UNIT – 1

PYTHON

- NUMBERS
- STRINGS
- VARIABLES
- LISTS
- TUPLES
- DICTIONARIES
- SETS
- COMPARISION

NUMBERS :

Number data types store numeric values. They are immutable data types, which means that changing the value of a number data type results in a newly allocated object.

Different types of Number data types are :

- int
- float
- complex

INT :

int is the whole number, including negative numbers but not fractions. In Python, there is no limit to how long an integer value can be.

Float

This is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation. . Some examples of numbers that are represented as floats are 0.5 and -7.823457.

Complex type

A complex number is a number that consists of real and imaginary parts. For example, 2 + 3j is a complex number where 2 is the real component, and 3 multiplied by j is an imaginary part.

Type Conversion in Python

We can convert one number into the other form by two methods:

Using Arithmetic Operations:

We can use operations like addition, and subtraction to change the type of number implicitly(automatically), if one of the operands is float. This method is not working for complex numbers.

Using built-in functions

We can also use built-in functions like int(), float() and complex() to convert into different types explicitly.

```
a = 2

print(float(a))

b = 5.6

print(int(b))

c = '3'

print(type(int(c)))

d = '5.6'

print(type(float(c)))
```

Python provides several built-in numeric functions that you can use for mathematical operations. Here are some commonly used numeric functions in Python:

1. **abs()**: Returns the absolute value of a number.

   For example:

```
num = -10

abs_num = abs(num)

 print(abs_num)

# Output: 10
```

2. **round()**: Rounds a number to the nearest integer or to the specified number of decimal places.

   For example:

```
num = 3.14159

rounded_num = round(num, 2)

 print(rounded_num)

# Output: 3.14
```

3. **min()**: Returns the minimum value from a sequence of numbers or arguments.

   For example:

```
numbers = [5, 2, 9, 1, 7]
 min_num = min(numbers)
 print(min_num)
 # Output: 1
```

4. **max()**: Returns the maximum value from a sequence of numbers or arguments.

   For example:

```
numbers = [5, 2, 9, 1, 7]
 max_num = max(numbers)
print(max_num)
# Output: 9
```

5. **sum()**: Returns the sum of a sequence of numbers.

   For example:

```
numbers = [1, 2, 3, 4, 5]
 sum_num = sum(numbers)
 print(sum_num)
# Output: 15
```

Strings in Python:

In Python, a string is a sequence of characters enclosed in either single quotes ('') or double quotes (""). It is a fundamental data type used to represent and manipulate textual data.

Here are some common operations and examples related to strings in Python:

Creating a string :

```
my_string = "Hello, world!"
```

Accessing characters in a string:

```
my_string = "Hello"

print(my_string[0])    # Output: 'H'

print(my_string[1])    # Output: 'e'
```

String concatenation:

```
string1 = "Hello"

string2 = " world!"

result = string1 + string2

print(result)

 # Output: "Hello world!"
```

String length :

```
my_string = "Hello"

length = len(my_string)

print(length)  # Output: 5
```

String slicing :

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

String formatting (using the % operator):

```
name = "Alice"
age = 25
message = "My name is %s and I'm %d years old." % (name, age)
print(message)
# Output: "My name is Alice and I'm 25 years old."
```

String interpolation (using f-strings):

```
name = "Alice"
age = 25
message = f"My name is {name} and I'm {age} years old."
print(message)
# Output: "My name is Alice and I'm 25 years old."
```

String methods:

```
my_string = "Hello, world!"
print(my_string.upper())  # Output: "HELLO, WORLD!"
print(my_string.lower())  # Output: "hello, world!"
print(my_string.startswith("Hello"))  # Output: True
print(my_string.endswith("world!"))  # Output: True

print(my_string.split(", "))  # Output: ['Hello', 'world!']
```

These are just a few examples of string operations in Python.

Strings are immutable, meaning their contents cannot be changed once created.

However, you can create new strings by applying various string operations.

Variables

Variables are the reserved memory locations used to store values with in a Python Program. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, Python interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

Creating Python Variables

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
counter = 100          # Creates an integer variable
miles   = 1000.0       # Creates a floating point variable
name    = "Zara Ali"   # Creates a string variable
```

Delete a Variable

You can delete the reference to a number object by using the del statement. The syntax of the del statement is −

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the del statement. For example −

```
del var

del var_a, var_b
```

Example

Following examples shows how we can delete a variable and if we try to use a deleted variable then Python interpreter will throw an error:

```
counter = 100

print (counter)


del counter

print (counter)
```

This will produce the following result:

```
100
Traceback (most recent call last):
File "main.py", line 7, in <module>
print (counter)
NameError: name 'counter' is not defined
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously which means you can create multiple variables at a time. For example −

```
a = b = c = 100


print (a)

print (b)

print (c)
```

This produces the following result:

```
100
100
```

```
100
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a,b,c = 1,2,"Zara Ali"


print (a)
print (b)
print (c)
```

This produces the following result:

```
1
2
Zara Ali
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Zara Ali" is assigned to the variable c.

Python Variable Names

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number or any special character like $, (, * % etc.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
- Python reserved keywords cannot be used naming the variable.

Example

Following are valid Python variable names:

```
counter = 100
_count  = 100
name1 = "Zara"
name2 = "Nuha"
```

```
Age  = 20
zara_salary = 100000


print (counter)
print (_count)
print (name1)
print (name2)
print (Age)
print (zara_salary)
```

This will produce the following result:

