A simple basic function to print a something on the screen

```
print("Hello World")
```

```
Hello World
```

Variables: A named memory location which is used to store a value in it.

```
a = 20
print(a)
```

```
20
```

```
 a = 20
 b = 30
 print(a+b)
```

```
50
```

#Exercise Take user input and print the following:

• We check in a patient named John Smith.

• He is 20 years old.

• He is a new patient.

```
name = "John Smith"
age = 20
new_patient = True
print(name, ",", age,",", new_patient)
```

```
John Smith , 20 , True
```

User Prompt in python

1.Generic Syntax: variable = input("prompt") [for string data type]

2.for other data type: variable= input(data_type("prompt"))

```
#Another way
name = input("Enter the name:")
age = int(input("Enter the age:"))
new_patient = True
print(name, ",", age,",", new_patient)
```

```
Enter the name:John Smith
Enter the age:22
John Smith , 22 , True
```

Type Conversion

Conversion of a data type to another data type

```
birth_year = input("Enter your birth year:")
age = 2024-int(birth_year)#without integer it will show error
print(age)
```

```
Enter your birth year:2002
22
```

```
a = int(input("Enter the number: "))
b = int(input("Enter the number: "))
sum = a+b
print(sum)
```

```
Enter the number: 11
Enter the number: 12
23
```

```
a = int(input("Enter the number: "))
b = float(input("Enter the number: "))
sum = a+b
print(sum)
```

```
Enter the number: 11
Enter the number: 21
32.0
```

```
a = int(input("Enter the number: "))
b = int(input("Enter the number: "))
sum = (float)(a+b)
print(sum)
```

```
Enter the number: 11
Enter the number: 21
32.0
```

```
a = input("Enter the number: ")
b = input("Enter the number: ")
sum = a+b
print(sum)
```

```
Enter the number: 11
Enter the number: 11
1111
```

```
a = int(input("Enter the number: "))
b = int(input("Enter the number: "))
sum = a+b
print(sum)
```

```
Enter the number: 11
Enter the number: 12
23
```

```
a = input("Enter the number: ")
b = input("Enter the number: ")
sum = (int)(a+b)
print(sum)
```

```
Enter the number: 11
Enter the number: 12
1112
```

```
type(sum)
```

```
int
```

String is a data type that is called an array of characters enclosed within double quotes.

```
course = "Hello How are you?"
course.capitalize()
```

```
{"type":"string"}
```

```
course.find("o")
```

```
4
```

```
course.upper()
```

```
{"type":"string"}
```

```
course.replace("o","a")
```

```
{"type":"string"}
```

Arithmatic operators

```
print(2+3)#Addition
print(2-3)#Subtraction
print(2*3)#Multiplication
print(2/3)#floating point Division
print(2//3)#integger value division
print(2**3)#power
print(2%3)#Modulus gives remainder in normal case but when number 1 < number 2 then number 1 is straight away the answer.

5
-1
6
0.6666666666666666
0
8
2
```

#Logical Operator

```
print((2>3)and(3>1))# Logical and operator returns true if both conditions are true.
print((2>3)or(3>1))# Logical or operator returns true if either of the conditions are true.
print(not(2>3))# Returns the inverted value of the original one

False
True
True
```

Relational Operator

```
print(2>3)
print(2<3)
print(2>=3)
print(2<=3)
print(2==3)
print(2!=3)

False
True
False
True
False
True
```

Conditional Statements

It simply means they give results as per the condition specified which is connected to the block of code. Based on the condition these statements gives the result.

Types:

1. if
2. if-elif-else [if else if in other language like java,c,c++]
3. Nested if [if inside another if condition]
4. if-else

```
if 2>3:
  print("Hello")
#---------------------------------------
if 2<3:
  print("Hello")
else:
  print("Bye")
#---------------------------------------
number_1 = int(input("Enter a number:"))
number_2 = int(input("Enter a number:"))
operator = input("Enter an operator:")
if operator == "+":
  print(number_1+number_2)
elif operator == "-":
  print(number_1-number_2)
elif operator == "*":
  print(number_1*number_2)
elif operator == "/":
  print(number_1/number_2)
else:
  print("Invalid operator")
#---------------------------------------
if 2<3:
  print("Hello")
  if 3<4:
    print("How are you?")
  else:
    print("Bye")
else:
  print("Bye")
```

```
Hello
Enter a number:11
Enter a number:12
Enter an operator:+
23
Hello
How are you?
```

Weight kg to lbs convertor or vice versa

```
Weight = int(input("Weight:"))
value_of_weight = input("(K)g or (L)bs:")
if value_of_weight == "L" or value_of_weight == "l":
  print("Weight in Kg:",Weight*0.453592)
elif value_of_weight == "K" or value_of_weight == "k":
  print("Weight in lbs:",Weight*2.2046)
else:
  print("Invalid input")
```

```
Weight:69
(K)g or (L)bs:K
Weight in lbs: 152.1174
```

Lists

These are python Data Structures that are used to store the elements of the same type in a pair of square brackets.

