

## 19 面试必问概念精讲：作用域与名字空间-慕课专栏

---

 imooc.com/read/76/article/1914

作为 *Python* 面试官，每次面试中我几乎都会和候选人聊起 **作用域** 以及 **名字空间** 等基本概念。但就算这么基础的内容，也有不少人没有完全掌握，也因此与工作机会失之交臂。

```
PI = 3.14
```

```
def circle_area(r):  
    return PI * r ** 2
```

```
class Dog(object):  
  
    def __init__(self, name):  
        self.name = name  
  
    def yelp(self):  
        print('woof, i am', self.name)
```

以这个程序为例，代码中出现的每个变量的作用域分别是什么？程序中总共涉及几个名字空间？*Python* 以怎样的顺序查找一个变量呢？为了解答这些问题，需要对 *Python* 变量的作用域以及名字空间有准确的认识。

## 名字绑定

---

### 赋值

---

在 *Python* 中，变量只是一个与实际对象绑定起来的字，变量定义本质上就是建立名字与对象的约束关系。因此，赋值语句本质上就是建立这样的约束关系，将右边的对象与左边的名字绑定在一起：

```
a = 1
```

我经常在面试中问：除了赋值语句，还有哪些语句可以完成名字绑定？能准确回答的候选人寥寥无几。实际上，除了赋值语句外，*Python* 中还有好几类语句均与名字绑定相关，我们接着——介绍。

### 模块导入

---

我们导入模块时，也会在当前上下文创建一个名字，并与被导入对象绑定：

```
import xxx  
from xxx import yyy
```

### 函数、类定义

---

我们定义函数/类时，本质上是创建了一个函数/类对象，然后将其与函数/类名绑定：

```
def circle_area(r):  
    return PI * r ** 2  
  
class Dog(object):  
    pass
```

## as关键字

---

除此此外， `as` 关键字也可以在当前上下文建立名字约束关系：

```
import xxx as yyy  
from xxx import yyy as zzz  
  
with open('/some/file') as f:  
    pass  
  
try:  
  
except SomeError as e:
```

以上这几类语句均可在当前上下文建立名字约束，有着与赋值语句类似的行为，因此可以看作是 **广义的赋值语句**。

## 作用域

---

现在问题来了，一个名字引入后，它的可见范围有多大呢？

我们以一个面试真题开始讨论：以下例子中 3 个 `print` 语句分别输出什么？

```
a = 1  
  
def f1():  
    print(a)  
  
def f2():  
    a = 2  
    print(a)  
  
print(a)
```

例子中，第1行引入的名字 `a` 对整个模块都可见，第4行和第10行均可访问到它，因此这两个地方输出 `1`；而第7行引入的名字 `a` 却只有函数 `f2` 内部可以访问到，第8行优先访问内部定义的 `a`，因此这里将输出 `2`。

由此可见，在不同的代码区域引入的名字，其影响范围是不一样的。第 1 行定义的 `a` 可以影响到 `f1`，而 `f2` 中定义的 `a` 却不能。再者，一个名字可能在多个代码区域中定义，但最终只能使用其中一个。

一个名字能够施加影响的程序正文区域，便是该名字的 **作用域**。在 **Python** 中，一个名字在程序中某个区域能否起作用，是由名字引入的位置决定的，而不是运行时动态决定的。因此，**Python** 具有 **静态作用域**，也称为 **词法作用域**。那么，程序的作用域是如何划分的呢？

**Python** 在编译时，根据语法规则将代码划分为不同的 **代码块**，每个代码块形成一个 **作用域**。首先，整个 **.py** 文件构成最顶层的作用域，这就是 **全局作用域**，也称为 **模块作用域**；其次，当代码遇到 **函数定义**，函数体成为当前作用域的 **子作用域**；再次，当代码遇到 **类定义**，类定义体成为当前作用域的子作用域。

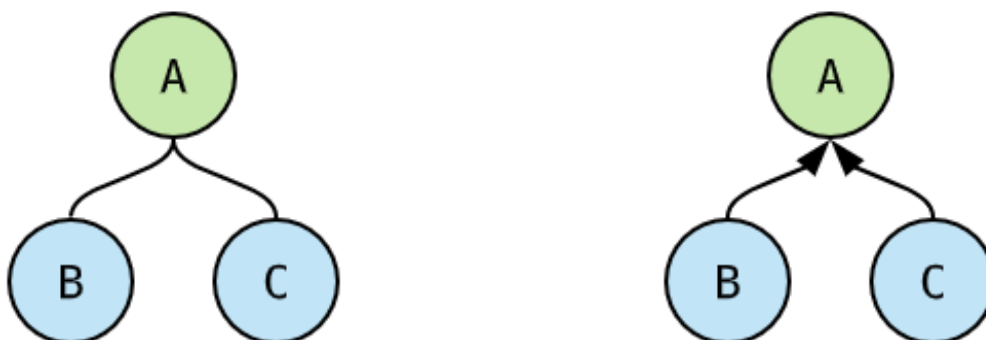
一个名字在某个作用域引入后，它的影响范围就被限制在该作用域内。其中，全局作用域对所有直接或间接内嵌于其中的子作用域可见；函数作用域对其直接子作用域可见，并且可以传递。