```
100
100
Zara
Nuha
20
100000
```

Example

Following are invalid Python variable names:

```
1counter = 100
$_count  = 100
zara-salary = 100000


print (1counter)
print ($count)
print (zara-salary)
```

This will produce the following result:

```
File "main.py", line 3
1counter = 100
^
SyntaxError: invalid syntax
```

Python Local Variable

Python Local Variables are defined inside a function. We can not access variable outside the function.

A Python functions is a piece of reusable code and you will learn more about function in Python - Functions tutorial.

Following is an example to show the usage of local variables:

```
def sum(x,y):
sum = x + y
return sum
print(sum(5, 10))
```

```
15
```

Python Global Variable

Any variable created outside a function can be accessed within any function and so they have global scope. Following is an example of global variables:

```
x = 5
y = 10
def sum():
sum = x + y
return sum
print(sum())
```

This will produce the following result:

```
15
```

LISTS IN PYTHON

Lists in Python are one of the most commonly used data structures. They are used to store multiple items of any data type in a single variable. Lists are created using square brackets [ ] and items are separated by commas.

Lists are **ordered**, **changeable**, and **allow duplicate values**. You can access list items by using positive or negative indexes, or by using loops. You can also perform various operations on lists, such as adding, removing, sorting, slicing, copying, etc.

You can access the elements of a list by using their index positions, starting from 0 for the first element. You can also use negative indexes to access the elements from the end of the list, starting from -1 for the last element.

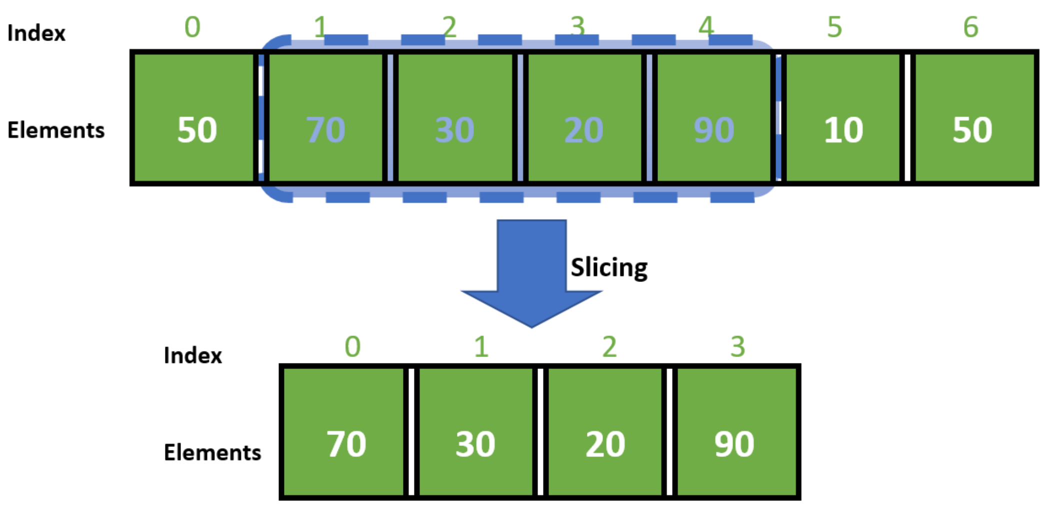
For example :

```
print(my_list[0]) # prints "apple"

print(my_list[-1]) # prints "melon"
```

You can also use slicing to get a range of elements from a list, by specifying the start and end indexes separated by a colon : .

For example:

```
print(my_list[1:4]) # prints ["banana", "cherry", "orange"]

print(my_list[:3]) # prints ["apple", "banana", "cherry"]

print(my_list[3:]) # prints ["orange", "kiwi", "melon"]
```



Lst[1:5]

You can modify the elements of a list by assigning new values to them using the index operator [ ]. You can also use methods like append(), insert(), remove(), pop(), sort(), reverse(), etc. to manipulate the list.

For example:

my_list[1] = "blueberry" # changes the second element to "blueberry"

my_list.append("strawberry") # adds "strawberry" to the end of the list

my_list.insert(2, "lemon") # inserts "lemon" at the third position

my_list.remove("orange") # removes "orange" from the list

my_list.pop() # removes and returns the last element of the list

my_list.sort() # sorts the list in ascending order

my_list.reverse() # reverses the order of the list

TUPLES:

A tuple is an ordered collection of elements, enclosed in parentheses **()**. Tuples are similar to lists, but they are immutable, meaning their elements cannot be changed once defined. Here's an example of creating a tuple in Python:

```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
```

In the above example, my_tuple is a tuple that contains integers and strings. Here are a few important things to note about tuples:

1. **Accessing Elements:** You can access individual elements of a tuple using indexing, similar to lists. The indexing starts from 0.

   For example:

```
print(my_tuple[0]) # Output: 1

print(my_tuple[3]) # Output: 'a'
```

2. **Tuple Slicing:** You can also use slicing to extract a subset of elements from a tuple. Slicing works similarly to lists.

   For example:

```
print(my_tuple[1:4]) # Output: (2, 3, 'a')
```

3. **Immutable:** Unlike lists, tuples are immutable, which means you cannot modify their elements. Once a tuple is created, you cannot add, remove, or modify its elements.
4. **Length and Count:** You can find the length of a tuple using the len() function and count the occurrences of a specific element using the count() method.

   For example:

```
print(len(my_tuple)) # Output: 6

print(my_tuple.count('a')) # Output: 1
```

5. **Tuple Concatenation:** You can concatenate two tuples using the + operator, which creates a new tuple.

For example:

```
new_tuple = my_tuple + ('x', 'y', 'z')

print(new_tuple) # Output: (1, 2, 3, 'a', 'b', 'c', 'x', 'y', 'z')
```

6. **Tuple Unpacking:** You can assign the elements of a tuple to multiple variables in a single line. The number of variables must match the number of elements in the tuple.

   For example:

```
a, b, c, d, e, f = my_tuple

print(c) # Output: 3
```

These are some of the basic operations and concepts related to tuples in Python. Tuples are often used to represent a collection of related values that should not be modified, such as coordinates, database records, or key-value pairs.

DICTIONARIES:

A dictionary is a collection of key-value pairs enclosed in curly braces **{}**. Dictionaries are also sometimes referred to as associative arrays or hash maps. Here's an example of creating a dictionary in Python:

```
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

In the above example, my_dict is a dictionary that stores information about a person, including their name, age, and city. Here are some important points about dictionaries:

1. **Accessing Values:** You can access the values in a dictionary by referring to its keys.

   For example:

```
print(my_dict['name']) # Output: 'John'

print(my_dict['age']) # Output: 30
```

2. **Modifying Values:** You can modify the values associated with specific keys in a dictionary. Dictionaries are mutable, so you can change, add, or remove key-value pairs.

For example:

```
my_dict['age'] = 35 # Modifying the 'age' value

my_dict['city'] = 'San Francisco' # Modifying the 'city' value
my_dict['occupation'] = 'Engineer' # Adding a new key-value pair

del my_dict['name'] # Removing the 'name' key-value pair
```

3. **Dictionary Methods:** Python provides various methods to work with dictionaries. Some commonly used methods include:

   - **keys()**: Returns a list of all the keys in the dictionary.
   - **values()**: Returns a list of all the values in the dictionary.
   - **items()**: Returns a list of tuples containing the key-value pairs.

```
print(my_dict.keys()) # Output: ['age', 'city', 'occupation']
print(my_dict.values()) # Output: [35, 'San Francisco', 'Engineer']
print(my_dict.items()) # Output: [('age', 35), ('city', 'San Francisco'),
('occupation', 'Engineer')]
```

4. **Dictionary Iteration:** You can iterate over the keys, values, or items of a dictionary using a for loop.

   For example:

```
for key in my_dict: print(key, my_dict[key])
```

5. **Length and Membership:** You can find the number of key-value pairs in a dictionary using the len() function. You can also check for the presence of a key using the in keyword.

   For example:

```
print(len(my_dict)) # Output: 3 (after modifications above)

print('name' in my_dict) # Output: False

print('age' in my_dict) # Output: True
```

Dictionaries are useful for storing and retrieving data based on unique keys. They provide a fast and efficient way to access values using their associated keys.

SETS IN PYTHON:

A set is an unordered collection of unique elements. It is defined by enclosing elements in curly braces ({}) or by using the built-in set() function. Sets are mutable, meaning you can add or remove elements from them.

EXAMPLES:

CREATING A SET:

```
# Creating an empty set

my_set = set()

# Creating a set with initial values

my_set = {1, 2, 3}
```

ADDING ELEMENT TO A SET:

```
my_set = {1, 2, 3}

my_set.add(4)

# my_set is now {1, 2, 3, 4}

# Adding multiple elements at once

my_set.update([5, 6, 7])

# my_set is now {1, 2, 3, 4, 5, 6, 7}
```

REMOVING ELEMENT FROM A SET:

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

my_set.remove(3)

# my_set is now {1, 2, 4, 5}


# Removing an element that does not exist will raise a KeyError
```

```
my_set.remove(6)


# Alternatively, you can use discard() to remove an element, but it won't raise an error if the element doesn't exist.
my_set.discard(6)


# Removing and returning an arbitrary element from the set
element = my_set.pop()
```

SET OPREATION:

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


for element in my_set:
    print(element)
```

These are some of the basic operations you can perform on sets in Python. Sets are useful when you want to work with unique elements or need to perform operations like union, intersection, and difference on collections of elements.

COMPARISON IN PYTHON:

In Python, comparison refers to the process of evaluating whether two values are equal, not equal, greater than, less than, greater than or equal to, or less than or equal to each other. Python provides several comparison operators that allow you to perform these comparisons. Here are the most commonly used comparison operators in Python

1. Equal to (==): This operator checks if two values are equal.
2. Not equal to (!=): This operator checks if two values are not equal.
3. Greater than (>): This operator checks if the value on the left is greater than the value on the right.

4. Less than (<): This operator checks if the value on the left is less than the value on the right.
5. Greater than or equal to (>=): This operator checks if the value on the left is greater than or equal to the value on the right.
6. Less than or equal to (<=): This operator checks if the value on the left is less than or equal to the value on the right.

Here's an example that demonstrates the usage of comparison operators in Python:

```
x = 5

y = 10


# Equal to

print(x == y)  # False


# Not equal to

print(x != y)  # True


# Greater than

print(x > y)  # False


# Less than

print(x < y)  # True


# Greater than or equal to

print(x >= y)  # False


# Less than or equal to
```

```
print(x <= y)  # True
```

In this example, we have two variables x and y with values 5 and 10, respectively. We then perform various comparisons using the comparison operators. The result of each comparison is printed, indicating whether the comparison is True or False.

