Functions

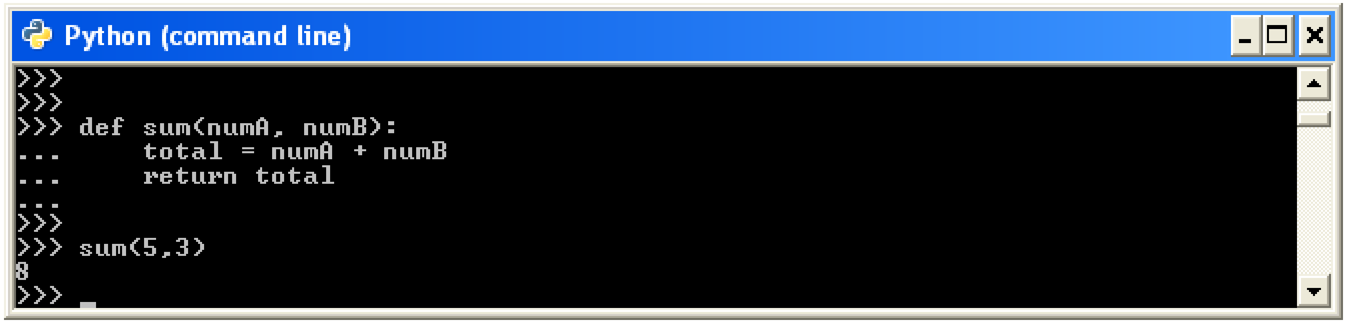
Introduction

- Each function is a collection of statements that can be executed more than once in a program.
- Functions can receive arguments and they can calculate and return a value back to the caller.

The `def` Statement

- We create a function by calling the def statement. Each function we create is assigned with a name. We can later use that name in order to call it.

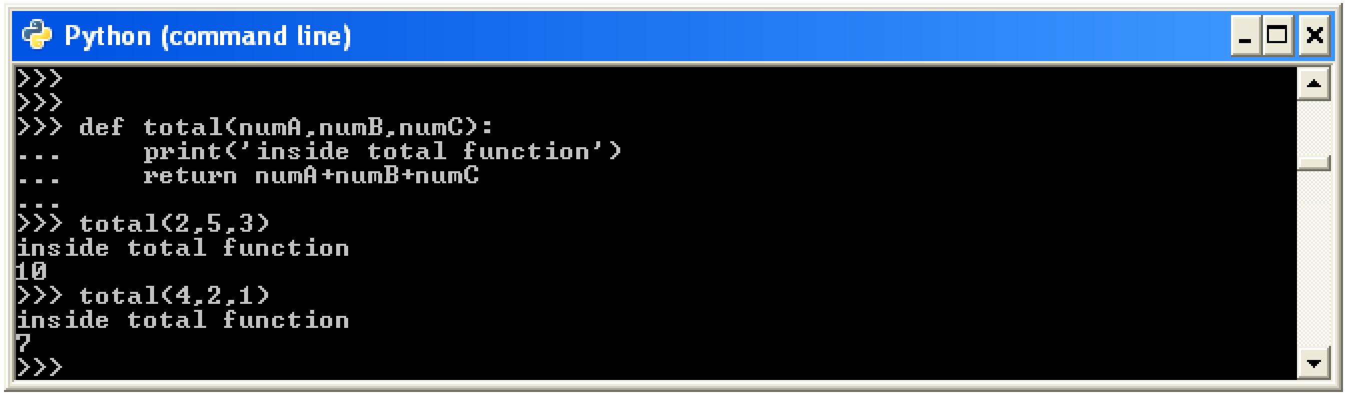
```
def function_name (param1, param2, param3,... paramN):
    statements
```



```
>>>
>>>
>>> def sum(numA, numB):
...     total = numA + numB
...     return total
...
>>>
>>> sum(5,3)
8
>>> _
```

The `def` Statement

- The execution of '`def`' takes place in run-time. Only then the object function is created.



```
>>>
>>>
>>> def total(numA,numB,numC):
...     print('inside total function')
...     return numA+numB+numC
...
>>> total(2,5,3)
inside total function
10
>>> total(4,2,1)
inside total function
7
>>>
```

- The definition of our function is a statement. We can place a function definition wherever we can place a statement.

The `def` Statement

```
def sum(a,b):
    total = a + b
    return total

print(sum(4,3))
```

The `def` Statement

- We can place different definitions for the same function and using a simple if statement choosing which of those versions will be defined.

```
...
if test:
    def func():
        ...
else:
    def func():
        ...
...
```

The `def` Statement

- Each function is just an object. The name assigned to each function is just a name. We use that name in order to call the function.

Function Attributes

- Because a function is an object we can add new attributes we choose.

```
def sum(a,b):
    c = a + b
    return c

sum.version = 101
sum.author = "haim michael"
sum.priority = 3
print(sum.author)
```

Variables Scope

- The place where we assign a name with a value in our code determines its scope.
- Names we assign within the scope of a function are considered as local variables. When the function ends they disappear.
- Names we assign within the scope of an enclosing def are considered as non local to the nested function.
- Names we assign outside of all functions are globals.

The Local Scope

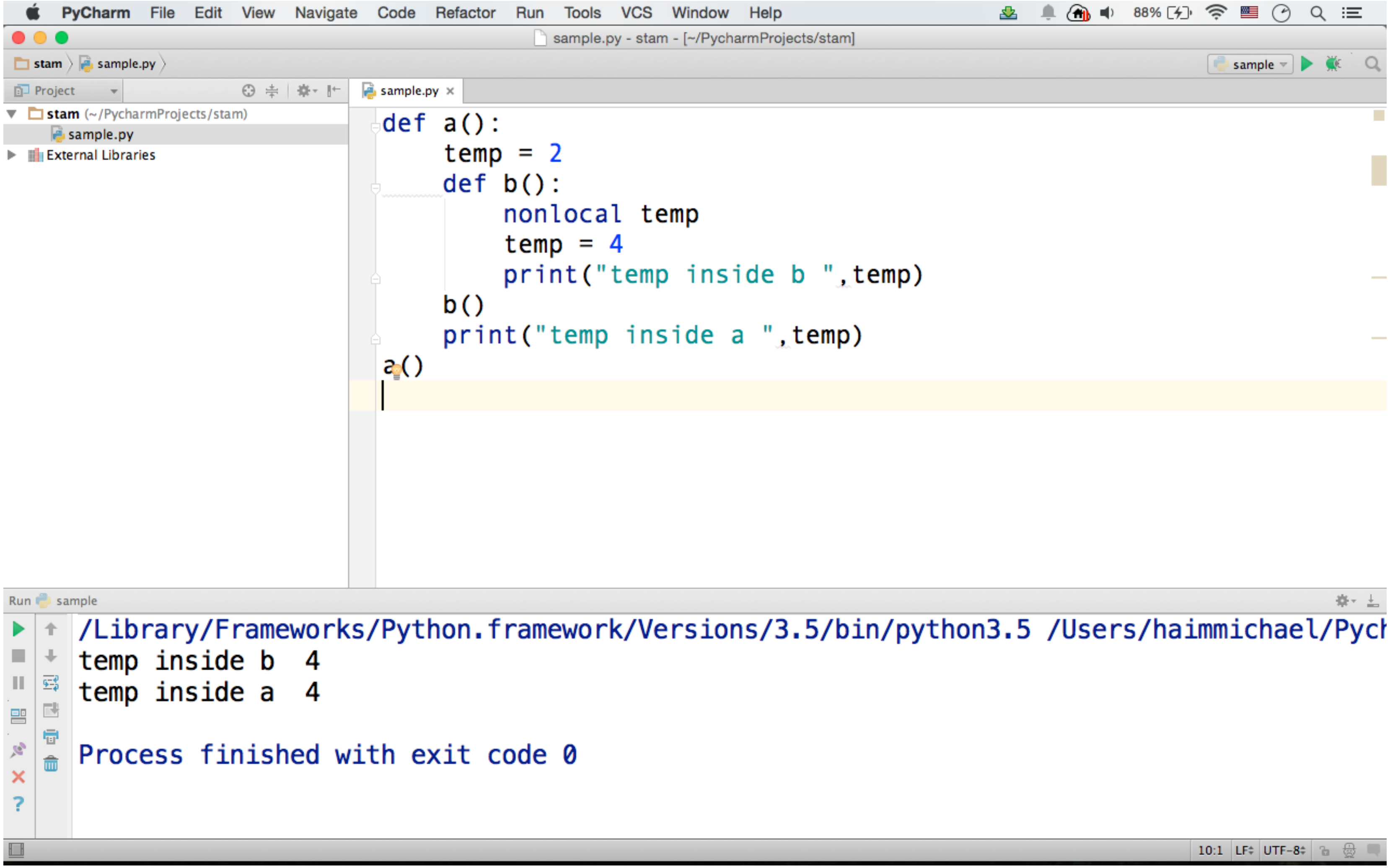
- Each time a function is called a new local scope is created. We can think of using '`def`' as of creating a new local scope. Each function call creates a new local scope. This behavior allows us to code recursive functions.