按照这个划分方式，真题中的代码总共有3个作用域：**A** 为最外层作用域，即全局作用域；**f1** 函数体形成作用域 **B**，是 **A** 的子作用域；**f2** 函数体又形成作用域 **C**，也是 **A** 的子作用域。

```
1 a = 1
2
3 def f1():
4     print(a)
5
6 def f2():
7     a = 2
8     print(a)
9
10 print(a)
11
```

作用域 **A** 定义的变量 **a** 对于对 **A** 及其子作用域 **B**、**C** 可见，因此 **f1** 也可以访问到。理论上，**f2** 也可以访问到 **A** 中的 **a**，只不过其作用域 **C** 也定义了一个 **a**，优先访问本作用域内的。**C** 作用域内定义的任何名字，对 **A** 和 **B** 均不可见。

**A B C** 三个作用域嵌套关系如左下所示，访问关系如右下所示：



箭头表示访问关系，例如作用域 **B** 中的语句可以访问到作用域 **A** 中的名字，反过来则不行。

## 闭包作用域

这个例子借助闭包实现提示信息定制功能：

```
pi = 3.14
```

```
def circle_area_printer(hint):
```

```
    def print_circle_area(r):  
        print(hint, pi * r ** 2)
```

```
    return print_circle_area
```

```
circle_area_en = circle_area_printer('Circle Area:')
```

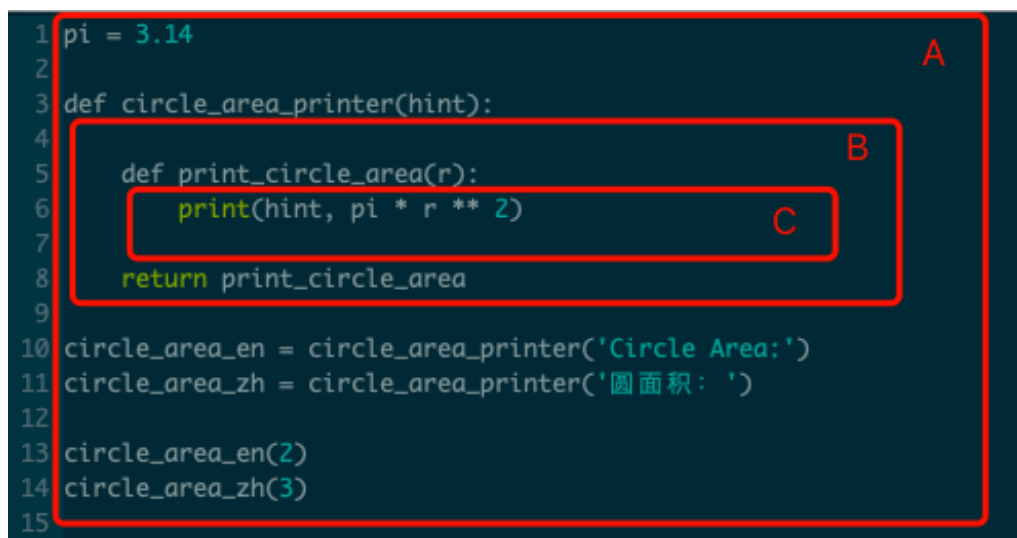
```
circle_area_zh = circle_area_printer('圆面积：')
```

```
circle_area_en(2)
```

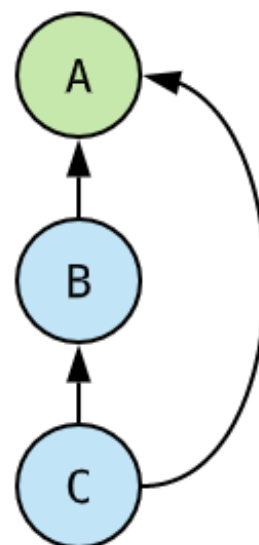
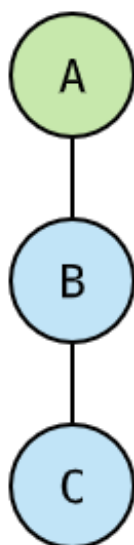
```
circle_area_zh(3)
```

