Python3 🐍

- Easy to Learn: Python has a simple syntax that resembles the English language.
- Powerful: It provides efficient high-level data structures and supports object-oriented programming.
- Dynamic and Interpreted: Python's dynamic typing and interpreted nature make it suitable for rapid prototyping and scripting across various platforms.

Python Syntax 🧑‍💻

- Python Syntax Compared to Other Programming Languages Readability Focused:
- Python was designed with readability in mind, with similarities to the English language and mathematical influence. New Line Delimiters:
- Python uses new lines to complete a command, instead of semicolons (;) or parentheses ({}). Indentation for Scope:
- Python relies on indentation (whitespace) to define the scope of loops, functions, and classes. Other programming languages often use curly brackets {} for this purpose.

In [ ]:

```python
if 5 > 2:
  print("Five is greater than two!")
```

```
Five is greater than two!
```

Comments 📌

- Python supports comments to add in-code documentation.
- Comments start with a #. Everything after the # on the same line is ignored by Python.

In [ ]:

```python
# This is a comment
print("rahul i love you!!")
```

```
rahul i love you!!
```

Docstrings

1. python also has extended documentation capability called docstrings
2. docstrings can b one line or multiple lines Docstrings are also comments
3. uses triple quotes at the beginning and end of the docstring:

In [ ]:

```python
# Single-line docstring
"""This is a single-line docstring."""
print("rahul marry me ")
```

```
rahul marry me
```

In [ ]:

```
# Multi-line docstring
"""
This is a multiline
docstring.
"""
print("can't live without you rahul")
```

```
can't live without you rahul
```

variables 🌱

1. python is completly object oriented and not "satically typed"
2. python has no command for declaring a variable
3. a variable is created the moment u first assign the a value to it
4. a variable can have short name like x and y or more descriptive name

rules for python variables:

- A varible name must start with a letter or underscore char.
- a varible name cannot start with a number.
- a variable name can only contain alpha-numeric char and underscores (a-z,0-9 and _).
- a varible names are case sensitive (age, Age, AGE are the different varaibles).

In [ ]:

```
x = 2
y = 'rahul chirra'
z = 99.9  #float
print(x)
print(y)
print(z)
```

```
2
rahul chirra
99.9
```

In [ ]:

```
# Test variables
integer_variable = 5
string_variable = 'rahul'

print(integer_variable)
print(string_variable)
```

```
5
rahul
```

In [ ]:

```
variable_with_changed_type = 4  # Initially of type int
variable_with_changed_type = 'batman'  # Now of type str

print(variable_with_changed_type)
```

```
batman
```

Operators 🛡️

Arithmetic operatios

- aritthetic operations are used with numeric values to perform common mathematical operations
- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)
- Exponentiation (**)
- Floor Division (//)

In [ ]:

```
# Addition
print(5 + 3)
```

```
8
```

In [ ]:

```
# Subtraction
print(5 - 3)
```

```
2
```

In [ ]:

```
# Multiplication
print(5 * 3)
print(isinstance(5 * 3, int))
```

```
15
True
```

In [ ]:

```
# Division
print(5 / 3)
print(8 / 4)
print(isinstance(5 / 3, float))
print(isinstance(8 / 4, float))
```

```
1.6666666666666667
2.0
True
True
```

In [ ]:

```
# Modulus
print(5 % 3)
```

```
2
```

In [ ]:
```
# Exponentiation
print(5 ** 3)
print(2 ** 3)
print(2 ** 4)
print(2 ** 5)
print(isinstance(5 ** 3, int))
```

```
125
8
16
32
True
```

In [ ]:
```
# Floor division
print(5 // 3)
print(6 // 3)
print(7 // 3)
print(9 // 3)
print(isinstance(5 // 3, int))
```

```
1
2
2
3
True
```

Bitwise Operators: ♻️

- AND (&): Sets each bit to 1 if both bits are 1.
- OR (|): Sets each bit to 1 if one of the two bits is 1.
- NOT (~): Inverts all the bits.
- XOR (^): Sets each bit to 1 if only one of the two bits is 1.
- Signed Right Shift (>>): Shifts bits to the right, filling with the sign bit.
- **Left** Shift (<<): Shifts bits to the left, filling with zeros.

In [ ]:
```
# Bitwise operators in Python
# AND
# Sets each bit to 1 if both bits are 1.
# Example: 5 = 0b0101, 3 = 0b0011
print(5 & 3)
```

```
1
```

In [ ]:
```
# OR
# Sets each bit to 1 if one of two bits is 1.
# Example: 5 = 0b0101, 3 = 0b0011
print(5 | 3)
```

```
7
```

In [ ]:

```python
# NOT
# Inverts all the bits.
print(~5)
```

```
-6
```

In [ ]:

```python
# XOR
# Sets each bit to 1 if only one of two bits is 1.
# Example: 5 = 0b0101, 3 = 0b0011
print(5 ^ 3)
```

```
6
```

In [ ]:

```python
# Signed right shift
# Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.
# Example: 5 = 0b0101
print(5 >> 1)
print(5 >> 2)
```

```
2
1
```

In [ ]:

```python
# Zero fill left shift
# Shift left by pushing zeros in from the right and let the leftmost bits fall off.
# Example: 5 = 0b0101
print(5 << 1)
print(5 << 2)
```

```
10
20
```

Assignment operators ❎

- Assignment operators are used to assign values to variables

In [ ]:

```python
# Assignment: =
number = 5
print(number)
```

```
5
```

In [ ]:

```python
# Multiple assignment
# The variables first_variable and second_variable simultaneously get the new values 0 and 1.
first_variable, second_variable = 0, 1
print(first_variable)
print(second_variable)
```

```
0
1
```

In [ ]:

```
# Switching variable values using multiple assignment
first_variable, second_variable = second_variable, first_variable
print(first_variable)
print(second_variable)
```

```
1
0
```

- Augmented assignment operators

In [ ]:

```
# Assignment: +=
number = 5
number += 3
print(number)
```

```
8
```

In [ ]:

```
# Assignment: -=
number = 5
number -= 3
print(number)
```

```
2
```

In [ ]:

```
# Assignment: *=
number = 5
number *= 3
print(number)
```

```
15
```

In [ ]:

```
# Assignment: /=
number = 8
number /= 4
print(number)
```

```
2.0
```

In [ ]:

```
# Assignment: %=
number = 8
number %= 3
print(number)

number = 5
number %= 3
print(number)
```

```
2
2
```

In [ ]:

```
# Assignment: //=
number = 5
number //= 3
print(number)
```

```
1
```

In [ ]:

```
# Assignment: **=
number = 5
number **= 3
print(number)
```

```
125
```

In [ ]:

```
# Assignment: &=
number = 5  # 0b0101
number &= 3  # 0b0011
print(number)  # 0b0001
```

```
1
```

In [ ]:

```
# Assignment: |=
number = 5  # 0b0101
number |= 3  # 0b0011
print(number)  # 0b0111
```

```
7
```

