Course Agenda

- Introduction to Python Programming
- Programming Basics - variables, operators, decision making, iteration, sequences
- Data Structures in Python - list, tuple, set, dictionary
- Functions - Functiona Arguments, lambda functions, function objects, map - filter - reduce
- Exception Handling
- Object Oriented Programming
- Regular Expressions
- Connecting to external data sources
- requests library

Introduction to Python Programming

- Data Types in Python
- Variables in Python
- Data Type conversion
- Accept inputs from user
- Operators in Python

Python is –

- Open source
- Interpreter based
- Platform independent
- Current version – 3.11.x
- Download it from here - https://www.python.org/download/releases/3.0/

Python Data Types

1. int: Integer numbers, e.g., 42, -7
2. float: Floating-point numbers (decimal), e.g., 3.14, -0.001
3. complex: Complex numbers, e.g., 1+2j, -3+4j
4. bool: Boolean values, either True or False
5. str: String, a sequence of characters, e.g., "hello", 'world'
6. bytes: Immutable sequence of bytes, e.g., b'hello'

In [ ]:
```
10 # int
```

In [ ]:
```
15.5 # float
```

```
In [ ]: "abcd" # str
```

```
In [ ]: True # bool
```

```
In [ ]: complex(2,3) # complex
```

Python Containers or Data structures

Containers are any object that holds an arbitrary number of other objects. Generally, containers provide a way to access the contained objects and to iterate over them. Examples of built-in containers include tuples, lists, sets, dictionary.

1. `List`
   - Definition: An ordered, mutable (changeable) collection of items.
   - Syntax: Created using square brackets [].
2. `Tuple`
   - Definition: An ordered, immutable (unchangeable) collection of items.
   - Syntax: Created using parentheses ().
3. `Set`
   - Definition: An unordered collection of unique items.
   - Syntax: Created using curly braces {} or the set() function.
4. `Dictionary`
   - Definition: An unordered collection of key-value pairs. Keys must be unique and immutable.
   - Syntax: Created using curly braces {} with key-value pairs separated by colons :.

```
In [ ]: [1, 2, 3, 4, 'abc'] # list
```

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

```
In [ ]: {2, 's', 3, 5, 5} # set
```

```
In [ ]: {1 : "Jane", 2:"George", 3:"Sam"} # dictionary
```

Variables in Python

- A Python variable points to a reserved memory location
- Data or objects are stored in these memory locations
- Variables can be declared by any name or even alphabets

Ex. Define variable name and assign value to the variable

```
In [1]: name = "Jane"
        print("Welcome", name)
```

```
Welcome Jane
```

`type()` - returns class type of the argument(object) passed as parameter

In [2]:
```python
a = True
print(type(a))
```
```
<class 'bool'>
```

In [3]:
```python
a = 10
print(type(a))
```
```
<class 'int'>
```

In [4]:
```python
a = 10.5
print(type(a))
```
```
<class 'float'>
```

In [5]:
```python
a = "abcd"
print(type(a))
```
```
<class 'str'>
```

In [6]:
```python
lst = [1, 2, 3, 4]
print(type(lst))
```
```
<class 'list'>
```

Ex. WAP to take name of user as input and print a welcome message

In [7]:
```python
name = input("Enter your name - ")
print("Welcome", name)
```
```
Welcome George
```

Ex. WAP to take two numbers as input from user and print their sum.

In [12]:
```python
num1 = int(input("Enter a number - "))
num2 = int(input("Enter a number - "))

print(num1 + num2)
```
```
12
```

Data Type Conversion

`Implicit Conversion:` Conversion done by Python interpreter without programmer's intervention

In [9]:
```python
a = 10  # int
b = 2.5 # float
a + b
```
Out[9]:
```
12.5
```

`Explicit Conversion:` Conversion that is user-defined that forces an expression to be of specific data type

In [11]:
```
a = "10"  # str
b = 5     # int

int(a) + b
```
Out[11]:
```
15
```

Common Type Casting Functions:

- int(): Converts a value to an integer.
- float(): Converts a value to a floating-point number.
- str(): Converts a value to a string.
- bool(): Converts a value to it bool equivalent
- list(): Converts a value to a list.
- tuple(): Converts a value to a tuple.
- set(): Converts a value to a set.
- dict(): Converts a sequence of key-value pairs into a dictionary.

