Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

Functions

```
1) def function1():

   print("text1")

   print("text2")


print("outside function")

function1()

function1()


2) def function2(x)

          return 2*x

    a=function2(3)

    print(a)

    b=function2()
```

```
3) def function3(x, y)

        return x+y

    e=function3(4,3)

    print(e)



def my_func():

      x = 10

      print("Value inside function:",x)

x = 20

my_func()

print("Value outside function:",x)
```

Output

```
Value inside function: 10

Value outside function: 20
```

Here, we can see that the value of `x` is 20 initially. Even though the function `my_func()` changed the value of `x` to 10, it did not effect the value outside the function.

This is because the variable `x` inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword `global`.

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Python Built-in Function

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, `print()` function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are 68 built-in functions. They are listed below alphabetically along with brief description.

Method	Description
Python abs()	returns absolute value of a number
Python all()	returns true when all elements in iterable is true
Python any()	Checks if any Element of an Iterable is True
Python ascii()	Returns String Containing Printable Representation
Python bin()	converts integer to binary string
Python bool()	Converts a Value to Boolean
Python bytearray()	returns array of given byte size
Python bytes()	returns immutable bytes object

Examples:

String Methods

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Example

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Example

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

Learn more about String Methods with our String Methods Reference

---

Check String

To check if a certain phrase or character is present in a string, we can use the keywords `in` or `not in`.

Example

Check if the phrase "ain" is present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

Example

Check if the phrase "ain" is NOT present in the following text:

```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" not in txt
print(x)
```

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable `a` with variable `b` into variable `c`:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Example

To add a space between them, add a `" "`:

```
a = "Hello"
b = "World"
```

```
c = a + " " + b
print(c)
```

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash `\` followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Python User-defined Functions

In this tutorial, you will find the advantages of using user-defined functions and best practices to follow.

What are user-defined functions in Python?

Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed.

Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.

3. Programmars working on large project can divide the workload by making different functions.

Example of a user-defined function

-
```
# Program to illustrate
# the use of user-defined functions

def add_numbers(x,y):
    sum = x + y
    return sum

num1 = 5
num2 = 6

print("The sum is", add_numbers(num1, num2))
```

Python Function Arguments

In Python, you can define a function that takes variable number of arguments. You will learn to define such functions using default, keyword and arbitrary arguments in this article.

Arguments

In user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result into an error. Here is an example.

```
def greet(name,msg):
    """This function greets to
    the person with the provided message"""
    print("Hello",name + ', ' + msg)

greet("Monica","Good morning!")
```

Output

```
Hello Monica, Good morning!
```

Here, the function `greet()` has two parameters.

Since, we have called this function with two arguments, it runs smoothly and we do not get any error.