第3章 函数

李宇

东北大学-计算机学院-智慧系统实验室

liyu@cse.neu.edu.cn

函数例子 – 投影仪

- 投影仪是一个黑盒
- 我并不知道它如何工作
- 只知道它需要的接口: input/output
- 连接任意一台能够与投影仪通信的设备
- 这个黑盒会把图像从设备源投影到墙上
- **用到了抽象(ABSTRACTION) 的概念**: 使用时无需知道投影仪如何工作

函数例子 – 投影仪

- 将奥运会的超大图像投影分解为单独的投影仪的单独任务
- 每台投影机接受输入并产生单独的输出
- 所有投影机一起工作以产生更大的图像
- **分解(DECOMPOSITION) 的概念**: 不同的设备一起工作以实现最终目标

用分解的概念来构建

- 在投影仪的例子中，把任务拆分到多个设备上
- 在编程中，把代码分解成**模块(modules)**
  - 独立
  - 用来拆分代码
  - 可重用
  - 让代码更有组织性
- 这一讲中，用**函数(functions)**来分解
- 未来的课里，用**类(classes)**来分解

用抽象来隐藏细节

- 在投影机示例中，知道如何使用它的说明就足够了，无需知道如何构建它。
- 在编程中，将一段代码视为一个黑盒
  - 看不到细节
  - 无需看到细节
  - 不想看到细节
  - 隐藏繁杂的代码细节
- 通过函数的**说明(function specifications)**或者**文档字符串(docstrings)**来达到抽象

函数

- 函数(functions): 一段可被复用的代码
- 函数在被**调用**(“**called**” or “**invoked**”)前不会被运行
- 函数的构成:
  - 有一个**函数名(name)**
  - 有0个或多个**参数(parameters)**
  - 有一个**文档字符串(docstring)** (推荐但不必须)
  - 有一个**函数体(body)**
  - **返回(returns)** something or None

如何写/调用函数

关键字

函数名

参数 parameter

函数的文档字符串 或者使用说明

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

函数体 body

```
is_even(3)
```

使用函数名和参数来调用函数

函数体

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

执行一些语句

关键字

将被被返回的表达式

变量作用域 (scope)

- **实参(actual parameter)**的值 在函数被调用时与**形参(formal parameter)** 绑定
- 进入新的函数时创建一个作用域**(scope/frame/environment)**
- **作用域(scope)** 是名称和对象的映射（类似字典）

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

x是形参

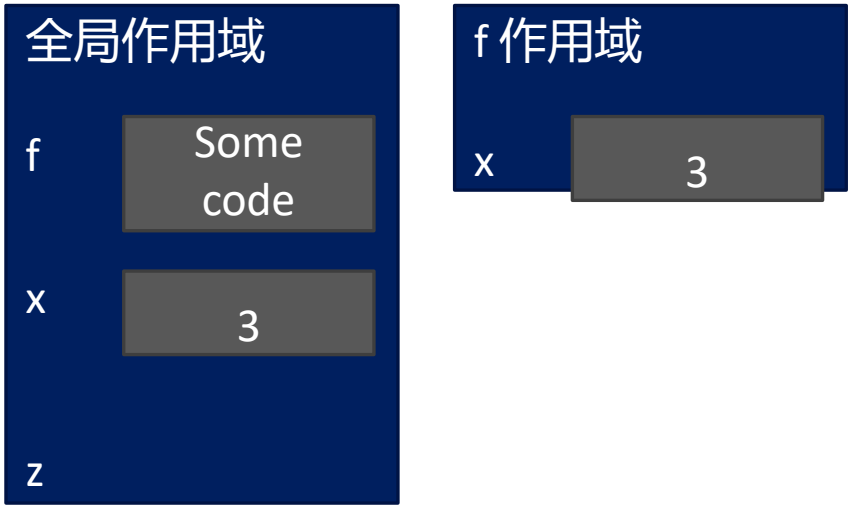
函数定义

实参

主程序代码
*初始化一个变量x
*调用函数f(x)
*把函数调用的返回值赋值给变量z

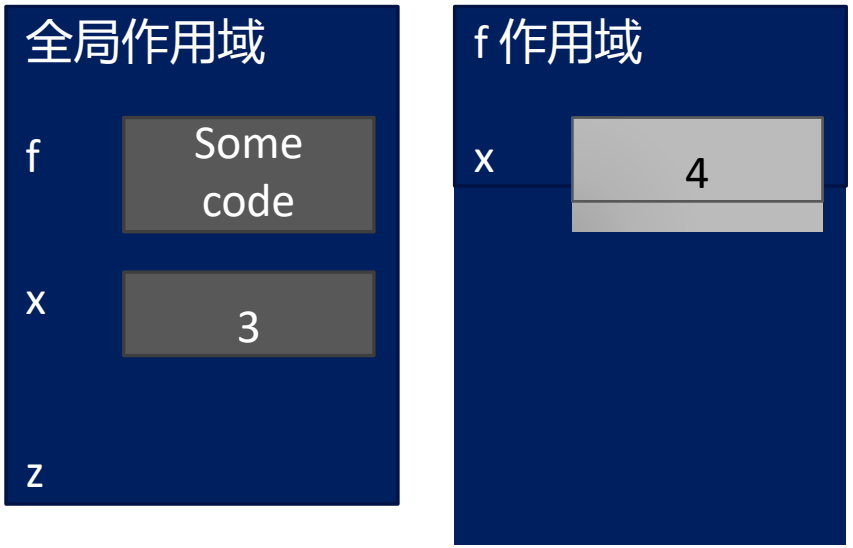
变量作用域 (scope)


```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



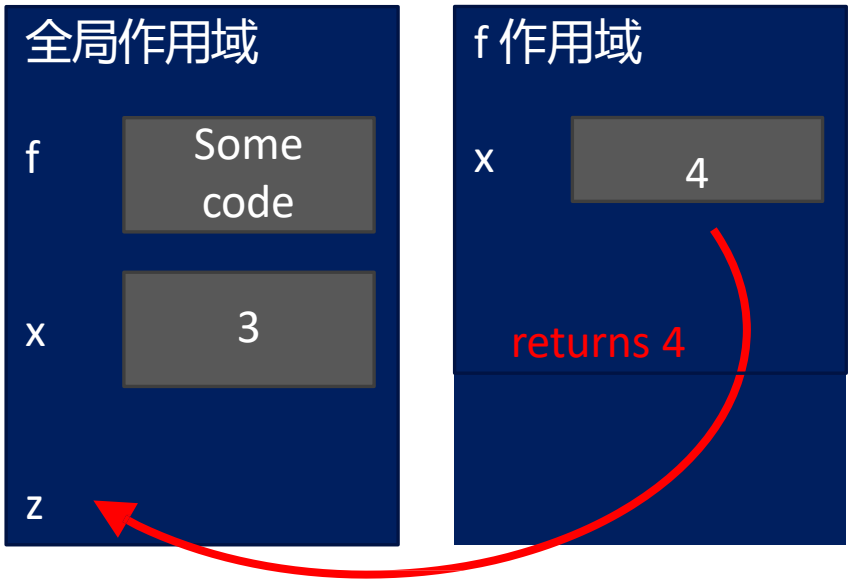
变量作用域 (scope)

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



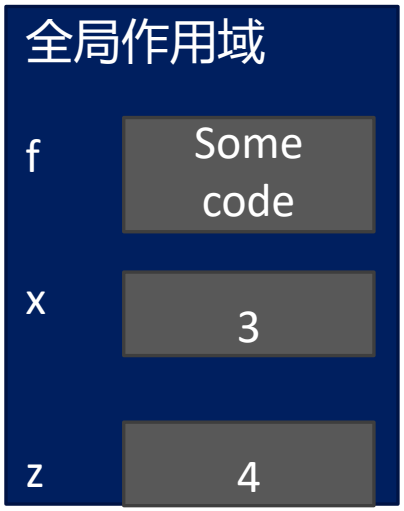
变量作用域 (scope)

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



变量作用域 (scope)


