

迭代器和生成器

a generator
expression



is

a generator



always is

an iterator



next()

*lazily produce
next value*

a generator
function



is

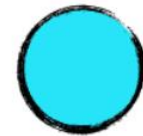
always is

iter()



(an) iterable

typically is



a container

produces



{list, set, dict}
comprehension

迭代器(**iterator**)

- 迭代器是带状态的对象，可用来表示有限或者无限的数据流
- 使用**next()**函数可以获取迭代器中的下一个值，如果迭代器中没有更多的元素，则抛出StopIteration异常
- 使用**iter()**函数可以为数据结构（可迭代对象）创建一个迭代器对象

可迭代对象(**iterable**)

- 可以返回迭代器的对象

```
x = [1, 2, 3]
```

```
y = iter(x)
```

```
z = iter(x)
```

```
next(y)                # => 1
```

```
next(y)                # => 2
```

```
next(z)                # => 1
```

```
type(x)                # => <class 'list'>
```

```
type(y)                # => <class 'list_iterator'>
```

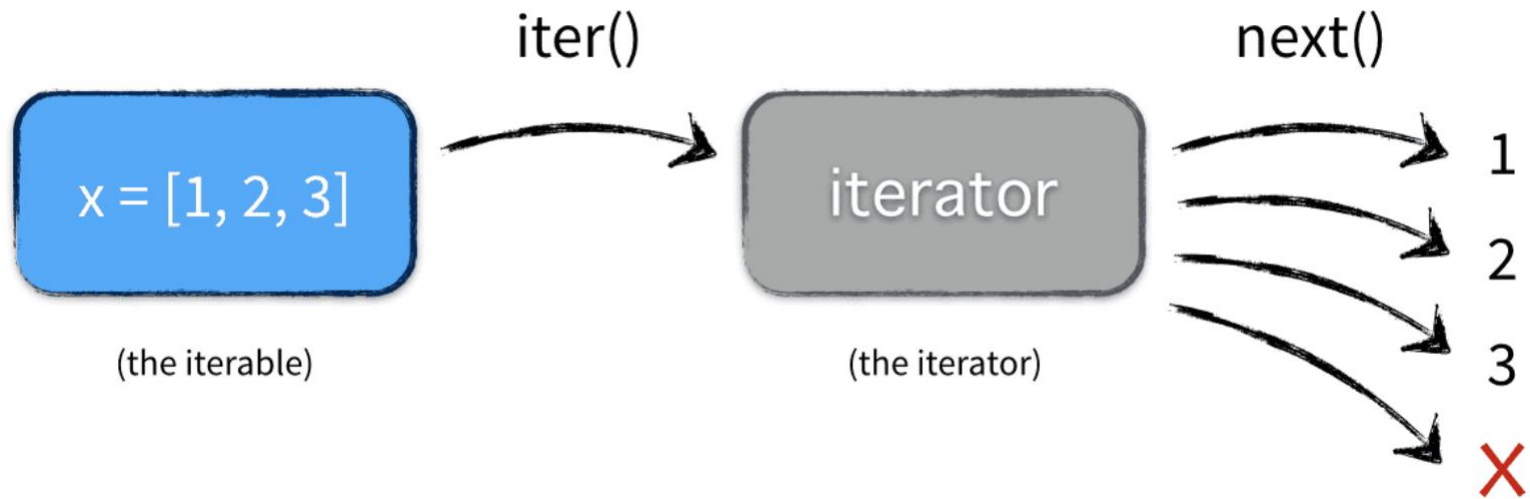
For循环使用了迭代器

```
for data in data_source:  
    process(data)
```

is really

```
for data in iter(data_source):  
    process(data)
```

```
x = [1, 2, 3]
for elem in x:
    ...
```



生成器(generator)

Regular Functions

Return a single, computed value

Each call generates a new private namespace and new local variables, then variables are thrown away

Generators

Return an iterator that generates a stream of values

Local variables aren't thrown away when exiting a function - you can resume where you left off!

一个简单的生成器

```
def generate_ints(n):  
    for i in range(n):  
        yield i
```

关键字yield告诉
Python将函数变为生
成器

```
g = generate_ints(3)  
type(g) # => <class 'generator'>  
next(g) # => 0  
next(g) # => 1  
next(g) # => 2  
next(g) # raises StopIteration
```


一个简单的生成器

```
def generate_ints(n):  
    for i in range(n):  
        a = yield i  
        if a == 100:  
            print('a is 100')  
a = generate_ints(3)  
print(next(a))  
print(a.send(100))  
print(next(a))
```

生成器允许在执行时，
由外部传入值，影响
生成器执行结果

```
>>>  
0  
1  
a is 100  
2
```

生成器

```
def generate_fibs():  
    a, b = 0, 1  
    while True:  
        a, b = b, a + b  
        yield a
```

Fibonacci数列的无限
数据流

```
g = generate_fibs()
```

```
next(g) # => 1
```

```
next(g) # => 1
```

```
next(g) # => 2
```

```
next(g) # => 3
```

```
next(g) # => 5
```

```
max(g) # Oh no! what  
happens?
```

生成器表达式(**generator expression**)

- “惰性的” 列表推导式

(**expensive_function**(data)
for data in iterable)

生成器表达式

```
needle in  
(expensive_fn(item)  
    for item in haystack)
```

```
needle in  
[expensive_fn(item)  
    for item in haystack]
```

有什么区别？

生成器表达式

```
g = (x*2 for x in range(10))  
type(g)                # => <class 'generator'>
```

```
l = [x*2 for x in range(10)]  
type(l)                # => <class 'list'>
```

函数

函数的定义

- 关键字def用于定义一个新函数

```
def fn_name(param1, param2):  
    value = do_something()  
    return value
```

函数的定义

微实例5.1：生日歌。

过生日时要为朋友唱生日歌，歌词为：

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear <名字>

Happy birthday to you!

