Python程序设计

函数 - 作用域、参数和进阶应用

刘安
苏州大学，计算机科学与技术学院

http://web.suda.edu.cn/anliu/

本节涉及到的知识点

- 变量作用域
- 参数匹配模式
- lambda表达式构造匿名函数
- 函数式编程工具
- 生成器函数
- 生成器表达式

作用域

- 变量名可以使用的地方

```
>>> def area(r):
        pi = 3.14
        return pi * r ** 2

>>> area(1)
3.14
>>> pi
Traceback (most recent call last):
  File "<pyshell#750>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
```

def也是复合语句，所以首行末尾需要冒号，其后的代码块需要缩进

局部/非局部/全局变量

- 变量的作用域取决于**它被赋值的地方**
  - 如果一个变量在函数内赋值，它对于该函数而言是**局部**的，即只能在函数内部使用该变量
  - 如果一个变量在外层函数中赋值，对于内层函数而言，它是**非局部**的
  - 如果一个变量在所有函数外赋值，它对整个文件（交互式命令行可以看成是一个文件）来说是**全局**的
    - 全局变量可以在函数内部被**读取**
    - 如果要在函数内部对全局变量赋值，必须通过global语句对该变量进行声明（否则就创建了一个局部变量）

局部变量和全局变量

```
>>> def area(r):
        pi = 3.14
        return pi * r ** 2

>>> area(1)
3.14
```

pi是局部变量

```
>>> pi = 3.142
>>> def area(r):
        return pi * r ** 2

>>> area(1)
3.142
```

pi是全局变量

```
>>> pi = 3.142
>>> def area(r):
        pi = 3.14
        return pi * r ** 2

>>> area(1)
3.14
>>> pi
3.142
```

函数area中创建局部变量pi
没有影响全局变量pi的值

```
>>> pi = 3.142
>>> def area(r):
        global pi
        pi = 3.14
        return pi * r ** 2

>>> area(1)
3.14
>>> pi
3.14
```

函数area中通过global语句
声明其中的pi是全局变量

变量名解析的LEGB规则

- 遇到一个变量名时，依次查找：
  - L (局部作用域)：在一个函数内部被赋值的变量，但不包括被global关键字声明的变量
  - E (外层作用域)：外层函数的局部作用域中的变量，从内向外（直到全局作用域）
  - G (全局作用域)：所有函数之外被赋值的变量，或者在某个函数中通过global关键字声明的变量
  - B (内置作用域)：在builtins模块中预先赋值好的名称
- 如果再上述过程中没有查到，抛出异常

内置作用域

- 实际上是一个名为builtins的标准库模块，其中预先定义了很多常用的变量名，比如函数sum和print
- 注意：该模块并不是内置的，使用前需要导入
- 回顾：random也是一个标准库模块，如果要使用其中预定义的变量，比如函数randint，必须要导入该模块

```
>>> random.randint(0, 10)
Traceback (most recent call last):
  File "<pyshell#915>", line 1, in <module>
    random.randint(0, 10)
NameError: name 'random' is not defined
>>> import random
>>> random.randint(0, 10)
9
>>> sum([1, 2, random.randint(0, 10)])
10
```

为什么sum可以直接用呢?

内置作用域

- 导入builtins模块，然后直接使用其中定义的变量名
- 不导入builtins模块，根据LEGB规则查找

```
>>> sum([1, 2, random.randint(0, 10)])
10
>>>
>>> import builtins
>>> builtins.sum([1, 2, 3])
6
>>> sum is builtins.sum
True
```

变量名解析举例

- 分析右侧代码中各个变量名所在的作用域
  - L：全局
  - m (line 2)：全局
  - s：全局
  - f：全局
  - m (line 8)：局部
  - sum：内置
  - min：内置
  - print：内置

```
 1 L = [1, 2, 3]
 2 m = 0
 3 s = 0
 4
 5 def f():
 6     global s
 7     s = sum(L)
 8     m = min(L)
 9
10 f()
11 print(s, m)
```