根据前面介绍的规则，我们对代码进行作用域划分，结果如下：

```
1 pi = 3.14  
2  
3 def circle_area_printer(hint):  
4     def print_circle_area(r):  
5         print(hint, pi * r ** 2)  
6     return print_circle_area  
7  
8  
9  
10 circle_area_en = circle_area_printer('Circle Area:')  
11 circle_area_zh = circle_area_printer('圆面积：')  
12  
13 circle_area_en(2)  
14 circle_area_zh(3)  
15
```



A B C 三个作用域嵌套关系如左下所示，访问关系如右下所示：



毫无疑问，*B C* 均在全局作用域 *A* 内，因此都可以访问到 *A* 中的名字。由于 *B* 是函数作用域，对其子作用域 *C* 可见。因此，*hint* 属于 *B* 作用域，而位于 *C* 作用域的语句可以访问它，也就不奇怪了。

## 类作用域

我们接着以一个简单的类为例，考察类作用域：

```
slogan = 'life is short, use python.'
```

```
class Dog(object):
```

```
    group = ''
```

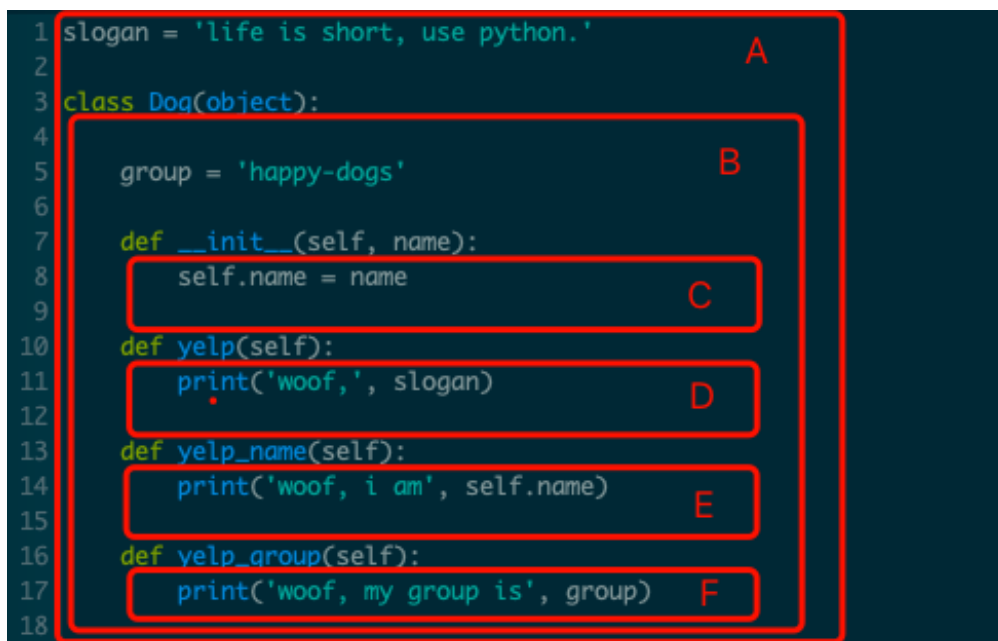
```
    def __init__(self, name):
        self.name = name
```

```
    def yelp(self):
        print('woof,', slogan)
```

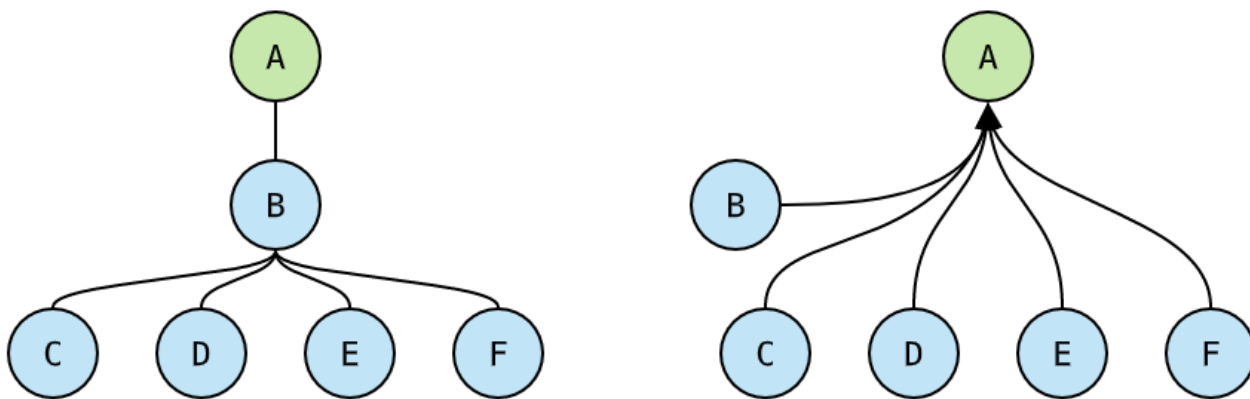
```
    def yelp_name(self):
        print('woof, i am', self.name)
```

```
    def yelp_group(self):
        print('woof, my group is', group)
```

