What are Functions?

Block of code for specific tasks.

Black Box Concept:

Ignore internal code; focus on inputs/outputs.

Advantages:

- Reusability
- Avoids repetition

Two Key Points Regarding Functions:

1. Abstraction

- Hides internal workings.
- Users know **"what"** it does, not **"how"**.

2. Decomposition

- Splits systems into modules.
- Each module offers specific functionality.
- Modules can impact others.

In [ ]:

```
# Components of a Function

def function_name(parameters):
    """docstring"""
    statement(s)

`def`       ---> Function start.
Name        ---> Function identifier.
Params      ---> Input values.
Colon (`:`) ---> Ends header.
Docstring   ---> Function description.
Body        ---> Statements.
`return`    ---> Output value (optional).

function_name(values)
```

Let's create a function

In [17]:

```
# Check if number is even/odd
def is_even(number):
    """
    This function states that, if a given number is odd or even
    Input - any valid integer
    Output - odd/even
    Created By - Roshan the swagger
    Last edited - 30 Jul 2025
    """
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
```

In [4]:

```
is_even(33)
```

Out[4]:

```
'Odd'
```

In [8]:
```python
is_even(27)
```

Out[8]:
```
'Odd'
```

In [12]:
```python
for i in range(1,11):
    print(i,"-->", is_even(i))
```

```
1 --> Odd
2 --> Even
3 --> Odd
4 --> Even
5 --> Odd
6 --> Even
7 --> Odd
8 --> Even
9 --> Odd
10 --> Even
```

In [7]:
```python
print(is_even.__doc__)
```

```

    This function states that, if a given number is odd or even
    Input - any valid integer
    Output - odd/even
    Created By - Roshan the swagger
    Last edited - 30 Jul 2025

```

In [14]:
```python
print.__doc__
```

Out[14]:
```
"print(value, ..., sep=' ', end='\\n', file=sys.stdout, flush=False)\n\nPrints the values to a stream, or to sys.stdout
by default.\nOptional keyword arguments:\nfile:  a file-like object (stream); defaults to the current sys.stdout.\nsep:
string inserted between values, default a space.\nend:   string appended after the last value, default a newline.\nflus
h: whether to forcibly flush the stream."
```

In [15]:
```python
type.__doc__
```

Out[15]:
```
"type(object) -> the object's type\ntype(name, bases, dict, **kwds) -> a new type"
```

Functions: 2 Perspectives

1. Creator's perspective
2. User's perspective

In [13]:
```
pwd
```

Out[13]:
```
'C:\\Users\\Ahmed Ali\\Python\\Python_Programming'
```

In [18]:
```
is_even(7)
```

Out[18]:
```
'Odd'
```

In [21]:
```
is_even("Hello")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 1
----> 1 is_even("Hello")

Cell In[17], line 10, in is_even(number)
      2 def is_even(number):
      3     """
      4     This function states that, if a given number is odd or even
      5     Input - any valid integer
   (...)
      8     Last edited - 30 Jul 2025
      9     """
---> 10     if number % 2 == 0:
     11         return "Even"
     12     else:

TypeError: not all arguments converted during string formatting
```

In [27]:

```
def is_even(number):
    if type(number) == int:
        if number%2 == 0:
            print("Even")
        else:
            print("Odd")
    else:
        print("Not allowed")
```

In [30]:

```
is_even("Hello")
```

```
Not allowed
```

In [ ]:

```
# Creating `is_even.py` file w `is_even()` function ---> to import into Jupyter Notebook.
```

In [35]:

```
import function
```

In [39]:

```
function.is_even("Hellow")
```

Out[39]:

```
'Not Allowed'
```

Parameters Vs Arguments

Parameters:

- Vars in `()` during func definition..
- Defined in func declaration.

```
def func(param1, param2):
    # Body
```

Arguments:

- Values passed at func call.
- Inputs during function invocation.

```
func(arg1, arg2)
```

1. *Default Argument*

2. *Positional Argument*

3. *Keyword Argument*

4. *Arbitrary Argument (`*args`)*

```
In [40]: def power(a,b):
             return a**b
```

```
In [41]: power(2,3)

Out[41]: 8
```

```
In [44]: power(3, 2)

Out[44]: 9
```

```
In [42]: power(3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[42], line 1
----> 1 power(3)

TypeError: power() missing 1 required positional argument: 'b'
```

In [45]:
```
power()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[45], line 1
----> 1 power()

TypeError: power() missing 2 required positional arguments: 'a' and 'b'
```

In [46]:
```
# Default Argument: Function arguments with default values.
def power(a=1, b=1):
    return a**b
```