The `nonlocal` Keyword

- Adding the `nonlocal` keyword will turn the variable into a local variable that belongs to the enclosing scope.

```
def a():
    temp = 2
    def b():
        nonlocal temp
        temp = 4
        print("temp inside b ",temp)
    b()
    print("temp inside a ",temp)
a()
```

The `nonlocal` Keyword



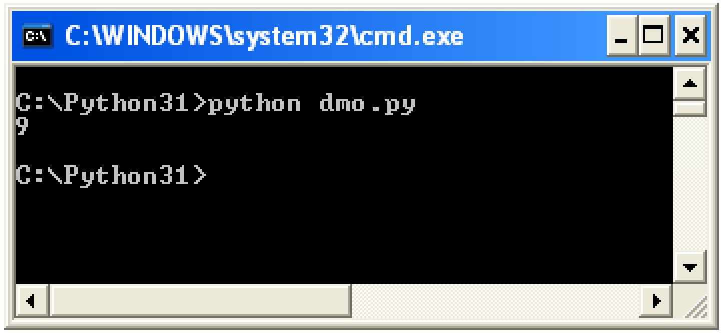
PyCharm File Edit View Navigate Code Refactor Run Tools VCS Window Help
88%
sample.py - stam - [~/PycharmProjects/stam]
stam
sample.py
sample
Project
stam (~/PycharmProjects/stam)
sample.py
External Libraries
sample.py
Run sample
10:1 LF UTF-8

```python
def a():
    temp = 2
    def b():
        nonlocal temp
        temp = 4
        print("temp inside b ",temp)
    b()
    print("temp inside a ",temp)
a()
```

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Pych
temp inside b  4
temp inside a  4

Process finished with exit code 0
```

The `nonlocal` Statement

```
def doSomethingA():
    number = 7
    def doSomethingB():
        nonlocal number;
        number = 9;
    doSomethingB()
    print(number)

doSomethingA()
```



```
C:\Python31>python dmo.py
9

C:\Python31>
```

The Global Scope

- Each module is considered as a global scope. Each module has its own namespace.
- Variables we create within a given module become attributes of the module object itself.
- The global scope spans over one single file only. Variables created in one file aren't accessible from other files.

The Global Scope

- If we want to use variables created in a specific module (file) outside of it we must import that module (file).

The Global Scope

```
import other
sum = other.temp + 2
print(sum)
```

demo.py

```
temp = 12
```

other.py

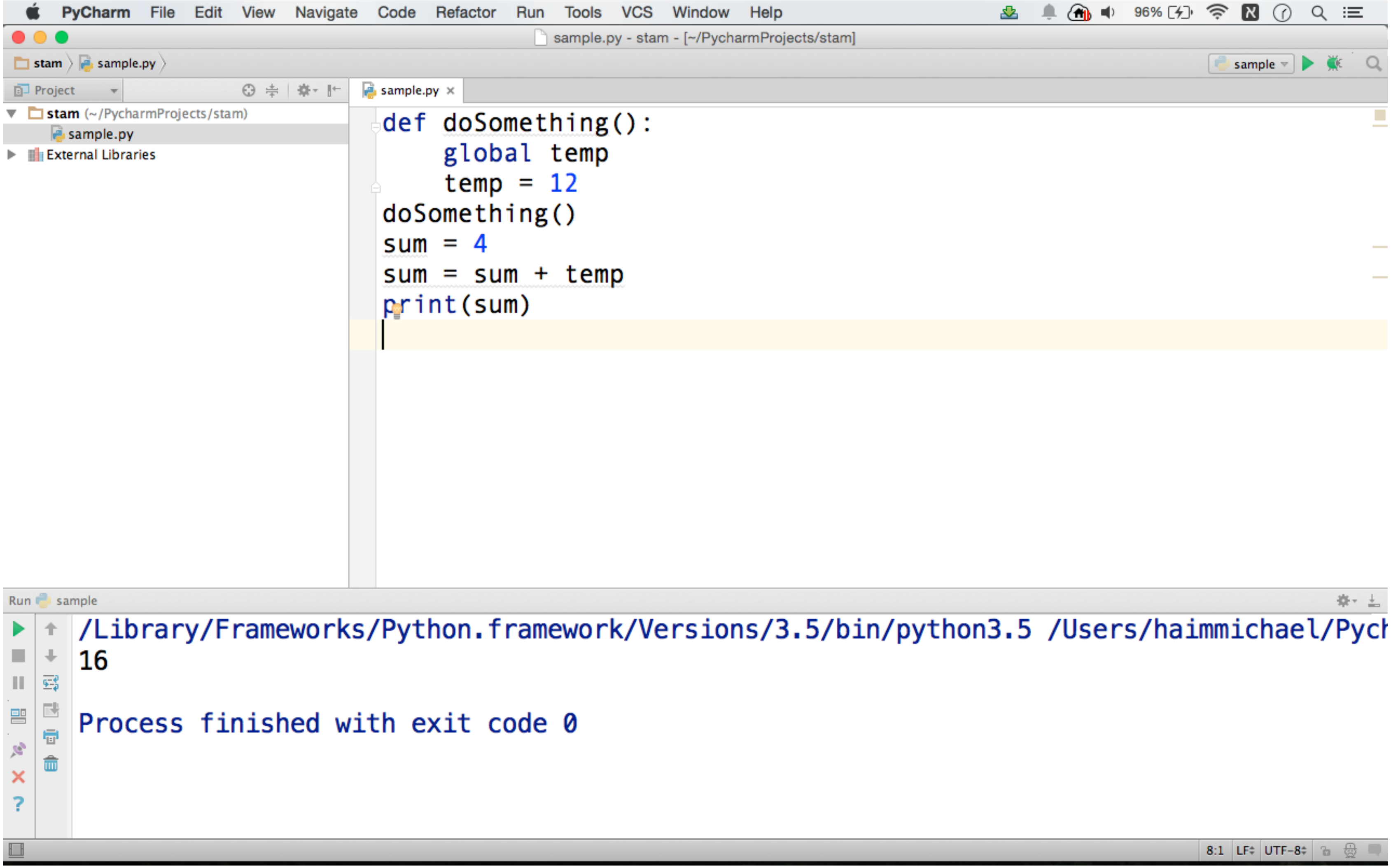
The Global Scope

- Defining a variable within a function together with the '`global`' keyword will turn it into a global variable.
- The variable will become a global variable for the top level enclosing module.

The Global Scope

