程序结构

- 作用域、环境和状态
  - 名字空间
- 高阶函数
  - 函数的函数参数
  - **lambda** 表达式
- **random** 随机函数包

作用域 (scope)

- Python 程序以模块为单位，一个 .py 文件是一个模块
  - 程序里，任何有定义的名字 (标识符)，其使用范围都只是程序代码中的某一部分 (一段静态文本) —— 该名字的作用域
- 全局作用域：一个模块构成一个全局作用域
  - 全局定义：在模块表层创建的各种名字 (变量名/函数名/...)，具有全局作用域，在整个模块的范围内都有效
  - 全局变量/函数：具有全局作用域的变量/函数
- 局部作用域：一个函数定义确定一个局部作用域
  - 局部变量/函数：在局部作用域内定义的变量/函数，只在该作用域范围内 (即该函数的体) 有效
  - 函数形式参数：是函数的局部变量，只能在函数体使用
  - (还有其他确定局部作用域的语言结构，之后介绍)

作用域嵌套、同名定义遮蔽 (1)

- 全局作用域和局部作用域形成了作用域的嵌套
- Python 允许在一个函数定义的内部定义其它函数 (即，局部函数定义)，
  ➔ 也形成作用域的嵌套
- 问题：对于程序里每一个名字的每一次使用，如果确定实际对应于哪个作用域里定义的变量/函数？
- 基本规则：
  1. 在不同作用域里定义的名字，即使同名也相互无关
     - 例，两个不同函数可以有同名的参数，相互无关
  2. 当作用域出现嵌套时，内层作用域里定义的名字将遮蔽外围作用域里的同名定义

```
>>> def f(n):
...     print = n
...     print(n)
```

```
>>> f(0)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    f(0)
  File "<pyshell#16>", line 3, in f
    print(n)
TypeError: 'int' object is not callable
```

作用域嵌套、同名定义遮蔽 (2)

- 全局作用域和局部作用域形成了作用域的嵌套
- Python 允许在一个函数定义的内部定义其它函数 (即，局部函数定义)，
  ➔ 也形成作用域的嵌套
- 问题：对于程序里每一个名字的每一次使用，如果确定实际对应于哪个作用域里定义的变量/函数？
- 基本规则：

```
>>> print = 'A wrong way'
>>> print('Hello world!')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print('Hello world!')
TypeError: 'str' object is not callable
```

1. 在不同作用域里定义的名
   - 例，两个不同函数可以
2. 当作用域出现嵌套时，内域里的同名定义
3. 在一个作用域里，一个名字只有最多一个定义
   - 例，不能同时有一个全局变量 x 和一个全局函数 x

在程序里引入新定义 (名字)

- 用 def 定义的函数：属于显式定义
  - def 语句以明确的方式，在其所在的作用域里定义指定的函数名，并建立它和相应函数对象的关联关系
- 函数的形式参数：属于显式描述
  - 形参是一个函数的局部变量，其初值来自实参，可重新赋值
- 用 import 语句导入模块
- 有关变量的规定 (Python 中变量无须说明，属于隐式定义)
  - Python 的基本原则：赋值即定义
    - 在一个 (全局或局部) 作用域里，给一个变量进行赋值，即是在相应作用域里定义了这个变量 (如果没有其他说明)
  - for 的循环变量看作赋值 (按默认规定)，for 语句执行结束后该变量仍存在，其值为循环中的最后取值

变量和作用域：例1

```
x = 2
y = 3
z = 4

def f1 (x):
    y = x**2 + z
    return z**2 - y

print(f1(x + 2 * y + 3 * z))
```

- 说明：
  - 模块层面，定义了全局变量 x, y, z，全局函数 f1
  - 函数 f1：有形参 x，函数体里定义局部变量 y (赋值即定义)，则函数体里全局变量 x，y 被遮蔽
  - f1 函数体里，未对 z 赋值，则 return 语句中使用的是全局变量 z
  - print 函数调用语句，位于模块层面，所用名字均为全局定义

