Lesson:

Functions Basics, Parameter Passing, Iterators

PW SKILLS

Topics Covered

- What are Functions
- Roles and Importance of Functions
- Function Declaration and Calling
- Function Arguments
- Default Arguments
- ‘return` statements and `print` statements
- Variable length args
- Iterator Function

1. What are Functions?

Functions are nothing but a block of code which performs a specific task and can be called again and again as per requirement in the code.

A generic way to write a simple function is shown below.

```
def function_name():
    # function_name is your choice

    # function body starts

    # write the actions to be performed by the functions
    pass
    # function body ends


# calling function in the code using a () at the end.
function_name()
```

Functions are like Mini Programs: Function is a small, reusable program within your big program. You can give it a name and tell it what to do.

Avoid Doing the Same Thing Over and Over: Instead of writing the same code again and again, you can use a function. It's like having a magic tool that does a specific job for you.

Keep Your Code Neat and Clear: Functions help organize your code. You don't need to understand how the magic tool works; you just need to know how to use it. This makes your code easier to read and work with.

Roles and Importance of Functions:

Functions in Python programming play a crucial role and are important for several reasons, and I'll explain this in easy language:

1. **Simplifying Tasks:** Functions are like helpers that you can create to do specific jobs. Instead of writing the same code again and again, you can make a function to do it once. This makes your program shorter and easier to understand.

2. **Keeping Things Organized:** In a big program, there can be lots of code. Functions help keep things organized. You give each job a name and put the code for that job inside the function. So, you know where to find things.

3. **Reusability:** Once you create a function, you can use it in different parts of your program whenever you need that job done. This saves time, reduces mistakes, and makes your code more efficient.

4. **Collaboration:** When working with others, functions make it easier to share and understand the code. You can tell your friends, "Use this function to do that task," and they'll understand what it does.

Function Declaration and Calling:

Example 1:

```
# function with a print statement

def greetings():
    print('Hi there from PWskills')

greetings() #function calling

# output: Hi there from PWskills
```

Example 2:

```
# function to return the sum of two numbers taking the input form user

def sum_():
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print(a+b)
sum_() #function calling
```

Here is a example where we use function to avoid the repetitive use of code:

```
# Block One
BIKE = True
CAR = True
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")


# Block Two
BIKE = True
CAR = False
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")

# Block Three
BIKE = False
CAR = True
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")

# Block Four
BIKE = False
CAR = False
TRAVEL_100_KM = BIKE or CAR

print(f"You have BIKE: {BIKE}")
print(f"You have CAR: {CAR}")
print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")
```

Output:

```
You have BIKE: True
You have CAR: True
You can travel 100 KMs: True

You have BIKE: True
You have CAR: False
You can travel 100 KMs: True

You have BIKE: False
You have CAR: True
You can travel 100 KMs: True

You have BIKE: False
You have CAR: False
You can travel 100 KMs: False
```

Function Calling:

```
def travel_or_not(BIKE, CAR):
    TRAVEL_100_KM = BIKE or CAR

    print(f"You have BIKE: {BIKE}")
    print(f"You have CAR: {CAR}")
    print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")


travel_or_not(False, False)  #function calling

travel_or_not(False, True)  #function calling

travel_or_not(True, False)  #function calling

travel_or_not(True, True)  #function calling
```

Function Calling Output:

```
You have BIKE: False
You have CAR: False
You can travel 100 KMs: False

You have BIKE: False
You have CAR: True
You can travel 100 KMs: True

You have BIKE: True
You have CAR: False
You can travel 100 KMs: True

You have BIKE: True
You have CAR: True
You can travel 100 KMs: True
```

We can make it still more concise:

```
def travel_or_not(BIKE, CAR):
    TRAVEL_100_KM = BIKE or CAR

    print(f"You have BIKE: {BIKE}")
    print(f"You have CAR: {CAR}")
    print(f"You can travel 100 KMs: {TRAVEL_100_KM}\n")

BIKE = [False, False, True, True]
CAR = [False, True,False, True]

for bike, car in zip(BIKE, CAR):
    travel_or_not(bike, car)  #function calling
```