In [ ]:

```
# Assignment: ^=
number = 5  # 0b0101
number ^= 3  # 0b0011
print(number)  # 0b0110
```

```
6
```

In [ ]:

```
# Assignment: >>=
number = 5
number >>= 3
print(number)
```

```
0
```

In [ ]:

```
# Assignment: <<=
number = 5
number <<= 3
print(number)
```

```
40
```

comparison operators 📺

- Comparison operators are used to compare two values

```
In [ ]: # Equal
        number = 5
        print(number == 5)
```

```
True
```

```
In [ ]: # Not equal
        number = 5
        print(number != 3)
```

```
True
```

```
In [ ]: # Greater than
        number = 5
        print(number > 3)
```

```
True
```

```
In [ ]: # Less than
        number = 5
        print(number < 8)
```

```
True
```

```
In [ ]: # Greater than or equal to
        number = 5
        print(number >= 5)
        print(number >= 4)
```

```
True
True
```

```
In [ ]: # Less than or equal to
        number = 5
        print(number <= 5)
        print(number <= 6)
```

```
True
True
```

Logical operators ⌚

- Logical operators are used to combine conditional statements
- and
- or
- not

In [ ]:
```
# Let's work with these numbers to illustrate logic operators.
first_number = 5
second_number = 10
```

In [ ]:
```
# and
# Returns True if both statements are true
print(first_number > 0 and second_number < 20)
```

```
False
```

In [ ]:
```
# or
# Returns True if one of the statements is true
print(first_number > 5 or second_number < 20)
```

```
True
```

In [ ]:
```
# not
# Reverse the result, returns False if the result is true
print(not first_number == second_number)
print(first_number != second_number)
```

```
True
True
```

Identity operators 🦴

- Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location

In [ ]:
```
# Identity operators

# Lists for demonstration
first_fruits_list = ["apple", "banana"]
second_fruits_list = ["apple", "banana"]
third_fruits_list = first_fruits_list
```

In [ ]:
```
# is
# Returns true if both variables are the same object
print(first_fruits_list is third_fruits_list)
```

```
True
```

In [ ]:
```
# is not
# Returns true if both variables are not the same object
print(first_fruits_list is not second_fruits_list)
```

```
True
```

In [ ]:
```
# Demonstrating the difference between "is" and "=="
# "==" checks for equality of content
print(first_fruits_list == second_fruits_list)
```

```
True
```

Membership operators 🍃

- Membership operators are used to test if a sequence is present in an object

In [ ]:
```
# Membership operators

# Fruit list for demonstration
fruit_list = ["apple", "banana"]
```

In [ ]:
```
# in
# Returns True if a sequence with the specified value is present in the object
print("banana" in fruit_list)
```

```
True
```

In [ ]:
```
# not in
# Returns True if a sequence with the specified value is not present in the object
print("pineapple" not in fruit_list)
```

```
True
```

Data Types in python 📚 🎧 ☕

Numbers.

- There are three numeric types in Python:
- int (e.g. 2, 4, 20)
    - bool (e.g. False and True, acting like 0 and 1)
- float (e.g. 5.0, 1.6)
- complex (e.g. 5+6j, 4-3j)

In [ ]:

```
# Integer type
positive_integer = 1
negative_integer = -3255522
big_integer = 35656222554887711

print(isinstance(positive_integer, int))
print(isinstance(negative_integer, int))
print(isinstance(big_integer, int))
```

```
True
True
True
```

In [ ]:

```
# Boolean
true_boolean = True
false_boolean = False

print(true_boolean)
print(not false_boolean)

print(isinstance(true_boolean, bool))
print(isinstance(false_boolean, bool))
```

```
True
True
True
True
```

In [ ]:

```
# Casting boolean to string.
print(f"String representation of True: {str(true_boolean)}")
print(f"String representation of False: {str(false_boolean)}")
print("Boolean tests passed.\n")
```

```
String representation of True: True
String representation of False: False
Boolean tests passed.
```

In [ ]:

```
# Casting boolean to string
print(str(true_boolean))
print(str(false_boolean))
```

```
True
False
```

In [ ]:
```
# Float type
float_number = 7.0
float_number_via_function = float(7)
float_negative = -35.59

print(float_number == float_number_via_function)
print(isinstance(float_number, float))
print(isinstance(float_number_via_function, float))
print(isinstance(float_negative, float))
```

```
True
True
True
True
```

In [ ]:
```
# Scientific notation with float
float_with_small_e = 35e3
float_with_big_e = 12E4

print(float_with_small_e)
print(float_with_big_e)
print(isinstance(12E4, float))
print(isinstance(-87.7e100, float))
```

```
35000.0
120000.0
True
True
```

In [ ]:
```
# Complex type
complex_number_1 = 5 + 6j
complex_number_2 = 3 - 2j

print(isinstance(complex_number_1, complex))
print(isinstance(complex_number_2, complex))
print(complex_number_1 * complex_number_2)
```

```
True
True
(27+8j)
```

In [ ]:
```
# Basic operations
print(2 + 4)  # Addition
print(2 * 4)  # Multiplication
print(12 / 3)  # Division
print(12 / 5)
print(17 / 3)
```

```
6
8
4.0
2.4
5.666666666666667
```

In [ ]:
```
# Modulo
print(12 % 3)
print(13 % 3)
```

```
0
1
```

In [ ]:
```
# Floor division
print(17 // 3)
```

```
5
```

In [ ]:
```
# Exponentiation
print(5 ** 2)
print(2 ** 7)
```

```
25
128
```

In [ ]:
```
# Mixed type operation
print(4 * 3.75 - 1)
```

```
14.0
```

Strings in python ♡📷

- A string is a sequence of characters enclosed in either single (') or double (") quotes. Python strings are immutable, meaning they cannot be modified after creation

In [ ]:
```
# String type
name_1 = "rahul"
name_2 = 'chirra'
print(name_1 == name_2)
print(isinstance(name_1, str))
print(isinstance(name_2, str))
```

```
False
True
True
```

In [ ]:
```
# Escaping quotes
single_quote_string = 'doesn\'t'
double_quote_string = "doesn't"
print(single_quote_string == double_quote_string)
```

```
True
```

In [ ]:

```
# Newline character
multiline_string = 'First line.\nSecond line.'
print(multiline_string)
```

```
First line.
Second line.
```

In [ ]:

```
# Indexing
word = 'Python'
print(word[0])  # First character
print(word[5])  # Fifth character
print(word[-1])  # Last character
print(word[-2])  # Second-last character
print(word[-6])  # Sixth from the end
print(isinstance(word[0], str))
```

```
P
n
n
o
P
True
```

In [ ]:

```
# Slicing
print(word[0:2])  # Characters from position 0 (included) to 2 (excluded)
print(word[2:5])  # Characters from position 2 (included) to 5 (excluded)
print(word[:2] + word[2:])
print(word[:4] + word[4:])
print(word[:2])  # Characters from the beginning to position 2 (excluded)
print(word[4:])  # Characters from position 4 (included) to the end
print(word[-2:])  # Characters from the second-last to the end
```

```
Py
tho
Python
Python
Py
on
on
```