Python lists come with a variety of built-in methods that allow you to manipulate and interact with the data they contain. Here's a comprehensive list of the methods available for Python lists:

1. **append(x)**: Adds an item `x` to the end of the list.
2. **extend(iterable)**: Extends the list by appending elements from the iterable.
3. **insert(i, x)**: Inserts an item `x` at a given position `i`.
4. **remove(x)**: Removes the first item from the list whose value is equal to `x`. Raises a ValueError if not found.
5. **pop([i])**: Removes and returns the item at the given position `i` in the list. If no index is specified, `pop()` removes and returns the last item in the list.
6. **clear()**: Removes all items from the list.
7. **index(x[, start[, end]])**: Returns the index of the first item whose value is equal to `x`. Raises a ValueError if not found.
8. **count(x)**: Returns the number of times `x` appears in the list.
9. **sort(key=None, reverse=False)**: Sorts the items of the list in place (the arguments can be used for sort order).
10. **reverse()**: Reverses the elements of the list in place.
11. **copy()**: Returns a shallow copy of the list.

These methods provide powerful ways to work with lists, allowing for dynamic data manipulation and organization in Python programming.

```
names = ["John", "Bob", "Mosh", "Sam", "Mary"]
print(names)
```

```
['John', 'Bob', 'Mosh', 'Sam', 'Mary']
```

```
names[4] = "Yennefer"
print(names)
```

```
['John', 'Bob', 'Mosh', 'Sam', 'Yennefer']
```

```
names.append("Alice")
print(names)
```

```
['John', 'Bob', 'Mosh', 'Sam', 'Yennefer', 'Alice']
```

```
names.count("John")
```

```
1
```

```
names.insert(2,"Angela")
print(names)
```

```
['John', 'Bob', 'Angela', 'Mosh', 'Sam', 'Yennefer', 'Alice']
```

```
names.remove("Angela")
print(names)
```

```
['John', 'Bob', 'Mosh', 'Sam', 'Yennefer', 'Alice']
```

```
names.extend(["Alice","Bob"])
print(names)
```

```
['John', 'Bob', 'Mosh', 'Sam', 'Yennefer', 'Alice', 'Alice', 'Bob']
```

```
names.sort()
print(names)
```

```
['Alice', 'Alice', 'Bob', 'Bob', 'John', 'Mosh', 'Sam', 'Yennefer']
```

```
names.pop(1)
print(names)
```

```
['Alice', 'Bob', 'Bob', 'John', 'Mosh', 'Sam', 'Yennefer']
```

```
names.copy()
print(names)
```

```
['Alice', 'Bob', 'Bob', 'John', 'Mosh', 'Sam', 'Yennefer']
```

```
names
```

```
['Alice', 'Bob', 'Bob', 'John', 'Mosh', 'Sam', 'Yennefer']

names.remove("Bob")
print(names)

['Alice', 'Bob', 'John', 'Mosh', 'Sam', 'Yennefer']

names.clear()
print(names)

[]

print("Alice" in names)

False
```

Loops

In Python, there are primarily two types of loops: **for loops** and **while loops**. Here's a detailed description of each along with their syntax:

1. For Loop

The `for` loop is used to iterate over a sequence (like a list, tuple, dictionary, set, or string).

Syntax:

```
for variable in iterable:
    # code block to execute
```

Example:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

2. While Loop

The `while` loop repeatedly executes a block of code as long as a specified condition is true.