`int()` conversion

In [13]:
```
x = 10.8 # float
int(x)
```
Out[13]:
```
10
```

In [14]:
```
x = "10" # int value in str format
int(x)
```
Out[14]:
```
10
```

In [15]:
```
x = True # bool
int(x)
```
Out[15]:
```
1
```

In [16]:
```
x = False # bool
int(x)
```
Out[16]:
```
0
```

In [17]:
```
x = "abcd" # str
int(x)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 2
      1 x = "abcd" # str
----> 2 int(x)

ValueError: invalid literal for int() with base 10: 'abcd'
```

```
In [18]: x = "10.8" # float value in str format
         int(x)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 2
      1 x = "10.8" # float value in str format
----> 2 int(x)

ValueError: invalid literal for int() with base 10: '10.8'
```

`float()` conversion

```
In [19]: x = True # bool
         float(x)
```

```
Out[19]: 1.0
```

```
In [20]: x = False # bool
         float(x)
```

```
Out[20]: 0.0
```

```
In [21]: x = 10 # int
         float(x)
```

```
Out[21]: 10.0
```

```
In [22]: x = "10.8" # float value in str format
         float(x)
```

```
Out[22]: 10.8
```

```
In [23]: x = "abcd" # str
         float(x)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[23], line 2
      1 x = "abcd" # str
----> 2 float(x)

ValueError: could not convert string to float: 'abcd'
```

`str()` conversion

```
In [24]: x = 10
         str(x)

Out[24]: '10'

In [25]: lst = [1,2,3,4,5]
         str(lst)

Out[25]: '[1, 2, 3, 4, 5]'
```

`bool()` conversion

```
In [26]: x = 0
         bool(x)

Out[26]: False

In [27]: x = 1
         bool(x)

Out[27]: True

In [28]: x = 0.0
         bool(x)

Out[28]: False

In [29]: x = 1.0
         bool(x)

Out[29]: True

In [30]: x = 10
         bool(x)

Out[30]: True

In [31]: x = "abc"
         bool(x)

Out[31]: True

In [32]: x = "True"
         bool(x)

Out[32]: True

In [33]: x = "False"
         bool(x)

Out[33]: True
```

In [34]:
```
x = None
bool(x)
```
Out[34]:
```
False
```

Note - bool() returns True for any value except 0 or 0.0 or None

---

Operators in Python

Operators are special symbols in Python that carry out computations. The value that the operator operates on is called as operand.

1. Arithmetic Operators

These operators perform arithmetic operations on numeric values.

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : Division
- `%` : Modulus (remainder of division)
- `**` : Exponentiation (power)
- `//` : Floor division (division that results in the largest integer less than or equal to the quotient)

In [35]:
```
a = 10
b = 6
```
In [36]:
```
print("Addition - ", a+b)
```
```
Addition -  16
```
In [37]:
```
print("Substraction - ", a-b)
```
```
Substraction -  4
```
In [38]:
```
print("Multiplication - ", a*b)
```
```
Multiplication -  60
```
In [39]:
```
print("Division - ", a/b) # returns result in float format
```
```
Division -  1.6666666666666667
```
In [40]:
```
print("Floor Division - ", a//b) # returns integer part of the division
```
```
Floor Division -  1
```
In [41]:
```
print("Exponential - ", a**b) # returns the result of a to the power b
```

```
Exponential -  1000000
```

In [42]:
```
print("Modulous - ", a%b) # returns remainder of the division
```

```
Modulous -  4
```

Ex. WAP to calculate BMI of a person.

In [43]:
```
weight = int(input("Enter your weight in kgs - "))
height = float(input("Enter your height in mtrs - "))
bmi = weight / (height ** 2)

print("BMI - ", round(bmi, 2))
```

```
BMI -  21.26
```

Ex. WAP to accept hours and rate per hour from user and compute gross pay.

In [ ]:
```

```

2. Comparison Operators

These operators compare two values and return a boolean result (True or False).

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

In [44]:
```
a = 10
b = 7
```

In [45]:
```
print(a < b)
```

```
False
```

In [46]:
```
print(a <= b)
```

```
False
```

In [47]:
```
print(a > b)
```

```
True
```

In [48]:
```
print(a >= b)
```

```
True
```

In [49]:
```
print(a==b)
```

```
False
```

In [50]:
```
print(a!=b)
```

```
True
```

Ex. WAP to take a number as input and write condition to check if the number is greater than 10.. (Output must be a bool value)

In [53]:
```
num = int(input("Enter a number - "))
num > 10
```

Out[53]:
```
False
```

Ex. WAP to take a number as input and write condition to check if the number is divisible by 5. (Output must be a bool value)

In [55]:
```
num = int(input("Enter a number - "))
num % 5 == 0
```

Out[55]:
```
True
```

3. Logical Operators

These operators are used to combine conditional statements.

- `and` : Returns True if both statements are true
- `or` : Returns True if at least one of the statements is true
- `not` : Reverses the result, returns False if the result is true

Ex. WAP to check if the number is divisible by 5 and greater than 10

In [58]:
```
num = int(input("Enter a number - "))
num % 5 == 0 and num > 10
```

Out[58]:
```
False
```

Ex. WAP to check if the number is either divisible by 5 or greater than 10 or both

In [60]:
```
num = int(input("Enter a number - "))
num % 5 == 0 or num > 10
```

Out[60]:
```
True
```

In [61]:
```
not True
```

Out[61]:
```
False
```

In [62]:
```
not False
```

Out[62]:
```
True
```

In [63]:
```
not(10 - 5 * 2)
```

Out[63]:
```
True
```

In [ ]:
```

```

4. Basic Assignment Operator

`=` : Assigns the value on the right to the variable on the left.

