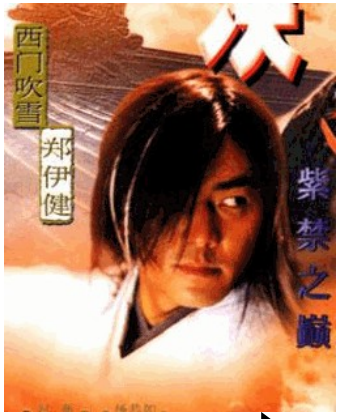
Python的名字、作用域与名字空间

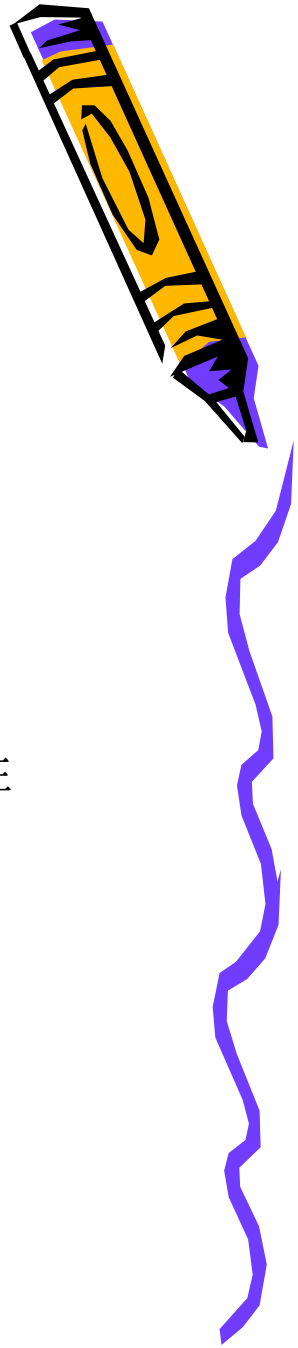
Robert Chen

2007.09.01



西门吹雪 紫禁之巅 决战 叶孤城

名字（符号）的意义



名字具有任意性，当这种任意性被限制时，就趋向于静态语言

Python

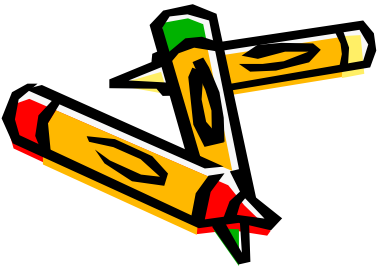
```python
def demo(a, b):
    return a + b
```

C++

```cpp
public int demo(int a, int b) {
    return a + b;
}
```

Python 中，符号（名字）的受限更少，具有比 C++ 更强的动态性

在 Python 中，名字的解析始终是一个核心的问题



名字的解析

通过名字的解析，我们才能获得名字背后真实的事物（对象），从而确定符号串的语义

```
1   a = 1
2   b = 2
3 - def demo(a, b):
4       return a + b
```

```
1   a = "hello"
2   b = "python"
3 - def demo(a, b):
4       return a + b
```

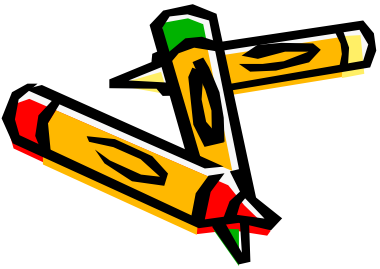
两类解析

```
1   a = 1
2   b = 2
3 - def demo(a, b):
4       return a + b
```

名字解析

```
1   import sys
2 - print sys.path
```

属性解析



名字空间

名字的解析实质上是一个搜索的过程：在名字空间中搜索名字

```
1   code = """
2  -def show():
3       print a
4
5   show()
6   """
7
8  -exec code in {'a' : 1}
```

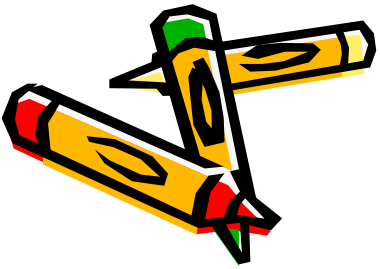
```
D:\Python\91>python exec_demo.py
1
```

名字空间：程序运行时名字的栖身之处

在 Python 内部，使用 dict 作为运行期间的名字空间

名字解析 ：

```
10  namespace = {...}
11 -def resolve_name(name):
12 -    if name in namespace:
13          return namespace[name]
```



