Basics of Python Programming Language

1. Python Basics

1.1 Syntax and semantics of Python

syntax refers to the structure and rules that define the language, while semantics refers to the meaning and behavior of the language.

In [1]:
```
# Basic syntax

name = "Shaik Basheer Ahmed"
age = 24

print(f"hello my name is {name} and i'm  {age} years old")
```

```
hello my name is Shaik Basheer Ahmed and i'm  24 years old
```

1.2. Indentation

indentation refers to the spacing or tabs used to indicate the level of nesting or indentation in a Python code block.

In [2]:
```
# using indentation
age = 24
if age  > 30:
    print("You are old")
else:
    print('you are young')
```

```
you are young
```

In [3]:
```
#without indentation
age = 24
if age > 30:
print("You are old")
else:
    print('you are young')
```

```
  Cell In[3], line 4
    print("You are old")
    ^
IndentationError: expected an indented block after 'if' statement on line 3
```

In [4]:
```
# types inference
name = "Shaik Basheer Ahmed"
age = 24
print(type(name))
print(type(age))
```

```
<class 'str'>
<class 'int'>
```

1.3. print() function

print() function is used to display text or values on the console. It accepts one or more arguments separated by commas and prints them to the standard output stream.

In [5]:
```
# print() function example

print('Hello World!')
```

```
Hello World!
```

Conclusion:

1. understanding the syntax and semantics of python is essential for writing clean and efficient code that is easy to read and understand.
2. Indentation is used to indicate the level of nesting or indentation in a Python code block.
3. Types inference is used to determine the type of a variable based on its value.

2. Variables and Data Types in Python

1.1 Variables

Variables are used to store values in a computer program. They are named entities that can hold different types of data, such as numbers, strings, lists, dictionaries, and more.

In [6]:
```
# creating a variable

# creating a variable using numbers
a = 10
b = 20
c = a + b
print(c)

# creating a variable using strings
name = "Shaik Basheer Ahmed"
print(name)

# creating a variable using lists
my_list = [1, 2, 3, 4, 5]
print(my_list)

# creating a variable using tuples
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)
```

```
# creating a variable using dictionaries
my_dict = {'name': 'Shaik Basheer Ahmed', 'age': 24}
print(my_dict)

# creating a variable using sets
my_set = {1, 2, 3, 4, 5}
print(my_set)
```

```
30
Shaik Basheer Ahmed
[1, 2, 3, 4, 5]
(1, 2, 3, 4, 5)
{'name': 'Shaik Basheer Ahmed', 'age': 24}
{1, 2, 3, 4, 5}
```

In [9]:
```
# declaring a variable
x = 10
print(x)

# redeclaring a variable
x = 20
print(x)
```

```
10
20
```

In [8]:
```
# deleting a variable

# del x
#print(x)
```

In [10]:
```
# getting of variable type

name = "Shaik Basheer Ahmed"
age = 24
print(type(name))
print(type(age))
```

```
<class 'str'>
<class 'int'>
```

In [11]:
```
# casting variables

x = 10
y = 2.5
z = "20"

print('x =', x)
print('y =', y)
print('z =', z)

print('x =', type(x))
print('y =', type(y))
print('z =', type(z))

x = int(x)
```

```
y = float(y)
z = str(z)

print('x =', x)
print('y =', y)
print('z =', z)

print('x =', type(x))
print('y =', type(y))
print('z =', type(z))
```

```
x = 10
y = 2.5
z = 20
x = <class 'int'>
y = <class 'float'>
z = <class 'str'>
x = 10
y = 2.5
z = 20
x = <class 'int'>
y = <class 'float'>
z = <class 'str'>
```

In [12]:
```
# case sensitive variables

age = "24"
Age = "30"
print('age:',age)
print('Age:',Age)
```

```
age: 24
Age: 30
```

In [13]:
```
# multiple variables

name, age = "Shaik Basheer Ahmed", 24
print(name, age)

print(f'hi {name} you are {age} years old now!')
```

```
Shaik Basheer Ahmed 24
hi Shaik Basheer Ahmed you are 24 years old now!
```

2.2 Concatenation and f-strings

Concatenation is the process of joining two or more strings into a single string. It is done using the + operator.