In [ ]:

```
# Out of range slice indexes
print(word[4:42])
print(word[42:])
```

```
on
```

In [ ]:

```
# Immutability
print('J' + word[1:])
print(word[:2] + 'py')
```

```
Jython
Pypy
```

In [ ]:

```python
# Length
characters = 'supercalifragilisticexpialidocious'
print(len(characters))
```

```
34
```

In [ ]:

```python
# Multiline strings
multi_line_string = '''\
        First line
        Second line
    '''
print(multi_line_string)
```

```
        First line
        Second line

```

In [ ]:

```python
# String operators
print(3 * 'un' + 'ium')
python = 'Py' 'thon'
print(python)
text = (
    'Put several strings within parentheses '
    'to have them joined together.'
)
print(text)
prefix = 'Py'
print(prefix + 'thon')
```

```
unununium
Python
Put several strings within parentheses to have them joined together.
Python
```

In [ ]:

```python
# String methods
hello_world_string = "Hello, World!"
string_with_whitespaces = " Hello, World! "
print(string_with_whitespaces.strip())
print(len(hello_world_string))
print(hello_world_string.lower())
print(hello_world_string.upper())
print(hello_world_string.replace('H', 'J'))
print(hello_world_string.split(','))
```

```
Hello, World!
13
hello, world!
HELLO, WORLD!
Jello, World!
['Hello', ' World!']
```

In [ ]:

```
print('low letter at the beginning'.capitalize())
print('low letter at the beginning'.count('t'))
print('Hello, welcome to my world'.find('welcome'))
print('Welcome to my world'.title())
print('I like bananas'.replace('bananas', 'apples'))
```

```
Low letter at the beginning
4
7
Welcome To My World
I like apples
```

In [ ]:

```
my_tuple = ('John', 'Peter', 'Vicky')
print(', '.join(my_tuple))
```

```
John, Peter, Vicky
```

In [ ]:

```
print('ABC'.isupper())
print('AbC'.isupper())
print('CompanyX'.isalpha())
print('Company 23'.isalpha())
print('1234'.isdecimal())
print('a21453'.isdecimal())
```

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

In [ ]:

```
# String formatting
year = 2018
event = 'conference'
print(f'Results of the {year} {event}')
```

```
Results of the 2018 conference
```

In [ ]:

```
yes_votes = 42_572_654
no_votes = 43_132_495
percentage = yes_votes / (yes_votes + no_votes)
print('{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage))
```

```
 42572654 YES votes  49.67%
```

In [ ]:

```
greeting = 'Hello, world.'
first_num = 10 * 3.25
second_num = 200 * 200
print(str(greeting))
print(repr(greeting))
print(str(1/7))
print(repr((first_num, second_num, ('spam', 'eggs'))))
```

```
Hello, world.
'Hello, world.'
0.14285714285714285
(32.5, 40000, ('spam', 'eggs'))
```

In [ ]:

```
pi_value = 3.14159
print(f'The value of pi is {pi_value:.3f}.')
```

```
The value of pi is 3.142.
```

In [ ]:

```
table_data = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
for name, phone in table_data.items():
    print(f'{name:7}==>{phone:7d}')

print('We are {} who say "{}!"'.format('knights', 'Ni'))
print('{0} and {1}'.format('spam', 'eggs'))
print('{1} and {0}'.format('spam', 'eggs'))
```

```
Sjoerd ==>   4127
Jack   ==>   4098
Dcab   ==>   7678
We are knights who say "Ni!"
spam and eggs
eggs and spam
```

In [ ]:

```
formatted_string = 'This {food} is {adjective}.'.format(
    food='spam',
    adjective='absolutely horrible'
)
print(formatted_string)
```

```
This spam is absolutely horrible.
```

In [ ]:

```
formatted_string = 'The story of {0}, {1}, and {other}.'.format(
    'Bill',
    'Manfred',
    other='Georg'
)
print(formatted_string)
```

```
The story of Bill, Manfred, and Georg.
```

In [ ]:

```
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
formatted_string = 'Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:
d}'.format(table)
print(formatted_string)
```

```
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

In [ ]:

```
formatted_string = 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format
(**table)
print(formatted_string)
```

```
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

List in Python 🐬

- A list in Python is a versatile and widely used data structure that stores a collection of items. These items can be of any data type (e.g., integers, strings, floats, or even other lists

In [ ]:

```
# List type
squares = [1, 4, 9, 16, 25]
print(isinstance(squares, list))
```

```
True
```

In [ ]:

```
print(squares[0])  # Indexing returns the item
print(squares[-1])
print(squares[-3:])  # Slicing returns a new list
print(squares[:])
print(squares + [36, 49, 64, 81, 100])
```

```
1
25
[9, 16, 25]
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [ ]:

```
# Lists are mutable
cubes = [1, 8, 27, 65, 125]
cubes[3] = 64  # Replace the wrong value
print(cubes)
cubes.append(216)
cubes.append(7 ** 3)
print(cubes)
```

```
[1, 8, 27, 64, 125]
[1, 8, 27, 64, 125, 216, 343]
```

In [ ]:

```
# Assignment to slices
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters[2:5] = ['C', 'D', 'E']
print(letters)
letters[2:5] = []
print(letters)
letters[:] = []
print(letters)
```

```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
['a', 'b', 'f', 'g']
[]
```

In [ ]:

```
# Built-in length function
letters = ['a', 'b', 'c', 'd']
print(len(letters))
```

```
4
```

In [ ]:

```
# Nested lists
list_of_chars = ['a', 'b', 'c']
list_of_numbers = [1, 2, 3]
mixed_list = [list_of_chars, list_of_numbers]
print(mixed_list)
print(mixed_list[0])
print(mixed_list[0][1])
```

```
[['a', 'b', 'c'], [1, 2, 3]]
['a', 'b', 'c']
b
```

In [ ]:

```
# List methods
fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
fruits.append('grape')
print(fruits)
fruits.remove('grape')
print(fruits)
```

```
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana', 'grape']
['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
```

In [ ]:

```
fruits.insert(0, 'grape')
print(fruits)
print(fruits.index('grape'))
print(fruits.index('orange'))
print(fruits.index('banana'))
print(fruits.index('banana', 5))
print(fruits.count('tangerine'))
print(fruits.count('banana'))
```

```
['grape', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
0
1
4
7
0
2
```

In [ ]:

```
fruits_copy = fruits.copy()
print(fruits_copy)
fruits_copy.reverse()
print(fruits_copy)
fruits_copy.sort()
print(fruits_copy)
print(fruits.pop())
print(fruits)
fruits.clear()
print(fruits)
```

```
['grape', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
banana
['grape', 'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple']
[]
```

In [ ]:

```
# Del statement
numbers = [-1, 1, 66.25, 333, 333, 1234.5]
del numbers[0]
print(numbers)
del numbers[2:4]
print(numbers)
del numbers[:]
print(numbers)
```