```
def doSomething():
    global temp
    temp = 12
doSomething()
sum = 4
sum = sum + temp
print(sum)
```

The Global Scope


PyCharm File Edit View Navigate Code Refactor Run Tools VCS Window Help
sample.py - stam - [~/PycharmProjects/stam]
stam
sample.py
Project
stam (~/PycharmProjects/stam)
sample.py
External Libraries
Run sample
8:1 LF UTF-8

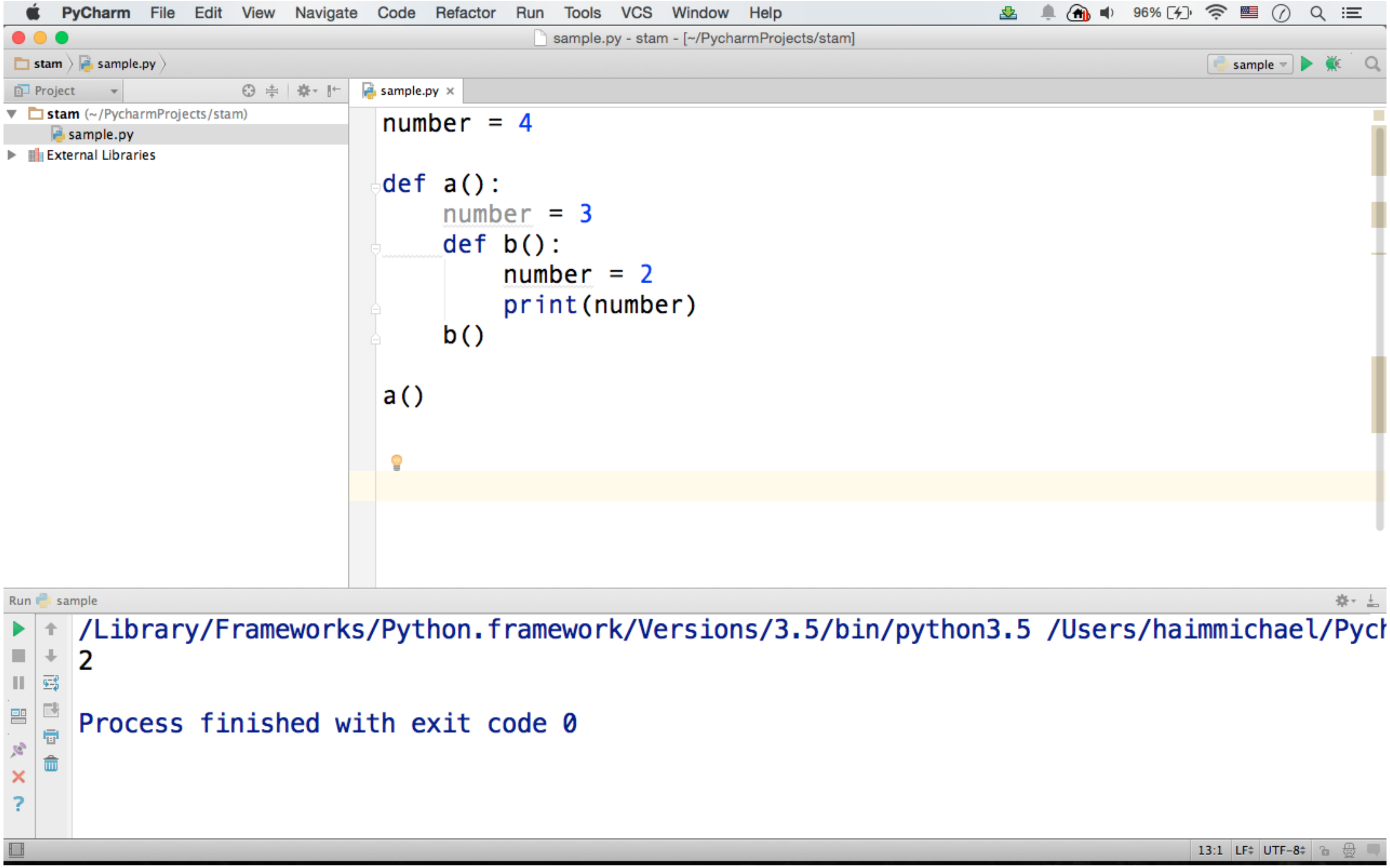
```
def doSomething():
    global temp
    temp = 12
doSomething()
sum = 4
sum = sum + temp
print(sum)
```

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Pych
16

Process finished with exit code 0
```

The LEGB Rule

- When referring a name in our code the search is carried on through the following four scopes: `local`, `enclosing`, `global` and `built-in`.
- LEGB stands for Local, Enclosing, Global and Built-in.

The LEGB Rule



PyCharm File Edit View Navigate Code Refactor Run Tools VCS Window Help
sample.py - stam - [~/PycharmProjects/stam]
stam
sample.py
Project
stam (~/PycharmProjects/stam)
sample.py
External Libraries
Run sample
13:1 LF UTF-8

```
number = 4

def a():
    number = 3
    def b():
        number = 2
        print(number)
    b()

a()
```

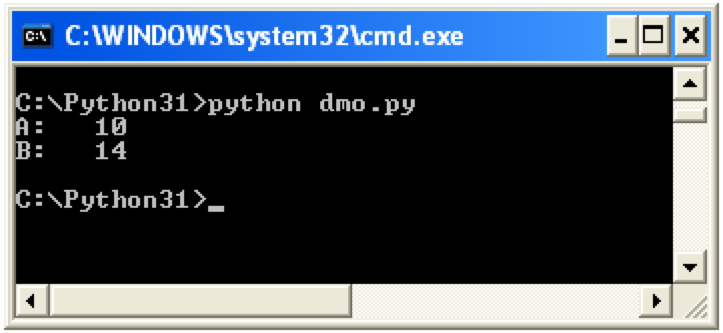
```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Pych
2

Process finished with exit code 0
```

Accessing Globals

- Using the `global` keyword we can specify a specific variable we want to be treated as if it is a global variable.

Accessing Globals

```
number = 10

def doSomethingA():
    number = 12

def doSomethingB():
    global number
    number = 14

doSomethingA()
print("A:   ",number)

doSomethingB()
print("B:   ",number)
```



```
C:\Python31>python dmo.py
A:    10
B:    14

C:\Python31>_
```