In [14]:
```
# concatenation of variables

name = "Shaik Basheer Ahmed"
age = 24
print('hi ' + name + ' you are ' + str(age) + ' years old now!')
```

```
hi Shaik Basheer Ahmed you are 24 years old now!
```

2.3 Local & Global Variables

Local variables are defined inside a function or a block of code and are only accessible within that block.

Global variables are defined outside of any function or block of code and are accessible throughout the program.

In [15]:
```
# local  variable

def my_function():
    x = 10
    print(x)

my_function()
print(x)
```

```
10
10
```

In [16]:
```
# global variable

x = 10

def my_function():
    print(x)

my_function()
print(x)
```

```
10
10
```

3. Data Types in Python

data types refers to the different categories of data that can be stored in a computer program. Python has several built-in data types, such as numbers, strings, lists, tuples, dictionaries, and sets.

Data types has two types:

1. Primitive data types
2. Non-primitive data types

- In primitive data types, the value of a variable is directly stored in the variable itself. Example: int, float, string, boolean, None, etc.
- In non-primitive data types, the value of a variable is stored in a memory location. Example: list, tuple, dictionary, set, etc.

In [17]:
```python
#sample code for Primitive data types

integer = 10 # integer
float = 2.5 # float
string = 'Shaik Basheer' # string
is_active = True # Boolean

# Operations
sum_integer = integer + 10
product_float = float * 2
concat_text = string + ' Ahmed'
toggle = not is_active

# print the results
print(sum_integer) # Output: 20
print(product_float) # Output: 5.0
print(concat_text) # Output: Shaik Basheer Ahmed
print(toggle) # Output: False
```

```
20
5.0
Shaik Basheer Ahmed
False
```

In [ ]:
```python
# sample code on Non-primitive data types

list = [1, 2, 3, 4, 5] # list
tuple = (1, 2, 3, 4, 5) # tuple
dictionary = {'name': 'Shaik Basheer Ahmed', 'age': 24} # dictionary
set = {1, 2, 3, 4, 5} # set

# Operations
append_list = list.append(6)
remove_list = list.remove(3)
concat_tuple = tuple + (6,)
access_dictionary = dictionary['name']
add_set = set.add(6)

# print the results
print(append_list) # Output: None
print(remove_list) # Output: None
print(concat_tuple) # Output: (1, 2, 3, 4, 5, 6)
print(access_dictionary) # Output: Shaik Basheer Ahmed
print(add_set) # Output: None
```

```
None
None
(1, 2, 3, 4, 5, 6)
Shaik Basheer Ahmed
None
```

4. Operations in Python

Operations are the actions that can be performed on data types. Python supports a wide range of operations, including arithmetic operations, comparison

operations, logical operations, bitwise operations,

In [18]:
```
# Arithmetic Operators

a = 10
b = 5

print(a + b) # Output: 15
print(a - b) # Output: 5
print(a * b) # Output: 50
print(a / b) # Output: 2.0
print(a % b) # Output: 0
print(a // b) # Output: 2
print(a ** b) # Output: 10000

# Assignment Operators

a = 10
b = 5

a += b
print(a) # Output: 15

a -= b
print(a) # Output: 10

a *= b
print(a) # Output: 50

a /= b
print(a) # Output: 2.0

a %= b
print(a) # Output: 0

a //= b
print(a) # Output: 2

a **= b
print(a) # Output: 10000

# Comparison Operators

a = 10
b = 5

print(a == b) # Output: False
print(a != b) # Output: True
print(a > b) # Output: True
print(a < b) # Output: False
print(a >= b) # Output: True
print(a <= b) # Output: False

# Logical Operators
```

```python
a = True
b = False

print(a and b) # Output: False
print(a or b) # Output: True
print(not a) # Output: False

# Membership Operators

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

print(5 in a) # Output: True
print(6 not in a) # Output: True

# Identity Operators

a = 10
b = 10

print(a is b) # Output: True
print(a is not b) # Output: False

# Bitwise Operators

a = 10
b = 5

print(a & b) # Output: 0
print(a | b) # Output: 15
print(a ^ b) # Output: 15
print(~a) # Output: -11
print(a << 2) # Output: 40
print(a >> 2) # Output: 2
```

```
15
5
50
2.0
0
2
100000
15
10
50
10.0
0.0
0.0
0.0
False
True
True
False
True
False
False
True
False
True
True
True
False
0
15
15
-11
40
2
```

