Table of Contents

慕课网**python-python**进阶

简介：

Python基础分《Python入门》和《Python进阶》两门课程，《Python进阶》是第二门课程，学习该课程前，请先学习《Python入门》,效果会更好。《Python进阶》课程详细介绍Python强大的函数式编程和面向对象编程，掌握Python高级程序设计的方法。

课程须知:

本课程是Python入门的后续课程

1. 掌握Python编程的基础知识
2. 掌握Python函数的编写
3. 对面向对象编程有所了解更佳

老师告诉你能学到什么？

1. 什么是函数式编程
2. Python的函数式编程特点
3. Python的模块
4. Python面向对象编程
5. Python强大的定制类

第**1**章 课程介绍

本课程是Python开发入门课程的进阶篇，将详细讲解Python函数式编程和面向对象编程的概念，通过练习掌握函数式编程和面向对象编程的方法。

1-1 课程介绍(mp4)

课程介绍.mp4

第**2**章 函数式编程

本章讲解Python函数式编程概念，高阶函数的概念和实际用法，以及装饰器函数的原理和实现方式。

2-1 课程介绍(mp4)

python中函数式编程简介.mp4

2-2 python中高阶函数(mp4)

python中高阶函数.mp4

2-3 python把函数作为参数

在2.1小节中，我们讲了高阶函数的概念，并编写了一个简单的高阶函数：

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

如果传入abs作为参数f的值：

```
add(-5, 9, abs)
```

根据函数的定义，函数执行的代码实际上是：

```
abs(-5) + abs(9)
```

由于参数 x, y 和 f 都可以任意传入，如果 f 传入其他函数，就可以得到不同的返回值。

任务

利用add(x,y,f)函数，计算：

$\sqrt{x} + \sqrt{y}$

?不会了怎么办 计算平方根可以用函数：

```
>>> math.sqrt(2)
1.4142...
```

参考代码:

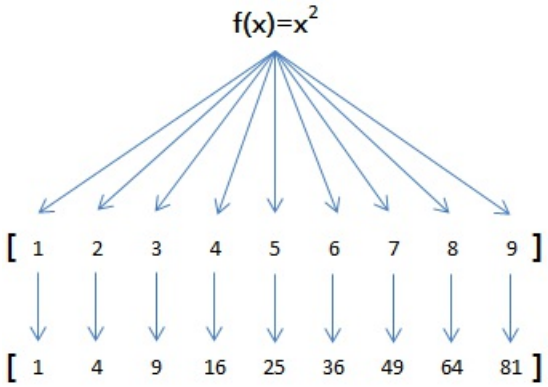
```
import math
def add(x, y, f):
    return f(x) + f(y)
print add(25, 9, math.sqrt)
```

2-4 python中map()函数

map()是 Python 内置的高阶函数，它接收一个函数 f 和一个 list，并通过把函数 f 依次作用在 list 的每个元素上，得到一个新的 list 并返回。

例如，对于list [1, 2, 3, 4, 5, 6, 7, 8, 9]

如果希望把list的每个元素都作平方，就可以用map()函数：



因此，我们只需要传入函数f(x)=x*x，就可以利用map()函数完成这个计算：

```
def f(x):
    return x*x
print map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

输出结果：

```
[1, 4, 9, 10, 25, 36, 49, 64, 81]
```

注意：map()函数不改变原有的 list，而是返回一个新的 list。

利用map()函数，可以把一个 list 转换为另一个 list，只需要传入转换函数。

由于list包含的元素可以是任何类型，因此，map() 不仅仅可以处理只包含数值的 list，事实上它可以处理包含任意类型的 list，只要传入的函数f可以处理这种数据类型。

任务

假设用户输入的英文名字不规范，没有按照首字母大写，后续字母小写的规则，请利用map()函数，把一个list（包含若干不规范的英文名字）变成一个包含规范英文名字的list：

输入：['adam', 'LISA', 'barT'] 输出：['Adam', 'Lisa', 'Bart']

?不会了怎么办 format_name(s)函数接收一个字符串，并且要返回格式化后的字符串，利用map()函数，就可以输出新的 list。

参考代码:

```
def format_name(s):
    return s[0].upper() + s[1:].lower()
print map(format_name, ['adam', 'LISA', 'barT'])
```

2-5 python中reduce()函数

reduce()函数也是Python内置的一个高阶函数。reduce()函数接收的参数和 map()类似，一个函数 f，一个list，但行为和map()不同，reduce()传入的函数 f 必须接收两个参数，reduce()对list的每个元素反复调用函数f，并返回最终结果值。

例如，编写一个f函数，接收x和y，返回x和y的和：

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

调用 reduce(f, [1, 3, 5, 7, 9])时，reduce函数将做如下计算：

```
先计算头两个元素：f(1, 3)，结果为4；
再把结果和第3个元素计算：f(4, 5)，结果为9；
再把结果和第4个元素计算：f(9, 7)，结果为16；
再把结果和第5个元素计算：f(16, 9)，结果为25；
由于没有更多的元素了，计算结束，返回结果25。
```

上述计算实际上是对 list 的所有元素求和。虽然Python内置了求和函数sum()，但是，利用reduce()求和也很简单。

reduce()还可以接收第3个可选参数，作为计算的初始值。如果把初始值设为100，计算：

```
reduce(f, [1, 3, 5, 7, 9], 100)
```

结果将变为125，因为第一轮计算是：

计算初始值和第一个元素：f(100, 1)，结果为101。

任务

Python内置了求和函数sum()，但没有求积的函数，请利用recude()来求积：

输入：[2, 4, 5, 7, 12] 输出：2*4*5*7*12的结果

?不会了怎么办 reduce()接收的函数f需要两个参数，并返回一个结果，以便继续进行下一轮计算。

参考代码:

```
def prod(x, y):
    return x * y
print reduce(prod, [2, 4, 5, 7, 12])
```

2-6 python中filter()函数

filter()函数是 Python 内置的另一个有用的高阶函数，filter()函数接收一个函数 f 和一个list，这个函数 f 的作用是对每个元素进行判断，返回 True或 False，filter()根据判断结果自动过滤掉不符合条件的元素，返回由符合条件元素组成的新list。

例如，要从一个list [1, 4, 6, 7, 9, 12, 17]中删除偶数，保留奇数，首先，要编写一个判断奇数的函数：

```
def is_odd(x):
    return x % 2 == 1
```

然后，利用filter()过滤掉偶数：

```
filter(is_odd, [1, 4, 6, 7, 9, 12, 17])
```

结果：[1, 7, 9, 17]

利用filter()，可以完成很多有用的功能，例如，删除 None 或者空字符串：

```
def is_not_empty(s):
    return s and len(s.strip()) > 0
filter(is_not_empty, ['test', None, '', 'str', '  ', 'END'])
```

结果：['test', 'str', 'END']

注意: s.strip(rm) 删除 s 字符串中开头、结尾处的 rm 序列的字符。

当rm为空时，默认删除空白符（包括'\n', '\r', '\t', ' ')，如下：

```
a = '     123'
a.strip()
```

结果： '123'

```
a='\t\t123\r\n'
a.strip()
```

结果：'123'

任务 请利用filter()过滤出1~100中平方根是整数的数，即结果应该是：

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

?不会了怎么办 filter() 接收的函数必须判断出一个数的平方根是否是整数，而 math.sqrt()返回结果是浮点数。

参考代码:

```
import math
def is_sqr(x):
    r = int(math.sqrt(x))
    return r*r==x
print filter(is_sqr, range(1, 101))
```

2-7 python中自定义排序函数

Python内置的 sorted()函数可对list进行排序：

```
>>>sorted([36, 5, 12, 9, 21])

[5, 9, 12, 21, 36]
```

但 sorted()也是一个高阶函数，它可以接收一个比较函数来实现自定义排序，比较函数的定义是，传入两个待比较的元素 x, y，如果 x 应该排在 y 的前面，返回 -1，如果 x 应该排在 y 的后面，返回 1。如果 x 和 y 相等，返回 0。

因此，如果我们要实现倒序排序，只需要编写一个reversed_cmp函数：

```
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0
```

这样，调用 sorted() 并传入 reversed_cmp 就可以实现倒序排序：

```
>>> sorted([36, 5, 12, 9, 21], reversed_cmp)
[36, 21, 12, 9, 5]
```

sorted()也可以对字符串进行排序，字符串默认按照ASCII大小来比较：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])
['Credit', 'Zoo', 'about', 'bob']
```

