Python Output

In [1]:
```
# Python is case sensitive
print('Hello World')
```

```
Hello World
```

In [50]:
```
print(hey) # it throws error beacuse only string is always in " "
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9556\1311815035.py in <module>
----> 1 print(hey) # it throws error beacuse only string always in " "

NameError: name 'hey' is not defined
```

In [3]:
```
print(7)
```

```
7
```

In [4]:
```
print(True)
```

```
True
```

In [5]:
```
# how print function strng in python -- it print all the values which we want to pr
print('Hello',3,4,5,True)
```

```
Hello 3 4 5 True
```

In [6]:
```
# sep
print('Hello',3,4,5,True,sep='/') # sep is separator where in print function space
```

```
Hello/3/4/5/True
```

In [7]:
```
print('hello')
print('world')
```

```
hello
world
```

In [8]:
```
# end
print('you',end="=")
print ('me')
```

```
you=me
```

Data Types

1. Integers

In [9]:
```
print(8)
```

```
8
```

2. Float (Decimal)

In [10]:
```
print(8.44)
```

```
8.44
```

3. Boolean

In [11]:
```
print(True)
print(False)
```

```
True
False
```

4. String (Text)

In [12]:
```
print('Hello gem')
```

```
Hello gem
```

5. Complex

In [13]:
```
print(5+6j)
```

```
(5+6j)
```

6. List

In Python, a list is a data type used to store a collection of values. It is one of the built-in data types and is classified as a sequence type. Lists are ordered, mutable (which means you can change their contents), and can contain elements of different data types, including integers, floats, strings, or even other lists.

You can create a list in Python by enclosing a comma-separated sequence of values within square brackets [ ]. For example:

In [ ]:
```
print([1,2,3,4])
```

7. Tuple

In Python, a tuple is another data type used to store a collection of values, similar to a list. However, there are some key differences between tuples and lists:

- Immutable: The most significant difference is that tuples are immutable, meaning once you create a tuple, you cannot change its contents (add, remove, or modify elements). Lists, on the other hand, are mutable, and you can modify them after creation.
- Syntax: Tuples are created by enclosing a comma-separated sequence of values within parentheses (). Lists are created with square brackets []. For example:
- Performance: Due to their immutability, tuples can be more efficient in terms of memory and performance for certain use cases compared to lists.

Tuples are often used when you have a collection of values that should not be changed during the course of your program. For example, you might use tuples to represent

coordinates (x, y), dates (year, month, day), or other data where the individual components should remain constant.

In [14]:
```
print((1,2,3,4))
```

```
(1, 2, 3, 4)
```

8. sets

In Python, a set is a built-in data type used to store an unordered collection of unique elements. Sets are defined by enclosing a comma-separated sequence of values within curly braces {} or by using the built-in set() constructor. Sets automatically eliminate duplicate values, ensuring that each element is unique within the set.

In [15]:
```
print({1,2,3,4,5})
```

```
{1, 2, 3, 4, 5}
```

9. Dictionary

In Python, a dictionary is a built-in data type used to store a collection of key-value pairs. Each key in a dictionary maps to a specific value, creating a relationship between them. Dictionaries are also known as associative arrays or hash maps in other programming languages.

In [16]:
```
print({'name':'Nitish','gender':'Male','weight':70})
```

```
{'name': 'Nitish', 'gender': 'Male', 'weight': 70}
```

How to know which type of Datatype is?

In [18]:
```
type([1,2,3,4])
```

Out[18]:
```
list
```

In [19]:
```
type({'age':20})
```

Out[19]:
```
dict
```

3. Variables

In Python, variables are used to store data values. These values can be numbers, strings, lists, dictionaries, or any other data type. Variables are essential for manipulating and working with data in your programs. Here's how you declare and use variables in Python:

1. Variable Declaration: You declare a variable by assigning a value to it using the assignment operator '='.

In [21]:
```
x = 5  # Assigning the integer value 5 to the variable 'x'
name = "Alice"  # Assigning a string value to the variable 'name'
```

1. Variable Names: Variable names (also known as identifiers) must adhere to the following rules:

- They can contain letters (a-z, A-Z), digits (0-9), and underscores (_).
- They cannot start with a digit.
- Variable names are case-sensitive, so myVar and myvar are treated as different variables.
- Python has reserved keywords (e.g., if, for, while, print) that cannot be used as variable names.

1. Data Types: Python is dynamically typed, which means you don't need to declare the data type of a variable explicitly. Python will determine the data type automatically based on the assigned value.

In [22]:
```
x = 5  # 'x' is an integer variable
name = "Alice"  # 'name' is a string variable
```

1. Reassignment: You can change the value of a variable by assigning it a new value.

In [23]:
```
x = 5
x = x + 1  # Updating the value of 'x' to 6
```

In [24]:
```
print(x)
```

```
6
```

Multiple Assignment: Python allows you to assign multiple variables in a single line.

In [26]:
```
a, b, c = 1, 2, 3  # Assigning values 1, 2, and 3 to variables a, b, and c, respec
```

In [28]:
```
print(a)
```

```
1
```

In [29]:
```
a = 1
b = 2
c = 3
print(a,b,c)
```

```
1 2 3
```

In [30]:
```
a=b=c= 5
print(a,b,c)
```

```
5 5 5
```

What is comments in Python?

In Python, comments are used to annotate and provide explanations within your code. Comments are not executed by the Python interpreter; instead, they are meant for human readers to understand the code better. Comments are ignored by the interpreter during program execution.

Python supports two types of comments:

1. **Single-line Comments**: Single-line comments start with the hash symbol (`#`) and continue until the end of the line. They are used to add comments on a single line.

```
# This is a single-line comment
x = 5  # Assigning a value to 'x'
```

Everything after the `#` symbol on that line is considered a comment.

In [32]:
```
# This is a single-line comment
x = 5  # Assigning a value to 'x'
```

1. **Multi-line or Block Comments**: Python does not have a specific syntax for multi-line comments like some other languages (e.g., C, Java). However, you can create multi-line comments by using triple-quotes (`'''` or `"""`) as a string delimiter. These strings are not assigned to any variable and are ignored by the interpreter. This is a common practice for writing docstrings (documentation within functions and modules).

In [34]:
```
'''
This is a multi-line comment.
It can span multiple lines.
'''
def my_function():
    """
    This is a docstring.
    It provides documentation for the function.
    """
    pass
```

While these triple-quoted strings are not technically comments, they are often used as a way to document code effectively.

Comments are crucial for making your code more understandable, both for yourself and for others who may read your code. They help explain the purpose of variables, functions, and complex algorithms, making it easier to maintain and debug code. Good commenting practices can significantly improve code readability and maintainability.

4. User Input

How to get Input from the user in python?

In [35]:
```
input('Enter Email')
```

```
Enter EmailSagar@95
```

Out[35]:
```
'Sagar@95'
```

In [36]:
```
# take input from users and store them in a variable
fnum = int(input('enter first number'))
snum = int(input('enter second number'))
#print(type(fnum),type(snum))
# add the 2 variables
result = fnum + snum
# print the result
```

```
print(result)
print(type(fnum))
```

```
enter first number1
enter second number2
3
<class 'int'>
```

5. Type Conversion

How to covert One Datatypeinto another In python?

- int(): Converts a value to an integer data type. This is useful when you want to convert a string or a floating-point number to an integer.

In [44]:
```
str_number = "123"
int_number = int(str_number)  # Converts the string "123" to an integer
int_number
```

Out[44]:
```
123
```

- float(): Converts a value to a floating-point data type. It is used to convert integers or strings containing numeric values to floating-point numbers.

In [45]:
```
int_value = 42
float_value = float(int_value)  # Converts the integer 42 to a float
float_value
```

Out[45]:
```
42.0
```

In [37]:
```
a = 23
b = "24"

print(a+b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9556\3112067906.py in <module>
      2 b = "24"
      3
----> 4 print(a+b)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

It Gives error beacause We do not add float into integer datatyupe so we have to convert above opeartion

In [38]:
```
a = 23
b = "24"

print(float(a)+float(b))
```

```
47.0
```

6. Literals

In Python, literals are used to represent fixed values in your code. These values are not variables or expressions but rather constants that have a specific value and data type associated with them. Python supports various types of literals,

In [46]:
```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal

#Float Literal
float_1 = 10.5
float_2 = 1.5e2 # 1.5 * 10^2
float_3 = 1.5e-3 # 1.5 * 10^-3

#Complex Literal
x = 3.14j

print(a, b, c, d)
print(float_1, float_2,float_3)
print(x, x.imag, x.real)
```

```
10 100 200 300
10.5 150.0 0.0015
3.14j 3.14 0.0
```

In [48]:
```
# binary
x = 3.14j
print(x.imag)
```

```
3.14
```

In [49]:
```
string = 'This is Python'
strings = "This is Python"
char = "C"
multiline_str = """This is a multiline string with more than one line code."""
unicode = u"\U0001f600\U0001F606\U0001F923"
raw_str = r"raw \n string"

print(string)
print(strings)
print(char)
print(multiline_str)
print(unicode)
print(raw_str)
```

```
This is Python
This is Python
C
This is a multiline string with more than one line code.
😀😆🤣
raw \n string
```

Operators In Python

1.Arithmetic Operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Floor Division (//)
- Modulus (%)
- Exponentiation (**)

In [2]:
```
print(5+6) # Addtion -> adding the numbers

print(5-6) # subtraction-> subtract the number

print(5*6) # Multiplication -> Multiply the number

print(5/2) # Divsion -> Divide the numnber

print(5//2) # Floor Division -> It trasform into integer number= 2.5 convert into 2

print(5%2) # Modulus -> It Provides remainder of the Divsion

print(5**2) # Exponential -> raising a number to a certain power.(raised to power)
```

```
11
-1
30
2.5
2
1
25
```

2. Comparison Operators/ Relational Opeartors:

- Equal to (==)
- Not equal to (!=)
- Less than (<)
- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)

In [5]:
```
print(4==4)

print(4!=4)

print(4<5)

print(4>5)

print(4<=4)
```

```
print(4>=4)
```

```
True
False
True
False
True
True
```

2. Logical Operators:

- Logical AND (and)
- Logical OR (or)
- Logical NOT (not)

In [7]:
```
p = True
q = False

print(p and q) # true and false -> 1 and 0 = 0
print(p or q) # true or false -> 1 or 0 = 1
print(not p)
```

```
False
True
False
```

3. Assignment Operators:

- Assignment (=)
- Add and Assign (+=)
- Subtract and Assign (-=)
- Multiply and Assign (*=)
- Divide and Assign (/=)
- Floor Divide and Assign (//=)
- Modulus and Assign (%=)
- Exponentiate and Assign (**=)

In [12]:
```
x = 10
x += 5
print(x)  # Equivalent to x = x + 5
```

```
15
```

In [16]:
```
x = 10
x -= 3  # Equivalent to x = x - 3
x *= 2  # Equivalent to x = x * 2
x /= 4  # Equivalent to x = x / 4
x //= 2 # Equivalent to x = x // 2
x %= 3  # Equivalent to x = x % 3
x **= 2 # Equivalent to x = x ** 2
```

4.Bitwise Operators:

- Bitwise AND (&)

- Bitwise OR (|)
- Bitwise XOR (^)
- Bitwise NOT (~)
- Left Shift (<<)
- Right Shift (>>)

In [23]:
```
m = 5  # 101 in binary
n = 3  # 011 in binary
bitwise_and = m & n  # 001 (1 in decimal)
print(bitwise_and)

bitwise_or = m | n   # 111 (7 in decimal)
print(bitwise_or)

bitwise_xor = m ^ n  # 110 (6 in decimal)
print(bitwise_xor)

bitwise_not_m = ~m   # -6 (in decimal)
print(bitwise_not_m)

left_shift = m << 1  # 010 (2 in decimal)
print(left_shift)

right_shift = m >> 1 # 010 (2 in decimal)
print(right_shift)
```

```
1
7
6
-6
10
2
```

These are the basic operators in Python. You can use them to perform various operations on variables and values in your Python programs.

Loops In Python

- In Python, loops are used to execute a block of code repeatedly
- There are two main types of loops in Python: ### 1. 'For' loops ### 2. 'While' loops ### 3. Loops Controls Statement

What is the use of Loops?why we used that?? (interview Question)

Answer:

- Loops in Python are used to execute a block of code repeatedly.
- They are essential for automating repetitive tasks, processing collections of data (e.g., lists or strings), and iterating through sequences or until a specific condition is met.
- for loops are commonly used for iteration, while while loops are used when you need to repeat code based on a condition.
- Loop control statements like break and continue provide flexibility in managing loop execution, making Python a powerful language for tasks that involve repetition and iteration.

1. For loops

- A 'for loop' is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence.

In [2]:
```
# if we have to print 1 to 10 number then we use fo loop

for i in range(1,11):   # in for loop we have to sepcify the range or also mention
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

In [4]:
```
# for loop for datatype like tuple

for i in (1,2,3,4,5,6,7,8):
    print(i)

# for loop is very very flexible and powerful in python it used with all datatypes
```

```
1
2
3
4
5
6
7
8
```

In [7]:
```
# how to print 1 to 20 with differnce of 2

for i in range(1,21,2): # 2 is the
    print(i)
```

```
1
3
5
7
9
11
13
15
17
19
```

- The 2 inside the range() function is called the "step" or "stride." It determines the increment between each consecutive value in the range. In this case, the loop will iterate over the numbers from 1 to 20, with a step of 2.
- So, the loop will print the following numbers: 1, 3, 5, 7, 9, 11, 13, 15, 17, and 19. The loop starts at 1, increments by 2 in each iteration, and stops before reaching or exceeding 21.