5. Escape Character Sequences

Escape characters or sequences are illegal characters for Python and never get printed as part of the output. When backslash is used in Python programming, it allows the program to escape the next characters.

Types of escape sequence:

code | Description ' - Single quote \ - Backslash \n - Newline \r - Carriage return \t - Tab \b - Backspace \f - Form feed \v - Vertical tab

In [ ]:
```
# ' single quote
txt = 'It\'s alright.'
print(txt)
```

```
It's alright.
```

In [21]:
```
#  \ backslash
txt = 'this will insert one \\ (blackslash).'
print(txt)
```

this will insert one \ (blackslash).

In [ ]:
```
txt = 'Hello\nworld!' # new line
print(txt)
```

```
Hello
world!
```

In [23]:
```
txt = 'Hello\rWorld!' # carriage return
print(txt)
```

```
World!
```

In [24]:
```
txt = 'Hello\tWorld!' # horizontal tab
print(txt)
```

```
Hello	World!
```

In [26]:
```
txt = 'Hello\bWorld!' # backspace erases one character
print(txt)
```

```
HellWorld!
```

6. Type of conversions in Python

It is a process of converting one data type to another data type.

There are two types of type conversion:

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion: Python automatically converts one data type to another data type. For example, if you assign a string to a variable, Python will automatically convert it to a string.

Explicit Type Conversion: Python allows you to explicitly convert one data type to another data type. For example, you can use the int() function to convert a string to an integer.

In [3]:
```
# Implicit Conversion
# Converting int to float

int_num = 123
float_num = 12.3

# Converting float to int
result = int_num + float_num

# Print the result
print(result)
print('Data Type:', type(result))
```

```
135.3
Data Type: <class 'float'>
```

In [8]:
```python
# Explicit Conversion

num_str = 12
num_int = 23

print('Data Type of num_str before type casting:', type(num_str))

# explicit type casting
num_str = int(num_int)

print('Data Type of num_str after type casting:', type(num_str))

result = num_int + num_str

print(result)
print('Data Type:', type(result))
```

```
Data Type of num_str before type casting: <class 'int'>
Data Type of num_str after type casting: <class 'int'>
46
Data Type: <class 'int'>
```

7. Flow Control, Loops & Control Statements in Python

Flow control refers to the logic of a program that controls the order in which statements are executed.

There are two types of flow control:

1. Conditional Statements
2. Loops

Conditional Statements: Conditional statements are used to control the flow of a program. They allow you to execute different blocks of code based on a certain condition.

Types of conditional statements:

1. if
2. elif
3. else

- If use for executing a block of code if a certain condition is true.
- elif use for executing a block of code if the previous conditions are false and the current condition is true.
- else use for executing a block of code if all previous conditions are false.

Loops: Loops are used to repeat a block of code multiple times. They allow you to execute the same block of code multiple times.

Types of loops:

1. while
2. for

- While loop use for executing a block of code as long as a certain condition is true.
- For loop use for executing a block of code for each item in a sequence.

break & continue statement: break statement use for breaking out of a loop. continue statement use for skipping the current iteration of a loop and moving on to the next iteration.

pass statement: pass statement use for doing nothing.