Compound Assignment Operators

These operators perform an operation on a variable and then assign the result back to that variable.

- `+=` : Adds the right operand to the left operand and assigns the result to the left operand.
- `-=` : Subtracts the right operand from the left operand and assigns the result to the left operand.
- `*=` : Multiplies the left operand by the right operand and assigns the result to the left operand.
- `/=` : Divides the left operand by the right operand and assigns the result to the left operand.
- `%=` : Takes the modulus of the left operand by the right operand and assigns the result to the left operand.
- `//=` : Performs floor division on the left operand by the right operand and assigns the result to the left operand.
- `**=` : Raises the left operand to the power of the right operand and assigns the result to the left operand.

In [ ]:
```
x = 10 + 5 * 3
```

In [64]:
```
a = 10

a = a + 5
```

In [ ]:
```
a += 5
```

Membership Operators

Membership operators checks whether a value is a member of a sequence.

Sequence Object -

A `sequence` is defined as a collection of arbitrary number of elements. The sequence may be a list, a string, a tuple, or a dictionary

- in - The `in` operator is used to check if a value exists in any sequence object or not.
- not in - A `not in` works in an opposite way to an 'in' operator. A 'not in' evaluates to True if a value is not found in the specified sequence object. Else it returns a False.

In [65]:
```
"a" in "abcd"
```

```
Out[65]:  True
```

```
In [66]: "a" not in "abcd"
```

```
Out[66]:  False
```

Ex. WAP to check if entered name is present in the mentioned list or not

```
In [67]: names = ["Jane", "George", "Sam"]

         "Jane" in names
```

```
Out[67]:  True
```

```
In [68]: names = ["Jane", "George", "Sam"]

         "Jack" in names
```

```
Out[68]:  False
```

```
In [69]: names = ["Jane", "George", "Sam"]

         "J" in names
```

```
Out[69]:  False
```

Ex. WAP to check if entered character is a vowel or not

```
In [71]: ch = input("Enter a character - ")
         ch in "aeiou"
```

```
Out[71]:  False
```

Ex. WAP to calculate the hypoteneous of a right angled triangle when sides are given

```
In [72]: import math
         ht = 4
         b = 3
         hypt = math.sqrt((ht ** 2) + (b ** 2))
         hypt
```

```
Out[72]:  5.0
```

Decision Making

Decision-making statements in Python allow you to control the flow of execution based on certain conditions. These statements include if, elif, and else and can be used to execute different blocks of code depending on whether conditions are True or False.

- **Basic if Statement**
  - The if statement evaluates a condition and executes a block of code if the condition is True.
  - Syntax:

```
if condition:
    # block of code
```

- **if-else Statement**
  - The if-else statement evaluates a condition and executes one block of code if the condition is True, and another block if the condition is False.
  - Syntax:

```
if condition:
    # block of code if condition is True
else:
    # block of code if condition is False
```

- **if-elif-else Statement**
  - The if-elif-else statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions evaluates to True.
  - If none of the conditions are True, the else block is executed.
  - Syntax:

```
if condition1:
    # block of code if condition1 is True
elif condition2:
    # block of code if condition2 is True
elif condition3:
    # block of code if condition3 is True
else:
    # block of code if none of the conditions are True
```

- **Nested if Statements**
  - You can nest if statements within other if statements to check multiple conditions in a hierarchical manner.
  - Syntax:

```
if condition1:
    # block of code if condition1 is True
    if condition2:
        # block of code if condition2 is True
    else condition3:
        # block of code if condition2 is False
else:
    # block of code if condition1 is False
```

- **One-Line if-else**
  - Syntax:
    `value_if_true if condition else value_if_false`

Examples -

Ex. Check if the character is vowel or not

In [74]:
```
ch = input("Enter a character - ")
if ch in "aeiou" :
    print("Vowel")
else:
    print("Consonant")
```

```
Consonant
```

Ex. Check if the character is vowel or not, ignore cases

In [76]:
```
ch = input("Enter a character - ").lower()
if ch in "aeiou" :
    print("Vowel")
else:
    print("Consonant")
```

```
Vowel
```

Ex. Check if the character is vowel or not

- ignore cases
- check if single character else print invalid

In [81]:
```
ch = input("Enter a character - ").lower()
if len(ch) == 1 :
    if ch in "aeiou" :
        print("Vowel")
    else:
        print("Consonant")
else:
    print("Invalid")
```

```
Consonant
```

Ex. Check if the character is vowel or not

- ignore cases
- check if single character else print invalid
- validate for non alpha characters