```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



如果函数没有return

```
def is_even( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    i%2 == 0
```

没有return语句

- **如果没有return,** Python 返回 **None**
- None 表示没有值

把函数做为参数

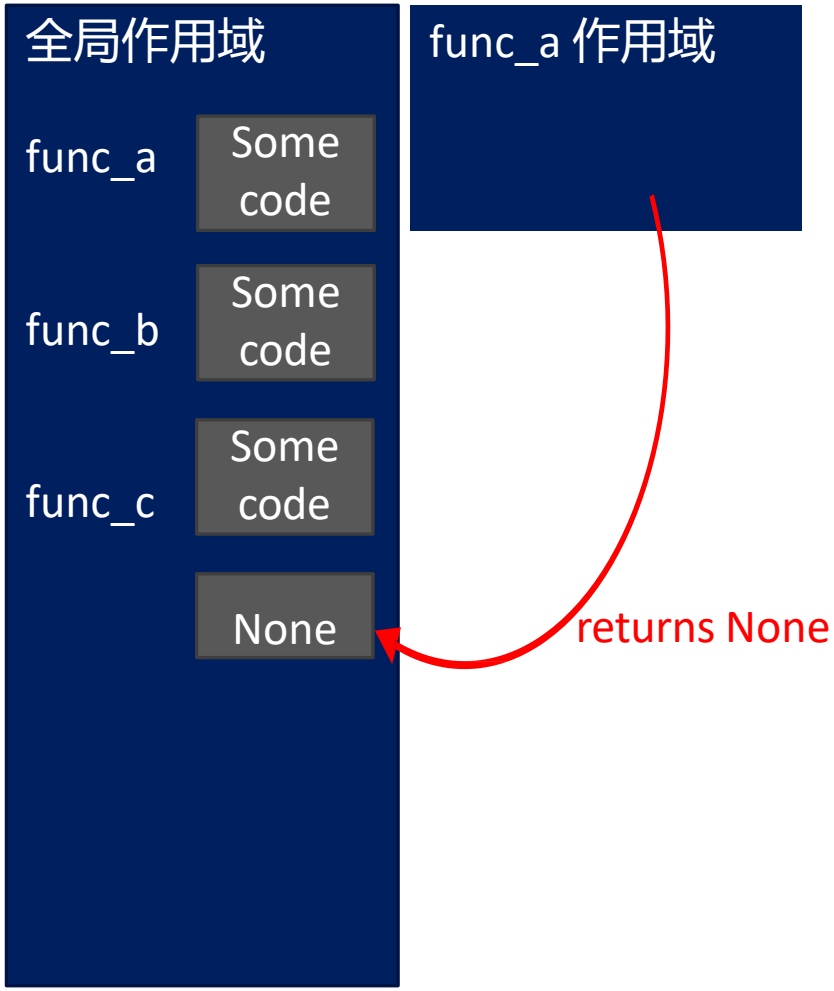
实参可以传入任意类型，包括函数

```
def func_a():
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```

调用func_a, 没有参数
调用func _b, 有一个参数
调用func_c，有一个函数类型的参数

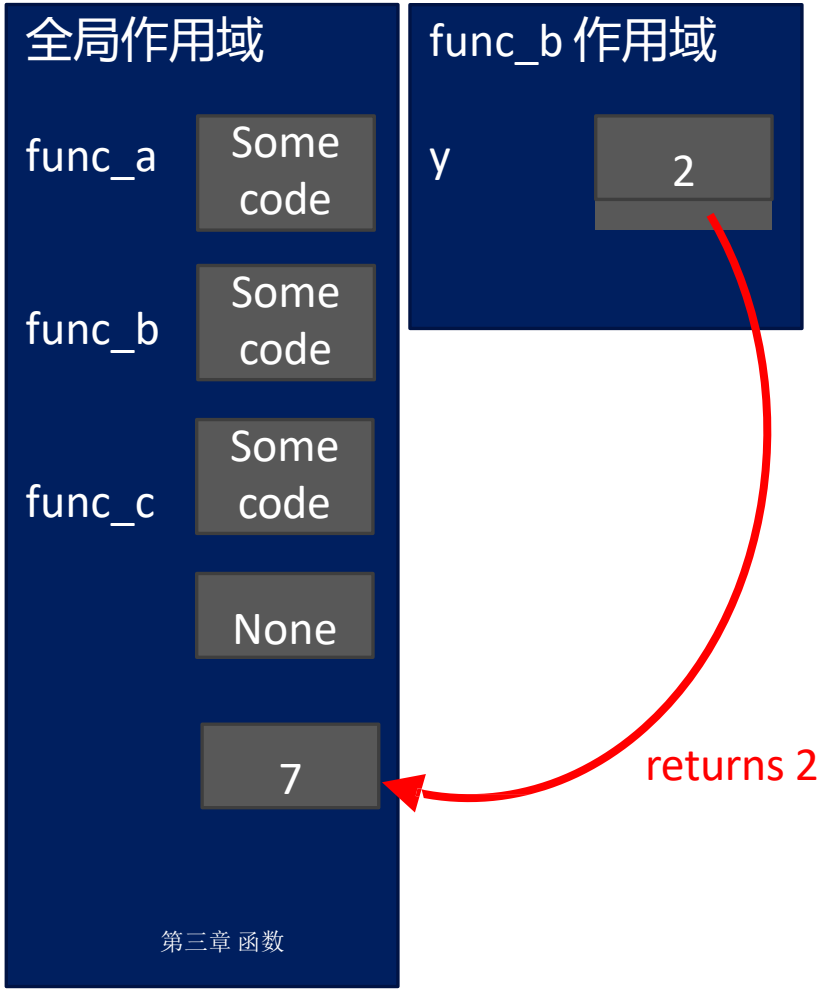
把函数做为参数

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```



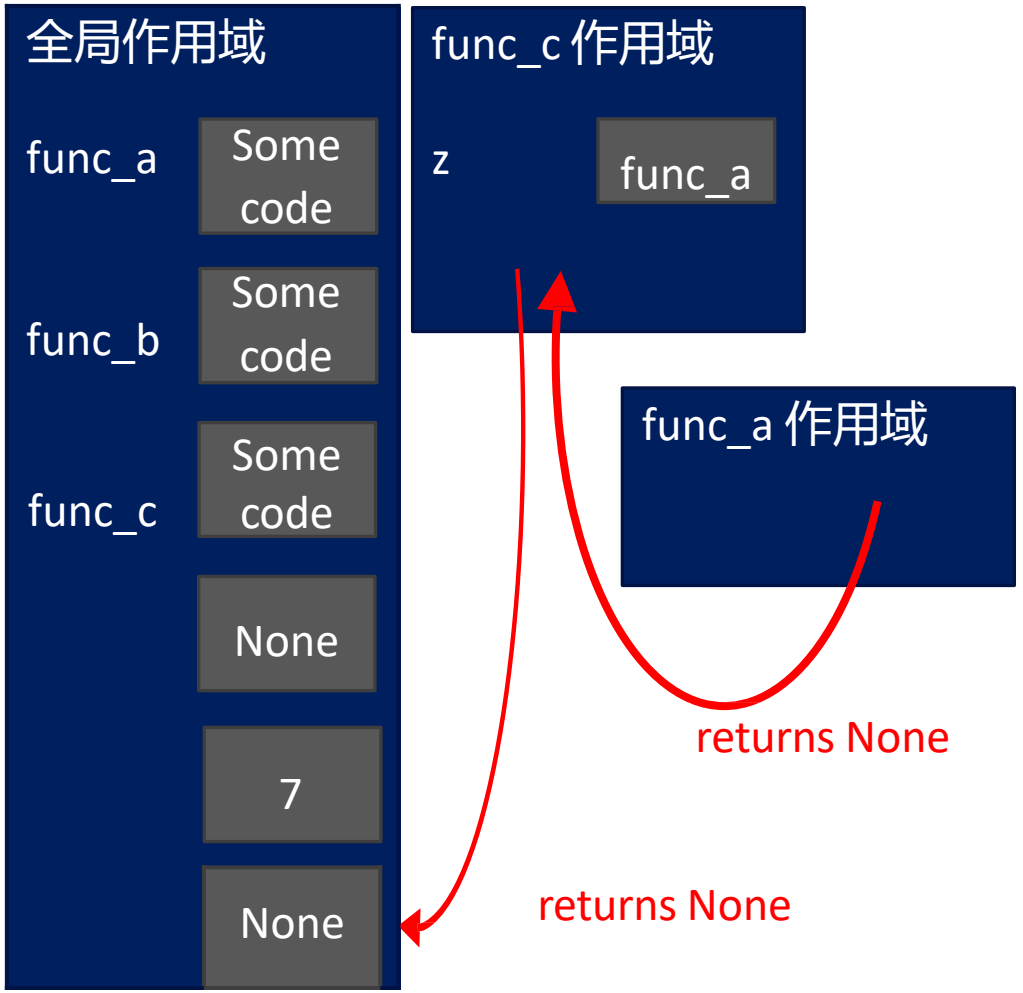
把函数做为参数

```
def func_a():
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```



把函数做为参数

```
def func_a():
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```



作用域的例子

- 在函数内，**可以获取**外部定义的变量
- 在函数内，**不可以修改**外部定义的变量– **可以使用全局变量(global variables)**, 但不推荐

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

x在f的作用域内被重定义

局部和全局作用域中的x是不同的两个对象

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x从函数g外部获取

在函数g中的x从全局作用域中获取到值=5

```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

unboundLocalError: 从全局作用域中获取的x不能在函数体内修改

作用域的例子

- 在函数内，**可以获取**外部定义的变量
- 在函数内，**不可以修改**外部定义的变量– **可以使用全局变量(global variables)**，但不推荐

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

```
def g(y):
    print(x)

x = 5
g(x)
print(x)
```

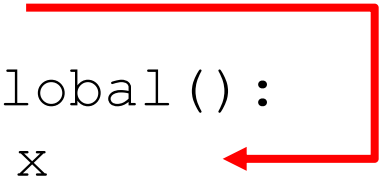
```
def h(y):
    x += 1

x = 5
h(x)
print(x)
```

打印出的x是从
全局作用域中获取的

全局变量

- global 关键字

```
x = 1
def change_global():
    global x
    x = x + 1

change_global()
x
#2
```



