, while unpacking is beneficial when you have a dictionary of values that you want to pass as arguments to a function.

26.3 Default Parameters:

Default parameters in a function allow you to specify a default value for a parameter, which is used if the caller does not provide a corresponding argument. This feature provides flexibility and makes it optional for the caller to provide certain values.

Example:
```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

# Example usage:
greet("Alice")
greet("Bob", "Hi")
```

26.4 Recursive Functions:

Recursive functions in programming are functions that call themselves. They are particularly useful for solving problems that can be broken down into smaller, similar subproblems. Here's an example of a recursive function:

Example:
```
def factorial(n):
   # Base case: the factorial of 0 is 1
   if n == 0:
      return 1
   # Recursive case: n! = n * (n-1)!
   else:
      return n * factorial(n-1)

# Example usage:
result = factorial(5)
print(f"The factorial of 5 is: {result}")
```

Let's look at another example of a recursive function. This time, we'll implement the classic Fibonacci sequence:

Example:
```
def fibonacci(n):
   # Base cases: Fibonacci of 0 is 0, Fibonacci of 1 is 1
   if n == 0:
      return 0
   elif n == 1:
      return 1
   # Recursive case: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
   else:
      return fibonacci(n-1) + fibonacci(n-2)

# Example usage:
result = fibonacci(6)
print(f"The 6th Fibonacci number is: {result}")
```

In the Fibonacci sequence, each number is the sum of the two preceding ones, starting from 0 and 1. The fibonacci function calculates the Fibonacci number at a given position n. The base cases are defined for n equal to 0 and 1, where the Fibonacci sequence values are 0 and 1, respectively. The recursive case expresses that the Fibonacci number at position n is the sum of the Fibonacci numbers at positions (n-1) and (n-2).

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
             = (fibonacci(4) + fibonacci(3)) + (fibonacci(3) + fibonacci(2))
             = ((fibonacci(3) + fibonacci(2)) + (fibonacci(2) + fibonacci(1)))
               + ((fibonacci(2) + fibonacci(1)) + (fibonacci(1) + fibonacci(0)))
             = ...
             = 8
```

26.5 Functions Examples:

Question:
Create a Function that performs mathematical operations called calculate. The mathematical operations are addition, subtraction, and division. The Function accepts three Parameters: the first number, the second number, and the type of calculation operation. Name them as you wish. All you have to do is perform the calculation operation based on the inputs in case the person wrote the type of calculation operation wrong. A message appears to him that this operation does not exist. The available calculations are add, subtract, multiply. A person can write the calculation in uppercase, lowercase, or mixed letters without a problem. For example, add, ADD, aDd, a person can write only the first letter of the arithmetic operation. For example, subtract, he can write S or s. If the person does not write the arithmetic operation at all, do the default operation, which is addition.

CODE:
```
def calculate(num1, num2, operation):
    if operation == "add" or operation == "a":
        result = num1 + num2
        print(f"{num1} + {num2} = {result}")
    elif operation == "subtract" or operation == "s":
        result = num1 - num2
        print(f"{num1} - {num2} = {result}")
    elif operation == "multiply" or operation == "m":
        result = num1 - num2
        print(f"{num1} * {num2} = {result}")
    else:
        print("Please Enter A Valid Choice")

num1 = int(input("Please Enter The First Number: "))
num2 = int(input("Please Enter The Second Number: "))
operation = input("Please Enter The Operation You Wants To Make [add, subtract, 
multiply]: ").strip().lower()

calculate(num1, num2, operation)
```

Question:
Create a Function that performs mathematical operations in the name of addition. The Function accepts an unknown number of parameters. All you have to do is find the sum of all the numbers that are passed to the Function. If the number 10 is among the numbers, do not add it to the calculation if the number 5 is among the numbers. Subtract numbers from the rest of the numbers instead of adding on.

CODE:
```
def addition(*items):

    total = 0

    for number in items:
        if number == 10:
            continue
        elif number == 5:
            total -= number
        else:
            total+=number

    return total

print(addition(2,2,2,2,2,10,5))
```

Question:
Create a function that prints data about the person and his skills. The function accepts the person's name and an unknown number of skills after it. All you have to do is print a welcome message for the person and then print his skills under the name. If he does not have skills, you must show a message stating that he does not have skills yet.

CODE:
```
def showSkills(*personal):
   if len(personal) == 1:
      print(f"Hello Mr.{personal[0]} You Have No Skills To Show")
   else:
      name = personal[0]
      skills = personal[1:]
      print(f"Hello Mr.{name} Your Skills Are: ")
      for skill in skills:
         print(f"- {skill}")

showSkills("ahmad", "ruby", "python", "js")
```

Question:

Create a function that prints a welcome message for the person. The function asks you for three parameters, which are the person’s name, age, and country. If the person does not write any data, a default message appears.

CODE:
```
def say_hello(name='Unknown', age='Unknown', country='Unknown'):
   return f"Hello {name} Your Age Is {age} And You Live In {country}"