编写程序为Mike和Lily输出生日歌。

函数的定义

- 最简单的实现方法是重复使用print()语句，如下：

```
print("Happy birthday to you!")  
print("Happy birthday to you!")  
print("Happy birthday, dear Mike!")  
print("Happy birthday to you!")
```

函数的定义

```
# m5.1HappyBirthday.py
```

```
def happy():  
    print("Happy birthday to you!")
```

```
def happyB(name):  
    happy()  
    happy()  
    print("Happy birthday, dear {}".format(name))  
    happy()
```

```
happyB("Mike")  
print()  
happyB("Lily")
```

函数的定义

>>>

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Mike!

Happy birthday to you!

Happy birthday to you!

Happy birthday to you!

Happy birthday, dear Lily!

Happy birthday to you!

函数调用的过程

- 程序调用一个函数需要执行以下四个步骤：
 1. 调用程序在调用处暂停执行；
 2. 在调用时将实参复制给函数的形参；
 3. 执行函数体语句；
 4. 函数调用结束给出返回值，程序回到调用前的暂停处继续执行。

函数调用的过程

`name="Mike"`

```
happyB("Mike")  —→  def happyB(name):  
print()           happy()  
happyB("Lily")    happy()  
                  print("Happy birthday, dear!".format(name))  
                  happy()
```

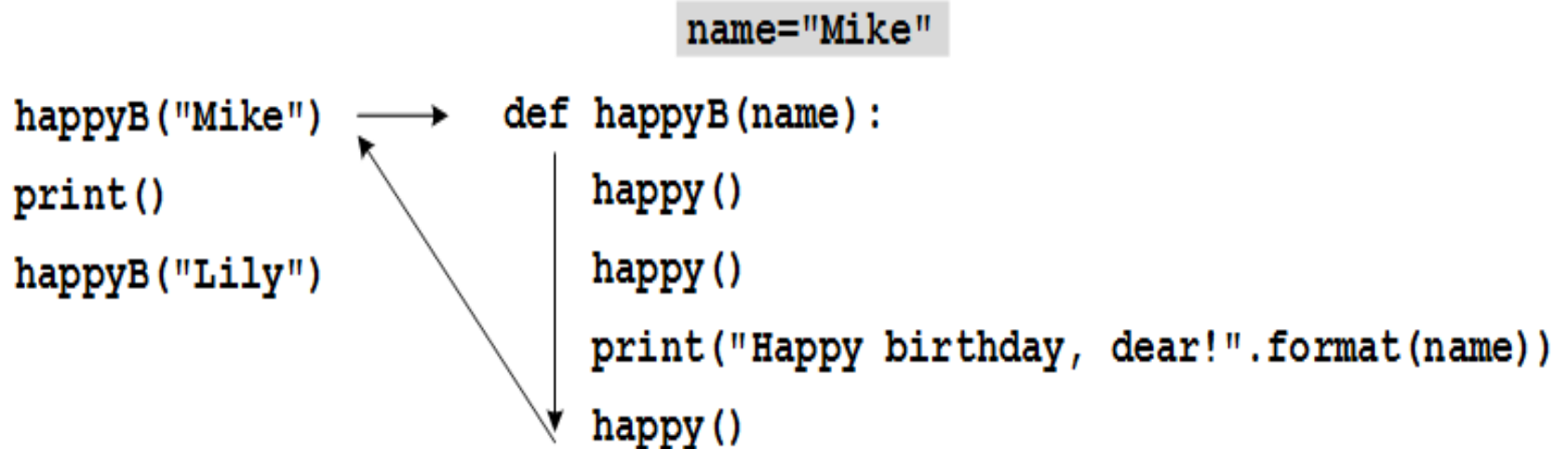
微实例5.1中happyB()的被调用过程

函数调用的过程

`name="Mike"`

```
happyB("Mike")  →  def happyB(name):  
print()          happy() → def happy():  
happyB("Lily")   happy() ↘ print("Happy birthday to you!")  
                  print("Happy birthday, dear!".format(name))  
                  happy()
```

函数调用的过程



函数返回值

- 所有函数均有返回值
- 没有return语句的函数会返回None
- 可以通过元组来返回多个值
`return value1, value2, value3`

参数传递方式

Pass-By-Value or Pass-By-Reference?