In [83]:
```
ch = input("Enter a character - ").lower()
if len(ch) == 1 and ch.isalpha():
    if ch in "aeiou" :
        print("Vowel")
    else:
        print("Consonant")
else:
    print("Invalid")
```

```
Invalid
```

Ex. Check if the character is vowel or not

- ignore cases
- check if single character else print invalid
- validate for non alpha characters
- custom error messages

In [88]:
```
ch = input("Enter a character - ").lower()
if len(ch) > 1 :
    print("More than one character")
elif not ch.isalpha():
    print("Not an alphabet")
elif ch in "aeiou" :
    print("Vowel")
else:
    print("Consonant")
```

```
Consonant
```

Ex. WAP to take weight in kgs and height in mtrs from user. Calculate BMI and print if the the health status based on following chart

BMI Categories

- Underweight - BMI < 18.5
- Normal weight - BMI between 18.5-24.9
- Overweight - BMI between 25-29.9
- Obesity - BMI > 30

In [ ]:
```

```

Ex. WAP to accept hours and rate per hour from user and compute gross pay.

- for 40 hrs pay the standard rate
- if overtime then pay 1.5 times of rate for the additional hrs.

In [91]:
```
rate= int(input("Enter rate per hour - "))
hrs = int(input("Enter number of hours worked - "))

if hrs <= 40 :
    gross_pay = rate * hrs
else:
    gross_pay = (40 * rate) + ((hrs - 40) * 1.5 * rate)
print(gross_pay)
```

```
4750.0
```

Ex. Toss a coin and guess the outcome

WAP to simulate coin toss and compare the outcome with guess made by user. The program must return "Invalid input" if user enters a wrong or invalid value as a guess.

- Take guess as heads or tails as input from user
- Validate input - If guess is heads or tails then valid else print invalid
- for valid guess
    - generate outcome randomly as heads or tails
    - compare guess with outcome and print win or lost

In [95]:
```
import random
guess = input("Enter your guess as heads or tails - ").lower()
choices = ("heads", "tails")
if guess in choices :
    outcome = random.choice(choices)
    print("Outcome generated - ", outcome)
    if guess == outcome :
        print("Win")
    else:
        print("Lost")
else:
    print("Invalid input")
```

```
Outcome generated -  tails
Win
```

In [97]:
```
import math

print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'a
sin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1', 'fabs', 'fact
orial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'lo
g1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'r
emainder', 'sin', 'sinh', 'sqrt', 'sumprod', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

In [100…
```
math.pi
```

Out[100…
```
3.141592653589793
```

random module in Python

- Generates a random value

- Importing random module
  import random as r

- Frequently used functions

    - **r.random()** – Generates a random float number between 0.0 to 1.0
    - **r.randint()** – Returns a random integer between the specified integers

- **r.randrange()** – Returns a randomly selected element from the range created by the start, stop and step arguments
- **r.choice()** – Returns a randomly selected element from a non-empty sequence
- **r.shuffle()** – This functions randomly reorders the elements in a list

In [ ]:
```
import random as r # as is an operator - used to assign a variable to a moudule whi
```

In [ ]:
```
r.random() # generates a random value between 0 and 1
```

In [ ]:
```
r.randint(1,5) # generates a random integer between the mentioned start and end val
```

In [ ]:
```
r.choice([1,2,3,4]) # generates a random value from the mentioned iterable or seque
```

In [ ]:
```
r.choice(["abc", "pqr", "xyz"])
```

In [ ]:
```
r.choice("abcde")
```

Loops

Loops are used to execute of a specific block of code in repetitively

while loop

- The ‘while loop’ in Python is used to iterate over a block of code as long as the test expression holds true
- Event based loop
- Event occurring inside the loop determines the number of iterations
- This loop is used when the number of times to iterate is not known to us beforehand

Ex. Write a code to validate user input for an integer input

In [102…
```
num = input("Enter a number - ")
while not num.isdigit():
    num = input("You have entered wring input please enter a number - ")
num = int(num)
print(num**2)
```

```
25
```

In [103…
```
while True:
    num = input("Enter a number - ")
    if num.isdigit() :
        num = int(num)
        break
    print("Invalid input")
print(num**2)
```

```
Invalid input
Invalid input
Invalid input
25
```

break statement

- The ‘break’ statement ends the loop and resumes execution at the next statement
- The break statement can be used in both ‘while’ loop and ‘for’ loop
- It is always used with conditional statements

continue statement

- The ‘continue’ statement in Python ignores all the remaining statements in the iteration of the current loop and moves the control back to the beginning of the loop
- The continue statement can be used in both ‘while’ loop and ‘for’ loop
- It is always used with conditional statements

for loop

- The ‘for loop’ in Python is used to iterate over the items of a sequence object like list, tuple, string and other iterable objects
- The iteration continues until we reach the last item in the sequence object
- Counter driven loop
- This loop is used when the number of times to iterate is predefined

Ex. WAP to print square of numbers in the given list

In [104…

