Functions

Introduction to Functions

A Function is a sequence of statements/instructions that performs a particular task. A function is like a black box that can take certain input(s) as its parameters and can output a value after performing a few operations on the parameters. A function is created so that one can use a block of code as many times as needed just by using the name of the function.

Why Do We Need Functions?

- **Reusability:** Once a function is defined, it can be used over and over again. You can call the function as many times as it is needed. Suppose you are required to find out the area of a circle for 10 different radii. Now, you can either write the formula (π*r*r) 10 times or you can simply create a function that takes the value of the radius as an input and returns the area corresponding to that radius. This way you would not have to write the same code (formula) 10 times. You can simply invoke the function every time.
- **Neat code:** A code containing functions is concise and easy to read.
- **Modularisation:** Functions help in modularizing code. Modularization means dividing the code into smaller modules, each performing a specific task.
- **Easy Debugging:** It is easy to find and correct the error in a function as compared to raw code.

Defining Functions In Python A function, once defined, can be invoked as many times as needed by using its name, without having to rewrite its code.
A function in Python is defined as per the following syntax:

```
def <function-name>(<parameters>):
     """ Function's docstring """
     <Expressions/Statements/Instructions>
```

- Function blocks begin with the keyword **def** followed by the **function name** and **parentheses** (( )).
- The input **parameters** or **arguments** should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function is optional - the documentation string of the function or **docstring**. The **docstring** describes the functionality of a function.
- The code block within every function starts with a **colon** (:) and is **indented**. All statements within the same code block are at the same indentation level.
- The **return** statement exits a function, optionally passing back an expression/value to the function caller.

Let us define a function to add two numbers.

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

The above function returns the sum of two numbers **a** and **b**.

The return Statement

A **return** statement is used to end the execution of the function call and it "returns" the result (value of the expression following the **return** keyword) to the caller. The statements after the return statements are not executed. If the **return** statement is without any expression, then the special value **None** is returned.

In the example given above, the sum a+b is returned.

Note: In Python, you need not specify the return type i.e. the data type of the returned value.

Calling/Invoking A Function

Once you have defined a function, you can call it from another function, program, or even the Python prompt. To use a function that has been defined earlier, you need to write a **function call**.

A **function call** takes the following form:

```
<function-name> (<value-to-be-passed-as-argument>)
```

The function definition does not execute the function body. The function gets executed only when it is called or invoked. To call the above function we can write:

```
add(5,7)
```

In this function call, a = 5 and b = 7.

Arguments And Parameters

As you know you can pass values to functions. For this, you define variables to receive values in the **function definition** and you send values via a function call statement. For example, in the **add()** function, we have variables **a** and **b** to receive the values and while calling the function we pass the values 5 and 7. We can define these two types of values:

- **Arguments:** The values being passed to the function from the function call statement are called arguments. Eg. **5** and **7** are arguments to the **add()** function.
- **Parameters:** The values received by the function as inputs are called parameters. Eg. **a** and **b** are the parameters of the **add()** function.

Types Of Functions

We can divide functions into the following two types:

1. User-defined functions: Functions that are defined by the users. Eg. The add() function we created.
2. Inbuilt Functions: Functions that are inbuilt in Python. Eg. The print() function.

Scope Of Variables

All variables in a program may not be accessible at all locations in that program. Part(s) of the program within which the variable name is legal and accessible, is called the scope of the variable. A variable will only be visible to and accessible by the code blocks in its scope.

There are broadly two kinds of scopes in Python −

- Global scope
- Local scope

Global Scope

A variable/name declared in the top-level segment **(__main__)** of a program is said to have a global scope and is usable inside the whole program (Can be accessed from anywhere in the program).

In Python, a variable declared outside a function is known as a global variable. This means that a global variable can be accessed from inside or outside of the function.

Local Scope

Variables that are defined inside a function body have a local scope. This means that local variables can be accessed only inside the function in which they are declared.

The Lifetime of a Variable

The lifetime of a variable is the time for which the variable exists in the memory.

- The lifetime of a Global variable is the entire program run (i.e. they live in the memory as long as the program is being executed).
- The lifetime of a Local variable is its function's run (i.e. as long as their function is being executed).

Creating a Global Variable

Consider the given code snippet:

```
x = "Global Variable"
def foo():
    print("Value of x: ", x)
foo()
```

Here, we created a global variable **x** = "Global Variable". Then, we created a function **foo** to print the value of the global variable from inside the function. We get the output as:

```
Global Variable
```

Thus we can conclude that we can access a global variable from inside any function.