UNIT-2

CODE STRUCTURES

1. If Statement:

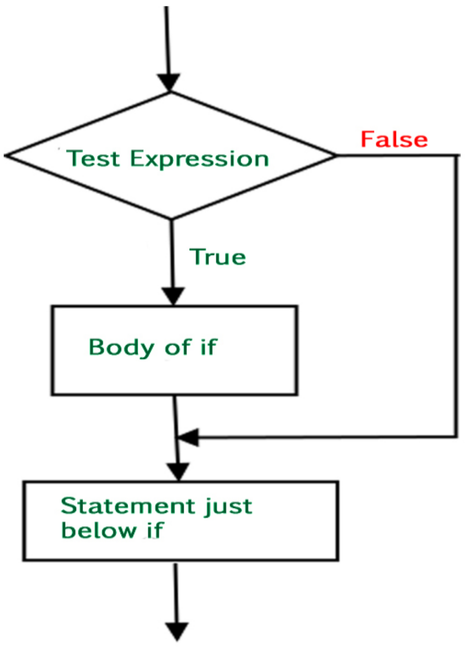
The if statement evaluates condition.

- If condition is evaluated to True, the code inside the body of if is executed.
- If condition is evaluated to False, the code inside the body of if is skipped.

SYNTAX:

```
if condition:

# Statements to execute if

# condition is true
```

Flow Chart:



Example:

```
number = 5

if number > 0:

    Print (Number is positive.)
```

OUTPUT:

```
Number is positive.
```

2. IF ELSE STATEMENT

- The if...else statement evaluates the given condition:

If the condition evaluates to True,

- the code inside if is executed
- the code inside else is skipped

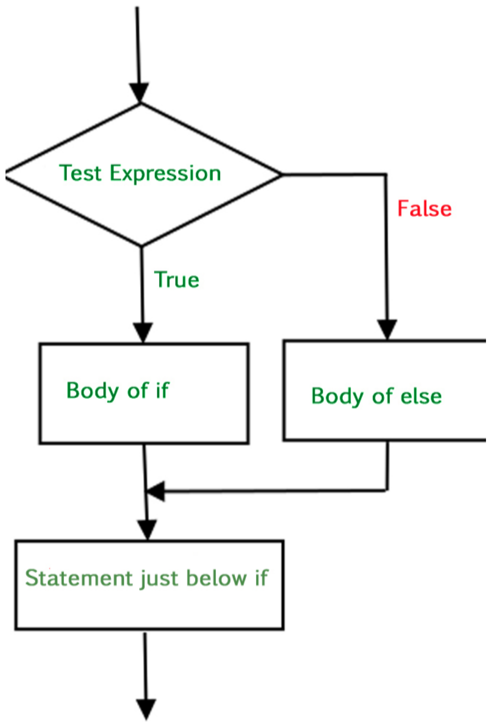
If the condition evaluates to False,

- the code inside else is executed
- the code inside if is skipped

SYNTAX:

```
if (condition):

    # Executes this block if

    # condition is true

else:

    # Executes this block if

    # condition is false
```

Flow Chart:



Example:

Age = 10

if (Age > 18):

print (Eligible for vote)

else:

print (Not Eligible for vote)

OUTPUT:

Not Eligible for vote

3. IF ELIF ELSE STATEMENT

- The if...else statement is used to execute a block of code among two alternatives.

- However, if we need to make a choice between more than two alternatives, then we use the if...elif...else statement.

Here,

- If condition1 evaluates to true, code block 1 is executed.
- If condition1 evaluates to false, then condition2 is evaluated.
    - If condition2 is true, code block 2 is executed.
    - If condition2 is false, code block 3 is executed

SYNTAX:

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

Example:

```
number = 0
if number > 0:
print (Positive number)
elif number = = 0:
print (Zero)
```

else:

print (Negative number)

Decorators in Python

In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.

The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

Prerequisites for learning decorators

Before we learn about decorators, we need to understand a few important concepts related to Python functions. Also, remember that everything in Python is an object, even functions are objects.

Nested Function

We can include one function inside another, known as a nested function. For example,

```
def outer(x):
    def inner(y):
        return x + y
    return inner

add_five = outer(5)
result = add_five(6)
```

```
print(result)  # prints 11

# Output: 11
```

Here, we have created the inner() function inside the outer() function.

Pass Function as Argument

We can pass a function as an argument to another function in Python. For Example,

```
def add(x, y):
    return x + y

def calculate(func, x, y):
    return func(x, y)

result = calculate(add, 4, 6)
print(result)  # prints 10
```
Run Code

Output

```
10
```

In the above example, the `calculate()` function takes a function as its argument. While calling `calculate()`, we are passing the `add()` function as the argument.
In the `calculate()` function, arguments: `func`, `x`, `y` become `add`, `4`, and `6` respectively.

And hence, `func(x, y)` becomes `add(4, 6)` which returns **10**.

Return a Function as a Value

In Python, we can also return a function as a return value. For example,

```
def greeting(name):
    def hello():
        return "Hello, " + name + "!"
    return hello

greet = greeting("Atlantis")
print(greet())  # prints "Hello, Atlantis!"

# Output: Hello, Atlantis!
```
Run Code

In the above example, the `return hello` statement returns the inner `hello()` function. This function is now assigned to the *greet* variable.
That's why, when we call `greet()` as a function, we get the output.