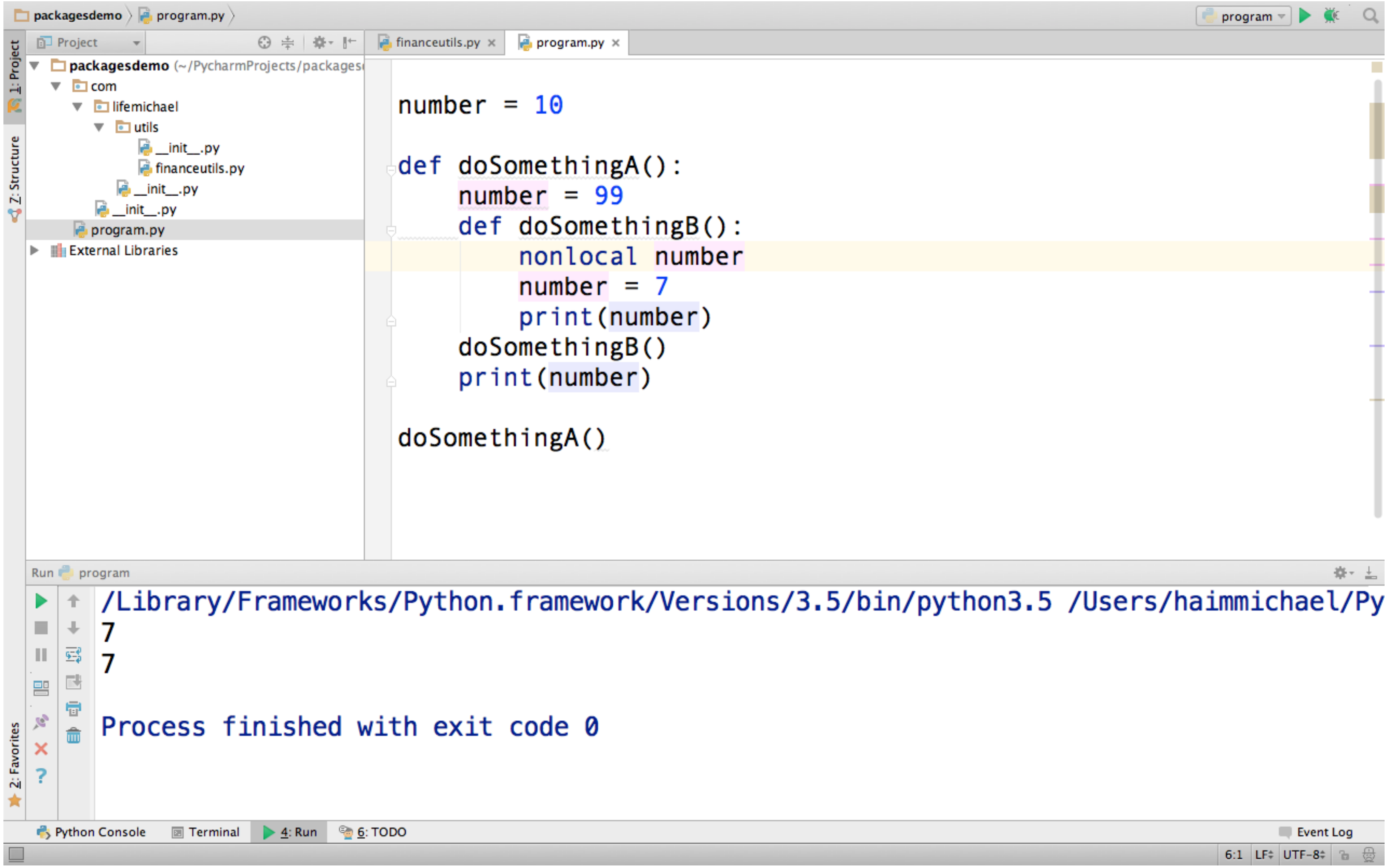
Nested Functions

- Defining functions we can get them one within the other.
- Referencing a variable ('x') from within a given current scope (function) first searches for a lexically enclosing function. From inner to outer till reaching the current global scope (the module file).

Nested Functions

- If the variable (x) is referenced as a `nonlocal` variable within our function the assignment will change the variable (x) in the closest enclosing function's local scope.

Nested Functions



```
number = 10

def doSomethingA():
    number = 99
    def doSomethingB():
        nonlocal number
        number = 7
        print(number)
    doSomethingB()
    print(number)

doSomethingA()
```

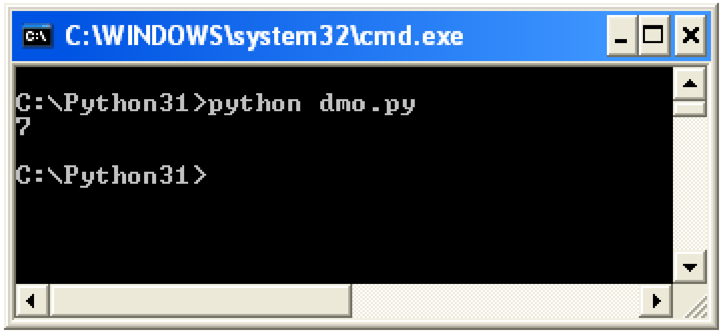
```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Py
7
7

Process finished with exit code 0
```

Returned Functions

- We can define a function that its returned value is another function.
- When one function returns a function it includes its definition.
- The returned function is capable of referring variables that belong to the scope of the outer one.

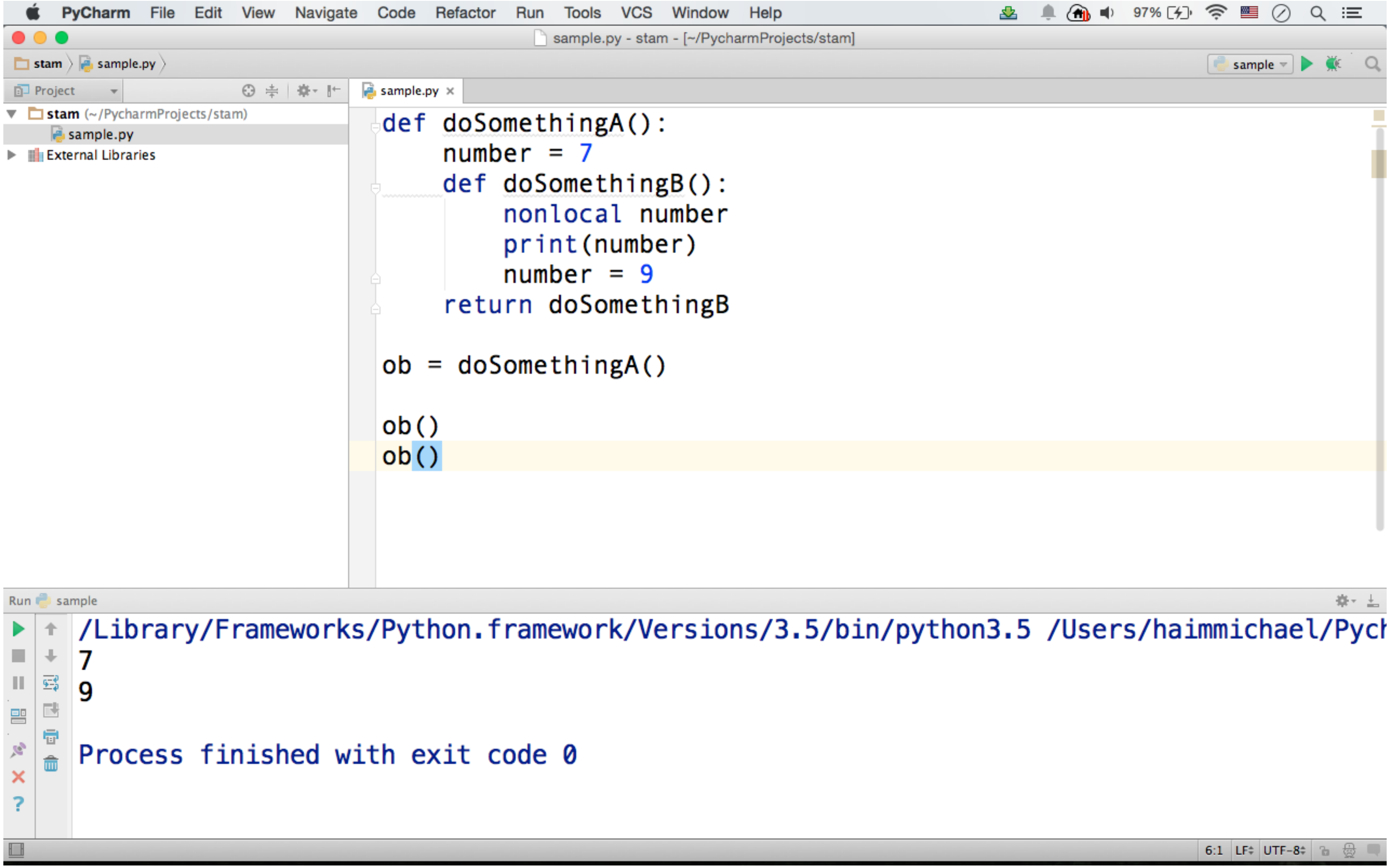
Returned Functions

```
def doSomethingA():
    number = 7
    def doSomethingB():
        print(number)
    return doSomethingB

ob = doSomethingA()

ob()
```



```
C:\Python31>python dmo.py
7

C:\Python31>
```