- 变量被复制到函数的本地符号表
 - 但变量仅仅是对象的引用
- 所以我们可以认为是pass-by-object-reference
 - 如果可变对象被传递进函数，那么调用者可以看到变量变化

例子

```
def f1(a):  
    a = 0  
    print("a = ", a)  
  
b = 100  
f1(b)  
print("b = ", b)
```

```
>>>  
a = 0  
b = 100
```

```
def f2(a, b):  
    a = 2  
    b[0] = 'changed'  
  
x = 1  
L = [1, 2]  
f2(x, L)  
print("x = ", x, "L = ", L)
```

```
>>>  
x = 1 L = ['changed', 2]
```

参数的匹配与解包

参数的匹配

- 默认情况下，按照位置顺序进行匹配

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

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

```
f(a=1, c=3, b=2)          >>> 1 2 3
```

默认参数值

- 我们可以为一个或者多个参数指定默认值
 - 调用函数时可以传递比函数定义时更少的参数
- 有什么好处？
 - 为函数展示了一个更加简便的接口
 - 给参数提供合理的默认值

默认参数值

required parameter prompt

```
def ask_yn(prompt,  
           retries=4,  
           complaint='Enter Y/N' ):
```

optional parameter retries
defaults to 4

optional parameter complaint
defaults to 'Enter Y/N'

默认参数值

```
def ask_yn(prompt, retries=4, complaint='Enter Y/N!'):  
    for i in range(retries):  
        ok = input(prompt)  
        if ok == 'Y':  
            return True  
        if ok == 'N':  
            return False  
        print(complaint)  
    return False
```


函数调用

`ask_yn(prompt, retries=4, complaint='...')`

Call with only the mandatory argument

```
ask_yn('Really quit?')
```

Call with one keyword argument

```
ask_yn('OK to overwrite the file?', retries=2)
```

Call with one keyword argument - in any order!

```
ask_yn('Update status?', complaint='Just Y/N')
```

Call with all of the keyword arguments

```
ask_yn('Send text?', retries=2, complaint='Y/N please!')
```

可变位置参数

- 形式如*args的参数可以接收任意数量的值，并将这些值封装为一个元组

```
# product accepts any number of arguments:  
def product(*nums, scale=1):  
    p = scale  
    for n in nums:  
        p *= n  
    return p
```

解包可变位置参数

```
# Suppose we want to find 2 * 3 * 5 * 7 * ... up to 100  
def is_prime(n):    pass    # Some implementation  
  
# Extract all the primes  
primes = [number for number in range(2, 100)  
           if is_prime(number)]  
  
# primes == [2, 3, 5, ...]  
print(product(*primes)) # equiv. to product(2, 3, 5, ...)
```

the syntax `*seq` unpacks a sequence
into its constituent components

可变关键字参数

- 形如**kwargs的参数可以接收任意数量的关键字实参值，并封装为一个字典kwargs

```
authorize(  
    "If music be the food of love, play on.",  
    playwright="Shakespeare",  
    act=1,  
    scene=1,  
    speaker="Duke Orsino"  
)
```

```
# > If music be the food of love, play on.  
# -----  
# act: 1  
# scene: 1  
# speaker: Duke Orsino  
# playwright: Shakespeare
```

可变关键字参数

```
def authorize(quote, **speaker_info):  
    print(">", quote)  
    print("-" * (len(quote) + 2))  
    for k, v in speaker_info.items():  
        print(k, v, sep=' : ')
```

```
speaker_info = {  
    'act': 1,  
    'scene': 1,  
    'speaker': "Duke Orsino",  
    'playwright': "Shakespeare"  
}
```

解包可变关键字参数

```
info = {  
    'sonnet': 18,  
    'line': 1,  
    'author': "Shakespeare"  
}  
authorize("Shall I compare thee to a summer's day", **info)  
  
# > Shall I compare thee to a summer's day  
# -----  
# line: 1  
# sonnet: 18  
# author: Shakespeare
```

例子：格式化字符串

all positional arguments
go into args

```
str.format(*args, **kwargs)
```

all keyword arguments
go into kwargs

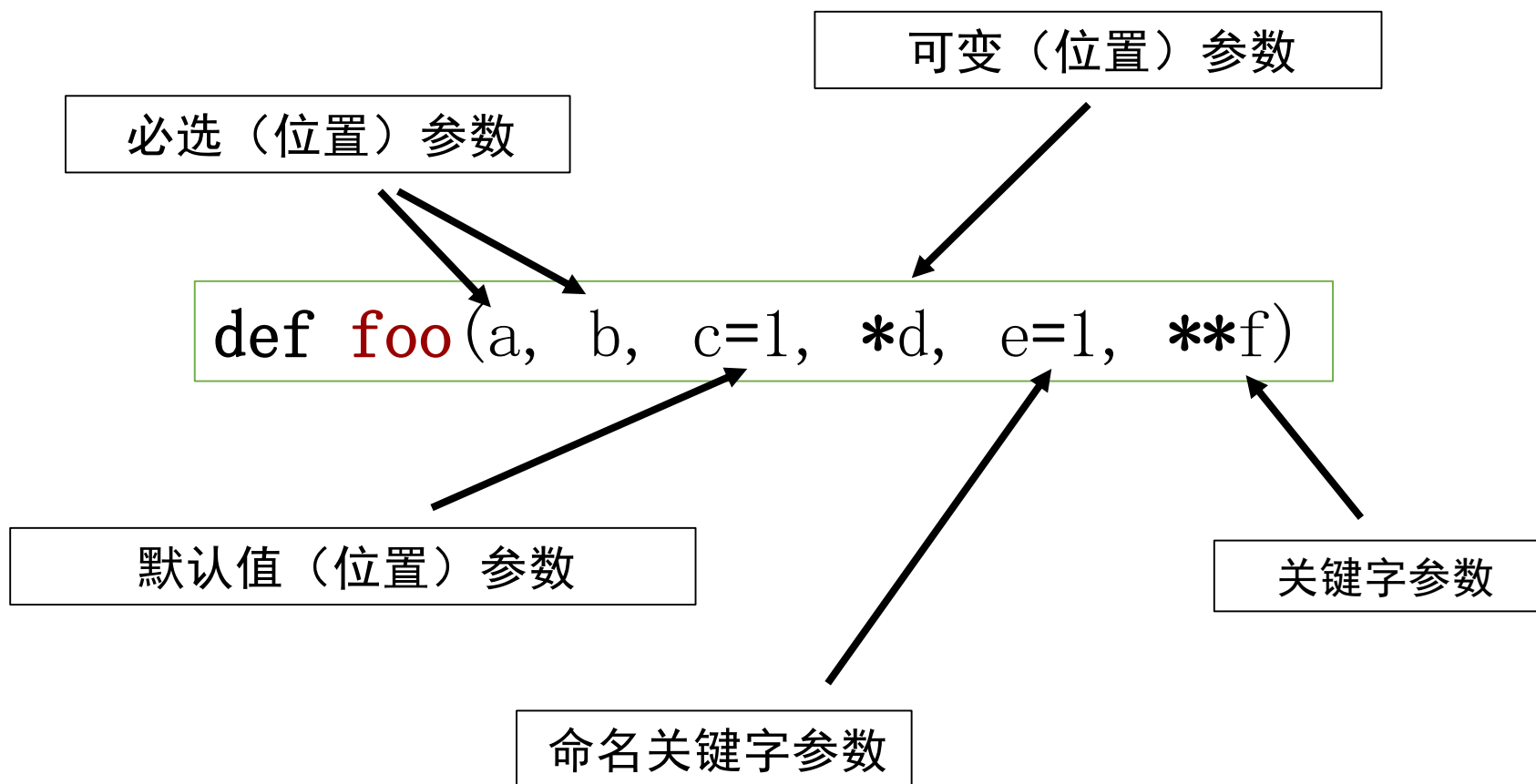
例子：格式化字符串

```
"{0} {b} {1} {a} {0} {2}".format(  
    5, 8, 9, a='z', b='x'  
)
```