属性解析

```
1  import sys
2  print sys.path
```

Python 中能够参与属性解析的对象都有一个内部的名字空间 ：__dict__

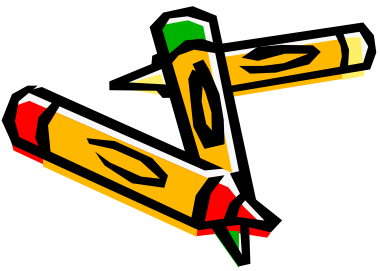
module object,
class object,
instance object
function object
……

```
>>> def demo():
        pass

>>> demo.__dict__
{}
>>> demo.__dict__['name'] = 'python'
>>> demo.__dict__
{'name': 'python'}
>>> demo.name
'python'
```

[function object's namespace]

obj.name →

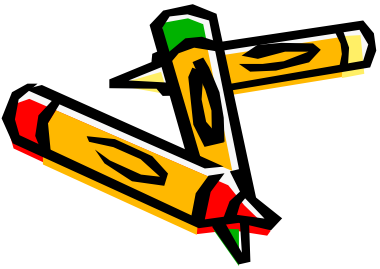
```
def resolve_name(obj, name):
    if name in obj.__dict__:
        return obj.__dict__[name]
```



属性解析 (2)

```
class A(object):
    def show(self):
            print 'hello A'
```

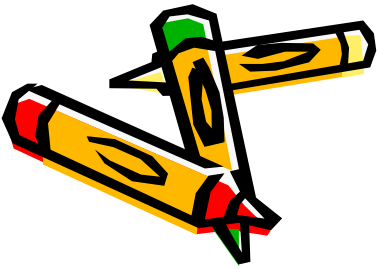
```
>>> A.show
<unbound method A.show>
>>> A.__dict__['show']
<function show at 0x00D38E70>
>>>
>>> A.__dict__['show']()

Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    A.__dict__['show']()
TypeError: show() takes exactly 1 argument (0 given)
>>> A.__dict__['show'](1)
hello A
>>>
>>>
>>> A.__dict__['show'] = None

Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    A.__dict__['show'] = None
TypeError: 'dictproxy' object does not support item assignment
```



属性解析 (3)

```
>>> class A(object):
        def showA(self):
                print 'hello A'


>>> class B(A):
        def showB(self):
                print 'hello B'
```

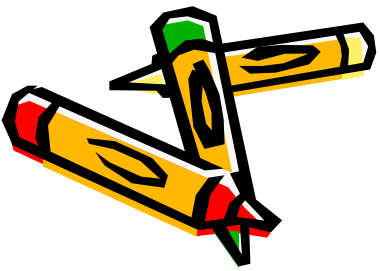
```
>>> B.showA
<unbound method B.showA>
>>> B.__dict__['showA']

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    B.__dict__['showA']
KeyError: 'showA'
>>> B.__dict__.keys()
['__module__', '__doc__', 'showB']
```



```
>>> B.__bases__
(<class '__main__.A'>,)
>>> B.__bases__[0].__dict__['showA']
<function showA at 0x00D38E30>
>>>
```

属性解析 (4)

```
def resolve_name(klass, name):
    if name in klass.__dict__:
        return klass.__dict__[name]
    else:
        base_klasses = klass.__bases__
        for base_klass in base_classes:
            obj = resolve_name(base_klass, name)
            if obj:
                return obj
```



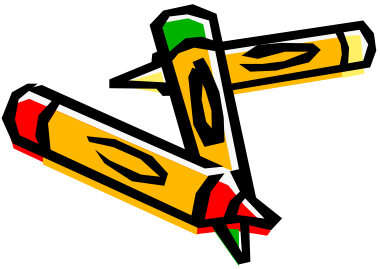
名字解析— Python

```
1  a = "out"
2
3  def demo():
4      print a
5
6  demo()
```

```
D:\Python\91>python demo.py
out
```