变量和作用域：例2

```
# 特殊情况
x = 1

def f2():
    y = x
    x = 2    # 名字 x 是哪个作用域里的名字?
    return x + y

f2()
```

```
    y = x
UnboundLocalError: local variable 'x'
referenced before assignment
```

- 说明：
  - 在一个作用域里，一个名字至多有一个定义 (基本原则)

    在 f2 的函数体 (局部作用域) 里，x 只能有一个定义
  - 赋值即定义，在这个作用域里 x 被赋值 ➔ x 是其中的局部变量
  - 在执行语句 y = x 时，局部变量 x 还没有定义 (被赋值)，因此取 x 的值是非法的

理解程序行为的重要概念：环境和状态 (1)

- 环境：程序运行时，所有有定义的名字
  - 包括：标准类型名 (float 等)，内置函数名 (print 等)，内置常量名 (True 等)，自定义变量/函数名，…
  - Python 系统启动时，自动建立一个预定义环境
- 语句的执行依赖当前环境：需要用某个变量的值，即到环境里去找；找到则取相应的值使用，否则报告变量无定义错误 (函数情况类似)
- 语句的执行也可能改变环境
  - 对原本无定义的变量名赋值，或 def 语句定义新的函数：将在环境中引入新定义的名字，并建立该名字与其值 (普通数据对象/函数对象) 之间的关联关系，即“名字-对象”约束关系
  - del 语句：从环境中删除有定义的名字

    形式：**del 变量描述, …**

    # 简单情况：变量描述即是名字；其他情况后面介绍，细节可以参考 Python 文档

环境和状态 (2)

- 环境中，每个名字都与一个对象相关联

```
>>> num = 1331
>>> pi = 3.1416
>>> city = "Beijing"
```

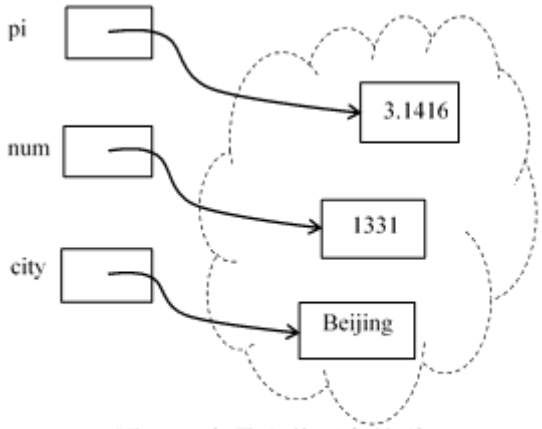


图 4.1. 变量和值对象关联

- **内置函数 id**：对任何对象都有定义，返回其参数对象的标识 (identity，编号，一个整数值)
- 在一个对象的生存期间，其标识唯一且不变

```
>>> id(num)
48339728
>>> id(print)
1343352
>>> id(True)
1603334368
>>> id(id)
1342416
```

环境和状态 (3)

- 状态：在程序运行中的每个时刻，环境中所有有定义的名字及其关联值
  - 状态决定表达式的值、语句的效果
  - 语句的执行可能改变环境的状态，或者改变环境
  - 一个程序，在一次执行中将经历一系列状态
- 执行不同的语句对程序的状态和环境有不同的影响；例

```
x = x + 3          # 一定改变状态，不改变环境
x = y + 3          # 可能改变状态，可能改变环境
print(x, y)        # 改变状态和环境吗？
```

- 环境、状态是理解程序行为的基础
  - 编程时，需要考虑当前环境中哪些名字可以用，它们在当前状态下的关联值是什么

名字空间 (namespace)

