More on Functions

IC152 Lecture 10
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

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.

name is looked:
First in local
Then in enclosing
Then in global
Then in built-in
Not found, then error

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()`

```
Printing var from inner function 100
Printing var from outer function: 100
```

Global scope

LEGB rule

```
 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)
```

local scope

Using global variables is considered bad programming practice

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

```
def outer():
    # defines local scope of outer()
    # also defines enclosing scope of inner()
    def inner():
        print(number)
    inner()


number = 100
outer()
```

1. Inside `inner()`: local scope, but `number` dosent exist
2. Inside `outer()`: enclosing scope, but `number` dosent exist there either
3. Global scope, `number` exists here

```
100
```

Printing the global value in this case

LEGB rule

Built-in scope

For built-in functions

eg. `len()`

Remember: do not redefine built-in names!

Arguments

```
 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
```

```
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

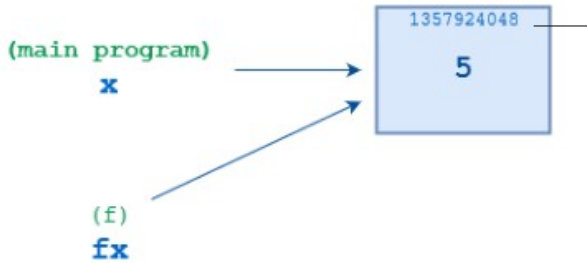
```
8
9   def AddEnd(data=[]):
10      data.append('end')
11      return data
```

```
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

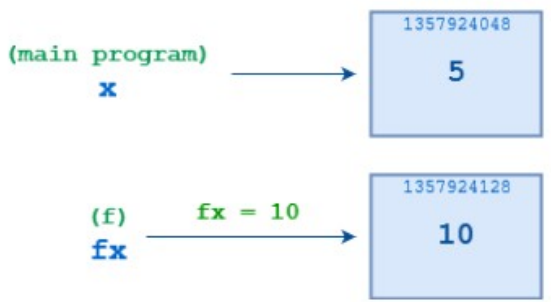
```
 9    def f(fx):
10        fx = 10
```

```
In [47]: x=5

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

In [49]: f(x)

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



Use `id()`

`f()` rebinds x to a new object in line 10



From https://realpython.com

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

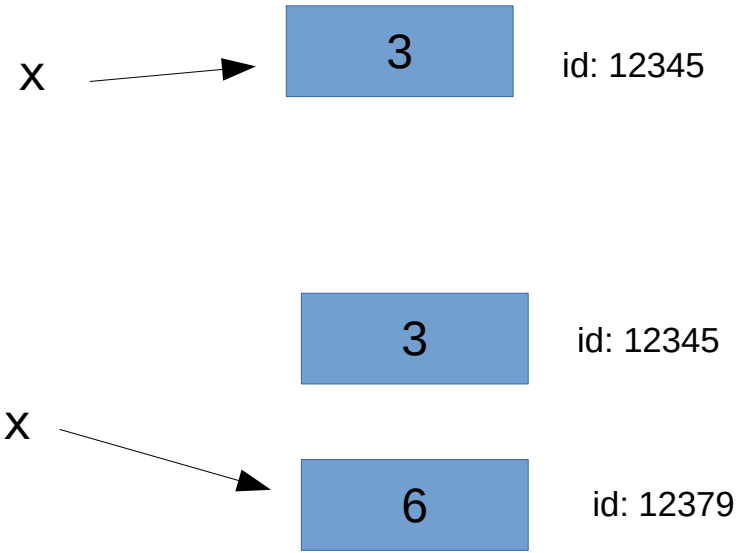
```
In [55]: y
Out[55]: [1, 2, 3]

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

```
 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
```

Return more than one value

Multiple assignment

Swap1: Call by value

Swap2: Call by reference

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

Recursion

- A recursive function calls itself
- Factorial

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

Recurrence relation

```
 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
```

- Fibonacci numbers

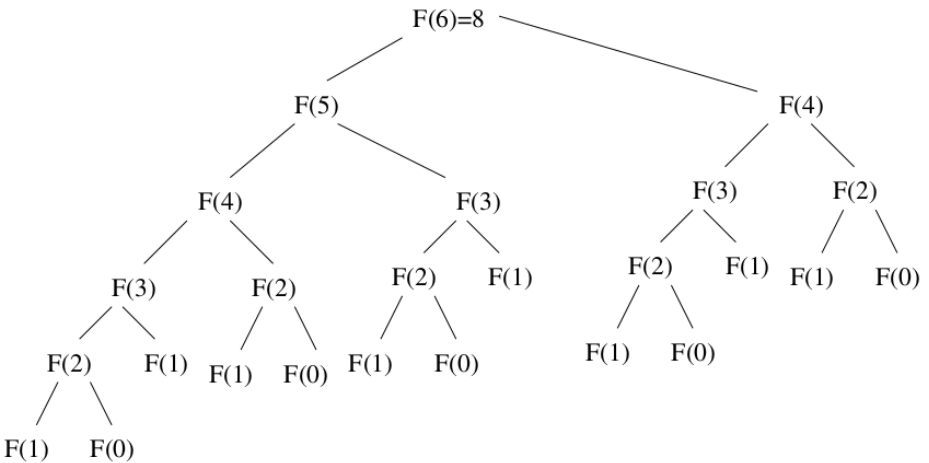
  $F_n = F_{n-1} + F_{n-2}$

  $F_0 = 0, F_1 = 1$

- $F_2=1, F_3=2, F_4=3, F_5=5,$ {8,13,21,34,55,89,144...}

```
11
12  # Compute Fib number by recursion: exponential time
13  def fibrec(n):
14      if (n==0):
15          #print('Debug: Base case 0')
16          return 0
17      if (n==1):
18          #print('Debug: Base case 1')
19          return 1
20      return (fibrec(n-1)+fibrec(n-2))
21
```

Takes exponential time!