What if you want to change the value of a Global Variable from inside a function?

Consider the code snippet:

```
x = "Global Variable"
def foo():
    x = x -1
    print(x)
foo()
```

In this code block, we tried to update the value of the global variable **x**. We get an output as:

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

This happens because, when the command `x=x-1`, is interpreted, Python treats this **x** as a local variable and we have not defined any local variable **x** inside the function **foo()**.

Creating a Local Variable

We declare a local variable inside a function. Consider the given function definition:

```
def foo():
    y = "Local Variable"
    print(y)
foo()
```

We get the output as:

```
Local Variable
```

Accessing A Local Variable Outside The Scope

```
def foo():
    y = "local"
foo()
print(y)
```

In the above code, we declared a local variable **y** inside the function **foo()**, and then we tried to access it from outside the function. We get the output as:

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

We get an error because the lifetime of a local variable is the function it is defined in. Outside the function, the variable does not exist and cannot be accessed. In other words, a variable cannot be accessed outside its scope.

Global Variable And Local Variable With The Same Name

Consider the code given:

```
x = 5
def foo():
    x = 10
    print("Local:", x)
foo()
print("Global:", x)
```

In this, we have declared a global variable **x = 5** outside the function **foo()**. Now, inside the function **foo()**, we re-declared a local variable with the same name, **x**. Now, we try to print the values of **x**, inside, and outside the function. We observe the following output:

```
Local: 10
Global: 5
```

In the above code, we used the same name **x** for both global and local variables. We get a different result when we print the value of **x** because the variables have been declared in different scopes, i.e. the local scope inside **foo()** and the global scope outside **foo()**.

When we print the value of the variable inside **foo()** it outputs **Local: 10**. This is called the local scope of the variable. In the local scope, it prints the value that it has been assigned inside the function.

Similarly, when we print the variable outside **foo()**, it outputs global **Global: 5**. This is called the global scope of the variable, and the value of the global variable **x** is printed.

Python Default Parameters

Function parameters can have default values in Python. We can provide a default value to a parameter by using the assignment operator (=). Here is an example.

```
def wish(name, wish="Happy Birthday"):

"""This function wishes the person with the provided message.
If the message is not provided, it defaults to "Happy
Birthday" """

    print("Hello", name + ', ' + wish)

greet("Rohan")
greet("Hardik", "Happy New Year")
```

```
Hello Rohan, Happy Birthday
Hello Hardik, Happy New Year
```

Output

In this function, the parameter **name** does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter **wish** has a default value of "Happy Birthday". So, it is optional during a call. If an argument is passed corresponding to the parameter, it will overwrite the default value, otherwise it will use the default value.

Important Points to be kept in mind while using default parameters:

- Any number of parameters in a function can have a default value.
- The conventional syntax for using default parameters states that once we have passed a default parameter, all the parameters to its right must also have default values.

- In other words, non-default parameters cannot follow default parameters.

For example, if we had defined the function header as:

```
def wish(wish = "Happy Birthday", name):
...
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Thus to summarise, in a function header, any parameter can have a default value unless all the parameters to its right have their default values.

Variable Arguments

In Python, variable arguments allow a function to accept an arbitrary number of arguments. There are two main ways to implement variable arguments in Python:

- ***args:** The *args parameter allows a function to accept any number of positional arguments. When the function is called, all the positional arguments passed are collected into a tuple, which can then be accessed within the function using the args parameter.
  Example:

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

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

- ****kwargs**: The **kwargs parameter allows a function to accept any number of keyword arguments as a dictionary. When the function is called, all the keyword arguments passed are collected into a dictionary, which can then be accessed within the function using the kwargs parameter.
  Example:

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