Returned Functions



PyCharm File Edit View Navigate Code Refactor Run Tools VCS Window Help
sample.py - stam - [~/PycharmProjects/stam]
stam
sample.py
Project
stam (~/PycharmProjects/stam)
sample.py
External Libraries
Run sample
6:1 LF UTF-8

```
def doSomethingA():
    number = 7
    def doSomethingB():
        nonlocal number
        print(number)
        number = 9
    return doSomethingB

ob = doSomethingA()

ob()
ob()
```

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Pych
7
9

Process finished with exit code 0
```

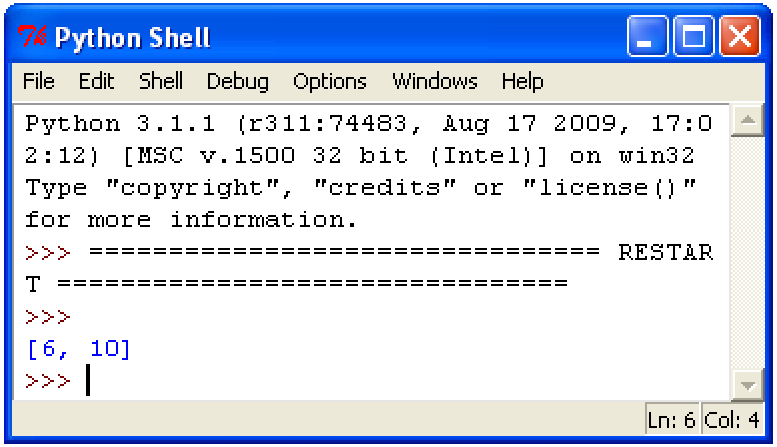
Arguments

- When calling a function passing over names, we actually pass the references held by these names.
- Assigning new references to the parameter names within the function scope doesn't effect the names the caller passed.
- Changing a mutable object from within the function the caller code should feel that as well.

Sequence Returned Value

- We can define a function that returns a tuple, or any other sequence type.

```
#dmo

def f(a,b):
    numA = 2 * a
    numB = 2 * b
    return [numA,numB]

x = f(3,5)

print(x)
```



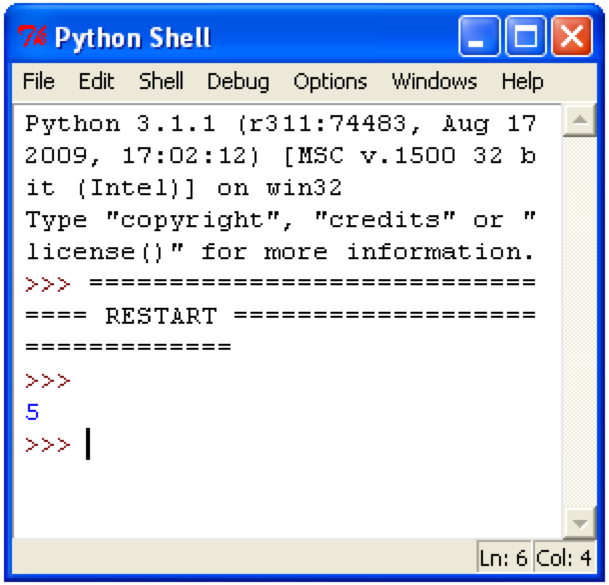
```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:0
2:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()"
for more information.
>>> ================================ RESTAR
T ================================
>>> 
[6, 10]
>>> 
```

© 2008 Haim Michael 20151020

The `func(name)` Syntax

- By default, the arguments we pass must match by position, left to right, and we must pass exactly as many arguments as required.

```
def f(a,b):
    sum = a+b
    print(sum)

f(2,3)
```

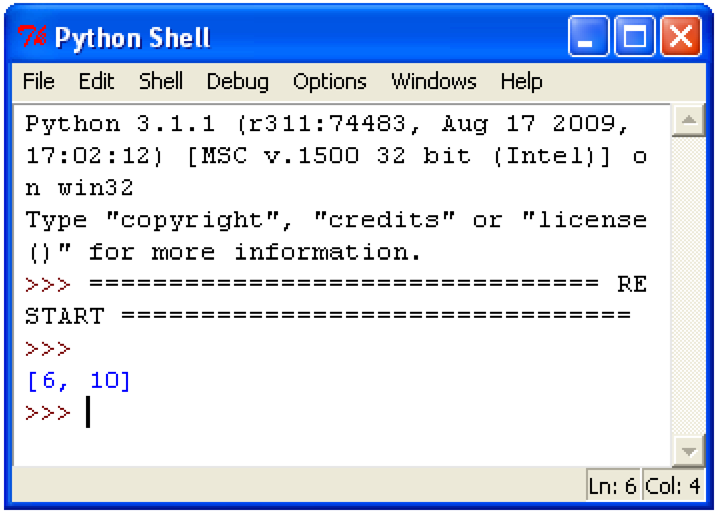


```
Python 3.1.1 (r311:74483, Aug 17
2009, 17:02:12) [MSC v.1500 32 b
it (Intel)] on win32
Type "copyright", "credits" or "
license()" for more information.
>>> ============================
==== RESTART ===================
=============
>>> 
5
>>> 
```

The `func(name=value)` Syntax

- Calling a function we can specify which parameters should receive a value by using the argument's name in the `name=value` syntax.

```
#dmo

def f(a,b):
    numA = 2 * a
    numB = 2 * b
    return [numA,numB]

x = f(a=3,b=5)

print(x)
```

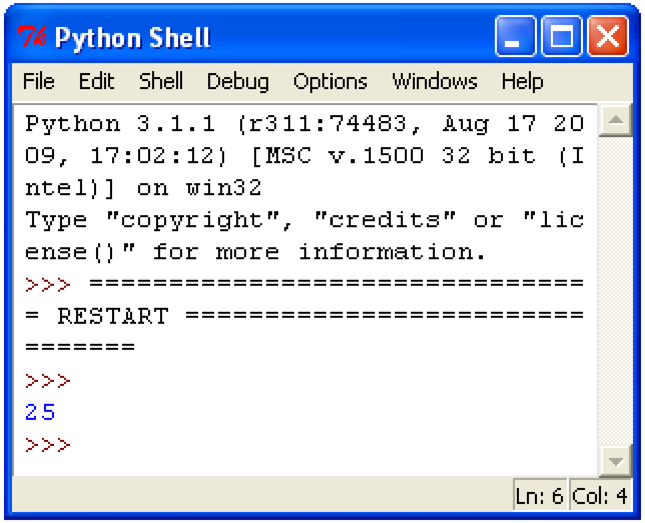


```
Python 3.1.1 (r311:74483, Aug 17 2009,
17:02:12) [MSC v.1500 32 bit (Intel)] o
n win32
Type "copyright", "credits" or "license
()" for more information.
>>> ================================ RE
START ================================
>>>
[6, 10]
>>> 
```

The `func(*name)` Syntax

- Adding * to the sequence we pass over to the function, the function will be capable of unpacking the passed argument into discrete separated parameters.