```
numbers = [1, 2, 3, 4, 5]
for i in numbers :
    print(i, " - ", i **2)
```

```
1  -  1
2  -  4
3  -  9
4  -  16
5  -  25
```

Ex. WAP to print square of all even numbers in the given list

In [105…

```
numbers = [1, 2, 3, 4, 5]
for i in numbers :
    if i % 2 == 0 :
        print(i, " - ", i **2)
```

```
2  -  4
4  -  16
```

Ex. WAP to accept a word from user and print vowels in the word

In [108…
```
word = input("Enter a word - ")
flag = True
for i in "aeiou" :
    if i in word :
        print(i)
        flag = False
if flag :
    print("There are no vowels")
```

```
a
e
i
o
```

Ex. Write a python code to print the product of all elements in the numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11, 43]?

In [109…
```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11, 43]
math.prod(numbers)
```

Out[109…
```
54489600
```

In [110…
```
sum(numbers)
```

Out[110…
```
91
```

Ex. WAP to perform product of first 10 natural numbers

In [ ]:
```

```

range(`[start], stop, [step]`)

- `start` - Optional. An integer number specifying at which position to start. Default is 0
- `stop` - Required. An integer number specifying at which position to end.
- `step` - Optional. An integer number specifying the incrementation. Default is 1

In [111…
```
range(1, 11)   # 1 - 10
```

Out[111…
```
range(1, 11)
```

In [112…
```
for i in range(1,11):
    print(i, end = ",")
```

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

In [113…
```
for i in range(11):
    print(i, end = ",")
```

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

In [114…
```
for i in range(1, 50, 3):
    print(i, end = ", ")
```

```
1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49,
```

In [115…

```python
for i in range(1, 50, 5):
    print(i, end = ", ")
```

```
1, 6, 11, 16, 21, 26, 31, 36, 41, 46,
```

In [116…

```python
for i in range(50, 0, -1):
    print(i, end = ", ")
```

```
50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30,
29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9,
8, 7, 6, 5, 4, 3, 2, 1,
```

Examples

Ex. WAP to accept an integer from user and print a table for given number.

3 x 1 = 3

.

.

3 x 10 = 30

In [ ]:

Ex. WAP to print prime number between 1 - 100

In [ ]:

for .. else

- if a for loop is `terminated` abruptly using a `break` statement then else block is `NOT` executed
- if for loop is executed successfully till its last iteration, then else block is `executed`

In [ ]:

Python Containers

Object	Container Object	Sequence Type	Element Type	Enclosed in	Immutability	Duplicates
str()	No	ordered/indexed	characters	"" or ''	Yes	Yes
tuple()	Yes	ordered/indexed	mixed data (heterogeneous)	()	Yes	Yes

Object	Container Object	Sequence Type	Element Type	Enclosed in	Immutability	Duplicates
list()	Yes	ordered/indexed	mixed data (heterogeneous)	[]	No	Yes
set()	Yes	unordered	heterogeneous (immutable objects)	{}	No	No
dict()	Yes	unordered	Key - immutable Value - any type	{}	No	Key - No Value - Yes

---

Strings in Python

Strings are -

- an ordered sequence of characters
- enclosed in a pair of single quotes or pair of double quotes
- immutable

Empty string

In [117…
```
string = ''

string = ""
```

In [119…
```
word = input("Enter a word - ")
if word :
    print(word)
else:
    print("Empty string")
```

```
Empty string
```

Note - bool() of empty str is always

Defining a string

In [120…
```
string = "aero-plane"
```

Indexing in Strings

- Each character in a string has a unique index, starting from 0 for the first character up to n-1 for the last character, where n is the length of the string.
- **Positive Indexing** - Positive indexing starts from 0 and goes up to n-1.
  - Index 0 corresponds to the first character.
  - Index 1 corresponds to the second character, and so on.
- **Negative Indexing** - Negative indexing starts from -1 for the last character and goes up to -n for the first character.
  - Index -1 corresponds to the last character.
  - Index -2 corresponds to the second last character, and so on.
- **Accessing Substrings** - You can also use slicing to access substrings. The syntax for slicing is `string[start:stop:step]` , where:
  - start is the starting index (inclusive).
  - stop is the ending index (exclusive).
  - step is the step size (optional).

In [121…
```
string
```
Out[121…
```
'aero-plane'
```

Extract first element from the string

In [122…
```
string[0]
```
Out[122…
```
'a'
```

Extract 5th element from the string

In [123…
```
string[4]
```
Out[123…
```
'-'
```

Ex. Extract last element from the string

In [124…
```
string[-1]
```
Out[124…
```
'e'
```

Ex. Extract first 3 characters from string

In [125…
```
string[0:3]
```
Out[125…
```
'aer'
```

Ex. Extract all characters from index position 3

In [126…
```
string[3 : ]
```

```
Out[126… 'o-plane'
```

Ex. Extract last 4 characters from the string

```
In [128… string[-4 : ]
```

```
Out[128… 'lane'
```

Reverse of string

```
In [129… string[::-1]
```

```
Out[129… 'enalp-orea'
```

Ex. WAP to generate a new string by swapping first and last characers

ex - abcde - ebcda

```
In [130… word = input("Enter a word - ")
         word[-1] + word[1 : -1] + word[0]
```

```
Out[130… 'ebcda'
```

Operations on Strings

- Concatenation
- Repetition
- Membership
- Iteration

```
In [ ]:
```

`Concatenation` - merging two strings into a single object using the + operator.