变量名解析举例

- 根据LEGB规则的查找顺序，局部作用域中的变量会“隐藏”全局作用域或者内置作用域中的同名变量，全局作用域中的变量会“隐藏”内置作用域中的同名变量

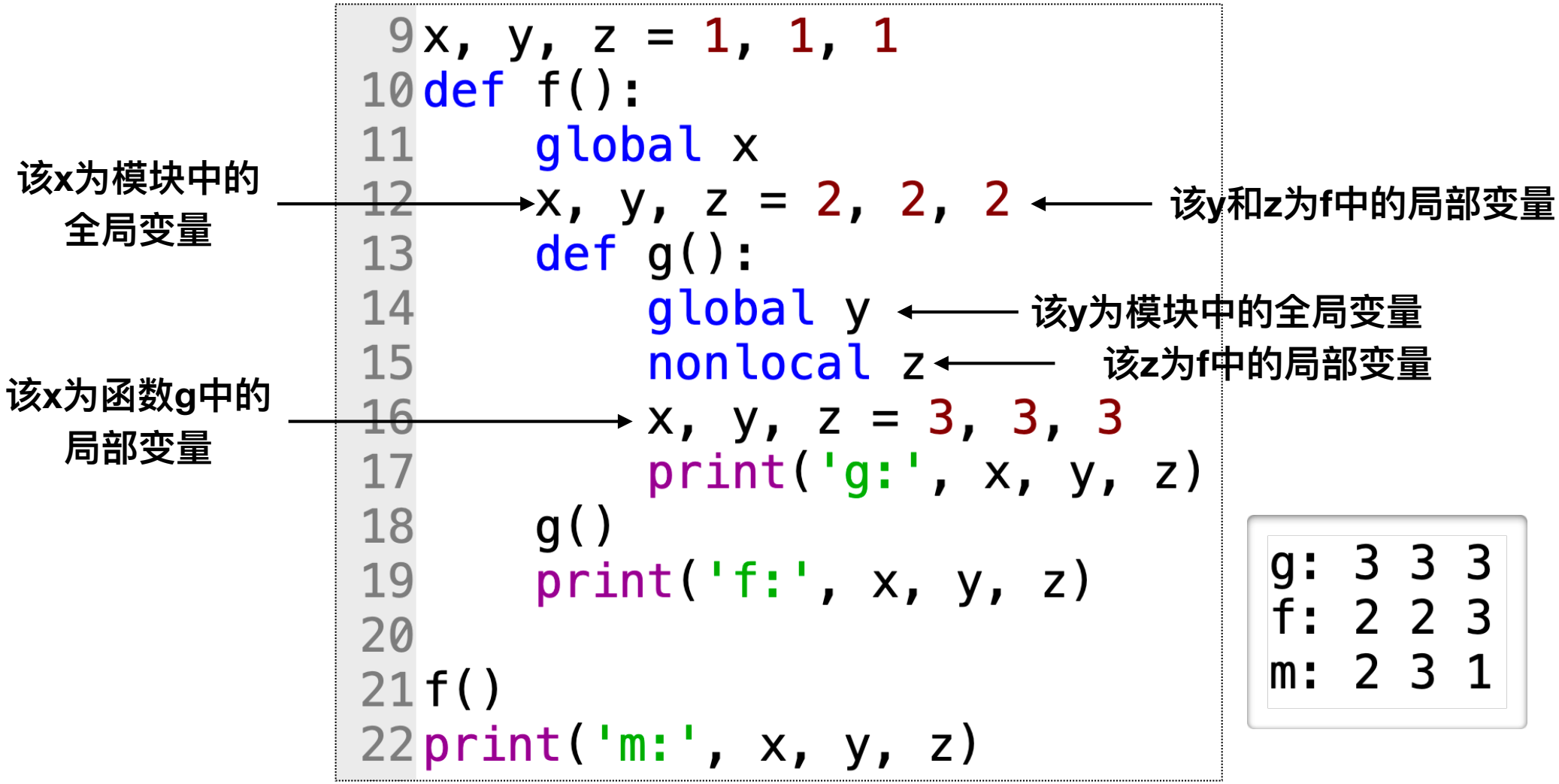
```
>>> import builtins
>>> min = 0 # 变量min在全局作用域，不再是内置作用域中的函数
>>> def f():
        sum = builtins.sum([1, 2, 3]) # 变量sum在局部作用域
        min = builtins.min([1, 2, 3]) # 变量min在局部作用域


>>> f()
>>> print(sum, min)
<built-in function sum> 0
```

global和nonlocal

- 函数f内部的赋值语句x=value的效果受global和nonlocal影响
- global（参见第5页的4个例子）
  - 如果在函数f内部用global对x声明
    - 如果在全局作用域中已经有x，那么会修改x的值
    - 否则，创建x并赋值（x仍然属于全局作用域）
  - 否则，创建x并赋值（x属于函数f的局部作用域，此时可能在全局作用域中存在同名的变量x）

global和nonlocal

- 函数f内部的赋值语句x=value的效果受global和nonlocal影响
- nonlocal
  - 如果在函数f内部用nonlocal对x声明
    - 如果在函数f的任意一个外层函数中已经有x（注意：可能多个外层函数都有x），那么会修改包含函数f的最内层函数中的x的值
    - 否则抛出异常SyntexError: no binding for nonlocal ‘x’（注意：即使在全局作用域中已经有x）
  - 否则，创建x并赋值（x属于函数f的局部作用域）




```
 9 x, y, z = 1, 1, 1
10 def f():
11     global x
12     x, y, z = 2, 2, 2
13     def g():
14         global y
15         nonlocal z
16         x, y, z = 3, 3, 3
17         print('g:', x, y, z)
18     g()
19     print('f:', x, y, z)
20
21 f()
22 print('m:', x, y, z)
```

```
g: 3 3 3
f: 2 2 3
m: 2 3 1
```

```
 9 x = 1
10 def f():
11     x = 2
12     def g():
13         x = 3
14         def h():
15             x = 4
16         h()
17         print('g:', x)
18     g()
19     print('f:', x)
20 f()
21 print('m:', x)
```

	g	f	m
x	3	2	1

```
 9 x = 1
10 def f():
11     x = 2
12     def g():
13         x = 3
14         def h():
15             nonlocal x
16             x = 4
17         h()
18         print('g:', x)
19     g()
20     print('f:', x)
21 f()
22 print('m:', x)
```

	g	f	m
x	4	2	1

```
 9 x = 1
10 def f():
11     x = 2
12     def g():
13         #x = 3
14         def h():
15             nonlocal x
16             x = 4
17         h()
18         print('g:', x)
19     g()
20     print('f:', x)
21 f()
22 print('m:', x)
```

	g	f	m
x	4	4	1

```
 9 x = 1
10 def f():
11     x = 2
12     def g():
13         x = 3
14         def h():
15             nonlocal x
16             x = 4
17         h()
18         print('g:', x)
19     g()
20     print('f:', x)
21 f()
22 print('m:', x)
```

	g	f	m
x	4	2	1

```
    nonlocal x
    ^
SyntaxError: no binding for nonlocal 'x' found
```

```
 9 x = 1
10 def f():
11     x = 2
12     def g():
13         #x = 3
14         def h():
15             nonlocal x
16             x = 4
17         h()
18         print('g:', x)
19     g()
20     print('f:', x)
21 f()
22 print('m:', x)
```

```
 9 x = 1
10 def f():
11     #x = 2
12     def g():
13         #x = 3
14         def h():
15             nonlocal x
16             x = 4
17         h()
18         print('g:', x)
19     g()
20     print('f:', x)
21 f()
22 print('m:', x)
```

	g	f	m
x	4	4	1

参数匹配模式

- 位置参数
- 关键字参数
- 默认值参数
- 可变长参数
- 可变长参数解包

位置参数

- 根据参数和值的位置（从左到右）将值传递给参数

```
>>> def f(sid, age, grade):
        print(sid, age, grade)


>>> f(101, 19, 91)
101 19 91
```