```
def f(x1,y1,x2,y2):
    return (y2-y1)*(y2-y1)+(x2-x1)*(x2-x1)

ob = [0,0,4,3]
num = f(*ob)
print(num)
```

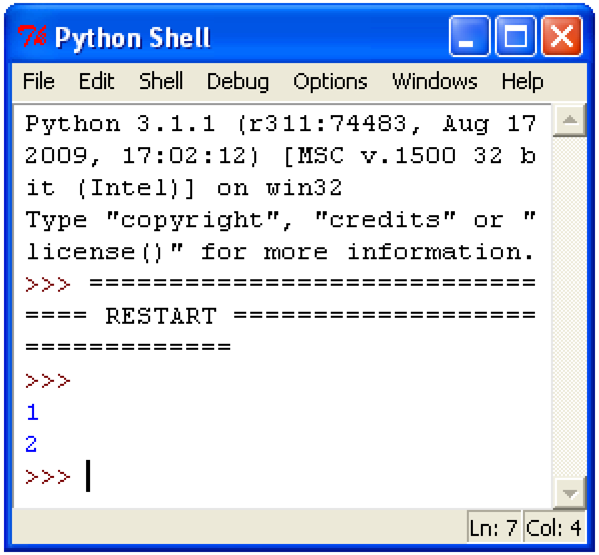


```
Python 3.1.1 (r311:74483, Aug 17 20
09, 17:02:12) [MSC v.1500 32 bit (I
ntel)] on win32
Type "copyright", "credits" or "lic
ense()" for more information.
>>> ==============================
= RESTART ========================
=======
>>>
25
>>>
```

The `func(**name)` Function

- Adding ** to the argument name, when calling the function a collection of key/value pairs in the form of a dictionary will be expected to be passed over to the function as individual keyword arguments.

```
def f(a,b):
    print(a)
    print(b)

ob = {'a':1,'b':2}
f(**ob)
```



```
Python 3.1.1 (r311:74483, Aug 17
2009, 17:02:12) [MSC v.1500 32 b
it (Intel)] on win32
Type "copyright", "credits" or "
license()" for more information.
>>> ============================
==== RESTART ===================
=============
>>>
1
2
>>> 
```

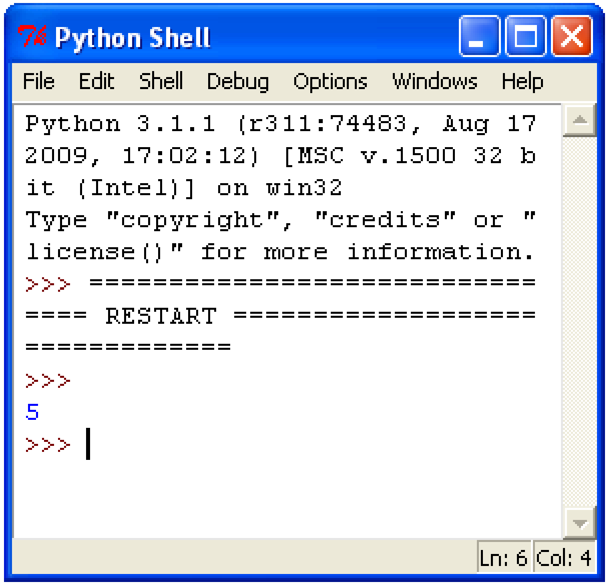
The `func(**name)` Function

- When defining a function we must specify the parameters in the following order. The first should be the normal parameters. After these parameters we should specify the default parameters followed by the *name parameters and followed by the **name ones.

The `def func(name)` Syntax

- Defining a simple function, the passed values should match by position or name.

```
def f(a,b):
    sum = a+b
    print(sum)

f(2,3)
```

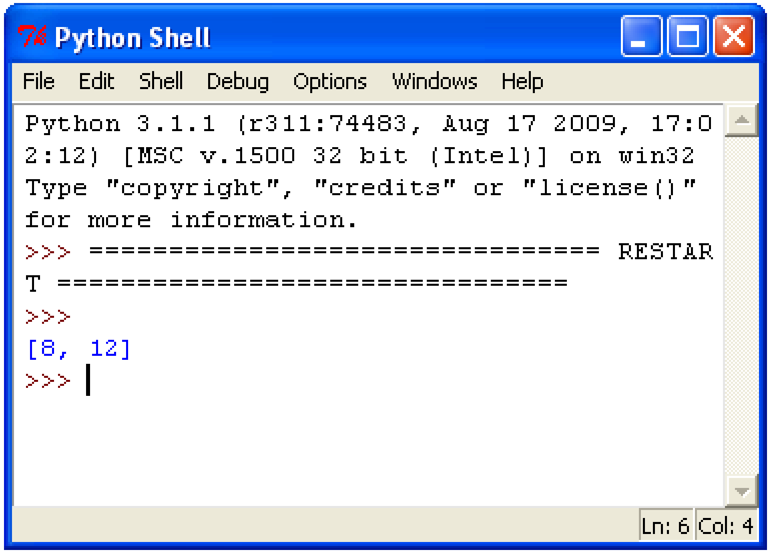


```
Python 3.1.1 (r311:74483, Aug 17
2009, 17:02:12) [MSC v.1500 32 b
it (Intel)] on win32
Type "copyright", "credits" or "
license()" for more information.
>>> ============================
==== RESTART ===================
=============
>>>
5
>>> 
```

The `def func(name=value)` Syntax

- Defining a function we can use the argument's name in the `name=value` syntax in order to specify default values for specific arguments.

```
def f(a=4,b=6):
    numA = 2 * a
    numB = 2 * b
    return [numA,numB]

x = f()

print(x)
```

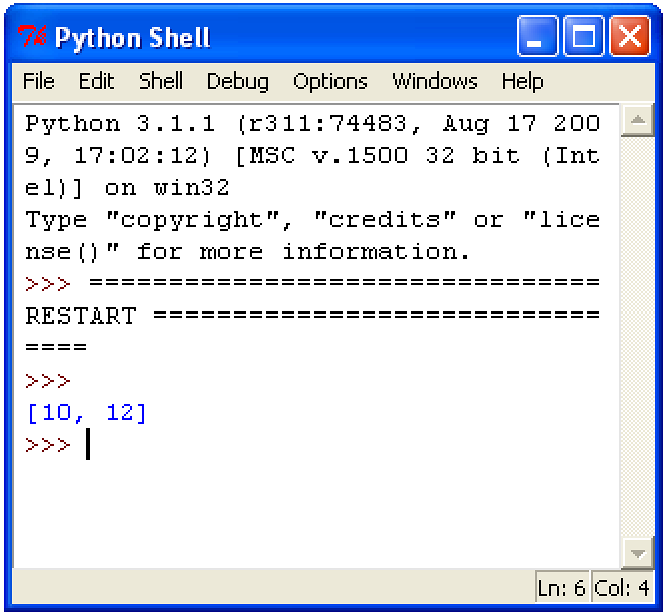


```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:0
2:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()"
for more information.
>>> ================================ RESTAR
T ================================
>>>
[8, 12]
>>> 
```

The `def func(name=value)` Syntax

- This code sample includes a function with two parameters. The first is a normal positioned one. The second has a default value.