=> 5x8z59

```
args = (5, 8, 9)  
kwargs = {'a': 'z', 'b': 'x'}
```


一个有效的函数定义



例子:

```
def fe(a, b, c=0, *args, e, **kwargs):  
    print("a = ", a, "b = ", b, "c = ", c,  
          "args = ", args, "e = ", e, "kwargs = ", kwargs)
```

```
fe(1, 2, 3, 4, 5, 6, e=7, f=8, g=9)
```

```
fe(1, 2, e=1, f=8)
```

例子:

```
def fe(a, b, c=0, *args, e, **kwargs):  
    print("a = ", a, "b = ", b, "c = ", c,  
          "args = ", args, "e = ", e, "kwargs = ", kwargs)
```

```
fe(1, 2, 3, 4, 5, 6, e=7, f=8, g=9)
```

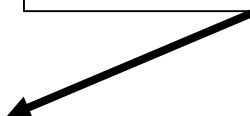
```
>>>  
a = 1 b = 2 c = 3 args = (4, 5, 6) e = 7 kwargs = {'f': 8, 'g': 9}
```

```
fe(1, 2, e=1, f=8)
```

```
>>>  
a = 1 b = 2 c = 0 args = () e = 1 kwargs = {'f': 8}
```

例子：

*作为特殊分隔符

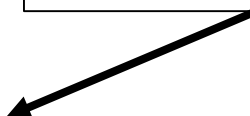


```
def fe(a, b = 0, *, c, d):  
    print("a = ", a, "b = ", b, "c = ", c,  
          "d = ", d)
```

```
fe(1, 2, c = 3, d = 4)
```

例子:

*作为特殊分隔符



```
def fe(a, b = 0, *, c, d):  
    print("a = ", a, "b = ", b, "c = ", c,  
          "d = ", d)
```