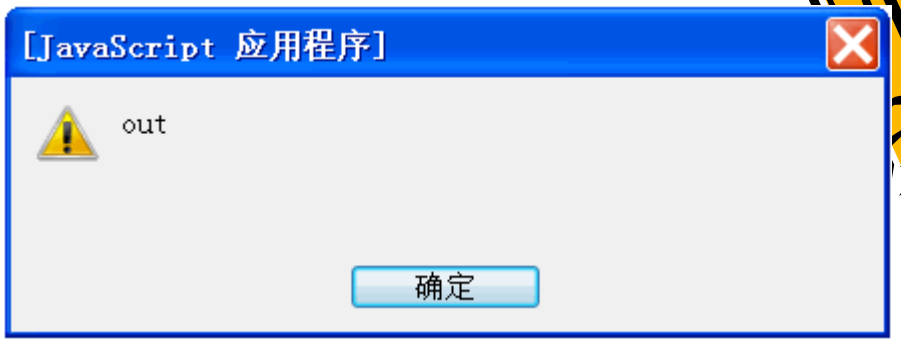
```
1  a = "out"
2
3  def demo():
4      print a
5      a = "in"
6
7  demo
```

```
D:\Python\91>python demo.py
Traceback (most recent call last):
  File "demo.py", line 7, in <module>
    demo()
  File "demo.py", line 4, in demo
    print a
UnboundLocalError: local variable 'a' referenced before assignment
```



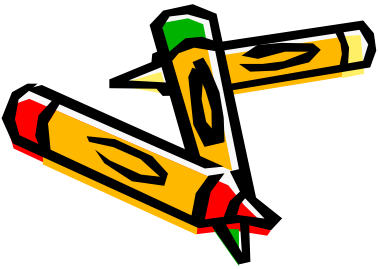
名字解析— JavaScript

```
<script type="text/javascript">
    var a = "out";
    function demo() {
        alert(a);
    }
    demo();
</script>
```




```
<script type="text/javascript">
    var a = "out";
    function demo() {
        alert(a);
        var a = "in";
    }
    demo();
</script>
```

在一个 module （.py 文件）内部的名字解析拥有特殊的规则 ： 最内嵌套作用域规则



作用域

约束 ：名字与值的关联关系，在 python 中，就是 dict 中的一个 (key, value)

一个约束起作用的那一段程序正文区域称为这个**约束的作用域**。

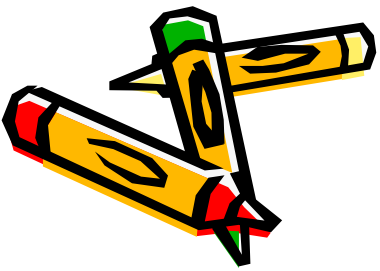
而一个**作用域**则是指一段程序正文区域。
在这个区域里，可能有很多个约束在起作用；
一旦出了这个正文区域，这些约束都不起作用了

```
1  a = 1 #[1]
2  def f():
3      a = 2 #[2]
4      print a #[3]输出：2
5  print a #[4]输出：1
```

3 、4 两行代码构成一个作用域

约束【2 】不能跨越作用域，影响第5 行代码的输出

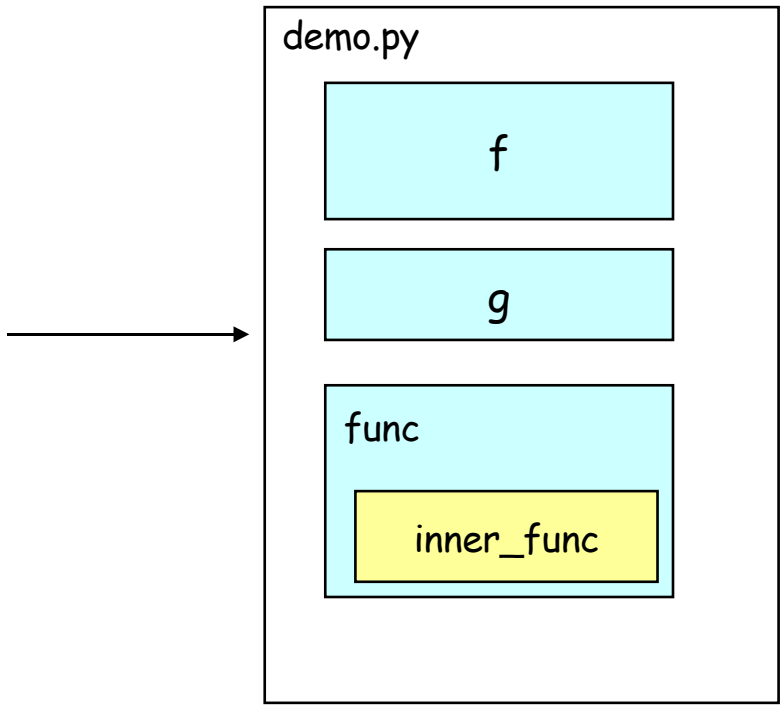
在一个 module 中，可能存在多个作用域，每一个作用域对应一个名字空间



嵌套作用域

```
1   a = 1
2
3   def f():
4       a = 2
5       print a
6
7   def g():
8       pass
9
10  def func():
11      name = 1
12      def inner_func():
13          pass
14
15
16  print a
```