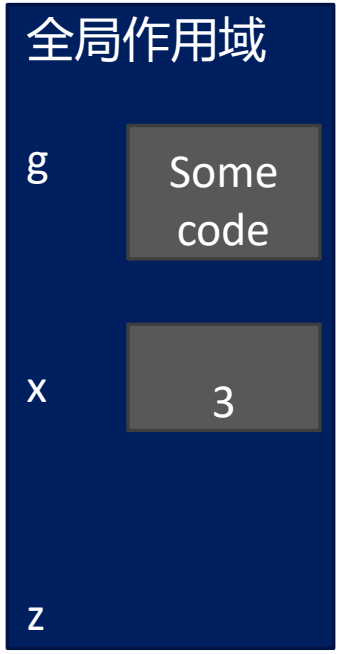
去掉global x后，报错

- 函数内声明变量是全局变量，才能在调用函数后使全局变量得到修改

作用域进阶

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```

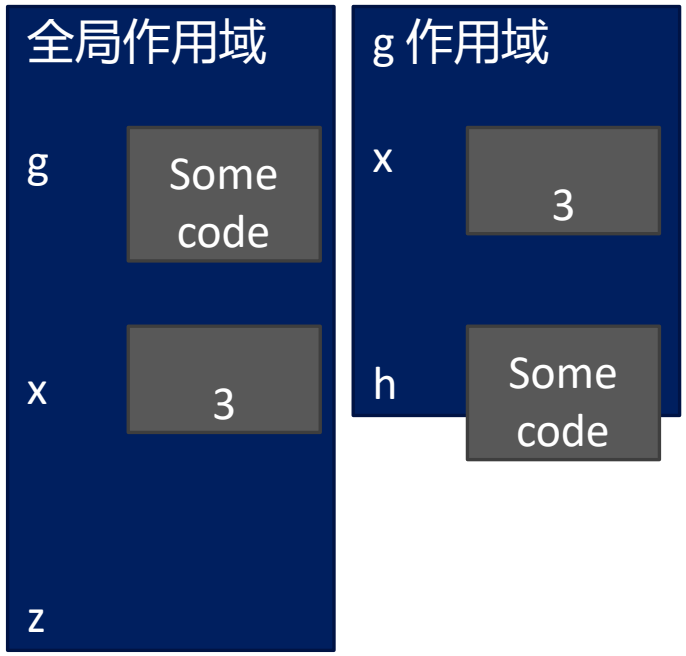
Some code



作用域进阶

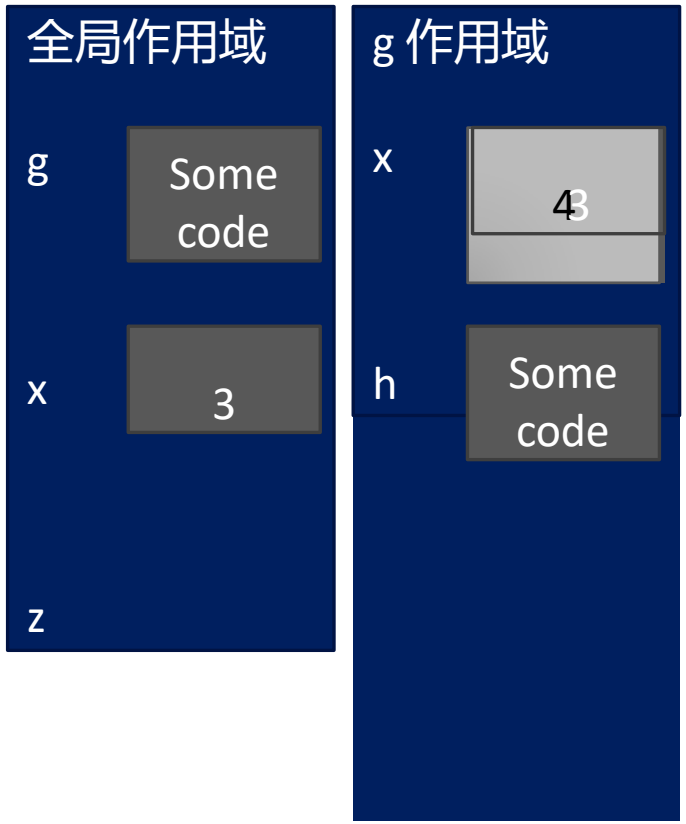
进入到函数g内时

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



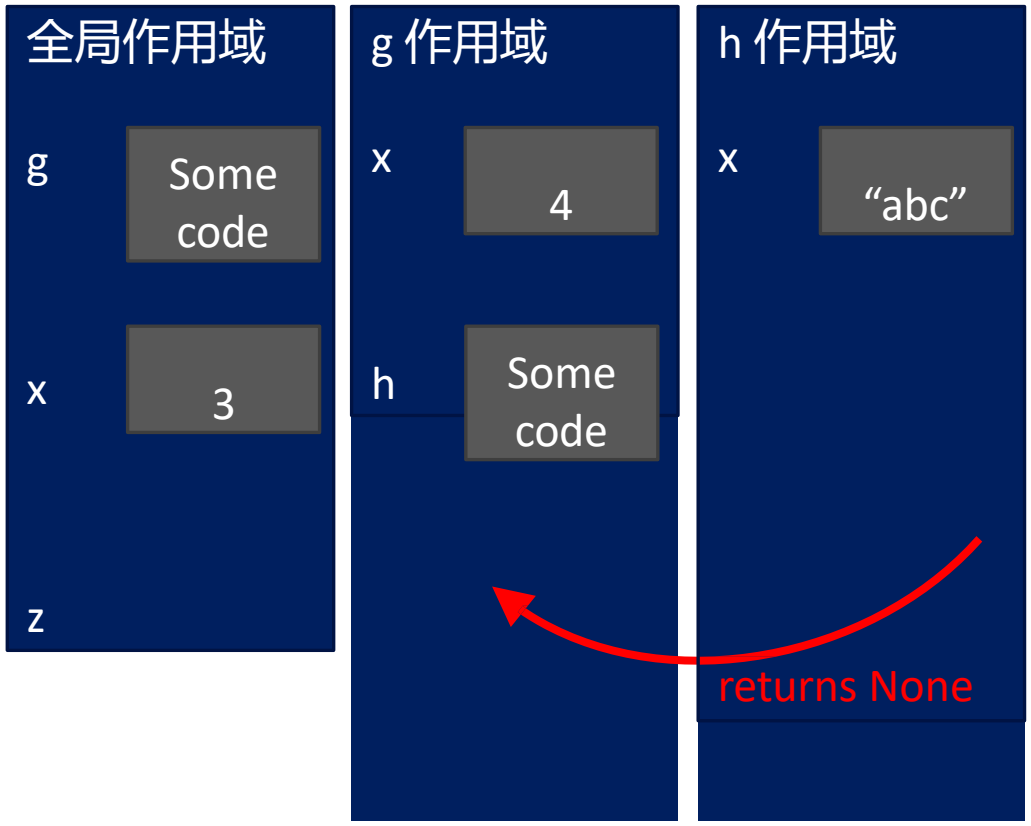
作用域进阶

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



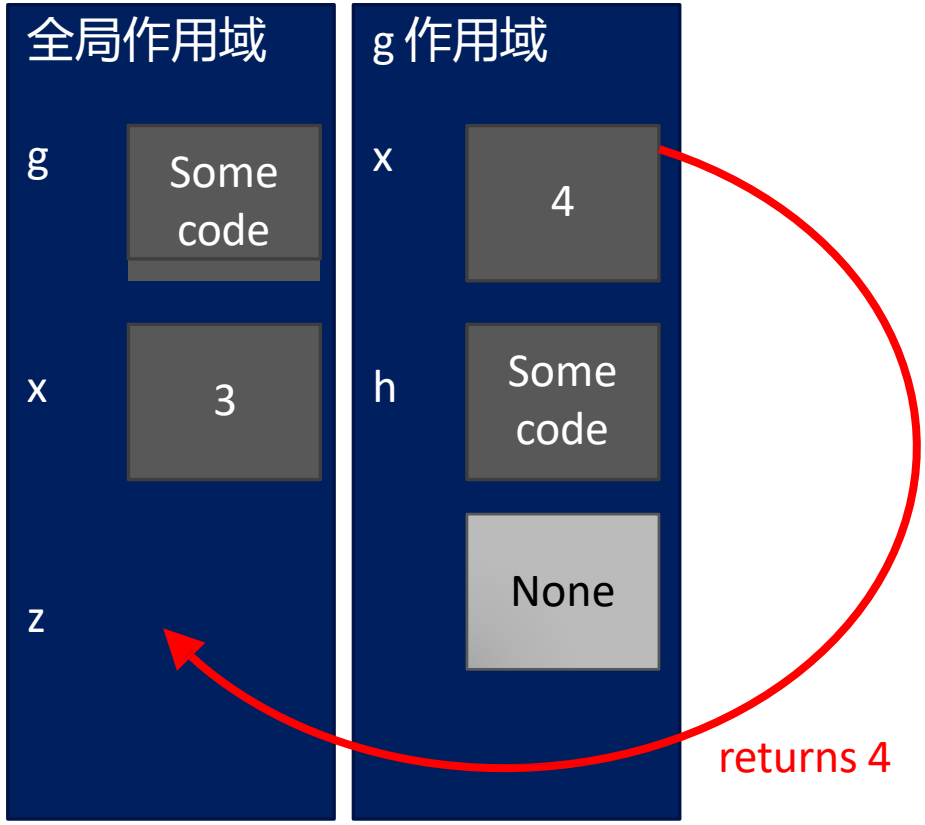
作用域进阶

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



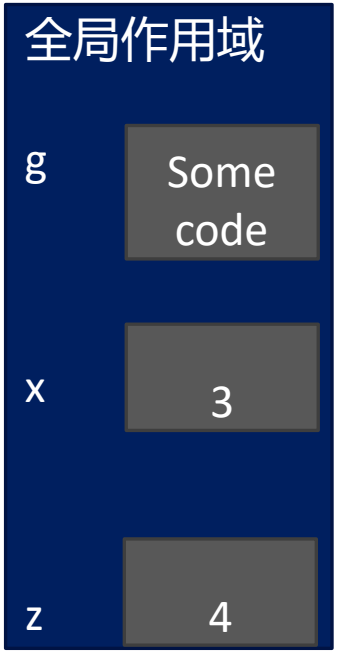
作用域进阶

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



作用域进阶

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



程序结束时

给函数编写文档

- 用注释：#
- 独立文档字符串(docstring)
  - def语句后面的字符串

```
def square(x):
    'Calculates the square of the number x.'
    return x * x
```

  - 访问文档字符串时
  - __doc__是一个魔法属性

```
>>> square.__doc__
'Calculates the square of the number x.'
```

函数参数

- 之前：按照参数位置传入参数（顺序很重要）
- 按照关键字

```
def hello_1(greeting, name):