Output:

```
You have BIKE: False
You have CAR: False
You can travel 100 KMs: False

You have BIKE: False
You have CAR: True
You can travel 100 KMs: True

You have BIKE: True
You have CAR: False
You can travel 100 KMs: True

You have BIKE: True
You have CAR: True
You can travel 100 KMs: True
```

Function Arguments:

- In the above example, `BIKE` and `CAR` inside the brackets after `function_name` are called arguments. i.e; they are used to pass some input to the function while calling.
- Function arguments are values that you provide to a function when you call it. These arguments can be used by the function to perform specific tasks.

Example1:

```
# function to define the sum of two numbers
# here a and b are called arguments

def sum_(a,b):
    return a+b

# call the function with arguments
sum_(2,3)

#output
5
```

Example2:

```
# function to find the square of a number
def square(a):
    return a**2

# call the function with arguments
square(5)

#output
25
```

Example3:

```
# power of any number
def power(base, power):
    return base ** power

# call the function with arguments
power(6, 2)

#output
36
```

Example4:

```
# sum of numbers in list
def sum_(a):

    s = 0
    for i in a:
        s += i
    return s

# here the input must be a list of numbers
sum_([1,2,3,4,5,6])

#output
21
```

Default Arguments:

- You can provide default values for function parameters. If an argument isn't passed when calling the function, the default value is used.

Consider a problem statement where you have to return a sum of 3 or less than three numbers given.

```
# Function to find sum of 3 numbers
def sum_(a, b, c):
    return a+b+c
```

```
# default values are assigned when declaring the functions itself
def sum_(a=0, b=0, c=0):
    return a+b+c

#call the function with two arguments
sum_(2,3)

#output
5
```

```
# function to find the product of 3 or less numbers
def prod(a=1, b=1, c=1):
    return a* b* c

#call the function with two arguments
prod(2,3)
#output
6
```

```
# print the name and country
def details_1():
    name = input("please write your name: ")
    country = input("please write your country:  ")
    print(name, country)

# function calling
details_1()
```

Output:

```
please write your name: PWskills
please write your country:  India

PWskills India
```

Example2: (return)

```
def details_2():
    name = input("please write your name: ")
    country = input("please write your country:  ")
    return name, country

# function calling
details_2()
```

Output:

```
please write your name: PWskills
please write your country:  India

('PWskills', 'India')
```

Both look like they are giving the same answer but in actual for 'return' statement the output is in tuple and we can use it further.

```
# you can even use tuple unpacking using return statement
name_, country_ = details_2()

print(name_, country_)
```

Output:

```
please write your name: PWskills
please write your country:  India
PWskills India
```

Variable length args:

- Variable-length arguments refer to the ability of a function to accept a variable number of arguments.
- At some point there may be a chance that you don't know how many args that you should pass.
- Variable-length arguments allow a function to accept a varying number of arguments. There are two types of variable-length arguments: *args and **kwargs.

1. *args (Arbitrary Arguments):

- It allows a function to accept any number of positional arguments.
- The asterisk (*) before the parameter name is used to denote that it can take multiple arguments.
- The arguments passed with *args are treated as a tuple.

Example1

```
def my_function(*args):
    for arg in args:
        print(arg)

my_function(1, 2, 3)

# Output: 1
#         2
#         3
```

Example2

```
# print the args whatever length it may be
def print_(*args):
    for i in args:
        print(i)

print_(2,3,5,6,8)  #calling function




#output
2
3
5
6
8
```

Example2

```
# find the sum of 2 or more numbers
def sum_(*args):
    s = 0
    for i in args:
        s += i
    return s

# no matter the length of the input function returns the output
sum_(1,2,3,5,6,8,9) #function calling

#output
34
```