Syntax:

```
while condition:
    # code block to execute
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

3. Nested Loops

Both `for` and `while` loops can be nested inside each other.

Syntax:

```
for variable in iterable:
    for inner_variable in inner_iterable:
        # code block to execute
```

Example:

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

4. Loop Control Statements

Python also provides control statements to alter the flow of loops:

- **break**: Exits the loop prematurely.

  Example:

  ```
  for num in range(10):
      if num == 5:
          break
      print(num)
  ```

- **continue**: Skips the current iteration and continues with the next one.

  Example:

  ```
  for num in range(5):
      if num == 3:
          continue
      print(num)
  ```

- **else**: Can be used with loops to execute a block of code when the loop finishes normally (not terminated by `break`).

  Example:

```
    for num in range(5):
        print(num)
    else:
        print("Loop completed!")
```

These loops and control statements provide flexibility for iterating and managing the flow of execution in Python programs.

```
for i in range(5):
  for j in range(i):
    print("*", end=" ")
  print()
```

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

```
for i in range(5):
  for j in range(i+1):
    print("*", end=" ")
  print()
```

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

```
for i in range(5):
  for j in range(5-i):
    print("*", end=" ")
  print()
```

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

```
for i in range(5):
  for j in range(5-i):
    print(" ", end=" ")
  for j in range(i+1):
    print("*", end=" ")
  print()
```

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

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

```
for i in range(5):
  for j in range(i+1):
    print(" ", end=" ")
  for j in range(5-i):
    print("*", end=" ")
  print()
```

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

```
for i in range(5):
  for j in range(i+1):
    print(" ", end=" ")
  for j in range(5-i):
    print("*", end=" ")
    for k in range(5-i):
      print("*", end=" ")
  print()
```

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

Function Declaration in python

def function_name(parameters):

```
code block
```

function_name()

def is the keyword to declare a function.

In Python, functions are reusable blocks of code that perform a specific task. They can take inputs, process them, and return outputs. Here's a detailed description of functions in Python, including their syntax, types, and key concepts.

1. Defining a Function

You define a function using the `def` keyword, followed by the function name and parentheses containing any parameters.

Syntax:

```
def function_name(parameters):
    # code block
    return value  # optional
```

Example:

```
def greet(name):
    return f"Hello, {name}!"
```

2. Calling a Function

Once a function is defined, you can call it by using its name followed by parentheses.

Example:

```
print(greet("Alice"))  # Output: Hello, Alice!
```

3. Parameters and Arguments

- **Parameters**: Variables listed inside the parentheses in the function definition.
- **Arguments**: Values you pass to the function when calling it.

Example with parameters:

```
def add(a, b):
    return a + b

result = add(5, 3)  # Here, 5 and 3 are arguments
```

4. Types of Parameters

- **Positional Parameters**: Arguments are assigned to parameters based on their position.
- **Keyword Parameters**: Arguments are assigned to parameters by explicitly naming them.

  Example:

  ```
  def person(name, age):
      return f"{name} is {age} years old."

  print(person(age=30, name="Bob"))  # Output: Bob is 30 years old.
  ```

- **Default Parameters**: You can provide default values for parameters.

  Example:

```
    def greet(name, greeting="Hello"):
        return f"{greeting}, {name}!"

    print(greet("Alice"))          # Output: Hello, Alice!
    print(greet("Bob", "Hi"))      # Output: Hi, Bob!
```

- **Variable-length Parameters**: Use `*args` for non-keyword variable-length arguments and `**kwargs` for keyword variable-length arguments.

  Example:

```
    def add(*args):
        return sum(args)

    print(add(1, 2, 3))  # Output: 6

    def info(**kwargs):
        return kwargs

    print(info(name="Alice", age=30))  # Output: {'name': 'Alice',
    'age': 30}
```

5. Return Statement

The `return` statement is used to exit a function and optionally pass back a value.

Example:

```
def square(x):
    return x * x

result = square(4)  # result is 16
```

6. Lambda Functions

Lambda functions are anonymous functions defined with the `lambda` keyword. They can take any number of arguments but can only have one expression.

Syntax:

```
lambda arguments: expression
```

Example:

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

7. Scope of Variables

Variables defined inside a function are local to that function and cannot be accessed outside of it. Conversely, variables defined outside of any function are global and can be accessed anywhere.

Example:

```
def my_function():
    local_var = 5  # Local variable
    return local_var

print(my_function())  # Output: 5
# print(local_var)    # This would raise an error
```

8. Docstrings

You can document a function using a docstring, which is a string literal that occurs as the first statement in a function. It describes what the function does.

Example:

```
def multiply(a, b):
    """Multiply two numbers and return the result."""
    return a * b

print(multiply.__doc__)  # Output: Multiply two numbers and return the
result.
```

9. Higher-Order Functions

Functions can take other functions as arguments or return them as results.

Example:

```
def apply_function(func, value):
    return func(value)

print(apply_function(square, 5))  # Output: 25
```

Summary

Functions in Python are powerful tools for organizing and reusing code. They support various types of parameters, can return values, and enable the creation of complex programs through abstraction and encapsulation. Understanding how to define and use functions effectively is essential for writing clean and efficient Python code.