Python Decorators

As mentioned earlier, A Python decorator is a function that takes in a function and returns it by adding some functionality.

In fact, any object which implements the special `__call__()` method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")

# Output: I am ordinary
```
Run Code

Here, we have created two functions:

- `ordinary()` that prints `"I am ordinary"`
- `make_pretty()` that takes a function as its argument and has a nested function named `inner()`, and returns the inner function.

We are calling the `ordinary()` function normally, so we get the output `"I am ordinary"`. Now, let's call it using the decorator function.

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional behavior to decorated function
        print("I got decorated")

        # call original function
        func()
    # return the inner function
    return inner
```

```
# define ordinary function
def ordinary():
    print("I am ordinary")

# decorate the ordinary function
decorated_func = make_pretty(ordinary)

# call the decorated function
decorated_func()
```
Run Code

Output

```
I got decorated
I am ordinary
```

In the example shown above, `make_pretty()` is a decorator. Notice the code,

```
decorated_func = make_pretty(ordinary)
```

- We are now passing the `ordinary()` function as the argument to the `make_pretty()`.
- The `make_pretty()` function returns the inner function, and it is now assigned to the *decorated_func* variable.

```
decorated_func()
```

Here, we are actually calling the `inner()` function, where we are printing

PYTHON GENERATORS

What are Generators ?

- Generators are functions that return an iterator
- They generate values lazily, one at a time, rather than producing the entire sequence at once
- Each value is computed on-demand, saving memory and improving performance

Generator Functions :

- Generator functions are defined using the yield keyword instead of return
- They can pause and resume their execution, retaining their local state

Example :

```
def count_up_to(n):
    i = 0
    while i < n:
        yield i
        i += 1
# Usage
numbers = count_up_to(5)
print(next(numbers))  # Output: 0
print(next(numbers))  # Output: 1
print(next(numbers))  # Output: 2
```

Generator Expressions :

- Generator expressions are concise and memory-efficient
- They are similar to list comprehensions but enclosed in parentheses instead of brackets

Example :

```
squares = (x ** 2  for x  in range(5))

print(next(squares))  # Output: 0

print(next(squares))  # Output: 1

print(next(squares))  # Output: 4
```

Lazy Evaluation :

- Generators follow the principle of lazy evaluation
- Values are computed on-demand, reducing memory consumption
- Ideal for working with large or infinite sequences

Example :

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
fib = fibonacci()
for _ in range(10):
    print(next(fib))
```

Chaining Generators :

- Generators can be chained together to perform complex operations

Example :

```
def numbers():
    yield from range(5)
```

```
def squares(nums):
    yield from (x ** 2 for x in nums)
result = squares(numbers())
print(next(result))  # Output: 0
print(next(result))  # Output: 1
print(next(result))  # Output: 4
```

Benefits of Generators :

Memory efficiency :

Values are generated on-the-fly, reducing memory usage

Improved performance :

Laziness allows for faster code execution

Simplified code :

Generators enable cleaner and more readable code

Use Cases :

Large datasets :

Process large datasets in a memory-efficient manner

Infinite sequences :

Generate values on-the-fly without exhausting memory

Stream processing :

Handle real-time data streams without buffering

NAMESPACE:

In Python, a namespace is a system that organizes and manages names (identifiers) to avoid naming conflicts and provide a way to access variables, functions, classes, and other objects. It acts as a container or a context in which names are unique and can be used to refer to specific objects. Namespaces help in organizing and categorizing code elements and provide a way to differentiate between objects with the same name

Python has various types of namespaces:

- Built-in namespace
- Global namespace
- Local namespace
- Module namespace

Built-in Namespace:

It contains the names of all built-in functions, types, and exceptions provided by Python itself.

Example:

```
print(len("Hello"))  # 'len' is a built-in function from the built-in namespace
```

Global Namespace:

It contains names defined at the top level of a module or those explicitly declared as global within a function.

Example:

```
global_var = 10  # 'global_var' is defined in the global namespace
```

```
def some_function():
    global global_var
    global_var += 5  # Accessing and modifying 'global_var' from the global namespace


some_function()
print(global_var)  # Output: 15
```

Local Namespace:

It exists within a function or method and contains local variables and parameters.

Example:

```
def some_function():
    local_var = 20  # 'local_var' is defined in the local namespace
    print(local_var)


some_function()  # Output: 20
```

Module Namespace:

It contains the names defined within a module, including variables, functions, and classes.

Example:

```
# my_module.py
module_var = "Module Variable"  # 'module_var' is defined in the module namespace


def module_function():
    print("Module Function")


class MyClass:
    pass


# main.py
import my_module


print(my_module.module_var)     # Accessing 'module_var' from the module namespace
my_module.module_function()   # Calling 'module_function' from the module namespace
obj = my_module.MyClass()   # Creating an instance of 'MyClass' from the module namespace
```

SCOPE IN PYTHON

SCOPE:

In Python, scope refers to the region or context in which a variable, function, or other names are defined and can be accessed. The scope determines the visibility and lifetime of names and controls their accessibility throughout the program.

Python has several types of scopes:

- global scope
- local scope
- nested scope

Global Scope:

Variables defined outside of any function or class have global scope. They can be accessed from any part of the program.

Example:

```
global_var = 10  # Variable with global scope
```

```
def some_function():
    print(global_var)  # Accessing the global variable

some_function()  # Output: 10
```

Local Scope:

Variables defined within a function have local scope and are accessible only within that function.

Example:

```
def some_function():
    local_var = 20  # Variable with local scope
    print(local_var)

some_function()  # Output: 20

# Trying to access the local variable outside the function will raise an error
print(local_var)  # NameError: name 'local_var' is not defined
```

Nested Scope:

When a function is defined inside another function, it creates a nested scope. Variables defined in the outer function can be accessed within the inner function.

Example:

```
        def outer_function():
    outer_var = 30  # Variable in the outer function's scope

    def inner_function():
        print(outer_var)  # Accessing the variable from the outer
scope

    inner_function()

outer_function()  # Output: 30
```

Built-in Scope:

The built-in scope contains names that are built into Python itself, such as functions like **print()** and **len()**. These names are accessible from anywhere in the program.

Example:

```
print(len("Hello"))  # Using the built-in function 'len()' from the built-in scope
```

UNIT-3

1. **Standalone Programs**:

A standalone program in Python is a self-contained script or application that can be executed independently. To create a standalone program, you typically write your code in a Python script file (with a .py extension) and execute it using the Python interpreter.

Example of a simple standalone program:

Code:

```
# hello.py

print("Hello, World!")
```

To run the program, you execute it from the command line:

OUTPUT:

```
hello.py
```

2. **Command-Line Arguments:**

You can pass command-line arguments to Python scripts using the sys.argv list from the sys module. Additionally, libraries like argparse provide a more structured and user-friendly way to handle command-line arguments.

Example using sys.argv:

CODE:

```
import sys

# Access command-line arguments

if len(sys.argv) > 1:

    print(f"Hello, {sys.argv[1]}!")

else:

    print("Hello, World!")
```

Command-line execution:

OUTPUT:

```
python hello.py Alice
```

3. **Modules and the import Statement:**

Python code can be organized into modules, which are files containing Python code. Modules can be imported into other Python scripts using the import statement.

Example of using an imported module:

CODE:

```
# mymodule.py

def greet(name):

    return f"Hello, {name}!"


# main.py

import mymodule


result = mymodule.greet("Alice")

print(result)
```

4. **The Python Standard Library:**

The Python Standard Library is a comprehensive set of modules and functions that come with Python. It covers a wide range of tasks, from file I/O and regular expressions to networking and data manipulation. You can use these modules without needing to install additional packages.

Example of using the math module from the standard library:

CODE:

```
import math


result = math.sqrt(16)

print(result)  # Output: 4.0
```

The Python Standard Library is a valuable resource for Python developers, as it provides solutions to many common programming tasks. You can explore the library's documentation to find modules that suit your specific needs.

5. **Define a Class with class:**

A class is defined using the class keyword in Python. It serves as a blueprint for creating objects, specifying their attributes (variables) and methods (functions).

CODE:

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age
```

6. **Inheritance:**

   Inheritance allows you to create a new class that inherits attributes and methods from an existing class. The new class is called the child or subclass.

CODE:

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id
```

7. **Override a Method:**

You can override a method in the child class by defining a method with the same name. This allows the child class to provide its own implementation.

8. **Add a Method:**

You can add new methods to a class to extend its functionality. This is done by defining methods within the class.

CODE:

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

    def greet(self):
        return f"Hello, my name is {self.name}."
```

9. **Get Help from Parent with super:**

The super() function is used to call a method from the parent class, enabling you to access and use the parent class's methods in the child class.

10. **In self Defense:**

The self keyword refers to the instance of the class. It's used to access and modify the instance's attributes and methods.

11. **Get and Set Attribute Values with Properties:**

You can use properties to get and set attribute values while encapsulating the implementation details.

CODE:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width  # Private attribute
        self._height = height  # Private attribute
    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        if value > 0:
            self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value > 0:
            self._height = value
```

12. **Name Mangling for Privacy:**

In Python, you can use name mangling to make attributes "private" by prefixing them with double underscores. This does not make them entirely private but makes them less accessible.

CODE:

```
class MyClass:
    def __init__(self):
        self.__private_var = 42
```

13. **Method Types:**

There are three primary types of methods in Python classes:

Instance methods: Take self as the first parameter and work on instance-specific data.

Class methods: Take cls as the first parameter and work on class-specific data.

Static methods: Do not take self or cls and work with data that is not instance- or class-specific.

14. **Duck Typing:**

Python follows the "duck typing" principle, meaning that the type or class of an object is determined by its behavior, not its explicit type. If an object quacks like a duck, it's treated as a duck.

15. **Special Methods:**

Python has a set of special methods, also known as "magic methods" or "dunder methods," that allow you to define how objects of a class behave in various contexts. For example, __init__ is used for object initialization, and __str__ for string representation.

16. **Composition:**

Composition is the practice of building more complex classes by combining or "composing" simpler classes. It promotes code reusability and modular design.

CODE:

```
class Engine:
    def start(self):
        print("Engine started")


class Car:
    def __init__(self):
        self.engine = Engine()


    def start(self):
        self.engine.start()
```

These concepts are essential for working with object-oriented programming in Python. Classes and objects provide a powerful way to structure and organize your code, leading to more maintainable and reusable solutions.

UNIT-4

1. **File Input / Output (I/O):**

Python provides built-in functions to work with files, making it simple to store and retrieve data. You can open, read, and write to files using the open() function.

Example: Storing data to a text file and retrieving it.

CODE:

```
# Storing data
with open('data.txt', 'w') as file:
    file.write("This is some data to store.")