In [15]:
```
# if condition:

age = 24

if age > 18:
    print('You are a adult:')
else:
    print('You are a minor')
```

```
You are a adult:
```

In [ ]:
```
# elif condition:

age = int(input('Enter your age:'))

if age > 18:
    print('You are a adult:')
elif age == 18:
    print('You are a minor')
else:
    print('You are a child')
```

```
You are a child
```

In [26]:
```
# loop
for i in range(1,11):
    print(i)
```

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

In [35]:
```python
# While loop

count = 0

while count < 10: # here what we did is we are checking if the count is less
    print(count) # here we are printing the count
    count += 1  # here we are increasing the count by 1
```

```
0
1
2
3
4
5
6
7
8
9
```

In [37]:
```python
# break, continue & pass statement

for i in range(1, 11):
    if  i ==7: # means if i is equal to 7
        print(f'Reach {i}, exciting from the loop')
        break # exit the loop when i is 7
    if i % 2 == 0:
        print(f'{i} is even, skipping')
        continue # skip even numbers
    print(f'{i} is odd')
    pass # placeholder, no action
```

```
1 is odd
2 is even, skipping
3 is odd
4 is even, skipping
5 is odd
6 is even, skipping
Reach 7, exciting from the loop
```

8. Slicing in Python

Slicing is a technique in Python that allows you to extract a subset of a sequence, such as a string, list, tuple, or range.

syntax: sequence[start:end:step]

- start: The index of the first element to include in the slice.
- end: The index of the element after the last element to include in the slice.
- step: The step size to use when iterating over the sequence.

There are two types of slicing:

1. Positive Slicing
2. Negative Slicing

Positive Slicing: Positive slicing means that you start from the beginning of the sequence and go to the end.

Negative Slicing: Negative slicing means that you start from the end of the sequence and go to the beginning.

In [38]:
```
# Positive slicing

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

print(my_list[0:5]) # Output: [1, 2, 3, 4, 5]
print(my_list[:5]) # Output: [1, 2, 3, 4, 5]
```

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

In [39]:
```
# Negative slicing

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

print(my_list[-5:]) # Output: [6, 7, 8, 9, 10]
print(my_list[:-5]) # Output: [1, 2, 3, 4, 5]
```

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

additionally, you can also use the step parameter to specify the step size.

In [40]:
```
# Extended Slicing

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

print(my_list[::2]) # Output: [1, 3, 5, 7, 9]
print(my_list[1::2]) # Output: [2, 4, 6, 8, 10]
print(my_list[::-1]) # Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

In [41]:
```
# omitting indices

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

```
print(my_list[:])
print(my_list[::])
print(my_list[:10])
print(my_list[::2])
print(my_list[::-1])
```

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

In [42]:
```
# Reverse slicing

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

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

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

doing with list, string, tuple

In [43]:
```
# List
lst = [0, 1, 2, 3, 4, 5]
print(lst[1:4])      # [1, 2, 3]
print(lst[:3])       # [0, 1, 2]
print(lst[2:])       # [2, 3, 4, 5]
print(lst[::2])      # [0, 2, 4]
print(lst[-3:])      # [3, 4, 5]
print(lst[::-1])     # [5, 4, 3, 2, 1, 0]

# String
s = "Hello, World!"
print(s[0:5])        # Hello
print(s[7:])         # World!
print(s[::-1])       # !dlroW ,olleH

