Python - Functional Programming

FUNCTIONS

Contents

- Arguments Vs Parameters.
- Positional Arguments.
- Unpacking Iterable.
- *args (n-Positional arguments).

Functions

- Python's functions are **first-class objects**.
- We can assign them to
  - variables,
  - store them in data structures,
  - pass them as arguments to other functions,
  - return them as values from other functions.
- Types:
  - UDFs(User-Defined)
  - Built-Ins

Arguments Vs Parameters

- Semantics!

```
def my_func(a,b):
    #some code here
```

In this context, a and b are called **parameters** of **my_func**

a and b are variables, local to my_func

When we make a call to the function:

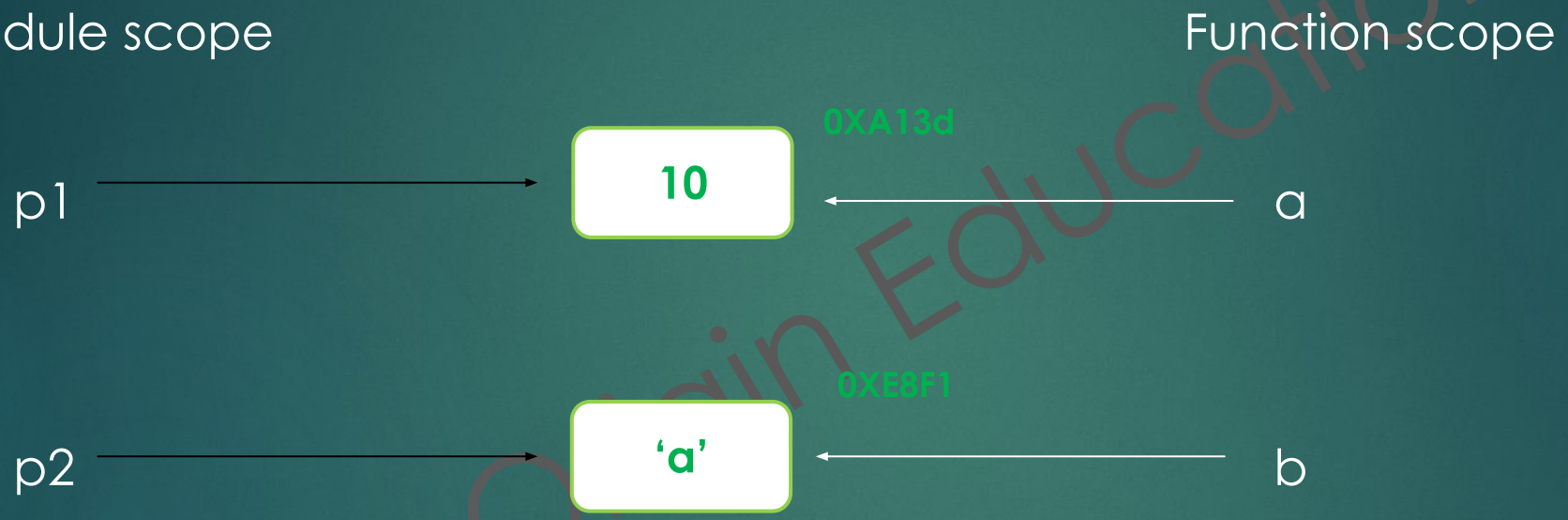
```
p1 = 10
p2 = 'a'
my_func(p1,p2)
```

In this context, p1 and p2 are called **arguments** of **my_func**

p1 and p2 are passed by reference

i.e. the memory addresses of p1 and p2 are passed

How its looks internally



Positional Arguments

- Way of assigning arguments to parameters: via the order in which they are passed (position)

```
def my_func(a,b):
    #some code here
```

If call the function then python does this

Case#1 : my_func(10, 20) → a = 10, b = 20

Case#2: my_func(20, 10) → a = 20, b = 10

Unpacking Iterable

- Packed Values:
- Packed values refers to values that are **bundled** together in some way.
- Tuples and Lists are obvious:
  - t = (1,2,3)
  - l = [1,2,3]
- Even a string is considered to be a packed value:
  - s = “HELLO”
- Sets and dictionaries are also packed values:
  - s1 = {1,2,3}
  - d1 = {1:1,2:2,3:3}

Note: any iterable can be considered a packed value.

How to Unpack the Packed value

- Unpacking is the act of splitting packed values into individual variables contained in a list or tuple.
- Example:

  a, b, c = [1, 2, 3] this is actually a tuple of 3 variables: a, b and c

  3 elements in [1, 2, 3] → need 3 variables to unpack.

  a → 1 b → 2 c → 3
- The unpacking into individual variables is based on the relative positions of each element.
- Does this remind you of how positional arguments were assigned to parameters in function calls?

Simple Application of Unpacking

- swapping values of two variables

a = 10 a = 20

b =20 b = 10

"**legacy** way "

```
tmp = a
a = b
b = tmp
```

"using **unpacking**"

```
a,b = b,a
```

This works because in Python, the entire RHS is evaluated first and completely then assignments are made to the LHS

Extended Unpacking

Use case for *

- We don't always want to unpack every single item in an iterable.
- We may, for example, want to unpack the first value, and then unpack the remaining values into another variable.
- l = [1, 2, 3, 4, 5, 6]
- How to achieve it we have below ways:
  - We can achieve this using slicing: a = l[0]
    b = l[1:]
  - simple unpacking/Parallel assignments: a, b = l[0], l[1:]
  - * operator : a, *b = l

Apart from cleaner syntax, it also works with any iterable, not just sequence types!

Extended Unpacking

- following also works:

```
a, b, *c = 1, 2, 3, 4, 5                a = 1 b = 2 c = [3, 4, 5]

a, b, *c, d = [1, 2, 3, 4, 5]           a = 1 b = 2 c = [3, 4] d = 5

a, *b, c, d = 'python'                  a = 'p'          c = 'o'
                                        b = ['y', 't', 'h']   d = 'n'
```

The * operator can only be used once in the LHS an unpacking assignment

*args

- The * parameter name is arbitrary – you can make it whatever you want
- It is customary (but not required) to name it *args
- Recall this:

```
a, b, *c = 10, 20, 'a', 'b'
```

→ a=10 b=20 c=['a', 'b']

```
def func1(a, b, *c):
    # code func1(10, 20, 'a', 'b')
```

- func1(10, 20, 'a', 'b') → a=10 b=20

  c=('a', 'b') this is a tuple, not a list

Note: *args **exhausts positional arguments**

Unpacking Arguments

```
def func1(a, b, c):
    # code

l = [10, 20, 30]
```

Lets unpack the list first and then pass it to the function:

```
func1(*l)
```

→ a = 10 b = 20 c = 30

send me your suggestions!

- Email: sundar.muthuraman.offical@gmail.com
- Contact me : +91 9962988838