'Zoo'排在'about'之前是因为'Z'的ASCII码比'a'小。

任务

对字符串排序时，有时候忽略大小写排序更符合习惯。请利用sorted()高阶函数，实现忽略大小写排序的算法。

输入：['bob', 'about', 'Zoo', 'Credit'] 输出：['about', 'bob', 'Credit', 'Zoo']

?不会了怎么办 对于比较函数cmp_ignore_case(s1, s2)，要忽略大小写比较，就是先把两个字符串都变成大写（或者都变成小写），再比较。

参考代码:

```
def cmp_ignore_case(s1, s2):
    u1 = s1.upper()
    u2 = s2.upper()
    if u1 < u2:
        return -1
    if u1 > u2:
        return 1
    return 0
print sorted(['bob', 'about', 'Zoo', 'Credit'], cmp_ignore_case)
```

2-8 python中返回函数

Python的函数不但可以返回int、str、list、dict等数据类型，还可以返回函数！

例如，定义一个函数 f()，我们让它返回一个函数 g，可以这样写：

```
def f():
    print 'call f()...'
    # 定义函数g:
    def g():
        print 'call g()...'
    # 返回函数g:
    return g
```

仔细观察上面的函数定义，我们在函数 f 内部又定义了一个函数 g。由于函数 g 也是一个对象，函数名 g 就是指向函数 g 的变量，所以，最外层函数 f 可以返回变量 g，也就是函数 g 本身。

调用函数 f，我们会得到 f 返回的一个函数：

```
>>> x = f()   # 调用f()
call f()...
>>> x   # 变量x是f()返回的函数：
<function g at 0x1037bf320>
>>> x()   # x指向函数，因此可以调用
call g()...   # 调用x()就是执行g()函数定义的代码
```

请注意区分返回函数和返回值：

```
def myabs():
    return abs   # 返回函数
def myabs2(x):
    return abs(x)   # 返回函数调用的结果，返回值是一个数值
```

返回函数可以把一些计算延迟执行。例如，如果定义一个普通的求和函数：

```
def calc_sum(lst):
    return sum(lst)
```

调用calc_sum()函数时，将立刻计算并得到结果：

```
>>> calc_sum([1, 2, 3, 4])
10
```

但是，如果返回一个函数，就可以“延迟计算”：

```
def calc_sum(lst):
    def lazy_sum():
        return sum(lst)
    return lazy_sum
```

`#` 调用calc_sum()并没有计算出结果，而是返回函数:

```
>>> f = calc_sum([1, 2, 3, 4])
>>> f
```

```
<function lazy_sum at 0x1037bfaa0>
```

`#` 对返回的函数进行调用时，才计算出结果:

```
>>> f()
10
```

由于可以返回函数，我们在后续代码里就可以决定到底要不要调用该函数。

任务 请编写一个函数calc_prod(lst)，它接收一个list，返回一个函数，返回函数可以计算参数的乘积。

?不会了怎么办 先定义能计算乘积的函数，再将此函数返回。

参考代码:

```
def calc_prod(lst):
    def lazy_prod():
        def f(x, y):
            return x * y
        return reduce(f, lst, 1)
    return lazy_prod
f = calc_prod([1, 2, 3, 4])
print f()
```

2-9 python中闭包

在函数内部定义的函数和外部定义的函数是一样的，只是他们无法被外部访问：

```
def g():
    print 'g()...'

def f():
    print 'f()...'
    return g
```

将 g 的定义移入函数 f 内部，防止其他代码调用 g：

```
def f():
    print 'f()...'
    def g():
        print 'g()...'
    return g
```

但是，考察上一小节定义的 calc_sum 函数：

```
def calc_sum(lst):
    def lazy_sum():
        return sum(lst)
    return lazy_sum
```

注意: 发现没法把 lazy_sum 移到 calc_sum 的外部，因为它引用了 calc_sum 的参数 lst。

像这种内层函数引用了外层函数的变量（参数也算变量），然后返回内层函数的情况，称为闭包（Closure）。

闭包的特点是返回的函数还引用了外层函数的局部变量，所以，要正确使用闭包，就要确保引用的局部变量在函数返回后不能变。举例如下：

希望一次返回3个函数，分别计算1x1,2x2,3x3:

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果全部都是 9（请自己动手验证）。

原因就是当count()函数返回了3个函数时，这3个函数所引用的变量 i 的值已经变成了3。由于f1、f2、f3并没有被调用，所以，此时他们并未计算 i*i，当 f1 被调用时：

```
>>> f1()
9     # 因为f1现在才计算i*i，但现在i的值已经变为3
```

因此，返回函数不要引用任何循环变量，或者后续会发生变化的变量。

任务

返回闭包不能引用循环变量，请改写count()函数，让它正确返回能计算1x1、2x2、3x3的函数。

?不会了怎么办 考察下面的函数 f:

```
def f(j):
    def g():
        return j*j
    return g
```

它可以正确地返回一个闭包g，g所引用的变量j不是循环变量，因此将正常执行。

在count函数的循环内部，如果借助f函数，就可以避免引用循环变量i。

参考代码:

```
def count():
    fs = []
    for i in range(1, 4):
        def f(j):
            def g():
                return j*j
            return g
        r = f(i)
        fs.append(r)
    return fs
f1, f2, f3 = count()
print f1(), f2(), f3()
```

2-10 python中匿名函数

高阶函数可以接收函数做参数，有些时候，我们不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算 f(x)=x2 时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

```
>>> map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 lambda x: x * x 实际上就是：

```
def f(x):
    return x * x
```

关键字lambda 表示匿名函数，冒号前面的 x 表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不写return，返回值就是该表达式的结果。

使用匿名函数，可以不必定义函数名，直接创建一个函数对象，很多时候可以简化代码：

```
>>> sorted([1, 3, 9, 5, 0], lambda x,y: -cmp(x,y))
[9, 5, 3, 1, 0]
```

返回函数的时候，也可以返回匿名函数：

```
>>> myabs = lambda x: -x if x < 0 else x
>>> myabs(-1)
1
>>> myabs(1)
1
```

任务

利用匿名函数简化以下代码：

```
def is_not_empty(s):
    return s and len(s.strip()) > 0
filter(is_not_empty, ['test', None, '', 'str', '  ', 'END'])
```

?不会了怎么办 定义匿名函数时，没有return关键字，且表达式的值就是函数返回值。

参考代码:

```
print filter(lambda s: s and len(s.strip())>0, ['test', None, '', 'str', '  ', 'END'])
```

2-11 python中decorator装饰器(mp4)

python中decorator装饰器.mp4

2-12 python中编写无参数decorator

Python的 decorator 本质上就是一个高阶函数，它接收一个函数作为参数，然后，返回一个新函数。

使用 decorator 用Python提供的 @ 语法，这样可以避免手动编写 f = decorate(f) 这样的代码。

考察一个@log的定义：

```
def log(f):
    def fn(x):
        print 'call ' + f.__name__ + '()...'
        return f(x)
    return fn
```

对于阶乘函数，@log工作得很好：

```
@log
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial(10)
```

结果：

```
call factorial()...
3628800
```

但是，对于参数不是一个的函数，调用将报错：

```
@log
def add(x, y):
    return x + y
print add(1, 2)
```

结果：

```
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    print add(1,2)
TypeError: fn() takes exactly 1 argument (2 given)
```

因为 add() 函数需要传入两个参数，但是 @log 写死了只含一个参数的返回函数。

要让 @log 自适应任何参数定义的函数，可以利用Python的 *args* 和 *kw，保证任意个数的参数总是能正常调用：

```
def log(f):
    def fn(*args, **kw):
        print 'call ' + f.__name__ + '()...'
        return f(*args, **kw)
    return fn
```

现在，对于任意函数，@log 都能正常工作。

任务

请编写一个@performance，它可以打印出函数调用的时间。

?不会了怎么办 计算函数调用的时间可以记录调用前后的当前时间戳，然后计算两个时间戳的差。

参考代码:

```
import time
def performance(f):
    def fn(*args, **kw):
        t1 = time.time()
        r = f(*args, **kw)
        t2 = time.time()
        print 'call %s() in %fs' % (f.__name__, (t2 - t1))
        return r
    return fn

@performance
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial(10)
```

2-13 python中编写带参数decorator

考察上一节的 @log 装饰器：

```
def log(f):
    def fn(x):
        print 'call ' + f.__name__ + '()...'
        return f(x)
    return fn
```

发现对于被装饰的函数，log打印的语句是不能变的（除了函数名）。

如果有的函数非常重要，希望打印出'[INFO] call xxx()...'，有的函数不太重要，希望打印出'[DEBUG] call xxx()...'，这时，log函数本身就需要传入'INFO'或'DEBUG'这样的参数，类似这样：