In [8]:
```
# how to print 10 to 1 in reverse order

for i in range(10,0,-1):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

Program - The current population of a town is 10000. The population of the town is increasing at the rate of 10% per year. You have to write a program to find out the population at the end of each of the last 10 years.

In [12]:
```
# for this we use for loop
curr_pop = 10000
for i in range(10,0,-1):
    print(i,curr_pop)
```

```
    curr_pop /= 1.1
```

```
10 10000
9 9090.90909090909
8 8264.462809917353
7 7513.148009015775
6 6830.134553650703
5 6209.213230591548
4 5644.739300537771
3 5131.5811823070635
2 4665.07380209733
1 4240.976183724845
```

2. While Loop

- A while loop is used to repeatedly execute a block of code as long as a certain condition is true.

In [14]:
```
# if we want to print 1 to 5 with the help of while
count = 1
while count <= 5:
    print(count)
    count += 1
```

```
1
2
3
4
5
```

In [16]:
```
# Program ->print 12 table

num = int(input('Enter the Number'))

i = 1

while i<11:
    print(num * i)
    i+=1
```

```
Enter the Number12
12
24
36
48
60
72
84
96
108
120
```

3. Loop Control Statements:

In addition to basic loops, Python provides loop control statements that allow you to customize loop behavior:

- 'break': Used to exit the loop prematurely based on a certain condition.

- 'continue': Skips the current iteration of the loop and proceeds to the next iteration.
- 'else' clause: Can be used with a for or while loop to specify a block of code that will be executed after the loop has finished normally (i.e., without hitting a break statement).

while loop with else

In [17]:
```
x = 1

while x < 3:
    print(x)
    x += 1

else:
    print('limit Crossed')
```

```
1
2
limit Crossed
```

Break and Continue

In [20]:
```
# Using break

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
    if num == 6:
        break  # Exit the loop when num equals 6
    print(num)

print("Loop ended")
```

```
1
2
3
4
5
Loop ended
```

In this example, the loop will print numbers 1 through 5 and then exit the loop when num becomes 6 due to the break statement. The "Loop ended" message will be printed after the loop finishes.

In [21]:
```
# Using continue

numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        continue  # Skip printing 3 and continue with the next iteration
    print(num)

print("Loop ended")
```

```
1
2
4
5
Loop ended
```

In this example, when num equals 3, the continue statement is encountered, which skips the printing of 3 and moves on to the next iteration. As a result, the loop will print numbers 1, 2, 4, and 5. The "Loop ended" message will be printed after the loop finishes.

In [23]:
```
# Using Continue and Break
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        continue  # Skip printing 3
    elif num == 5:
        break     # Exit the loop when 5 is encountered
    print(num)

print("Loop finished normally")
```

```
1
2
4
Loop finished normally
```

Strings In Python (part-1)

In Python, a string is a sequence of characters enclosed within either single (' '), double (" "), or triple (''' ' ''', """ """ or ''' ''') quotes. Strings are used to represent text data and are one of the fundamental data types in Python

- Strings are sequence of Characters
- In Python specifically, strings are a sequence of Unicode Characters
- Immutable: Strings in Python are immutable, meaning once you create a string, you cannot change its content. You can create new strings based on the original string, but the original string remains unchanged.

1. Creating Strings

In [ ]:
```
# create string with the help of single quotes
s = 'Hello World'
print(s)
```

```
Hello World
```

In [ ]:
```
# create string with the help of Double quotes
s = "Hello World"
print(s)
```

```
Hello World
```

when we use single and double quotes?

-> In Python, you can create strings using both single quotes (') and double quotes ("), and they are functionally equivalent. The choice between using single or double quotes mainly depends on your personal preference or specific coding style guidelines.

If your string contains a quotation mark character (either a single quote or double quote), you can use the other type of quote to define the string without escaping the inner quotes. This can make your code more readable:

In [ ]:
```
single_quoted_string = 'He said, "Hello!"'
double_quoted_string = "He said, 'Hello!'"
# here we use single and double quotes
print(single_quoted_string)
print(double_quoted_string)
```

```
He said, "Hello!"
He said, 'Hello!'
```

In [ ]:
```
# multiline strings
s = '''hello'''
s = """hello"""
s = str('hello')
print(s)
```

```
hello
```

2. Accessing Substrings from a String

Indexing

Indexing in Python refers to the process of accessing individual elements or characters within a sequence, such as a string. Strings in Python are sequences of characters, and you can access specific characters in a string using indexing. Indexing is done using square brackets [ ], and Python uses a zero-based indexing system, meaning the first element has an index of 0, the second element has an index of 1, and so on.

we have two types of Indexing:

1. Positive indexing
2. Negative Indexing

In [ ]:
```
s = "Hello World"

# Accessing individual characters using positive indexing
print(s[0])  # Accesses the first character, 'H'
print(s[1])  # Accesses the second character, 'e'
```

```
H
e
```

In [ ]:
```
# Accessing individual characters using negative indexing (counting from the end)
print(s[-1])  # Accesses the last character, 'd'
print(s[-2])  # Accesses the second-to-last character, 'l'
```

```
d
l
```

Slicing

In Python, slicing is a technique used to extract a portion of a string, list, or any other sequence-like data structure. When it comes to strings, slicing allows you to create a new string by specifying a range of indices to extract a substring from the original string.

In [ ]:
```
# string[start:end]
```

- string: The original string you want to slice.
- start: The index from which the slicing begins (inclusive).
- end: The index at which the slicing ends (exclusive).

In [ ]:
```
s = 'hello world'
print(s[1:5]) # from 1 index that is e to 4 index=o
print(s[:6]) # from 0 to 5
print(s[1:]) # from 1 to all
print(s[:]) # all string
print(s[-5:-1]) # with negative slicing
```

```
ello
hello
ello world
hello world
worl
```

In [ ]:
```
# we use step function in slicing
s = 'hello world'
print(s[6:0:-2])
print(s[::-1])
```

```
wol
dlrow olleh
```

In [ ]:
```
s = 'hello world'
print(s[-1:-6:-1])
```

```
dlrow
```

3. Editing and Deleting in Strings

In [ ]:
```
s = 'hello world'
s[0] = 'H'
# it throws error
# Python strings are immutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4564\2237226474.py in <module>
      1 s = 'hello world'
----> 2 s[0] = 'H'
      3
      4 # Python strings are immutable

TypeError: 'str' object does not support item assignment
```

Immutable: Strings in Python are immutable, meaning once you create a string, you cannot change its content. You can create new strings based on the original string, but the original string remains unchanged.

Strings In Python (part-2)

operations on strings

In Python, strings are sequences of characters and are very versatile. You can perform a wide range of operations on strings, including but not limited to:

1. Concatenation

You can concatenate (combine) two or more strings using the '+' operator:

In [ ]:
```
# Arithmatic opertion in string
a = 'Hello'
b = 'world'
print(a + b)

# if we want space
print(a + " " + b)
```

```
Helloworld
Hello world
```

In [ ]:
```
# how to print delhi 14 times
print("delhi"*14)
```

```
delhidelhidelhidelhidelhidelhidelhidelhidelhidelhidelhidelhidelhi
```

In [ ]:
```
print("*"*50)
```

```
**************************************************
```

2. Indexing and Slicing

You can access individual characters of a string using indexing, and you can extract substrings using slicing

In [ ]:
```
s = "Hello world"
```

In [ ]:
```
s[3] # it slice one word from the string
```

Out[ ]:
```
'l'
```

In [ ]:
```
s[4:9]
```

Out[ ]:
```
'o wor'
```

3. String Methods

Python provides many built-in string methods for common operations like converting to uppercase, lowercase, finding substrings, replacing, and more

Upper

In [ ]:
```
text = "Hello, World!"
uppercase_text = text.upper()  # Convert to uppercase
print(uppercase_text)
```

```
HELLO, WORLD!
```

Lower

In [ ]:
```
lowercase_text = text.lower()  # Convert to lowercase
print(lowercase_text)
```

```
hello, world!
```

find

In [ ]:
```
index = text.find("World")     # Find the index of a substring
print(index)
```

```
7
```

replace

In [ ]:
```
new_text = text.replace("Hello", "Hi")  # Replace a substring
print(new_text)
```

```
Hi, World!
```

4. String Formatting

You can format strings using f-strings or the str.format() method

In [ ]:
```
name = 'nitish'
gender = 'male'

'Hi my name is {} and I am a {}'.format(name,gender)
```

Out[ ]:
```
'Hi my name is nitish and I am a male'
```

3. String Splitting and Joining

You can split a string into a list of substrings using the split() method and join a list of strings into one string using the join() method

In [ ]:
```
'hi my name is KHAN '.split()
```

Out[ ]:
```
['hi', 'my', 'name', 'is', 'KHAN']
```

In [ ]:
```
" ".join(['hi', 'my', 'name', 'is', 'KHAN'])
```

Out[ ]:
```
'hi my name is KHAN'
```

4. String Length

You can find the length of a string using the len() function

In [ ]:
```
text = "Hello, World!"
length = len(text)  # Returns the length of the string (13 in this case)
print(length)
```

```
13
```

5. Strip

In [ ]:
```
'hey                              '.strip() # it drop the Unwanted space prenet
```

Out[ ]:
```
'hey'
```

In [ ]:
```

```

Problems on Strings

1 . Find the length of a given string without using the len() function

In [ ]:
```
s = input('enetr the string')

counter = 0

for i in s:
    counter += 1

print(s)
print('lenght of string is',counter)
```

```
Dishant
lenght of string is 7
```

2. Extract username from a given email.

- Eg if the email is xyz76@gmail.com
- then the username should be xyz76

In [ ]:
```
s = input('enter the mail')

pos = s.index('@')

print(s)
print(s[0:pos])
```

```
xyz123@gmail.com
xyz123
```

3. Count the frequency of a particular character in a provided string.

- Eg 'hello how are you' is the string, the frequency of h in this string is 2.

In [ ]:
```
s = input('enter the email')
term = input('what would like to search for')

counter = 0
for i in s:
  if i == term:
    counter += 1

print('frequency',counter)
```

```
frequency 1
```

4. Write a program which can remove a particular character from a string.

In [ ]:
```
s = input('enter the string')
term = input('what would like to remove')
```

```
result = ''

for i in s:
  if i != term:
    result = result + i

print(result)
```

```
hi how are you
```

5. Write a program that can check whether a given string is palindrome or not.

- abba
- malayalam

In [ ]:
```
s = input('enter the string')
print(s)
flag = True
for i in range(0,len(s)//2):
  if s[i] != s[len(s) - i -1]:
    flag = False
    print('Not a Palindrome')
    break

if flag:
  print('Palindrome')
```

```
abba
Palindrome
```

6. Write a program to count the number of words in a string without split()

In [ ]:
```
s = input('enter the string')
print(s)
L = []
temp = ''
for i in s:

  if i != ' ':
    temp = temp + i
  else:
    L.append(temp)
    temp = ''

L.append(temp)
print(L)
```

```
hi how are you
['hi', 'how', 'are', 'you']
```

7. Write a python program to convert a string to title case without using the title()

In [ ]:
```
s = input('enter the string')
print(s)

L = []
for i in s.split():
```

```
  L.append(i[0].upper() + i[1:].lower())

print(" ".join(L))
```

```
how can i help you
How Can I Help You
```

8. Write a program that can convert an integer to string.

In [ ]:
```
number = int(input('enter the number'))
print(number)

digits = '0123456789'
result = ''
while number != 0:
  result = digits[number % 10] + result
  number = number//10