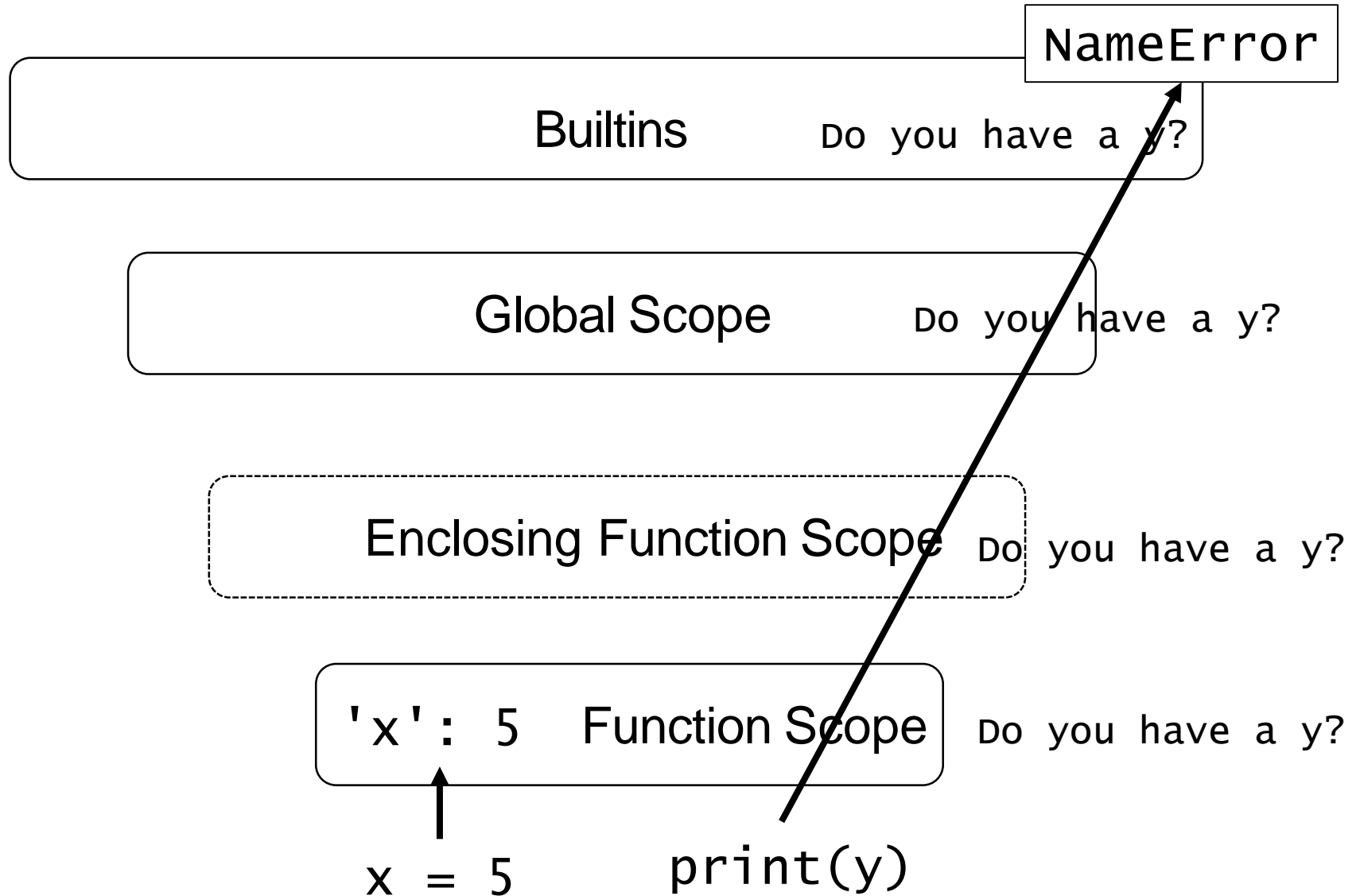
```
fe(1, 2, c = 3, d = 4)
```

```
>>>  
a = 1 b = 2 c = 3 d = 4
```

函数作用域

函数作用域

- 函数执行引入新的本地符号表(local symbol table)
 - 回顾行李标签和手提箱的比喻：函数执行引入新的行李区域
- 变量赋值： $x = 5$
 - 在本地符号表添加条目
- 变量引用： `print(y)`
 - 首先，查找本地符号表
 - 其次，查找全局(上一层)符号表
 - 最后，查找内置符号(`print`, `input`, etc)



本地作用域

```
x = 2
def foo(y):
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)
```

```
foo(3)
```

```
# print {'y': 3, 'z': 5}
# print 2
# print 2, 3, 5
```

本地作用域

```
x = 2
def foo(y):
    x = 41
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)
```

添加'x': 41到本地符号表，本地变量名'x'覆盖全局变量名'x'

```
foo(3)
```

```
# print { 'x': 41, 'y': 3, 'z': 5 }
# print 2
# print 41, 3, 5
```

global关键字

- 在函数内部试图改变全局变量的值，需使用global关键字

```
x = 0
def foo():
    global x
    x = 100
```

```
foo()
print(x)
```

```
>>> 100
```

内嵌作用域

- Python中可以嵌套使用def关键字定义函数
- 调用外层函数时，运行到内层def语句仅完成对内层函数定义

```
x = 0
def f1():
    x = 100
    def f2():
        print(x)
```

```
f1()
```

```
>>>
```

```
x = 0
def f1():
    x = 100
    def f2():
        print(x)
    f2()
```

```
f1()
```

```
>>> 100
```

闭包

- 嵌套作用域在嵌套的函数返回后仍然有效

```
def make_averager():  
    series = []  
  
    def averager(new_value):  
        series.append(new_value)  
        total = sum(series)  
        return total/len(series)  
  
    return averager  
  
avg = make_averager()
```

闭包 →

自由变量 →

闭包

```
def make_averager() :  
    series = []  
  
    def averager(new_value) :  
        series.append(new_value)  
        total = sum(series)  
        return total/len(series)  
  
    return averager  
  
avg = make_averager()
```

```
>>> avg(10)  
10.0
```

```
>>> avg(11)  
10.5
```

```
>>> avg(12)  
11.0
```

nonlocal关键字

- 内嵌函数内部对嵌套作用域中的变量进行修改，需使用nonlocal关键字

```
def make_averager() :  
    count = 0  
    total = 0  
  
    def averager(new_value) :  
        count += 1  
        total += new_value  
        return total / count  
  
    return averager  
  
avg = make_averager()
```

```
>>> avg(10)
```

```
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
```

```
    File "<input>", line 6, in averager
```

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



```
def make_averager() :  
    count = 0  
    total = 0  
  
    def averager(new_value) :  
        nonlocal count, total  
        count += 1  
        total += new_value  
        return total / count  
  
    return averager  
  
avg = make_averager()
```

例子

```
def test(num):  
    in_num = num  
    def nested(label):  
        nonlocal in_num  
        in_num += 1  
        print(label, in_num)  
    return nested
```

不同内嵌
函数副本
的内嵌变
量彼此独
立

```
f1 = test(0)  
f1('a')  
f1('b')  
f1('c')  
  
f2 = test(100)  
f2('abc')
```

```
>>>  
a 1  
b 2  
c 3  
abc 101
```