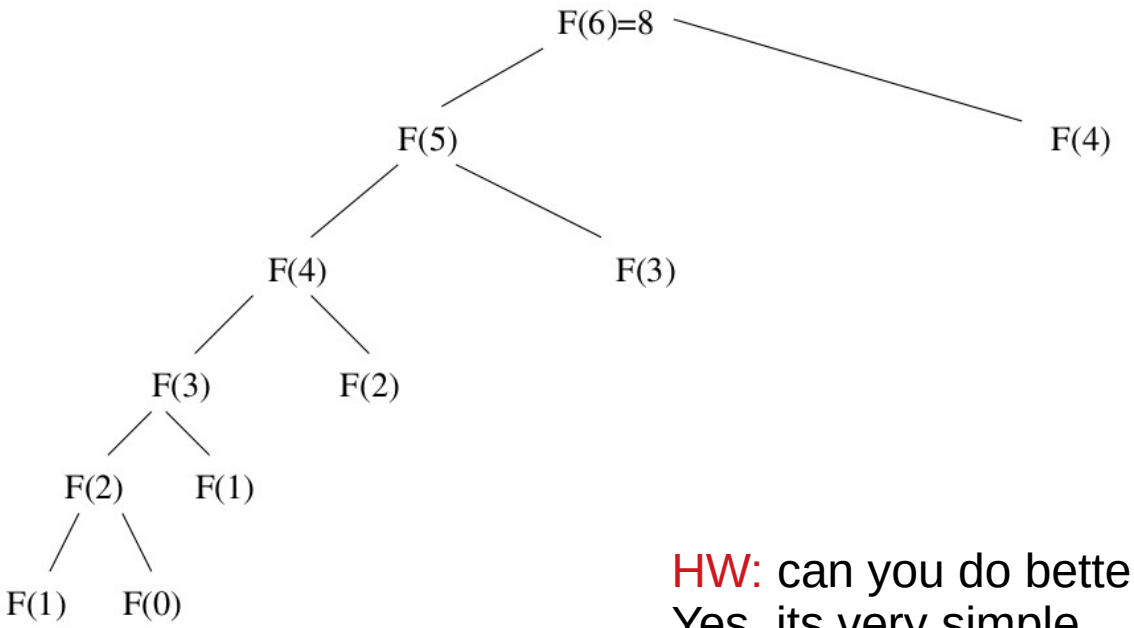
Pic from Algorithm Design Manual by S.Skiena

Can we do better?

Store the computed values to use later!
Lets use a list

```
10  # Compute Fib number by caching: reuse values computed previously
11  def fibcache(n):
12      if g_Fiblist[n] == -1:
13          g_Fiblist[n] = fibcache(n-1)+fibcache(n-2)
14      return g_Fiblist[n]
15
16
17
18
19  x = int(input('Enter a non-negative number: '))
20
21
22  # max Fib number
23  MAXFIB = 100
24  assert x < MAXFIB, 'Max fib exceeded!'
25
26  # global list for caching
27  # initialize with -1
28  g_Fiblist = [-1]*MAXFIB
29  # set 0 and 1
30  g_Fiblist[0] = 0
31  g_Fiblist[1] = 1
32  print('F_cache' + str(x)+': ' + str(fibcache(x)))
33
```

Computation tree while using caching



HW: can you do better?
Yes, its very simple.

Demo code

```
 8
 9 import time
10
11 # Compute Fib number by recursion: exponential time
12 def fibrec(n):
13     if (n==0):
14         #print('Debug: Base case 0')
15         return 0
16     if (n==1):
17         #print('Debug: Base case 1')
18         return 1
19     return (fibrec(n-1)+fibrec(n-2))
20
21 # Compute Fib number by caching: reuse values computed previously
22 def fibcache(n):
23     if g_Fiblist[n] == -1:
24         g_Fiblist[n] = fibcache(n-1)+fibcache(n-2)
25     return g_Fiblist[n]
26
27 # main program
28 x = int(input('Enter a non-negative number: '))
29
30 # compute the time taken
31 tic = time.perf_counter()
32 print('F_rec '+str(x)+': ' + str(fibrec(x)))
33 toc = time.perf_counter()
34 print(f'Time taken rec: {toc - tic:0.4f} seconds')
35
36 # max Fib number
37 MAXFIB = 100
38 assert x < MAXFIB, 'Max fib exceeded!'
39 tic = time.perf_counter()
40 # global list for caching
41 # initialize with -1
42 g_Fiblist = [-1]*MAXFIB
43 # set 0 and 1
44 g_Fiblist[0] = 0
45 g_Fiblist[1] = 1
46 print('F_cache ' + str(x)+': ' + str(fibcache(x)))
47 toc = time.perf_counter()
48 print(f'Time taken cache: {toc - tic:0.4f} seconds')
49
```