print(result)
print(type(result))
```

```
143243
143243
<class 'str'>
```

In [ ]:
```

```

Interview Question Based on Basics-Operators-If-else-loops-and-Strings-In-PYTHON

1. What is Python? What are the benefits of using Python?

- Python is a high-level, interpreted, general-purpose programming language. Being a general-purpose language, it can be used to build almost any type of application with the right tools/libraries. Additionally, python supports objects, modules, threads, exception-handling, and automatic memory management which help in modelling real-world problems and building applications to solve these problems.

Benefits of using Python:

- Python is a general-purpose programming language that has a simple, easy-to-learn syntax that emphasizes readability and therefore reduces the cost of program maintenance. Moreover, the language is capable of scripting, is completely open-source, and supports third-party packages encouraging modularity and code reuse.
- Its high-level data structures, combined with dynamic typing and dynamic binding, attract a huge community of developers for Rapid Application Development and deployment.

2. What is Python, and why is it often called a "high-level" programming language?

- Python is a high-level programming language known for its simplicity and readability. It is called "high-level" because it abstracts low-level details and provides a more human-readable syntax, making it easier to write and understand code.

3. What is an **Interpreted language**?

- An Interpreted language executes its statements line by line. Languages such as Python, Javascript, R, PHP, and Ruby are prime examples of Interpreted languages. Programs written in an interpreted language runs directly from the source code, with no intermediary compilation step.

4. What is **PEP 8** and why is it important?

- PEP stands for Python Enhancement Proposal. A PEP is an official design document providing information to the Python community, or describing a new feature for Python or its processes. PEP 8 is especially important since it documents the style guidelines for Python Code. Apparently contributing to the Python open-source community requires you to follow these style guidelines sincerely and strictly.

- Read more - https://realpython.com/python-pep8/#:~:text=PEP%208%2C%20sometimes%20spelled%20PEP8,and%20consistency%20of

5. What are the common built-in data types in Python?

Python provides several built-in data types to represent different kinds of data. Some of the common built-in data types in Python include:

1. **int:** This data type is used to represent integers, both positive and negative whole numbers. For example, `x = 5` or `y = -10`.
2. **float:** Floats are used to represent floating-point numbers, which include decimal points. For example, `pi = 3.14159` or `value = 2.5`.
3. **str:** Strings are used to represent sequences of characters, such as text. They can be enclosed in single quotes (' '), double quotes (" "), or triple quotes for multi-line strings. For example, `name = "Alice"` or `sentence = 'Hello, World!'`.
4. **bool:** Booleans represent either `True` or `False`. They are used for logical operations and comparisons. For example, `is_true = True` or `is_false = False`.
5. **list:** Lists are ordered collections of items. They can contain elements of different data types, and the elements can be changed (mutable). For example, `my_list = [1, 2, 'three', True]`.
6. **tuple:** Tuples are similar to lists but are immutable, meaning their elements cannot be changed after creation. They are often used to represent fixed collections of items. For example, `my_tuple = (1, 2, 'three')`.
7. **dict (dictionary):** Dictionaries are collections of key-value pairs, where each key is unique. They are used to store and retrieve data using keys. For example, `person = {'name': 'John', 'age': 30}`.
8. **set:** Sets are unordered collections of unique elements. They are used for tasks like removing duplicates or checking for membership. For example, `my_set = {1, 2, 3}`.
9. **NoneType:** The `None` type represents the absence of a value or a null value. It is often used to indicate that a variable or result has no meaningful value assigned to it.
10. **complex:** Complex numbers represent numbers with both real and imaginary parts. They are written as `a + bj`, where `a` is the real part, and `b` is the imaginary part. For example, `z = 3 + 4j`.
11. **bytes and bytearray:** These data types are used to represent sequences of bytes, often used for binary data or working with file I/O.
12. **range:** The `range` type is used to generate a sequence of numbers, commonly used in for loops and iterations.

13. **datetime:** The `datetime` data type is used to represent dates and times and provides functionality for working with dates and times.

6. Explain the difference between '==' and '!=' operators in Python.

- '==' is used for equality comparison and returns True if two values are equal.
- '!=' is used for inequality comparison and returns True if two values are not equal.

7. Identity operator (is) vs ==?

- Here's the main difference between python "==" vs "is:"
- Identity operators: The "is" and "is not" keywords are called identity operators that compare objects based on their identity. Equality operator: The "==" and "!=" are called equality operators that compare the objects based on their values.

8. Modules vs packages vs Library

Python uses some terms that you may not be familiar with if you're coming from a different language. Among these are modules, packages, and libraries.

- A **module** is a Python file that's intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.
- A **package** is a collection of related modules that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules. This folder will often contain a special **init** file that tells Python it's a package, potentially containing more modules nested within subfolders
- A **library** is an umbrella term that loosely means "a bundle of code." These can have tens or even hundreds of individual modules that can provide a wide range of functionality. Matplotlib is a plotting library. The Python Standard Library contains hundreds of modules for performing common tasks, like sending emails or reading JSON data. What's special about the Standard Library is that it comes bundled with your installation of Python, so you can use its modules without having to download them from anywhere.

9. How do you use the 'if' statement in Python, and what is the purpose of 'elif' and 'else'?

- The 'if' statement is used for conditional execution. 'elif' (short for "else if") is used to test multiple conditions, and 'else' is used to specify code that should be executed when the 'if' or 'elif' conditions are not met.

10. What is the purpose of a 'for' loop in Python, and how does it work?

- A 'for' loop is used for iterating over a sequence (such as a list, tuple, or string) or other iterable objects. It repeatedly executes a block of code for each item in the sequence.

11. How can you check if a substring exists within a given string in Python?

- You can use the 'in' operator to check if a substring exists within a string for example:

In [ ]:
```
text = "Python is great"
if "is" in text:
    print("Substring 'is' found")
```

```
Substring 'is' found
```

List in Python (Part-1)

What are Lists?

List is a data type where you can store multiple items under 1 name. More technically, lists act like dynamic arrays which means you can add more items on the fly.

 Alt text

Characterstics of a List:

- Ordered
- Changeble/Mutable
- Hetrogeneous
- Can have duplicates
- are dynamic
- can be nested
- items can be accessed
- can contain any kind of objects in python

Creating a List

In [ ]:
```
# Empty
print([])

# 1D -> Homo
print([1,2,3,4,5])

# 2D
print([1,2,3,[4,5]])

# 3D
print([[[1,2],[3,4]]])

# Heterogenous
print([1,True,5.6,5+6j,'Hello'])

# Using Type conversion
print(list('hello'))
```

```
[]
[1, 2, 3, 4, 5]
[1, 2, 3, [4, 5]]
[[[1, 2], [3, 4]]]
[1, True, 5.6, (5+6j), 'Hello']
['h', 'e', 'l', 'l', 'o']
```

Accessing Items from a List

- Indexing
- Slicing

In [ ]:
```
# indexing
L = [[[1,2],[3,4]],[[5,6],[7,8]]]
L1 = [1,2,3,4]

# Positive Indexing
print(L1[1:4])

print(L[0][0][1]) # for 2

#How to extract 6
print(L[1][0][1])
```

```
[2, 3, 4]
[]
2
6
```

In [ ]:
```
# Negative indexing
L = [[[1,2],[3,4]],[[5,6],[7,8]]]
L1 = [1,2,3,4]

print(L[-1])

# how to extract 8 with negative
print(L[-1][-1][-1])
```

```
[[5, 6], [7, 8]]
8
```

In [ ]:
```
# Slicing
L = [1,2,3,4,5,6]

print(L[::-1])
```

```
[6, 5, 4, 3, 2, 1]
```

Adding Items to a List

In [ ]:
```
# Apend -> The append method is used to add an item to the end of a list.
L =  [1,2,3,4,5]
L.append(True)
print(L)
```

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

In [ ]:
```
# Extend -> The extend method is used to append elements from an iterable (e.g., ar
L = [1,2,3,4,5]
L.extend([2])
L
```

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

In [ ]:
```
# insert -> The insert method allows you to add an item at a specific position in t
l = [1,2,3,4]

l.insert(1,100)
print(l)
```

```
[1, 100, 2, 3, 4]
```

Editing items in a List

In [ ]:
```
l = [1,2,3,4,5,6]

# editing with indexing
l[-1] = 300
print(l)

# editong with slicing
l[1:4] = [200,300,400]
print(l)
```

```
[1, 2, 3, 4, 5, 300]
[1, 200, 300, 400, 5, 300]
```

Deleting items from a List

In [ ]:
```
# del -> The del statement is used to remove an item from a list based on its index
l = [1,2,3,4,5]

#indexing
del l[2]
print(l)

# slicing
del l[2:4]
print(l)
```

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

In [ ]:
```
# remove -> The remove method is used to remove the first occurrence of a specific
l = [1,2,3]

l.remove(2)

print(l)
```

```
[1, 3]
```

In [ ]:
```
# pop -> The pop method is used to remove and return an item from the list based or
# If you don't provide an index, it will remove and return the last item by default
L = [1,2,3,4,5]

L.pop()

print(L)
```

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

In [ ]:
```
# clear -> The clear method is used to remove all items from the list, effectively
L = [1,2,3,4,5]

L.clear()

print(L)
```

```
[]
```

In [ ]:
```

```

In [ ]:

In [ ]:

List in Python (Part-2)

Operations on Lists:

There are three types of Opeartion in List:

1. Arithmatic
2. Membership
3. Loop

In [ ]:
```
# Arithmatic opeartion (+, *)

L1 = [1,2,3,4,5]
L2 = [9,8,7,6,5]

# concatenation/Merge
print(L1 + L2)


print(L1*3)
print(L2*4)
```

```
[1, 2, 3, 4, 5, 9, 8, 7, 6, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
[9, 8, 7, 6, 5, 9, 8, 7, 6, 5, 9, 8, 7, 6, 5, 9, 8, 7, 6, 5]
```

In [ ]:
```
# membership
L1 = [1,2,3,4,5]
L2 = [1,2,3,4,[5,6]]

print(5 not in L1)
print([5,6] in L2)
```

```
False
True
```

In [ ]:
```
# loops
L1 = [1,2,3,4,5]
L2 = [1,2,3,4,[5,6]]
L3 = [[[1,2],[3,4]],[[5,6],[7,8]]]

for i in L2:
    print(i)
```

```
1
2
3
4
[5, 6]
```

List Functions

In [ ]:
```
# len/min/max/sorted
L = [2,1,5,7,0]

print(len(L))
```

```
print(max(L))
print(min(L))
print(sorted(L))
```

```
5
7
0
[0, 1, 2, 5, 7]
```

In [ ]:
```
# count
l = [1,2,3,456,67867]

l.count(456)
```

Out[ ]:
```
1
```

In [ ]:
```
# index
l = [3,5,7,9,3,23]

l.index(5)
```

Out[ ]:
```
1
```

In [ ]:
```
# reverse
l = [1,2,3,4,6,78]

l.reverse()
print(l)
```

```
[78, 6, 4, 3, 2, 1]
```

In [ ]:
```
# sort vs sorted
L = [2,1,5,7,0]
print(L)

print(sorted(L))

print(L)

L.sort()

print(L)
```

```
[2, 1, 5, 7, 0]
[0, 1, 2, 5, 7]
[2, 1, 5, 7, 0]
[0, 1, 2, 5, 7]
```

If you want to sort a list in-place, you should use the sort method. If you want to create a new sorted list without modifying the original list, you should use the sorted function

List Comprehension

List Comprehension provides a concise way of creating lists.

newlist = [expression **for** item **in** iterable **if** condition == **True**]

Advantages of List Comprehension

- More time-efficient and space-efficient than loops.

- Require fewer lines of code.
- Transforms iterative statement into a formula.

In [ ]:
```
# Add 1 to 10 numbers to the list

# if we use for loop
L=[]
for i in range(1,11):
    L.append(i)

print(L)
```

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

In [ ]:
```
# By List Comprehension
L = [i for i in range(1,11)]
print(L)
```

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

In [ ]:
```
# Scaler multiplication on vecrtor
c = [2,3,4]
v = -3

L = [v*i for i in c]
print(L)
```

```
[-6, -9, -12]
```

In [ ]:
```
# Add sqaures
L = [2,3,4,5,6]

[i**2 for i in L]
```

Out[ ]:
```
[4, 9, 16, 25, 36]
```

In [ ]:
```
# Print all numbers divisible by 5 in the range of 1 to 50

[i for i in range(1,51) if i%5 == 0]
```

Out[ ]:
```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Disadvantages of Python Lists

- Slow
- Risky usage
- eats up more memory

In [ ]:
```
a = [1,2,3]
b = a.copy()

print(a)
print(b)

a.append(4)
print(a)
print(b)

# lists are mutable
```

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

In [ ]:

Tuples In Python

A tuple in Python is an immutable, ordered collection of elements. It is similar to a list in that it can store a variety of data types, including numbers, strings, and other objects, but unlike lists, tuples cannot be modified once they are created.

This immutability means that you can't add, remove, or change elements within a tuple after it's been defined.

Characterstics

- Ordered
- Unchangeble
- Allows duplicate

Creating Tuples

In [ ]:
```
# How to create empty tuple
t1 =()
print(t1)

# creating tuple with single item
t2 = ('hello',)
print(t2)
print(type(t2))

# homo
t3 = (1,2,3,4,5)
print(t3)

# hetero
t4 = (1,2.5,True,[1,2,3])
print(t4)

# tuple
t5 = (1,2,3,(4,5))
print(t5)

# using type conversion
t6 = tuple('hello')
print(t6)
```

```
()
('hello',)
<class 'tuple'>
(1, 2, 3, 4, 5)
(1, 2.5, True, [1, 2, 3])
(1, 2, 3, (4, 5))
('h', 'e', 'l', 'l', 'o')
```

Accessing items in Tuples

- Indexing
- Slicing

In [ ]:
```
print(t3)
```

```
(1, 2, 3, 4, 5)
```

In [ ]:
```
# extract 4 from the tuple
print(t3[3])

# extract 3,4,5
print(t3[2:])
```

```
4
(3, 4, 5)
```

In [ ]:
```
print(t5)
t5[-1][0]
```

```
(1, 2, 3, (4, 5))
```
Out[ ]:
```
4
```

Editing items in tuple/Deleting in tuple

- Tuple are immutable like string it cannot change or Delete

In [ ]:
```
print(t5)
del t5[-1]
```

```
(1, 2, 3, (4, 5))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
c:\Users\disha\Downloads\Pythoon 100 Days\100_Days_OF_Python\Day11 - Tuples In Python.ipynb Cell 10 line 2
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Days/100_Days_OF_Python/Day11%20-%20Tuples%20In%20Python.ipynb#X22sZmlsZQ%3D%3D?line=0'>1</a> print(t5)
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Days/100_Days_OF_Python/Day11%20-%20Tuples%20In%20Python.ipynb#X22sZmlsZQ%3D%3D?line=1'>2</a> del t5[-1]

TypeError: 'tuple' object doesn't support item deletion
```

Operation on Tuples

In [ ]:
```
# + and *
t1 = (1,2,3,4)
t2 = (5,6,7,8)

print(t1 + t2)

# membership
print(1 in t1)

# iteration/loops
for i in t3:
    print(i)
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
True
1
2
3
4
5
```

Tuple Functions

In [ ]:
```
# len/sum/min/max/sorted
t = (1,2,3,4,5)

print(len(t))

print(sum(t))

print(min(t))

print(max(t))

print(sorted(t))
```

```
5
15
1
5
[1, 2, 3, 4, 5]
```

In [ ]:
```
# count
t = (1,2,3,4,5)

t.count(3)
```

Out[ ]:
```
1
```

In [ ]:
```
# index
t = (4,64567,3454,11,33,55)

t.index(64567)
```

Out[ ]:
```
1
```

Special Syntax

In [ ]:
```
# tuple unpacking
a,b,c = (1,2,3)
print(a,b,c)
```

```
1 2 3
```

In [ ]:
```
a = 1
b = 2
a,b = b,a

print(a,b)
```

```
2 1
```

In [ ]:
```
a,b,*others = (1,2,3,4)
print(a,b)
```

```
print(others)
```

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

In [ ]:
```
# zipping tuples
a = (1,2,3,4)
b = (5,6,7,8)

tuple(zip(a,b))
```

Out[ ]:
```
((1, 5), (2, 6), (3, 7), (4, 8))
```

Difference between Lists and Tuples

Lists and tuples are both data structures in Python, but they have several differences in terms of syntax, mutability, speed, memory usage, built-in functionality, error-proneness, and usability. Let's compare them in these aspects:

1. Syntax:
   - Lists are defined using square brackets `[ ]`, e.g., `my_list = [1, 2, 3]`.
   - Tuples are defined using parentheses `( )`, e.g., `my_tuple = (1, 2, 3)`.
2. Mutability:
   - Lists are mutable, which means you can change their elements after creation using methods like `append()`, `insert()`, or direct assignment.
   - Tuples are immutable, which means once created, you cannot change their elements. You would need to create a new tuple if you want to modify its contents.
3. Speed:
   - Lists are slightly slower than tuples in terms of performance because they are mutable. Modifying a list may involve resizing or copying elements, which can introduce overhead.
   - Tuples, being immutable, are generally faster for accessing elements because they are more memory-efficient and do not require resizing or copying.
4. Memory:
   - Lists consume more memory than tuples due to their mutability. Lists require extra memory to accommodate potential resizing and other internal bookkeeping.
   - Tuples are more memory-efficient since they don't have the overhead associated with mutable data structures.
5. Built-in Functionality:
   - Both lists and tuples have common operations like indexing, slicing, and iteration.
   - Lists offer more built-in methods for manipulation, such as `append()`, `insert()`, `remove()`, and `extend()`. Lists are better suited for situations where you need to add or remove elements frequently.
   - Tuples have a more limited set of operations due to their immutability but offer security against accidental modifications.
6. Error-Prone:

- Lists are more error-prone when it comes to accidental modification, especially in large codebases, as they can be changed throughout the code.
- Tuples are less error-prone since they cannot be modified once created, making them more predictable.

