Sharif University of Technology
Department of Computer Engineering

Fundamentals of Programming

Python Language

Arman Malekzadeh
PhD Candidate in Artificial Intelligence

Table of contents

Functions

Functions, Parameters and Arguments in Python

- A **function** is a block of code that can be reused and called at any time.
- **Parameters** are values that are passed into a function when it is called.
- **Arguments** are the actual values that are passed into the function when it is called.
- The number of parameters and arguments must match when calling a function.
- Parameters and arguments can have default values to make them optional.

Parameters and Arguments of a Function in Python

- Parameters are the names used when defining a function.
- Arguments are the values passed to the function when it is called.

```
def add(x, y): # x and y are parameters
    return x + y
add(4, 5) # 4 and 5 are arguments
# Output: 9
```

Example: Sum of Even Numbers in a List

```
def sum_even_numbers(num_list):
    total = 0
    for num in num_list:
        if num % 2 == 0:
            total += num
    return total

numbers = [1, 2, 3, 4, 5, 6]
print("The sum of even numbers is:", sum_of_even_numbers(numbers))
```

Python Functions with Default Parameters

```python
def greet(name="World"):
    return "Hello, " + name

print(greet())  # Hello, World
print(greet("Alice")) # Hello, Alice
```

```python
def greet(first_name, second_name, postfix="!"):
    return "Hello, " + first_name + " " + second_name + postfix

print(greet("Alice", "Smith")) # Hello, Alice Smith!
print(greet("Alice", "Smith", "!!!")) # Hello, Alice Smith!!!
```

Recursive Function Example: Factorial

```
def factorial(n):
"""This function returns the factorial of a number using recursion"""
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
def fibonacci(n):
    if n <= 0:
        return "Input should be a positive integer."
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

First-Class Function in Python

A first-class function is a function that can be passed as an argument, returned as a value from other functions and assigned to variables. Here is an example:

```
def square(x):
    return x * x

def cube(x):
    return x * x * x

def calculate(func, num):
    return func(num)

squared = calculate(square, 5) # returns 25
cubed = calculate(cube, 5) # returns 125
```

In this example, `calculate` is a higher-order function that takes two arguments: a function (`func`) and a number (`num`). It then calls the provided function with the number as an argument. In Python3, all functions are first-class.

Higher-Order Function Example

A higher-order function is a function that takes one or more functions as arguments, or returns a function as its result. Here is an example in Python:

```
def apply_func(func, value):
    return func(value)

def double(x):
    return x * 2

result = apply_func(double, 5)
print(result) # Output: 10
```

In this example, the function `apply_func` is a higher-order function because it accepts another function `double` as an argument.

Lambda Function in Python

```
x = lambda a : a + 10
print(x(5))
```

In the given python code, we have defined a lambda function that takes one argument `a` and returns `a+10`. When we call this function with a value of 5, it returns 15.

```
add = lambda x, y: x + y
print(add(5, 3)) # Outputs: 8
```

This lambda function takes two arguments (x and y) and returns their sum.

`*args` Example

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

my_function('Hello', 'from', 'Python')
```

This function will print all arguments it receives. `*args` is used to pass a non-keyworded, variable-length argument list to your function.

```
def sum_all(*args):
    sum = 0
    for num in args:
        sum += num
    return sum

print(sum_all(1, 2, 3, 4, 5)) # prints: 15
```

Example: Iterating Over a Dictionary

```
my_dict = {'ali': 15, 'maryam': 17, 'hesam': 20, 'sara': 19, 'mohammad': 16
                                                }

for key, value in my_dict.items():
    print(f"The key is {key} and the value is {value}")
```

This Python code creates a dictionary called `my_dict` with three key-value pairs. It then uses the `items()` method to iterate over each pair, printing both the key and the value.

**kwargs Example

```
def example_function(**kwargs):
    for key, value in kwargs.items():
        print(f"The value of {key} is {value}")

example_function(First_Name="John", Last_Name="Doe")
```

In this example, we have a Python function `example_function` which accepts any number of keyword arguments (`**kwargs`). The function then iterates over these arguments, printing out the key and corresponding value for each one. We then call this function with two keyword arguments: `First_Name` and `Last_Name`.

Example: Manipulating a list

```
def manipulate_list(input_lst):
    output_lst = []
    for idx, element in enumerate(input_lst):
        if idx == 0:
            pass # Used as a placeholder for future code
        else:
            if element % 2 == 0:
                output_lst.append(element)
            else:
                continue # Skips the current iteration of a loop and moves
                                                        on to the next one.
    return output_lst
```

Python Function Parameter Types

In Python, types of function parameters aren't explicitly mentioned in their definition. However, from Python 3.5 onwards, we can use *type hints* to indicate the expected types.

```
def greet(name: str) -> str:
    return 'Hello ' + name
```

- In this example, `name: str` indicates that the `name` parameter should be a string (`str`), and `-> str` indicates that the function should return a string.
- Remember that these are just hints; Python won't enforce them. They serve as documentation and for tooling support.

Calling by Reference vs Calling by Value

- **Calling by Value:** The method copies the value of an argument into a formal parameter of the function. In Python, primitives like integers and strings are passed by value.

```
def change_value(num):
    num = 10
    print("Inside function:", num) # Inside function: 10

num = 20
change_value(num)
print("Outside function:", num) # Outside function: 20
```

Calling by Reference vs Calling by Value

- **Calling by Reference:** The method copies the address of an argument into the formal parameter. In Python, objects like lists are passed by reference.

```
def change_list(my_list):
    my_list.append([1, 2, 3])
    print("Inside function:", my_list) # Inside function: [10, 20, 30, [1, 2, 3
                                                          ]]

my_list = [10, 20, 30]
change_list(my_list)
print("Outside function:", my_list) # Outside function: [10, 20, 30, [1, 2, 3]]
```

Python uses a system which might be best described as "pass-by-object-reference". When you pass an argument into a function in Python, what is passed is actually a reference to that object.

Python Data Types and Their Mutability

Data Type	Mutability
Integer	Immutable
Float	Immutable
Complex	Immutable
Boolean	Immutable
String	Immutable
Tuple	Immutable
Frozen Set	Immutable
List	Mutable
Set	Mutable
Dictionary	Mutable

Table: Python data types and their mutability

References

References I

[1] B Downey, A. (2015). Think Python: How to Think Like a Computer Scientist-2nd Edition.

[2] Deitel, H. M., & Deitel, P. J. (2004). C: How to program. Pearson Educacion.

Sharif University of Technology
Department of Computer Engineering

Arman Malekzadeh

Fundamentals of Programming

Python Language