```
[1, 66.25, 333, 333, 1234.5]
[1, 66.25, 1234.5]
[]
```

In [ ]:

```
# List comprehensions
squares = [x ** 2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [ ]:

```
combinations = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
print(combinations)
```

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

In [ ]:

```
vector = [-4, -2, 0, 2, 4]
print([x * 2 for x in vector])
print([x for x in vector if x >= 0])
print([abs(x) for x in vector])
```

```
[-8, -4, 0, 4, 8]
[0, 2, 4]
[4, 2, 0, 2, 4]
```

In [ ]:

```
fresh_fruit = ['  banana', '  loganberry ', 'passion fruit  ']
print([fruit.strip() for fruit in fresh_fruit])
print([(x, x ** 2) for x in range(6)])
vector = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print([num for elem in vector for num in elem])
```

```
['banana', 'loganberry', 'passion fruit']
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [ ]:

```
# Nested list comprehensions
matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
]
transposed_matrix = [[row[i] for row in matrix] for i in range(4)]
print(transposed_matrix)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In [ ]:

```
print(list(zip(*matrix)))
```

```
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Tuples in Python

In [ ]:

```
# Basic tuple
fruits_tuple = ("apple", "banana", "cherry")
print(isinstance(fruits_tuple, tuple))
print(fruits_tuple[0])
print(fruits_tuple[1])
print(fruits_tuple[2])
```

```
True
apple
banana
cherry
```

In [ ]:

```
# You cannot change values in a tuple
try:
    fruits_tuple[0] = "pineapple"
except TypeError as e:
    print(f"Error: {e}")
```

```
Error: 'tuple' object does not support item assignment
```

In [ ]:

```
# Using the tuple() constructor
fruits_tuple_via_constructor = tuple(("apple", "banana", "cherry"))
print(isinstance(fruits_tuple_via_constructor, tuple))
print(len(fruits_tuple_via_constructor))
```

```
True
3
```

In [ ]:

```
# Omitting brackets when initializing tuples
another_tuple = 12345, 54321, 'hello!'
print(another_tuple)
```

```
(12345, 54321, 'hello!')
```

In [ ]:

```
# Nested tuples
nested_tuple = another_tuple, (1, 2, 3, 4, 5)
print(nested_tuple)
```

```
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

In [ ]:

```
# Empty and singleton tuples
empty_tuple = ()
print(len(empty_tuple))
```

```
0
```

In [ ]:

```
singleton_tuple = 'hello',
print(len(singleton_tuple))
print(singleton_tuple)
```

```
1
('hello',)
```

In [ ]:

```
# Tuple packing and unpacking
packed_tuple = 12345, 54321, 'hello!'
first_tuple_number, second_tuple_number, third_tuple_string = packed_tuple
print(first_tuple_number)
print(second_tuple_number)
print(third_tuple_string)
```

```
12345
54321
hello!
```

In [ ]:

```
# Swapping using tuples
first_number = 123
second_number = 456
first_number, second_number = second_number, first_number
print(first_number)
print(second_number)
```

```
456
123
```

Sets in python 💲

- A set is a collection which is unordered and unindexed.
- In Python sets are written with curly brackets.
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

In [ ]:

```
# Sets
fruits_set = {"apple", "banana", "cherry"}
print(isinstance(fruits_set, set))
```

```
True
```

In [ ]:

```
# Use the set() constructor to make a set
fruits_set_via_constructor = set(("apple", "banana", "cherry"))
print(isinstance(fruits_set_via_constructor, set))
```

```
True
```

In [ ]:

```
# Set methods
fruits_set = {"apple", "banana", "cherry"}
```

In [ ]:

```
# Check if the item is in the set
print("apple" in fruits_set)
print("pineapple" not in fruits_set)
```

```
True
True
```

In [ ]:

```
# Return the number of items
print(len(fruits_set))
```

```
3
```

In [ ]:

```
# Add an item
fruits_set.add("pineapple")
print("pineapple" in fruits_set)
print(len(fruits_set))
```

```
True
4
```

In [ ]:

```
# Remove an item
fruits_set.remove("pineapple")
print("pineapple" not in fruits_set)
print(len(fruits_set))
```

```
True
3
```

In [ ]:

```
#Demonstrate set operations on unique letters from two words
first_char_set = set('abracadabra')
second_char_set = set('alacazam')

print(first_char_set)
print(second_char_set)
```

```
{'d', 'r', 'a', 'c', 'b'}
{'z', 'a', 'm', 'l', 'c'}
```

In [ ]:

```
# Letters in the first word but not in the second
print(first_char_set - second_char_set)
```

```
{'d', 'r', 'b'}
```

In [ ]:

```
# Letters in the first or second word or both
print(first_char_set | second_char_set)
```

```
{'d', 'r', 'z', 'a', 'm', 'l', 'c', 'b'}
```

In [ ]:

```
# Common letters in both words
print(first_char_set & second_char_set)
```

```
{'a', 'c'}
```

In [ ]:

```
# Letters in the first or second word but not both
print(first_char_set ^ second_char_set)
```

```
{'m', 'l', 'd', 'r', 'b', 'z'}
```

In [ ]:

```
# Set comprehensions
word = {char for char in 'abracadabra' if char not in 'abc'}
print(word)
```

```
{'d', 'r'}
```

Dictionary in python

- A dictionary is a collection which is unordered, changeable, and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

In [ ]:

```
# Dictionary
fruits_dictionary = {
    'cherry': 'red',
    'apple': 'green',
    'banana': 'yellow',
}

print(isinstance(fruits_dictionary, dict))
```

```
True
```

In [ ]:

```
# Access dictionary elements by keys
print(fruits_dictionary['apple'])
print(fruits_dictionary['banana'])
print(fruits_dictionary['cherry'])
```

```
green
yellow
red
```

In [ ]:

```
# Check whether a key exists in the dictionary
print('apple' in fruits_dictionary)
print('pineapple' not in fruits_dictionary)
```

```
True
True
```

In [ ]:

```
# Change the apple color to "red"
fruits_dictionary['apple'] = 'red'
fruits_dictionary
```

Out[ ]:

```
{'cherry': 'red', 'apple': 'red', 'banana': 'yellow'}
```

In [ ]:

```
# Add a new key/value pair to the dictionary
fruits_dictionary['pineapple'] = 'yellow'
print(fruits_dictionary['pineapple'])
```

```
yellow
```

In [ ]:

```
# Retrieve keys in insertion order
print(list(fruits_dictionary))
print(sorted(fruits_dictionary))
```

```
['cherry', 'apple', 'banana', 'pineapple']
['apple', 'banana', 'cherry', 'pineapple']
```

In [ ]:

```
# Delete a key:value pair
del fruits_dictionary['pineapple']
print(list(fruits_dictionary))
```

```
['cherry', 'apple', 'banana']
```

In [ ]:

```
# Build a dictionary using the dict() constructor
dictionary_via_constructor = dict([('sape', 4139), ('guido', 4127), ('jack', 4
098)])

print(dictionary_via_constructor['sape'])
print(dictionary_via_constructor['guido'])
print(dictionary_via_constructor['jack'])
```

```
4139
4127
4098
```

In [ ]:

```
# Dictionary comprehensions
dictionary_via_expression = {x: x**2 for x in (2, 4, 6)}

print(dictionary_via_expression[2])
print(dictionary_via_expression[4])
print(dictionary_via_expression[6])
```

```
4
16
36
```

In [ ]:

```
# Using keyword arguments for simple string keys
dictionary_for_string_keys = dict(sape=4139, guido=4127, jack=4098)

print(dictionary_for_string_keys['sape'])
print(dictionary_for_string_keys['guido'])
print(dictionary_for_string_keys['jack'])
```

```
4139
4127
4098
```

Type casting.