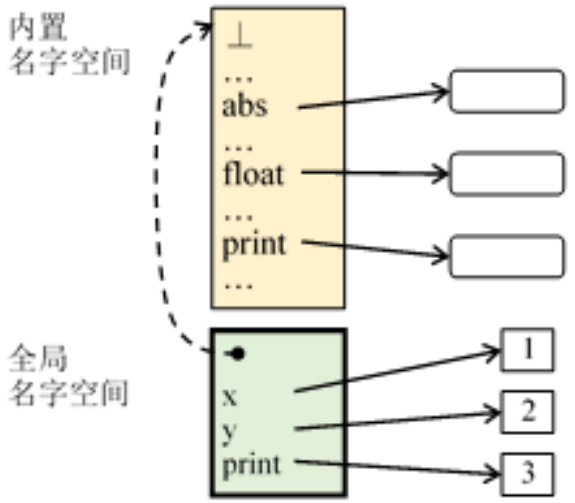
- 名字空间：一组有定义的名字及其关联对象
  - 属于程序动态运行时的概念 (c.f. 作用域对应于一段静态代码)
  - 一组名字空间构成了程序的运行环境 (和状态)
- 启动 Python 解释器后
  - 首先，建立一个内置名字空间 (built-in namespace)，包括所有标准常量名、标准函数名、标准类型名等，以及相应的值
  - 之后，再建立一个“空”的全局名字空间 (global namespace)
- 标准函数 dir() 返回当前名字空间里所有名字 (其结果为字符串作为元素的表)

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

环境的结构

- **Python 解释器启动后**
  - 初始环境的结构：全局名字空间是当前名字空间，其外围名字空间 **(enclosing namespace)** 是内置名字空间
  - 程序的所有操作：默认在当前名字空间里执行，并产生作用
    - 查找名字及其关联值：从当前名字空间开始，逐层往外围进行 —— 名字空间查找规则
    - 引入新名字及其关联值：在当前名字空间里进行
  - 内置名字空间：总是最外围的，最后被考虑和使用

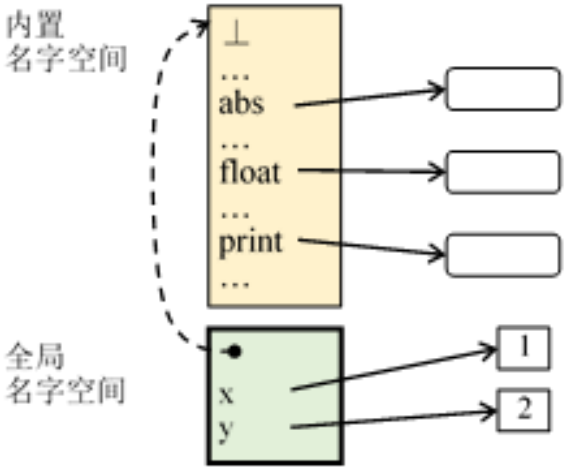
```
>>> x, y = 1, 2
>>> print = 3
>>> print(3)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    print(3)
TypeError: 'int' object is not callable
```



环境的结构

- **Python 解释器启动后**
  - 初始环境的结构：全局名字空间是当前名字空间，其外围名字空间 **(enclosing namespace)** 是内置名字空间
  - 程序的所有操作：默认在当前名字空间里执行，并产生作用
    - 查找名字及其关联值：从当前名字空间开始，逐层往外围进行 —— 名字空间查找规则
    - 引入新名字及其关联值：在当前名字空间里进行
  - 内置名字空间：总是最外围的，最后被考虑和使用

```
>>> del print
>>> print(3)
3
```



执行模块，导入模块

- 在 IDLE 中运行一个模块 (run a module) 时，解释器
  - 新建立一个初始运行环境：当前名字空间为全局名字空间，其中加入所有全局定义的名字，其外围是内置名字空间
  - 顺序处理模块中的语句序列 (环境变化情况类似于交互式执行)
  - 所有语句执行完毕，环境停在当时状态，IDLE 回到交互方式
- 导入模块 (以 math 为例)
  - 执行“from math import *”，把 math 模块里定义的所有功能 (函数名，常量名如 pi 及其关联值) 加入当前名字空间
  - 执行“from math import sin, cos, sqrt”，把名字 sin, cos, sqrt 及其关联值 (函数对象) 加入当前名字空间
  - 执行“import math”将模块名 math 加入当前名字空间并导入模块的全部功能，但将其约束在 math 模块对象里，可以通过 math.xxx “间接”使用 (使用模块里的 xxx 功能)