# Tuple
t = (10, 20, 30, 40, 50)
print(t[1:3])        # (20, 30)
print(t[::2])        # (10, 30, 50)
```

```
[1, 2, 3]
[0, 1, 2]
[2, 3, 4, 5]
[0, 2, 4]
[3, 4, 5]
[5, 4, 3, 2, 1, 0]
Hello
World!
!dlroW ,olleH
(20, 30)
(10, 30, 50)
```

9. Data Structures in Python

Data structures are used to store and organize data in a way that makes it easy to access and manipulate.

1. List
2. Tuple
3. Dictionary
4. Set

List: A list is a data structure that stores a collection of items in a linear order. they are mutable, which means you can change the elements of a list after it has been created. the symbol is [] square brackets .

List methods:

- append(): Adds an element to the end of the list.
- insert(): Inserts an element at a specified index in the list.
- remove(): Removes an element from the list.
- pop(): Removes and returns the element at a specified index in the list.
- clear(): Removes all elements from the list.
- index(): Returns the index of the first occurrence of an element in the list.
- count(): Returns the number of times an element appears in the list.
- sort(): Sorts the elements of the list in place.
- reverse(): Reverses the order of the elements in the list.

Tuple: A tuple is a data structure that stores a collection of items in a linear order. they are immutable, which means you cannot change the elements of a tuple after it has been created. the symbol is () parentheses.

Tuple methods:

- count(): Returns the number of times an element appears in the tuple.
- index(): Returns the index of the first occurrence of an element in the tuple.

Dictionary: A dictionary is a data structure that stores a collection of key-value pairs. they are mutable, which means you can change the elements of a dictionary after it has been created. the symbol is {} curly brackets.

Dictionary methods:

- get(key[, default]): Return value for key, else default.
- pop(key[, default]): Remove and return value for key.
- popitem(): Remove and return last key-value pair.
- clear(): Remove all items.
- update(dict/iterable): Update with key-value pairs.
- keys(): Return view of keys.
- values(): Return view of values.

- items(): Return view of key-value pairs.
- setdefault(key[, default]): Return value for key, set default if absent.
- copy(): Return shallow copy.

Set: A set is a data structure that stores a collection of unique items. they are mutable, which means you can change the elements of a set after it has been created. the symbol is {} curly brackets.

Set methods:

- add(element): Add an element to the set.
- remove(element): Remove an element from the set.
- discard(element): Remove an element from the set if present.
- pop(): Remove and return an arbitrary element from the set.
- clear(): Remove all elements from the set.
- union(other_set): Return a new set with all elements from both sets.
- intersection(other_set): Return a new set with elements common to both sets.
- difference(other_set): Return a new set with elements in the first set but not in the second set.
- symmetric_difference(other_set): Return a new set with elements in either set but not both.

List

In [44]:
```
# Initialize an list
list = [3, 1, 2, 4, 5, 6, 7, 8, 9, 10]

#append() method, adds an element to the end of the list
list.append(11)
print(list) # Output: [3, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]

#insert() method, inserts an element at a specific index
list.insert(0, 0)
print(list) # Output: [0, 3, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]

#remove() method, removes an element from the list
list.remove(3)
print(list) # Output: [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11]

#pop() method, removes and returns the element at a specific index
list.pop(0)
print(list) # Output: [1, 2, 4, 5, 6, 7, 8, 9, 10, 11]

#clear() method, removes all elements from the list
list.clear()
print(list) # Output: []

#count() method, returns the number of occurrences of an element in the list
```

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list.count(3)) # Output: 1

#sort() method, sorts the elements of the list in ascending order
list = [3, 1, 2, 4, 5, 6, 7, 8, 9, 10]
list.sort()
print(list) # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#reverse() method, reverses the order of the elements in the list
list = [3, 1, 2, 4, 5, 6, 7, 8, 9, 10]
list.reverse()
print(list) # Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

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

Tuple

In [ ]:
```
# Initialize a Tuple
tuple = (3, 1, 2, 4, 3, 5, 6, 7, 8,3, 9, 10)

#count() method, returns the number of occurrences of an element in the tupl
print(tuple.count(3)) # Output: 3

#index() method, returns the index of the first occurrence of an element in
print(tuple.index(3)) # Output: 0
```

```
3
0
```

Dictionary

In [50]:
```
# Initialize a Dictionary
dict = {'name': 'Shaik Basheer Ahmed', 'age': 24}

# get() method, returns the value of a key in the dictionary
print(dict.get('name')) # Output: Shaik Basheer Ahmed

# pop() method, removes and returns the value of a key in the dictionary
print(dict.pop('age')) # Output: 24

# popitem() method, removes and returns the last inserted key-value pair in
print(dict.popitem()) # Output: ('age', 24)

# clear() method, removes all key-value pairs from the dictionary
dict.clear()
print(dict) # Output: {}

# Update a Dictionary
```

```
dict = {'name': 'Shaik Basheer Ahmed', 'age': 24}
dict.update({'country': 'India'})
print(dict) # Output: {'name': 'Shaik Basheer Ahmed', 'age': 24, 'country':