```
@log('DEBUG')
def my_func():
    pass
```

把上面的定义翻译成高阶函数的调用，就是：

```
my_func = log('DEBUG')(my_func)
```

上面的语句看上去还是比较绕，再展开一下：

```
log_decorator = log('DEBUG')
my_func = log_decorator(my_func)
```

上面的语句又相当于：

```
log_decorator = log('DEBUG')
@log_decorator
def my_func():
    pass
```

所以，带参数的log函数首先返回一个decorator函数，再让这个decorator函数接收my_func并返回新函数：

```
def log(prefix):
    def log_decorator(f):
        def wrapper(*args, **kw):
            print '[%s] %s()...' % (prefix, f.__name__)
            return f(*args, **kw)
        return wrapper
    return log_decorator

@log('DEBUG')
def test():
    pass
print test()
```

执行结果：

```
[DEBUG] test()...
None
```

对于这种3层嵌套的decorator定义，你可以先把它拆开：

```
# 标准decorator:
def log_decorator(f):
    def wrapper(*args, **kw):
        print '[%s] %s()...' % (prefix, f.__name__)
        return f(*args, **kw)
    return wrapper
return log_decorator

# 返回decorator:
def log(prefix):
    return log_decorator(f)
```

拆开以后会发现，调用会失败，因为在3层嵌套的decorator定义中，最内层的wrapper引用了最外层的参数prefix，所以，把一个闭包拆成普通的函数调用会比较困难。不支持闭包的编程语言要实现同样的功能就需要更多的代码。

任务

上一节的@performance只能打印秒，请给 @performace 增加一个参数，允许传入's'或'ms'：

```
@performance('ms')
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
```

?不会了怎么办 要实现带参数的@performance，就需要实现：

my_func = performance('ms')(my_func)

需要3层嵌套的decorator来实现。

参考代码:

```
import time
def performance(unit):
    def perf_decorator(f):
        def wrapper(*args, **kw):
            t1 = time.time()
            r = f(*args, **kw)
            t2 = time.time()
            t = (t2 - t1) * 1000 if unit=='ms' else (t2 - t1)
            print 'call %s() in %f %s' % (f.__name__, t, unit)
            return r
        return wrapper
    return perf_decorator

@performance('ms')
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial(10)
```

2-14 python中完善decorator

@decorator可以动态实现函数功能的增加，但是，经过@decorator“改造”后的函数，和原函数相比，除了功能多一点外，有没有其它不同的地方？

在没有decorator的情况下，打印函数名：

```
def f1(x):
    pass
print f1.__name__
```

输出： f1

有decorator的情况下，再打印函数名：

```
def log(f):
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
@log
def f2(x):
    pass
print f2.__name__
```

输出： wrapper

可见，由于decorator返回的新函数函数名已经不是'f2'，而是@log内部定义的'wrapper'。这对于那些依赖函数名的代码就会失效。decorator还改变了函数的 `__doc__` 等其它属性。如果要让调用者看不出一个函数经过了@decorator的“改造”，就需要把原函数的一些属性复制到新函数中：

```
def log(f):
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

这样写decorator很不方便，因为我们也很难把原函数的所有必要属性都一个一个复制到新函数上，所以Python内置的functools可以用来自动化完成这个“复制”的任务：

```
import functools
def log(f):
    @functools.wraps(f)
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
```

最后需要指出，由于我们把原函数签名改成了 `(*args, **kw)` ，因此，无法获得原函数的原始参数信息。即便我们采用固定参数来装饰只有一个参数的函数：

```
def log(f):
    @functools.wraps(f)
```

```
    def wrapper(x):
        print 'call...'
        return f(x)
    return wrapper
```

也可能改变原函数的参数名，因为新函数的参数名始终是 'x'，原函数定义的参数名不一定叫 'x'。

任务

请思考带参数的@decorator，@functools.wraps应该放置在哪：

```
def performance(unit):
    def perf_decorator(f):
        def wrapper(*args, **kw):
            ???
        return wrapper
    return perf_decorator
```

?不会了怎么办 注意@functools.wraps应该作用在返回的新函数上。

参考代码:

```
import time, functools
def performance(unit):
    def perf_decorator(f):
        @functools.wraps(f)
        def wrapper(*args, **kw):
            t1 = time.time()
            r = f(*args, **kw)
            t2 = time.time()
            t = (t2 - t1) * 1000 if unit=='ms' else (t2 - t1)
            print 'call %s() in %f %s' % (f.__name__, t, unit)
            return r
        return wrapper
    return perf_decorator

@performance('ms')
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial.__name__
```

2-15 python中偏函数

当一个函数有很多参数时，调用者就需要提供多个参数。如果减少参数个数，就可以简化调用者的负担。

比如，int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

```
>>> int('12345')
12345
```

但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 进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设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，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

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

这样，我们转换二进制就非常方便了：

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

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，functools.partial可以把一个参数多的函数变成一个参数少的新函数，少的参数需要在创建时指定默认值，这样，新函数调用的难度就降低了。

任务

在第7节中，我们在sorted这个高阶函数中传入自定义排序函数就可以实现忽略大小写排序。请用functools.partial把这个复杂调用变成一个简单的函数：

sorted_ignore_case(iterable) ?不会了怎么办 要固定sorted()的cmp参数，需要传入一个排序函数作为cmp的默认值。

参考代码:

```
import functools
sorted_ignore_case = functools.partial(sorted, cmp=lambda s1, s2: cmp(s1.upper(), s2.upper()))
print sorted_ignore_case(['bob', 'about', 'Zoo', 'Credit'])
```

第**3**章 模块

本章讲解如何使用Python的模块，如何编写和导入模块，以及如何安装并使用第三方模块。

3-1 python中模块和包的概念(mp4)

python中模块和包的概念.mp4

3-2 python之导入模块

要使用一个模块，我们必须首先导入该模块。Python使用import语句导入一个模块。例如，导入系统自带的模块 math：

```
import math
```

你可以认为math就是一个指向已导入模块的变量，通过该变量，我们可以访问math模块中所定义的所有公开的函数、变量和类：

```
>>> math.pow(2, 0.5) # pow是函数
1.4142135623730951

>>> math.pi # pi是变量
3.141592653589793
```

如果我们只希望导入用到的math模块的某几个函数，而不是所有函数，可以用下面的语句：

```
from math import pow, sin, log
```

这样，可以直接引用 pow, sin, log 这3个函数，但math的其他函数没有导入进来：

```
>>> pow(2, 10)
1024.0
>>> sin(3.14)
0.0015926529164868282
```

如果遇到名字冲突怎么办？比如math模块有一个log函数，logging模块也有一个log函数，如果同时使用，如何解决名字冲突？

如果使用import导入模块名，由于必须通过模块名引用函数名，因此不存在冲突：

```
import math, logging
print math.log(10)   # 调用的是math的log函数
logging.log(10, 'something')   # 调用的是logging的log函数
```

如果使用 from...import 导入 log 函数，势必引起冲突。这时，可以给函数起个“别名”来避免冲突：

```
from math import log
from logging import log as logger   # logging的log现在变成了logger
print log(10)   # 调用的是math的log
logger(10, 'import from logging')   # 调用的是logging的log
```

任务

Python的os.path模块提供了 isdir() 和 isfile()函数，请导入该模块，并调用函数判断指定的目录和文件是否存在。

注意:

1. 由于运行环境是平台服务器，所以测试的也是服务器中的文件夹和文件，该服务器上有/data/webroot/resource/python文件夹和/data/webroot/resource/python/test.txt文件，大家可以测试下。
2. 当然，大家可以在本机上测试是否存在相应的文件夹和文件。

import os print os.path.isdir(r'C:\Windows') print os.path.isfile(r'C:\Windows\notepad.exe') ?不会了怎么办 注意到os.path模块可以以若干种方式导入：

import os import os.path from os import path from os.path import isdir, isfile 每一种方式调用 isdir 和 isfile 都有所不同。

参考代码:

```
import os
print os.path.isdir(r'/data/webroot/resource/python')
print os.path.isfile(r'/data/webroot/resource/python/test.txt')
```

3-3 python中动态导入模块

如果导入的模块不存在，Python解释器会报 ImportError 错误：

```
>>> import something
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named something
```

有的时候，两个不同的模块提供了相同的功能，比如 StringIO 和 cStringIO 都提供了StringIO这个功能。

这是因为Python是动态语言，解释执行，因此Python代码运行速度慢。

如果要提高Python代码的运行速度，最简单的方法是把某些关键函数用 C 语言重写，这样就能大大提高执行速度。

同样的功能，StringIO 是纯Python代码编写的，而 cStringIO 部分函数是 C 写的，因此 cStringIO 运行速度更快。

利用ImportError错误，我们经常在Python中动态导入模块：

```
try:
    from cStringIO import StringIO
except ImportError:
    from StringIO import StringIO
```

