functions_demo

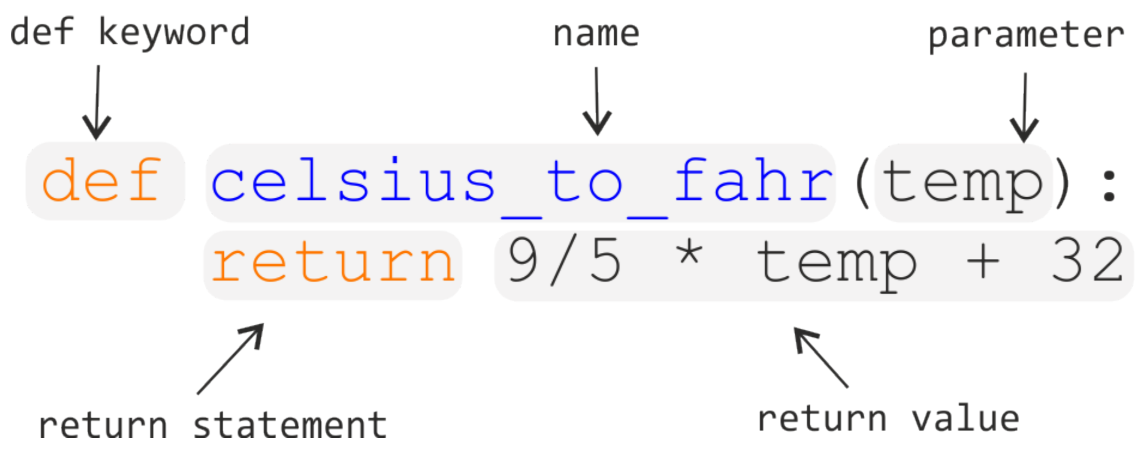
October 8, 2019

Functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions.

```
In [20]: from IPython.display import Image
         Image(filename='/home/rohan/pythonfuncFunction_anatomy.png',height=500,width=600)
```

```
Out[20]:
```



The function definition opens with the keyword def followed by the name of the function (fahr_to_celsius) and a parenthesized list of parameter names (temp). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a return keyword followed by the return value.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

```
In [15]: def fahr_to_celsius(temp):
             return ((temp - 32) * (5/9))
```

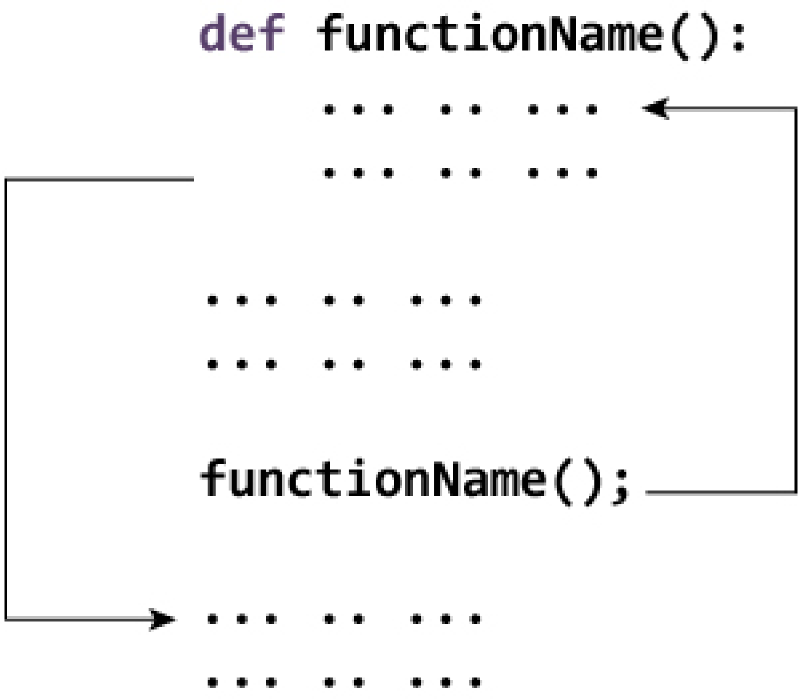
```
In [16]: print('freezing point of water:', fahr_to_celsius(32), 'C')
         print('boiling point of water:', fahr_to_celsius(212), 'C')
```

```
freezing point of water: 0.0 C
boiling point of water: 100.0 C
```

Pass by Reference or pass by value?
How Function works in Python?

```
In [22]: from IPython.display import Image
         Image(filename='/home/rohan/python-how-function-works_1.jpg',height=400,width=400)
```

Out[22]:



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.

Example

```
In [23]: def my_func():
             x = 10
```

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

        x = 20
        my_func()
        print("Value outside function:",x)
```

```
Value inside function: 10
Value outside function: 20
```

Rules of global Keyword
The basic rules for global keyword in Python are:

- When we create a variable inside a function, it's local by default.
- When we define a variable outside of a function, it's global by default.(no need of use global keyword).
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect

```
In [31]: c = 1 # global variable

         def add():
             print(c)

         add()
```

```
1
```

```
In [34]: c = 1 # global variable

         def add():
             global c
             c = c + 2 # increment c by 2
             print(c)

         add()
```

```
3
```

Python Default Arguments
We can provide a default value to an argument by using the assignment operator (=)

```
In [26]: def greet(name, msg = "Good morning!"):
            print("Hello",name + ', ' + msg)

         greet("Tulasi")
         greet("Rohan","How do you do?")
```

```
Hello Tulasi, Good morning!
Hello Rohan, How do you do?
```

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 msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

Python Keyword Arguments

```
In [28]: # 2 keyword arguments
         greet(name = "Rohan",msg = "How do you do?")