```
def square(x):
  return x*x
def apply_function(func, value):
```

```
    return func(value)
print(apply_function(square, 5))
```

```
25
```

```
def multiply(a, b):
    """Multiply two numbers and return the result."""
    return a * b

print(multiply.__doc__)  # Output: Multiply two numbers and return the
result.
```

```
Multiply two numbers and return the result.
```

```
add = lambda x, y: x + y
print(add(11,12))
```

```
23
```

1. Write a program in python to calculate the factorial of numbers till a particular number.

```
def factorial():
  n = int(input("Enter a number:"))
  fact = 1
  for i in range(1, n+1):
    fact = fact * i
    print(fact)
factorial()
```

```
Enter a number:10
1
2
6
24
120
720
5040
40320
362880
3628800
```

Tuples

These are python's immutable data structures, which simply means once they are declared they cannot be changed in a program during the execution of the program.

In Python, a **tuple** is a built-in data type that is used to store a collection of items. Tuples are similar to lists, but they have some key differences:

Characteristics of Tuples

1. **Immutable**: Once a tuple is created, its elements cannot be changed, added, or removed. This immutability makes tuples suitable for storing fixed collections of items.
2. **Ordered**: Tuples maintain the order of elements. The items in a tuple can be accessed using their index.
3. **Heterogeneous**: Tuples can hold items of different data types, including integers, strings, lists, and even other tuples.
4. **Can Contain Duplicates**: Tuples can have duplicate values.

Creating a Tuple

You can create a tuple by placing a comma-separated sequence of values inside parentheses `()`.

Syntax:

```
tuple_name = (item1, item2, item3)
```

Example:

```
my_tuple = (1, 2, 3, "apple", "banana")
```

Accessing Tuple Elements

You can access tuple elements using indexing, where the index starts at 0.

Example:

```
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: apple
```

Slicing Tuples

You can slice tuples to get a subset of elements.

Example:

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

Tuple Methods

Tuples have a few built-in methods:

1. **count(value)**: Returns the number of times a specified value appears in the tuple.

   Example:

```
my_tuple = (1, 2, 2, 3)
print(my_tuple.count(2))  # Output: 2
```

2. **index(value)**: Returns the index of the first occurrence of a specified value. Raises a ValueError if the value is not found.

   Example:

```
print(my_tuple.index(2))  # Output: 1
```

Tuple Packing and Unpacking

- **Packing**: Creating a tuple by assigning multiple values to a single variable.

  Example:

```
packed_tuple = 1, 2, 3  # Tuple packing
```

- **Unpacking**: Assigning the elements of a tuple to multiple variables.

  Example:

```
a, b, c = packed_tuple
print(a, b, c)  # Output: 1 2 3
```

Nested Tuples

Tuples can contain other tuples as elements.

Example:

```
nested_tuple = (1, (2, 3), 4)
print(nested_tuple[1])  # Output: (2, 3)
```

Advantages of Using Tuples

1. **Performance**: Tuples are generally faster than lists for iteration due to their immutability.
2. **Data Integrity**: The immutability of tuples makes them a good choice for fixed data that should not change.
3. **Hashable**: Tuples can be used as keys in dictionaries, while lists cannot.

Conclusion

Tuples are a versatile and efficient way to store collections of items in Python. Their immutability, ordering, and ability to hold heterogeneous data make them suitable for various programming scenarios. Understanding how to create and manipulate tuples is essential for effective Python programming.

```
t1 = ('1','2','3','4','5')
print(t1)
```

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

```
t1.count('1')
```

```
1
```

```
t1.index('1')
```

```
0
```

```
t1[1:3]
```

```
('2', '3')
```

```
t2 = ('1', '2', '3', '4', 'hello', 'aur kya haal chaal')
print(t2.count('hello'))
print(t2.index('aur kya haal chaal'))
```

```
1
5
```

```
print(t2[1:3])
```

```
('2', '3')
```