# Keys() method, returns a list of all the keys in the dictionary
print(dict.keys()) # Output: dict_keys(['name', 'age', 'country'])

# Values() method, returns a list of all the values in the dictionary
print(dict.values()) # Output: dict_values(['Shaik Basheer Ahmed', 24, 'Indi

# Items() method, returns a list of all the key-value pairs in the dictionar
print(dict.items()) # Output: dict_items([('name', 'Shaik Basheer Ahmed'), (

# Copy() method, returns a copy of the dictionary
dict = {'name': 'Shaik Basheer Ahmed', 'age': 24}
dict_copy = dict.copy()
print(dict_copy) # Output: {'name': 'Shaik Basheer Ahmed', 'age': 24}
```

```
Shaik Basheer Ahmed
24
('name', 'Shaik Basheer Ahmed')
{}
{'name': 'Shaik Basheer Ahmed', 'age': 24, 'country': 'India'}
dict_keys(['name', 'age', 'country'])
dict_values(['Shaik Basheer Ahmed', 24, 'India'])
dict_items([('name', 'Shaik Basheer Ahmed'), ('age', 24), ('country', 'Indi
a')])
{'name': 'Shaik Basheer Ahmed', 'age': 24}
```

Set

In [51]:

```
# Initialize a Set
set = {3, 1, 2, 4, 5, 6, 7, 8, 9, 10}

# add() method, adds an element to the set
set.add(11)
print(set) # Output: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

# remove() method, removes an element from the set
set.remove(3)
print(set) # Output: {1, 2, 4, 5, 6, 7, 8, 9, 10, 11}

# discard() method, removes an element from the set if it is present
set.discard(2)
print(set) # Output: {1, 3, 4, 5, 6, 7, 8, 9, 10, 11}

# pop() method, removes and returns an arbitrary element from the set
set.pop()
print(set) # Output: {2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

# clear() method, removes all elements from the set
set.clear()

# recreate the set
set = {3, 1, 2, 4, 5, 6, 7, 8, 9, 10}
```

```python
# union() method, returns a new set with all elements from both sets
set1 = {1, 2, 3}
set2 = {4, 5, 6}
set3 = set1.union(set2)
print(set3) # Output: {1, 2, 3, 4, 5, 6}

# intersection() method, returns a new set with elements common to both sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.intersection(set2)
print(set3) # Output: {3}

# difference() method, returns a new set with elements in the first set but
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.difference(set2)
print(set3) # Output: {1, 2}

# symmetric_difference() method, returns a new set with elements in either t
set1 = {1, 2, 3}
set2 = {3, 4, 5}
set3 = set1.symmetric_difference(set2)
print(set3) # Output: {1, 2, 4, 5}

# issubset() method, returns True if the first set is a subset of the second
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}
print(set1.issubset(set2)) # Output: True
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
{1, 2, 4, 5, 6, 7, 8, 9, 10, 11}
{1, 4, 5, 6, 7, 8, 9, 10, 11}
{4, 5, 6, 7, 8, 9, 10, 11}
{1, 2, 3, 4, 5, 6}
{3}
{1, 2}
{1, 2, 4, 5}
True
```

10. Functions in Python

Functions are blocks of code that perform a specific task. They can be used to break down complex tasks into smaller, more manageable parts.

Types of Functions:

- Built-in Functions:
- User-defined Functions:
- Lambda Functions:
- Anonymous Functions:
- Recursive Functions:

1. Built-in Functions: print(), len(), max(), min(), etc. It is a predefined function that is available in Python. example: print(), len(), max(), min(), etc.
2. User-defined Functions: def It is a function that is defined by the user. example: def add(a, b): return a + b
3. Lambda Functions: map(), filter(), reduce() It is a function that is defined using a lambda expression. example: lambda x: x * 2
4. Anonymous Functions: It is a function that is defined without a name. example: (lambda a, b: a + b) (2, 3)
5. Recursive Functions: It is a function that calls itself. example: def factorial(n): return 1 if n == 0 else n * factorial(n-1)

In [52]:
```
# Initialize a list
list = [3, 1, 2, 4, 5, 6, 7, 8, 9, 10]
```

In [53]:
```
# 1. Built-in functions
num = len(list)
print('Built-in functions:', num) # Output: 10
```

