

More on Functions

IC152 Lecture 9
Feb 2021

LEGB rule for Python scope

LEGB is the order for name lookup

Local:

Names defined inside the block

Created at function call, not at definition

name is looked:

First in local

Then in enclosing

Then in global

Then in built-in

Not found, then error

Enclosing or nonlocal scope:

Only for nested functions

Scope is the enclosing function

Global scope:

Top-most scope

Built-in scope:

Special scope for built-in things: keywords, exceptions etc.

Local scope

```
8
9  def square(base):
10     result = base ** 2
11     print(f'The square of {base} is: {result}')
12
13  def cube(base):
14     result = base ** 3
15     print(f'The cube of {base} is: {result}')
16
17
18  x = 4
19  square(x)
20  cube(x)
21
```

```
The square of 4 is: 16
The cube of 4 is: 64
```

No confusion which
base is being
referred to

Enclosing or nonlocal scope

Only for **nested** functions: functions defined inside other functions

```
8
9  def outer():
10     var = 100
11     def inner():
12         print('Printing var from inner function',var)
13
14     inner()
15     print('Printing var from outer function:',var)
16
17
18
19
20  outer()
21
22
```

Local scope of `outer()` is
the enclosing scope of
`inner()`

```
10132/lectures/lec0_functions/code /
Printing var from inner function 100
Printing var from outer function: 100
```

Global scope

LEGB rule

Using global variables is considered **bad** programming practice

```
8  def f():
9      x = 5
10     g()
11     print('in f',x)
12
13  def g():
14      x = 7
15      h()
16
17  def h():
18      print('in h',x)
19
20
21  x = 3
22  f()
23  print('in main',x)
24
```

local scope

HW: what happens if this were a nested function?

global variable

```
in h 3
in f 5
in main 3
```

printing the global value of x

Summary of 3 scopes

LEGB rule

```
8
9  def outer():
10     # defines local scope of outer()
11     # also defines enclosing scope of inner()
12     def inner():
13         print(number)
14     inner()
15
16
17 number = 100
18 outer()
```

1. Inside `inner()`: local scope, but `number` doesn't exist
2. Inside `outer()`: enclosing scope, but `number` doesn't exist there either
3. Global scope, `number` exists here

100

→ Printing the global value
in this case

Built-in scope

LEGB rule

For built-in functions

eg. `len()`

Remember: do not redefine built-in names!

Arguments

```
7
8
9 def nameAdd1(fn,ln):
10     return fn + ' ' + ln
11
12
13 def nameAdd2(fname,lname):
14     return fname + ' ' + lname
15
16
17 def nameAdd3(fname,lname='Gupta'):
18     return fname + ' ' + lname
19
20
```

```
In [22]: print(nameAdd1('Ajay','Gupta'))
Ajay Gupta

In [23]: print(nameAdd2(lname='Nair',fname='Rahul'))
Rahul Nair

In [24]: print(nameAdd3(lname='Kumari',fname='Preeti'))
Preeti Kumari

In [25]: print(nameAdd3('Anshul'))
Anshul Gupta
```

Positional arguments

Number and order must match

Keyword arguments

Order can change

Default values

Can omit some arguments


```
8
9 def AddEnd(data=[]):
10     data.append('end')
11     return data
```

```
In [37]: a = ['Monday']
```

```
In [38]: AddEnd(a)
```

```
Out[38]: ['Monday', 'end']
```

```
In [39]: AddEnd(a)
```

```
Out[39]: ['Monday', 'end', 'end']
```

```
In [40]: b = AddEnd()
```

```
In [41]: b = AddEnd()
```

```
In [42]: b
```

```
Out[42]: ['end', 'end']
```

Unexpected behaviour while
using mutable objects.