7. Usability:

- Lists are typically used when you need a collection of items that can change over time. They are suitable for situations where you want to add, remove, or modify elements.
- Tuples are used when you want to create a collection of items that should not change during the program's execution. They provide safety and immutability.

In summary, the choice between lists and tuples depends on your specific needs. Use lists when you require mutability and dynamic resizing, and use tuples when you want to ensure immutability and need a more memory-efficient, error-resistant data structure.

Sets In Python

In Python, a set is an unordered collection of unique elements. Sets are used to store multiple items, but unlike lists and tuples, they do not allow duplicate values. The elements in a set are enclosed in curly braces '{}' and separated by commas.

Characterstics:

- Unordered
- Mutable
- No Duplicates
- Can't contain mutable data types

Creating Sets

In [ ]:
```
# empty set
s = set()
print(s)
print(type(s))

# 1D and 2D
s1 = {1,2,3}
print(s1)

s2 = {1,2,3,(4,5)}
print(s2)

# Homo and hetro
s3 = {1,'hello',4.5,(1,2,3)}
print(s3)

# using type conversion
s4 = set([1,2,3])
print(s4)

#duplicated not allowed
s5 = {1,1,2,2,3}
print(s5)
```

```
set()
<class 'set'>
{1, 2, 3}
{3, 1, (4, 5), 2}
{1, (1, 2, 3), 'hello', 4.5}
{1, 2, 3}
{1, 2, 3}
```

In [ ]:
```
#sets can't have a mutable items
#s6 ={1,2,3,[3,4]}
#print(s6)
# it thows error
```

In [ ]:
```
# True or false
s1 = {1,2,3}
s2 = {3,2,1}
```

```
print(s1 == s2)
```

```
True
```

Accessing Items

- In Python sets, you cannot access items by indexing or slicing because sets are unordered collections of unique elements, and they do not have a specific order like lists or tuples. Therefore, indexing and slicing operations, which are common with ordered sequences like lists and strings, are not applicable to sets.

Adding Items

In [ ]:
```
#add
s = {1,2,3,4,5}
s.add(5)
print(s)
```

```
{1, 2, 3, 4, 5}
```

In [ ]:
```
#update
s.update([5,6,7])
print(s)
```

```
{1, 2, 3, 4, 5, 6, 7}
```

Deleting Items

Sets Doesn't support item Deletion

In [ ]:
```
# del
s= {1,2,3,4,5}
del(s)
# it complete delete set
```

In [ ]:
```
# Discard
s = {1,2,3,4}
s.discard(3)
print(s)
```

```
{1, 2, 4}
```

In [ ]:
```
#remove
s = {1,2,3,4}
s.remove(4)
print(s)
```

```
{1, 2, 3}
```

In [ ]:
```
# pop
s.pop()
# it select random number from set
```

Out[ ]:
```
2
```

In [ ]:
```
# clear
s.clear()
print(s)
```

```
set()
```

Set opeartion

In [ ]:
```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}

# union
print(s1 | s2)

# intersection
print(s1 & s2)

# Differences
print(s1 - s2)
print(s2 - s1)

# symmetric diffrences
print(s1 ^ s2)

# membership test
print(1 in s1)
print(1 not in s1)

# iteration
for i in s1:
    print(i)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5}
{1, 2, 3}
{8, 6, 7}
{1, 2, 3, 6, 7, 8}
True
False
1
2
3
4
5
```

Set Function

In [ ]:
```
# len/sum/min/max/sorted
s = {3,1,4,5,2,7}

print(len(s))

print(sum(s))

print(min(s))

print(max(s))

print(sorted(s))
```

```
6
22
1
7
[1, 2, 3, 4, 5, 7]
```

In [ ]:
```python
# union/update
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}

# s1 | s2
s1.union(s1)

s1.update(s2)
print(s1)
print(s2)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5, 6, 7, 8}
```

In [ ]:
```python
# intersection/intersection_update
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}

s1.intersection(s2)

s1.intersection_update(s2)
print(s1)
print(s2)
```

```
{4, 5}
{4, 5, 6, 7, 8}
```

In [ ]:
```python
# difference/difference_update
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}

s1.difference(s2)

s1.difference_update(s2)
print(s1)
print(s2)
```

```
{1, 2, 3}
{4, 5, 6, 7, 8}
```

In [ ]:
```python
# symmetric_difference/symmetric_difference_update
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}

s1.symmetric_difference(s2)

s1.symmetric_difference_update(s2)
print(s1)
print(s2)
```

```
{1, 2, 3, 6, 7, 8}
{4, 5, 6, 7, 8}
```

In [ ]:
```python
# isdisjoint/issubset/issuperset
s1 = {1,2,3,4}
s2 = {7,8,5,6}

s1.isdisjoint(s2)
```

Out[ ]:
```
True
```

In [ ]:
```
s1 = {1,2,3,4,5}
s2 = {3,4,5}

s1.issuperset(s2)
```

Out[ ]:
```
True
```

In [ ]:
```
# copy
s1 = {1,2,3}
s2 = s1.copy()

print(s1)
print(s2)
```

```
{1, 2, 3}
{1, 2, 3}
```

Frozenset

Frozen set is just an immutable version of a Python set object

In [ ]:
```
# create frozenset
fs1 = frozenset([1,2,3])
fs2 = frozenset([3,4,5])

fs1 | fs2
```

Out[ ]:
```
frozenset({1, 2, 3, 4, 5})
```

Set Comprehension

In [ ]:
```
# examples

{i**2 for i in range(1,11) if i>5}
```

Out[ ]:
```
{36, 49, 64, 81, 100}
```

In [ ]:
```
{i+2 for i in range(1,11) if i>5}
```

Out[ ]:
```
{8, 9, 10, 11, 12}
```

Dictionary In Python

Dictionary in Python is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.

In some languages it is known as map or assosiative arrays.

dict = { 'name' : 'xyz' , 'age' : 24 , 'gender' : 'male' }

Characterstics:

- Mutable
- Indexing has no meaning
- keys can't be duplicated
- keys can't be mutable items

Create Dictionary

In [ ]:
```
# empty
d = {}
print(d)

#1D
d1 = {'name':'xyz', 'Age':23, 'gender':'male'}
print(d1)

# with mixed keys
d2 = {(2,2,3):1,'hello':'world'}
print(d2)

# 2D dictionary
s = {
    'name':'ramesh',
     'college':'bit',
     'sem':4,
     'subjects':{
         'dsa':50,
         'maths':67,
         'english':34
     }
}
print(s)

# using sequence and dict function
d4 = dict([('name','xyz'),('age',23),(3,3)])
print(d4)

#mutable items keys
d6 = {'name':'xyz', (1,2,3):2}
print(d6)
```

```
{}
{'name': 'xyz', 'Age': 23, 'gender': 'male'}
{(2, 2, 3): 1, 'hello': 'world'}
{'name': 'ramesh', 'college': 'bit', 'sem': 4, 'subjects': {'dsa': 50, 'maths': 6
7, 'english': 34}}
{'name': 'xyz', 'age': 23, 3: 3}
{'name': 'xyz', (1, 2, 3): 2}
```

Accessing Items

In [ ]:
```
my_dict = {'name': 'Jack', 'age': 26}
my_dict['name'] # you have to write keys
```

Out[ ]:
```
'Jack'
```

Adding key pair

In [ ]:
```
print(d4)
```

```
{'name': 'xyz', 'age': 23, 3: 3}
```

In [ ]:
```
d4['gender'] = 'male'
print(d4)

d4['weight'] = 70
print(d4)
```

```
{'name': 'xyz', 'age': 23, 3: 3, 'gender': 'male'}
{'name': 'xyz', 'age': 23, 3: 3, 'gender': 'male', 'weight': 70}
```

Remove key-value pair

In [ ]:
```
d = {'name': 'xyz', 'age': 24, 3: 3, 'gender': 'male', 'weight': 72}

# pop
d.pop(3) # it remove three
print(d)

#popitems
d.popitem() # it remove last item in the dictionary
print(d)

# del
del d['name']
print(d)

#clear
d.clear() # it clear dictionary
print(d)
```

```
{'name': 'xyz', 'age': 24, 'gender': 'male', 'weight': 72}
{'name': 'xyz', 'age': 24, 'gender': 'male'}
{'age': 24, 'gender': 'male'}
{}
```

Editing key-value pair

In [ ]:
```
print(s)
```

```
{'name': 'ramesh', 'college': 'bit', 'sem': 4, 'subjects': {'dsa': 50, 'maths': 6
7, 'english': 34}}
```

In [ ]:
```
s['subjects']['dsa'] = 80
s
```

Out[ ]:
```
{'name': 'ramesh',
 'college': 'bit',
 'sem': 4,
 'subjects': {'dsa': 80, 'maths': 67, 'english': 34}}
```

Dictionary Operations

- Membership
- Iteration

In [ ]:
```
print(s)
```

```
{'name': 'ramesh', 'college': 'bit', 'sem': 4, 'subjects': {'dsa': 80, 'maths': 6
7, 'english': 34}}
```

In [ ]:
```
# membership
'name' in s
```

Out[ ]:
```
True
```

In [ ]:
```
'ramesh' in s # it use on it keys not on values
```

Out[ ]:
```
False
```

In [ ]:
```
# ITERATION
for i in s:
  print(i,s[i])
```

```
name ramesh
college bit
sem 4
subjects {'dsa': 80, 'maths': 67, 'english': 34}
```

Dictionary Functions

In [ ]:
```
print(s)
```

```
{'name': 'ramesh', 'college': 'bit', 'sem': 4, 'subjects': {'dsa': 80, 'maths': 6
7, 'english': 34}}
```

In [ ]:
```
# len/sorted
print(len(s))

sorted(s)
```

Out[ ]:
```
4
['college', 'name', 'sem', 'subjects']
```

In [ ]:
```
# items/keys/values

print(s.items())

print(s.keys())
```

```
print(s.values())
```

```
dict_items([('name', 'ramesh'), ('college', 'bit'), ('sem', 4), ('subjects', {'ds
a': 80, 'maths': 67, 'english': 34})])
dict_keys(['name', 'college', 'sem', 'subjects'])
dict_values(['ramesh', 'bit', 4, {'dsa': 80, 'maths': 67, 'english': 34}])
```

In [ ]:
```
# update
d1 = {1:2,3:4,4:5}
d2 = {4:7,6:8}

d1.update(d2)
print(d1)
```

```
{1: 2, 3: 4, 4: 7, 6: 8}
```

Dictionary Comprehension

In [ ]:
```
# print 1st 10 numbers and their squares
{i:i**2 for i in range(1,11)}
```

Out[ ]:
```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

In [ ]:
```
distances = {'delhi':1000,'mumbai':2000,'bangalore':3000}
print(distances.items())
```

```
dict_items([('delhi', 1000), ('mumbai', 2000), ('bangalore', 3000)])
```

In [ ]:
```
# using zip
days = ["Sunday", "Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]
temp_C = [30.5,32.6,31.8,33.4,29.8,30.2,29.9]

{i:j for (i,j) in zip(days,temp_C)}
```

Out[ ]:
```
{'Sunday': 30.5,
 'Monday': 32.6,
 'Tuesday': 31.8,
 'Wednesday': 33.4,
 'Thursday': 29.8,
 'Friday': 30.2,
 'Saturday': 29.9}
```

In [ ]:
```
# using if condition
products = {'phone':10,'laptop':0,'charger':32,'tablet':0}

{key:value for (key,value) in products.items() if value>0}
```

Out[ ]:
```
{'phone': 10, 'charger': 32}
```

In [ ]:
```
# Nested Comprehension
# print tables of number from 2 to 4
{i:{j:i*j for j in range(1,11)} for i in range(2,5)}
```

Out[ ]:
```
{2: {1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18, 10: 20},
 3: {1: 3, 2: 6, 3: 9, 4: 12, 5: 15, 6: 18, 7: 21, 8: 24, 9: 27, 10: 30},
 4: {1: 4, 2: 8, 3: 12, 4: 16, 5: 20, 6: 24, 7: 28, 8: 32, 9: 36, 10: 40}}
```

OOP in Python

(Object-Oriented Programming)

- Object-Oriented Programming (OOP) is a programming paradigm that uses objects to organize and structure code.
- Python is a multi-paradigm programming language, and it supports object-oriented programming.
- OOP is a powerful way to design and structure your code, making it more modular, reusable, and maintainable.
- In Python, **everything is an object**, and you can use OOP principles to create and manipulate these objects.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

1. Python Class

- A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

In [ ]:
```
class ClassName:
    # Statement-1
    .
    .
```

```
    .
    # Statement-N
```

Creating an Empty Class in Python

In [ ]:
```
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

2. Python Objects

- The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.
- More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

In [ ]:
```
obj = Dog()
```

The Python self

When working with classes in Python, the term "self" refers to the instance of the class that is currently being used. It is customary to use "self" as the first parameter in instance methods of a class. Whenever you call a method of an object created from a class, the object is automatically passed as the first argument using the "self" parameter. This enables you to modify the object's properties and execute tasks unique to that particular instance.