demo.py



最内嵌套作用域规则：

由一个赋值语句引进的名字在这个赋值语句所在的作用域里是可见（起作用）的，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的引进同样名字的另一条赋值语句所**遮蔽**。

作用域的静态特性

作用域是语法层面的概念，这意味着 Python 代码的行为将取决于其在源文件中的位置。在Python 中，一个约束在程序正文的某个位置是否起作用，是由该约束在文本中的位置唯一决定的，而不是在运行时动态决定的。

```
1   a = 1
2
3   def f():
4       a = 2
5       def g():
6           print a #[1]:输出2
7       return g
8
9   func = f()
10  func()
```

closure

```
1   owner = 'module2'
2
3   def show_owner():
4       print owner
```

module2.py

```
1   import module2
2
3   owner = 'module1'
4   module2.show_owner()
```

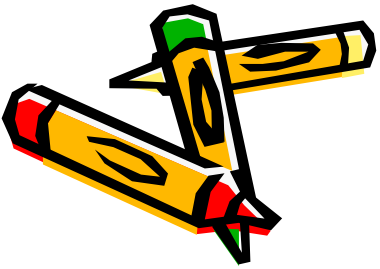
module1.py

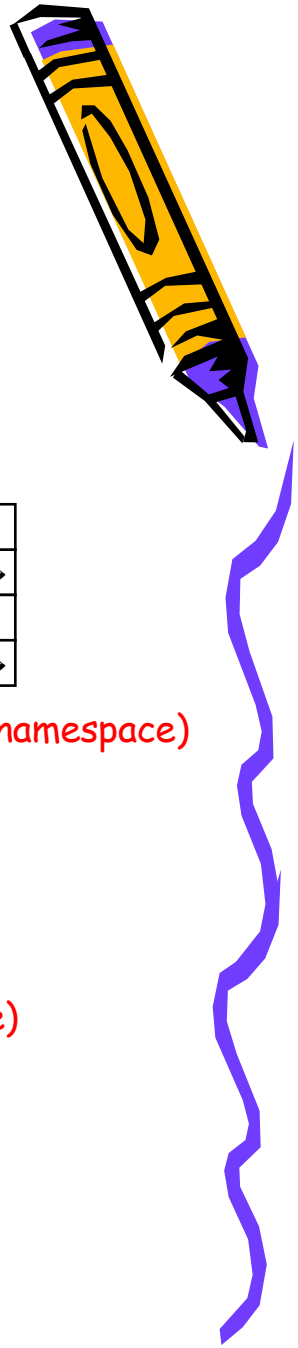
```
D:\Python\91>python module1.py
module2
```

?

LEGB 解析链

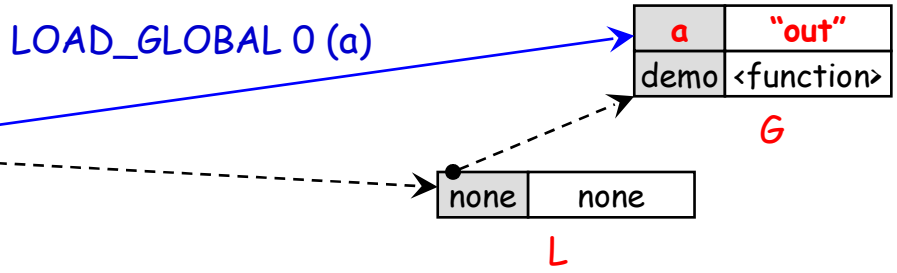
```
1   a = 1
2
3   def f():
4       a = 2
5       print a
6
7   def g():
8       pass
9
10  def func():
11      name = 1
12      def inner_func():
13          pass
14
15
16  print a
```