```
def f(a,b=6):
    numA = 2 * a
    numB = 2 * b
    return [numA,numB]

x = f(5)

print(x)
```

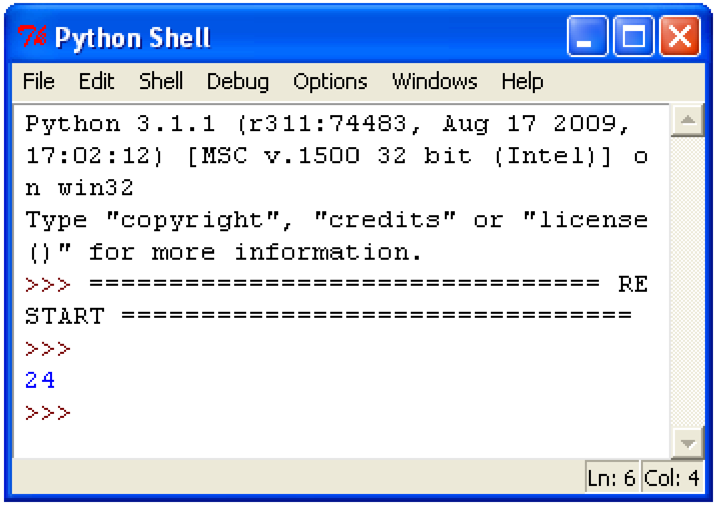


```
Python 3.1.1 (r311:74483, Aug 17 200
9, 17:02:12) [MSC v.1500 32 bit (Int
el)] on win32
Type "copyright", "credits" or "lice
nse()" for more information.
>>> ================================
RESTART ============================
====
>>> 
[10, 12]
>>> 
```

The `def func(*name)` Syntax

- Adding * to the parameter name in the function definition collects unmatched positional arguments into a tuple.

```
#dmo

def sum(*tpl):
    sum = 0
    for num in tpl:
        sum = sum + num
    return sum

print(sum(3,4,6,2,3,6))
```



```
Python 3.1.1 (r311:74483, Aug 17 2009,
17:02:12) [MSC v.1500 32 bit (Intel)] o
n win32
Type "copyright", "credits" or "license
()" for more information.
>>> ================================ RE
START ================================
>>>
24
>>>
```

The def func(*name) Syntax

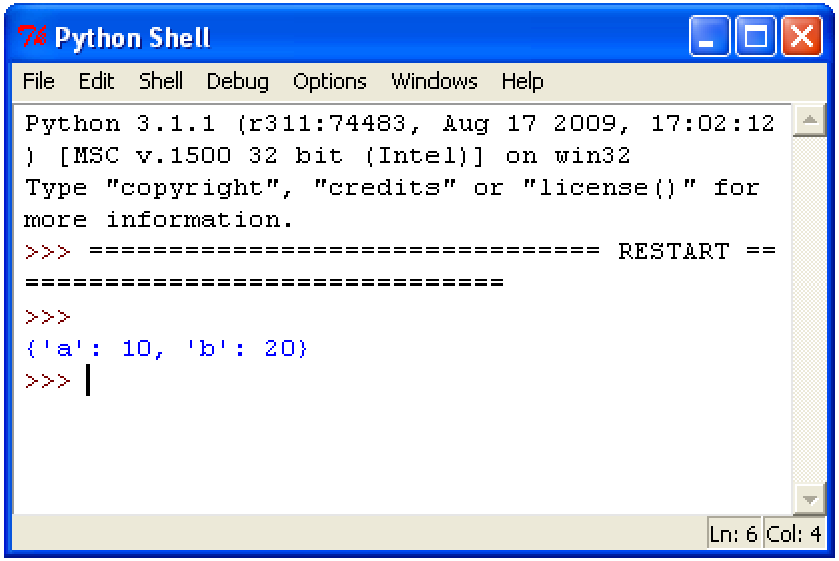
- It is common to name the parameter in these cases with args. This way it is clear that the function can get any number of arguments. They will be all packed in one itrable object.

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

print(sum(3,4,6,2,3,6))
```

The `def func(**name)` Syntax

- Adding ** to the parameter name in the function definition collects unmatched positional arguments into a dictionary.

```
def f(**args):
    print(args)

f(a=10,b=20)
```



```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12
) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for
more information.
>>> ================================ RESTART ==
==============================
>>> 
{'a': 10, 'b': 20}
>>> 
```

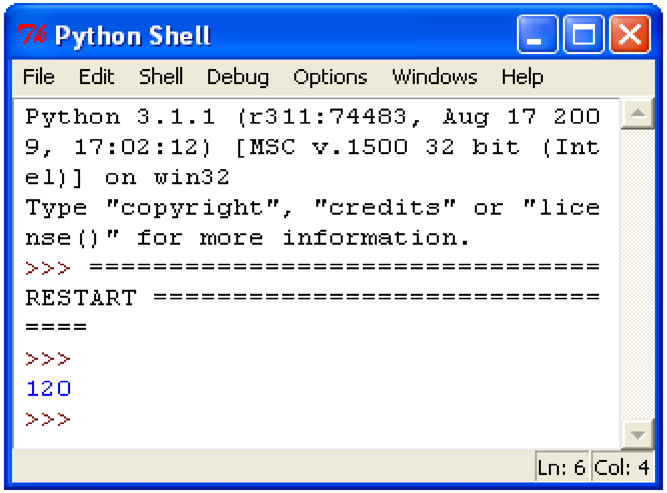
The `def func(**name)` Syntax

- Adding ** to the parameter name in the function definition collects unmatched positional arguments into a dictionary.

Indirect Function Call

- When assigning a function to one of our variables we can append `()` to that variable and use it in order to call that function.

```
def factorial(a):
    if a==0:
        return 1
    else:
        return a * factorial(a-1)

f = factorial
print(f(5))
```



```
Python 3.1.1 (r311:74483, Aug 17 200
9, 17:02:12) [MSC v.1500 32 bit (Int
el)] on win32
Type "copyright", "credits" or "lice
nse()" for more information.
>>> ================================
RESTART ============================
====
>>>
120
>>>
```

Functions Are Objects

- Because the functions are objects we can process a function as any other object.

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__', '__dict__',
'__doc__', '__eq__', '__format__', '__ge__', '__get__',
'__getattribute__', '__globals__', '__gt__', '__hash__',
'__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
>>>
```

Function Annotations

- When we define a function we can optionally specify the types of its parameters and the type of the returned value.

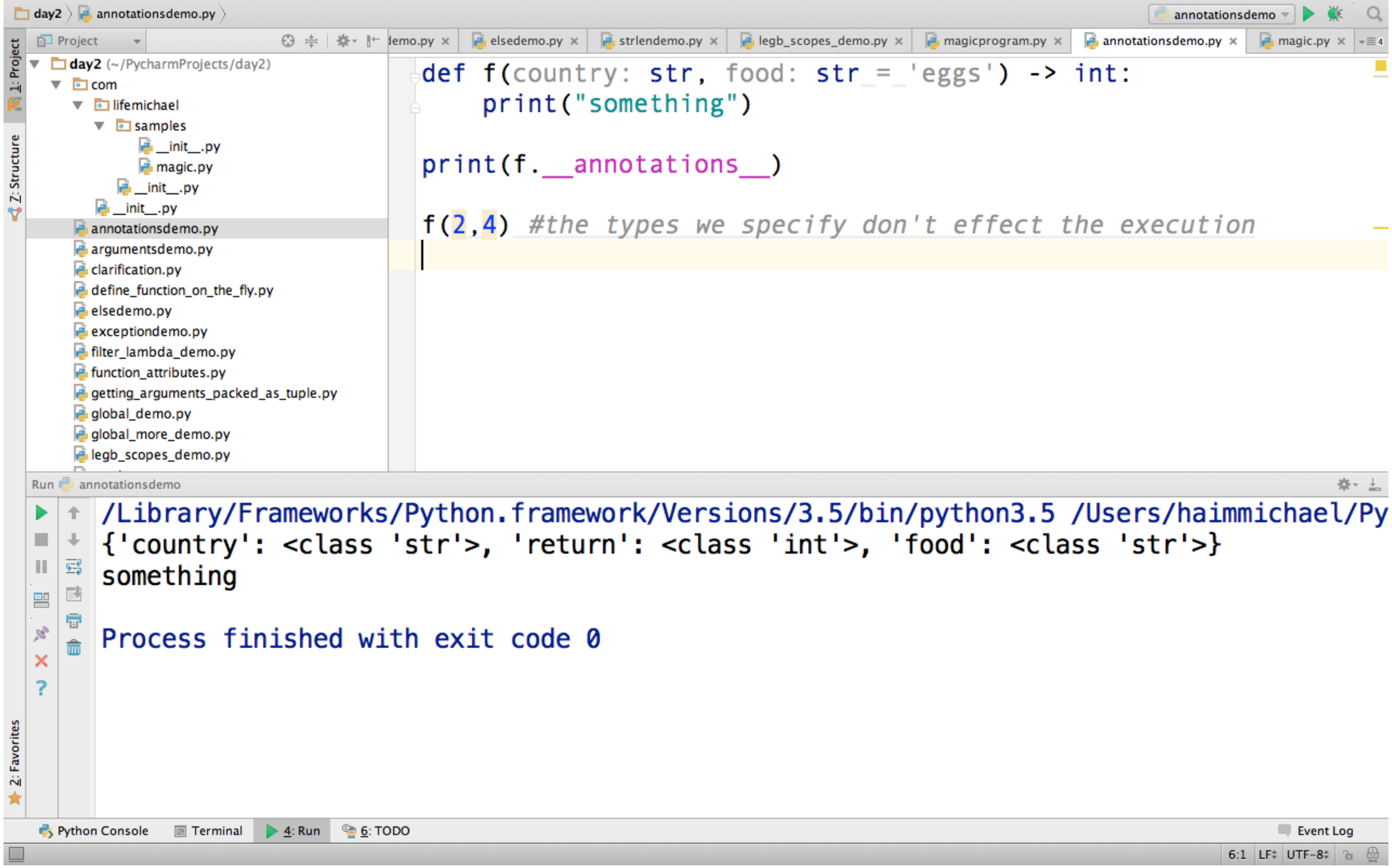
```
def f(country: str, food: str = 'eggs') -> int:
    print("something")
```

- The function annotations hold metadata information about the function, and specifically about the types of its parameters and the type of the returned value.

Function Annotations

- We can access the function annotations by referring the `__annotations__` attribute each function has.
- The function annotations hold optional meta data that doesn't effect the way the function is executed.

Function Annotations