      print('{}, {}!'.format(greeting, name))
>>> hello_1(greeting='Hello', name='world')
#Hello, world!
#按照关键字传参时，参数的顺序无关紧要。
>>> hello_1(name='world', greeting='Hello')
#Hello, world!
```

关键字参数

- 关键字参数可以在函数定义时赋默认值

```
def hello_3(greeting='Hello', name='world'):
      print('{}, {}!'.format(greeting, name))
# 调用函数时可以不传入任何参数
>>> hello_3()
#Hello, world!
# 可以只传入部分参数（此时按照传入顺序传参）
>>> hello_3('Greetings')
#Greetings, world!
# 也可全部传入
>>> hello_3('Greetings', 'universe')
#Greetings, universe!
```

收集参数

- 使用 * 号收集参数，调用时可以传入多个参数

```
def print_params(*params):
      print(params)
>>> print_params(1, 2, 3)
#(1, 2, 3)
```

- 收集余下的位置参数

```
def print_params_2(title, *params):
      print(title)
      print(params)
#调用时传入多个参数
>>> print_params_2('Params:', 1, 2, 3)
#Params:
#(1, 2, 3)
```

收集的参数存在元组里

收集参数

- *号不能收集关键字参数，**号能

```
def print_params_3(**params):
      print(params)

>>> print_params_3(x=1, y=2, z=3)
#{'z': 3, 'x': 1, 'y': 2}
```

- *号和**号还能在调用函数时分配参数

```
def add(x, y):
      return x + y

params = (1, 2)
>>> add(*params)
#3
```