根据前面介绍的规则，我们对代码进行作用域划分，结果如下：



其中，*B* 对应着类定义体，暂且叫做类作用域。各个作用域嵌套关系以及访问关系分别如下：



同样，全局作用域 *A* 对其他所有内嵌于其中的作用域可见。因此，函数 *yelp* 作用域 *D* 可以访问到全局作用域 *A* 中的名字 *slogan*。但是由于 *B* 不是函数作用域，对其子作用域不可见。因此，*yelp\_group* 函数作用域 *F* 访问不到类作用域 *B* 中的名字 *group*，而 *group* 在全局作用域 *A* 中未定义，第 17 行便抛异常了。

## 复杂嵌套

### 函数-类

在 *Python* 中，类可以动态创建，甚至在函数中返回。在函数中创建并返回类，可以按函数参数对类进行动态定制，有时很有用。那么，这种场景中的作用域又该如何划分呢？我们一起来看看一个简单的例子：

```

slogan = 'life is short, use python.'

def make_dog(group_name):

    class Dog(object):

        group = group_name

        def __init__(self, name):
            self.name = name

        def yelp(self):
            print('woof,', slogan)

        def yelp_name(self):
            print('woof, i am', self.name)

        def yelp_group(self):
            print('woof, my group is', self.group)

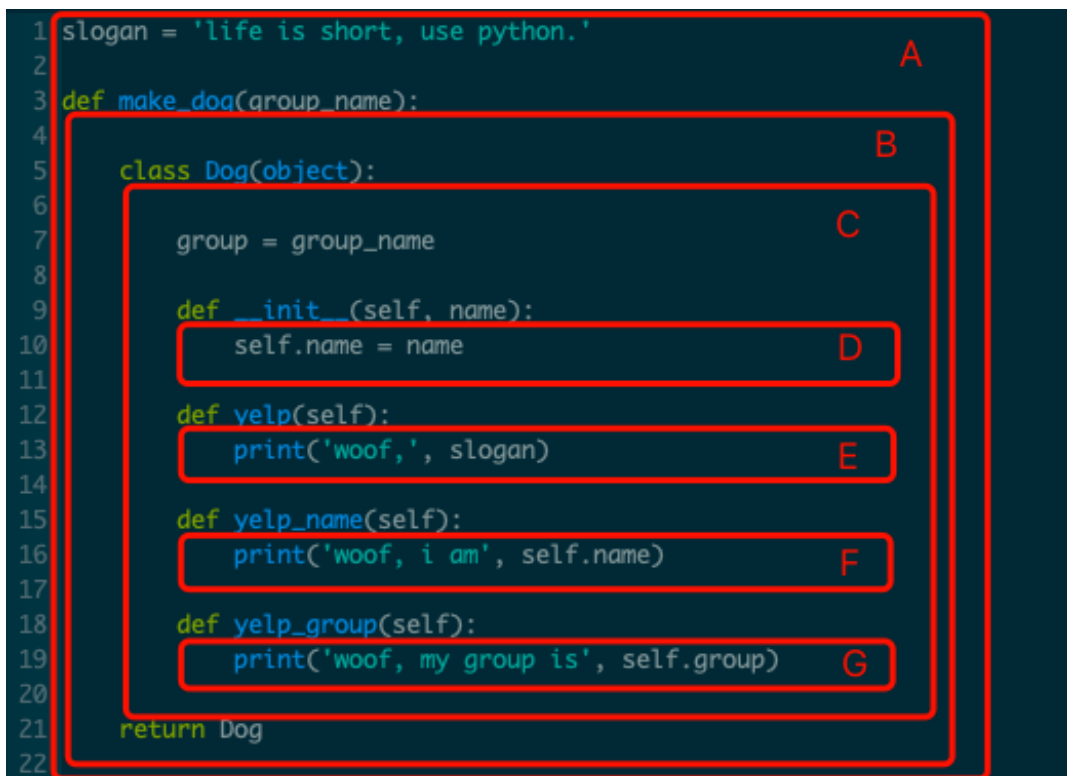
    return Dog

if __name__ == '__main__':
    Dog = make_dog('silly-dogs')

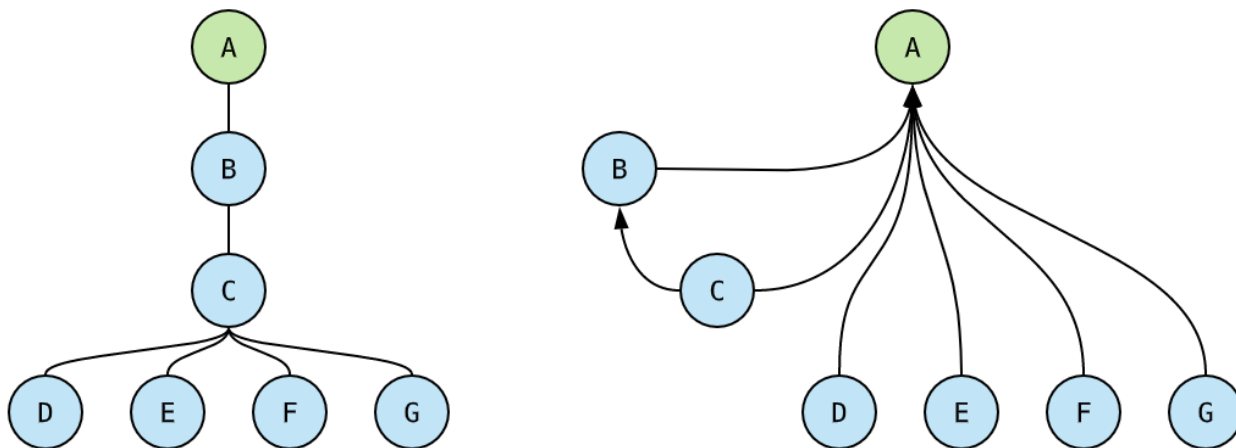
    tom = Dog(name='tom')
    tom.yelp_group()

