Chapter 6: Functions

Functions are blocks of code identified by a name, which can receive predetermined parameters.

In Python, functions:

- Can return objects or not.
- Accept *Doc Strings*.
- Accept optional parameters (with *defaults*). If no parameter is passed, it will be equal to the *default* defined in the function.
- Accepts parameters to be passed by name. In this case, the order in which the parameters were passed does not matter.
- Have their own namespace (local scope), and therefore may obscure definitions of global scope.
- Can have their properties changed (usually by decorators).

Doc Strings are strings that are attached to a Python structure. In functions, *Doc strings* are placed within the body of the function, usually at the beginning. The goal of *Doc Strings* is to be used as documentation for this structure.

Syntax:

```
def func(parameter1, parameter2=default_value):
    """
    Doc String
    """
    <code block>
    return value
```

The parameters with *default* value must be declared after the ones without *default* value.

Example (factorial without recursion):

```
def fatorial(n):
    n = n if n > 1 else 1
    j = 1
    for i in range(1, n + 1):
        j = j * i
```

```
    return j


# Testing...
for i in range(1, 6):
    print (i, '->', fatorial(i))


1 -> 1
2 -> 2
3 -> 6
4 -> 24
5 -> 120
```

Example (factorial with recursion):

```
def factorial(num):
    """Fatorial implemented with recursion."""
    if num <= 1:
        return 1
    else:
        return(num * factorial(num - 1))



# Testing factorial()
print (factorial(5))

# 5 * (4 * (3 * (2) * (1))


120
```

Example (Fibonacci series with recursion):

```
def fib(n):
    """Fibonacci:
    fib(n) = fib(n - 1) + fib(n - 2) se n > 1
    fib(n) = 1 se n <= 1
    """
    if n > 1:
        return fib(n - 1) + fib(n - 2)
    else:
        return 1

# Show Fibonacci from 1 to 5
```

```
for i in [1, 2, 3, 4, 5]:
    print (i, '=>', fib(i))
```

```
1 => 1
2 => 2
3 => 3
4 => 5
5 => 8
```

Example (Fibonacci series without recursion):

```
def fib(n):
    # the first two values
    l = [1, 1]

    # Calculating the others
    for i in range(2, n + 1):
        l.append(l[i -1] + l[i - 2])

    return l[n]

# Show Fibonacci from 1 to 5
for i in [1, 2, 3, 4, 5]:
    print (i, '=>', fib(i))
```

```
1 => 1
2 => 2
3 => 3
4 => 5
5 => 8
```

```
def test(a, b):
    print(a, b)
    return a + b

print(test(1,2))
test(b=1,a=2)
```

```
1 2
3
2 1
```

3

```
def test_new(a, b, c):
    pass


def test(a, b):
    print(a, b)
    return a*a, b*b

x, a = test(2 , 5)

print(x)
print(type(x))
print(a)
print(type(a))
```

```
2 5
4
<class 'int'>
25
<class 'int'>
```

```
def test(a, b):
    print(a, b)
    return a*a, b*b, a*b

x, a = test(2 , 5)

print(x)
print(type(x))
print(a)
print(type(a))


def test(a, b):
    print(a, b)
    return a*a, b*b, "asdf"

x = test(2 , 5)

print(x)
```

```
print(type(x))
```

```
2 5
(4, 25, 'asdf')
<class 'tuple'>
```

```
def test(a=100, b=1000):
    print(a, b)
    return a, b

x = test(2, 5)
print(x)
print(test(10))
```

```
2 5
(2, 5)
10 1000
(10, 1000)
```

```
def test(a=100, b=1000):
    print(a, b)
    return a, b

print(test(b=10))
```

```
100 10
(100, 10)
```

```
def test(d, c, a=100, b=1000):
    print(d, c, a, b)
    return d, c, a, b

x = test(c=2, d=10, b=5)
print(x)
x = test(1, 2, 3, 4)
print(x)
print(test(10, 2))
```

```
10 2 100 5
(10, 2, 100, 5)
1 2 3 4
```

```
(1, 2, 3, 4)
10 2 100 1000
(10, 2, 100, 1000)
```

Example (RGB conversion):

```
def rgb_html(r=0, g=0, b=0):
    """Converts R, G, B to #RRGGBB"""

    return '#%02x%02x%02x' % (r, g, b)

def html_rgb(color='#000000'):
    """Converts #RRGGBB em R, G, B"""

    if color.startswith('#'): color = color[1:]

    r = int(color[:2], 16)
    g = int(color[2:4], 16)
    b = int(color[4:], 16)

    return r, g, b # a sequence

print (rgb_html(200, 200, 255))
print (rgb_html(b=200, g=200, r=255)) # what's happened?
print (html_rgb('#c8c8ff'))
```

```
#c8c8ff
#ffc8c8
(200, 200, 255)
```

Observations:

- The arguments with default value must come last, after the non-default arguments.
- The default value for a parameter is calculated when the function is defined.
- The arguments passed without an identifier are received by the function in the form of a list.
- The arguments passed to the function with an identifier are received in the form of a dictionary.
- The parameters passed to the function with an identifier should come at the end of the parameter list.

Example of how to get all parameters:

```
# *args - arguments without name (list)
# **kargs - arguments with name (ditcionary)

def func(*args, **kargs):
    print (args)
    print (kargs)

func('weigh', 10, unit='k')
```

```
('weigh', 10)
{'unit': 'k'}
```

In the example, `kargs` will receive the named arguments and `args` will receive the others.

The interpreter has some *builtin* functions defined, including `sorted()`, which orders sequences, and `cmp()`, which makes comparisons between two arguments and returns -1 if the first element is greater, 0 (zero) if they are equal, or 1 if the latter is higher. This function is used by the routine of ordering, a behavior that can be modified.

Example:

```
data = [(4, 3), (5, 1), (7, 2), (9, 0)]

# Comparing by the last element
def _cmp(x, y):
    return cmp(x[-1], y[-1])

print ('List:', data)
```

```
List: [(4, 3), (5, 1), (7, 2), (9, 0)]
```

Python also has a *builtin* function `eval()`, which evaluates code (source or object) and returns the value.

Example:

```
print (eval('12. / 2 + 3.3'))
```

```
9.3
```

```
def listing(lst):
    for l in lst:
        print(l)

d = {"mayank":40, "janki mohan johri":68}
listing(d)
```

```
mayank
janki mohan johri
```