关键字参数

- 在函数调用时，通过name=value来明确指定名为name的参数接受值value

```
>>> def f(sid, age, grade):
        print(sid, age, grade)


>>> f(101, 19, 91)
101 19 91
>>>
>>> f(grade = 91, sid = 101, age = 19)
101 19 91
```

默认值参数

- 在函数定义时，通过name=value为name参数指定默认值value。如果在函数调用时没有为name参数传入值，则使用该默认值

```
>>> def f(sid = 100, age = 19, grade = 90):
        print(sid, age, grade)


>>> f() # 未提供任何值
100 19 90
>>>
>>> f(101) # 仅提供一个值，根据位置，该值赋给sid
101 19 90
>>>
>>> f(102, 20) # 提供两个值，根据位置，分别赋给sid和age
102 20 90
```

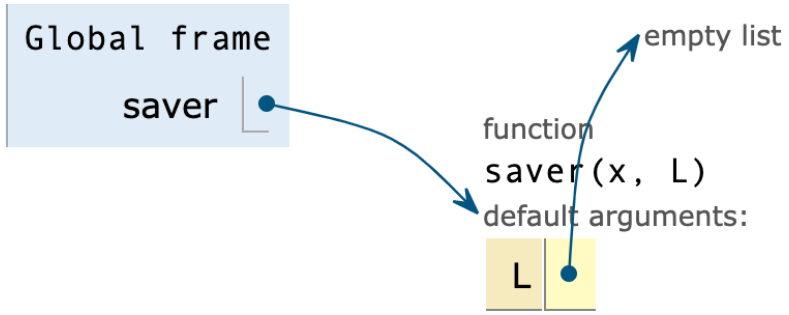
默认值参数和可变对象

- 当def语句执行时，默认值参数被求值并保存下来
- 如果默认值是一个可变对象比如列表，那么要特别注意

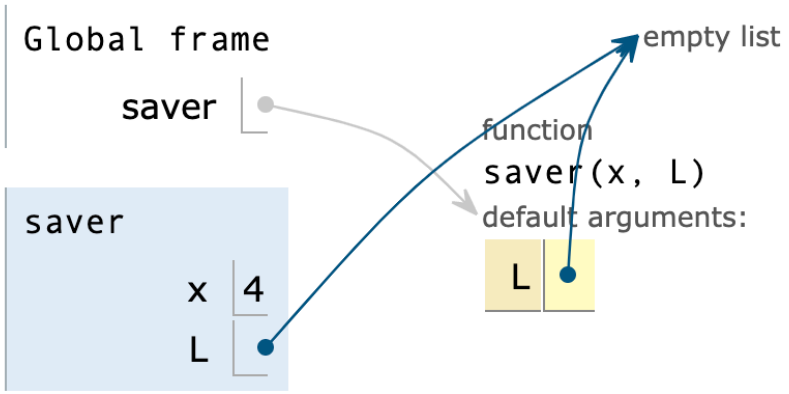
```
>>> def saver(x, L = []): # 将x保存在列表L中，如果没指定L，使用[]
        L.append(x)
        print(L)


>>> L = [1]
>>> saver(3, L) # 将3保存至L中
[1, 3]
>>>
>>> saver(4) # 将4保存至默认空列表中
[4]
>>>
>>> saver(6) # 将6保存至默认空列表中
[4, 6]
```

这里为什么不是[6]呢?

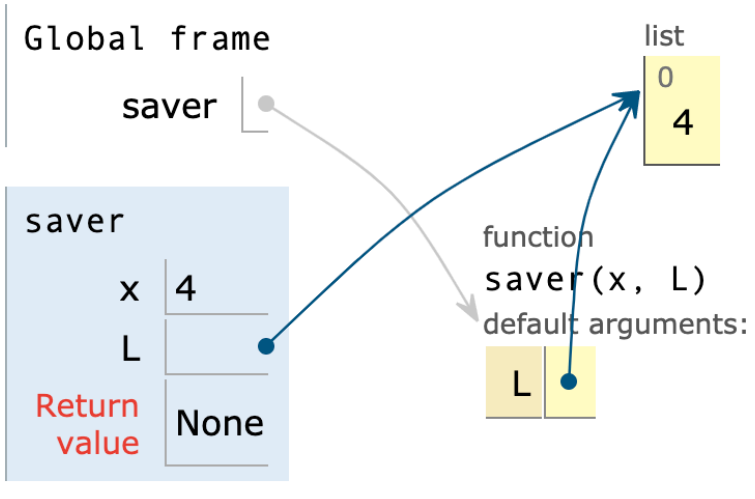
```
1 def saver(x, L = []):
2     L.append(x)
3
4 saver(4)
5 saver(6)
```