In [ ]:
```
class mynumber:
    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(self.value)

obj1 = mynumber(17)
obj1.print_value()
```

```
17
```

Python Class self Constructor

When working with classes, it's important to understand that in Python, a class constructor is a special method named **init** that gets called when you create an instance (object) of a class. This method is used to initialize the attributes of the object. Keep in mind that the self parameter in the constructor refers to the instance being created and allows you to access

and set its attributes. By following these guidelines, you can create powerful and efficient classes in Python.

In [ ]:
```
# _init_
class Subject:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


obj = Subject('Maths', 'Science')
print(obj.attr1)
print(obj.attr2)
```

```
Maths
Science
```

OOP in Python Part - 2

What is Function and method in Python??

In Python, functions and methods are both used to define blocks of reusable code, but they have some key differences:

1. Function:

- A function is a block of organized, reusable code that performs a specific task.
- Functions in Python are defined using the def keyword, followed by a function name, a set of parameters in parentheses, a colon, and the function's code block.
- Functions can take input parameters (arguments) and return a value using the return statement. However, not all functions need to return a value.
- Functions can be called by their name, and they are usually defined at the module level (i.e., not inside a class or another function).

2. Method:

- A method is similar to a function, but it is associated with an object or a class.
- Methods are defined within a class and are used to define the behaviors or actions that objects of that class can perform.
- Methods are called on instances of a class and typically operate on the data (attributes) of the instance.
- The first parameter of a method is usually named 'self', which refers to the instance on which the method is called.

In summary, functions are standalone blocks of code, while methods are functions defined within a class and are associated with instances of that class. Methods often work with the data (attributes) of the instance, whereas functions can be called independently of any specific object or class.

In [ ]:
```
# _init_
class Subject:

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2


obj = Subject('Maths', 'Science')
print(obj.attr1)
print(obj.attr2)
```

Magic Methods:

In Python, "magic methods," also known as "dunder methods" (short for "double underscore methods"), are special methods that have double underscores at the beginning and end of their names, such as _ *init* , *str* , *add* _ , etc. These methods are also sometimes referred to as "special methods" or "dunder methods" because of their naming convention.

Magic methods are used to define how objects of a class behave in various situations and allow you to customize the behavior of your custom classes. They are automatically invoked by the Python interpreter in response to specific operations or built-in functions.

1. init :

- The _ *init* _ method in Python is like a special function that gets called when you create a new object from a class.
- It's often referred to as the "constructor" because it helps you set up, or "initialize," the initial state of an object.

Parameterized constructor -> It wants input while form an object

In [ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

In [ ]:
```
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
# person1 will have name set to "Alice" and age set to 30, while person2 will have
```

In [ ]:
```
print(person1)
```

```
<__main__.Person object at 0x00000220D6DEAD90>
```

- when we print person 1 then python don't know how to show Alice with age 30 ..python show only Memory location where it store
- so for we call str

2. str :

- The **str** method in Python is a special method (often called a "magic method" or "dunder method") that you can define in your custom classes to provide a **human-readable string** representation of objects created from that class.
- This method is automatically called when you use the str() function or the print() function on an object.

In [ ]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def __str__(self):
        return f"{self.name} ({self.age} years old)"

# Create a Person object
person = Person("Alice", 30)

# When you use str() or print(), Python calls __str__ automatically
print(str(person))  # Output: "Alice (30 years old)"
print(person)       # Output: "Alice (30 years old)"
```

```
Alice (30 years old)
Alice (30 years old)
```

Some Problems On OOP:

Write OOP classes to handle the following scenarios:

- A user can create and view 2D coordinates
- A user can find out the distance between 2 coordinates
- A user can find find the distance of a coordinate from origin
- A user can check if a point lies on a given line
- A user can find the distance between a given 2D point and a given line

1. A user can create and view 2D co-ordinates

In [ ]:
```
class Point:

    def __init__(self,x,y):
        self.x_cod = x
        self.y_cod = y

    def __str__(self):
        return "<{},{}>".format(self.x_cod,self.y_cod)
```

In [ ]:
```
x = 4
y = 5
print(Point(4,5))
```

```
<4,5>
```

2 . A user can find out the distance between 2 coordinates

In [ ]:
```
class Point:

    def __init__(self,x,y):
        self.x_cod = x
        self.y_cod = y

    def __str__(self):
        return "<{},{}>".format(self.x_cod,self.y_cod)

    def euclidean_distance(self,other):
        return ((self.x_cod-self.y_cod)**2 + (self.y_cod-self.x_cod)**2)**0.5
```

In [ ]:
```
p1 = Point(0,1)
p2 = Point(1,1)

p1.euclidean_distance(p2)
```

Out[ ]:
```
1.4142135623730951
```

1. A user can find find the distance of a coordinate from origin

In [ ]:
```
class Point:
    def __init__(self,x,y):
        self.x_cod = x
        self.y_cod = y
```

```
    def distance_from_origin(self):
        return (self.x_cod**2 + self.y_cod**2)**0.5
```

In [ ]:
```
p1 = Point(0,10)
p2 = Point(1,10)

print(p1.distance_from_origin())
print(p2.distance_from_origin())
```

```
10.0
10.04987562112089
```

1. A user can check if a point lies on a given line

In [ ]:
```
class Line:

    def __init__(self,A,B,C):
        self.A = A
        self.B = B
        self.C = C

    def __str__(self):
        return '{}x + {}y + {} = 0'.format(self.A, self.B, self.C)

    def point_on_line(line,point):
        if line.A*point.x_cod + line.B*point.y_cod + line.C == 0:
            return 'point lies on line'
        else:
            return 'does not on line'
```

In [ ]:
```
l1 = Line(1,1,-2)
p1 = Point(1,10)

l1.point_on_line(p2)
```

Out[ ]:
```
'does not on line'
```

In [ ]:
```
class Line:

  def __init__(self,A,B,C):
    self.A = A
    self.B = B
    self.C = C

  def shortest_distance(line,point):
    return abs(line.A*point.x_cod + line.B*point.y_cod + line.C)/(line.A**2 + line.
```

In [ ]:
```
l1 = Line(1,1,-2)
p1 = Point(1,10)

l1.shortest_distance(p1)
```

Out[ ]:
```
6.363961030678928
```

Encapsulation - OOP in Python

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) and is supported in Python, just like in many other object-oriented languages. It refers to the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. Additionally, it restricts direct access to some of the object's components, providing a controlled and well-defined interface to interact with the object.

In Python, encapsulation is implemented using access modifiers and naming conventions:

1. **Public:** Attributes and methods that are accessible from outside the class without any restrictions. By default, all attributes and methods in Python are public.

2. **Protected:** Attributes and methods that are intended to be used only within the class and its subclasses. In Python, you can denote a protected attribute or method by prefixing it with a single underscore (e.g., `_my_attribute`).

3. **Private:** Attributes and methods that are intended to be used only within the class. In Python, you can denote a private attribute or method by prefixing it with a double underscore (e.g., `__my_attribute`).

In [ ]:
```
# if we have a one class

class Myclass:
    def __init__(self):
        self.public_attribute = 55
        self._protected_attribute = 'Hello'
        self.__private_attribute = 'World'

    def public_method(self):
        return "This is public method"

    def _protected_method(self):
        return "This is Protected Method"

    def __private_method(self):
        return "this is private Method"
```

In [ ]:
```
obj = Myclass()
```

In [ ]:
```
# we have to access the public attribaute dirctly and don't give any tyoe of error

print(obj.public_attribute)
print(obj.public_method())
```

```
55
This is public method
```

In [ ]:
```
# so now we try for Protected attribute
print(obj._protected_attribute)
print(obj._protected_method())
```

```
Hello
This is Protected Method
```

In [ ]:
```
# Accessing private attributes and methods directly will result in an error:
print(obj.__private_attribute)   # Raises an AttributeError
print(obj.__private_method())    # Raises an AttributeError
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
c:\Users\disha\Downloads\Python 100 Days\100_Days_OF_Python\Day17 - Encapsulation in OOP in Python.ipynb Cell 7 line 2
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=0'>1</a> # Accessing private attributes and methods directly will result in an error:
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=1'>2</a> print(obj.__private_attribute)   # Raises an AttributeError
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day17%20-%20Encapsulation%20in%20OOP%20in%20Python.ipynb#W6sZmlsZQ%3D%3D?line=2'>3</a> print(obj.__private_method())

AttributeError: 'Myclass' object has no attribute '__private_attribute'
```

You can undestand with Encapsulation with this short story:

In a bustling city, there was a high-tech security system protecting a valuable treasure. The system had a keypad (public interface) that allowed authorized personnel to enter a secret code. Behind the keypad was a locked room (protected layer) with advanced security features, and inside that room was a safe (private layer) holding the treasure. Only a select few knew the secret code, ensuring the treasure's safety while providing access to those who needed it. This is encapsulation in action, safeguarding valuable data while allowing controlled access.

Top five benefits of encapsulation:

1. **Controlled Access:** Encapsulation restricts direct access to internal data, ensuring that it is only modified through well-defined methods, which helps prevent unintended errors and maintain data integrity.

2. **Modularity:** Encapsulation promotes modular code by isolating the implementation details of a class. This makes it easier to update or replace components without affecting the rest of the system.

3. **Information Hiding:** It hides complex internal details, reducing the complexity for users of a class and allowing developers to change the implementation without impacting external code.

4. **Security:** By controlling access to sensitive data and methods, encapsulation enhances security, reducing the risk of unauthorized or malicious manipulation.

5. **Reusability:** Encapsulation facilitates the creation of reusable and self-contained classes, which can be easily integrated into different parts of a program or shared across projects, saving development time and effort.

Common uses of encapsulation in programming:

1. **Data Protection:** Encapsulation restricts direct access to data, ensuring that it can only be modified through controlled methods. This protects data integrity and prevents unintended modifications or errors.

2. **Abstraction:** It allows you to present a simplified and high-level view of an object, hiding the complex implementation details. This abstraction makes it easier for users of the class to understand and work with it.

3. **Code Organization:** Encapsulation helps organize code by bundling related data and methods within a class. This modular approach enhances code readability and maintainability, making it easier to manage large codebases.

4. **Security:** By encapsulating sensitive data and operations, you can control who has access to them. This enhances security by preventing unauthorized access or manipulation of critical information.

5. **Reusability:** Encapsulated classes can be reused in different parts of a program or in different projects. This reusability saves development time and promotes the creation of robust, well-tested components.

Class Relationships in object-oriented programming (OOP) with Python:

1. **Aggregation (Has-a Relationship):**
    - **Definition:** Aggregation is a class relationship where one class (the container or composite class) contains or references another class (the contained or component class) as part of its structure. It models a "has-a" relationship, where the container class has one or more instances of the contained class.
    - **Usage:** Aggregation is often used to model objects that are composed of or contain other objects. It can be implemented using attributes or references in the container class.
    - **Example:** In a university system, the `University` class may contain `Department` objects, and each `Department` contains `Professor` objects. This represents an aggregation relationship because a university has departments, and each department has professors.
2. **Inheritance (Is-a Relationship):**
    - **Definition:** Inheritance is a type of class relationship where a subclass (derived class) inherits properties and behaviors from a superclass (base class). It models an "is-a" relationship, where the subclass is a more specific or specialized version of the superclass.
    - **Usage:** In Python, inheritance is established using the `class SubClass(BaseClass)` syntax. The subclass inherits attributes and methods from the superclass and can also add its own or override the inherited ones.
    - **Example:** If you have a base class `Vehicle`, you can create subclasses like `Car` and `Bicycle` that inherit attributes and methods from `Vehicle` while adding their own specific properties.

These two class relationships, inheritance and aggregation, are fundamental concepts in OOP and are used extensively in software design to model different types of relationships between classes and objects.

1. Aggregation(Has a Realtionship)

In object-oriented programming (OOP) with Python, aggregation is a type of association between classes where one class, known as the "container" or "composite" class, contains or references another class, known as the "contained" or "component" class. Aggregation represents a "has-a" relationship, where the container class has one or more instances of the contained class as part of its own structure.

Aggregation is often used to model relationships between objects when one object is composed of or contains other objects. It is a way to create more complex objects by combining simpler objects. A classic example of aggregation is a university system where a

University class can contain **Department** objects, and each Department can contain `**Professor** objects.

In [ ]:
```
class Professor:
    def __init__(self, name):
        self.name = name

class Department:
    def __init__(self, name):
        self.name = name
        self.professors = []  # Aggregation: Department contains Professor objects

class University:
    def __init__(self, name):
        self.name = name
        self.departments = []  # Aggregation: University contains Department objec