导入模块

- 导入模块时可以为模块对象指定关联名，之后通过关联名使用
  - **import math as mt**
- 可以一次导入多个模块
  - **import 模块1 as 名字1, 模块2 as 名字2, …**
- 除带星号的形式外，其他形式的 import 语句可以写在函数体里
  - 在函数调用时被执行，相应的名字及其关联值被导入函数执行所建的局部名字空间

```
def f():
    import math
    print(math.e)

f()
print(math.pi)
```

```
    print(math.pi)
NameError: name 'math' is not defined
```

函数的调用与环境 (1)

- 函数调用时，在执行函数体之前，解释器的部分操作：
  - 根据函数名，找到应该执行的函数对象
    - 按照名字空间的查找规则，查找可能经过一系列名字空间，直至内置名字空间
  - 从左到右依次求值实参表达式，得到一组实参对象
  - 为函数建立一个局部名字空间，并根据作用域的嵌套关系设置其外围名字空间
  - 在局部名字空间里，加入
    - 所有形参，并为其建立与实参对象的关联 (默认按位置一一约束)
    - 所有局部定义 (但其关联值暂时无定义)
  - 把新建的局部名字空间设定为当前名字空间
  - 开始执行函数体代码

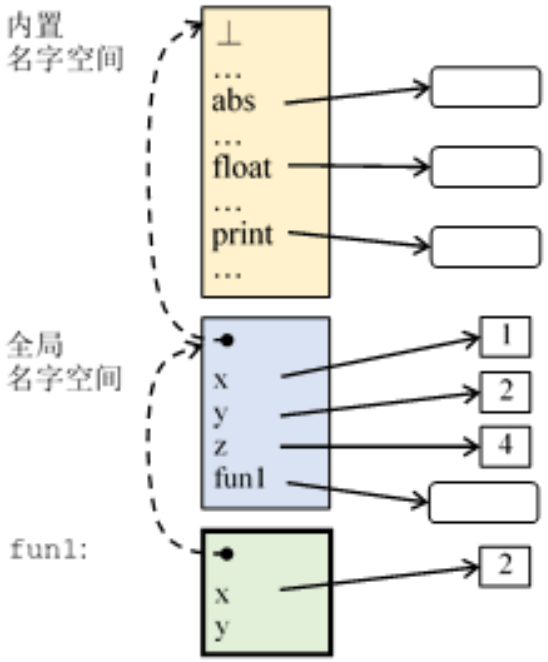
函数的调用与环境 (2)

- 函数执行完成，即函数体代码结束或者遇 return，解释器的部分操作：
  - 计算出需要返回的值 (如果有)
  - 撤销局部名字空间
  - 回到函数调用之前的环境
    - 恢复函数调用所在名字空间为当前名字空间
  - 完成可能出现的函数值的赋值
  - 从函数调用点之后继续执行
- 函数执行结束后，其执行中建立局部约束关系全部失效 (被丢弃)
  - 相关局部环境已撤销，其中的所有约束不再存在
  - 再次调用同一个函数时，将重新建立一个新的局部名字空间
    - 与之前这种函数执行时建立的局部名字空间无关

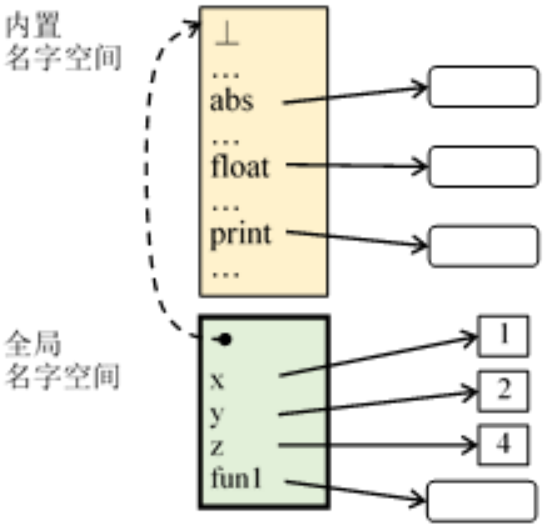
函数的调用与环境 (2)

```
x, y = 1, 2
z = 4

def fun1(x):
    y = x**2 + z
    return z**2 - y - 1

y = fun1(x + 1)
```