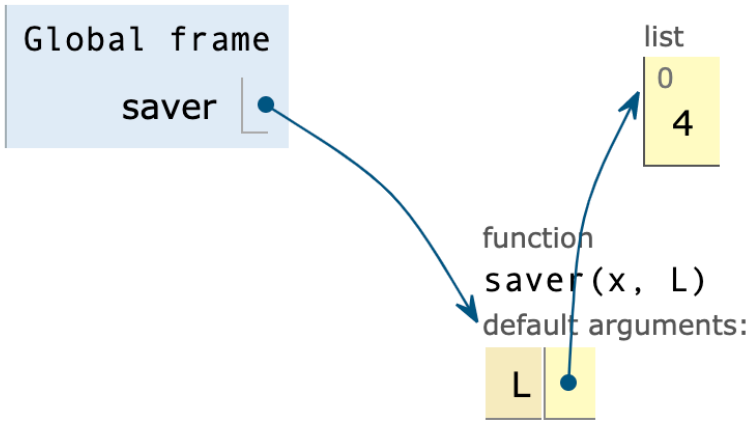

```
1 def saver(x, L = []):
2     L.append(x)
3
4 saver(4)
5 saver(6)
```



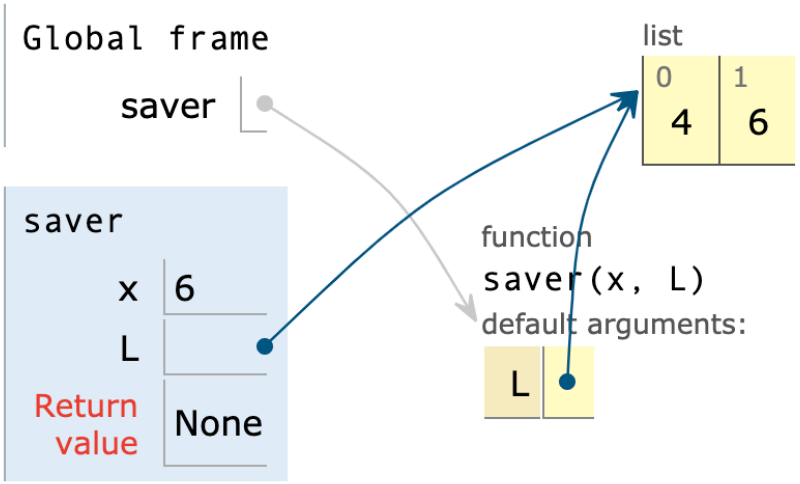

```
1 def saver(x, L = []):
2     L.append(x)
3
4 saver(4)
5 saver(6)
```



```
1  def saver(x, L = []):
2      L.append(x)
3
4  saver(4)
5  saver(6)
```




```
1  def saver(x, L = []):
2      L.append(x)
3
4  saver(4)
5  saver(6)
```



默认值参数和可变对象

- 当def语句执行时，默认值参数被求值并保存下来
- 如果期望的默认值是一个可变对象，可以如下处理

```
>>> def saver(x, L = None): # 将x保存在列表L中，如果没指定L，使用[]
        if L is None:
            L = [] # 每一次都会产生一个新的空列表
        L.append(x) # 如果没有指定L，那么就将x加入空列表
        print(L)


>>> saver(1)
[1]
>>> saver(2)
[2]
```

可变长参数

- 目的：让函数可以接受任意数量的参数
- 在函数定义中，*args将不能匹配的基于位置的参数放入元组args中，**kwargs将关键字参数放入字典kwargs中

```
>>> def f(*args): # *args将所有基于位置的参数放入一个元组中
        print(args) # 输出元组args的内容


>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3)
(1, 2, 3)
```

可变长参数

- 目的：让函数可以接受任意数量的参数
- 在函数定义中，*args将不能匹配的基于位置的参数放入元组args中，**kwargs将关键字参数放入字典kwargs中

```
>>> def f(**kwargs): # **kwargs将所有关键字参数放入字典kwargs中
        print(kwargs)


>>> f()
{}
>>> f(a = 1, b = 2)
{'a': 1, 'b': 2}
```

可变长参数

- 目的：让函数可以接受任意数量的参数
- 在函数定义中，*args将不能匹配的基于位置的参数放入元组args中，**kwargs将关键字参数放入字典kwargs中

```
>>> def f(a, *args, **kwargs):
        print(a, args, kwargs)


>>> f(1, 2, 3, x = 1, y = 2)
1 (2, 3) {'x': 1, 'y': 2}
```

  - 1按照位置传递给a，剩余的位置参数2和3放入元组args中，关键字参数x和y放入字典kwargs中

参数解包

- 在调用函数时通过*解包参数序列，通过**解包字典，使其成为关键字参数

```
>>> def f(a, b, c):
        print(a, b, c)


>>> x = [1, 2, 3]
>>> f(*x) # f函数需要3个参数，*将列表x解包
1 2 3
>>>
>>> D = {'b':1, 'c':2, 'a':3}
>>> f(**D) # f函数需要3个参数，**将字典D解包
3 1 2
```

参数模型小结

位置参数	值和参数根据位置从左至右进行匹配
关键字参数	调用函数时通过name=value指定 参数name获得值value
默认值参数	def语句中通过name=value指定 如果调用函数时值过少使得参数name 没有获得值，则使用默认值value
可变长参数	def语句中通过*收集不能匹配的位置参数 通过**收集所有的关键字参数
参数解包	调用函数时通过*解包实参序列 通过**解包实参字典

通过lambda表达式创建函数

- def语句创建一个函数，并将其赋给一个变量
- lambda表达式也生成一个函数
  - lambda 参数1，参数2，…，参数n：表达式

```
>>> lambda x, y, z : x + y + z
<function <lambda> at 0x10f814e18>
>>>
>>> add = lambda x, y, z : x + y + z # 将变量add绑定到函数
>>> add(1, 2, 3) # 调用函数
6
>>>
>>> (lambda x, y, z : x + y + z) (1, 2, 3) # 直接调用函数
6
```