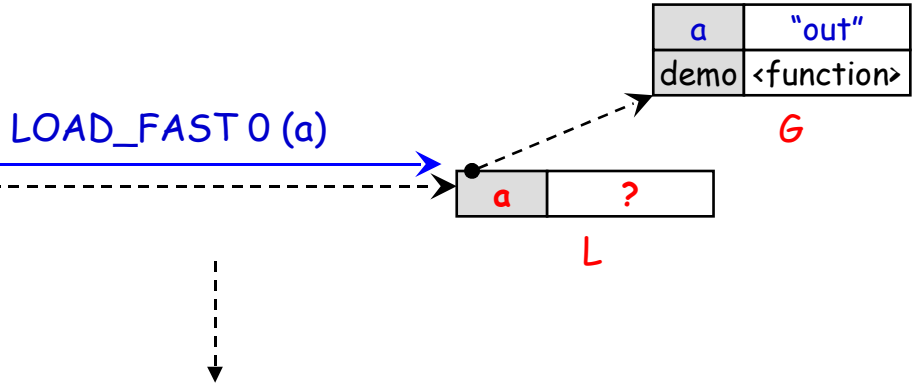


案例剖析

```
1  a = "out"
2
3  def demo():
4      print a
5
6  demo()
```




```
1  a = "out"
2
3  def demo():
4      print a
5      a = "in"
6
7  demo
```




```
D:\Python\91>python demo.py
Traceback (most recent call last):
  File "demo.py", line 7, in <module>
    demo()
  File "demo.py", line 4, in demo
    print a
UnboundLocalError: local variable 'a' referenced before assignment
```

案例剖析 (2)

```
1   owner = 'module2'
2
3   def show_owner():
4       print owner
```

module2.py

LOAD_GLOBAL 0 (owner)

owner	"module2"
show_owner	<function>

G

```
1   import module2
2
3   owner = 'module1'
4   module2.show_owner()
```

module1.py

```
1   a = 1
2
3   def f():
4       a = 2
5       def g():
6           print a #[1]:输出2
7       return g
8
9   func = f()
10  func()
```

closure

最内嵌套作用域规则是“闭包”的结果?
“闭包”是最内嵌套作用域规则的实现方案?

最内嵌套作用域规则是语言设计时的设计策略
形而上的“道”

而闭包则实现语言时的一种方案
形而下的“器”

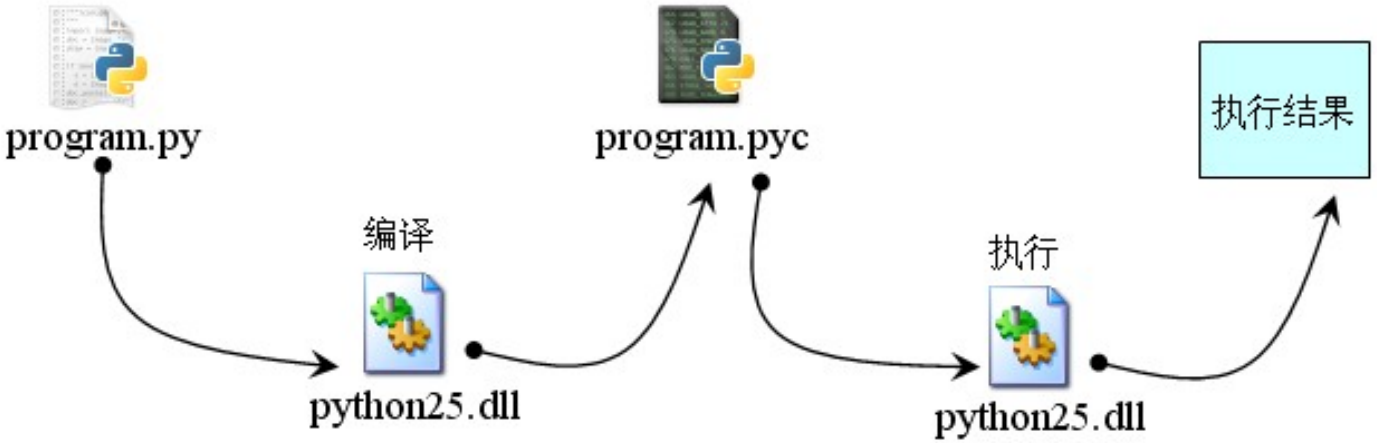
小结解析

```
1   import sys
2
3   msg = 'hello world'
4
5   class A(object):
6       def set(self, name):
7           self.name = name #"self":名字解析；"self.name":属性解析；"name":名字解析
8
9       def show(self, show_name):
10          if show_name: #"show_name":名字解析
11              print self.name #"self":名字解析；"self.name":属性解析
12          else:
13              print msg #"msg":名字解析
14
15  a = A() #"A":名字解析
16  a.set('python') #"a":名字解析；"a.set":属性解析
17  a.show(False)  #"a":名字解析；"a.show":属性解析
```