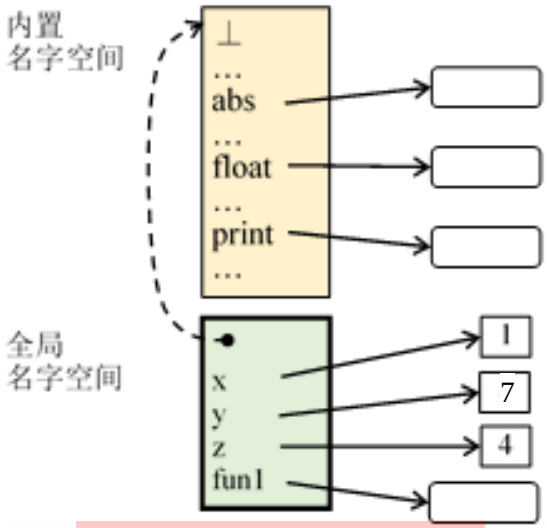
解释器根据静态的作用域嵌套结构来确定程序动态执行时的名字空间的嵌套关系



进入函数 fun1 时的状态



定义 fun1 后的环境和状态

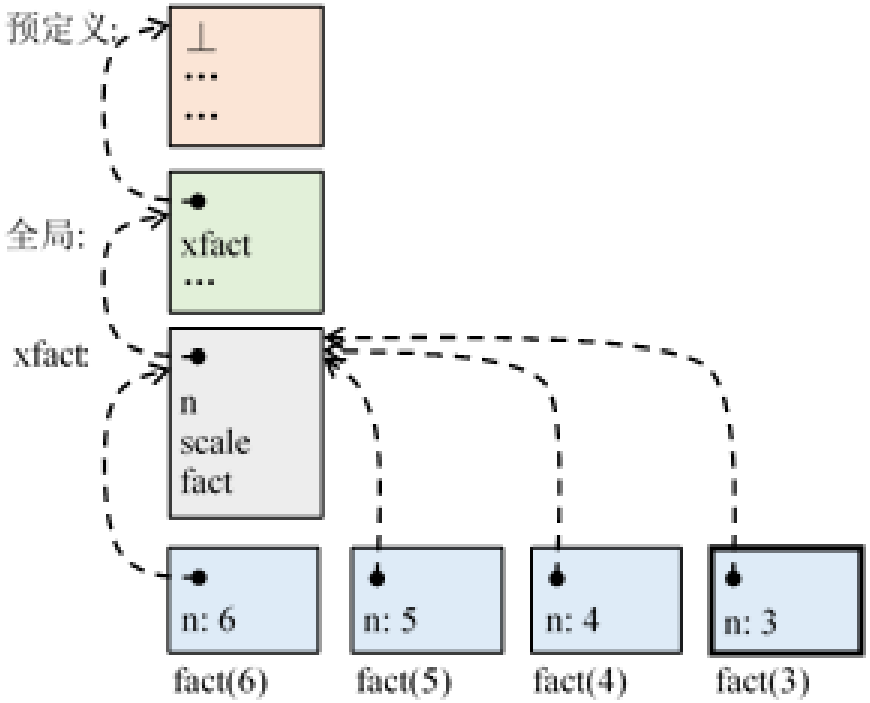


完成函数调用和赋值后的状态

递归函数的调用和环境

- 递归函数的定义，同样构成一个局部作用域
- 递归函数的每个调用，解释器都会建立一个独立的局部名字空间

```
###### 递归函数调用和环境
###### 定义带缩放系数的阶乘函数
def xfact(n, scale):
    def fact(n):
        if n == 0:
            return 1
        else:
            return fact(n - 1) * n

    return fact(n) * scale

print(xfact(6, 2.37))
```



Try：pythontutor.com

- 调用 **xfact**：在全局环境下，建立一个 **xfact** 局部名字空间
- **xfact** 调用 **fact**：建立 **fact** 局部名字空间，其外围是 **xfact** 名字空间
- **fact** 递归调用 **fact**：建立新局部名字空间，外围也是 **xfact** 名字空间
  - 每个递归调用，所建立的局部名字空间相互独立 (不同名字空间里，参数 **n** 的约束对象值不同)
- 每个递归调用退出后，相应名字空间被撤销，执行转到其调用函数