# Test cases
print(say_hello("Ahmad", 21, "Palestine"))
print(say_hello())
```

Question:

Create the appropriate function that produces all the outputs in the following examples. The data can increase or decrease

CODE:
```
def getScore(**data):
   for course, score in data.items:
      print(f"{course} => {score}")

getScore(Math=90, Science=80, Language=70)
```

Question:
Create the appropriate function that produces all the outputs in the following examples. The data can increase or decrease

CODE:
```
def getScore(**data):
   for course, score in data.items:
      print(f"{course} => {score}")

getScore(Math=90, Science=80, Language=70)
```

Question:
Create the appropriate function that outputs all the outputs in the following examples. We have new data, which is the person's name. The data can increase or decrease if the name is not written. Do not show the first welcome line if he does not have skills. Show a message stating that he does not have a Score as In the example

CODE:
```
def getPropleScore(name = None, **scores):
   if scores:
      if name is not None:
         print(f"Hello {name} This Is Your Score Table:")
      for course, score in scores.items():
         print(f"{course} => {score}")
   else:
      print("Hello Ahmed You Have No Scores To Show")

# Test
getPropleScore("Mahmoud", Logic=70, Problems=60)

# Test
getPropleScore(Logic=70, Problems=60)

# Test
getPropleScore("Ahmed")
```

27.0 File Management in Python:

27.1 Open File

```
my_file = open("/home/jawabreh/Desktop/test", "r")
```

27.2 Read File Content

```
my_file = open("/home/jawabreh/Desktop/test", "r")
print(my_file.read())
```

27.3 Close File

It’s important to close the file after finishing, because it stays in the memory if you dont close it.

```
my_file = open("/home/jawabreh/Desktop/test", "r")
print(my_file.read())
my_file.close()
```

27.4 Write In a File

```
my_file = open("/home/jawabreh/Desktop/test", "w")
my_file.write("New Content")
```

In this case we deleting the content of the file and adding new content

---

27.5 Append Data In a File

```
my_file = open("/home/jawabreh/Desktop/test", "w")
my_file.write("New Content")
```

In this case we are keeping the existing content and adding new content or data

28.0 Pythons Built In Functions:

1-) sum(iterable, start): returns the sum of your iterable

Example:
```
c = [1,2,3,4,5,6,7]
print(sum(c))
```
OUTPUT: 28

Example:
```
c = [1,2,3,4,5,6,7]
#first argument is the iterable, second one is by default 0 and it’s used to add a specific value to the total of your iterable
print(sum(c, 10))
```
OUTPUT: 38

2-) round(): used to round a floating-point number to a specified number of digits after the decimal point

Example:
```
rounded_int = round(5.67)
print(rounded_int)
```
OUTPUT: 6

3-) range(start. end, step): start is by default 0, step is by default 1, it’s used to generate a sequence of numbers

Example:
```
for i in range(5):
    print(i)
```
OUTPUT: 0 1 2 3 4

Example:
```
for i in range(1, 10, 2):
    print(i)
```
OUTPUT: 1 3 5 7 9

4-) abs(): return the absolute value of a number. The absolute value of a number is its distance from zero on the number line

Example:

```
num1 = -10
absolute_num1 = abs(num1)
print(absolute_num1)
```

OUTPUT: 10

5-) pow(): calculate the power of a number

Example:

```
result1 = pow(2, 3)
print(result1)
```

OUTPUT: 8

Example:

```
# Example 2: Calculate 2 to the power of 3, modulo 5
result2 = pow(2, 3, 5)
print(result2)
```

OUTPUT: 3

6-) min(): used to find the minimum value among a set of values or elements. It can take multiple arguments or an iterable

Example:

```
minimum_value = min(5, 3, 8, 1, 6)
print(minimum_value
```

OUTPUT: 1

Example:

```
numbers = [7, 2, 9, 1, 4]
minimum_number = min(numbers)
print(minimum_number)
```

OUTPUT: 1

7-) max(): find the maximum value among a set of values or elements. Similar to min(), it can take multiple arguments or an iterable

Example:
```
maximum_value = max(5, 3, 8, 1, 6)
print(maximum_value)
```
OUTPUT: 8

8-) map(): used to apply a specified function to all items in an input iterable (e.g., a list) and return an iterable (usually a list) of the results

Example:
```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))
```
OUTPUT: [1, 4, 9, 16, 25]

9-) filter(): used to filter elements of an iterable based on a specified function, which acts as a filter.

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

10-) reduce(): used to apply a specified function cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single accumulated result.

Example:
```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)
```
OUTPUT: 15

11-) enumerate(): built-in function that allows you to iterate over an iterable (such as a list, tuple, or string) while keeping track of the index and the corresponding element.

Example:

```
fruits = ['apple', 'banana', 'orange', 'grape']
for index, fruit in enumerate(fruits):
   print(f"Index {index}: {fruit}")
```

OUTPUT:

```
Index 0: apple
Index 1: banana
Index 2: orange
Index 3: grape
```

12-) reversed(): built-in function that returns a reversed iterator of a given iterable object.

Example:

```
original_list = [1, 2, 3, 4, 5]
reversed_list = reversed(original_list)
reversed_list_as_list = list(reversed_list)
print("Original List:", original_list)
print("Reversed List:", reversed_list_as_list)
```

OUTPUT:

```
Original List: [1, 2, 3, 4, 5]
Reversed List: [5, 4, 3, 2, 1]
```

28.1 Pythons Built In Functions Examples:

Question:
Create a function called remove_chars that removes the first and last characters from any String. Use the map to make this function pass through all the elements of the list in the assignment. Create a variable named cleaned_list and store the result of using the map in it. Make a loop on the variable cleaned_list to print all the names. In the list, use the same example, but using the Lambda Function directly inside the loop

CODE:

First Way:
```
def remove_chars(friends):
    return friends[1:-1]

friends_map = ["AEmanS", "AAhmedS", "DSamehF", "LOsamaL"]
cleaned_list = list(map(remove_chars, friends_map))

for name in cleaned_list:
    print(name)
```

Second Way:
```
friends_map = ["AEmanS", "AAhmedS", "DSamehF", "LOsamaL"]

for name in map(lambda friend: friend[1:-1], friends_map):
    print(name)
```

Question:
Create a function called get_names that returns the names that end with the letter m. Use the filter to make this function pass through all the List elements in the assignment. Create a variable named names and store the result of using the filter in it. Make a loop on the names variable to print all The names in the list. Use the same example, but using a Lambda Function directly inside the loop

CODE:
First Way:
```
friends_filter = ["Osama", "Wessam", "Amal", "Essam", "Gamal", "Othman"]
names = list(filter(lambda friends: friends[-1] == "m", friends_filter))
for name in names:
    print(name)
```

Second Way:

```
def get_names(friends):
    if friends[-1] == "m":
        return friends

friends_filter = ["Osama", "Wessam", "Amal", "Essam", "Gamal", "Othman"]

names = list(filter(get_names, friends_filter))

for name in names:
    print(name)
```

Question:
Use what you have learned to calculate the product of all the numbers in the list (nums). You must know that the elements of the list can change, so the solution must work on any list. Do the same solution using a Lambda Function.

CODE:

First Way:

```
def calculate(nums):
    print(sum(nums))

nums = [2, 4, 6, 2, 3]
calculate(nums)
```

Second Way:

```
nums = [2, 4, 6, 2, 3]
calculate = lambda nums: print(sum(nums))
calculate(nums)
```

29.0 Date and Time in Python:

Date and time in python is a module that must be imported first to work with date and time. import datetime

29.1 Working With Date and Time:

- get current date and time
  print(datetime.datetime.now())

- get current year
  print(datetime.datetime.now().year)

- get current month
  print(datetime.datetime.now().month)

- get current day
  print(datetime.datetime.now().day)

- get current time
  print(datetime.datetime.now().time())

- get current time (hours)
  print(datetime.datetime.now().time().hour)

- get current time (minutes)
  print(datetime.datetime.now().time().minute)

- get current time (seconds)
  print(datetime.datetime.now().time().seconds)

Example:
Create a simple application that calculates your age in days, you will provide your birthday date and it will print your age in days.

CODE:

```
date_now = datetime.datetime.now()
birthday = datetime.datetime(2002, 1, 30)
age = date_now - birthday
print(f"{age.days} Days")
```

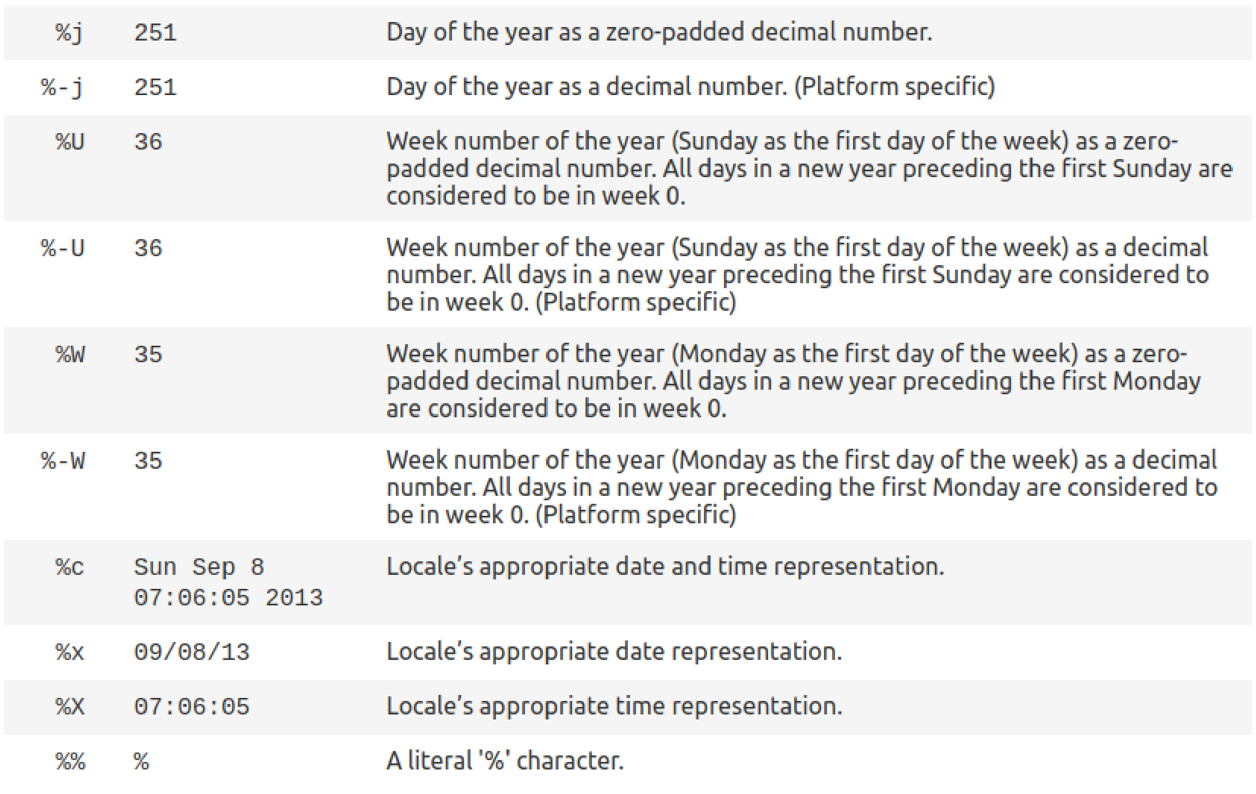
29.2 Date and Time Formatting:

In Python, date and time formatting is efficiently handled by the datetime module. You can represent dates and times in various formats using the strftime method, which allows you to create custom string representations. For instance, to display the current date and time in a human-readable format, you can use datetime.now().strftime("%Y-%m-%d %H:%M:%S"). Here, the format string specifies the order and style of the date and time components. Python's flexibility in formatting caters to diverse requirements, making it straightforward for developers, like yourself, to tailor time representations according to specific project needs.

Example:

```
a_date = datetime.datetime(2002,  1, 30)
formated_date = a_date.strftime("%d %b %Y")
print(formated_date)
```

29.3 Python strftime cheat sheet



%j	251	Day of the year as a zero-padded decimal number.
%-j	251	Day of the year as a decimal number. (Platform specific)
%U	36	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.
%-U	36	Week number of the year (Sunday as the first day of the week) as a decimal number. All days in a new year preceding the first Sunday are considered to be in week 0. (Platform specific)
%W	35	Week number of the year (Monday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Monday are considered to be in week 0.
%-W	35	Week number of the year (Monday as the first day of the week) as a decimal number. All days in a new year preceding the first Monday are considered to be in week 0. (Platform specific)
%c	Sun Sep 8 07:06:05 2013	Locale's appropriate date and time representation.
%x	09/08/13	Locale's appropriate date representation.
%X	07:06:05	Locale's appropriate time representation.
%%	%	A literal '%' character.

Code	Example	Description
`%a`	`Sun`	Weekday as locale's abbreviated name.
`%A`	`Sunday`	Weekday as locale's full name.
`%w`	`0`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.
`%d`	`08`	Day of the month as a zero-padded decimal number.
`%-d`	`8`	Day of the month as a decimal number. (Platform specific)
`%b`	`Sep`	Month as locale's abbreviated name.
`%B`	`September`	Month as locale's full name.
`%m`	`09`	Month as a zero-padded decimal number.
`%-m`	`9`	Month as a decimal number. (Platform specific)
`%y`	`13`	Year without century as a zero-padded decimal number.
`%Y`	`2013`	Year with century as a decimal number.
`%H`	`07`	Hour (24-hour clock) as a zero-padded decimal number.
`%-H`	`7`	Hour (24-hour clock) as a decimal number. (Platform specific)
`%I`	`07`	Hour (12-hour clock) as a zero-padded decimal number.
`%-I`	`7`	Hour (12-hour clock) as a decimal number. (Platform specific)
`%p`	`AM`	Locale's equivalent of either AM or PM.
`%M`	`06`	Minute as a zero-padded decimal number.
`%-M`	`6`	Minute as a decimal number. (Platform specific)
`%S`	`05`	Second as a zero-padded decimal number.
`%-S`	`5`	Second as a decimal number. (Platform specific)
`%f`	`000000`	Microsecond as a decimal number, zero-padded to 6 digits.
`%z`	`+0000`	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive).
`%Z`	`UTC`	Time zone name (empty string if the object is naive).
`%j`	`251`	Day of the year as a zero-padded decimal number.
`%-j`	`251`	Day of the year as a decimal number. (Platform specific)

29.4 Date and Time Examples:

Example:
Print a message with the number of days between the following date “2021, 6, 25” and today

CODE:
```
import datetime

a_date = datetime.datetime(2021, 6, 25)
today_date = datetime.datetime.now()
result = today_date - a_date
print(result.days)
```

Example:
Print the date and time of the current day in more than one way. See the example to understand the idea.