```
my_function(a=1, b=2, c=3)  # Output: a 1, b 2, c 3
my_function(x='foo', y='bar', z='baz')  # Output: x foo,
y bar, z baz
```

Combining both *args and **kwargs allows a function to accept both positional and keyword arguments simultaneously.
It's important to note that the parameter names (args and kwargs) are just conventions and can be named differently, but using these names makes the code more readable and understandable to other Python developers.

Built-in functions

Python comes with a rich set of built-in functions that are readily available for use without the need for importing any external modules. These built-in functions cover a wide range of tasks, from mathematical operations to string manipulation, data type conversion, and more. Here are some commonly used built-in functions in Python:

1. Mathematical Functions:
    - abs(): Returns the absolute value of a number.
    - pow(): Returns the power of a number.
    - round(): Rounds a number to a specified number of decimal places.
2. Type Conversion Functions:
    - int(): Converts a number or string to an integer.
    - float(): Converts a number or string to a floating-point number.
    - str(): Converts an object to a string.
3. Sequence Functions:
    - len(): Returns the length of a sequence (e.g., string, list, tuple).
    - max(): Returns the maximum value in a sequence.
    - min(): Returns the minimum value in a sequence.
4. Iterable Functions:
    - map(): Applies a function to all items in an iterable.
    - filter(): Filters elements from an iterable based on a function.
    - sorted(): Returns a new sorted list from the elements of any iterable.
5. String Functions:

- str.upper(), str.lower(): Converts a string to uppercase or lowercase.
- str.capitalize(): Converts the first character of a string to uppercase.
- str.split(): Splits a string into a list of substrings based on a delimiter.

6. Input/Output Functions:
    - print(): Prints objects to the standard output.
    - input(): Reads input from the user via the console.
7. File Handling Functions:
    - open(): Opens a file and returns a file object.
    - read(), readline(), readlines(): Reads data from a file.
    - write(): Writes data to a file.
8. Other Utility Functions:
    - range(): Generates a sequence of numbers.
    - enumerate(): Returns an enumerate object containing tuples of indices and values.
    - type(): Returns the type of an object.

These are just a few examples of the many built-in functions available in Python. You can find the complete list of built-in functions in the Python documentation: https://docs.python.org/3/library/functions.html

Problem Statement: Check Palindrome

A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward. In other words, if you reverse the characters of a palindrome, you get the same sequence.

For example:

- "radar" is a palindrome because if you reverse its letters, you get "radar".
- "level" is a palindrome because if you reverse its letters, you get "level".
- "12321" is a palindrome because if you reverse its digits, you get "12321".

Approach :

1. **reverse_of_number(num) function**:
    - Initialize res to 0, which will store the reversed number.

- Initialize pow_of_10 to 1, which represents 10^0 (1), indicating the current place value multiplier.
- Use a while loop to iterate until the num is greater than 0.
- Inside the loop:
    - Get the last digit of num using the modulo (%) operator and store it in variable x.
    - Update res by multiplying it by 10 (to shift digits one place to the left) and adding x, effectively appending x to the reversed number.
    - Update num by integer division (//) by 10, effectively removing the last digit.
- Finally, return the reversed number stored in res.

2. **check_palindrome(num) function:**
    - Call the reverse_of_number() function to get the reversed form of the input number num. Store this reversed number in the variable reverse.
    - Check if the reverse is equal to the original number num.
    - Return True if they are equal, indicating that num is a palindrome. Otherwise, return False.

```
def reverse_of_number(num):
    res = 0
    pow_of_10 = 1 # 10^0
    while num > 0:
        x = num % 10 # last digit
        res = res * 10 + x
        num = num // 10
    return res

def check_palindrome(num):
    reverse = reverse_of_number(num)
    return reverse == num
```

Problem Statement:check_armstrong

An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits. For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

Approach:

1. **count_digits(n) function**:
    - Initialize the result to 0.
    - Use a while loop to iterate until n is greater than 0.
    - Inside the loop, increment the result by 1 for each iteration, indicating that one more digit has been encountered.
    - Update n by integer division (//) by 10, effectively removing the last digit of n.
    - Finally, return the count of digits stored in the result.
2. **check_armstrong(num) function:**
    - First, call the count_digits() function to get the count of digits in the number num. Store this count in variable d.
    - Initialize sum_of_digit_power to 0 to store the sum of the digits raised to the power of d.
    - Store the original value of num in a temporary variable temp.
    - Use a while loop to iterate until num is greater than 0.
    - Inside the loop:
        - Get the last digit of num using the modulo (%) operator and store it in variable x.
        - Add x raised to the power of d to sum_of_digit_power.
        - Update num by integer division (//) by 10, effectively removing the last digit.
    - Finally, return True if sum_of_digit_power is equal to the original number temp, indicating that num is an Armstrong number. Otherwise, return False.

```
def count_digits(n):
    result = 0
    while n > 0:
        result += 1 # we have one more digit
        n = n // 10

    return result

def check_armstrong(num):
    d = count_digits(num) # gives us count of the digits
    sum_of_digit_power = 0
    temp = num # we will temporarily store num
```

```
    while num > 0:
        x = num % 10 # x is the last digit of current value
of num
        sum_of_digit_power += x**d
        num = num // 10

    return sum_of_digit_power == temp
```