全局和非局部声明 (1)

- 新需求：在函数里，如何修改全局定义 (或在外围函数里定义) 的变量？
  - 例如：如何统计递归定义 **fib** 函数被递归调用执行的次数 ？

```
count = 0

def fib(n):
    count += 1

    if n < 1: return 0
    if n == 1: return 1
    return fib(n - 1) + fib(n - 2)

print('fib(5) =', fib(5))
print('The fib function has been called', count, 'times.')
```

```
    count += 1

UnboundLocalError: local variable
'count' referenced before assignment
```

全局和非局部声明 (2)

- 按照“赋值即定义”原则，函数内部无法修改全局/非局部的变量

  ➔ 需要在函数里使用声明语句

- 全局变量声明语句

  global 变量, …　　*# 写在局部作用域中，可列任意多个变量名*

  语义：声明本函数体里出现的 (这些) 变量是全局变量，到全局作用域里去找其定义 (如果无定义，给其赋值则将建立全局定义)

- 非局部变量声明语句

  nonlocal 变量, …　　*# 写在内嵌的局部作用域中*

  语义：声明在本函数体出现的 (这些) 变量不是局部变量，到本函数外围的非全局作用域里查找定义 (无定义则报错)；如果外围有多层非全局作用域，从内向外逐层检查找最近的定义

- 实例：统计递归定义 **fib** 函数被递归调用执行的次数

小结：Python 名字规则

1. 赋值即定义规则：在任何一个作用域里，给一个变量 (名字) 赋值，则在该作用域里定义具有这个名字的局部变量
2. 全局和非局部声明规则：在一个作用域中，把某个变量声明为全局或非局部变量，对其赋值就是对具有该名字的全局或非局部变量作赋值
   - 通过 global 或 nonlocal 声明，可以建立“赋值即定义规则”的例外情况
3. 定义唯一性规则：在任何一个作用域里，具有某个名字的变量至多有一个；在一个作用域里，同一个名字的所有出现都表示同一个变量
4. 不同定义相互无关规则：在不同作用域里有定义的变量，即使同名也相互无关
5. 作用域遮蔽规则：如果一个作用域嵌套在另一作用域里 (以另一作用域作为外围)，局部变量定义将遮蔽外围同名变量定义

小结：变量 (名字) 查找

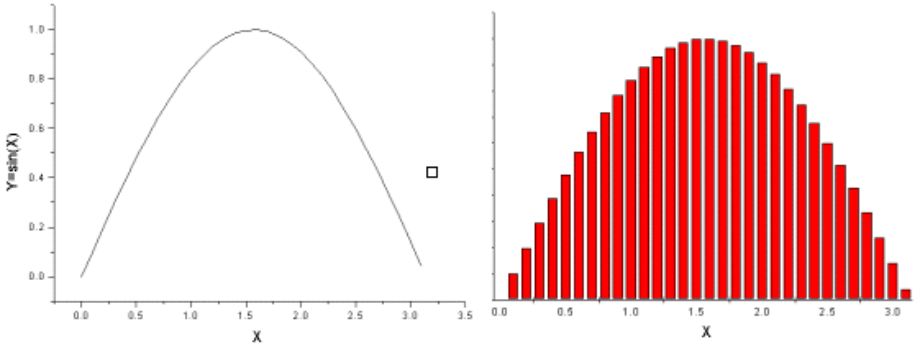
- 假设在函数 f 的体里出现名字 x，要确定其定义，按顺序检查：
  - 如 f 有参数 x 或局部函数定义 x，则 x 的定义确定
  - 如 x 被声明为 global，到全局名字空间去找 x 的定义
  - 如 x 被声明为 nonlocal，到 f 定义所在的作用域 (f 的外围作用域) 找 x 的定义 (可能在更外围，但非全局作用域)
  - 如 x 在 f 的体里被赋值，x 是 f 的局部变量
  - 否则 (x 未被声明也没赋值) x 非局部，到 f 定义所在的作用域找 x 的定义，如没有就到更外围的作用域去找；这种查找一直进行到内置作用域；如果最终找不到 x 的定义，则 x 未定义 (报告错误)
- 注意
  - global 声明：明确说明有关变量的定义在全局名字空间
  - nonlocal 声明：只说明从外围定义域出发去找定义，因此要求变量必须“已有定义”