最内嵌套作用域规则：

由一个赋值语句引进的名字在这个赋值语句所在的作用域里是可见（起作用）的，而且在其内部嵌套的每个作用域里也可见，除非它被嵌套于内部的引进同样名字的另一条赋值语句所遮蔽。

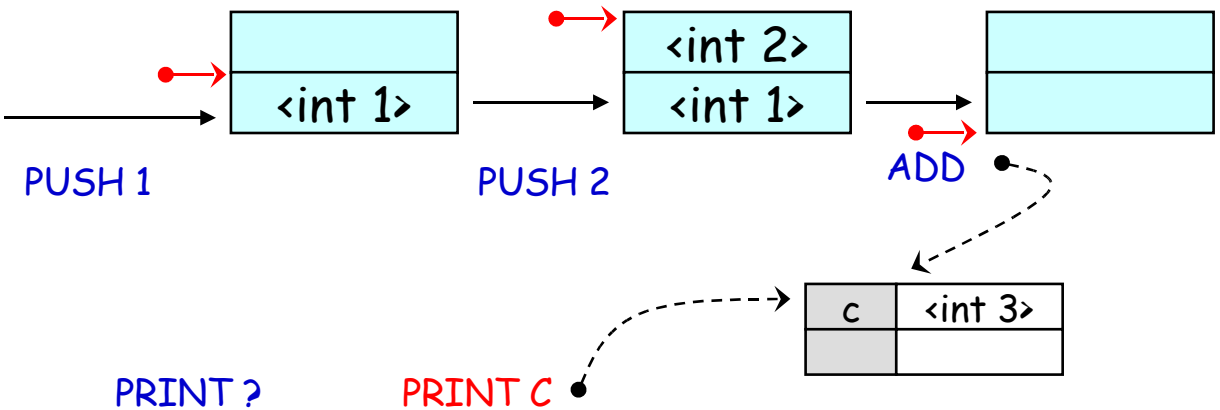
Python 运行模型



Python 虚拟机藏身于 python25.dll 中，它是一种基于堆栈的虚拟机，类似于 JVM

```
1  a = 1
2  b = 2
3  c = a + b
4  print c
```