```
# Today Is "2021, 8, 10"

"2021-08-10"
"Aug 10, 2021"
"10 - Aug - 2021"
"10 / Aug / 21"
"10 / August / 2021"
"Tue, 10 August 2021"
```

CODE:
```
import datetime

today_date = datetime.datetime.now().date()

print(today_date)
print(today_date.strftime("%b %d  %Y"))
print(today_date.strftime("%d - %b - %Y"))
print(today_date.strftime("%d / %b / %y"))
print(today_date.strftime("%d / %b / %Y"))
print(today_date.strftime("%a, %d  %b  %y"))
```

30.0 Generators

Generators are a very powerful tool in Python for dealing with iteration . They are often used in professional Python development due to their efficiency and memory-saving capabilities. Generators allow you to create iterators in a more concise and memory-efficient manner compared to lists.

- Generators are functions with the “yield” keyword instead of “return”.

- It supports iteration and returns generator iterators by calling “yield”.

- Generator function can have one or more yield.

- By using next() it resumes from where it’s called “yield” not from the beginning.

- When it’s called, it’s not not starting automatically, it only gives you the control.

Advantages of Generators:

- Memory Efficiency: Generators produce values on-the-fly and don't store them all in memory at once, making them efficient for large datasets.

- Lazy Evaluation: Values are generated one at a time, allowing for efficient processing of data streams.

Use Cases:

- Processing Large Datasets: Generators are suitable for situations where loading an entire dataset into memory is impractical.

- Infinite Sequences: Generators can represent infinite sequences since they generate values on demand.

So for example if we are working with a machine learning project, and as we all know machine learning project are data hungry project, which means their datasets are huge, so if your going with the normal way of uploading the dataset

30.1 Generators Examples:

Example:
Use Yield with what you learned to reverse the String, then make sure that the Loop works successfully and produces the desired result.

CODE:
```
def reverse_string(my_string):
    for char in my_string[::-1]:
        yield char

for ch in reverse_string("AHMAD"):
    print(ch)
```

31.0 Decorators

- Decorators take a function, add some functionality then return it.
- Decorators wrap other function and enhance their behavior
- Decorators is higher order function (function accept function as parameter)

31.1 Decorators Examples:

Example:
Create a Decorator Function that adds a message before the Function and a Separator consisting of a Hash after the Function. Use the Decorator with the current Functions. See the example to understand what is required.

EXPECTED OUTPUT:

```
"Sugar Added From Decorators"
"Tea Created"
"####################"
"Sugar Added From Decorators"
"Coffe Created"
"####################"
```

CODE:

```
def myDecorator(func):

   def wrapperFunction():
      print("Sugar Added From Decorators")
      func()
      print("#" * 20)
   return wrapperFunction

@myDecorator
def make_tea():
  print("Tea Created")

@myDecorator
def make_coffe():
  print("Coffe Created")

make_tea()
make_coffe()
```

32.0 Documentation and DocString

Documentation and docstrings are crucial aspects of writing maintainable and understandable code in Python.

32.1 Documentation:

Documentation refers to any written or spoken information that describes how to use, understand, or maintain software. In the context of Python, documentation includes not only comments within the code but also external resources that provide information about the code. There are several types of documentation:

- Inline Comments: These are comments within the code itself, using the # symbol. They provide brief explanations of specific lines or blocks of code. While helpful, they are limited in scope.

- External Documentation: This includes separate files or documents that provide a broader overview of the project, such as README files, user guides, or API documentation.

- Code Documentation: This involves adding comments, docstrings, and annotations directly within the code to explain its functionality and usage.

32.2 Code Documentation and DocString

Docstrings are string literals that occur as the first statement in a module, function, class, or method definition. They are used to provide documentation for various elements in Python code. Docstrings can be accessed using the __doc__ attribute.

Example:

```
def add_numbers(a, b):
    """
    Adds two numbers and returns the result.

    Parameters:
    - a (int): The first number.
    - b (int): The second number.

    Returns:
    int: The sum of the two numbers.
    """
    return a + b
```

In this example, the docstring provides information about the function's purpose, parameters, and return value. Properly formatted docstrings help tools like Sphinx generate documentation automatically and are valuable for developers using or contributing to your code.

There are different docstring styles, such as the Google style, reStructuredText, and NumPy/SciPy style. Choose a style that aligns with your project or the conventions of your team.

Remember that good documentation not only benefits others who read your code but also serves as a valuable resource for future-you, helping you understand and maintain

Note: pylint is a very powerful tool to check your code and give you a rating.

33.0 Exceptions and Errors

33.1 Exception Raising

In Python, raising errors allows you to indicate that something unexpected or erroneous has occurred during the execution of your code. This helps you handle exceptional cases and communicate issues to users or developers. Here's an overview of how to raise errors in Python:

Example:
```
def divide_numbers(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

In this example, the divide_numbers function raises a ValueError if the divisor (b) is zero. The try-except block catches the exception, allowing you to handle the error gracefully.

There are many types of exceptions, the choice between them depends on the scenario in the above example we used ValueError, when the second argument of a division or modulo operation is zero we will raise ZeroDivisionError.

Python Built-in Exceptions Documentation:
https://docs.python.org/3/library/exceptions.html

33.2 Exception Handling

Error handling in Python is crucial for writing robust and reliable code. Python provides a mechanism to handle exceptions, which are unexpected events that occur during the execution of a program.

Example:
```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handling the specific exception
    print("Error: Division by zero!")
```

In the above example:

- The try block contains the code where you anticipate an exception might occur.
- The except block specifies what to do if an exception is raised in the try block.