函数作为对象

把函数视作对象

- 在Python中，函数是一等对象
 - 在运行时创建
 - 能赋值给变量或数据结构中的元素
 - 能作为参数传给函数
 - 能作为函数的返回结果

把函数视作对象

```
def echo(arg): return arg

type(echo)    # <class 'function'>
hex(id(echo))  # 0x1003c2bf8
print(echo)    # <function echo at 0x1003c2bf8>

foo = echo
hex(id(foo))   # 0x1003c2bf8
print(foo)     # <function echo at 0x1003c2bf8>
```

高阶函数

- 函数名也是变量
- 一个函数可以接收另一个函数作为参数，这种函数就称之为高阶函数

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

```
add(-5, 6, abs)
```

```
x = -5  
y = 6  
f = abs  
f(x) + f(y) ==> abs(-5) +  
abs(6) ==> 11  
return 11
```

Map/Filter

common pattern

```
output = []  
for element in iterable:  
    val = function(element)  
    output.append(val)  
return output
```

```
[function(element)  
    for element in iterable]
```



```
[len(s) for s in languages]  
["python", "perl", "java", "c++"]
```



```
[ 6      4      4      3 ]
```

```
[len(s) for s in languages]
["python", "perl", "java", "c++"]
```



len



```
[ 6      4      4      3 ]
```

```
[len(s) for s in languages]  
["python", "perl", "java", "c++"]
```



len



```
[ 6      4      4      3 ]
```

```
[len(s) for s in languages]  
["python", "perl", "java", "c++"]
```



```
[ 6         4         4         3 ]
```

apply some function to every
element of a sequence

```
map(fn, iter)
```

```
[len(s) for s in languages]  
["python", "perl", "java", "c++"]
```



map(len, languages)



<

6

4

4

3

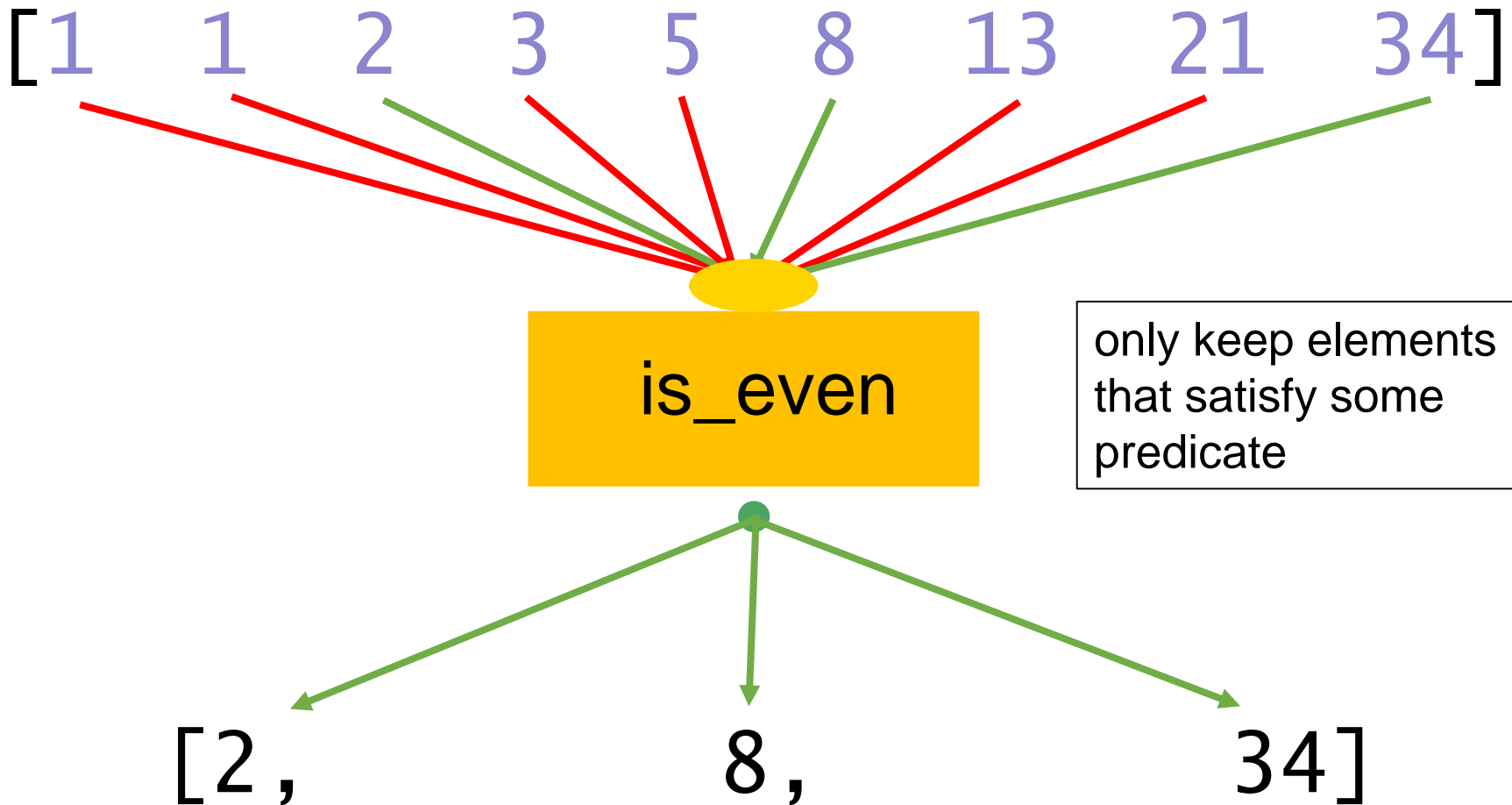
>

another common pattern

```
output = []  
for element in iterable:  
    if predicate(element):  
        output.append(element)  
return output
```