```
In [131… "abc" + "pqr"
```

```
Out[131… 'abcpqr'
```

`Repetition` - The repetition operator * will make multiple copies of that particular object and combines them together.

```
In [133… print("-" * 50)
```

```
--------------------------------------------------
```

Strings are immutable and cannot be modified

```
In [135… word = "chennai"
         word[0]
```

```
Out[135… 'c'
```

```
In [136… word[0] = "C"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[136], line 1
----> 1 word[0] = "C"

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

```
In [ ]:
```

Built-in Functions

- **len()** - returns length of the string
- **min(), max()** - returns minimum and maximum element from the string
- **sorted()** - sorts the characters of the string and returns a list

```
In [137… string = "Mississippi"

         len(string)
```

```
Out[137… 11
```

```
In [138… min(string)
```

```
Out[138… 'M'
```

```
In [139… max(string)
```

```
Out[139… 's'
```

```
In [140… sorted(string) # returns a list object
```

```
Out[140… ['M', 'i', 'i', 'i', 'i', 'p', 'p', 's', 's', 's', 's']
```

Strings Methods

- **str.index(`obj`)** - returns index of the first occurence of the character
- **str.count(`obj`)** - returns count of number of occurences of the charcter
- **str.upper()** - returns string of uppercase characters
- **str.lower()** - returns string of lowercase characters
- **str.title()** - returns string of sentence case charaters
- **str.isupper()** - checks if all characters are uppercase

- **str.islower()** - checks if all characters are lowercase
- **str.isdigit()** - checks if all characters are digits
- **str.isalpha()** - checks if all characters are alphabets
- **str.isalnum()** - checks if all characters are either alphabets or digits
- **str.split(`delimiter`)** - splits the string on the mentioned delimiter and returns a list of obtained parts of the string
- **str.replace(`str` , `str`)** - replaces all the mentioned characters with the specified string and returns a new string
- **str.strip(`delimiter`)** - removes whitespace characters from start and end of the string (delimiter can also be specified)
- `delimiter` **.join**(`sequence`) - it is called on a string object which acts as a delimiter to join all string elements in the sequence passed as an argument to join()

In [141…
```
string
```
Out[141…
```
'Mississippi'
```
In [145…
```
string.index("i")
```
Out[145…
```
1
```
In [143…
```
string.count("i")
```
Out[143…
```
4
```
In [146…
```
string.lower()
```
Out[146…
```
'mississippi'
```
In [147…
```
string.upper()
```
Out[147…
```
'MISSISSIPPI'
```
In [148…
```
string.replace("i", "*")
```
Out[148…
```
'M*ss*ss*pp*'
```
In [150…
```
profit = "($1,200)"

obj = str.maketrans("(", "-", "$,)")
int(profit.translate(obj))
```
Out[150…
```
-1200
```

Replace all vowels in a string with *

In [151…
```
string = "Singapore"
obj = str.maketrans("aeiou", "*****")
string.translate(obj)
```

Out[151…
```
'S*ng*p*r*'
```

In [ ]:
```

```

Examples

Ex. WAP to convert the given string -

string = "I am in Python class"

o/p - 'ssalc nohtyP ni ma I'

o/p - 'I Am In Python Class'

In [152…
```
string = "I am in Python class"
string[::-1]
```

Out[152…
```
'ssalc nohtyP ni ma I'
```

In [153…
```
string.title()
```

Out[153…
```
'I Am In Python Class'
```

Ex. WAP to print foloowing pattern

In [ ]:
```
*
**
***
****
*****
```

In [154…
```
for i in range(1, 6):
    print("*"*i)
```

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

In [157…
```
string = "SingaPOre"
"p" in string
```

Out[157…
```
False
```

In [156…
```
string = "SingaPOre"
```

```
"p" in string.casefold()
```

Out[156…

```
True
```

Tuples in Python

Tuples -

- are Python containers
- are an ordered sequence of mixed data
- enclose elements in a pair of round brackets, separated by commas
- are immutable