元组中有要相加的数

用*号把元组的1和2分配给参数x和y，可分配可迭代对象

混合使用


```
def save_ranking(*args, **kwargs):
    print(args)
    print(kwargs)
save_ranking('ming', 'alice', 'tom', four='wilson',five='roy')
# ('ming', 'alice', 'tom')
# {'four': 'wilson', 'five': 'roy'}

def save_ranking(**kwargs, *args):
    ...
```

关键字参数不能在位置参数前声明

递归

- 递归：函数调用自身

```
def recursion():
    return recursion()

recursion()
```

无穷递归，理论上永不结束，会引发异常报错

- 合理的递归：把问题按照递归思想拆分
  - 基线条件：满足条件时函数直接返回一个值
  - 递归条件：包含一个或多个调用，解决问题的一部分

递归计算阶乘

- 使用循环

```
def factorial(n):
    result = n
    for i in range(1, n):
        result *= i
    return result
```

- 使用递归

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

1的阶乘=1
(基线条件，满足时返回一个值)

n>1时，阶乘为n-1的阶乘再乘 n
(递归条件，解决问题的一部分)

Python内置函数

红色：已讲
绿色：一看就懂

内置函数				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

内置函数

- callable(object)
  - 检查一个对象是否可调用
  - 函数、方法、lambda 表达式、 类以及实现了 **__call__** 方法的类实例, 都返回 true

```
class C:
    def printf(self):
        print('This is class C!')
objC = C()
callable(C)             #True 调用类会产生对应的类实例
callable(C.printf)      #True
callable(objC)          #False 类C没实现__call__()方法
callable(objC.printf)   #True

    def __call__(self):
#添加方法       print('Put this method to class!')
```

内置函数

- range函数
  - range(stop)
  - range(start, stop[, step]) #step为可选参数
- sorted函数
  - >>>help(sorted) #查看sorted的文档
  - #sorted(iterable, /, *, key=None, reverse=False)
    - / 代表位置参数的结束
    - * 强制之前的参数只通过位置指定（也就是iterable）
    - * 之后的参数全为关键字参数

```
def myfun(a, *, b, c):
    print(a, b, c)
myfun(1,b=2,c=3)#1, 2, 3
myfun(1,2,3)  #报错takes 1 positional argument but 3 were given
```

内置函数

- zip()
  - help(zip)
    - zip(iter1 [,iter2 [...]]) --> zip object
  - 方括号内的参数皆为可选参数，可省略。
  - 可以缝合1/多个可迭代对象
- bool()
  - bool(x) -> bool
  - 何时返回False?
    - bool()，无参数时
    - x为0、‘’、False、None、空元组/列表/字典

内置函数

- all(iterable, /)
  - Return True if bool(x) is True for all values x in the iterable.
- any(iterable, /)
  - Return True if bool(x) is True for any x in the iterable.
- bin(number, /)
  - Return the binary representation of an integer.

```
>>> bin(2796202)
# '0b1010101010101010101010'

>>> bin("三")
# Error

>>> bin(0x11)
# '0b10001'

>>> bin(2.0)
# TypeError: 'float' object cannot be interpreted as an integer

>>> bin(2+3j)
# TypeError: 'float' object cannot be interpreted as an integer
```

内置函数

- globals()
  - 返回全局作用域内 变量名/值 对的**字典 (本体、非拷贝)**
    - globals **可以修改**

```
z = 7 #定义全局变量
def foo(arg):
        return arg
print( globals() )
```

#{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, **'z': 7, 'foo': <function foo at 0x110633670>**}

```
globals()["z"] = 8 #修改变量z的值
print( "z=",z )
# z= 8
```

内置函数

- locals()
  - 返回字典，包含**当前位置**作用域中的变量名：值

```
z = 7 #定义全局变量
def foo(arg):
    z = 6
    print(locals())
foo(3)
#{'arg': 3, 'z': 6}
print(locals())
```

#{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, **'z': 7, 'foo': <function foo at 0x110633670>**}

  - 在全局处调用locals()，作用等同于globals()

返回作用域的函数

- locals()
  - 返回局部作用域内 变量名/值 对的**字典 的拷贝**

```
def foo(arg,a):
    x =1
    y = 'xxxxxx'
    for i in range(10):
        j = 1
        k = i
    print(locals())
    locals()['x'] = 2 #修改局部作用域
    print(locals())
foo(1,2)

#两次打印无变化
#{'arg': 1, 'a': 2, 'x': 1, 'y': 'xxxxxx', 'i': 9, 'j': 1, 'k': 9}
```

内置函数

- vars([object]) -> dictionary
  - 返回对象object的属性：属性值的字典对象，如果没有参数，就打印当前调用位置的属性和属性值 类似 locals()

```
#没有传参数，作用相当于locals()
vars()
```

{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, **'z': 7, 'foo': <function foo at 0x110633670>**}

```
class Foo:
    a=1
    def f(x):
        return x
foo=Foo()
print(vars(Foo))
```

{'__module__': '__main__', **'a': 1, 'f': <function Foo.f at 0x104c731f0>**, '__dict__': <attribute '__dict__' of 'Foo' objects>, '__weakref__': <attribute '__weakref__' of 'Foo' objects>, '__doc__': None}

内置函数

- vars([object]) -> dictionary
  - 传入对象，返回对象的 __dict__ 属性：
    - __dict__ 属性是包含对象的可变属性的字典。

```
class Person:
  name = "Bill"
  age = 19
  country = "USA"

vars(Person)
#{'__module__': '__main__',
              'name': 'Bill',
              'age': 19,
              'country': 'USA',
              '__dict__': <attribute '__dict__' of 'Person' objects>,
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              '__doc__': None}
#效果等同于
#print(Person.__dict__)
```

内置函数

- dir([object]) -> list of strings
  - 函数不带参数时，返回当前范围内的变量、方法和定义的类型列表

```
>>>dir() # 获得当前模块(此处为全局)的属性列表
#['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

  - 带参数时，返回参数的属性、方法列表

```
>>> dir([ ])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append'
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