- int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number) literal, or a string literal (providing the string represents a whole number)
- float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
- str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

In [ ]:

```
# Type casting to integer
print(int(1))
print(int(2.8))
print(int('3'))
```

```
1
2
3
```

In [ ]:

```
# Type casting to float
print(float(1))
print(float(2.8))
print(float("3"))
print(float("4.2"))
```

```
1.0
2.8
3.0
4.2
```

In [ ]:

```
# Type casting to string
print(str("s1"))
print(str(2))
print(str(3.0))
```

```
s1
2
3.0
```

Control Flow 💻

IF statement

- An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages

In [ ]:

```
# IF statement

number = 15
conclusion = ''

if number < 0:
    conclusion = 'Number is less than zero'
elif number == 0:
    conclusion = 'Number equals to zero'
elif number < 1:
    conclusion = 'Number is greater than zero but less than one'
else:
    conclusion = 'Number bigger than or equal to one'

print(conclusion)
```

```
Number bigger than or equal to one
```

FOR statement

- The for statement in Python iterates over the items of any sequence (a list or a string), in the order they appear in the sequence

In [ ]:

```python
# Measure some strings
words = ['cat', 'window', 'defenestrate']
words_length = 0

for word in words:
    words_length += len(word)

print(words_length)  # Output: Total length of words
```

```
21
```

In [ ]:

```python
# Modifying a sequence during iteration using a slice copy
for word in words[:]:
    if len(word) > 6:
        words.insert(0, word)

print(words)  # Output: Updated list with inserted items
```

```
['defenestrate', 'cat', 'window', 'defenestrate']
```

In [ ]:

```python
# Iterating over a sequence of numbers with range()
iterated_numbers = []
for number in range(5):
    iterated_numbers.append(number)

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

```
[0, 1, 2, 3, 4]
```

In [ ]:

```python
# Iterating over indices using range() and len()
words = ['Mary', 'had', 'a', 'little', 'lamb']
concatenated_string = ''
for word_index in range(len(words)):
    concatenated_string += words[word_index] + ' '

print(concatenated_string)
```

```
Mary had a little lamb
```

In [ ]:

```python
# Using enumerate() for indices and values
concatenated_string = ''
for word_index, word in enumerate(words):
    concatenated_string += word + ' '

print(concatenated_string)
```

```
Mary had a little lamb
```

In [ ]:

```
# Looping through dictionaries with items()
knights_names = []
knights_properties = []
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for key, value in knights.items():
    knights_names.append(key)
    knights_properties.append(value)

print(knights_names)
print(knights_properties)
```

```
['gallahad', 'robin']
['the pure', 'the brave']
```

In [ ]:

```
# Enumerating indices and values
indices = []
values = []
for index, value in enumerate(['tic', 'tac', 'toe']):
    indices.append(index)
    values.append(value)

print(indices)
print(values)
```

```
[0, 1, 2]
['tic', 'tac', 'toe']
```

In [ ]:

```
# Looping over two sequences with zip()
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
combinations = []
for question, answer in zip(questions, answers):
    combinations.append(f'What is your {question}?  It is {answer}.')

print(combinations)
```

```
['What is your name?  It is lancelot.', 'What is your quest?  It is the holy
grail.', 'What is your favorite color?  It is blue.']
```

In [ ]:

```
# Range function
print(list(range(5)))
print(list(range(5, 10)))
print(list(range(0, 10, 3)))
print(list(range(-10, -100, -30)))
```

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
[0, 3, 6, 9]
[-10, -40, -70]
```

WHILE statement

- The while loop executes as long as the condition remains true.

In [ ]:
```
# WHILE statement

# Raising a number to a certain power using a while loop
number = 2
power = 5
result = 1

while power > 0:
    result *= number
    power -= 1

# Output the result of 2^5
print(result)
```

```
32
```

TRY statement

- "try" statement is used for exception handling. When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement.
- The "try" block lets you test a block of code for errors.
- The "except" block lets you handle the error.
- The "else" block lets you execute the code if no errors were raised.
- The "finally" block lets you execute code, regardless of the result of the try- and except blocks

In [ ]:
```
# Example 1: Catching an exception
exception_has_been_caught = False

try:
    print(not_existing_variable)  # This will raise a NameError
except NameError:
    exception_has_been_caught = True

print(exception_has_been_caught)
```

```
True
```

In [ ]:

```
# Example 2: Handling a specific exception with a message
exception_message = ''

try:
    print(not_existing_variable)  # This will raise a NameError
except NameError:
    exception_message = 'Variable is not defined'

print(exception_message)
```

```
Variable is not defined
```

In [ ]:

```
# Example 3: Using else for code that runs when no exceptions occur
message = ''

try:
    message += 'Success.'
except NameError:
    message += 'Something went wrong.'
else:
    message += 'Nothing went wrong.'

print(message)
```

```
Success.Nothing went wrong.
```

In [ ]:

```
# Example 4: Using finally to execute code regardless of exceptions
message = ''

try:
    print(not_existing_variable)  # This will raise a NameError
except NameError:
    message += 'Something went wrong.'
finally:
    message += 'The "try except" is finished.'

print(message)
```

```
Something went wrong.The "try except" is finished.
```

BREAK statement

- The break statement, like in C, breaks out of the innermost enclosing "for" or "while" loop.

In [ ]:

```
# Terminate the loop if the number we need is found
number_to_be_found = 42
number_of_iterations = 0

for number in range(100):
    if number == number_to_be_found:
        # Break here and don't continue the loop
        break
    else:
        number_of_iterations += 1

# Print the number of iterations before the loop was terminated
print(number_of_iterations)
```

```
42
```

CONTINUE statement

- The continue statement is borrowed from C, continues with the next iteration of the loop.