```

这个例子借助函数实现类属性 *group* 动态定制，以不同的 *group\_name* 调用函数即可获得不同的 *Dog* 类。根据前面介绍的规则，我们对代码进行作用域划分，结果如下：



各个作用域嵌套关系以及访问关系分别如下：



同样，全局作用域  $A$  对其他所有内嵌于其中的作用域可见。由于作用域  $B$  是函数作用域，因此子作用域  $C$  中的语句能够范围  $B$  中的名字。

经过以上分析，例子程序应该输出 `woof, my group is silly-dogs`。

## 类-类

我们接着考察类嵌套的情形：

```
class Foo(object):

    bar_name = 'BAR'

    class Bar(object):

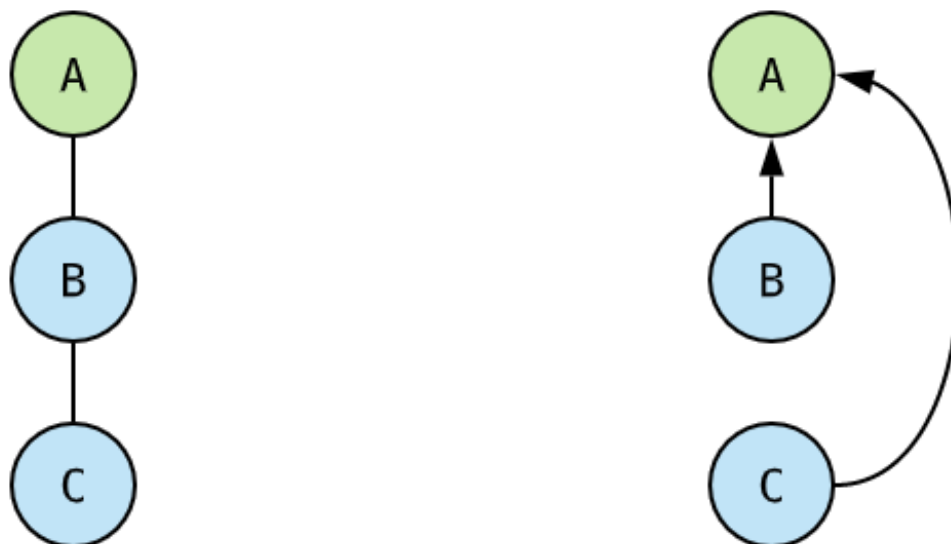
        name = bar_name

if __name__ == '__main__':
    bar = Foo.Bar()
    print(bar.name)
```