PUSH 1
PUSH 2
ADD
PRINT

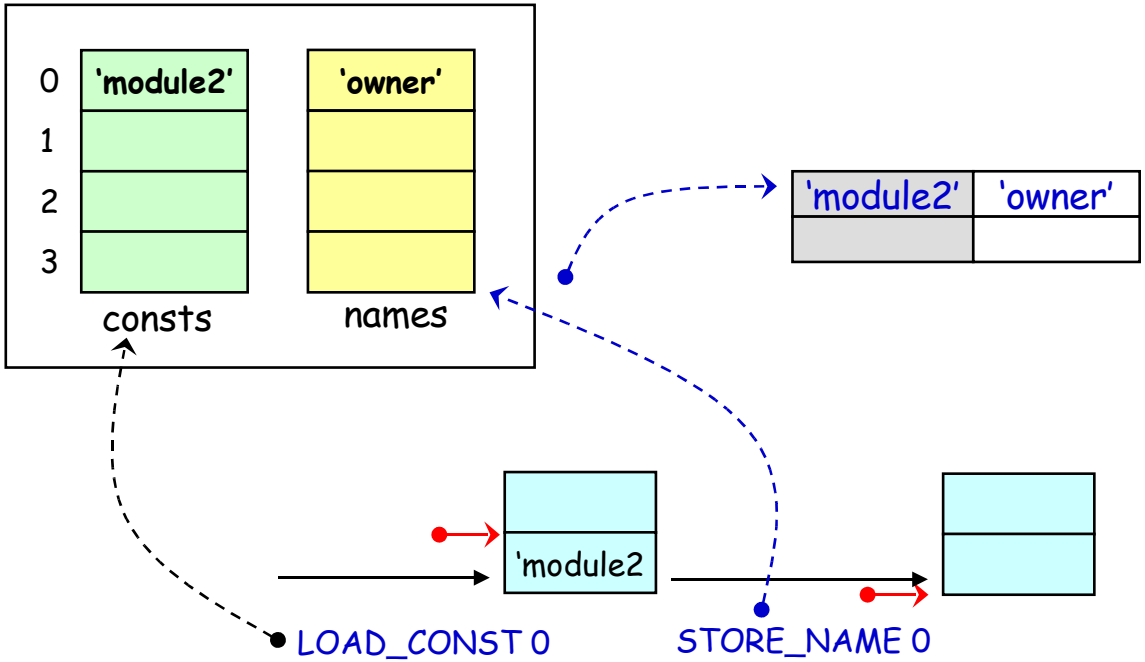


名字的创建—赋值语句

创建一个对象，并为这个对象取一个名字

创建一个对象，将对象及其名字放入某个名字空间中

```
owner = 'module2'
LOAD_CONST    0  ('module2')
STORE_NAME    0  (owner)
```



庞大的赋值语句族

def func():

```
LOAD_CONST     0  ([code object func])
MAKE_FUNCTION      0
STORE_NAME     0  (func)
```

from os import path:

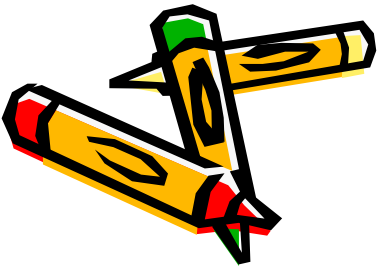
```
LOAD_CONST     2  (('path',))
IMPORT_NAME    1  (os)
IMPORT_FROM    2  (path)
STORE_NAME     2  (path)
```

l = [1, 'python']

```
LOAD_CONST     1  (1)
LOAD_CONST     2  ('python')
BUILD_LIST     2
STORE_NAME     1  (l)
```

class A(object):

```
LOAD_CONST     0  ('A')
LOAD_NAME      0  (object)
BUILD_TUPLE    1
LOAD_CONST     1  ([code object A])
MAKE_FUNCTION      0
CALL_FUNCTION      0
BUILD_CLASS
STORE_NAME     1  (A)
```

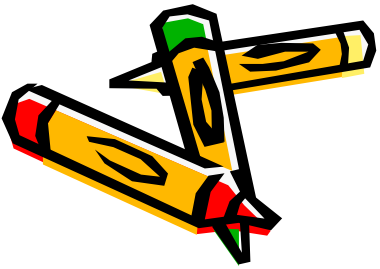


import 的特殊行为

```
>>> import sys
>>> locals().keys()
['__builtins__', '__name__', 'sys', '__doc__']
>>>
>>> 'django' in sys.modules
False
```

```
>>> import django
>>> locals().keys()
['__builtins__', 'django', 'sys', '__name__', '__doc__']
>>>
>>> 'django' in sys.modules
True
>>> sys.modules['django']
<module 'django' from 'C:\Python25\lib\site-packages\django\__init__.pyc'>
```

```
>>> del django
>>> locals().keys()
['__builtins__', 'sys', '__name__', '__doc__']
>>>
>>> 'django' in sys.modules
True
>>> sys.modules['django']
<module 'django' from 'C:\Python25\lib\site-packages\django\__init__.pyc'>
```



特殊的赋值语句

函数的参数传递可以视为一种赋值语句

```
1  info = "ruby"
2
3  def show():
4      print info
5
6  def show_info(info):
7      print info
8
9  show()
10 show_info("python")
```

'info'	'python'

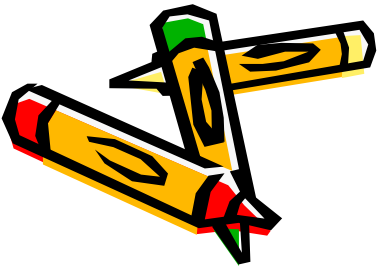
LOAD_FAST 0 (info)

'a'	'a'
'b'	'b'
'args'	<list>
'dicts'	<dict>

'c'	'd'

'name'	'python'
'age'	<int 15>

```
1  def demo(a, b, *args, *dicts):
2      pass
3
4  demo("a", "b", "c", "d", name="python", age=15)
```



Thanks
Any Question ?