匿名函数：lambda

- lambda表达式创建的函数没有名字，所以也称匿名函数
  - def语句在定义函数时，必然将函数对象与一个变量绑定
- lambda表达式创建的函数没有return语句
  - 表达式本身有一个值，该值就是函数的返回值
- 作用域查找规则和参数匹配模式也适用于lambda函数

```
>>> (lambda a = 'hello, ', b = 'world': a + b)() # 使用默认值
'hello, world'
>>>
>>> (lambda a = 'hello, ', b = 'world': a + b)(b = 'python')
'hello, python'
```

lambda的使用场景

- 回顾：sort方法和sorted函数都接受一个key参数，其值是一个函数对象，默认值为None
- 可以使用lambda表达式定义函数传递给key
  - 将正整数按照各位数字和的大小来排序

```
>>> L = [123, 45, 6, 7, 18, 9, 22]
>>> sorted(L, key = lambda x : sum([int(c) for c in str(x)]))
[22, 123, 6, 7, 45, 18, 9]
```

  - 根据列表的特定元素来排序

对学生列表使用简单排序

- 按照年龄升序排序

```
>>> sorted(students, key = lambda x : x[1])
[[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85],
 [1001, 16, 90, 95, 90], [1006, 17, 80, 95, 95]]
```

- 按照总成绩升序排序

```
>>> sorted(students, key = lambda x : sum(x[2:]))
[[1006, 17, 80, 95, 95], [1009, 15, 95, 90, 85],
[1001, 16, 90, 95, 90], [1004, 15, 90, 85, 100]]
```

```
>>> students = [
    [1001, 16, 90, 95, 90],
    [1004, 15, 90, 85, 100],
    [1006, 17, 80, 95, 95],
    [1009, 15, 95, 90, 85],
    ]
```

学号	年龄	语文成绩	数学成绩	英语成绩
1001	16	90	95	90
1004	15	90	85	100
1006	17	80	95	95
1009	15	95	90	85

对学生列表使用多关键字排序

- 首先根据总成绩降序排序，如果总成绩相同，根据语文成绩降序排序，如果语文成绩还相同，根据年龄升序排序
- lambda表达式返回一个元组（回顾：元组如何比较大小）
  - 总成绩（降序）：-sum(x[2:])
  - 语文成绩（降序）：-x[2]
  - 年龄（升序）：x[1]

```
>>> sorted(students, key = lambda x : (-sum(x[2:])
, -x[2], x[1]))
[[1004, 15, 90, 85, 100], [1001, 16, 90, 95, 90],
[1009, 15, 95, 90, 85], [1006, 17, 80, 95, 95]]
```

关于lambda的奇怪问题

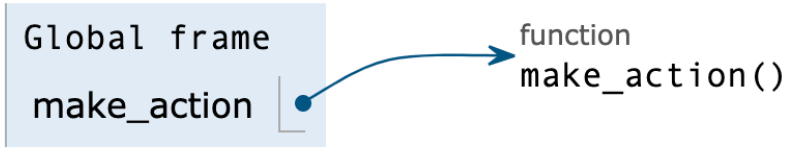
- 思考下列代码

```
>>> def make_action():
        acts = []
        for i in range(5):
            acts.append(lambda x: i**x)
        return acts

>>> acts = make_action()
>>> acts[0](2) # should be 0**2=0?
16
>>> acts[1](2) # should be 1**2=1?
16
```

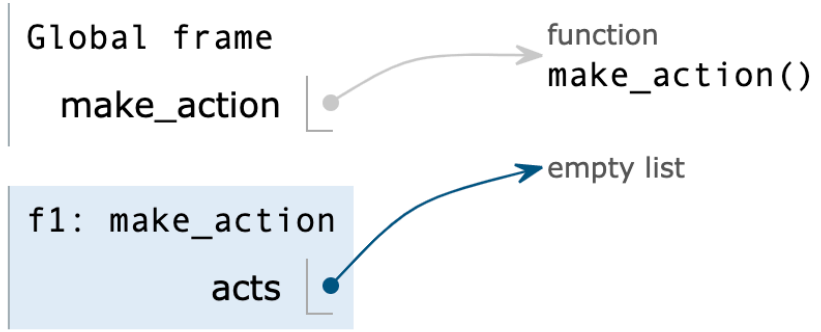
- 为什么i的值没有记住？

```
1 def make_action():
2     acts = []
3     for i in range(5):
4         acts.append(lambda x: i**x)
5     return acts
6
7 acts = make_action()
8 acts[0](2)
9 acts[1](2)
```

执行def语句只是创建一个函数对象并绑定到make_action上，并没有执行其中的代码

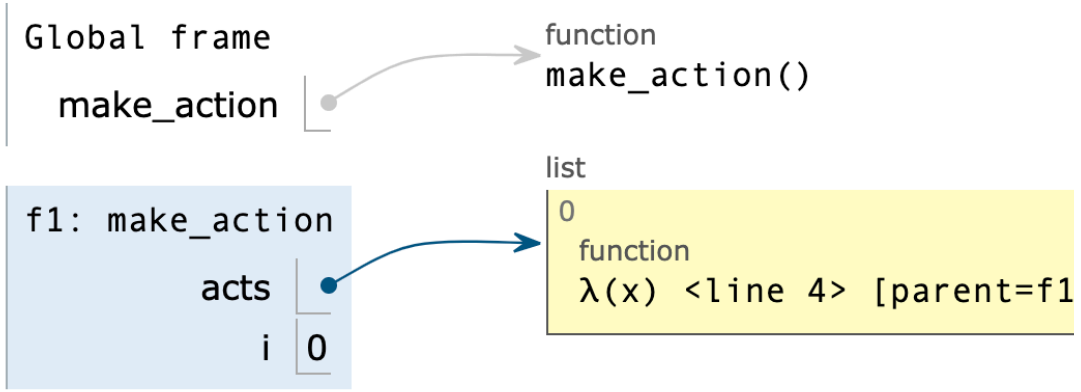



```
1 def make_action():
2     acts = []
3     for i in range(5):
4         acts.append(lambda x: i**x)
5     return acts
6
7 acts = make_action()
8 acts[0](2)
9 acts[1](2)
```