```
Built-in functions: 10
```

In [54]:
```
# 2. User-defined functions
def add(a, b):
    return a + b

result = add(3, 4)
print('User-defined functions:', result) # Output: 7
```

```
User-defined functions: 7
```

In [55]:
```
# 3. Lambda functions
result =  lambda x: x * 2
print('Lambda functions:', result(5)) # Output: 10
```

```
Lambda functions: 10
```

In [56]:
```
# 4. Ananonymous functions
result =  (lambda x, y: x + y)(3, 4)
print('Ananonymous functions:', result) # Output: 7
```

```
Ananonymous functions: 7
```

In [57]:
```
# 5. Recursive functions
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

result = factorial(5)
print('Recursive functions:', result) # Output: 120
```

```
Recursive functions: 120
```

11. Modules in Python

Modules are Python files that contain code that can be used in other Python files.

Types of Modules:

- Standard Modules:
- Custom Modules:
- External Modules:

1. Standard Modules: It is a module that is available in Python. example: import math
2. Custom Modules: It is a module that is created by the user. example: import my_module
3. External Modules: It is a module that is downloaded from the internet. example: import requests

In [58]:
```
# Standard Modules
import math
print('Standard Modules:', math.pi) # Output: 3.141592653589793
```

```
Standard Modules: 3.141592653589793
```

In [ ]:
```
# Custom Modules
```

In [61]:
```
# External Modules
!pip install numpy
import numpy as np
print('External Modules:', np.pi) # Output: 3.141592653589793
```

```
[notice] A new release of pip is available: 24.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
Requirement already satisfied: numpy in c:\users\ahmed\appdata\local\program
s\python\python313\lib\site-packages (2.2.3)
External Modules: 3.141592653589793
```

12. Object & Class in Python

Object: An object is a data structure that represents a real-world entity or a concept. example: a car, a person, a book, a computer, etc.

Class: A class is a blueprint for creating objects. example: Car, Person, Book, Computer, etc.

In [62]:
```
# Define a class
class Dog:
```

```
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age

    def bark(self):  # Instance method
        return f"{self.name} says Woof!"

# Create objects
dog1 = Dog("Buddy", 3)  # Object 1
dog2 = Dog("Max", 5)    # Object 2

# Access attributes and methods
print(dog1.name, dog1.age)  # Buddy 3
print(dog2.bark())          # Max says Woof!
```

```
Buddy 3
Max says Woof!
```

13. Inheritance in Python

Inheritance is a way of creating a new class from an existing class. It allows the new class (derived or child class) to inherit properties and behaviors (attributes and methods) of the existing class (base or parent class). This promotes code reusability and establishes an "is-a" relationship between the classes. Here are the common types of inheritance:

1. Single Inheritance: A derived class inherits from only one base class.
2. Multiple Inheritance: A derived class inherits from 1 more than one base class. This can lead to complexities like the "diamond problem" if the base classes have methods with the same name.
3. Multilevel Inheritance: A derived class inherits from a base class, and then another derived class inherits from that derived class, forming a chain of inheritance.
4. Hybrid Inheritance: This is a combination of two or more types of inheritance. For example, it could be a combination of single and multiple inheritance, or single and multilevel inheritance. The structure forms a lattice-like diagram.
5. Hierarchical Inheritance: This is a type of inheritance where multiple derived classes inherit from a single base class. This can be useful for creating a hierarchy of classes.

In [63]:
```
# 1. Single inheritance

# Base class
class Vehicle:
    def start_engine(self):
        print("Engine started.")

# Derived class inheriting from ONE base class (Vehicle)
class Car(Vehicle):
    def drive(self):
```

```
        print("Car is driving.")