In [47]:
```
power(2,3)
```

Out[47]:
```
8
```

In [48]:
```
power(2)
```

Out[48]:
```
2
```

In [49]:
```
power()
```

Out[49]:
```
1
```

In [52]:
```
# Positional Arguments: Values assigned by call order.
power(3, 2)
```

Out[52]:
```
9
```

In [51]:

```
# Keyword Argument: Values assigned to args by name at call time.

# NOTE: *Keyword args* will Overrides *Positional args*.

# Priority ---> Keyword args > Positional args.

power(b = 3,a = 2)
```

Out[51]:

```
8
```

In [52]:

```
# Arbitrary Argument: Accepts any number of args.
# Useful when the number of arguments is unknown.
def flexi(*number):
    product = 1
    for i in number:
        product *= i
    print(product)
```

In [53]:

```
flexi(1)
```

```
1
```

In [54]:

```
flexi(1, 2)
```

```
2
```

In [55]:

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

```
6
```

In [56]:

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

```
3628800
```

In [57]:

```
def flexi(*number): # Flexible inputs ---> tuple
    product = 1
    print(number)
    print(type(number))
    for i in number:
        product *= i
    print(product)
```

In [58]:

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

```
(1, 2, 3, 4, 5)
<class 'tuple'>
120
```

*args and **kwargs

`*args` : Variable-length positional arguments.

```
def func(*args)
```

`**kwargs` : Variable-length keyword arguments.

```
def func(**kwargs)
```

In [75]:

```
# *args
# Pass variable non-keyword args to func
def multiply(*kwargs):
  product = 1
  for i in kwargs:
    product *= i
  print(kwargs)
  return product
```

In [76]:

```python
multiply(1, 2, 3)
```

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

Out[76]:

```
6
```

In [77]:

```python
# **kwargs
# Pass any no. of keyword args (key-value pairs).
# Acts like a dict.
def display(**bushan):
    for (key, value) in bushan.items():
        print(key, '->', value)
```

In [81]:

```python
display(india = 'delhi', srilanka = 'colombo', nepal = 'kathmandu', pakistan = 'islamabad')
```

```
india -> delhi
srilanka -> colombo
nepal -> kathmandu
pakistan -> islamabad
```

Notes: while using `*args` and `**kwargs`

- Argument order: `normal` ---> `*args` ---> `**kwargs`
- The words " `args` " and " `kwargs` " are only a convention, you can use any name of your choice

How Functions Are Executed in Memory?

Functions in Python are defined when `def` is encountered. Execution continues until a function call (e.g., `print`) is made. Each call allocates a separate memory block for that function. Variables within a function are confined to its own block.

analogy ---> *RAM == city, program == house, function == room.*

Functions operate independently, like distinct programs; their memory is released post-completion.

In [82]:

```
#Without return statement
L = [1, 2, 3]
print(L.append(4))
print(L)
```

```
None
[1, 2, 3, 4]
```

Global Var and Local Var

Examples:

In [23]:

```
# Functions as Arguments
def func_a():
    print("inside func_a: ")
    # No return value ---> `None`
def func_b(y):
    print("inside func_b: ")
    return y
def func_c(z):
    print("inside func_c: ")
    return z()
print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```

```
inside func_a:
None
inside func_b:
7
inside func_c:
inside func_a:
None
```

In [40]:

```
# Variable scope & function behavior

def f(y):
    x = 1     # Local x
    x += 1
    print(x)

x = 5         # Global x
f(x)          # Calls f()
print(x)

# Functions have local scope. Global vars coexist but are not affected.
```

```
2
5
```

Local Variables: Inside function.

Global Variables: Outside any function, in main program.

In [24]:

```
def g(y):
    print(x)     # x (global) used in g()
    print(x + 1) # x (global) remains 5; new int (6) created, x unchanged

x = 5
g(x)
print(x)       # x = 5 remains unchanged
```

```
5
6
5
```

In [29]:

```
def h(y):
    x =
    x += 1  # Error: needs "global x" to modify x
x = 5
h(x)
print(x)

# Rule: Global vars: accessed but not modified in functions.
# Concept 1: Globals exist outside funcs, accessed by any func.
# Concept 2: Funcs without local vars can use globals.
# Concept 3: Locals access globals but can't modify.
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[29], line 4
      2     x += 1  # Error: needs "global x" to modify x
      3 x = 5
----> 4 h(x)
      5 print(x)

Cell In[29], line 2, in h(y)
      1 def h(y):
----> 2     x += 1

UnboundLocalError: local variable 'x' referenced before assignment
```

In [25]:

```
# EXPLICITLY Modifying Global Variables Locally
def h(y):
    global x # Note: Modifying global vars is discouraged
    x += 1
x = 5
h(x)
print(x)
```

```
6
```

In [26]:

```
# Complicated Scope
def f(x):
    x += 1
    print("in f(x): x =", x)
    return x

x = 3
z = f(x)
print("in main proram scope: z =", z)
print("in main program scope: x =", x)
```

```
in f(x): x = 4
in main proram scope: z = 4
in main program scope: x = 3
```

Nested Functions

In [27]:

```
def f():
    print("Inside f")
    def g():
        print("Inside g")
    g()
```

In [28]:

```
f()
```

```
Inside f
Inside g
```

In [29]:

```
g()
# Nested Function stays Abstracted/Hidden from main program
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[29], line 1
----> 1 g()

TypeError: g() missing 1 required positional argument: 'y'
```

In [26]:

```
def f():
    print("Inside f")
    def g():
        print("Inside g")
        f()
    g()
```

In [ ]:

```
f()
# Infinite Loop ---> Code will Crash ---> Kernel Dead
```

In [35]:

```
# Harder Scope
def g(x):
    def h():
        x = "abc"
    x += 1
    print("in g(x): x =", x)
    h()
    return x
x = 3
z = g(x)
```

```
in g(x): x = 4
```

In [2]:

```
# Complicated Scope
def g(x):
    def h(x):
        x += 1
        print("in h(x): x =", x)
    x += 1
    print("in g(x): x =", x)
    h(x)
    return x
x = 3
z = g(x)
print("in main proram scope: x =", x)
print("in main program scope: z =", z)
```

```
in g(x): x = 4
in h(x): x = 5
in main proram scope: x = 3
in main program scope: z = 4
```

Everything in Python an Object

Functions too

In [3]:

```
# Functions as Objects
```

In [4]:

```
def raise_to(num):
    return num**2
```

In [5]:

```
raise_to(3)
```

Out[5]:

```
9
```

In [6]:
```
raise_to(4)
```

Out[6]:
```
16
```

In [8]:
```
x = raise_to # aliasing
```

In [5]:
```
# since functions are objects just like int, str,
```

In [9]:
```
x(2)
```

Out[9]:
```
4
```

In [10]:
```
x(4)
```

Out[10]:
```
16
```

In [11]:
```
type(x)
```

Out[11]:
```
function
```

In [12]:
```
del raise_to# Del functions in Python
```

In [13]:
```
raise_to(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 raise_to(2)

NameError: name 'raise_to' is not defined
```

In [14]:
```
x(2) # Call by Object Reference
```

Out[14]:
```
4
```

In [15]:
```
type(x)
```

Out[15]:
```
function
```

In [16]:
```
L = [1, 2, 3, 4]
L
```

Out[16]:
```
[1, 2, 3, 4]
```

In [17]:
```
L = [1, 2, 3, 4, x]
L
```

Out[17]:
```
[1, 2, 3, 4, <function __main__.raise_to(num)>]
```

In [20]:
```
L[-1](-2) # sqr
 #x(-3)   -3 x -3 = +9
```

Out[20]:
```
4
```

In [21]:
```
L = [1, 2, 3, 4, x(5)]
L
```

Out[21]:
```
[1, 2, 3, 4, 25]
```

In [19]:
```
# In Python, Functions behave like any other Data type.
# Can be assigned, passed, and returned.
```

So What?

1. *Renaming Function:* `def new_name(old_name):`

2. *Deleting Function:* `del func_name`

3. *Storing Function:* `func_var = def_func()`

4. *Returning Function:* `return func_name`

5. *Function as Argument:* `def outer(func): func()`

In [25]:
```
# Function as argument/input
def func_a():
    print("inside func_a")

def func_c(z):
    print("inside func_c")
    return z()
print(func_c(func_a))
```

```
inside func_c
inside func_a
None
```

In [21]:
```
# Returning a Function + Nested Calling
def f():
    def x(a, b):
        return a + b
    return x
val = f()(3, 4)
print(val)
```

```
7
```

Functions are First-Class Citizens in Python.

In [13]:
```
# type & id
def square(num):
    return num**2
print(type(square))
print(id(square))
```

```
<class 'function'>
2836007176656
```

In [15]:

```python
# reassign
x = square
print(id(x))
x(3)
```

```
2836007176656
```

Out[15]:

```
9
```

In [8]:

```python
a = 2
b = a
b
```

Out[8]:

```
2
```

In [9]:

```python
# Deleting Function
del square
```

In [10]:

```python
square(3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 square(3)

NameError: name 'square' is not defined
```

In [13]:

```python
# Storing
L = [1, 2, 3, 4, square]
L[-1](3)
```

Out[13]:

```
9
```

In [14]:

```python
s = {square}
s
```

Out[14]:

```
{<function __main__.square(num)>}
```

Benefits of Functions

- **Modularity**: Self-contained code, modularizes login.
- **Reusability**: Write once, use forever.
- **Readability**: Organized and coherent.