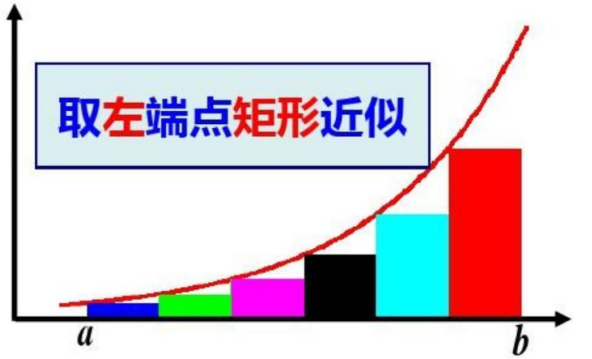
一个例子

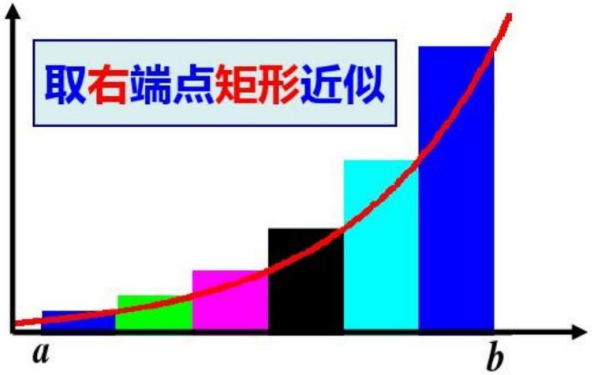
```
##### 说明嵌套作用域的代码示例
###### 注意：实践中，应尽量避免（在嵌套作用域里）重新定义同名变量
x = 3                   # x：全局定义
y = 5                   # y：全局定义

def f1(x):              # x：形参，f1 局部的
    def g(v):           # v：形参，g  局部的
        global y        # 声明 g 内的 y 是全局的
        nonlocal z      # 声明 g 内的 z 是非局部的（即 f1 局部的）
        u = v + x       # u：被赋值，函数 g 局部的；v：g 的形参；x：外围作用域里的定义，f1 的参数
        z = u + y       # 对非局部的 z 赋值；u：g 局部的；y：全局的
        y = z * 4       # 对全局的 y 赋值；z：非局部的

    g(x + 1)            # x：f1 的参数
    z = z + y           # z：被赋值，f1 局部的
    return z + 1        # z：f1 局部的

def f2(x):              # x：形参，f2 局部的
    z = x + y           # z：被赋值，f2 局部的；x：f2 的形参；y：外围作用域里的定义，全局的
    f1(z * x)           # f1：外围作用域的定义，全局函数；z&x：f2 局部的
    return z + y        # z：f2 局部的；y：外围作用域里的定义，全局的

w = f2(1)
print("w =",w)
```

函数的'函数参数'

- Python 支持以函数 (对象) 作为函数的参数 ➔ 可将计算片段参数化
  - 实际的计算过程，由调用时的实际参数提供
- 实例：定义通用的数值积分函数，求一元函数定积分的近似值
  - 用一个'函数参数'表示被积函数，另两个参数表示积分的区间
  - 简单的数值积分算法：矩形法
  - 细节：如何划分积分区间？(见演示)







函数的'函数参数'

- Python 支持以函数 (对象) 作为函数的参数 ➔ 可将计算片段参数化
  - 实际的计算过程，由调用时的实际参数提供
- 实例：定义通用的数值积分函数，求一元函数定积分的近似值
  - 用一个'函数参数'表示被积函数，另两个参数表示积分的区间
  - 简单的数值积分算法：矩形法
- 注意：
  - Python中函数的参数不需要说明类型， '函数参数' 只是约定
  - 在函数体里，'函数参数' 只能作为函数对象使用 (比如被调用)
  - 提供给 '函数参数' 的实参函数对象，应该满足各方面的要求，包括调用的形式、所需参数数量、以及各参数的类型等
    - 完全靠写程序的人保证！！