上述代码先尝试从cStringIO导入，如果失败了（比如cStringIO没有被安装），再尝试从StringIO导入。这样，如果cStringIO模块存在，则我们将获得更快的运行速度，如果cStringIO不存在，则顶多代码运行速度会变慢，但不会影响代码的正常执行。

try 的作用是捕获错误，并在捕获到指定错误时执行 except 语句。

任务

利用import ... as ...，还可以动态导入不同名称的模块。

Python 2.6/2.7提供了json 模块，但Python 2.5以及更早版本没有json模块，不过可以安装一个simplejson模块，这两个模块提供的函数签名和功能都一模一样。

试写出导入json 模块的代码，能在Python 2.5/2.6/2.7都正常运行。

?不会了怎么办 先尝试导入json，如果失败，再尝试导入simplejson as json

参考代码:

```
try:
    import json
except ImportError:
    import simplejson as json
print json.dumps({'python':2.7})
```

3-4 python之使用 `__future__`

Python的新版本会引入新的功能，但是，实际上这些功能在上一个老版本中就已经存在了。要“试用”某一新的特性，就可以通过导入 `__future__` 模块的某些功能来实现。

例如，Python 2.7的整数除法运算结果仍是整数：

```
>>> 10 / 3
3
```

但是，Python 3.x已经改进了整数的除法运算，“/”除将得到浮点数，“//”除才仍是整数：

```
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
```

要在Python 2.7中引入3.x的除法规则，导入 `__future__` 的division：

```
>>> from __future__ import division
>>> print 10 / 3
3.3333333333333335
```

当新版本的一个特性与旧版本不兼容时，该特性将会在旧版本中添加到 `__future__` 中，以便旧的代码能在旧版本中测试新特性。

任务

在Python 3.x中，字符串统一为unicode，不需要加前缀 u，而以字节存储的str则必须加前缀 b。请利用 `__future__` 的unicode_literals在Python 2.7中编写unicode字符串。

?不会了怎么办 使用 `from __future__ import unicode_literals` 将把Python 3.x的unicode规则带入Python 2.7中。

参考代码:

```
from __future__ import unicode_literals
s = 'am I an unicode?'
print isinstance(s, unicode)
```

3-5 python之安装第三方模块(mp4)

python之安装第三方模块.mp4

第**4**章 面向对象编程基础

本章讲解Python面向对象编程的概念，如何创建类和实例，如何定义类的属性和方法。

4-1 python之面向对象编程(mp4)

python之面向对象编程.mp4

4-2 python之定义类并创建实例

在Python中，类通过 class 关键字定义。以 Person 为例，定义一个Person类如下：

```
class Person(object):
    pass
```

按照 Python 的编程习惯，类名以大写字母开头，紧接着是(object)，表示该类是从哪个类继承下来的。类的继承将在后面的章节讲解，现在我们只需要简单地从object类继承。

有了Person类的定义，就可以创建出具体的xiaoming、xiaohong等实例。创建实例使用 类名+()，类似函数调用的形式创建：

```
xiaoming = Person()
xiaohong = Person()
```

任务

请练习定义Person类，并创建出两个实例，打印实例，再比较两个实例是否相等。

?不会了怎么办 要打印实例，直接使用print语句；

要比较两个实例是否相等，用==操作符。

参考代码:

```
class Person(object):
    pass
xiaoming = Person()
xiaohong = Person()
print xiaoming
print xiaohong
print xiaoming == xiaohong
```

4-3 python中创建实例属性

虽然可以通过Person类创建出xiaoming、xiaohong等实例，但是这些实例看上除了地址不同外，没有什么其他不同。在现实世界中，区分xiaoming、xiaohong要依靠他们各自的名字、性别、生日等属性。

如何让每个实例拥有各自不同的属性？由于Python是动态语言，对每一个实例，都可以直接给他们的属性赋值，例如，给xiaoming这个实例加上name、gender和birth属性：

```
xiaoming = Person()
xiaoming.name = 'Xiao Ming'
xiaoming.gender = 'Male'
xiaoming.birth = '1990-1-1'
```

给xiaohong加上的属性不一定要和xiaoming相同：

```
xiaohong = Person()
xiaohong.name = 'Xiao Hong'
xiaohong.school = 'No. 1 High School'
xiaohong.grade = 2
```

实例的属性可以像普通变量一样进行操作：

```
xiaohong.grade = xiaohong.grade + 1
```

任务

请创建包含两个 Person 类的实例的 list，并给两个实例的 name 赋值，然后按照 name 进行排序。

?不会了怎么办 sorted() 是高阶函数，接受一个比较函数。

参考代码:

```
class Person(object):
    pass
p1 = Person()
p1.name = 'Bart'

p2 = Person()
p2.name = 'Adam'

p3 = Person()
p3.name = 'Lisa'

L1 = [p1, p2, p3]
L2 = sorted(L1, lambda p1, p2: cmp(p1.name, p2.name))

print L2[0].name
print L2[1].name
print L2[2].name
```

4-4 python中初始化实例属性

虽然我们可以自由地给一个实例绑定各种属性，但是，现实世界中，一种类型的实例应该拥有相同名字的属性。例如，Person类应该在创建的时候就拥有 name、gender 和 birth 属性，怎么办？

在定义 Person 类时，可以为Person类添加一个特殊的 `__init__()` 方法，当创建实例时， `__init__()` 方法被自动调用，我们就能在此为每个实例都统一加上以下属性：

```
class Person(object):
    def __init__(self, name, gender, birth):
        self.name = name
        self.gender = gender
        self.birth = birth
```

`__init__()` 方法的第一个参数必须是 self（也可以用别的名字，但建议使用习惯用法），后续参数则可以自由指定，和定义函数没有任何区别。

相应地，创建实例时，就必须要提供除 self 以外的参数：

```
xiaoming = Person('Xiao Ming', 'Male', '1991-1-1')
xiaohong = Person('Xiao Hong', 'Female', '1992-2-2')
```

有了 `__init__()` 方法，每个Person实例在创建时，都会有 name、gender 和 birth 这3个属性，并且，被赋予不同的属性值，访问属性使用.操作符：

```
print xiaoming.name
# 输出 'Xiao Ming'
print xiaohong.birth
# 输出 '1992-2-2'
```

要特别注意的是，初学者定义 `__init__()` 方法常常忘记了 self 参数：

```
>>> class Person(object):
...     def __init__(name, gender, birth):
...         pass
...
>>> xiaoming = Person('Xiao Ming', 'Male', '1990-1-1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (4 given)
```

这会导致创建失败或运行不正常，因为第一个参数name被Python解释器传入了实例的引用，从而导致整个方法的调用参数位置全部没有对上。

任务

请定义Person类的 `__init__` 方法，除了接受 name、gender 和 birth 外，还可接受任意关键字参数，并把他们都作为属性赋值给实例。

?不会了怎么办 要定义关键字参数，使用 `**kw` ；

除了可以直接使用self.name = 'xxx'设置一个属性外，还可以通过 setattr(self, 'name', 'xxx') 设置属性。

参考代码:

```
class Person(object):
    def __init__(self, name, gender, birth, **kw):
        self.name = name
        self.gender = gender
        self.birth = birth
        for k, v in kw.iteritems():
            setattr(self, k, v)
xiaoming = Person('Xiao Ming', 'Male', '1990-1-1', job='Student')
print xiaoming.name
print xiaoming.job
```

4-5 python中访问限制

我们可以给一个实例绑定很多属性，如果有些属性不希望被外部访问到怎么办？

Python对属性权限的控制是通过属性名来实现的，如果一个属性由双下划线开头(__)，该属性就无法被外部访问。看例子：

```
class Person(object):
    def __init__(self, name):
        self.name = name
        self._title = 'Mr'
        self.__job = 'Student'
p = Person('Bob')
print p.name
# => Bob
print p._title
# => Mr
print p.__job
# => Error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute '__job'
```

可见，只有以双下划线开头的"__job"不能直接被外部访问。

但是，如果一个属性以" __xxx__ "的形式定义，那它又可以被外部访问了，以" __xxx__ "定义的属性在Python的类中被称为特殊属性，有很多预定义的特殊属性可以使用，通常我们不要把普通属性用" __xxx__ "定义。