# Retrieving data
with open('data.txt', 'r') as file:
    data = file.read()
print(data)
```

2. **Structured Text Files:**

Common structured formats like CSV (Comma-Separated Values) and JSON (JavaScript Object Notation) are widely used for storing structured data.

Example: Storing and retrieving data in JSON format.

CODE:

```
import json


data = {"name": "John", "age": 30, "city": "New York"}


# Storing data in JSON
with open('data.json', 'w') as json_file:
    json.dump(data, json_file)


# Retrieving data from JSON
with open('data.json', 'r') as json_file:
    retrieved_data = json.load(json_file)
```

```
print(retrieved_data)
```

3. **Relational Databases:**

Python supports various database connectors (e.g., sqlite3, MySQLdb, psycopg2) that allow you to interact with relational databases. You can create, read, update, and delete data in structured tables using SQL queries.

Example: Storing and retrieving data in a SQLite database.

CODE:

```
import sqlite3


# Storing data

conn = sqlite3.connect('mydb.db')

cursor = conn.cursor()

cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)")

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("John", 30))

conn.commit()

conn.close()


# Retrieving data

conn = sqlite3.connect('mydb.db')

cursor = conn.cursor()

cursor.execute("SELECT * FROM users")

retrieved_data = cursor.fetchall()

conn.close()

print(retrieved_data)
```

4. **NoSQL Data Stores:**

NoSQL databases like MongoDB or Redis can be used to store and retrieve data in a more flexible, non-relational format. You'll need to use specific libraries or clients for each NoSQL database.

5. **Web Services:**

Python can interact with web services and APIs to retrieve data from remote sources. Libraries like requests are commonly used for this purpose.

Example: Retrieving data from a web service.

CODE:

```
import requests
```

```
response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

data = response.json()

print(data)
```

The choice of data storage method depends on the nature of your data, the size of your dataset, and the use case. For small-scale data, files and structured text formats might suffice. For larger-scale applications with structured data, relational databases are a good choice. NoSQL databases are preferred for more flexible, unstructured data. Web services are ideal for data from remote sources, APIs, or web scraping.

6. **Web Clients:**

A web client in Python is an application that makes HTTP requests to retrieve web content or interact with web services. The most commonly used library for this purpose is requests. Here's a basic example of using it to retrieve data from a website:

CODE:

```
import requests


response = requests.get('https://www.example.com')

print(response.text)  # The HTML content of the webpage
```

7. **Web Servers:**

While Python is not the most common choice for building web servers, you can create simple web servers using libraries like Flask or Django. Here's a simple example using Flask to create a basic web server:

CODE:

```
from flask import Flask


app = Flask(__name)


@app.route('/')

def hello():

    return "Hello, World!"


if __name__ == '__main__':

    app.run()
```

When you run this script, it starts a web server that listens on port 5000 and responds with "Hello, World!" when you visit the root URL (http://localhost:5000).

8. **Web Services:**

Web services are often accessed by making HTTP requests to APIs. You can use the requests library to interact with web services by sending GET, POST, PUT, or DELETE requests. Here's an example of making a GET request to a JSON-based API:

CODE:

```
import requests


response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

data = response.json()

print(data)
```

9. **Automation:**

You can automate web-related tasks using Python for various purposes, such as web scraping, form filling, or interacting with web services. To automate interactions with websites, you can use libraries like Selenium or Beautiful Soup.

Example of using Selenium to automate a web browser:

CODE:

```
from selenium import webdriver


# Create an instance of a web browser (e.g., Chrome)

driver = webdriver.Chrome()


# Navigate to a website

driver.get('https://www.example.com')


# Find and interact with web elements (e.g., fill out a form)

search_box = driver.find_element_by_name('q')

search_box.send_keys('Python automation')

search_box.submit()


# Extract data from the page

search_results = driver.find_elements_by_css_selector('.g')
```

```
for result in search_results:
    print(result.text)


# Close the web browser
driver.quit()
```

Remember to install the required libraries (e.g., requests, Flask, Selenium) using pip before using them in your Python projects. These are just basic examples; you can build more complex applications and automation scripts tailored to your specific use cases.

UNIT-5

1. **Files and Directories:**

To work with files and directories in Python, you can use the built-in os and shutil modules. The os module provides functions for interacting with the operating system, including file and directory operations.

Example of creating a directory and writing a file:

CODE:

```
import os


# Create a directory

os.mkdir('my_directory')


# Write to a file

with open('my_directory/my_file.txt', 'w') as file:

    file.write('Hello, World!')
```

2. **Programs and Processes:**

You can run external programs and manage processes in Python using the subprocess module. It allows you to execute shell commands and interact with their input and output streams.

Example of running an external program:

CODE:

```
import subprocess


result = subprocess.run(['ls', '-l'], stdout=subprocess.PIPE, text=True)

print(result.stdout)
```

3. **Calendar and Clocks:**

Python provides the datetime module to work with dates and times. You can use it to retrieve the current date and time, format dates, and perform various date-related operations.

Example of working with dates and times:

CODE:

```
import datetime


# Get the current date and time
```

```
now = datetime.datetime.now()
print(now)