匿名函数、lambda 表达式

- **lambda 表达式：Python 里的一种表达式 (c.f. def 语句)**
  - 用于描述小的匿名函数 (即是“函数字面量”)
  - 可以作为函数使用，接受合适的实参则能完成相应的计算
- 语法：**lambda** 参数, ... : 表达式

  参数列表 （可以为空）

  lambda 表达式的体
  只能是“纯”表达式
  其中不能有赋值、循环等语句
- 语义：求值时，建立一个以 参数, ... 为形参的匿名函数对象

```
>>> lambda x, y: x**2 + y**2
<function <lambda> at 0x0000026D3122B7E0>
```

- 语用一：直接作为函数对象使用
  - 建立所列参数与实参之间的约束，并以表达式的值作为结果

```
>>> (lambda x, y: x**2 + y**2)(3, 4)
25
```

lambda 表达式

- 语用二：作为函数对象，赋给变量、传递给需要函数参数的函数等

```
>>> f = lambda x, y: x**2 + y**2
>>> f(5, -9)
106
```

```
>>> lst = [2, 4, 6, 8, 10]
>>> list(map(lambda x: int(str(x)*2), lst))
[22, 44, 66, 88, 1010]
```

- **lambda** 表达式确定一个局部作用域
  - 其参数只在 **lambda** 表达式的体的内部有效
  - **lambda** 表达式写在某个作用域内，将形成作用域的嵌套
    - **lambda** 表达式的体里可以使用外围作用域的名字

随机数包 random

- random 包提供 (伪) 随机数生成函数，详见标准库手册
- 常用随机数相关函数
  - random()：返回 [0.0, 1.0) 中的一个随机浮点数
  - randrange(n)，randrange(m, n)，randrange(m, n, d)：返回给定区间里的一个随机整数 (参考 range 的情况)
  - randint(m, n) ：相当于 randrange(m, n+1)
  - choice(s) ：从字符串/序列对象 s 中随机选出一个字符/元素
  - seed(n), seed()：用整数 n 或者系统 (当前时间) 重置随机数生成器；调用 seed 之后会重启一个随机序列
  - uniform，triangular，gauss，gammavariate，betavariate，expovariate 等函数可生成服从各种分布的随机浮点数
  - 请自行查看文档，了解其它功能的函数

随机数包 random

- random 包提供 (伪) 随机数生成函数，详见标准库手册
- 常用随机数相关函数
  - random()：返回 [0.0, 1.0) 中的一个随机浮点数
  - randrange(n)，randrange(m, n)，randrange(m, n, d)：返回给定区间里的一个随机整数 (参考 range 的情况)
  - randint(m, n) ：相当于 randrange(m, n+1)
  - choice(s) ：从字符串/序列对象 s 中随机选出一个字符/元素
  - seed(n), seed()：用整数 n 或者系统 (当前时间) 重置随机数生成器。调用 seed 之后会重启一个随机序列

```
>>> import random
>>> random.choice('0123456789')
'3'
>>> colors = ['red', 'blue', 'green', 'black', 'pink']
>>> random.choice(colors)
'green'
```

例：简单随机模拟框架

- 已知两个正整数互素的概率为 $\frac{6}{\pi^2}$，要求写程序验证这一概率的正确性
  - 思路：考虑做一组随机模拟试验，统计其中通过试验 (即互素) 和未通过试验这两方面的情况 (蒙特卡罗模拟)
- 通用的蒙特卡罗模拟框架：假设一共做 n 次试验

```
passed = 0
for i in range(n) :
    做一次随机试验
    如果通过， passed +=1
return passed / n
```

- 函数 montecarlo(test, num)：实现蒙特卡罗模拟框架
  - 函数参数 test： 传递具体的随机试验 (并不局限于互素试验)，其实参应是无参函数，返回真假值以表示一次试验通过与否
  - 参数 num：随机试验的次数