这个例子没有实际含义，纯粹为了考察嵌套类作用域问题。根据前面介绍的规则，我们对代码进行作用域划分：

各个作用域嵌套关系以及访问关系分别如下：





这里需要注意的是，类作用域 *B* 对子作用域 *C* 不可见，因此第 7 行就抛异常了。

## 名字空间

**作用域** 是语法层面的概念，是静态的。当程序开始执行后，作用域中的名字绑定关系需要存储在某个地方，这个地方就是 **名字空间**。由于名字绑定关系是有 **名字** 和 **对象** 组成的键值对，因而 *dict* 对象是理想的存储容器。

接下来，我们以计算圆面积的例子程序接着考察作用域背后的运行时实体—— **名字空间**。

```
pi = 3.14

def circle_area_printer(hint):

    def print_circle_area(r):
        print(hint, pi * r ** 2)

    return print_circle_area
```

## Globals

在 *Python* 中，每个 **模块** 背后都有一个 *dict* 对象，用于存储 **全局作用域** 中的名字，这就是 **全局名字空间** (*Globals*)。在上面这个例子中，全局名字空间至少包含两个名字：*pi* 和 *circle\_area\_printer*。由此可见，*Python* 的全局名字空间是以 **模块** 为单位划分的，而不是全局统一的。

其他模块如果也需要使用 *pi*，需要借助 *import* 语句将其导入。模块导入后，我们得到一个模块对象(假设例子代码位于 *circle.py*)：

```
>>> import circle
>>> type(circle)
<class 'module'>
```

接着，我们通过模块对象属性查找，便可得到 *circle* 模块全局名字空间中的 *pi*：

```
>>> print(circle.pi)
3.14
```

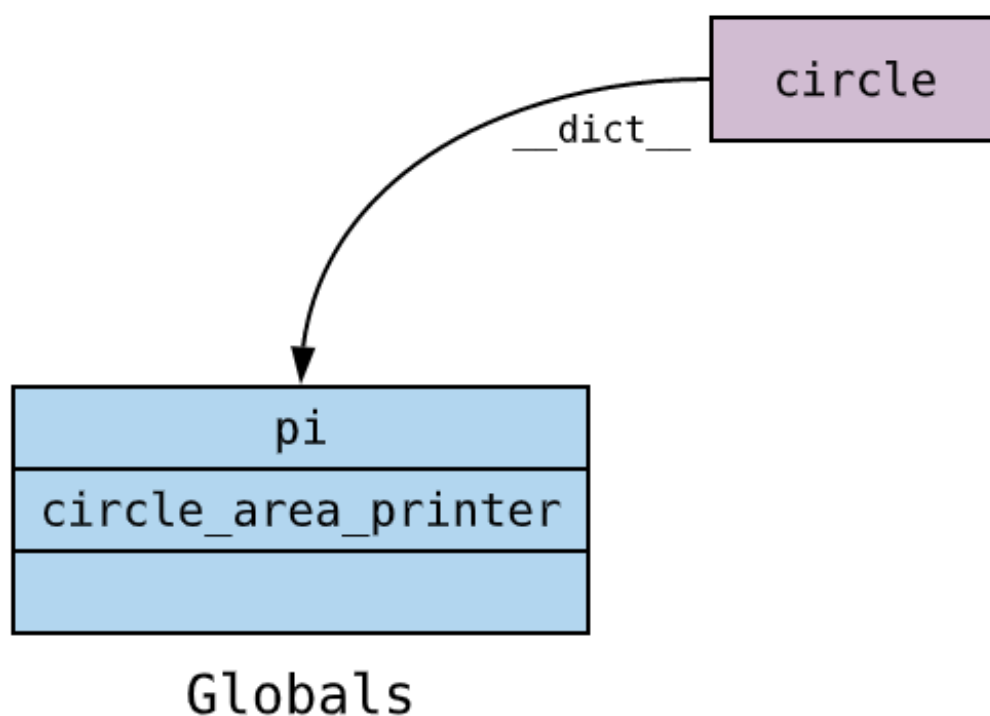
此外，我们还可以进一步确认 *\_circle\_area\_printer* 函数也可以通过模块对象属性的方式来访问：

```
>>> dir(circle)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'circle_area_printer', 'pi']
```