Hence, avoid

```

9  def f(x):
10     fx = 10
11

```

```

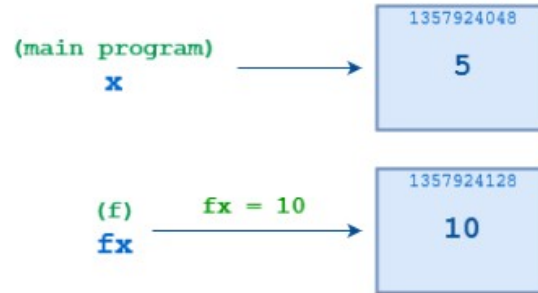
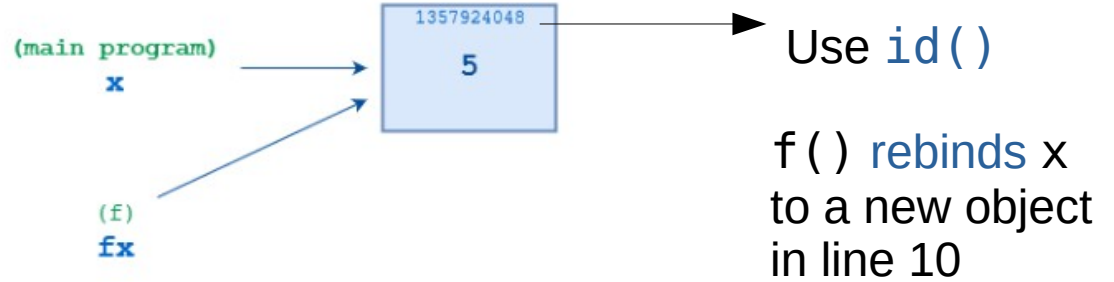
In [47]: x=5

In [48]: x
Out[48]: 5

In [49]: f(x)

In [50]: x
Out[50]: 5

```



From <https://realpython.com>

```
12 def g(x):  
13     x[0] = 10
```

```
In [55]: y  
Out[55]: [1, 2, 3]  
  
In [56]: g(y)  
  
In [57]: y  
Out[57]: [10, 2, 3]
```

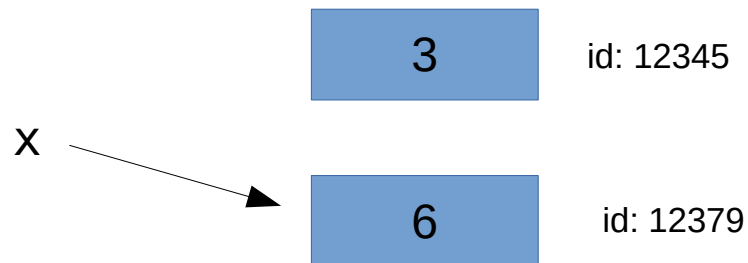
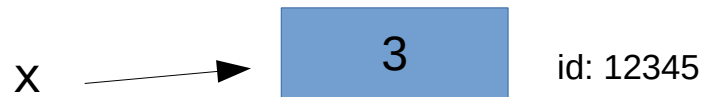


Contents of mutable objects can be changed

Immutable objects like int, str, tuple etc. cannot be changed

```
15  
16 def doubling(x):  
17     x = x*2  
18
```

```
In [59]: x=3  
  
In [60]: doubling(x)  
  
In [61]: x  
Out[61]: 3
```



```
19
20 def doubling2(x):
21     return x*2
```

HW: what if x is a list?

```
In [65]: x
Out[65]: 6

In [66]: id(x)
Out[66]: 94081603980768

In [67]: x
Out[67]: 6

In [68]: x=doubling2(x)

In [69]: id(x)
Out[69]: 94081603980960

In [70]: x
Out[70]: 12
```

It's a new x

Another example

Swap1:
Call by
value

Swap2:
Call by
reference

Return more than one
value

Multiple
assignment

```
8
9 def Swap1(a,b):
10     temp = a
11     a = b
12     b = temp
13
14 def Swap2(a,b):
15     return b,a
16
17
18
19 # multiple assignment
20 i,j = 2,3
21
22 print('Original i,j:',i,j)
23
24 Swap1(i,j)
25 print('After Swap1:',i,j)
26
27 i,j = Swap2(i,j)
28 print('After Swap2:',i,j)
29
```

```
Original i,j: 2 3
After Swap1: 2 3
After Swap2: 3 2
```

Recursion

- A recursive function **calls itself**
- Factorial

$$n! = 1$$

$$n=0$$

Recurrence relation

$$= n \times (n-1)! \quad n > 0$$

```
8
9 def Fact(n):
10     if (n==0):
11         print('Debug: n=0')
12         return 1
13     else:
14         print('Debug: n = ',n)
15         return n * Fact(n-1)
16
17 x = int(input('Enter a non-negative number: '))
18 print(Fact(x))
19
20
```

```
Enter a non-negative number: 17
Debug: n = 17
Debug: n = 16
Debug: n = 15
Debug: n = 14
Debug: n = 13
Debug: n = 12
Debug: n = 11
Debug: n = 10
Debug: n = 9
Debug: n = 8
Debug: n = 7
Debug: n = 6
Debug: n = 5
Debug: n = 4
Debug: n = 3
Debug: n = 2
Debug: n = 1
Debug: n=0
355687428096000
```