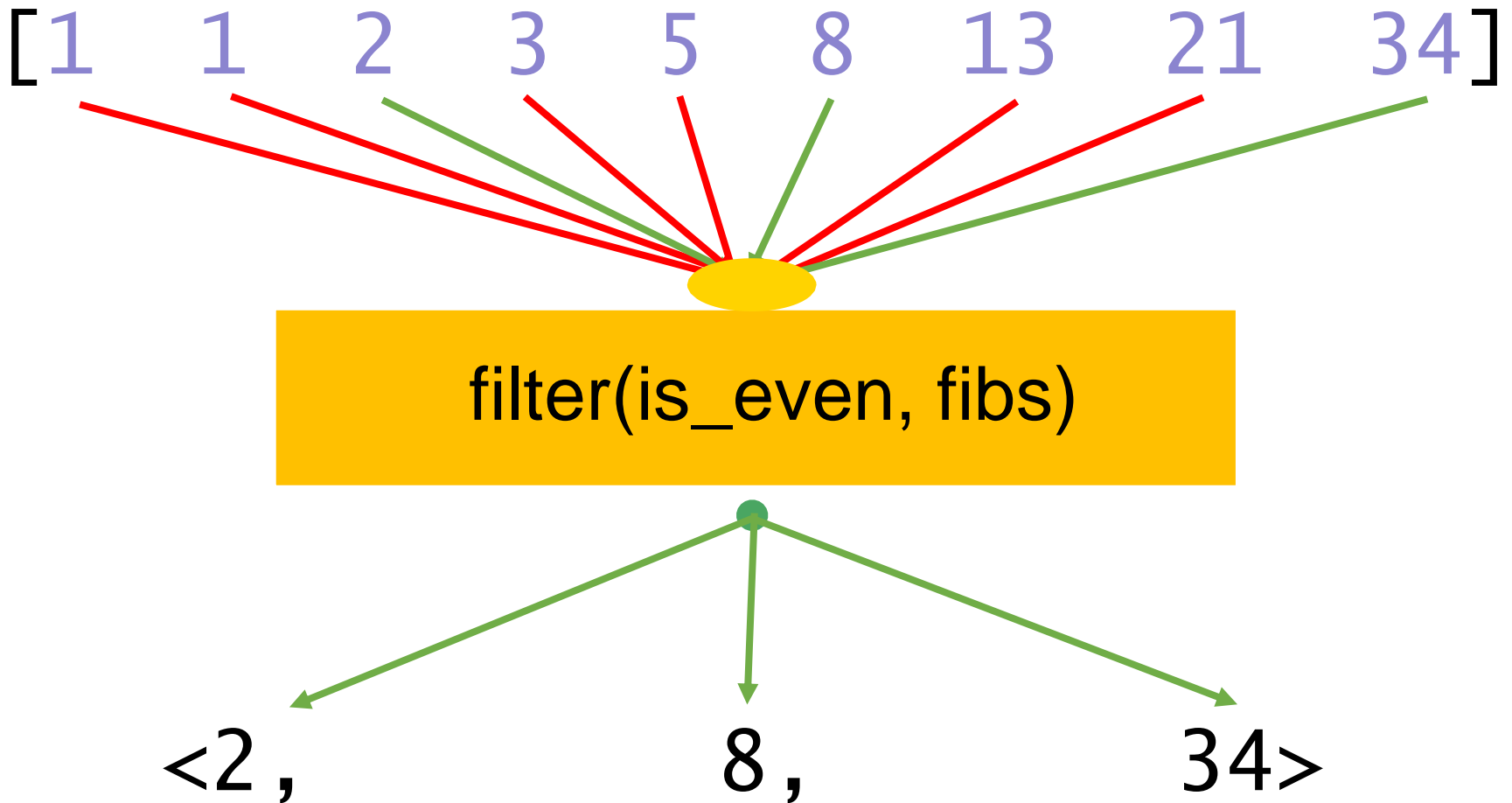
```
[element for element in iterable  
    if predicate(element)]
```

```
[num for num in fibs if is_even(num)]
```




```
filter(pred, iter)
```

```
[num for num in fibs if is_even(num)]
```



例子

给定如下的英文名

```
names = ["aARon", "tom", "abEl", "PARIS"]
```

使用`map`函数将名字规范化（首字母大写，其他字母小写）

使用`map`函数对以下两个数组中对应位置的数进行求和

```
arr1 = [1, 5, 7, 3, 9]
```

```
arr2 = [8, 4, 6, 2, 10]
```

使用`filter`函数筛选得出1-100中的所有素数

结合使用`filter`函数和`map`函数将第一题中的所有长度大于4的英文名规范化

例子

给定如下的英文名

```
names = ["aARon", "tom", "abEl", "PARIS"]
```

使用map函数将名字规范化（首字母大写，其他字母小写）

```
def normalize(name):  
    return name.title()  
  
rets = map(normalize, names)  
print(list(rets))
```

例子

```
# 使用map函数对以下两个数组中对应位置的数进行求和  
arr1 = [1, 5, 7, 3, 9]  
arr2 = [8, 4, 6, 2, 10]
```

```
rets = map(sum, zip(arr1, arr2))  
print(list(rets))
```