在 *Python* 中，一个对象可以访问哪些属性，称为对象的 **属性空间**。由于属性也是键值对，因此一般也是用 *dict* 来存储。通过观察以上代码行为，我们得到一个结论：模块的 **属性空间** 以及 **全局名字空间** 是同一个东西，都藏身于同一个 *dict* 对象。那么，我们怎么找到这个特殊的 *dict* 对象呢？答案是：

```
circle.__dict__
```

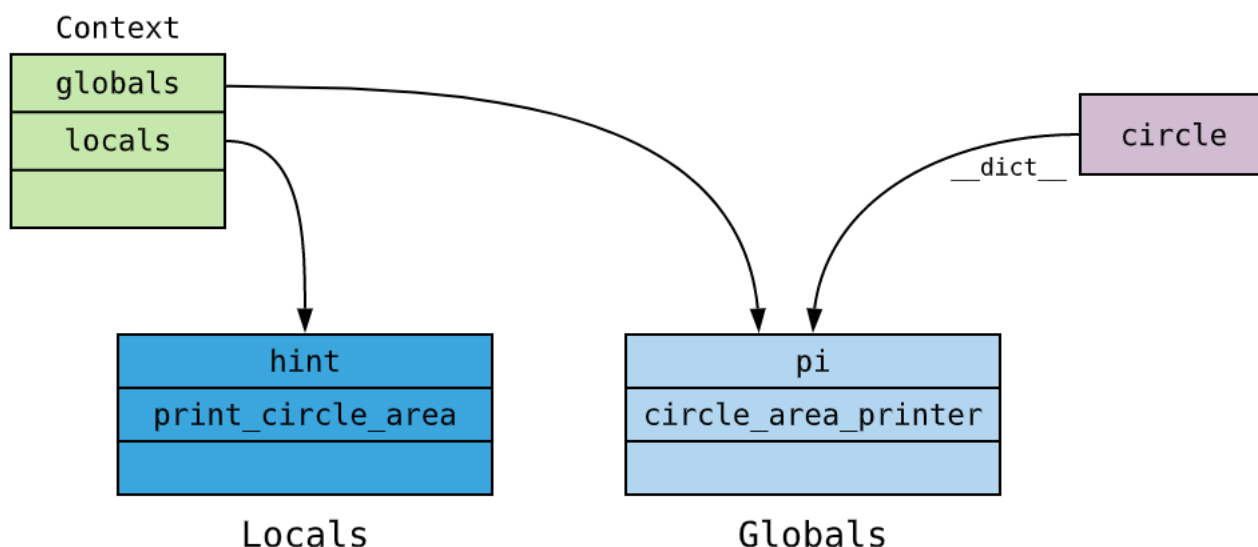
全局名字空间藏身于模块对象背后的 *dict* 对象中：



## Locals

*Python* 执行一个作用域内的代码时，需要一个容器来存储当前作用域内的名字，这就是 **局部名字空间** (*Locals*)。

当 *Python* 执行函数 *circle\_area\_printer* 时，将分配一个栈帧对象保存上线文信息以及执行状态，这个栈帧对象就是后面章节要介绍的 *PyFrameObject* 。作为代码执行时必不可少上下文信息之一，全局名字空间和局部名字空间也在栈帧对象上记录：