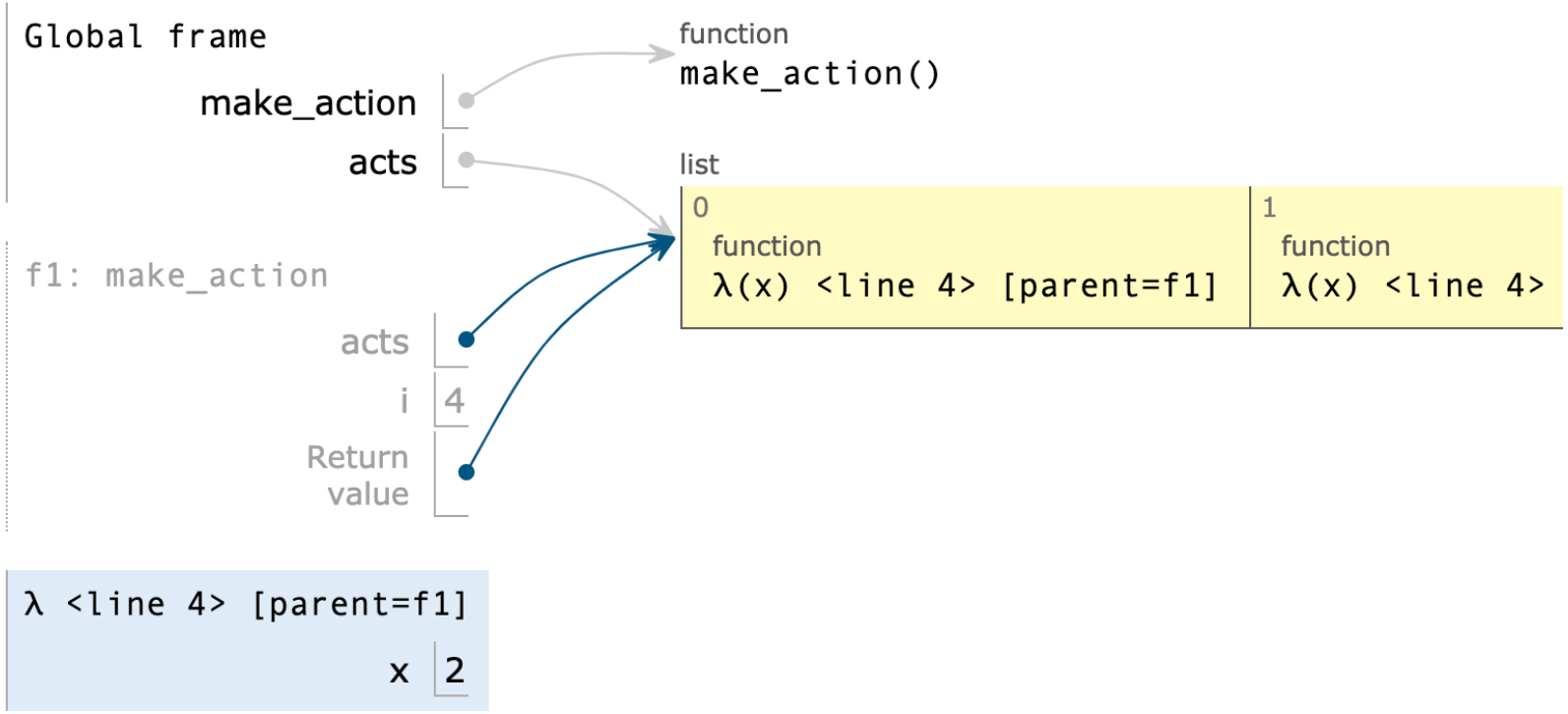
这里也只是创建一个匿名函数对象，将其放入列表中，没有执行函数代码，所以i的值没有记住！

```
1 def make_action():
2     acts = []
3     for i in range(5):
4         acts.append(lambda x: i**x)
5     return acts
6
7 acts = make_action()
8 acts[0](2)
9 acts[1](2)
```



```
1  def make_action():
2      acts = []
3      for i in range(5):
4          acts.append(lambda x: i**x)
5      return acts
6
7  acts = make_action()
8  acts[0](2)
9  acts[1](2)
```

现在开始执行lambda函数，x的值是2，i的值需要进行查找，
在函数make_action中，找到变量i，其值为4，
所以这里lambda函数返回24，也就是16**