# --- Demonstration ---
my_car = Car()
my_car.start_engine() # Inherited method
my_car.drive()      # Own method
```

```
Engine started.
Car is driving.
```

In [64]:
```
# 2. multiple inheritance

# First base class
class Flyer:
    def fly(self):
        print("Flying...")

# Second base class
class Swimmer:
    def swim(self):
        print("Swimming...")

# Derived class inheriting from MULTIPLE base classes (Flyer, Swimmer)
class Duck(Flyer, Swimmer):
    def quack(self):
        print("Quack!")

# --- Demonstration ---
donald = Duck()
donald.fly()    # Inherited from Flyer
donald.swim()   # Inherited from Swimmer
donald.quack()  # Own method
```

```
Flying...
Swimming...
Quack!
```

In [65]:
```
# 3. Multiple inheritance

# Grandparent class
class LivingThing:
    def breathe(self):
        print("Breathing...")

# Parent class inheriting from LivingThing
class Animal(LivingThing):
    def eat(self):
        print("Eating...")

# Child class inheriting from Animal (which inherited from LivingThing)
# A -> B -> C structure (LivingThing -> Animal -> Dog)
class Dog(Animal):
    def bark(self):
        print("Woof!")

# --- Demonstration ---
buddy = Dog()
```

```
buddy.breathe() # Inherited from LivingThing (Grandparent)
buddy.eat()     # Inherited from Animal (Parent)
buddy.bark()    # Own method
```

```
Breathing...
Eating...
Woof!
```

In [66]:
```
# 4. Hierarchical inheritance

# Base class
class Shape:
    def info(self):
        print("This is a shape.")

# First derived class inheriting from Shape
class Circle(Shape):
    def draw_circle(self):
        print("Drawing a circle.")

# Second derived class inheriting from the SAME base class (Shape)
class Square(Shape):
    def draw_square(self):
        print("Drawing a square.")

# --- Demonstration ---
c = Circle()
s = Square()

c.info() # Inherited from Shape
c.draw_circle()

s.info() # Inherited from Shape
s.draw_square()
```

```
This is a shape.
Drawing a circle.
This is a shape.
Drawing a square.
```

In [67]:
```
# 5. hybrid inheritance

# Base class
class A:
    def method_A(self): print("Method A")

# Derived from A (Single)
class B(A):
    def method_B(self): print("Method B")

# Another base class
class C:
    def method_C(self): print("Method C")

# Derived from B and C (Multiple, involves Multilevel A->B)
# This combination makes it Hybrid
class D(B, C):
```

```
    def method_D(self): print("Method D")

# --- Demonstration ---
obj_d = D()
obj_d.method_A() # Inherited via B (Multilevel)
obj_d.method_B() # Inherited from B
obj_d.method_C() # Inherited from C (Multiple)
obj_d.method_D() # Own method
```

```
Method A
Method B
Method C
Method D
```

14. *args and **kwargs in Python

*args and **kwargs are special syntax in Python that allows you to pass a variable number of arguments to a function. These arguments are collected into a tuple or dictionary, respectively.

1. *args: *args is a special syntax that allows you to pass a variable number of arguments to a function. It is used to collect positional arguments into a tuple.

- syntax: def function_name(*args):

2. **kwargs: **kwargs is a special syntax that allows you to pass a variable number of keyword arguments to a function. It is used to collect keyword arguments into a dictionary.

- syntax: def function_name(**kwargs):

In [68]:
```
# *args example
def sum_nums(*args):
    return sum(args)
print(sum_nums(1, 2, 3))  # 6
print(sum_nums(4, 5, 6, 7))  # 22

# **kwargs example
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_info(name="Alice", age=25)  # name: Alice, age: 25

# Combining *args and **kwargs
def combined(*args, **kwargs):
    print("args:", args)
    print("kwargs:", kwargs)
combined(1, 2, name="Bob", city="NY")  # args: (1, 2), kwargs: {'name': 'Bob
```

```
6
22
name: Alice
age: 25
args: (1, 2)
kwargs: {'name': 'Bob', 'city': 'NY'}
```