In [ ]:

```
# CONTINUE statement in FOR loop

# Lists to separate even and odd numbers
even_numbers = []
rest_of_the_numbers = []

for number in range(0, 10):
    # Check if the number is even
    if number % 2 == 0:
        even_numbers.append(number)
        # Skip the rest of the loop for this iteration
        continue
    rest_of_the_numbers.append(number)

# Output the results
print(even_numbers)
print(rest_of_the_numbers)
```

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

Functions in python

- The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented

In [ ]:

```
# Fibonacci function definition
def fibonacci_function_example(number_limit):
    """Generate a Fibonacci series up to number_limit."""
    fibonacci_list = []
    previous_number, current_number = 0, 1
    while previous_number < number_limit:
        fibonacci_list.append(previous_number)
        previous_number, current_number = current_number, previous_number + cu
rrent_number
    return fibonacci_list

# Call the Fibonacci function and print the result
print(fibonacci_function_example(300))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

In [ ]:

```
# Assign the Fibonacci function to another variable and call it
fibonacci_function_clone = fibonacci_function_example
print(fibonacci_function_clone(300))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

In [ ]:

```
# Functions as first-class objects
def greet(name):
    return 'Hello, ' + name

greet_someone = greet
print(greet_someone('rahul'))
```

```
Hello, rahul
```

In [ ]:

```
# Nested function example
def greet_again(name):
    def get_message():
        return 'Hello, '
    return get_message() + name

print(greet_again('veerasankar reddy'))
```

```
Hello, veerasankar reddy
```

In [ ]:

```
# Passing functions as parameters
def greet_one_more(name):
    return 'Hello, ' + name

def call_func(func):
    other_name = 'samba'
    return func(other_name)

print(call_func(greet_one_more))
```

```
Hello, samba
```

In [ ]:

```python
# Functions returning other functions
def compose_greet_func():
    def get_message():
        return 'Hello there!'
    return get_message

greet_function = compose_greet_func()
print(greet_function())
```

```
Hello there!
```

In [ ]:

```python
# Closure example
def compose_greet_func_with_closure(name):
    def get_message():
        return 'Hello there, ' + name + '!'
    return get_message

greet_with_closure = compose_greet_func_with_closure('rajamouli')
print(greet_with_closure())
```

```
Hello there, rajamouli!
```

Scopes and Namespaces

A NAMESPACE is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name.

In [ ]:

```python
# Global variable
test_variable = 'initial global value'

# Example demonstrating scopes and namespaces
def test_scopes():
    # Local variable in the enclosing function scope
    test_variable = 'initial value inside test function'

    def do_local():
        # Create a variable only accessible inside this function
        test_variable = 'local value'
        return test_variable

    def do_nonlocal():
        # Modify the variable from the enclosing scope
        nonlocal test_variable
        test_variable = 'nonlocal value'
        return test_variable

    def do_global():
        # Modify the global variable
        global test_variable
        test_variable = 'global value'
        return test_variable

    # Accessing the local variable
    print(test_variable)
    # Local assignment does not change the enclosing variable
    do_local()
    print(test_variable)

    # Nonlocal assignment modifies the variable in the enclosing scope
    do_nonlocal()
    print(test_variable)

    # Global assignment changes the global variable
    do_global()
    print(test_variable)

# Run the test scopes
test_scopes()

# Checking global variable access
print(test_variable)
```

```
initial value inside test function
initial value inside test function
nonlocal value
nonlocal value
global value
```

Default Argument Values

- The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow.

In [ ]:

```
# Function definition with default argument
def power_of(number, power=2):
    return number ** power

# Demonstration of the function

# Call the function with the default argument
result = power_of(3)
print(result)
```

```
9
```

In [ ]:

```
# Call the function with an overridden second argument
result = power_of(3, 2)
print(result)
```

```
9
```

In [ ]:

```
result = power_of(3, 3)
print(result)
```

```
27
```

Keyword Arguments

- Functions can be called using keyword arguments of the form kwarg=value.

In [ ]:

```python
def parrot(voltage, state='a stiff', action='voom', parrot_type='Norwegian Blue'):
    """Example of multi-argument function

    This function accepts one required argument (voltage) and three optional arguments
    (state, action, and type).
    """
    message = 'This parrot wouldn\'t ' + action + ' '
    message += 'if you put ' + str(voltage) + ' volts through it. '
    message += 'Lovely plumage, the ' + parrot_type + '. '
    message += 'It\'s ' + state + '!'
    return message

print(parrot(1000))
print(parrot(voltage=1000))

print(parrot(voltage=1000000, action='VOOOOOM'))
print(parrot(action='VOOOOOM', voltage=1000000))

print(parrot(1000000, 'bereft of life', 'jump'))

print(parrot(1000, state='pushing up the daisies'))

try:
    print(parrot())
except Exception as e:
    print("Error:", e)
try:
    print(parrot(110, voltage=220))
except Exception as e:
    print("Error:", e)
try:
    print(parrot(actor='John Cleese'))
except Exception as e:
    print("Error:", e)

def function_with_one_argument(number):
    return number
try:
    print(function_with_one_argument(0, number=0))
except Exception as e:
    print("Error:", e)

def test_function(first_param, *arguments, **keywords):
    """This function accepts its arguments through "arguments" tuple and keywords dictionary."""
    print("First param:", first_param)
    print("Arguments tuple:", arguments)
    print("Keywords dict:", keywords)
test_function(
    'first param',
    'second param',
    'third param',
    fourth_param_name='fourth named param',
    fifth_param_name='fifth named param',
)
```

```
This parrot wouldn't voom if you put 1000 volts through it. Lovely plumage, t
he Norwegian Blue. It's a stiff!
This parrot wouldn't voom if you put 1000 volts through it. Lovely plumage, t
he Norwegian Blue. It's a stiff!
This parrot wouldn't VOOOOOM if you put 1000000 volts through it. Lovely plum
age, the Norwegian Blue. It's a stiff!
This parrot wouldn't VOOOOOM if you put 1000000 volts through it. Lovely plum
age, the Norwegian Blue. It's a stiff!
This parrot wouldn't jump if you put 1000000 volts through it. Lovely plumag
e, the Norwegian Blue. It's bereft of life!
This parrot wouldn't voom if you put 1000 volts through it. Lovely plumage, t
he Norwegian Blue. It's pushing up the daisies!
Error: parrot() missing 1 required positional argument: 'voltage'
Error: parrot() got multiple values for argument 'voltage'
Error: parrot() got an unexpected keyword argument 'actor'
Error: function_with_one_argument() got multiple values for argument 'number'
First param: first param
Arguments tuple: ('second param', 'third param')
Keywords dict: {'fourth_param_name': 'fourth named param', 'fifth_param_nam
e': 'fifth named param'}
```

Arbitrary Argument Lists

- Function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

In [ ]:

```python
def test_function(first_param, *arguments):
    """This function accepts its arguments through an "arguments" tuple."""
    print("First parameter:", first_param)
    print("Arguments tuple:", arguments)


# Example usage of test_function
test_function('first param', 'second param', 'third param')


def concat(*args, sep='/'):
    """Concatenates arguments with the specified separator."""
    return sep.join(args)


# Example usage of concat
print(concat('earth', 'mars', 'venus'))
print(concat('earth', 'mars', 'venus', sep='.'))
```

```
First parameter: first param
Arguments tuple: ('second param', 'third param')
earth/mars/venus
earth.mars.venus
```

Unpacking Argument Lists

- Unpacking arguments may be executed via * and ** operators. See below for further details.