```python
def f(country: str, food: str = 'eggs') -> int:
    print("something")

print(f.__annotations__)

f(2,4) #the types we specify don't effect the execution
```

```
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5 /Users/haimmichael/Py
{'country': <class 'str'>, 'return': <class 'int'>, 'food': <class 'str'>}
something

Process finished with exit code 0
```

Anonymous Functions (Lambda)

- Using the `lambda` keyword we can define an anonymous function.

```
lambda param1, param2, param3...paramN : expression
```

- Unlike using `def`, when using `lambda` we get an expression. Not a statement.

Anonymous Functions (Lambda)

```
ob = lambda a,b,c:a+b+c
print(ob(1,2,3))
```

Anonymous Functions (Lambda)

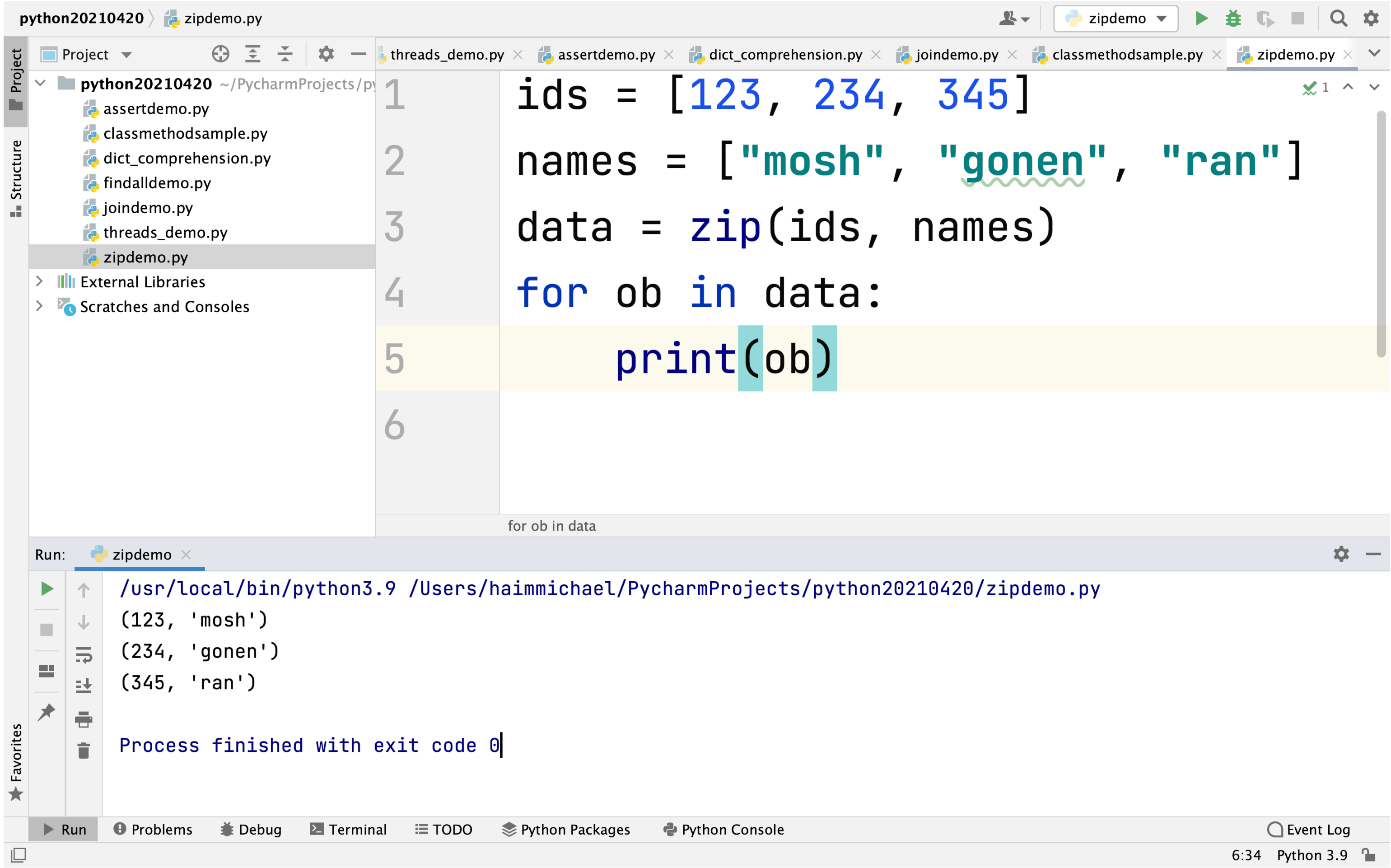
- Unlike using `def`, when using `lambda` we can have one single expression. We cannot have a block of statements.

The `zip()` Function

- Using the `zip` function we can zip together the elements coming from two different iterable objects and get a new iterable object that holds tuples that include those elements.

```
ids = [123, 234, 345]
names = ["mosh", "gonen", "ran"]
data = zip(ids, names)
for ob in data:
    print(ob)
```

The `zip()` Function



```
ids = [123, 234, 345]
names = ["mosh", "gonen", "ran"]
data = zip(ids, names)
for ob in data:
    print(ob)
```

```
/usr/local/bin/python3.9 /Users/haimmichael/PycharmProjects/python20210420/zipdemo.py
(123, 'mosh')
(234, 'gonen')
(345, 'ran')

Process finished with exit code 0
```

The `main()` Function

- When the Python runtime environment reads our source file, it executes the code it finds there.
- When the Python runtime is running a module (the source file) as the main program, it sets the special `__name__` variable to be with the value "`__main__`".
- When a file is being imported from another module, `__name__` will be set to the module's name.

The `main()` Function

```
def main():
    print("hello python!")

if __name__=="__main__":
    main()

print("yalla")
```