Namespace/context

- Naming system to make names unique to avoid ambiquity
- Python name spaces are implemented as dictionaries mapping names to values or objects.
- Some namespaces in python
  - global names of a module
  - local names in a function or method invocation
  - built-in names: this namespace contains built-in functions (e.g. abs(), cmp(), ...) and built-in exception names

Scope

The scope of a name is the area of a program where this name can be unambiguously used, for example inside of a function.

During program execution there are the following nested scopes available:

- the innermost scope is searched first and it contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope
- the next-to-last scope contains the current module's global names
- the outermost scope, which is searched last, is the namespace containing the built-in names

Global/Local variables

```
def foo():
    "accessing global variable"
    print "inside foo:" , s

s = "I am a global variable"
foo() #inside foo: I am a global variable
```

Global/Local variables:local

```
def foo():
    "accessing local variable"
    s = "I am local variable"
    print "inside foo:", s


s = "I am a global variable"
foo() #inside foo: I am local variable
print "outside foo:", s #outside foo: I am a global variable
```

```
def foo():
    "error: accessing local
variable before its definition"
    print s
    s = "I am local variable"


s = "I am a global variable"
foo() #UnboundLocalError....
```

Global/Local variables:global

```
def foo():
    "using global explicitly to show your intentions"
    global s
    s = "I am modified global variable"
    print "inside foo:", s

s = "I am a global variable"
foo()
print "outside foo:", s #outside foo: I am modified global variable
```

Global/Local variables:example

```
def foo(x,y):
    "example using local, global, arguments"
    x, y = -y, -x
    global a
    a = 10
    b = 20
    print "inside foo:", a,b,x,y


x, y = 1,2
a,b = 3,4
foo(x, y) #inside foo: 10 20 -2 -1
print "outside foo:",a,b,x,y #outside foo: 10 4 1 2
```

Scope

```
#  a and b are referring to the same list
a= [1,2,3]
b = a
id(a) == id(b)
a.append(4)
print "id(a) == id(b):",id(a) == id(b) #id(a) == id(b): True
print a                                #[1, 2, 3, 4]
print b                                #[1, 2, 3, 4]
```

Scope(call by reference/call by value)

```
a = [1,2,3,4]
def foo(x):
   "assignment makes a new list: no side effect"
   x = [5,6]
   return x

print foo.__doc__                                    #assignment makes a new list: no side effect
print "id(a) == id(foo(a)):",id(a) == id(foo(a))  #False
print foo(a)                                          #[5, 6]
print a                                               #[1, 2, 3, 4]
```

Scope(side effects)

```
a = [1,2,3,4]
def foo(x):
    "side effect: append changes the original list"
    x.append([5,6])
    return x

print foo.__doc__                          #side effect: append changes the original list
print "id(a) == id(foo(a)):",id(a) == id(foo(a))  #True
print foo(a)                                      #[1, 2, 3, 4, [5, 6], [5, 6]]
print a                                           #[1, 2, 3, 4, [5, 6], [5, 6]]
```

Scope(shallow copy)

```
a= [1,2,3, 4]
def foo(x):
    "send the shallow copy of original list to avoid side effects to some extent"
    x.append([5,6])
    return x


print foo.__doc__ #send the shallow copy of original list to avoid side effects to some extent
print "id(a) == id(foo(a[:])):",id(a) == id(foo(a[:])) #False
print foo(a[:])   #[1, 2, 3, 4, [5, 6]]
print a           #[1, 2, 3, 4]
```

Scopes(still side effects)

```
a = [2,3,'hello',[4,'yes']]
def foo(x):
    "still side effects on sublists in original list"
    x[3][0]="GREAT"
    return x

print foo.__doc__
print "id(a) == id(foo(a[:]))", id(a) == id(foo(a[:])) #False
print "id(a[3]) == id(foo(a)[3]):",id(a[3]) == id(foo(a[:])[3]) #True
print foo(a[:])     #[2, 3, 'hello', ['GREAT', 'yes']]
print a              #[2, 3, 'hello', ['GREAT', 'yes']]
```

Scopes(deep copy)

```
from copy import deepcopy
a = [2,3,'hello',[4,'yes']]
def foo(x):
    "deep copy eliminates side effects on sublists in original list"
    x[3][0]="GREAT"
    return x

print foo.__doc__
print "id(a[3]) == id(foo(deepcopy(a))):",id(a[3]) ==id(foo(deepcopy(a))) #False
print foo(deepcopy(a))    #[2, 3, 'hello', ['GREAT', 'yes']]
print a                   #[2, 3, 'hello', [4, 'yes']]
```

Scopes(nested functions)

```
a=0
def foo():
    a=1
    def bar():
        a=2
        def baz():
            a=3
            print "inside baz:",a     #inside baz: 3
        baz()
        print "inside bar:",a         #inside bar: 2
    bar()
    print "inside foo:",a             #inside foo: 1

foo()
print "outside foo:",a                #outside foo: 0
```

Scopes(closures)

```
def foo():
    "An example to create calling env. different from creation time env for function baz."
    a=1
    def bar():
        "bar returns a function baz"
        a=2
        def baz():
            "baz simply returns a as it sees"
            print "inside baz: a = ",a
            return a
        return baz
    a = 10
    print foo.__doc__           #An example to create calling env. different from creation time env for function baz.
    print "inside foo: a = ",a  #inside foo: a =  10
    print bar.__doc__           #bar returns a function baz
    a = 11
    f = bar()
    print f.__doc__             #baz simply returns a as it sees
    f()                         #inside baz: a =  2

foo()
```