# Format a date
formatted_date = now.strftime('%Y-%m-%d %H:%M:%S')
print(formatted_date)
```

For more advanced calendar and scheduling functionality, you can explore external libraries like schedule or calendar.

4. **Queues:**

Queues are used to facilitate communication and coordination between different parts of a program, typically in a concurrent or parallel context. In Python, the queue module provides various queue implementations, including Queue, LifoQueue, and PriorityQueue.

Example of using Queue for producer-consumer concurrency:

CODE:

```
import queue
import threading

def producer(q):
    for i in range(5):
        q.put(i)

def consumer(q):
    while True:
        item = q.get()
        print(f"Consumed: {item}")
        q.task_done()

q = queue.Queue()
producer_thread = threading.Thread(target=producer, args=(q,))
consumer_thread = threading.Thread(target=consumer, args=(q))

producer_thread.start()
```

```
consumer_thread.start()
```

5. **Processes:**

The multiprocessing module in Python allows you to create and manage multiple processes to achieve parallelism.

Example of using multiprocessing for parallel execution:

CODE:

```
import multiprocessing


def worker(num):
    print(f"Worker {num}")


processes = []
for i in range(4):
    process = multiprocessing.Process(target=worker, args=(i,))
    processes.append(process)
    process.start()


for process in processes:
    process.join()
```

6. **Threads:**

Python's threading module provides a way to create and manage threads for concurrent execution. However, due to the Global Interpreter Lock (GIL), Python threads are not suitable for CPU-bound tasks but are useful for I/O-bound tasks.

Green Threads and Gevent:

Gevent is a Python library that provides a high-level, cooperative multitasking framework for I/O-bound operations. It uses green threads (coroutines) to achieve concurrency without creating separate system threads.

Example of using Gevent for concurrent I/O operations:

CODE:

```
import gevent
from gevent import monkey


monkey.patch_all()
```

```
def task1():
    print("Task 1 started")
    gevent.sleep(1)
    print("Task 1 completed")


def task2():
    print("Task 2 started")
    gevent.sleep(0.5)
    print("Task 2 completed")


gevent.joinall([gevent.spawn(task1), gevent.spawn(task2)])
```

7. **Twisted:**

Twisted is an event-driven networking engine and framework for building networked applications. It provides abstractions for handling asynchronous network communication, making it well-suited for building servers and clients with high concurrency and scalability requirements.

8. **Redis:**

Redis is an in-memory data store that supports various data structures and provides high-performance, distributed data storage. It can be used for building concurrent applications and implementing task queues.

Example of using Redis as a task queue:

CODE:

```
import redis


r = redis.StrictRedis(host='localhost', port=6379, db=0)


# Push a task onto the queue
r.lpush('task_queue', 'task_data')


# Pop a task from the queue
task = r.rpop('task_queue')
print(f"Task: {task}")
```

These are just some of the methods and libraries you can use to implement concurrency in Python, depending on your specific use case and requirements. The choice of method depends on the nature of the tasks, performance needs, and the type of concurrency you want to achieve.

9. Network Patterns:

Network patterns refer to the high-level architectural models and communication paradigms used in networked applications. Patterns like Request-Response, Publish-Subscribe, and Peer-to-Peer are commonly used.

The Publish-Subscribe Model:

In the Publish-Subscribe model, publishers send messages to a topic or channel, and subscribers receive messages from that topic. Libraries like paho-mqtt or redis-py can be used for implementing publish-subscribe systems.

TCP/IP and Sockets:

The Transmission Control Protocol (TCP) and Internet Protocol (IP) are the foundational protocols of the internet. Python's socket module allows you to create network sockets for communication over TCP/IP.

Example of creating a simple TCP server and client:

CODE:

```
import socket


# Server

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind(('localhost', 12345))

server_socket.listen(5)


client_socket, client_address = server_socket.accept()

data = client_socket.recv(1024)


# Client

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client_socket.connect(('localhost', 12345))

client_socket.send(b"Hello, server")
```

ZeroMQ:

ZeroMQ is a high-performance asynchronous messaging library that simplifies complex network communication patterns. The pyzmq library is used for working with ZeroMQ in Python.

10. Internet Services:

Internet services refer to a wide range of services and applications available on the internet, such as email, web browsing, and instant messaging.

11. **Web Services and APIs:**

Web services and APIs provide a structured way for software systems to communicate over the internet. Python libraries like requests are commonly used to interact with web services and APIs.

Example of making an API request using the requests library:

CODE:

```
import requests


response = requests.get('https://api.example.com/data')

data = response.json()
```

12. **Remote Processing:**

Remote processing involves executing code or tasks on remote servers or distributed systems. Python provides libraries like paramiko for SSH-based remote execution and tools like Celery for distributed task processing.

Big Fat Data and MapReduce:

Big data processing involves handling large volumes of data using parallel processing techniques. The Hadoop ecosystem, including Hadoop Streaming with Python, can be used for MapReduce tasks.

Working in the Clouds:

Cloud computing platforms like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform provide cloud-based infrastructure and services. Python libraries and SDKs are available for interacting with these platforms.

Example of listing AWS S3 buckets using the boto3 library:

CODE:

```
import boto3


s3 = boto3.client('s3')

response = s3.list_buckets()

buckets = [bucket['Name'] for bucket in response['Buckets']]
```

These network-related topics cover a wide range of capabilities in Python, from basic network communication to advanced big data processing and cloud integration. The specific libraries and tools you use will depend on your use case and requirements.