# Creating objects
professor1 = Professor("John Doe")
professor2 = Professor("Jane Smith")

department1 = Department("Computer Science")
department1.professors.append(professor1)
department1.professors.append(professor2)

university = University("ABC University")
university.departments.append(department1)
```

In this example:

- 'University' has an aggregation relationship with 'Department' because it contains a list of 'Department' objects.
- 'Department' has an aggregation relationship with 'Professor' because it contains a list of Professor objects.

Aggregation is a way to represent the idea that one class is composed of or contains instances of another class, and it is often used to model real-world relationships between objects in a system.

Top five reasons why we use aggregation in object-oriented programming:

1. **Modularity and Code Reusability:** Aggregation allows for the creation of modular, reusable components, making it easier to build and maintain complex systems.

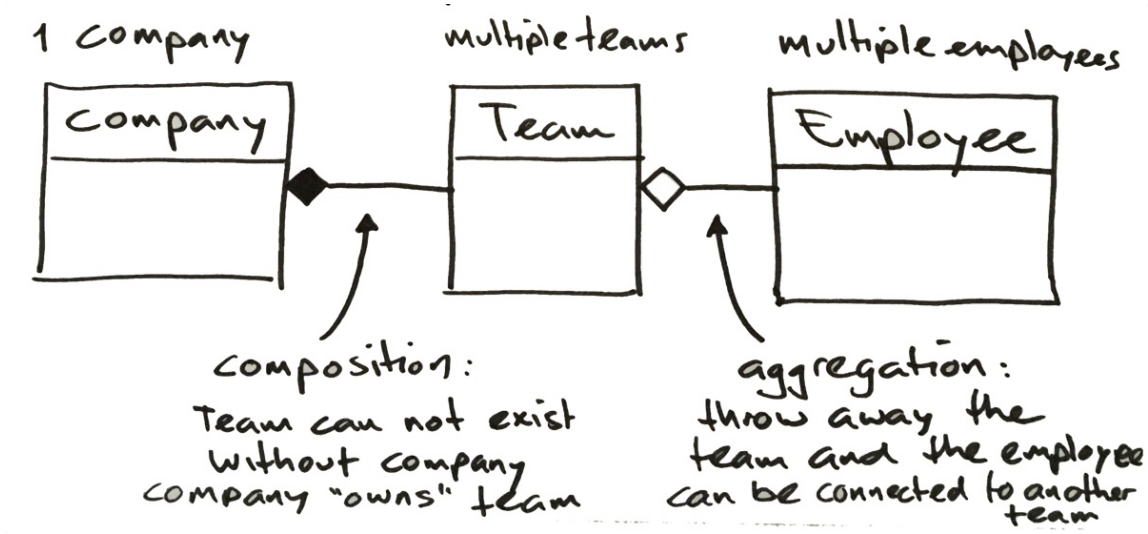
2. **Clearer Object Relationships:** It helps model real-world relationships accurately, enhancing the code's clarity and alignment with the problem domain.

3. **Flexible System Design:** Aggregation permits changes and extensions to be made to the codebase without affecting the entire system, ensuring adaptability to evolving requirements.

4. **Enhanced Encapsulation:** It supports improved data and behavior encapsulation by encapsulating the interaction details between components within the container class.

5. **Efficient Resource Utilization:** Aggregation can lead to efficient memory usage by sharing component instances among multiple container objects, making it particularly useful in resource-constrained environments.

One example of class diagram of Aggreagation



- The solid diamond indicates Composition. Notice how teams belong to the single company object. If the company object is deleted, the teams will have no reason to exist anymore.
- The open diamond indicates Aggregation. When a team is deleted, the employees that were in the team, still exist. Employees might also belong to multiple teams. A team object does not "own" an employee object.

Inheritance in OOP

What is Inheritance(is a relationship)

Inheritance is one of the fundamental concepts in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit properties and behaviors (attributes and methods) from an existing class (base class or superclass). This allows you to create a new class that is a modified or specialized version of an existing class, promoting code reuse and establishing a hierarchical relationship between classes.

Inheritance is typically used to model an "is-a" relationship between classes, where the derived class is a more specific or specialized version of the base class. The derived class inherits the attributes and methods of the base class and can also have its own additional attributes and methods or override the inherited ones.

In [ ]:
```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # Base class method, to be overridden by subclasses

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Creating objects
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Calling the speak method on objects
print(dog.speak())
print(cat.speak())
```

```
Buddy says Woof!
Whiskers says Meow!
```

In this example:

- 'Animal' is the base class, and 'Dog' and 'Cat' are subclasses of 'Animal'.
- Both 'Dog' and 'Cat' inherit the 'name' attribute and the 'speak' method from the 'Animal' class.
- However, each subclass overrides the 'speak' method to provide its own implementation, representing the specific behavior of each animal. Inheritance is a powerful mechanism in OOP because it allows you to create a hierarchy of classes, with each level of the hierarchy adding or modifying functionality as needed. This promotes code reuse, encapsulation, and abstraction, making it easier to manage and extend your codebase.

Inheritance is a fundamental concept in object-oriented programming (OOP) that offers several benefits for software development. Here are five key advantages of inheritance:

1. **Code Reusability:** Inheritance allows you to reuse code from existing classes (base or parent classes) in new classes (derived or child classes). This promotes code reusability, reduces redundancy, and saves development time. Developers can leverage well-tested and established code when creating new classes.

2. **Hierarchical Structure:** Inheritance enables the creation of a hierarchical structure of classes, with each derived class inheriting attributes and methods from its parent class. This hierarchical organization makes it easier to understand and manage the relationships between different classes in a complex system.

3. **Promotes Polymorphism:** Inheritance is closely linked to the concept of polymorphism, which allows objects of different classes to be treated as objects of a common base class. This promotes flexibility and extensibility in your code, making it easier to work with diverse types of objects in a unified manner.

4. **Supports Code Extensibility:** With inheritance, you can extend existing classes to add or modify functionality. Derived classes can override inherited methods to provide specialized behavior while still benefiting from the base class's shared attributes and methods. This makes it straightforward to adapt and extend your code to accommodate changing requirements.

5. **Enhances Maintenance:** Inheritance improves code maintenance because changes made to the base class are automatically reflected in all its derived classes. This reduces the risk of introducing bugs during updates and ensures that modifications are consistently applied throughout the codebase. It also helps maintain a consistent interface for objects derived from the same base class.

In summary, inheritance is a powerful mechanism in OOP that promotes code reusability, structure, flexibility, extensibility, and ease of maintenance. It is a cornerstone of building complex software systems by organizing and leveraging existing code effectively.

Inheritance in summary

- A class can inherit from another class.

- Inheritance improves code reuse
- Constructor, attributes, methods get inherited to the child class
- The parent has no access to the child class
- Private properties of parent are not accessible directly in child class
- Child class can override the attributes or methods. This is called method overriding
- super() is an inbuilt function which is used to invoke the parent class methods and constructor

Types of Inheritance

1. **Single Inheritance:**
    - In single inheritance, a class inherits properties and behaviors from a single parent class. This is the simplest form of inheritance, where each class has only one immediate base class.
2. **Multilevel Inheritance:**
    - In multilevel inheritance, a class derives from a class, which in turn derives from another class. This creates a chain of inheritance where each class extends the previous one. It forms a hierarchy of classes.
3. **Hierarchical Inheritance:**
    - In hierarchical inheritance, multiple derived classes inherit from a single base or parent class. This results in a structure where several classes share a common ancestor.
4. **Multiple Inheritance (Diamond Problem):**
    - Multiple inheritance occurs when a class inherits properties and behaviors from more than one parent class. This can lead to a complication known as the "diamond problem," where ambiguity arises when a method or attribute is called that exists in multiple parent classes. Some programming languages, like Python, provide mechanisms to resolve this ambiguity.
5. **Hybrid Inheritance:**
    - Hybrid inheritance is a combination of two or more types of inheritance mentioned above. It is used to model complex relationships in a system where multiple inheritance hierarchies may intersect.

These different types of inheritance allow developers to model various relationships and structures in object-oriented programming. Choosing the appropriate type of inheritance depends on the specific requirements and design of the software being developed.

In [ ]:
```
# single inheritance
class Phone:
    def __init__(self, price, brand, camera):
```

```
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

```
Inside phone constructor
Buying a phone
```

In [ ]:
```
# multilevel
class Product:
    def review(self):
        print ("Product customer review")

class Phone(Product):
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass
```

In [ ]:
```
# Hierarchical
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass
```

In [ ]:
```
# Multiple
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def review(self):
        print ("Customer review")
```

```
class SmartPhone(Phone, Product):
    pass
```

Polymorphism and Abstraction - Object-Oriented Programming (OOP)

Polymorphism:

Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP), along with encapsulation, inheritance, and abstraction. It allows objects of different classes to be treated as objects of a common superclass. In Python, polymorphism is implemented through method overriding and method overloading, which are two related concepts.

1. Method Overriding

2. Method Overloading

1. Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to provide its own behavior for a method while still adhering to the method signature defined in the superclass.

Here's an example of method overriding in Python:

In [ ]:
```
class Animal:
    def speak(self):
        pass

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

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

dog = Dog()
cat = Cat()

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

```
Woof!
Meow!
```

2.Method Overloading:

Method overloading allows a class to define multiple methods with the same name but different parameters. Python does not support traditional method overloading with different

parameter types like some other languages (e.g., Java or C++). Instead, Python achieves a form of method overloading using default arguments and variable-length argument lists.

Here's an example:

In [ ]:
```
class Calculator:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

calc = Calculator()

#result1 = calc.add(1, 2)        # Error: The second add method with three paramete
result2 = calc.add(1, 2, 3)     # This works fine.
```

In Python, only the latest defined method with a particular name is accessible. Traditional method overloading with different parameter types isn't supported.

In [ ]:
```
class Shape:

  def area(self,a,b=0):
    if b == 0:
      return 3.14*a*a
    else:
      return a*b

s = Shape()

print(s.area(2))
print(s.area(3,4))
```

```
12.56
12
```

Polymorphism allows you to write more flexible and reusable code by treating different objects in a consistent way, regardless of their specific class. This promotes code flexibility and makes it easier to extend and maintain your programs as they grow in complexity.

Abstraction

Abstraction is one of the four fundamental principles of Object-Oriented Programming (OOP), along with encapsulation, inheritance, and polymorphism. Abstraction is the process of simplifying complex reality by modeling classes based on the essential properties and behaviors of objects, while ignoring or hiding the non-essential details.

In software development, abstraction allows you to create a simplified representation of an object or system that focuses on what's relevant to your application and hides the unnecessary complexity. Here are some key aspects of abstraction:

1. **Modeling:** Abstraction involves defining classes and objects that capture the essential characteristics and behaviors of real-world entities or concepts. You create classes to represent these abstractions, defining their attributes (data) and methods (behavior) to interact with them.

2. **Hiding Complexity:** Abstraction allows you to hide the internal details and complexities of an object's implementation from the outside world. This is achieved through encapsulation, where you define private attributes and provide public methods to interact with the object.

3. **Generalization:** Abstraction often involves creating abstract classes or interfaces that define a common set of attributes and behaviors shared by multiple related classes. This promotes code reusability and flexibility through inheritance.

4. **Focus on What, Not How:** When you work with abstractions, you can focus on using objects based on what they do (their methods) rather than how they do it (their implementation details). This separation of concerns simplifies the design and maintenance of complex systems.

Here's a simple example of abstraction in Python:

In [ ]:
```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        pass

    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        print(f"{self.make} {self.model} is starting")

    def stop(self):
        print(f"{self.make} {self.model} is stopping")

class Motorcycle(Vehicle):
    def start(self):
        print(f"{self.make} {self.model} is revving up")

    def stop(self):
        print(f"{self.make} {self.model} is slowing down")
```

In this example, the `Vehicle` class represents an abstraction of a general vehicle with common attributes (make and model) and methods (start and stop). Concrete subclasses like `Car` and `Motorcycle` inherit from `Vehicle` and provide specific implementations for the start and stop methods. Users of these classes can interact with them without needing to know the exact implementation details of each vehicle type.

Abstraction helps in managing complexity, improving code organization, and making code more maintainable and understandable by focusing on high-level concepts and behaviors.

How to Use Abstraction:

In [ ]:
```
from abc import ABC,abstractmethod
class BankApp(ABC):
```

```
  def database(self):
    print('connected to database')

  @abstractmethod    # This is absreactmethod
  def security(self):
    pass

  @abstractmethod
  def display(self):
    pass
```

In [ ]:
```
class MobileApp(BankApp):

  def mobile_login(self):
    print('login into mobile')

  def security(self):         # if we use some function in abstractmethod then we h
    print('mobile security')

  def display(self):
    print('display')
```

In [ ]:
```
mob = MobileApp()
```

In [ ]:
```
mob.security()
```

```
mobile security
```

In [ ]:
```
obj = BankApp()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
c:\Users\disha\Downloads\Pythoon 100 Days\100_Days_OF_Python\Polymorphism and Abst
raction - Object-Oriented Programming (OOP).ipynb Cell 19 line 1
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Pythoon%20100%20Da
ys/100_Days_OF_Python/Polymorphism%20and%20Abstraction%20-%20Object-Oriented%20Pro
gramming%20%28OOP%29.ipynb#X26sZmlsZQ%3D%3D?line=0'>1</a> obj = BankApp()