         # 2 keyword arguments (out of order)
         greet(msg = "How do you do?",name = "Rohan")

         # 1 positional, 1 keyword argument
         greet("Rohan",msg = "How do you do?")
```

```
Hello Rohan, How do you do?
Hello Rohan, How do you do?
Hello Rohan, How do you do?
```

Having a positional argument after keyword arguments will result into errors. For example the function call as follows:

```
In [29]: greet(name="Rohan","How do you do?")
```

```
          File "<ipython-input-29-8af78a1999ef>", line 1
        greet(name="Rohan","How do you do?")
                          ^
    SyntaxError: positional argument follows keyword argument
```

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

```
In [30]: def greet(*names):
            # names is a tuple with arguments
            for name in names:
```

```
            print("Hello",name)

    greet("Tulasi","Rohan","Pradeep","Kumar")
```

```
Hello Tulasi
Hello Rohan
Hello Pradeep
Hello Kumar
```

lambda functions

In Python, anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called lambda functions.

Syntax of Lambda Function in python

lambda arguments: expression

```
In [35]: double = lambda x: x * 2

         # Output: 10
         print(double(5))
```

```
10
```

Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

```
In [36]: my_list = [1, 5, 4, 6, 8, 11, 3, 12]

         new_list = list(filter(lambda x: (x%2 == 0) , my_list))

         # Output: [4, 6, 8, 12]
         print(new_list)
```

```
[4, 6, 8, 12]
```

```
In [37]: my_list = [1, 5, 4, 6, 8, 11, 3, 12]

         new_list = list(map(lambda x: x * 2 , my_list))

         # Output: [2, 10, 8, 12, 16, 22, 6, 24]
         print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Python Recursion

What is recursion in Python?

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

```
In [40]: def calc_factorial(x):
             """This is a recursive function
             to find the factorial of an integer"""

             if x == 1:
                 return 1
             else:
                 return (x * calc_factorial(x-1))

         num = 5
         print("The factorial of", num, "is", calc_factorial(num))
```

```
The factorial of 5 is 120
```

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

```
In [42]: def bubble_sort(arr):
             def swap(i, j):
                 arr[i], arr[j] = arr[j], arr[i]

             n = len(arr)
             swapped = True

             x = -1
             while swapped:
                 swapped = False
                 x = x + 1
```

```
                for i in range(1, n-x):
                    if arr[i - 1] > arr[i]:
                        swap(i - 1, i)
                        swapped = True

            return arr
```

```
In [44]: my_list = [3,2,4,1,9,8,6,7,5]
         bubble_sort(my_list)
```

```
Out[44]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [45]: def selection_sort(arr):
             for i in range(len(arr)):
                 minimum = i

                 for j in range(i + 1, len(arr)):
                     # Select the smallest value
                     if arr[j] < arr[minimum]:
                         minimum = j

                 # Place it at the front of the
                 # sorted end of the array
                 arr[minimum], arr[i] = arr[i], arr[minimum]

             return arr
```

```
In [46]: selection_sort(my_list)
```

```
Out[46]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [47]: def insertion_sort(arr):

             for i in range(len(arr)):
                 cursor = arr[i]
                 pos = i

                 while pos > 0 and arr[pos - 1] > cursor:
                     # Swap the number down the list
                     arr[pos] = arr[pos - 1]
                     pos = pos - 1
                 # Break and do the final swap
                 arr[pos] = cursor

             return arr
```

```
In [48]: insertion_sort(my_list)
```

```
Out[48]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [49]: def merge_sort(arr):
             # The last array split
             if len(arr) <= 1:
                 return arr
             mid = len(arr) // 2
             # Perform merge_sort recursively on both halves
             left, right = merge_sort(arr[:mid]), merge_sort(arr[mid:])

             # Merge each side together
             return merge(left, right, arr.copy())


         def merge(left, right, merged):

             left_cursor, right_cursor = 0, 0
             while left_cursor < len(left) and right_cursor < len(right):

                 # Sort each one and place into the result
                 if left[left_cursor] <= right[right_cursor]:
                     merged[left_cursor+right_cursor]=left[left_cursor]
                     left_cursor += 1
                 else:
                     merged[left_cursor + right_cursor] = right[right_cursor]
                     right_cursor += 1

             for left_cursor in range(left_cursor, len(left)):
                 merged[left_cursor + right_cursor] = left[left_cursor]

             for right_cursor in range(right_cursor, len(right)):
                 merged[left_cursor + right_cursor] = right[right_cursor]

             return merged

In [50]: merge_sort(my_list)

Out[50]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```