  - dir()和vars()的区别：dir()只返回属性，vars()则返回属性与属性的值。传入vars()的参数需要有__dict__属性

内置函数

- globals()、locals()、vars()、dir()的不完全总结
  - 在全局处调用globals() 、locals()、vars()效果一样，返回全局作用域的字典
  - 在局部处(如函数内)调用locals()、vars()效果相同，vars()内需无参数、返回值修改无效
  - vars(object)有参数时，返回object的 __dict__ 属性
    - __dict__ 属性是包含对象的可变属性的字典。
  - dir(object)
    - 只返回object的属性、方法的列表，不包含值

内置函数

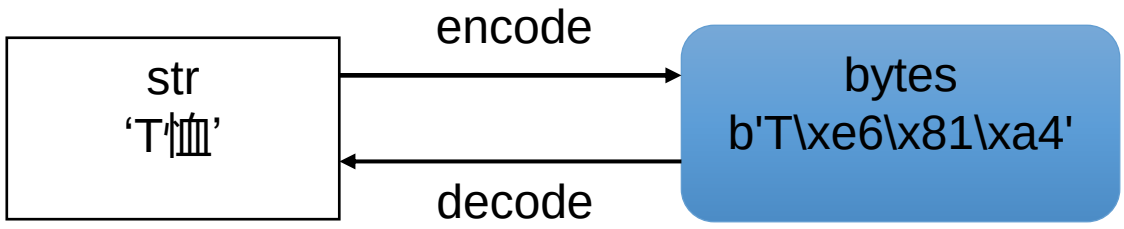
- ord(), chr()均为python内置函数
- ord() 以**一个字符**（长度为1的字符串）作为参数，**返回**对应的 **ASCII 数值**（0-255），或者 Unicode 数值(其他字符)

```
>>> ord('d')
#100
>>> ord('5')
#53
```

- 相反地，chr()函数是输入一个整数, 返回其对应的ascii或unicode符号

```
>>> chr(100)
#'d'
>>> chr(53)
#'5'
```

str和bytes



- str和bytes的相互转换
- **str.encode(‘encoding’) -> bytes**
- **bytes.decode(‘encoding’) -> str**
  - encoding 指的是具体的编码规则的名称，对于中文来说，它可以是这些值： ‘utf-8’(默认), ‘gb2312’, ‘gbk’, ‘big5’ 等等

```
"AAAbbb".encode()
# b'AAAbbb’

b'AAAbbb'.decode()
#'AAAbbb’
```

```
"T恤".encode()
# b'T\xe6\x81\xa4’

b = B'T\xe6\x81\xa4'
b.decode()
#‘T恤‘，B/b开头都行
```

```
b = "T恤".encode('gbk’)
# b'T\xd0\xf4’

b.decode('gbk’)
#'T恤'
```

内置函数

- str()、bytes()、repr()
  - str(object='') -> str
    - 无参数时，创建空字符串
    - str(-2.3) # '-2.3'

```
x = [-23.1,'Python',18]
str(x)
#“[-23.1, ‘Python’, 18]”
#可转换元组、列表、字典、集合等为字符串
```

```
x = b'T\xe6\x81\xa4'
str(x,'gbk',’ignore’)

#'T鎮'
```

  - str(bytes_or_buffer[, encoding[, errors]]) -> str

```
x = b'sss'
type(x) #bytes
str(b'sss’)

#"b'sss'"
```

```
x = b'T\xe6\x81\xa4'
str(x)

#"b'T\\xe6\\x81\\xa4'"
```

```
x = b'T\xe6\x81\xa4'
str(x,’UTF-8’)

#'T恤'
```

```
x = b'T\xe6\x81\xa4'
str(x,’gbk’)

#UnicodeDecodeError
```

内置函数

- errors处理有6种
- **strict (default):** 抛出UnicodeDecodeError.
- **ignore:** 它忽略不可编码的 Unicode
- **replace:** 它用问号替换不可编码的 Unicode
- **xmlcharrefreplace:** 它插入 XML 字符引用而不是不可编码的 Unicode
- **backslashreplace:** 插入 \uNNNN Espace 序列而不是不可编码的 Unicode
- **namereplace:** 插入 \N{...} 转义序列而不是不可编码的 Unicode

内置函数

- bytes函数
  - bytes() -> empty bytes object
    - 定义空的字节序列
  - bytes(int) -> bytes object of size given by the parameter initialized with null bytes
    - 定义指定个数的字节序列bytes，默认以0填充，不能是浮点数
  - bytes(iterable_of_ints) -> bytes
    - 定义指定内容的字节序列bytes
  - bytes(string, encoding[, errors]) -> bytes
    - 定义字节序列bytes，如果包含中文的时候必须设置编码格式
  - bytes(bytes_or_buffer) -> immutable copy of bytes_or_buffer

内置函数

- bytes函数
  - bytes() -> empty bytes object
    - 定义空的字节序列

```
a = bytes()
print(a)
print(type(a))

#b''
<class 'bytes'>
```

内置函数

- bytes函数
  - bytes(int) -> bytes object of size given by the parameter initialized with null bytes
    - 定义指定个数的字节序列bytes，默认以bytes null填充, 不能是浮点数

```
b1 = bytes(8)
print(b1)
print(type(b1))

#b'\x00\x00\x00\x00\x00\x00\x00\x00'
#<class 'bytes'>

b1 = bytes(1.0)

#TypeError: cannot convert 'float' object to bytes
```

```
s1 = b1.decode() #把左边的b1 decode
s1
print("s1:",s1)
print(type(s1))

#'\x00\x00\x00\x00\x00\x00\x00\x00'
#s1:
#<class 'str'>
```

内置函数

- ‘\x00’和 ‘’ 的区别
  - ‘\x00’和空字符 ’’ **是有区别的**

```
x = '\x00'
y = ''
z= b'\x00'
print(x)                    #
print(y)                    #
print(z)                    #
print(x == y)               #False
print(x == z)               #False
print(len(x))               #1
print(len(y))               #0
print('hello\x00world')     #helloworld

a = 'hello\x00world'
"\x00" in a                 #True
```

内置函数

- bytes函数
  - bytes(iterable_of_ints) -> bytes
    - 定义指定内容的字节序列bytes

bytes ([3,2])