以单下划线开头的属性"_xxx"虽然也可以被外部访问，但是，按照习惯，他们不应该被外部访问。

任务

请给Person类的 __init__ 方法中添加name和score参数，并把score绑定到__score属性上，看看外部是否能访问到。

?不会了怎么办 以双下划线开头的属性无法被外部访问，" __xxx__ "除外。

参考代码:

```
class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.__score = score

p = Person('Bob', 59)

print p.name
print p.__score
```

4-6 python中创建类属性

类是模板，而实例则是根据类创建的对象。

绑定在一个实例上的属性不会影响其他实例，但是，类本身也是一个对象，如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。

定义类属性可以直接在 class 中定义：

```
class Person(object):
    address = 'Earth'
    def __init__(self, name):
        self.name = name
```

因为类属性是直接绑定在类上的，所以，访问类属性不需要创建实例，就可以直接访问：

```
print Person.address
# => Earth
```

对一个实例调用类的属性也是可以访问的，所有实例都可以访问到它所属的类的属性：

```
p1 = Person('Bob')
p2 = Person('Alice')
print p1.address
# => Earth
print p2.address
# => Earth
```

由于Python是动态语言，类属性也是可以动态添加和修改的：

```
Person.address = 'China'
print p1.address
# => 'China'
print p2.address
# => 'China'
```

因为类属性只有一份，所以，当Person类的address改变时，所有实例访问到的类属性都改变了。

任务

请给 Person 类添加一个类属性 count，每创建一个实例，count 属性就加 1，这样就可以统计出一共创建了多少个 Person 的实例。

?不会了怎么办 由于创建实例必定会调用 `__init__()` 方法，所以在这里修改类属性 count 很合适。

参考代码:

```
class Person(object):
    count = 0
    def __init__(self, name):
        Person.count = Person.count + 1
        self.name = name
p1 = Person('Bob')
print Person.count
```

```
# => 1
p2 = Person('Alice')
print Person.count
# => 2
p3 = Person('Tim')
print Person.count
# => 3
```

4-7 python中类属性和实例属性名字冲突怎么办

修改类属性会导致所有实例访问到的类属性全部都受影响，但是，如果在实例变量上修改类属性会发生什么问题呢？

```
class Person(object):
    address = 'Earth'
    def __init__(self, name):
        self.name = name

p1 = Person('Bob')
p2 = Person('Alice')

print 'Person.address = ' + Person.address

p1.address = 'China'
print 'p1.address = ' + p1.address

print 'Person.address = ' + Person.address
print 'p2.address = ' + p2.address
```

结果如下：

```
Person.address = Earth
p1.address = China
Person.address = Earth
p2.address = Earth
```

我们发现，在设置了 p1.address = 'China' 后，p1访问 address 确实变成了 'China'，但是，Person.address和p2.address仍然是'Earch'，怎么回事？

原因是 p1.address = 'China'并没有改变 Person 的 address，而是给 p1这个实例绑定了实例属性address ，对p1来说，它有一个实例属性address（值是'China'），而它所属的类Person也有一个类属性address，所以:

访问 p1.address 时，优先查找实例属性，返回'China'。

访问 p2.address 时，p2没有实例属性address，但是有类属性address，因此返回'Earth'。

可见，当实例属性和类属性重名时，实例属性优先级高，它将屏蔽掉对类属性的访问。

当我们把 p1 的 address 实例属性删除后，访问 p1.address 就又返回类属性的值 'Earth'了：

```
del p1.address
print p1.address
# => Earth
```

可见，千万不要在实例上修改类属性，它实际上并没有修改类属性，而是给实例绑定了一个实例属性。

任务

请把上节的 Person 类属性 count 改为 __count，再试试能否从实例和类访问该属性。

?不会了怎么办 把count改为私有 `__count` ，这样实例变量在外部无法修改 `__count`

参考代码:

```
class Person(object):
    __count = 0
```

```
    def __init__(self, name):
        Person.__count = Person.__count + 1
        self.name = name
        print Person.__count

p1 = Person('Bob')
p2 = Person('Alice')

print Person.__count
```

4-8 python中定义实例方法

一个实例的私有属性就是以__开头的属性，无法被外部访问，那这些属性定义有什么用？

虽然私有属性无法从外部访问，但是，从类的内部是可以访问的。除了可以定义实例的属性外，还可以定义实例的方法。

实例的方法就是在类中定义的函数，它的第一个参数永远是 self，指向调用该方法的实例本身，其他参数和一个普通函数是完全一样的：

```
class Person(object):

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

    def get_name(self):
        return self.__name
```

`get_name(self)` 就是一个实例方法，它的第一个参数是self。 `__init__(self, name)` 其实也可看做是一个特殊的实例方法。

调用实例方法必须在实例上调用：

```
p1 = Person('Bob')
print p1.get_name()  # self不需要显式传入
# => Bob
```

在实例方法内部，可以访问所有实例属性，这样，如果外部需要访问私有属性，可以通过方法调用获得，这种数据封装的形式除了能保护内部数据一致性外，还可以简化外部调用的难度。

任务

请给 Person 类增加一个私有属性 `__score`，表示分数，再增加一个实例方法 get_grade()，能根据 `__score` 的值分别返回 A-优秀, B-及格, C-不及格三档。

?不会了怎么办 注意get_grade()是实例方法，第一个参数为self。

参考代码:

```
class Person(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def get_grade(self):
        if self.__score >= 80:
            return 'A'
        if self.__score >= 60:
            return 'B'
        return 'C'

p1 = Person('Bob', 90)
p2 = Person('Alice', 65)
p3 = Person('Tim', 48)

print p1.get_grade()
print p2.get_grade()
```

```
print p3.get_grade()
```

4-9 python中方法也是属性

我们在 class 中定义的实例方法其实也是属性，它实际上是一个函数对象：

```
class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def get_grade(self):
        return 'A'

p1 = Person('Bob', 90)
print p1.get_grade
# => <bound method Person.get_grade of <__main__.Person object at 0x109e58510>>
print p1.get_grade()
# => A
```

也就是说，p1.get_grade 返回的是一个函数对象，但这个函数是一个绑定到实例的函数，p1.get_grade() 才是方法调用。

因为方法也是一个属性，所以，它也可以动态地添加到实例上，只是需要用 types.MethodType() 把一个函数变为一个方法：

```
import types
def fn_get_grade(self):
    if self.score >= 80:
        return 'A'
    if self.score >= 60:
        return 'B'
    return 'C'

class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

p1 = Person('Bob', 90)
p1.get_grade = types.MethodType(fn_get_grade, p1, Person)
print p1.get_grade()
# => A
p2 = Person('Alice', 65)
print p2.get_grade()
# ERROR: AttributeError: 'Person' object has no attribute 'get_grade'
# 因为p2实例并没有绑定get_grade
```

给一个实例动态添加方法并不常见，直接在class中定义要更直观。

任务

由于属性可以是普通的值对象，如 str，int 等，也可以是方法，还可以是函数，大家看看下面代码的运行结果，请想一想 p1.get_grade 为什么是函数而不是方法：

```
class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
        self.get_grade = lambda: 'A'
```

```
p1 = Person('Bob', 90)
print p1.get_grade
print p1.get_grade()
```

?不会了怎么办 直接把 lambda 函数赋值给 self.get_grade 和绑定方法有所不同，函数调用不需要传入 self，但是方法调用需要传入 self。

4-10 python中定义类方法

和属性类似，方法也分实例方法和类方法。

在class中定义的全部是实例方法，实例方法第一个参数 self 是实例本身。

要在class中定义类方法，需要这么写：

```
class Person(object):
    count = 0
    @classmethod
    def how_many(cls):
        return cls.count
    def __init__(self, name):
        self.name = name
        Person.count = Person.count + 1

print Person.how_many()
p1 = Person('Bob')
print Person.how_many()
```

通过标记一个 @classmethod，该方法将绑定到 Person 类上，而非类的实例。类方法的第一个参数将传入类本身，通常将参数名命名为 cls，上面的 cls.count 实际上相当于 Person.count。

因为是在类上调用，而非实例上调用，因此类方法无法获得任何实例变量，只能获得类的引用。

任务

如果将类属性 count 改为私有属性 `__count`，则外部无法读取 `__score`，但可以通过一个类方法获取，请编写类方法获得 `__count` 值。

?不会了怎么办 注意类方法需要添加 @classmethod

参考代码:

```
class Person(object):
    __count = 0
    @classmethod
    def how_many(cls):
        return cls.__count
    def __init__(self, name):
        self.name = name
        Person.__count = Person.__count + 1

print Person.how_many()
p1 = Person('Bob')
print Person.how_many()
```