TypeError: Can't instantiate abstract class BankApp with abstract methods display,
security
```

The error message indicates that you are trying to instantiate an abstract class called BankApp, but this class contains abstract methods display and security that have not been implemented in the BankApp class or its subclasses.

In Python, an abstract class is a class that cannot be instantiated directly, and it often serves as a blueprint for other classes. Abstract classes can contain abstract methods, which are methods declared without an implementation in the abstract class. Subclasses of the abstract class are required to provide implementations for these abstract methods.

NumPy Fundamentals (Part-1)

NumPy, which stands for "Numerical Python," is a fundamental Python library for scientific and numerical computing. It provides support for arrays (including multi-dimensional arrays) and a wide range of mathematical functions to operate on these arrays. NumPy is a crucial library in the Python scientific computing ecosystem and is often used in conjunction with other libraries like SciPy, Matplotlib, and pandas for various data analysis and scientific computing tasks.

NumPy is an essential tool for data scientists, engineers, and researchers working on numerical and scientific computing tasks in Python due to its efficiency, flexibility, and extensive mathematical capabilities.

Here are some key features and aspects of NumPy:

1. Arrays: NumPy's primary data structure is the ndarray (n-dimensional array). These arrays are similar to Python lists but are more efficient for numerical operations because they allow you to perform element-wise operations and take advantage of low-level optimizations.

2. Efficiency: NumPy is implemented in C and Fortran, making it highly efficient for numerical computations. It also provides tools to interface with libraries written in these languages.

3. Mathematical Functions: NumPy includes a wide range of mathematical functions for performing operations on arrays, such as addition, subtraction, multiplication, division, and more advanced operations like matrix multiplication, Fourier transforms, and linear algebra operations.

4. Broadcasting: NumPy allows you to perform operations on arrays of different shapes and sizes through a feature called broadcasting, which automatically aligns dimensions and performs operations element-wise.

5. Random Number Generation: NumPy provides functions for generating random numbers and random data, which is useful for tasks like simulations and statistical analysis.

6. Integration with Other Libraries: NumPy is often used in conjunction with libraries like SciPy for scientific computing, Matplotlib for data visualization, and pandas for data manipulation and analysis.

Numpy Arrays Vs Python Sequences

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.

- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

How To Create NumPy Arrays

1. **np.array**:
   - `np.array` is used to create NumPy arrays, which can be one-dimensional, two-dimensional, or multi-dimensional.
   - You need to import NumPy with `import numpy as np` before using it.
   - You can create arrays from Python lists or nested lists.

In [ ]:
```
# np.array

import numpy as np  # we have to first import the numpy

a = np.array([1,2,3])
print(a)
```

```
[1 2 3]
```

1. **2D and 3D Arrays**:
   - NumPy allows you to create multi-dimensional arrays. You demonstrated 2D and 3D arrays using nested lists.

In [ ]:
```
# 2D
n = np.array([[1,2,3],[4,5,6]])
print(n)
```

```
[[1 2 3]
 [4 5 6]]
```

In [ ]:
```
# 3d
c = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(c)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

1. **dtype**:
   - You can specify the data type of the elements in a NumPy array using the `dtype` parameter.

- In your example, you created arrays with `dtype=float` and `dtype=int` to explicitly set the data type.

In [ ]:
```
# dtype
np.array([1,2,3],dtype=float)
```

Out[ ]:
```
array([1., 2., 3.])
```

In [ ]:
```
np.array([1,2,3],dtype=int)
```

Out[ ]:
```
array([1, 2, 3])
```

1. **np.arange**:
   - `np.arange` is used to create a range of values within a specified interval.
   - It generates an array of evenly spaced values, similar to Python's `range` function.

In [ ]:
```
# np.arange
np.arange(1,11,2)
```

Out[ ]:
```
array([1, 3, 5, 7, 9])
```

1. **Reshaping Arrays**:
   - You can reshape an array using the `reshape` method. This changes the dimensions of the array while maintaining the total number of elements.
   - In your example, you used `reshape` to create a 4D array from a 1D array.

In [ ]:
```
# with reshape
np.arange(16).reshape(2,2,2,2)
```

Out[ ]:
```
array([[[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]],


       [[[ 8,  9],
         [10, 11]],

        [[12, 13],
         [14, 15]]]])
```

1. **np.ones**:
   - `np.ones` creates an array filled with ones.
   - You specify the shape of the array as a tuple. For example, `np.ones((3,4))` creates a 3x4 array filled with ones.

In [ ]:
```
# np.ones
np.ones((3,4))
```

Out[ ]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

1. **np.zeros**:
   - `np.zeros` creates an array filled with zeros.
   - Similar to `np.ones`, you specify the shape of the array as a tuple.

In [ ]:
```
#np.zeros
np.zeros((3,4))
```

Out[ ]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

1. **np.random**:
   - `np.random.random` generates random numbers between 0 and 1 in a specified shape.
   - It's useful for generating random data for simulations or experiments.

In [ ]:
```
# np.random
np.random.random((3,4))
```

Out[ ]:
```
array([[0.56431434, 0.09281216, 0.52283734, 0.34159087],
       [0.3840281 , 0.07649243, 0.09074241, 0.4599812 ],
       [0.09965102, 0.6279051 , 0.54948164, 0.29137218]])
```

1. **np.linspace**:
   - `np.linspace` generates evenly spaced values over a specified range.
   - You specify the start, end, and the number of values you want. In your example, you used `dtype=int` to ensure integer values.

In [ ]:
```
# np.linspace
np.linspace(-10,10,10,dtype=int)
```

Out[ ]:
```
array([-10,  -8,  -6,  -4,  -2,   1,   3,   5,   7,  10])
```

1. **np.identity**:
   - `np.identity` creates an identity matrix, which is a square matrix with ones on the diagonal and zeros elsewhere.
   - You specify the size of the identity matrix as a single integer.

In [ ]:
```
# np.identity
np.identity(3)
```

Out[ ]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

These NumPy functions are essential tools for working with numerical data in Python. They provide the foundation for many scientific and data analysis tasks, making it easier to perform calculations and manipulate data efficiently.

NumPy Attributes and Operations

NumPy is a popular Python library for numerical and scientific computing. It provides a wide range of functionality for working with arrays and matrices. NumPy arrays are at the core of the library, and they come with a variety of attributes and methods for performing operations

Array Attributes

In [ ]:
```
import numpy as np
```

In [ ]:
```
# lets assume three array
a1 = np.arange(10,dtype=np.int32)

a2 = np.arange(12,dtype=float).reshape(3,4)

a3 = np.arange(8).reshape(2,2,2)

print(a1)
print('----------')
print(a2)
print('----------')
print(a3)
```

```
[0 1 2 3 4 5 6 7 8 9]
----------
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
----------
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

ndim

This attribute returns the number of dimensions of a NumPy array. For example:

In [ ]:
```
a1.ndim
```

Out[ ]:
```
1
```

In [ ]:
```
a2.ndim
```

Out[ ]:
```
2
```

In [ ]:
```
a3.ndim
```

Out[ ]:
```
3
```

- a1.ndim returns 1 because a1 is a one-dimensional array.

- a2.ndim returns 2 because a2 is a two-dimensional array.
- a3.ndim returns 3 because a3 is a three-dimensional array.

shape

The shape attribute returns a tuple representing the dimensions of the array. For example:

In [ ]:
```
print(a1.shape)
print(a2.shape)
print(a3.shape)
```

```
(10,)
(3, 4)
(2, 2, 2)
```

- a1.shape returns (10,) because a1 is a one-dimensional array with a length of 10.
- a2.shape returns (3, 4) because a2 is a two-dimensional array with 3 rows and 4 columns.
- a3.shape returns (2, 2, 2) because a3 is a three-dimensional array with dimensions 2x2x2.

size

The size attribute returns the total number of elements in the array. For example:

In [ ]:
```
print(a1.size)
print(a2.size)
print(a3.size)
a3
```

```
10
12
8
```

Out[ ]:
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

- a1.size returns 10 because a1 has 10 elements.
- a2.size returns 12 because a2 has 12 elements.
- a3.size returns 8 because a3 has 8 elements.

itemsize

The itemsize attribute returns the size (in bytes) of each element in the array. For example:

In [ ]:
```
a2.itemsize
```

Out[ ]:
```
8
```

In [ ]:
```
a3.itemsize
```

Out[ ]:
```
4
```

- a2.itemsize returns 8 because a2 is of type float64, which has 8 bytes per element.
- a3.itemsize returns 4 because a3 is of type int32, which has 4 bytes per element.

dtype

The dtype attribute returns the data type of the elements in the array. For example:

In [ ]:
```
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int32
float64
int32
```

- a1.dtype returns int32 because a1 is of type int32.
- a2.dtype returns float64 because a2 is of type float64.
- a3.dtype returns int32 because a3 is of type int32.

Changing Datatype

You can change the data type of an array using the astype method.

In [ ]:
```
# astype
a3.astype(np.int32)
```

Out[ ]:
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

Array operations

In [ ]:
```
a1 = np.arange(12).reshape(3,4)
a2 = np.arange(12,24).reshape(3,4)
```

In [ ]:
```
print(a1)
print('--------')
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
--------
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Scalar operations

These operations involve performing mathematical operations on the entire array with a scalar value..

In [ ]:
```
# arithmatic - all arithmatic opearation works like
print(a1 + 2)
print('-----')
print(a1 **2)
print('-----')
print(a1 - 5)
```

```
[[ 2  3  4  5]
 [ 6  7  8  9]
 [10 11 12 13]]
-----
[[  0   1   4   9]
 [ 16  25  36  49]
 [ 64  81 100 121]]
-----
[[-5 -4 -3 -2]
 [-1  0  1  2]
 [ 3  4  5  6]]
```

In [ ]:
```
# relational -> all relational opearators work like
a2 == 15
```

Out[ ]:
```
array([[False, False, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

In [ ]:
```
a2 > 5
```

Out[ ]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

vector operations

These operations involve performing mathematical operations between two arrays.

In [ ]:
```
# arithmetic
a1 ** a2
```

Out[ ]:
```
array([[          0,           1,       16384,    14348907],
       [          0, -1564725563,  1159987200,   442181591],
       [          0,  1914644777, -1304428544,  -122979837]])
```

NumPy provides powerful tools for working with arrays efficiently, making it a valuable library for scientific and numerical computations in Python.

NumPy - Array Functions

In [ ]:
```
import numpy as np
```

1. Creating NumPy Arrays:

- You can create NumPy arrays using `np.random.random()`, `np.arange()`, and reshaping methods like `reshape()`. In your example, you created three NumPy arrays, `a1`, `a2`, and `a3`.

In [ ]:
```
# first we have to assume three numpy arrays
a1 = np.random.random((3,3))
a1 = np.round(a1*100)

a2 = np.arange(12).reshape(3,4)

a3 = np.arange(12,24).reshape(4,3)

print(a1)
print('------------')
print(a2)
print('-------------')
print(a3)
```

```
[[74. 98. 34.]
 [89. 62. 11.]
 [29. 87. 44.]]
------------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
-------------
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]
```

2. Maximum, Minimum, Sum, and Product:

- You can find the maximum value in an array using `np.max()`.
- You can find the minimum value in an array using `np.min()`.
- You can calculate the sum of all elements in an array using `np.sum()`.
- You can calculate the product of all elements in an array using `np.prod()`.
- By specifying the `axis` parameter, you can find the maximum or minimum along columns (axis=0) or rows (axis=1).

In [ ]:
```
np.max(a1)
```

Out[ ]:
```
98.0
```

In [ ]:
```
np.min(a1)
```

Out[ ]:
```
11.0
```

In [ ]:
```
np.sum(a1)
```

Out[ ]:
```
528.0
```

In [ ]:
```
np.prod(a1)
```

Out[ ]:
```
1661426069717568.0
```

In [ ]:
```
# if we want seperate for columns and rows
# axis -> 0 = columns , 1 = row

np.max(a1,axis=0)
```

Out[ ]:
```
array([89., 98., 44.])
```

In [ ]:
```
np.max(a1,axis=1)
```

Out[ ]:
```
array([98., 89., 87.])
```

In [ ]:
```
np.min(a1,axis=0)
```

Out[ ]:
```
array([29., 62., 11.])
```

3. **Mean, Median, Standard Deviation, and Variance:**

- You can calculate the mean of an array using `np.mean()`.
- You can find the median value in an array using `np.median()`.
- You can calculate the standard deviation of an array using `np.std()`.
- You can compute the variance of an array using `np.var()`.
- Similar to the previous case, you can specify the `axis` parameter for calculations along columns or rows.

In [ ]:
```
np.mean(a1)
```

Out[ ]:
```
58.666666666666664
```

In [ ]:
```
np.median(a2)
```

Out[ ]:
```
5.5
```

In [ ]:
```
np.std(a1)
```

Out[ ]:
```
28.85211334450987
```

In [ ]:
```
np.var(a1)
```

Out[ ]:
```
832.4444444444446
```

In [ ]:
```
# for columns and rows
np.median(a1,axis=1)
```

Out[ ]:
```
array([74., 62., 44.])
```

In [ ]:
```
np.median(a1,axis=0)
```

Out[ ]:
```
array([74., 87., 34.])
```

4. Trigonometric Functions:

- You can apply trigonometric functions to an array, such as sine (`np.sin()`) and cosine (`np.cos()`), which perform element-wise calculations on the input array.

In [ ]:
```
np.sin(a1)
```

Out[ ]:
```
array([[-0.98514626, -0.57338187,  0.52908269],
       [ 0.86006941, -0.7391807 , -0.99999021],
       [-0.66363388, -0.82181784,  0.01770193]])
```

In [ ]:
```
np.cos(a1)
```

Out[ ]:
```
array([[ 0.17171734, -0.81928825, -0.84857027],
       [ 0.51017704,  0.67350716,  0.0044257 ],
       [-0.74805753,  0.56975033,  0.99984331]])
```

5. Dot Product:

- The `np.dot()` function computes the dot product of two arrays, which is particularly useful for matrix multiplication. In your example, you multiplied `a2` and `a3` using `np.dot()` to obtain the result.

In [ ]:
```
print(a2)

print('------------')

print(a3)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
------------
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]
```

In [ ]:
```
np.dot(a2,a3)
```

Out[ ]:
```
array([[114, 120, 126],
       [378, 400, 422],
       [642, 680, 718]])
```

6. Logarithms and Exponents:

- You can compute the natural logarithm of an array using `np.log()` .
- You can calculate the exponential of each element in an array using `np.exp()` .

In [ ]:
```
np.log(a1)
```

Out[ ]:
```
array([[4.30406509, 4.58496748, 3.52636052],
       [4.48863637, 4.12713439, 2.39789527],
       [3.36729583, 4.46590812, 3.78418963]])
```

In [ ]:
```
np.exp(a1)
```

Out[ ]:
```
array([[1.37338298e+32, 3.63797095e+42, 5.83461743e+14],
       [4.48961282e+38, 8.43835667e+26, 5.98741417e+04],
       [3.93133430e+12, 6.07603023e+37, 1.28516001e+19]])
```

7. **Rounding, Flooring, and Ceiling:**

- You can round the elements of an array to the nearest integer using `np.round()`.
- You can round down the elements to the nearest integer using `np.floor()`.
- You can round up the elements to the nearest integer using `np.ceil()`.

In [ ]:
```
a=np.random.random((2,3))*100
print(a)
```