#b'\x03\x02'

b1 = bytes([1.1, 2.2, 3, 4])
#TypeError: 'float' object cannot be interpreted as an integer

b1 = bytes([1, 'a', 2, 3])
#TypeError: 'str' object cannot be interpreted as an integer

bytes (range(13))

#b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c'
#'\x09' 是 \t
#'\x0a' 是 \n

b1 = bytes([1, 257])
#ValueError: bytes must be in range(0, 256)

内置函数

- bytes函数
  - bytes(string, encoding[, errors]) -> bytes
    - 定义字节序列bytes，如果包含中文的时候必须设置编码格式
  - bytes(bytes_or_buffer) -> immutable copy of bytes_or_buffer

```
b1 = bytes('abc', 'utf-8’)
print(b1)                                   #b'abc’

b2 = bytes(b'def’)
print(b2)                                   #b'def’

b3 = b'\x64\x65\x66'
print(b3)                                   #b'def’

b4 = bytes(b3)
print("b4 == b3 的结果是 ",(b4 == b3))  #b4 == b3 的结果是  True
print("b4 is b3 的结果是 ", (b4 is b3))    #b4 is b3 的结果是  True
```

内置函数

- bytearray函数
  - **bytearray用法和bytes几乎是一样的，bytearray是可变的，bytes不可变**
    - bytearray(iterable_of_ints) -> bytearray
    - bytearray(string, encoding[, errors]) -> bytearray
    - bytearray(bytes_or_buffer) -> mutable copy of bytes_or_buffer
    - bytearray(int) -> bytes array of size given by the parameter initialized with null bytes
    - bytearray() -> empty bytes array

```
b2 = bytes(b'def’)
b3 = bytearray(b2)
print(b3)
type(b3)
#bytearray(b'def’)
#bytearray
```

内置函数

- repr函数
  - repr(obj, /)
    - 函数str(object)将object转化成为适于人阅读的形式，而repr(object)转换成为解释器读取的形式
    - 打印对象时，会调用对象的__repr__()方法，第四章讲

```
>>> a = 10
>>> type(str(a))
<class 'str'>
>>> type(repr(a))
<class 'str'>
```

```
>>> str('123')
'123'
>>> repr('123')
"'123'"
>>> print(str('123'))
123
>>> print(repr('123'))
'123'                    #使用repr追求明确性，展示出其数据类型信息
>>> print(str(123))
123
>>> print(repr(123))
123
```

```
>>> print('123'.__repr__())
'123'
>>> print('123'.__str__())
123
```

print结合 str() ,调用了对象的 __str__ 方法
print结合 repr() ,调用对象的 __repr__ 方法

内置函数

- iter()
  - iter(iterable) -> iterator
    - 利用iterable对象生成迭代器

```
x = [1,2]
type(x)
y = iter(x)
type(y)
y.next() #可以使用next对y进行查询
#list_Iterator, 是一种Iterator
```

  - iter(callable, sentinel) -> iterator
    - 如果传递了第二个参数，则callable必须是一个可调用的对象（如，函数）。此时，iter函数创建了一个迭代器对象，每次调用这个迭代器对象的__next__()方法时，都会调用callable

偏函数

- functools模块提供了偏函数(Partial function) 的功能(不同于数学中的偏函数)

```
#把字符串转成int，int()函数默认按照十进制进行转换
int('12345')
# 12345

#int()函数还提供额外的base参数，默认值为10。如果传入
#base参数，就可以做N进制-10进制的转换
int('12345', base=8) #传入8进制，转成10进制
#5349
int('12345', 16) #传入16进制，转成10进制
#74565
```

偏函数

- 简化一下，要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，可以定义一个int2()函数，默认把base=2传进去：

```
def int2(x, base=2):
        return int(x, base)

int2('1000000')
#64
int2('1010101')
#85
```

- 再简化一下，functools.partial（偏函数）就是帮助我们创建一个偏函数，不需要我们自己定义int2()

偏函数

- 要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，可以定义一个int2()的函数，默认把base=2传进去：

```
import functools #导入模块
int2 = functools.partial(int, base=2)
int2('1000000')
#64
int2('1010101')
#85
```

传入一个函数

要固定住的参数（默认值）

返回值赋值给int2，int2()就是一个新的函数

```
#偏函数中base参数设定默认值为2，但也可以在新函数调用时传入其他值
int2('1000000', base=10)
#1000000
```

偏函数

- 创建偏函数时，可以接收 函数对象、*args(普通参数)和**kw(关键字参数)这3个参数

```
#当我们如此设置偏函数时，相当于固定了int()函数的关键字参数base
int2 = functools.partial(int, base=2)
int2('10010')

#上面偏函数的方法就相当于/等价于
kw = { 'base': 2 } #用字典构造base关键字参数
int('10010', **kw)

#当传入非关键字参数时，例如：
max10 = functools.partial(max, 10)
max10(5, 6, 7)
#10
#10并不是关键字参数，但偏函数会把10加进*args，所以
#虽然我们只传入5, 6, 7.但是10已经在*arg里了，最大值是10
```

os.path.split()

- os.path.split()
  - 传入一个文件的全路径作为参数
  - 将文件名和目录分开，以 “PATH” 中最后一个 ‘/’ 作为分隔符，返回一个元组，索引0为目录(路径)，索引1为文件名

```
import os
os.path.split('C:/soft/python/test.py’)
#('C:/soft/python', 'test.py')
os.path.split('C:/soft/python/test’)
#('C:/soft/python', 'test')
os.path.split('C:/soft/python/’)
#('C:/soft/python', '')
```

os.path.join()

- os.path.join()
  - 用于路径拼接
  - 传入多个路径作为参数
  - 会从最后一个以”/”开头的参数开始拼接，之前的参数全部丢弃。

```
import os
print(os.path.join('aaaa','/bbbb','ccccc.txt'))
# /bbbb/ccccc.txt
print(os.path.join('/aaaa','/bbbb','/ccccc.txt'))
# /ccccc.txt
print(os.path.join('aaaa','bbb','ccccc.txt'))
#aaaa/bbb/ccccc.txt
```

‘/bbbb’之前的参数被丢弃

‘/ccccc.txt’之前的参数被丢弃

会自动添加/

  - 拼接合法路径时，中间的路径不要以 ‘/‘ 开头

- 感谢