第**5**章 类的继承

本章讲解Python类的继承，如何判断实例类型，多态以及如何获取对象信息。

5-1 python中什么是继承(mp4)

python中什么是继承.mp4

5-2 python中继承一个类

如果已经定义了Person类，需要定义新的Student和Teacher类时，可以直接从Person类继承：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

定义Student类时，只需要把额外的属性加上，例如score：

```
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
```

一定要用 super(Student, self).**init**(name, gender) 去初始化父类，否则，继承自 Person 的 Student 将没有 name 和 gender。

函数super(Student, self)将返回当前类继承的父类，即 Person ，然后调用 `__init__()` 方法，注意self参数已在super()中传入，在 `__init__()` 中将隐式传递，不需要写出（也不能写）。

任务

请参考 Student 类，编写一个 Teacher类，也继承自 Person。

?不会了怎么办 要正确调用 super() 的 `__init__` 方法。

参考代码:

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course

t = Teacher('Alice', 'Female', 'English')
print t.name
print t.course
```

5-3 python中判断类型

函数isinstance()可以判断一个变量的类型，既可以用在Python内置的数据类型如str、list、dict，也可以用在我们自定义的类，它们本质上都是数据类型。

假设有如下的 Person、Student 和 Teacher 的定义及继承关系如下：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score

class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course

p = Person('Tim', 'Male')
s = Student('Bob', 'Male', 88)
t = Teacher('Alice', 'Female', 'English')
```

当我们拿到变量 p、s、t 时，可以使用 isinstance 判断类型：

```
>>> isinstance(p, Person)
True    # p是Person类型
>>> isinstance(p, Student)
False   # p不是Student类型
>>> isinstance(p, Teacher)
False   # p不是Teacher类型
```

这说明在继承链上，一个父类的实例不能是子类类型，因为子类比父类多了一些属性和方法。

我们再考察 s ：

```
>>> isinstance(s, Person)
True    # s是Person类型
>>> isinstance(s, Student)
True    # s是Student类型
>>> isinstance(s, Teacher)
False   # s不是Teacher类型
```

s 是Student类型，不是Teacher类型，这很容易理解。但是，s 也是Person类型，因为Student继承自Person，虽然它比Person多了一些属性和方法，但是，把 s 看成Person的实例也是可以的。

这说明在一条继承链上，一个实例可以看成它本身的类型，也可以看成它父类的类型。

任务

请根据继承链的类型转换，依次思考 t 是否是 Person，Student，Teacher，object 类型，并使用isinstance()判断来验证您的答案。

?不会了怎么办 注意t 是Teacher的实例，继承链是：

object <- Person <- Teacher

参考代码:

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score

class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course

t = Teacher('Alice', 'Female', 'English')

print isinstance(t, Person)
print isinstance(t, Student)
print isinstance(t, Teacher)
print isinstance(t, object)
```

5-4 python中多态

类具有继承关系，并且子类类型可以向上转型看做父类类型，如果我们从 Person 派生出 Student和Teacher ，并都写了一个 whoAmI() 方法：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def whoAmI(self):
        return 'I am a Person, my name is %s' % self.name

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
    def whoAmI(self):
        return 'I am a Student, my name is %s' % self.name

class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course
    def whoAmI(self):
        return 'I am a Teacher, my name is %s' % self.name
```

在一个函数中，如果我们接收一个变量 x，则无论该 x 是 Person、Student还是 Teacher，都可以正确打印出结果：

```
def who_am_i(x):
    print x.whoAmI()

p = Person('Tim', 'Male')
s = Student('Bob', 'Male', 88)
t = Teacher('Alice', 'Female', 'English')

who_am_i(p)
who_am_i(s)
who_am_i(t)
```

运行结果：

```
I am a Person, my name is Tim
I am a Student, my name is Bob
I am a Teacher, my name is Alice
```

这种行为称为多态。也就是说，方法调用将作用在 x 的实际类型上。s 是Student类型，它实际上拥有自己的 whoAmI()方法以及从 Person继承的 whoAmI方法，但调用 s.whoAmI()总是先查找它自身的定义，如果没有定义，则顺着继承链向上查找，直到在某个父类中找到为止。

由于Python是动态语言，所以，传递给函数 who_am_i(x)的参数 x 不一定是 Person 或 Person 的子类型。任何数据类型的实例都可以，只要它有一个whoAmI()的方法即可：

```
class Book(object):
    def whoAmI(self):
        return 'I am a book'
```

这是动态语言和静态语言（例如Java）最大的差别之一。动态语言调用实例方法，不检查类型，只要方法存在，参数正确，就可以调用。

任务

Python提供了open()函数来打开一个磁盘文件，并返回 File 对象。File对象有一个read()方法可以读取文件内容：

例如，从文件读取内容并解析为JSON结果：

```
import json
f = open('/path/to/file.json', 'r')
print json.load(f)
```

由于Python的动态特性，json.load()并不一定要从一个File对象读取内容。任何对象，只要有read()方法，就称为File-like Object，都可以传给json.load()。

请尝试编写一个File-like Object，把一个字符串 r'["Tim", "Bob", "Alice"]'包装成 File-like Object 并由 json.load() 解析。

?不会了怎么办 只要为Students类加上 read()方法，就变成了一个File-like Object。

参考代码:

```
import json

class Students(object):
    def read(self):
        return r'["Tim", "Bob", "Alice"]'

s = Students()

print json.load(s)
```

5-5 python中多重继承

除了从一个父类继承外，Python允许从多个父类继承，称为多重继承。

多重继承的继承链就不是一棵树了，它像这样：

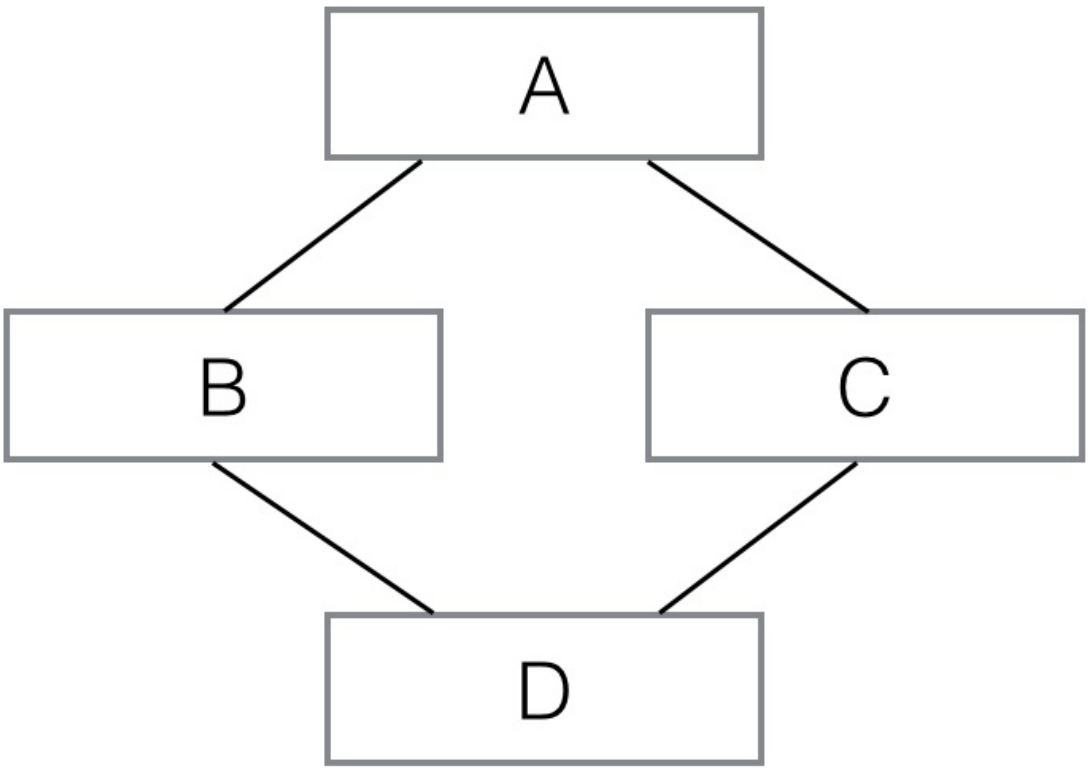
```
class A(object):
    def __init__(self, a):
        print 'init A...'
        self.a = a

class B(A):
    def __init__(self, a):
        super(B, self).__init__(a)
        print 'init B...'

class C(A):
    def __init__(self, a):
        super(C, self).__init__(a)
        print 'init C...'

class D(B, C):
    def __init__(self, a):
        super(D, self).__init__(a)
        print 'init D...'
```