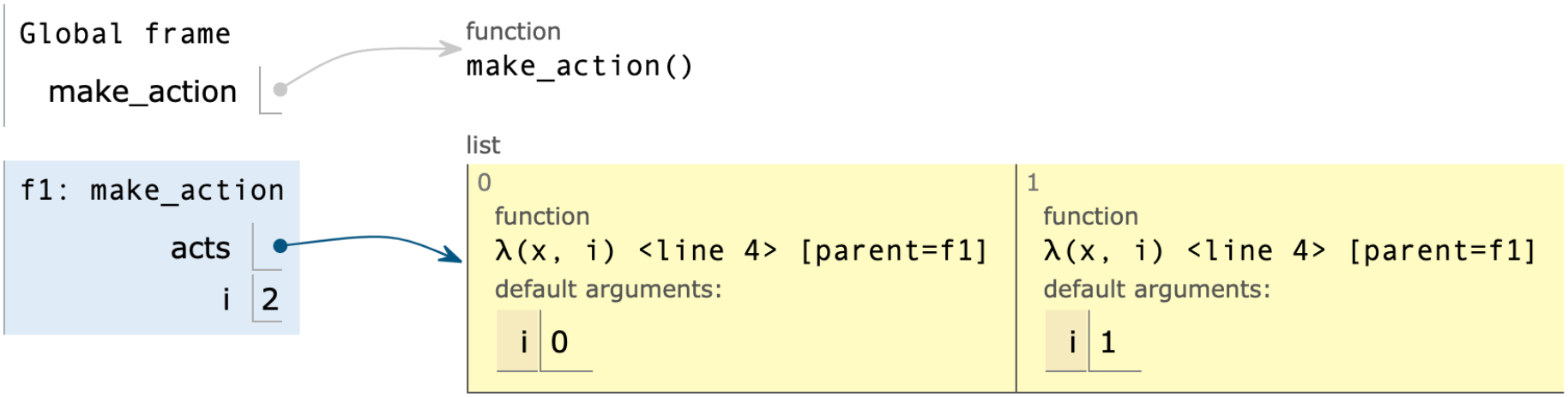


使用默认值参数

- 默认值参数在函数定义时被求值（而不是函数调用时），所以每次将一个lambda函数加入列表时，参数i都接受了一个默认值

```
>>> def make_action():
        acts = []
        for i in range(5):
            acts.append(lambda x, i = i: i**x)
        return acts

>>> acts = make_action()
>>> acts[0](2) # should be 0**2=0?
0
>>> acts[1](2) # should be 1**2=1?
1
```

```
1  def make_action():
2      acts = []
3      for i in range(5):
4          acts.append(lambda x, i = i: i**x)
5      return acts
6
7  acts = make_action()
8  acts[0](2)
9  acts[1](2)
```



默认值参数在函数定义时求值，
所以每次将一个lambda函数加入列表时，
参数i都接受了一个默认值

函数式编程工具

- 需求：在可迭代对象中的每个元素上调用一个函数
  - 将列表A中的每个元素传入函数f，返回值保存到列表B中
- 目的：简化该类需求的编程实现
- 内置函数
  - map和filter
- 标准库functools中的reduce函数

```
>>> def f(x):
        return x + 10

>>> A = [1, 2, 3, 4]
>>> B = []
>>> for x in A:
        B.append(f(x))

>>> B
[11, 12, 13, 14]
```

https://docs.python.org/3/library/functools.html

函数式编程工具map

- map(func, iterable)：在可迭代对象iterable中的每个元素上调用函数func，将所有的返回值放入一个可迭代对象，返回该对象

```
>>> map(lambda x: x+10, [1, 2, 3, 4])
<map object at 0x1115d1b00>
>>>
>>> list(map(lambda x: x+10, [1, 2, 3, 4]))
[11, 12, 13, 14]
>>>
>>> list(map(lambda x, y: x**y, [1, 2, 3], [2, 3, 4]))
[1, 8, 81]
>>>
>>> list(map(str.upper, ['hello', 'python']))
['HELLO', 'PYTHON']
```

函数式编程工具filter

- filter(func, iterable)：在可迭代对象iterable中的每个元素上调用函数func，将所有返回值为True的元素放入一个可迭代对象，返回该对象

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>>
>>> list(filter(lambda x: x % 2 == 0, range(-5, 5)))
[-4, -2, 0, 2, 4]
>>>
>>> list(filter(lambda x: x > 0, range(-5, 5)))
[1, 2, 3, 4]
```

函数式编程工具reduce

- reduce(func, iterable)：基于当前的聚合值a，在可迭代对象iterable中的每个元素x上调用函数func(x, a)生成一个新的聚合值，返回最终的聚合值

```
>>> import functools
>>> functools.reduce(lambda x, y: x+y, [1, 2, 3, 4])
10
>>>
>>> import operator
>>> functools.reduce(operator.add, [1, 2, 3, 4])
10
>>>
>>> functools.reduce(operator.mul, [1, 2, 3, 4])
24
```

https://docs.python.org/3/library/operator.html

用列表推导模拟map和filter

```
>>> list(map(lambda x: x**2, range(6)))
[0, 1, 4, 9, 16, 25]
>>>
>>> [x ** 2 for x in range(6)]
[0, 1, 4, 9, 16, 25]
>>>
>>> list(filter(lambda x: x%2==1, range(6)))
[1, 3, 5]
>>>
>>> [x for x in range(6) if x % 2 == 1]
[1, 3, 5]
>>>
>>> list(map(lambda x: x**2, filter(lambda x: x%2==1, range(6))))
[1, 9, 25]
>>>
>>> [x ** 2 for x in range(6) if x % 2 == 1]
[1, 9, 25]
```

编写自己的range函数

- 回顾：range函数返回一个整数序列

```
>>> for i in range(0, 5):
        print(i, end = ', ')


0, 1, 2, 3, 4,
```

- range函数的一种实现

```
>>> def my_range(start, stop, step = 1):
        numbers = []
        while start < stop:
            numbers.append(start)
            start += step
        return numbers

>>> for i in my_range(0, 5):
        print(i, end = ', ')
```

my_range的不足

- range的一种实现

```
>>> def my_range(start, stop, step = 1):
        numbers = []
        while start < stop:
            numbers.append(start)
            start += step
        return numbers
```

- 整数序列预先存放在一个列表中，如果该序列很长（比如十亿个整数），那么需要的内存可能无法被满足！

- 在for循环中，其实不需要一次性获得整个序列，而是每次循环获得序列的一个元素

- range的另一种实现：for循环可以验证函数my_range_v1确实生成了一个整数序列，但是该函数没有使用return返回一个列表，而是使用yield产生所需序列中的一个元素

- yield产生序列中的一个元素后，函数的执行会被挂起，当外部代码需要序列中的下一个元素时，函数会恢复上次挂起时的状态并继续运行