You can handle multiple exceptions by specifying them in a tuple, The except block corresponding to the first matching exception will be executed.

Example:

```
try:
   # Code that might raise an exception
   result = int("abc")
except (ValueError, TypeError):
   # Handling multiple exceptions
   print("Error: Invalid conversion!")
```

Example:

```
try:
   # Code that might raise an exception
   result = 10 / 2
except ZeroDivisionError:
   print("Error: Division by zero!")
else:
   print("Result:", result)
finally:
   print("This will be executed no matter what.")
```

In the above example:

- The else block contains code that will be executed if no exception is raised in the try block.
- The finally block contains code that will be executed regardless of whether an exception occurred or not

34.0 Regular Expressions

Regular expressions (regex or regexp) are a powerful tool for pattern matching and manipulation of strings in Python. They allow you to search for, match, and manipulate text based on patterns.We're using re module for regular expression, it's a built in module which means you don't need to install it.

1-) search: for pattern matching:

Example:

```
pattern = r"apple"
text = "I love apples and oranges."
match = re.search(pattern, text)

if match:
    print("Found:", match.group())
else:
    print("Not found.")
```

2-) group: for extracting groups:

Example:

```
pattern = r"(\d{2})/(\d{2})/(\d{4})"
text = "Date: 12/31/2022"
match = re.search(pattern, text)

if match:
    print("Day:", match.group(1))
    print("Month:", match.group(2))
    print("Year:", match.group(3))
```

3-) sub: sub to replace matched patterns in a string.:

Example:

```
pattern = r"\d+"
text = "I have 3 apples and 5 oranges."
updated_text = re.sub(pattern, "X", text)

print("Original:", text)
print("Updated:", updated_text)
```

4-) sub: sub to replace matched patterns in a string.:

Example:

```
pattern = r"\d+"
text = "I have 3 apples and 5 oranges."
updated_text = re.sub(pattern, "X", text)

print("Original:", text)
print("Updated:", updated_text)
```

Example:

consider a scenario where you have a list of phone numbers in various formats, and you want to extract and standardize them

CODE:

```
import re

phone_numbers = """
John: +1 (555) 123-4567
Alice: 555-7890
Bob: 1-800-555-1234
"""

# Define a regular expression pattern for matching phone numbers
phone_pattern = re.compile(r"\+?1?\s?\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}")

# Find all matches in the phone numbers
matches = phone_pattern.findall(phone_numbers)

# Display the standardized phone numbers
for match in matches:
    print("Standardized Phone Number:", match)
```

35.0 Object Oriented Programming in Python

Object-Oriented Programming is a paradigm that allows you to structure your code in a way that models real-world entities using classes and objects

35.1 Main Concepts of OOP in Python

1-) Class:

- A class is a blueprint or a template for creating objects. It defines the properties and behaviors that the objects of the class will have.
- In Python, you can define a class using the class keyword.
- Class name written in PascalCase (UpperCamelCase).
- Class contains methods and attributes

Example:
```
class Car:
    pass  # Body of the class
```

2-) Objects:

- An object is an instance of a class. It is a concrete entity based on a class and can have its own unique attributes (properties) and methods (functions).
- You create an object by instantiating a class.

Example:
```
my_car = Car()  # Creating an instance of the Car class
```

3-) Attributes:

- Attributes are the properties that characterize an object. They are variables that store data.
- Instance attributes are defined inside the constructor (__init__)
- You can define attributes within a class.

Example:
```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

4-) Methods:

- Methods are functions defined within a class that perform actions on the object or manipulate its attributes.
- Instance methods take self parameter which point to instance created
- The __init__ method is a special method called the constructor, used for initializing object attributes.

  Example:
  ```
  class Car:
      def __init__(self, brand, model):
          self.brand = brand
          self.model = model

      def display_info(self):
          print(f"{self.brand} {self.model}")
  ```

35.2 Class Methods and Instance Methods

Instance methods are the most common type of methods in Python classes. They operate on an instance of the class and can access and modify the instance's attributes.

Example:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} is barking!")

    def celebrate_birthday(self):
        self.age += 1
        print(f"{self.name} is now {self.age} years old!")

# Creating an instance of the Dog class
my_dog = Dog("Buddy", 3)

# Calling instance methods
my_dog.bark()
my_dog.celebrate_birthday()
```

In the above example, bark() is an instance method that doesn't modify any attributes, while celebrate_birthday() is an instance method that modifies the age attribute.

Class methods are bound to the class and not the instance of the class. They can be called on the class itself rather than on an instance. Class methods take a reference to the class as their first parameter (cls conventionally).

Example:

```
class Circle:
    pi = 3.14  # Class attribute

    def __init__(self, radius):
        self.radius = radius

    @classmethod
    def set_pi(cls, new_pi):
        cls.pi = new_pi

    def calculate_area(self):
        return self.pi * self.radius ** 2

# Creating instances of the Circle class
circle1 = Circle(5)
circle2 = Circle(7)

# Calling instance method to calculate area
area1 = circle1.calculate_area()
print(f"Area of circle1: {area1}")

# Calling class method to change pi value
Circle.set_pi(3.14159)

# Recalculating area using the updated pi value
area2 = circle2.calculate_area()
print(f"Area of circle2 with updated pi: {area2}")
```

35.3 Class Attributes and Instance Attributes

Instance attributes are specific to each instance of a class. They are defined within the __init__ method and are unique to each object created from the class.