看下图:



像这样，D 同时继承自 B 和 C，也就是 D 拥有了 A、B、C 的全部功能。多重继承通过 super()调用**init**()方法时，A 虽然被继承了两次，但**init**()只调用一次：

```
>>> d = D('d')
init A...
init C...
init B...
init D...
```

多重继承的目的是从两种继承树中分别选择并继承出子类，以便组合功能使用。

举个例子，Python的网络服务器有TCPServer、UDPServer、UnixStreamServer、UnixDatagramServer，而服务器运行模式有 多进程ForkingMixin 和 多线程ThreadingMixin两种。

要创建多进程模式的 TCPServer：

```
class MyTCPServer(TCPServer, ForkingMixin)
    pass
```

要创建多线程模式的 UDPServer：

```
class MyUDPServer(UDPServer, ThreadingMixin):
    pass
```

如果没有多重继承，要实现上述所有可能的组合需要 4x2=8 个子类。

任务

+-Person +- Student +- Teacher

是一类继承树；

+- SkillMixin +- BasketballMixin +- FootballMixin

是一类继承树。

通过多重继承，请定义“会打篮球的学生”和“会踢足球的老师”。

?不会了怎么办 多重继承需要从两个或更多的类派生。

参考代码:

```
class Person(object):
    pass

class Student(Person):
    pass

class Teacher(Person):
    pass

class SkillMixin(object):
    pass

class BasketballMixin(SkillMixin):
    def skill(self):
        return 'basketball'

class FootballMixin(SkillMixin):
```

```
    def skill(self):
        return 'football'

class BStudent(Student, BasketballMixin):
    pass

class FTeacher(Teacher, FootballMixin):
    pass

s = BStudent()
print s.skill()

t = FTeacher()
print t.skill()
```

5-6 python中获取对象信息

拿到一个变量，除了用 isinstance() 判断它是否是某种类型的实例外，还有没有别的方法获取到更多的信息呢？

例如，已有定义：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
    def whoAmI(self):
        return 'I am a Student, my name is %s' % self.name
```

首先可以用 type() 函数获取变量的类型，它返回一个 Type 对象：

```
>>> type(123)
<type 'int'>
>>> s = Student('Bob', 'Male', 88)
>>> type(s)
<class '__main__.Student'>
```

其次，可以用 dir() 函数获取变量的所有属性：

```
>>> dir(123)   # 整数也有很多属性...
['__abs__', '__add__', '__and__', '__class__', '__cmp__', ...]

>>> dir(s)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__',
 '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'gender', 'name', 'score', 'whoAmI']
```

对于实例变量，dir()返回所有实例属性，包括 `__class__` 这类有特殊意义的属性。注意到方法 `whoAmI` 也是 s 的一个属性。

如何去掉 `__xxx__` 这类的特殊属性，只保留我们自己定义的属性？回顾一下filter()函数的用法。

dir()返回的属性是字符串列表，如果已知一个属性名称，要获取或者设置对象的属性，就需要用 getattr() 和 setattr( )函数了：

```
>>> getattr(s, 'name')  # 获取name属性
'Bob'

>>> setattr(s, 'name', 'Adam')  # 设置新的name属性

>>> s.name
'Adam'

>>> getattr(s, 'age')  # 获取age属性，但是属性不存在，报错：
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

```
>>> getattr(s, 'age', 20)  # 获取age属性，如果属性不存在，就返回默认值20：
20
```

任务

对于Person类的定义：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

希望除了 name和gender 外，可以提供任意额外的关键字参数，并绑定到实例，请修改 Person 的 **init**()定 义，完成该功能。

?不会了怎么办 传入**kw 即可传入任意数量的参数，并通过 setattr() 绑定属性。

参考代码:

```
class Person(object):
    def __init__(self, name, gender, **kw):
        self.name = name
        self.gender = gender
        for k, v in kw.iteritems():
            setattr(self, k, v)

p = Person('Bob', 'Male', age=18, course='Python')
print p.age
print p.course
```

第6章 定制类

本章讲解Python的特殊方法，以及如何利用特殊方法定制类，实现各种强大的功能。

6-1 python中什么是特殊方法(mp4)

python中什么是特殊方法.mp4

6-2 python中 `__str__` 和 `__repr__`

如果要把一个类的实例变成 str，就需要实现特殊方法 `__str__()`：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def __str__(self):
        return '(Person: %s, %s)' % (self.name, self.gender)
```

现在，在交互式命令行下用 print 试试：

```
>>> p = Person('Bob', 'male')
>>> print p
(Person: Bob, male)
```

但是，如果直接敲变量 p：

```
>>> p
<main.Person object at 0x10c941890>
```

似乎 `__str__()` 不会被调用。

因为 Python 定义了 `__str__()` 和 `__repr__()` 两种方法， `__str__()` 用于显示给用户，而 `__repr__()` 用于显示给开发人员。

有一个偷懒的定义 `__repr__` 的方法：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
    def __str__(self):
        return '(Person: %s, %s)' % (self.name, self.gender)
    __repr__ = __str__
```

任务

请给Student 类定义 `__str__` 和 `__repr__` 方法，使得能打印出：

```
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
```

?不会了怎么办 只要为Students 类加上 `__str__()` 和 `__repr__()` 方法即可。

参考代码:

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

```
class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score
    def __str__(self):
        return '(Student: %s, %s, %s)' % (self.name, self.gender, self.score)
    __repr__ = __str__

s = Student('Bob', 'male', 88)
print s
```

6-3 python中 `__cmp__`

对 int、str 等内置数据类型排序时，Python的 sorted() 按照默认的比较函数 cmp 排序，但是，如果对一组 Student 类的实例排序时，就必须提供我们自己的特殊方法 `__cmp__()`：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def __str__(self):
        return '(%s: %s)' % (self.name, self.score)
    __repr__ = __str__

    def __cmp__(self, s):
        if self.name < s.name:
            return -1
        elif self.name > s.name:
            return 1
        else:
            return 0
```

上述 Student 类实现了 `__cmp__()` 方法， `__cmp__` 用实例自身self和传入的实例 s 进行比较，如果 self 应该排在前面，就返回 -1，如果 s 应该排在前面，就返回1，如果两者相当，返回 0。

Student类实现了按name进行排序：

```
>>> L = [Student('Tim', 99), Student('Bob', 88), Student('Alice', 77)]
>>> print sorted(L)
[(Alice: 77), (Bob: 88), (Tim: 99)]
```

注意: 如果list不仅仅包含 Student 类，则 `__cmp__` 可能会报错：

```
L = [Student('Tim', 99), Student('Bob', 88), 100, 'Hello']
print sorted(L)
```

请思考如何解决。

任务

请修改 Student 的 `__cmp__` 方法，让它按照分数从高到底排序，分数相同的按名字排序。

?不会了怎么办 需要先比较 score，在 score 相等的情况下，再比较 name。

参考代码:

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def __str__(self):
        return '(%s: %s)' % (self.name, self.score)

    __repr__ = __str__

    def __cmp__(self, s):
```

```
        if self.score == s.score:
            return cmp(self.name, s.name)
        return -cmp(self.score, s.score)

L = [Student('Tim', 99), Student('Bob', 88), Student('Alice', 99)]
print sorted(L)
```

6-4 python中 `__len__`

如果一个类表现得像一个list，要获取有多少个元素，就得用 len() 函数。

要让 len() 函数工作正常，类必须提供一个特殊方法**len()**，它返回元素的个数。

例如，我们写一个 Students 类，把名字传进去：

```
class Students(object):
    def __init__(self, *args):
        self.names = args
    def __len__(self):
        return len(self.names)
```

只要正确实现了**len()**方法，就可以用len()函数返回Students实例的“长度”：

```
>>> ss = Students('Bob', 'Alice', 'Tim')
>>> print len(ss)
3
```

任务

斐波那契数列是由 0, 1, 1, 2, 3, 5, 8...构成。

请编写一个Fib类，Fib(10)表示数列的前10个元素，print Fib(10) 可以打印出数列的前 10 个元素，len(Fib(10))可以正确返回数列的个数10。

?不会了怎么办 需要根据num计算出斐波那契数列的前N个元素。

参考代码:

```
class Fib(object):
    def __init__(self, num):
        a, b, L = 0, 1, []
        for n in range(num):
            L.append(a)
            a, b = b, a + b
        self.numbers = L

    def __str__(self):
        return str(self.numbers)

    __repr__ = __str__

    def __len__(self):
        return len(self.numbers)

f = Fib(10)
print f
print len(f)
```