```
>>> def my_range_v1(start, stop, step = 1):
        while start < stop:
            yield start # not return here!
            start += step
```

```
>>> for i in my_range_v1(0, 5):
        print(i, end = ', ')


0, 1, 2, 3, 4,
```

生成器函数

- 生成器函数
  - 通过yield返回结果后挂起本次执行（保存相关状态）
  - 再次调用函数时，通过恢复上次保存的状态信息，从挂起的地方继续执行
- 常规函数
  - 通过return返回结果后结束本次执行
  - 再次调用函数时，从函数开头执行

生成器函数的返回值

- 生成器函数的返回值实际上是一个生成器
- 生成器是一种可迭代对象，可以通过调用next函数来获得其中的元素

```
>>> g = my_range_v1(0, 5)
>>> g
<generator object my_range_v1 at 0x10416bf10>
>>> next(g) # g is iterable
0
>>> next(g)
1
```

生成器函数的执行过程

```
>>> def seq_3():
        print('Start')
        for i in range(3):
            print('Before yield: i =', i)
            yield i
            print('After yield: i =', i)
        print('End')


>>> def test_seq_3():
        a = seq_3()
        print('Before loop')
        for i in a:
            print(i)
        print('After loop')
```

```
>>> test_seq_3()
Before loop
Start
Before yield: i = 0
0
After yield: i = 0
Before yield: i = 1
1
After yield: i = 1
Before yield: i = 2
2
After yield: i = 2
End
After loop
```

关于斐波那契数列的问题

- 问题：1000之内的斐波那契数中有多少个奇数？

```
>>> def fib_naive():
        i, j = 1, 1 # first two elements
        cnt = 0 # num of odd
        while i < 1000:
            if i % 2: # new odd
                cnt += 1
            i, j = j, i + j # new fib elem
        return cnt
```

- fib_naive函数既生成了数据（斐波那契数列），又处理了数据（统计奇数个数）
- 如果将两者分开，函数功能更加清晰

生成数据

- 生成器函数
  - 可以根据需要生成任意大的斐波那契数

```
>>> def fibonacci():
        i, j = 1, 1 # first two elements
        while True:
            yield i
            i, j = j, i + j # new fib elem
```

```
>>> for i in fibonacci():
        if i >= 1000:
            break
        print(i, end = ', ')


1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
```

处理数据

- 普通函数：统计序列中奇数个数，不关心序列如何产生

```
>>> def odd_in_seq():
        cnt = 0
        for x in fibonacci():
            if x >= 1000:
                break
            if x % 2:
                cnt += 1
        return cnt

>>> odd_in_seq()
11
```

- 如何统计别的序列中的奇数个数？
  - 比如$a(n) = (11 \times n^2 - 11 \times n + 2)/2$

生成数据

- 生成序列$a(n) = (11 \times n^2 - 11 \times n + 2)/2$

```
>>> def seq_11_g():
        n = 1
        while True:
            yield (11*n**2-11*n+2)//2
            n += 1


>>> for i in seq_11_g():
        if i >= 1000:
            break
        print(i, end = ', ')


1, 12, 34, 67, 111, 166, 232, 309, 397, 496, 606, 727, 859,
```

处理数据

- 统计序列$a(n) = (11 \times n^2 - 11 \times n + 2)/2$中小于1000的奇数个数

```
>>> def odd_in_seq():
        cnt = 0
        for x in seq_11_g():
            if x >= 1000:
                break
            if x % 2:
                cnt += 1
        return cnt

>>> odd_in_seq()
7
```

```
>>> def odd_in_seq():
        cnt = 0
        for x in fibonacci():
            if x >= 1000:
                break
            if x % 2:
                cnt += 1
        return cnt

>>> odd_in_seq()
11
```

生成器表达式

- 统计range(1000)中能被2或者3整除的元素个数

```
>>> L = [x for x in range(1000) if not x % 2 or not x % 3]
>>> len(L)
667
```

- 不足：如果L很长，需要大量内存，甚至无法被满足！

- 能否不一次性生成整个结果序列，而是逐个元素生成呢？

- 生成器表达式：包含在圆括号中的列表推导表达式

  - 该表达式的值是一个生成器对象

```
>>> g = (x for x in range(1000) if not x % 2 or not x % 3)
>>> g # g is a generator
<generator object <genexpr> at 0x10416bd58>
```

生成器表达式

- 生成器表达式：包含在圆括号中的列表推导表达式
  - 该表达式的值是一个生成器对象

```
>>> g = (x for x in range(1000) if not x % 2 or not x % 3)
>>> g # g is a generator
<generator object <genexpr> at 0x10416bd58>
>>>
>>> len(g)
Traceback (most recent call last):
  File "<pyshell#162>", line 1, in <module>
    len(g)
TypeError: object of type 'generator' has no len()
```

  - 如何求g中元素的个数呢？

生成器表达式

- 统计range(1000)中能被2或者3整除的元素个数

```
>>> g = (1 for x in range(1000)
         if not x % 2 or not x % 3) # 每个符合条件的x生成一个1
>>>
>>> sum(g) # 对g中所有元素(均为1)求和，结果就是元素的个数
667
```

生成器是单遍迭代对象

- 生成器的内容（即对应序列中的元素）使用完毕之后清空，如果需要再次迭代，必须重新生成一个生成器

```
>>> g = (x for x in range(3))
>>> sum(g)
3
>>> sum(g)
0
>>> g = (x for x in range(3))
>>> sum(g)
3
```