Defining a tuple

In [158…

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

Out[158…

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

Empty Tuple

In [159…

```
tup = ()

tup = tuple()
```

Note - bool() of empty tuple is always False

Single value Tuple

In [160…

```
(10,)
```

Out[160…

```
(10,)
```

Indexing and Slicing

In [161…

```
fam = ("Rosie", 34, "Sam", 36, "Jonnah", 15, "Jessie", 12)
```

Ex. Extract 1st element from tuple -

In [162…

```
fam[0]
```

Out[162…

```
'Rosie'
```

Ex. Extract last element from tuple -

In [163…

```
fam[-1]
```

Out[163…

```
12
```

Ex. Extract first 3 elements from tuple -

In [164…

```
fam[0:3]
```

Out[164…

```
('Rosie', 34, 'Sam')
```

Ex. Extract last 2 element from tuple -

In [165…

```
fam[-2 :]
```

Out[165…

```
('Jessie', 12)
```

Ex. Extract "True" from the given Tuple

In [166…

```
mix_tuple = (('a', 1, True), 234567, 'Science', -5)
mix_tuple[0][2]
```

Out[166…

```
True
```

Operations on Tuples

- Iteration
- Membership
- Concatenation
- Repetition

In [ ]:

```
```

`Concatenation` **- merging two tuples into a single tuple object using the + operator.**

In [167…

```
(1, 2, 3) + (4, 5, 6)
```

Out[167…

```
(1, 2, 3, 4, 5, 6)
```

`Repetition` **- The repetition operator * will make multiple copies of that particular object and combines them together.**

In [169…

```
(1, 2, 3) * 2
```

Out[169…

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

Tuples are immutable hence cannot be modified

Built-in functions on Tuples

- **len()** - returns length of the tuple

- **min(), max()** - returns minimum and maximum element from the tuple
- **sorted()** - sorts the elements of the tuple and returns a list
- **sum()** - applicable to only numeric tuples, returns summation of all the elements int he tuple

In [170…
```
tup = (1, 5, 3, 7, 2, 4)
```

In [171…
```
len(tup)
```

Out[171…
```
6
```

In [172…
```
min(tup)
```

Out[172…
```
1
```

In [173…
```
max(tup)
```

Out[173…
```
7
```

In [174…
```
sorted(tup)
```

Out[174…
```
[1, 2, 3, 4, 5, 7]
```

In [175…
```
sum(tup)
```

Out[175…
```
22
```

Ex. WAP to reverse the tuple

In [176…
```
tup = (1, 4, 3, 2, 6, 5, 8)
tup[::-1]
```

Out[176…
```
(8, 5, 6, 2, 3, 4, 1)
```

Tuple Methods

- **tup.index(`object`)** - returns the index position of first occurence of the `object`
- **tup.count(`object`)** - returns the `count` of number of times the object is repeated in the tuple

In [177…
```
tup = ('car', 'bike', 'house',  'car', 'aeroplane', 'train', 'car')
```

In [178…
```
tup.index('car')
```

Out[178…
```
0
```

In [179…
```
tup.count('car')
```

Out[179…
```
3
```

Unpacking Tuples

In [180…

```
tup = 1, 2, 3  # packing of tuples

tup
```

Out[180…

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

In [181…

```
a, b, c = tup

a
```

Out[181…

```
1
```

In [182…

```
name, age = "Jane", 30

print(name, age)
```

```
Jane 30
```

Examples

Ex. WAP to print the word in the tuple and its len.

In [183…

```
tup = ('car', 'bike', 'house', 'aeroplane')
for i in tup :
    print(i, " - ", len(i))
```

```
car  -  3
bike  -  4
house  -  5
aeroplane  -  9
```

Ex. WAP to print the tuple and the sum of subtuples

In [184…

```
tup = ((3,9),(1,2,4),(1,4,8),(2,3))
for i in tup :
    print(i, " - ", sum(i))
```

```
(3, 9)  -  12
(1, 2, 4)  -  7
(1, 4, 8)  -  13
(2, 3)  -  5
```

Lists in Python

Lists -

- are Python containers

- are an ordered sequence of mixed data
- enclose elements in a pair of square brackets, separated by commas
- mutable

Empty List

```
In [185…
[]

list()
```

```
Out[185…
[]
```

Create a List

```
In [186…
friends = [ 'Ross', 'Monica', 'Joey', 'Chandler']
```

Retrive elements from List

Indexing and slicing

Ex. Extract first element from the list

```
In [187…
friends[0]
```

```
Out[187…
'Ross'
```

Ex. Extract second last element from the list

```
In [188…
friends[-2]
```

```
Out[188…
'Joey'
```

Ex. Extract first 3 elements from the list

```
In [189…
friends[0:3]
```

```
Out[189…
['Ross', 'Monica', 'Joey']
```

Add elements to List

lst.append(`object`) - appends element to the end of the list

Ex. Append 'Phoebe' to the end of the list

```
In [190…
friends.append("Phoebe")
print(friends)
```

```
['Ross', 'Monica', 'Joey', 'Chandler', 'Phoebe']
```

lst.insert(`object`) - inserts element at the mentioned index location

Ex. Insert 'Rachel' at index position 4

In [191…
```
friends.insert(4, "Rachel")
print(friends)
```

```
['Ross', 'Monica', 'Joey', 'Chandler', 'Rachel', 'Phoebe']
```

lst.extend(`seq-obj`) - always take a sequnce as parameter, appends all the elements from sequence to end of the list.