Example:

```
class Car:
    def __init__(self, brand, model):
        # Instance attributes
        self.brand = brand
        self.model = model

    def display_info(self):
        print(f"{self.brand} {self.model}")

# Creating instances of the Car class
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

# Accessing instance attributes
print(f"Car1: {car1.brand} {car1.model}")
print(f"Car2: {car2.brand} {car2.model}")

# Calling instance method to display information
car1.display_info()
car2.display_info()
```

In this example, brand and model are instance attributes. Each instance of the Car class has its own values for these attributes.

On the other hand, class attributes are shared among all instances of a class. They are defined outside the __init__ method and are the same for every object created from the class.

Example:

```
class BankAccount:
    # Class attribute
    interest_rate = 0.02

    def __init__(self, balance):
        # Instance attribute
        self.balance = balance

    def apply_interest(self):
        # Accessing class attribute within an instance method
        self.balance += self.balance * self.interest_rate

# Creating instances of the BankAccount class
account1 = BankAccount(1000)
account2 = BankAccount(5000)

# Accessing instance attributes
print(f"Account1 balance before interest: {account1.balance}")
print(f"Account2 balance before interest: {account2.balance}")

# Calling instance method to apply interest
account1.apply_interest()
account2.apply_interest()

# Accessing instance attributes after applying interest
print(f"Account1 balance after interest: {account1.balance}")
print(f"Account2 balance after interest: {account2.balance}")
```

In this example, interest_rate is a class attribute shared among all instances of the BankAccount class, while balance is an instance attribute specific to each account.

35.4 Important Concepts of OOP in Python

1-) Inheritance:

- Inheritance allows a class (subclass) to inherit properties and methods from another class (superclass).
- It promotes code reuse and the creation of a hierarchy of classes.

Example:

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

    def make_sound(self):
        pass  # Placeholder for subclasses to implement

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Creating instances of subclasses
my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")

# Calling the inherited method
print(f"{my_dog.name} says: {my_dog.make_sound()}")
print(f"{my_cat.name} says: {my_cat.make_sound()}")
```

In this example, the Dog and Cat classes inherit from the Animal class. They share the name attribute and override the make_sound method to provide their specific sound.

There is a concept in inheritance called Multiple inheritance, in which Python allows a class to inherit attributes and methods from more than one parent class. While it provides flexibility, it can also lead to challenges if not used carefully.

Example:

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

    def make_sound(self):
        pass  # Placeholder for subclasses to implement

class Flying:
    def fly(self):
        return f"{self.name} is flying!"

class Swimming:
    def swim(self):
        return f"{self.name} is swimming!"

class FlyingFish(Animal, Flying, Swimming):
    def make_sound(self):
        return "Bloop!"

nemo = FlyingFish("Nemo")

# Utilizing methods from multiple parent classes
print(nemo.make_sound())  # Inherited from Animal
print(nemo.fly())         # Inherited from Flying
print(nemo.swim())        # Inherited from Swimming
```

In this example, FlyingFish is a class that inherits from Animal, Flying, and Swimming. The FlyingFish class can access and use methods from all three parent classes (Animal, Flying, and Swimming).

However, when using multiple inheritance, it's important to consider potential conflicts and the order in which the parent classes are specified. Python uses a method resolution order (MRO) to determine the order in which base classes are searched when a method is called. You can check the MRO using the __mro__ attribute or the mro() method:

```
print(FlyingFish.__mro__)
# Output: (<class '__main__.FlyingFish'>, <class '__main__.Animal'>, <class
'__main__.Flying'>, <class '__main__.Swimming'>, <class 'object'>)
```

2-) Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables flexibility in using different classes interchangeably.

Example:

```
class Shape:
    def calculate_area(self):
        pass  # Placeholder for subclasses to implement

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

    def calculate_area(self):
        return 3.14 * self.radius ** 2

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

    def calculate_area(self):
        return self.length * self.width

# Using polymorphism with a list of shapes
shapes = [Circle(5), Rectangle(4, 6)]

# Calculating and displaying areas without knowing specific shape types
for shape in shapes:
    print(f"Area: {shape.calculate_area()}")
```

In this example, both the Circle and Rectangle classes inherit from the common base class Shape. The calculate_area method is polymorphic, allowing us to calculate the area of different shapes without knowing their specific types.

3-) Encapsulation:

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) that involves bundling data (attributes) and methods (functions) that operate on the data within a single unit, known as a class. The main idea is to hide the internal implementation details of a class from the outside world and provide controlled access to the class's functionalities. Encapsulation aims to achieve information hiding, modularity, and abstraction.

In Python, there are three levels of access:
Public (default): Members are accessible from outside the class.
Protected (single underscore _): Members should not be accessed from outside the class, but it is a convention, and access is still possible.
Private (double underscore __): Members are not accessible from outside the class.