In [ ]:

```
# Example 1: Normal call with separate arguments
print("Range with separate arguments:", list(range(3, 6)))
```

```
Range with separate arguments: [3, 4, 5]
```

In [ ]:

```
# Example 2: Call with arguments unpacked from a list
arguments_list = [3, 6]
print("Range with unpacked list arguments:", list(range(*arguments_list)))
```

```
Range with unpacked list arguments: [3, 4, 5]
```

In [ ]:

```
# Example 3: Function receiving named arguments via unpacked dictionary
def function_that_receives_named_arguments(first_word, second_word):
    return first_word + ', ' + second_word + '!'

arguments_dictionary = {'first_word': 'rahul', 'second_word': 'chirra'}
print(
    "Function with unpacked dictionary arguments:",
    function_that_receives_named_arguments(**arguments_dictionary),
)
```

```
Function with unpacked dictionary arguments: rahul, chirra!
```

Lambda Expressions

- Small anonymous functions can be created with the lambda keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope.

In [ ]:

```
def make_increment_function(delta):
    """This example uses a lambda expression to return a function"""
    return lambda number: number + delta

# Creating an increment function
increment_function = make_increment_function(42)

# Testing the increment function with various inputs
print(f"increment_function(0): {increment_function(0)}")  # Expected: 42
print(f"increment_function(1): {increment_function(1)}")  # Expected: 43
print(f"increment_function(2): {increment_function(2)}")  # Expected: 44

# Another use of lambda is to pass a small function as an argument.
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

# Sorting pairs by the text key using a lambda function
pairs.sort(key=lambda pair: pair[1])

# Displaying the sorted pairs
print(f"Sorted pairs: {pairs}")
```

```
increment_function(0): 42
increment_function(1): 43
increment_function(2): 44
Sorted pairs: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

:# Documentation Strings

- The first line of a docstring should briefly describe the purpose, without naming the object or type (unless it's a verb describing an operation). Start with a capital letter and end with a period. Leave a blank second line if more content follows. Use subsequent lines for details on usage, side effects, and conventions.

In [ ]:

```

```

In [1]:

```
def do_nothing():
    pass


# Sequential script execution
print("Documentation string of 'do_nothing':")
print(do_nothing.__doc__)
```

```
Documentation string of 'do_nothing':
None
```

Function Annotations.

- Function annotations are completely optional metadata information about the types used by user-defined functions.

In [3]:

```
def breakfast(ham: str, eggs: str = 'eggs') -> str:
    return ham + ' and ' + eggs


# Sequential execution
print("Function annotations of 'breakfast':")
print(breakfast.__annotations__)
```

```
Function annotations of 'breakfast':
{'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}
```

Function Decorators

- Function Decorators

In [6]:

```python
# This is the function that we want to decorate.
def greeting(name):
    return "Hello, {0}!".format(name)

# This function decorates another function's output with a <p> tag.
def decorate_with_p(func):
    def function_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return function_wrapper

# Using Python's decorator syntax.
@decorate_with_p
def greeting_with_p(name):
    return "Hello, {0}!".format(name)

# Adding another decorator for wrapping with <div>.
def decorate_with_div(func):
    def function_wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return function_wrapper

# Using multiple decorators.
@decorate_with_div
@decorate_with_p
def greeting_with_div_p(name):
    return "Hello, {0}!".format(name)

# Generalized decorator accepting a tag name.
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags('div')
@tags('p')
def greeting_with_tags(name):
    return "Hello, {0}!".format(name)

# Sequential execution
# Create a decorated function.
my_get_text = decorate_with_p(greeting)

# Display results.
print("Decorated function output with <p>:")
print(my_get_text('rahul'))  # With decorator.

print("Original function output without decoration:")
print(greeting('rahul'))  # Without decorator.

print("Function output decorated with @decorate_with_p:")
print(greeting_with_p('rahul'))

print("Function output decorated with <div> and <p>:")
print(greeting_with_div_p('rahul'))
print("Function output with generalized tag-based decorators:")
print(greeting_with_tags('rahul'))
```

```
Decorated function output with <p>:
<p>Hello, rahul!</p>
Original function output without decoration:
Hello, rahul!
Function output decorated with @decorate_with_p:
<p>Hello, rahul!</p>
Function output decorated with <div> and <p>:
<div><p>Hello, rahul!</p></div>
Function output with generalized tag-based decorators:
<div><p>Hello, rahul!</p></div>
```

Function Decorators

In [7]:

```
def greeting(name):
    return "Hello, {0}!".format(name)

def decorate_with_p(func):
    def function_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return function_wrapper

# Using the decorator manually
my_get_text = decorate_with_p(greeting)
print(my_get_text('fuck'))
print(greeting('you'))
```

```
<p>Hello, fuck!</p>
Hello, you!
```

In [9]:

```
# Using Python's decorator syntax
@decorate_with_p
def greeting_with_p(name):
    return "Hello, {0}!".format(name)

print(greeting_with_p('rahul'))  # Output with @decorate_with_p

def decorate_with_div(func):
    def function_wrapper(text):
        return "<div>{0}</div>".format(func(text))
    return function_wrapper
```

```
<p>Hello, rahul!</p>
```

In [10]:

```
# Combining decorators
@decorate_with_div
@decorate_with_p
def greeting_with_div_p(name):
    return "Hello, {0}!".format(name)

print(greeting_with_div_p('rishi'))
```

```
<div><p>Hello, rishi!</p></div>
```

In [11]:

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags('div')
@tags('p')
def greeting_with_tags(name):
    return "Hello, {0}!".format(name)

print(greeting_with_tags('rahul'))
```

```
<div><p>Hello, rahul!</p></div>
```

Class Definition Syntax

- Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

In [13]:

```
class GreetingClass:

    name = 'user'

    def say_hello(self):

        return 'Hello ' + self.name

    def say_goodbye(self):
        """Class method."""
        return 'Goodbye ' + self.name

# Instantiate the class
greeter = GreetingClass()

# Sequential operations with outputs
print(greeter.say_hello())  # Expected output: Hello user
print(greeter.say_goodbye())  # Expected output: Goodbye user
```

```
Hello user
Goodbye user
```

Class Definition Syntax.