Ex. Extend tuple of new_friends to the end of the friends list

In [192…
```
tup = ("Jane", "Rosie", "Jasmine")
friends.extend(tup)
```

In [193…
```
print(friends)
```

```
['Ross', 'Monica', 'Joey', 'Chandler', 'Rachel', 'Phoebe', 'Jane', 'Rosie', 'Jasmin
e']
```

Modify elements in List

Ex. Replace 'Rosie' with "Amy"

In [194…
```
friends.index("Rosie")
```

Out[194…
```
7
```

In [195…
```
friends[7] = "Amy"
print(friends)
```

```
['Ross', 'Monica', 'Joey', 'Chandler', 'Rachel', 'Phoebe', 'Jane', 'Amy', 'Jasmine']
```

Remove elements from a list

lst.pop() - deletes the last element from the list

In [196…
```
friends.pop()
print(friends)
```

```
['Ross', 'Monica', 'Joey', 'Chandler', 'Rachel', 'Phoebe', 'Jane', 'Amy']
```

lst.pop(`index-value`) - deletes the element at the mentioned `index-value` from the list

In [197…
```
friends.pop(2)
print(friends)
```

```
['Ross', 'Monica', 'Chandler', 'Rachel', 'Phoebe', 'Jane', 'Amy']
```

lst.remove(`obj`) - removes the mention `obj` from the list

In [198…
```
friends.remove("Ross")
print(friends)
```

```
['Monica', 'Chandler', 'Rachel', 'Phoebe', 'Jane', 'Amy']
```

Using slicing operator to remove multiple element from a list

In [199…
```
del friends[0:2]
friends
```
Out[199…
```
['Rachel', 'Phoebe', 'Jane', 'Amy']
```

Operations on Lists

- Iteration
- Membership
- Concatenation
- Repetition

In [200…
```
list_1 = [1, 2, 3, 4]
```

Conacatenation

In [201…
```
list_1 + [5, 6, 7]
```
Out[201…
```
[1, 2, 3, 4, 5, 6, 7]
```

Repetition

In [202…
```
list_1 * 5
```
Out[202…
```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Built-in functions on Lists

- **len()** - returns length of the list
- **min(), max()** - returns minimum and maximum element from the list
- **sorted()** - sorts the elements of the list and returns a list
- **sum()** - applicable to only numeric lists, returns summation of all the elements int the list

In [203…
```
numbers = [3, 4, 2, 6, 5, 1]
```

In [204…
```
len(numbers)
```
Out[204…
```
6
```

In [205…
```
min(numbers)
```
Out[205…
```
1
```

In [206…
```
sorted(list_1)
```
Out[206…
```
[1, 2, 3, 4]
```
In [207…
```
sum(numbers)
```
Out[207…
```
21
```
Ex. WAP to create a list of marks(out of 100) of a student in 5 subjects. Print his percentage scored.

In [208…
```
marks = []
for i in range(5):
    marks.append(int(input("Enter marks - ")))
round(sum(marks)/len(marks), 2)
```
Out[208…
```
70.0
```
Ex. Write the program to multiply the two lists

In [209…
```
num_list_1 = [1, 2, 3, 4]
num_list_2 = [0, 5, 2, 1]
```

Utility Functions

zip(`lst1` , `lst2` , ...) - returns a zip object, which is an sequence of tuples where the first item in each passed sequence is paired together, and then the second item in each passed sequence are paired together etc. If the passed sequences have different lengths, the iterator with the least items decides the length of the new iterator.

In [211…
```
list(zip(num_list_1, num_list_2))
```
Out[211…
```
[(1, 0), (2, 5), (3, 2), (4, 1)]
```
In [212…
```
for i in zip(num_list_1, num_list_2) :
    print(i)
```
```
(1, 0)
(2, 5)
(3, 2)
(4, 1)
```
In [213…
```
for i, j in zip(num_list_1, num_list_2) :
    print(i * j)
```
```
0
10
6
4
```
enumerate(`seq-obj` , `start=0`) - adds counter to an iterable and returns a sequence of tuples (the enumerate object).

In [215…
```
enumerate(friends)
```
Out[215…
```
<enumerate at 0x1f461a23880>
```
In [216…
```
list(enumerate(friends))
```
Out[216…
```
[(0, 'Rachel'), (1, 'Phoebe'), (2, 'Jane'), (3, 'Amy')]
```
In [217…
```
list(enumerate(friends, start = 101))
```
Out[217…
```
[(101, 'Rachel'), (102, 'Phoebe'), (103, 'Jane'), (104, 'Amy')]
```