Private Attributes and Methods:

Example:
```
class BankAccount:
   def __init__(self, balance):
      # Private attribute
      self.__balance = balance

   def __validate_transaction(self, amount):
      # Private method
      return amount > 0

   def deposit(self, amount):
      if self.__validate_transaction(amount):
         self.__balance += amount
         print(f"Deposit successful. New balance: {self.__balance}")
      else:
         print("Invalid deposit amount.")

   def withdraw(self, amount):
      if self.__validate_transaction(amount) and amount <= self.__balance:
         self.__balance -= amount
         print(f"Withdrawal successful. New balance: {self.__balance}")
      else:
         print("Invalid withdrawal amount or insufficient funds.")
```

4-) Getters and Setters:

- Encapsulation often involves providing controlled access to private attributes through getter and setter methods.
- Getter methods retrieve the value of a private attribute, and setter methods modify the value.

Example:

```
class Student:
    def __init__(self, name, age):
        # Private attributes
        self.__name = name
        self.__age = age

    # Getter methods
    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    # Setter methods
    def set_name(self, new_name):
        self.__name = new_name

    def set_age(self, new_age):
        if new_age > 0:
            self.__age = new_age
        else:
            print("Invalid age. Age must be greater than 0.")
```

35.5 OOP Examples

Example:
Write the contents of the class to display the result as below:

```
class Game:
  # Write Class Content

game_one = Game("Ys", "Falcom", 2010, 50)

print(f"Game Name Is \"{game_one.name}\", ", end="")
print(f"Developer Is \"{game_one.developer}\", ", end="")
print(f"Release Date Is \"{game_one.year}\", ", end="")
print(f"Price In Egypt Is {game_one.price_in_pounds()}", end="")

# Needed Output
# "Game Name Is "Ys", Developer Is "Falcom", Release Date Is "2010", Price In Egypt
Is 780.0 Egyptian Pounds"
```

CODE:

```
class Game:
    def __init__(self, name, developer, year, price):
        self.name = name
        self.developer = developer
        self.year = year
        self.price = price

    def price_in_pounds(self):
        pund_conversion_rate = 15.6
        punds_price = self.price * pund_conversion_rate
        return f"{punds_price} Egyptian Pounds"

game_one = Game("Ys", "Falcom", 2010, 50)

print(f"Game Name Is \"{game_one.name}\", ", end="")
print(f"Developer Is \"{game_one.developer}\", ", end="")
print(f"Release Date Is \"{game_one.year}\", ", end="")
print(f"Price In Egypt Is {game_one.price_in_pounds()}", end="")
```

Example:

Write the contents of the Class to display the result as in the example. Make sure that the person's surname is shown based on gender as follows: Mr For Male And Mrs For Female. Print the age message as in the example by subtracting the age from the number 40 and showing the message.

```
class User:
  # Write Class Content

user_one = User("Osama", "Mohamed", 38, "Male")
user_two = User("Eman", "Omar", 25, "Female")

print(user_one.full_details()) # Hello Mr Osama M. [02] Years To Reach 40
print(user_two.full_details()) # Hello Mrs Eman O. [15] Years To Reach 40
```

CODE (Option One):

```
class User:
    def __init__(self, name, surname, age, gender):
        self.name = name
        self.surname = surname
        self.age = age
        self.gender = gender

    def full_details(self):
        if self.gender == "Male":
            return f"Hello Mr {self.name} {self.surname[0:1]}. [{40 - self.age}] Years To
Reach 40"
        elif self.gender == "Female":
            return f"Hello Mrs {self.name} {self.surname[0:1]}. [{40 - self.age}] Years To
Reach 40"

user_one = User("Osama", "Mohamed", 38, "Male")
user_two = User("Eman", "Omar", 25, "Female")

print(user_one.full_details()) # Hello Mr Osama M. [02] Years To Reach 40
print(user_two.full_details()) # Hello Mrs Eman O. [15] Years To Reach 40
```

CODE (Option Two):

```
class User:
    def __init__(self, first_name, last_name, age, gender):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.gender = gender

    def title(self):
        return "Mr" if self.gender == "Male" else "Mrs"

    def age_difference(self):
        return abs(self.age - 40)

    def full_details(self):
        initials = self.last_name[0] if len(self.last_name) > 0 else ""
        return f"Hello {self.title()} {self.first_name} {initials}. " \
               f"[{self.age_difference():02}] Years To Reach 40"

# Creating instances of the User class
user_one = User("Osama", "Mohamed", 38, "Male")
user_two = User("Eman", "Omar", 25, "Female")

# Displaying the desired output
print(user_one.full_details())  # Hello Mr Osama M. [02] Years To Reach 40
print(user_two.full_details())  # Hello Mrs Eman O. [15] Years To Reach 40
```

- The title method determines whether to use "Mr" or "Mrs" based on the gender.
- The age_difference method calculates the difference between the user's age and 40.
- The full_details method combines the information to create the full message.

Example:
Write the contents of the class to display the result as in the example

```
class Message:
  # Write Class Content

print(Message.print_message())

# Output
# Hello From Class Message
```

CODE:

```
class Message:

    @classmethod
    def print_message(cls):
        return "Hello From Class Message"

print(Message.print_message())
```