例子

使用`filter`函数筛选得出1-100中的所有素数

```
def is_prime(num):  
    if num == 1:  
        return False  
    if num == 2:  
        return True  
    for i in range(2, num-1):  
        if num % i == 0:  
            return False  
    return True  
  
rets = filter(is_prime, range(1, 101))  
print(list(rets))
```

例子

结合使用`filter`函数和`map`函数将第一题中的所有长度大于4的英文名规范化

```
def is_longer_than_4(name):  
    return len(name) > 4  
  
rets = map(normalize, filter(is_longer_than_4, names))  
print(list(rets))
```

Lambda函数

lambda函数

关键字lambda创建
匿名函数

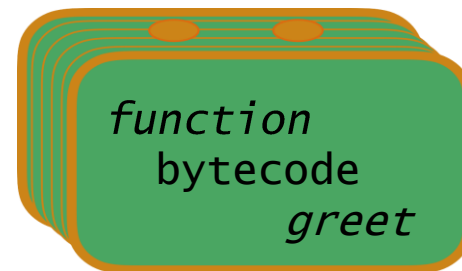
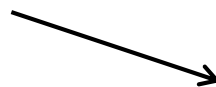
```
lambda params: expr(params)
```

返回表达式

定义函数 vs lambda函数

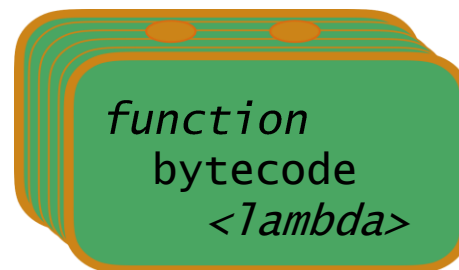
```
def greet():  
    print("Hi!")
```

greet



def将一个
名字与函数
绑定

```
lambda val: val ** 2  
lambda x, y: x * y  
lambda pair: pair[0] * pair[1]
```



lambda仅仅
创建一个函
数对象

```
(lambda x: x > 3)(4) # => True
```

例子

*# Squares from 0**2 to 9**2*

```
map(lambda val: val ** 2, range(10))
```

*# Tuples with positive second elements,
[(4, 1), (3, -2), (8, 0)]*

```
filter(lambda pair: pair[1] > 0, [(4, 1), (3, -2), (8, 0)])
```

装饰器

函数作为参数

```
# map(fn, iterable)
```

```
# filter(pred, iterable)
```

```
def perform_twice(fn, *args, **kwargs):  
    fn(*args, **kwargs)  
    fn(*args, **kwargs)
```

```
perform_twice(print, 5, 10, sep='&', end='...')  
# => 5&10... 5&10...
```

函数作为返回值

```
def make_divisibility_test(n):  
    def divisible_by_n(m):  
        return m % n == 0  
    return divisible_by_n
```

```
div_by_3 = make_divisibility_test(3)  
filter(div_by_3, range(10)) # generates 0, 3,  
6, 9  
make_divisibility_test(5)(10) # => True
```

装饰器(decorator)

- 装饰器是返回函数的高阶函数，在不改变被装饰函数的定义的情况下增强其功能

装饰器例子

```
def log(func):  
    def wrapper(*args, **kwargs):  
        print("Arguments:", args, kwargs)  
        return func(*args, **kwargs)  
    return wrapper
```


装饰器例子

```
def foo(a, b, c=1):  
    return (a + b) * c
```

```
foo = log(foo)
```

使用了两次foo

```
foo(2, 3)
```

```
>>> Arguments: (2, 3) {}  
# => returns 5
```

```
foo(2, 1, c=3)
```

```
>>> Arguments: (2, 1) {'c': 3}  
# => returns 9
```

装饰器例子

@log

```
def foo(a, b, c=1):  
    return (a + b) * c
```

```
foo(5, 3, c=2)
```

```
>>> Arguments: (5, 3) {'c': 2}  
# => returns 16
```

@decorator 将
装饰器应用于紧
接着的函数

带参数的装饰器

```
def log(text):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            print(text, ":", args, kwargs)  
            return func(*args, **kwargs)  
        return wrapper  
    return decorator  
  
def foo(a, b, c=1):  
    return (a + b) * c
```

```
foo = log("All arguments") (foo)
```

```
print(foo(2, 3))
```

```
print(foo(2, 1, c=3))
```

```
>>> All arguments: (2, 3) {}
```

```
# => returns 5
```

```
>>> ALL arguments: (2, 1) {'c': 3}
```

```
# => returns 9
```

```
@log("All arguments")  
def foo(a, b, c=1):  
    return (a + b) * c
```

累加装饰器

```
def bread(func):  
    def wrapper():  
        print("</'''''\>")  
        func()  
        print("<\_____/>")  
    return wrapper
```

```
def ingredients(func):  
    def wrapper():  
        print("#tomatoes#")  
        func()  
        print("~salad~")  
    return wrapper
```

```
def sandwich(food="--ham--"):  
    print(food)
```

```
sandwich()
```

```
>>>  
--ham--
```

```
sandwich = bread(ingredients(sandwich))  
sandwich()
```

```
>>>  
</' ' ' ' ' '\>  
#tomatoes#  
--ham--  
~salad~  
<\_____/>
```

```
@bread  
@ingredients  
def sandwich(food="--ham--"):  
    print(food)
```