```
[[12.34714055 34.11903111 28.73639357]
 [97.69723411 52.68276927 61.09195526]]
```

In [ ]:
```
np.round(a)
```

Out[ ]:
```
array([[12., 34., 29.],
       [98., 53., 61.]])
```

In [ ]:
```
np.floor(a)
```

Out[ ]:
```
array([[12., 34., 28.],
       [97., 52., 61.]])
```

In [ ]:
```
np.ceil(a)
```

Out[ ]:
```
array([[13., 35., 29.],
       [98., 53., 62.]])
```

These NumPy functions and operations are essential tools for numerical and scientific computing in Python, providing a wide range of capabilities for data manipulation, analysis, and mathematical calculations.

Check Out this for more NumPy Functions - https://numpy.org/doc/stable/reference/routines.math.html

NumPy Fundamentals(part-2)

In [ ]:
```
import numpy as np
```

Part 1 - Creating Arrays:

- `a1` , `a2` , and `a3` are NumPy arrays of different dimensions created using `np.arange` and `reshape` .
- `a1` is a 1D array with 10 elements.
- `a2` is a 2D array with a shape of (3, 4).
- `a3` is a 3D array with a shape of (2, 2, 2).

In [ ]:
```
a1 = np.arange(10)
a2 = np.arange(12).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)

print(a1)
print('-------------')
print(a2)
print('-------------')
print(a3)
```

```
[0 1 2 3 4 5 6 7 8 9]
-------------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
-------------
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

Part 2 - Indexing and Slicing:

- Demonstrates how to access elements in NumPy arrays.
- For a 1D array (`a1`), elements are accessed by their index.
- For a 2D array (`a2`), elements are accessed using row and column indices.
- Slicing is shown for both 1D and 2D arrays, allowing you to extract specific ranges of elements.

In [ ]:
```
# For 1D array
a1
```

Out[ ]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [ ]:
```
# how to extract 5 from a1?
print(a1[5])

# how to extract 8 from a1?
print(a1[6])
```

```
5
6
```

In [ ]:
```
# how to extract 3 to 8 from a1?
print(a1[3:9])
```

```
[3 4 5 6 7 8]
```

In [ ]:
```
# for 2D array
a2
```

Out[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [ ]:
```
# how to extract 4 from a2

print(a2[1,0])

# How to extract 11 from a2

print(a2[2,3])
```

```
4
11
```

In [ ]:
```
# how to extract 1-2, 9-11 from a2
a2[0:2,1::2]
```

Out[ ]:
```
array([[1, 3],
       [5, 7]])
```

In [ ]:
```
a2[::2,1::2]
```

Out[ ]:
```
array([[ 1,  3],
       [ 9, 11]])
```

In [ ]:
```
a2[1,::3]
```

Out[ ]:
```
array([4, 7])
```

In [ ]:
```
# For 3D Array
a3 = np.arange(27).reshape(3,3,3)
a3
```

Out[ ]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [ ]:
```
# how to extract 18
a3[2,0,0]
```

Out[ ]:
```
18
```

In [ ]:
```
# how to extract 19 and 20
a3[2,0,1:]
```

Out[ ]:
```
array([19, 20])
```

In [ ]:
```
a3[::2,0,::2]
```

Out[ ]:
```
array([[ 0,  2],
       [18, 20]])
```

In [ ]:
```
a3[2,1:,1:]
```

Out[ ]:
```
array([[22, 23],
       [25, 26]])
```

In [ ]:
```
a3[0,1,:]
```

Out[ ]:
```
array([3, 4, 5])
```

Part 3 - Iterating:

- Iterating through NumPy arrays using `for` loops.
- Shows how to loop through elements in 1D and 2D arrays.
- The `np.nditer` function is introduced for iterating through all elements in the array.

In [ ]:
```
a1
```

Out[ ]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [ ]:
```
for i in a1:
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [ ]:
```
a2
```

Out[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [ ]:
```
for i in a2:
    print(i)
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

In [ ]:
```
for i in np.nditer(a2):
  print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

In [ ]:
```
a3
```

Out[ ]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [ ]:
```
for i in a3:
  print(i)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
[[18 19 20]
 [21 22 23]
 [24 25 26]]
```

In [ ]:
```
# nditer
for i in np.nditer(a3):
  print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

Part 4 - Reshaping:

- Reshaping NumPy arrays using `np.reshape`.
- Reshaping is demonstrated on a 2D array (`a2`), changing its shape from (3, 4) to (4, 3).
- The `np.transpose` function is used to obtain the transpose of the array, effectively swapping rows and columns.

In [ ]:
```
a2
```

Out[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Reshape

In [ ]:
```
np.reshape(a2,(4,3))
```

Out[ ]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Transpose

In [ ]:
```
a2
```

Out[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [ ]:
```
np.transpose(a2)
```

Out[ ]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [ ]:
```
a2.T
```

Out[ ]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

Part 5 - Ravel and Stacking:

- The `ravel` method is used to flatten a multi-dimensional array into a 1D array.
- Stacking arrays horizontally (`hstack`) and vertically (`vstack`) is demonstrated.
- Horizontal stacking combines arrays side by side, while vertical stacking stacks arrays on top of each other.

raval

In [ ]:
```
a3.ravel()
```

Out[ ]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
```

Stacking

In [ ]:
```
# horizontal stacking
a4 = np.arange(12).reshape(3,4)
a5 = np.arange(12,24).reshape(3,4)
a5
```

Out[ ]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [ ]:
```
np.hstack((a4,a5))
```

Out[ ]:
```
array([[ 0,  1,  2,  3, 12, 13, 14, 15],
       [ 4,  5,  6,  7, 16, 17, 18, 19],
       [ 8,  9, 10, 11, 20, 21, 22, 23]])
```

In [ ]:
```
# Vertical stacking
np.vstack((a4,a5))
```

Out[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

NumPy Broadcasting: A Guide to Working with Arrays of Different Shapes

- In NumPy, broadcasting is a powerful mechanism that allows you to perform element-wise operations on arrays of different shapes, making it easier and more convenient to work with arrays of varying sizes without explicitly reshaping them. Broadcasting essentially extends or duplicates the smaller array to match the shape of the larger array, so that element-wise operations can be performed without raising shape or size compatibility errors.
- The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations.
- The smaller array is “broadcast” across the larger array so that they have compatible shapes.

In [ ]:
```
import numpy as np
```

In [ ]:
```
# same shape
a = np.arange(6).reshape(2,3)
b = np.arange(6,12).reshape(2,3)

print(a)
print('------------')
print(b)
print('------------')
print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
------------
[[ 6  7  8]
 [ 9 10 11]]
------------
[[ 6  8 10]
 [12 14 16]]
```

In [ ]:
```
# diff shape
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)

print(a)
print('------------')
print(b)

print('------------')
print(a+b)
```

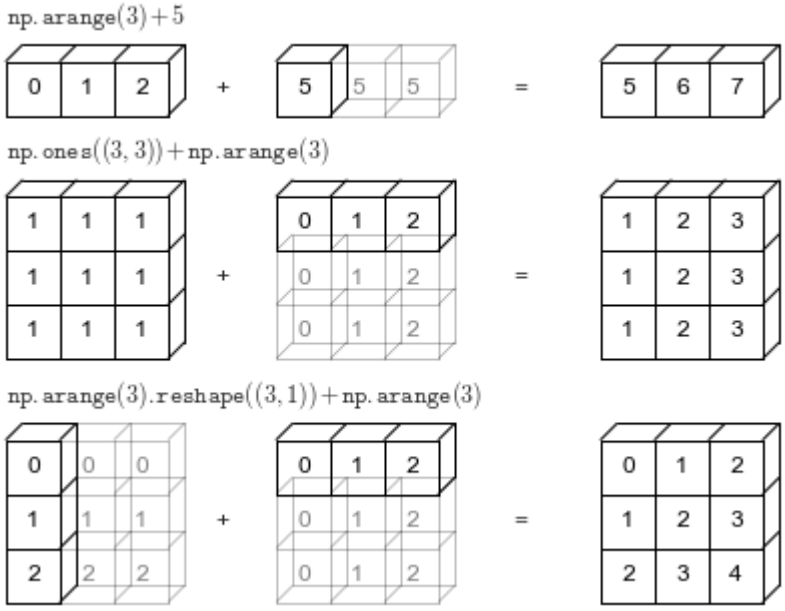
```
[[0 1 2]
 [3 4 5]]
------------
[[0 1 2]]
------------
[[0 2 4]
 [3 5 7]]
```

Broadcasting Rules

1. Make the two arrays have the same number of dimensions.: If two arrays have different numbers of dimensions, NumPy will pad the smaller shape with ones on the left side, making the shapes compatible for element-wise operations.

2. Make each dimension of the two arrays the same size.: If the shapes of the two arrays do not match in any dimension, NumPy will try to stretch the smaller dimension to match the larger one, provided that the smaller dimension's size is 1. If stretching is not possible, a "ValueError" will be raised.

3.Dimension: If the sizes of the dimensions are not 1 but still do not match, NumPy will raise a "ValueError."



Examples

Broadcasting Example 1: The shapes of a and b are (4,3) and (3,), respectively, and broadcasting is successful.

In [ ]:
```
a = np.arange(12).reshape(4,3)
b = np.arange(3)

print(a)
print('------------')
print(b)
print('------------')
print(a+b)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
------------
[0 1 2]
------------
[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

Example 2: Broadcasting does not work when the shapes of two arrays cannot be made compatible.

In [ ]:
```
a = np.arange(12).reshape(3,4)
b = np.arange(3)

print(a)
print('------------')
print(b)
print('------------')
print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
------------
[0 1 2]
------------
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
c:\Users\disha\Downloads\Python 100 Days\100_Days_OF_Python\Day25 - Broadcasting-NumPy.ipynb Cell 10 line 8
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X20sZmlsZQ%3D%3D?line=5'>6</a> print(b)
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X20sZmlsZQ%3D%3D?line=6'>7</a> print('------------')
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Days/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X20sZmlsZQ%3D%3D?line=7'>8</a> print(a+b)

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

The shapes of a and b are (3,4) and (3,), respectively, which are not compatible for broadcasting, resulting in a "ValueError."

Broadcasting Example 3: The shapes of a and b are (1,3) and (3,1), respectively, and broadcasting is successful.

In [ ]:
```
a = np.arange(3).reshape(1,3)
b = np.arange(3).reshape(3,1)

print(a)
print('------------')
print(b)
print('------------')
print(a+b)
```

```
[[0 1 2]]
------------
[[0]
 [1]
 [2]]
------------
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

In [ ]:
```
a = np.arange(3).reshape(1,3)
b = np.arange(4).reshape(4,1)

print(a)
print('------------')
print(b)
print('------------')
print(a + b)
```

```
[[0 1 2]]
------------
[[0]
 [1]
 [2]
 [3]]
------------
[[0 1 2]
 [1 2 3]
 [2 3 4]
 [3 4 5]]
```

Broadcasting Example 4: The shape of 'a' is (1,1), and the shape of 'b' is (2,2), and broadcasting is successful.

In [ ]:
```
a = np.array([1])
# shape -> (1,1)
b = np.arange(4).reshape(2,2)
# shape -> (2,2)

print(a)
print('------------')
print(b)
print('------------')
print(a+b)
```

```
[1]
------------
[[0 1]
 [2 3]]
------------
[[1 2]
 [3 4]]
```

Broadcasting Example 5: The shapes of 'a' and 'b' are (3,4) and (4,3), which are not compatible for broadcasting, resulting in a "ValueError."

In [ ]:
```
a = np.arange(12).reshape(3,4)
b = np.arange(12).reshape(4,3)

print(a)
print('------------')
print(b)
```

```
print('------------')
print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
------------
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
------------
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
c:\Users\disha\Downloads\Python 100 Days\100_Days_OF_Python\Day25 - Broadcasting-
NumPy.ipynb Cell 14 line 8
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Da
ys/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X24sZmlsZQ%3D%3D?line=
5'>6</a> print(b)
      <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Da
ys/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X24sZmlsZQ%3D%3D?line=
6'>7</a> print('------------')
----> <a href='vscode-notebook-cell:/c%3A/Users/disha/Downloads/Python%20100%20Da
ys/100_Days_OF_Python/Day25%20-%20Broadcasting-NumPy.ipynb#X24sZmlsZQ%3D%3D?line=
7'>8</a> print(a+b)

ValueError: operands could not be broadcast together with shapes (3,4) (4,3)
```