6-5 python中数学运算

Python 提供的基本数据类型 int、float 可以做整数和浮点的四则运算以及乘方等运算。

但是，四则运算不局限于int和float，还可以是有理数、矩阵等。

要表示有理数，可以用一个Rational类来表示：

```
class Rational(object):
    def __init__(self, p, q):
        self.p = p
        self.q = q
```

p、q 都是整数，表示有理数 p/q。

如果要让Rational进行+运算，需要正确实现**add**：

```
class Rational(object):
    def __init__(self, p, q):
        self.p = p
        self.q = q
    def __add__(self, r):
        return Rational(self.p * r.q + self.q * r.p, self.q * r.q)
    def __str__(self):
        return '%s/%s' % (self.p, self.q)
    __repr__ = __str__
```

现在可以试试有理数加法：

```
>>> r1 = Rational(1, 3)
>>> r2 = Rational(1, 2)
>>> print r1 + r2
5/6
```

任务

Rational类虽然可以做加法，但无法做减法、乘方和除法，请继续完善Rational类，实现四则运算。

提示：

```
减法运算：__sub__
乘法运算：__mul__
除法运算：__div__
```

?不会了怎么办 如果运算结果是 6/8，在显示的时候需要归约到最简形式3/4。

参考代码:

```
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

class Rational(object):
    def __init__(self, p, q):
        self.p = p
```

```
        self.q = q
    def __add__(self, r):
        return Rational(self.p * r.q + self.q * r.p, self.q * r.q)
    def __sub__(self, r):
        return Rational(self.p * r.q - self.q * r.p, self.q * r.q)
    def __mul__(self, r):
        return Rational(self.p * r.p, self.q * r.q)
    def __div__(self, r):
        return Rational(self.p * r.q, self.q * r.p)
    def __str__(self):
        g = gcd(self.p, self.q)
        return '%s/%s' % (self.p / g, self.q / g)
    __repr__ = __str__

r1 = Rational(1, 2)
r2 = Rational(1, 4)
print r1 + r2
print r1 - r2
print r1 * r2
print r1 / r2
```

6-6 python中类型转换

Rational类实现了有理数运算，但是，如果要把结果转为 int 或 float 怎么办？

考察整数和浮点数的转换：

```
>>> int(12.34)
12
>>> float(12)
12.0
```

如果要把 Rational 转为 int，应该使用：

```
r = Rational(12, 5)
n = int(r)
```

要让int()函数正常工作，只需要实现特殊方法 `__int__()`：

```
class Rational(object):
    def __init__(self, p, q):
        self.p = p
        self.q = q
    def __int__(self):
        return self.p // self.q
```

结果如下：

```
>>> print int(Rational(7, 2))
3
>>> print int(Rational(1, 3))
0
```

同理，要让float()函数正常工作，只需要实现特殊方法 `__float__()` 。

任务

请继续完善Rational，使之可以转型为float。

?不会了怎么办 将self.p转型为float类型，再作除法就可以得到float：

float(self.p) / self.q

参考代码:

```
class Rational(object):
    def __init__(self, p, q):
        self.p = p
        self.q = q

    def __int__(self):
        return self.p // self.q

    def __float__(self):
        return float(self.p) / self.q

print float(Rational(7, 2))
```

```
print float(Rational(1, 3))
```

6-7 python中 @property

考察 Student 类：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
```

当我们想要修改一个 Student 的 scroe 属性时，可以这么写：

```
s = Student('Bob', 59)
s.score = 60
```

但是也可以这么写：

```
s.score = 1000
```

显然，直接给属性赋值无法检查分数的有效性。

如果利用两个方法：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.__score = score
    def get_score(self):
        return self.__score
    def set_score(self, score):
        if score < 0 or score > 100:
            raise ValueError('invalid score')
        self.__score = score
```

这样一来，s.set_score(1000) 就会报错。

这种使用 get/set 方法来封装对一个属性的访问在许多面向对象编程的语言中都很常见。

但是写 s.get_score() 和 s.set_score() 没有直接写 s.score 来得直接。

有没有两全其美的方法？----有。

因为Python支持高阶函数，在函数式编程中我们介绍了装饰器函数，可以用装饰器函数把 get/set 方法“装饰”成属性调用：

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.__score = score
    @property
    def score(self):
        return self.__score
    @score.setter
    def score(self, score):
        if score < 0 or score > 100:
            raise ValueError('invalid score')
        self.__score = score
```

注意: 第一个score(self)是get方法，用@property装饰，第二个score(self, score)是set方法，用@score.setter装饰，@score.setter是前一个@property装饰后的副产品。

现在，就可以像使用属性一样设置score了：

```
>>> s = Student('Bob', 59)
>>> s.score = 60
>>> print s.score
60
>>> s.score = 1000
Traceback (most recent call last):
  ...
ValueError: invalid score
```

说明对 score 赋值实际调用的是 set方法。

任务 如果没有定义set方法，就不能对“属性”赋值，这时，就可以创建一个只读“属性”。

请给Student类加一个grade属性，根据 score 计算 A（>=80）、B、C（<60）。

?不会了怎么办 用 @property 修饰 grade 的 get 方法即可实现只读属性。

参考代码:

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.__score = score
    @property
    def score(self):
        return self.__score
    @score.setter
    def score(self, score):
        if score < 0 or score > 100:
            raise ValueError('invalid score')
        self.__score = score
    @property
    def grade(self):
        if self.score < 60:
            return 'C'
        if self.score < 80:
            return 'B'
        return 'A'
s = Student('Bob', 59)
print s.grade
s.score = 60
print s.grade
s.score = 99
print s.grade
```

6-8 python中 `__slots__`

由于Python是动态语言，任何实例在运行期都可以动态地添加属性。

如果要限制添加的属性，例如，Student类只允许添加 name、gender和score 这3个属性，就可以利用Python的一个特殊的 `__slots__` 来实现。

顾名思义，`__slots__` 是指一个类允许的属性列表：

```
class Student(object):
    __slots__ = ('name', 'gender', 'score')
    def __init__(self, name, gender, score):
        self.name = name
        self.gender = gender
        self.score = score
```

现在，对实例进行操作：

```
>>> s = Student('Bob', 'male', 59)
>>> s.name = 'Tim' # OK
>>> s.score = 99 # OK
>>> s.grade = 'A'
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'grade'
```

`__slots__` 的目的是限制当前类所能拥有的属性，如果不需要添加任意动态的属性，使用 `__slots__` 也能节省内存。

任务

假设Person类通过 `__slots__` 定义了name和gender，请在派生类Student中通过 `__slots__` 继续添加score的定义，使Student类可以实现name、gender和score 3个属性。

?不会了怎么办 Student类的 `__slots__` 只需要包含Person类不包含的score属性即可。

参考代码:

```
class Person(object):
    __slots__ = ('name', 'gender')
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

class Student(Person):
    __slots__ = ('score',)
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score

s = Student('Bob', 'male', 59)
s.name = 'Tim'
s.score = 99
print s.score
```

6-9 python中 `__call__`

在Python中，函数其实是一个对象：

```
>>> f = abs
>>> f.__name__
'abs'
>>> f(-123)
123
```

由于 f 可以被调用，所以，f 被称为可调用对象。

所有的函数都是可调用对象。

一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法 `__call__()` 。

我们把 Person 类变成一个可调用对象：

```
class Person(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

    def __call__(self, friend):
        print 'My name is %s...' % self.name
        print 'My friend is %s...' % friend
```

现在可以对 Person 实例直接调用：

```
>>> p = Person('Bob', 'male')
>>> p('Tim')
My name is Bob...
My friend is Tim...
```

单看 p('Tim') 你无法确定 p 是一个函数还是一个类实例，所以，在Python中，函数也是对象，对象和函数的区别并不显著。

任务

改进一下前面定义的斐波那契数列：

```
class Fib(object):
    ???
```

请加一个 `__call__` 方法，让调用更简单：

```
>>> f = Fib()
>>> print f(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

?不会了怎么办 要正确定义参数： `__call__(self, num)`

参考代码:

```
class Fib(object):
```

```
    def __call__(self, num):
        a, b, L = 0, 1, []
        for n in range(num):
            L.append(a)
            a, b = b, a + b
        return L

f = Fib()
print f(10)
```

第7章 课程总结

对课程进行概括性总结，并对后续课程进行简单说明。

7-1 课程总结(mp4)

课程总结.mp4

讲师：廖雪峰

> 全栈工程师，独立iOS开发者，精通Python / Java / JavaScript / Node / Objective-C / Scheme等。技术分享大拿，热爱跑步、爬山，热爱生活

本课程所有源代码