In [14]:

```
class ComplexNumber:
    """Example of the complex numbers class"""

    real = 0
    imaginary = 0

    def get_real(self):
        """Return real part of complex number."""
        return self.real

    def get_imaginary(self):
        """Return imaginary part of complex number."""
        return self.imaginary


# Display the default attributes and docstring
print("Default real part:", ComplexNumber.real)
print("Docstring of class:", ComplexNumber.__doc__)

# Change class attributes and display the updated values
ComplexNumber.real = 10
print("Updated real part:", ComplexNumber.real)

# Instantiate the class and display the instance attributes
complex_number = ComplexNumber()
print("Instance real part (from class):", complex_number.real)
print("Instance real part (via method):", complex_number.get_real())

# CLASS INSTANTIATION with constructor to initialize specific attributes
class ComplexNumberWithConstructor:
    """Example of the class with constructor"""
    def __init__(self, real_part, imaginary_part):
        self.real = real_part
        self.imaginary = imaginary_part

    def get_real(self):
        """Return real part of complex number."""
        return self.real

    def get_imaginary(self):
        """Return imaginary part of complex number."""
        return self.imaginary


# Instantiate the class with specific values and display them
complex_number_with_constructor = ComplexNumberWithConstructor(3.0, -4.5)
print("Real part of instantiated object:", complex_number_with_constructor.rea
l)
print("Imaginary part of instantiated object:", complex_number_with_constructo
r.imaginary)
```

```
Default real part: 0
Docstring of class: Example of the complex numbers class
Updated real part: 10
Instance real part (from class): 10
Instance real part (via method): 10
Real part of instantiated object: 3.0
Imaginary part of instantiated object: -4.5
```

Class Definition Syntax.

- The only operations understood by instance objects are attribute references:
- Data attributes
- Methods.

In [15]:

```python
class DummyClass:
    """Dummy class."""
    pass

# Create an instance of DummyClass
dummy_instance = DummyClass()

# Add a temporary attribute dynamically and display its value
dummy_instance.temporary_attribute = 1
print("Temporary attribute value:", dummy_instance.temporary_attribute)

# Delete the temporary attribute and confirm it is removed
del dummy_instance.temporary_attribute
try:
    print(dummy_instance.temporary_attribute)
except AttributeError:
    print("Temporary attribute has been deleted.")
```

```
Temporary attribute value: 1
Temporary attribute has been deleted.
```

Standalone Script Without Assertions.**bold text**

In [16]:

```python
class MyCounter:
    """A simple example of the counter class"""
    counter = 10

    def get_counter(self):
        """Return the counter"""
        return self.counter

    def increment_counter(self):
        """Increment the counter"""
        self.counter += 1
        return self.counter


# Method Objects demonstration
counter = MyCounter()

# Directly displaying the results instead of using assertions
print("Initial counter value (using instance method):", counter.get_counter())

# Storing method object and calling it later
get_counter = counter.get_counter
print("Stored method call result:", get_counter())

# Equivalent of calling the method via the class by passing the instance
print("Counter value (using class method with instance):", MyCounter.get_counter(counter))
```

```
Initial counter value (using instance method): 10
Stored method call result: 10
Counter value (using class method with instance): 10
```

Standalone Script Without Assertions

In [17]:

```
# Class and Instance Variables
class Dog:
    """Dog class example"""
    kind = 'canine'  # Class variable shared by all instances.

    def __init__(self, name):
        self.name = name  # Instance variable unique to each instance.

# Demonstration of class and instance variables
fido = Dog('Fido')
buddy = Dog('Buddy')

print("Fido's kind:", fido.kind)
print("Buddy's kind:", buddy.kind)

print("Fido's name:", fido.name)
print("Buddy's name:", buddy.name)

# Shared data issue with mutable objects
class DogWithSharedTricks:
    """Dog class example with wrong shared variable usage"""
    tricks = []  # Mistaken use of a class variable for mutable objects.

    def __init__(self, name):
        self.name = name  # Instance variable unique to each instance.

    def add_trick(self, trick):
        """Add trick to the dog"""
        self.tricks.append(trick)

fido = DogWithSharedTricks('Fido')
buddy = DogWithSharedTricks('Buddy')

fido.add_trick('roll over')
buddy.add_trick('play dead')

print("Shared tricks (Fido):", fido.tricks)
print("Shared tricks (Buddy):", buddy.tricks)

# Correct design using instance variables
class DogWithTricks:
    """Dog class example"""

    def __init__(self, name):
        self.name = name  # Instance variable unique to each instance.
        self.tricks = []  # creates a new empty list for each dog

    def add_trick(self, trick):
        """Add trick to the dog"""
        self.tricks.append(trick)

fido = DogWithTricks('Fido')
buddy = DogWithTricks('Buddy')

fido.add_trick('roll over')
buddy.add_trick('play dead')
```

```python

print("Tricks for Fido:", fido.tricks)
print("Tricks for Buddy:", buddy.tricks)
```

```
Fido's kind: canine
Buddy's kind: canine
Fido's name: Fido
Buddy's name: Buddy
Shared tricks (Fido): ['roll over', 'play dead']
Shared tricks (Buddy): ['roll over', 'play dead']
Tricks for Fido: ['roll over']
Tricks for Buddy: ['play dead']
```

Standalone Script Demonstrating **Inheritance**

In [19]:

```
# Base class
class Person:
    """Example of the base class"""
    def __init__(self, name):
        self.name = name

    def get_name(self):
        """Get person name"""
        return self.name


# Derived class
class Employee(Person):
    """Example of the derived class"""
    def __init__(self, name, staff_id):
        super().__init__(name)  # Using super() to initialize the base class
        self.staff_id = staff_id

    def get_full_id(self):
        """Get full employee id"""
        return self.get_name() + ', ' + self.staff_id


# Demonstrating inheritance and functionality
person = Person('CHIRRA')
employee = Employee('RAHUL', 'A23')

print("Person's name:", person.get_name())
print("Employee's name:", employee.get_name())
print("Employee's full ID:", employee.get_full_id())

# Demonstrating isinstance and issubclass functions
print("Is 'employee' an instance of Employee?", isinstance(employee, Employe
e))
print("Is 'person' an instance of Employee?", isinstance(person, Employee))

print("Is 'person' an instance of Person?", isinstance(person, Person))
print("Is 'employee' an instance of Person?", isinstance(employee, Person))

print("Is Employee a subclass of Person?", issubclass(Employee, Person))
print("Is Person a subclass of Employee?", issubclass(Person, Employee))
```

```
Person's name: CHIRRA
Employee's name: RAHUL
Employee's full ID: RAHUL, A23
Is 'employee' an instance of Employee? True
Is 'person' an instance of Employee? False
Is 'person' an instance of Person? True
Is 'employee' an instance of Person? True
Is Employee a subclass of Person? True
Is Person a subclass of Employee? False
```

Multiple **Inheritance**

- Some classes may derive from multiple classes. This means that the derived class would have its attributes, along with the attributes of all the classes that it was derived from.

In [22]:

```python
# Base class
class Clock:
    time = '11:23 PM'  # Example of class variable

    def get_time(self):
        return self.time

# pylint: disable=too-few-public-methods
class Calendar:
    date = '01/01/2025'

    def get_date(self):
        return self.date

# Python supports a form of multiple inheritance as well. A class definition with multiple
# base classes looks like this.
class CalendarClock(Clock, Calendar):
    """CalendarClock class inheriting from Clock and Calendar"""
    def __init__(self):
        # optionally initialize some attributes here if needed
        pass

calendar_clock = CalendarClock()

# Print statements to verify the functionality
print(calendar_clock.get_date())  # Expected output: '12/08/2018'
print(calendar_clock.get_time())  # Expected output: '11:23 PM'
```

```
01/01/2025
11:23 PM
```

ends.