Example4:

```
# Another example
def branch_and_subjects_in_graduation(*args, branch):
    print(f"My branch was {branch}")
    return f"I liked these subjects in graduation {args}"

results = branch_and_subjects_in_graduation("Digital Image Processing",
"Microprocessor", branch="Electronics engineering")
print(results)

#output
My branch was Electronics engineering
I liked these subjects in graduation ('Digital Image Processing',
'Microprocessor')
```

Example5:

```
def branch_and_subjects_in_graduation(*args, branch="Electronics engineering"):
    print(f"My branch was {branch}")
    return f"I liked these subjects in graduation {args}"

results = branch_and_subjects_in_graduation("Digital Image Processing",
"Microprocessor")
print(results)

#output
My branch was Electronics engineering
I liked these subjects in graduation ('Digital Image Processing',
'Microprocessor')
```

2. **kwargs (Keyword Arguments):

- It allows a function to accept any number of keyword arguments.
- The double asterisk (**) before the parameter name is used to denote that it can take multiple keyword arguments.
- The arguments passed with **kwargs are treated as a dictionary.

Example1:

```
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

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

```
def marks_in_subjects_of_semester(**kwargs):
    print(kwargs)

marks_in_subjects_of_semester(Digital_Image_Processing=78, Microprocessor= 79,
Signals_and_systems=83)
#output

{'Digital_Image_Processing': 78, 'Microprocessor': 79, 'Signals_and_systems':
83}
```

```
def marks_in_subjects_of_semester(**kwargs):

    # you can access the keyward arguments using tuple unpacking also by
kwargs.items()
    for subject, marks in kwargs.items():
        print(f"Score in {subject} = {marks}")

marks_in_subjects_of_semester(Digital_Image_Processing = 78, Microprocessor= 79,
Signals_and_systems=83)

#output

Score in Digital_Image_Processing = 78
Score in Microprocessor = 79
Score in Signals_and_systems = 83
```

8. Iterator Function:

Iterator in python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The iterator object is initialized using the `iter()` method. It uses the `next()` method for iteration.

Example1:

```
# Here is an example of a python inbuilt iterator
# value can be anything which can be iterate
iterable_value = 'PWskills'
iterable_obj = iter(iterable_value)

while True:
 try:

  # Iterate by calling next
  item = next(iterable_obj)
  print(item)
 except StopIteration:
  # exception will happen when iteration will over

  break
```

```
#output

P
W
s
k
i
l
l
s
```

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))

#output
apple
```

Example2:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))

#output
apple
```

```
print(next(myit))

#output
banana
```

```
print(next(myit))

#output
cherry
```

```
mystr = "PWskills"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))

#output
P
W
s
k
i
l
l
s
```

Example3:

```
# Sample built-in iterators

# Iterating over a list
print("List Iteration")
l = ["I'm", 'from', 'PWskills']
for i in l:
 print(i)
# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = (['Im', 'from', 'PWskills'])
for i in t:
 print(i)
```

```
# Iterating over a String
print("\nString Iteration")
s = "PWskills"
for i in s :
 print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
 print(f"{i} {d[i]}")

#output

List Iteration
I'm
from
PWskills

Tuple Iteration
Im
from
PWskills

String Iteration
P
W
s
k
i
l
l
s

Dictionary Iteration
xyz 123
abc 345
```

Example4:

Here the following class iterates from 10 to the given limit.

```
# An iterable user defined type
# OOPs will be discussed in further classes

class Test:

 # Constructor
 def __init__(self, limit):
  self.limit = limit

 # Creates iterator object
 # Called when iteration is initialized
 def __iter__(self):
  self.x = 10
  return self

 # To move to next element. In Python 3,
 # we should replace next with __next__
 def __next__(self):

  # Store current value ofx
  x = self.x

  # Stop iteration if limit is reached
  if x > self.limit:
   raise StopIteration

  # Else increment and return old value
  self.x = x + 1
  return x


# Prints numbers from 10 to 15
for i in Test(15):
 print(i)



#output

10
11
12
13
14
15

# Prints nothing when the limit is between 0 to 9
for i in Test(9):
 print(i)
```

class that stops after first 20 iterations

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)


#output

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```