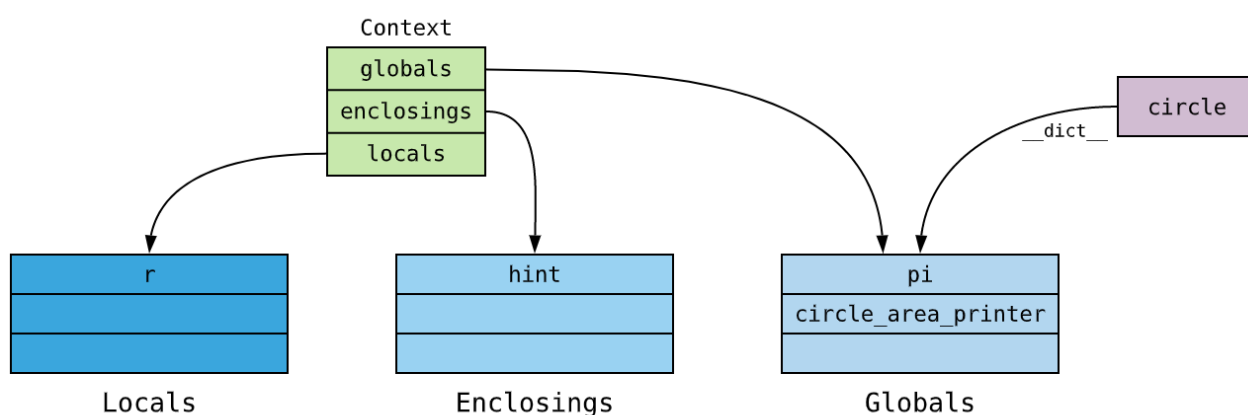


代码语句中涉及的名字查找，均在这个两个名字空间中进行：先查找局部名字空间，再查找全局名字空间。

## Enclosings

在作用域存在嵌套的情况下，*Python* 将内层代码块中依赖的所有外层名字存储在一个容器内，这就是 **闭包名字空间** (*Enclosings*)。

当 *Python* 执行函数 *print\_circle\_area* 时，依赖上层作用域中的名字 *hint*，*hint* 保存与一个独立的名称空间中：

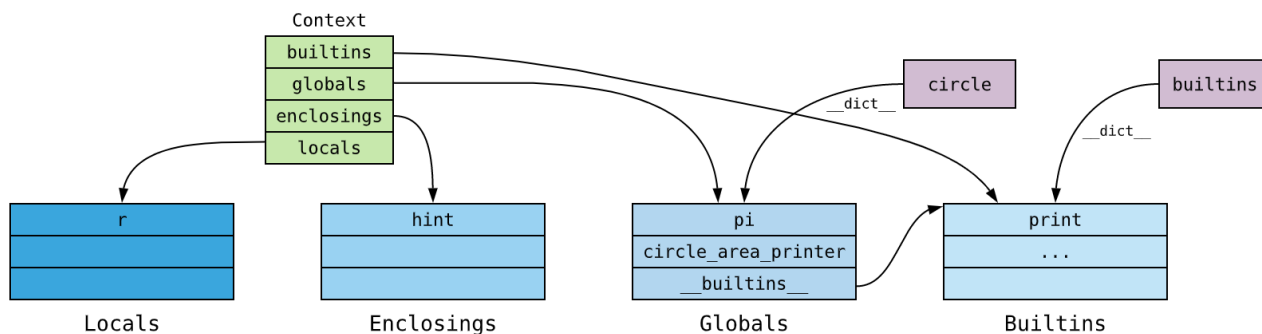


当 *Python* 执行到语句 `print(hint, pi * r ** 2)`，按照 *Local Enclosing Global* 这样的顺序查找语句中涉及的名字。其中，名字 *hint* 在 *Enclosing* 中找到；名字 *pi* 在 *Global* 中找到；名字 *r* 在 *Local* 中找到。

那么，*print* 函数又是如何找到的呢？这就要说到 **内建名字空间**。

## Builtin

*Python* 在 *builtin* 模块中提供了很多内建函数和类型，构成运行时的另一个名字空间 **内建名字空间** (*Builtin*)。像 *print* 这样的内建函数或类型，均需要在这个名字空间中查找：



顺便提一下，全局名字空间中有一个名字指向内建名字空间：

```
>>> import builtins
>>> circle.__builtins__ is builtins.__dict__
True
```

## 属性空间

*Python* 是一个动态语言，在运行时可以很灵活地为一些对象设置新属性。例如：

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.value = 'abc'
>>> a.value
'abc'
```

同样，对象 *a* 的属性在 *Python* 内部也是存储在 *dict* 对象中的，这就是该对象的 **属性空间**：

```
>>> a.__dict__
{'value': 'abc'}
```

修改代表对象属性空间的 *dict* 对象，将影响属性查找结果。由于属性空间中没有名字 *name*，因此属性查找失败：

```
>>> a.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'name'
```

当我们往 *dict* 对象中添加名字 *name* 后，属性查找便成功了：

```
>>> a.__dict__['name'] = 'tom'
>>> a.name
'tom'
```

## 总结

---

在 *Python* 中，一个名字(变量)可见范围由 **作用域** 决定，而程序作用域由语法静态划分，划分规则提炼如下：

- *.py* 文件(模块)最外层为 **全局作用域** ；
- 遇到函数定义，函数体形成子作用域；
- 遇到类定义，类定义体形成子作用域；
- 名字仅在其作用域以内可见；
- 全局作用域对其他所有作用域可见；
- 函数作用域对其直接子作用域可见，并且可以传递( **闭包** )；

与 **作用域** 相对应，*Python* 在运行时借助 *dict* 对象保存作用域中的名字，构成动态的 **名字空间**。这样的名字空间总共有 4 个：

- 内建名字空间
- 全局名字空间
- 闭包名字空间
- 局部名字空间

*Python* 语句在查找名字时，按照 *Local Enclosing Global Builtin* 这样的顺序在 4 个名字空间中查找，这也就是所谓的 **LEGB** 规则。