第一章

python中有很多内置函数，内置函数可以用来做运算符重载、类初始化等等，使得python的类能够和标准的集合类型一样处理，即代码风格统一。如：

```
__getitem__() # 重载【】运算符
__len__() # 即len()时会调用
__repr__() # 在控制台上时直接打印
__str__() # print时调用
__doc__() # 用于编辑说明文件
__contains__() # in运算符，默认按顺序做一次迭代搜索
__init__() # 初始化

# random里的choice包可以用于随机选择
from random import choice
choice(deck) # 在桌上随机抽一张牌出来
```

注意在使用repr打印的时候可以规定打印的风格如：

```
def __repr__(self):
    return ' Vector(%r,%r)' % (self.x,self.y)
    # 其中后面括号中的元组代替%r的位置
```

算数运算符的重载：

```
__add__() # 加法
__mul__() # 乘法
__abs__() # 绝对值
__sub__() # 减法
__bool__() # 布尔
```

第二章

可变序列
list、bytearray、array.array、collections.deque、memoryview

不可变序列
tuple、str、bytes

列表推导

```
codes=[ord(symbol) for symbol in symbols] # 列表推导,ord变成unicode码

beyond=[ord(s) for s in symbols if ords(s) > 127]
# python支持三元运算符，可以在后面增加筛选条件

beyond=list(filter(lambda c:c>127, map(ord,symbols)))
# filter+map也可以取得相同效果
# filter返回布尔值，map表示某种映射

tshirts=[(color,size) for color in colors for size in sizes]
# 可以多重for
```

生成器表达式

```
array.array('I', (ord(symbol) for symbol in symbols))
# 和列表推导类似，但节省雷村，逐个产生严肃，而不是先产生列表在传递到构造函数

for tshirts in ('%s %s' % (c,s) for c in colors for s in sizes):
    print(tshirt)
# %r和%s就是 repr()和str()的区别
```

拆包

```
for country,_ in travel:
        print(country)
# _为占位符，当不需要使用的时候，可以这么用

a,b=b,a # 逗号运算符产生元组
# 这种写法很优雅，无需产生新对象就交换了两个变量的值，也可以它作为函数返回

t=(20,8)
divmod(*t)
# *可以把可迭代对象编程函数参数
a,b,*rest=range(5)
# *可以处理剩下的元素，剩下的数被自动归入rest中，变成一个list
# 元组拆包是可以嵌套的
```

具名元组

```
Card=collections.namedtuple('Card',['rank','suit'])
City=namedtuple('City','name country population coordinates')
# 第一个参数为类名，第二个参数是所含属性的列表
City._fields
# 返回所有属性名称
```

切片

```
s[start:end:step]
# 表示从start到end，每隔step切出来一个，其中三个参数都可以省略
# end不包含

x[i:...] # 是x[i,:,:,:]的缩写

l[2:5]=[20,30] # 必须用列表类型赋值，数量不一定相等
del l[5:7] # 删除
```

序列的+和*运算

```
l*5 # 复制5遍列表中的元素，如果元素是引用需要特别注意

board=[['_']*3 ]*3 # 实际上创建了3个引用
# 修改其中一个，会导致3初同时修改
board=[['_']*3 for i in range(3)] # 这是正确的
```

bisect和insort

```
bisect.bisect(list,value) # 默认相等值在右边
# 省略lo和hi参数，用于缩小搜寻范围

bisect.insort(list,item) # 用于在顺序序列中插入
```

memoryview 内存，用不同方法解释

第三章

字典中的元素必须是可以散列的，如列表就是不可散列的。

字典推导

```
country_code={country:code for code,country in DIAL_CODES}
# 后者为一个元组的列表

country_code={country.upper():code for code,country in DIAL_CODES
                                                    if code<66}
# 也可以加入筛选条件
```

常见的映射方法

```
d.__delitem__(k) # del k 时调用
d.get(k,[default]) # 返回键k对应的值，如果没找到就返回default
d.__getitem__(k) # 实现【】
d.keys() # 返回所有的键
d.setdefault(k,[default]) # 字典中键k设为default，但不改变原来的值，没有则创建一个
d.setitem__(k,v) # 实现d[k]=v
d.update(m,[**kargs]) # 如果m是键值对迭代器则直接更新，如果不是则看成映射
d.values() # 返回值
```

用get方法或是 `d[k]` 时，如果没有元素会报错，这时一般用 `d.get(k,default)` 来代替，给找不到的键一个默认的返回值。

```
index.setdefault(word,[]).append(location)
# setdefault如果没找到返回[]，找到返回找到的值，不会改变原来的值
```

defaultdict 处理找不到的键，可以用这个，在创建defaultdict对象的时候就创建了处理找不到的键的方法。

```
index=collections.defaultdict(list) # 参数是一个list构造方法
```

集合

```
found=len(needles & haystack)
# 运算的二者都是集合，集合求交来求公共部分的数量快速而简洁

found=len(set(needles).intersection(haystack))
# 也可以这样写，只要有一方是集合，速度就快

set()
# 空集必须这样生成，用{}会生成dict

{chr(i) for i in range(32,256) if 'SIGN' in name(chr(i),'')}
# 集合推导的写法
```

集合操作

```
s|=z
s.update(it,...) # 和上面的语句相等，集合还支持很多其他运算符

s.isdisjoint(z) # 不相交
e in s # 属于
s<=z # 子集
s<z # 真子集
s>z # 同上
s>=z # 同上
```

不要一个一个添加，应该全部生成之后，再一次update

第5章

高阶函数 接受函数为参数，或是把函数作为结果返回的函数是高阶函数。

```
def factorial(n):
    return 1 if n<2 else n*factorial(n-1)

fact=factorial # 可以把函数赋值给变量fact，体现了函数一等性

map(fact, range(11)) # 对列表中的所有元素执行映射操作

from operator import add
from functools import reduce
reduce(add,range(100)) # reduce返回执行某个操作替代sum

all(iterable) # 每个元素都是真值，返回True
any(iterable) #  只要有真值，返回True
```

匿名函数

```
sorted(fruits,key=lambda word:word[::-1])
# lambda表达式
```

可调用对象

用户可以把一个对象声明成可以调用的对象，相对于c++中把类包装成函数

```
class BingoCage:
    def __call__(self):
        return self.pick()
    # 定于了这个内置方法后，类拥有函数的行为
```

定位参数和仅限关键字参数

`*`可以对应多个无名参数，`**`可以对应多个形如`key=value`的参数对

```
def tag(name,*content,cls=None,**attrs)
    return name
# name参数后面的参数计入到content数组里，而后面的键值对记录到attrs字典中

my_tag={'name':'img','title':'Sunset Boulevard',
        'src':'sunset.jpg','cls':'framed'}
tag(**my_tag)
# 用**展开字典时，里面键值对会自动对应成key=value传入函数中，即name=img等
# 因此在这个形式下content没有值
```

函数式编程

得益于`operator`和`functools`包的支持，能够进行函数式编程

```
from functools import reduce
def fact(n):
    return reduce(lambda a,b:a*b, range(1,n+1))
# 单独用reduce，用到了匿名函数，不简洁

from operator import mul
def fact(n):
    return reduce(mul,range(1,n+1))
# 利用operator中的mul变得简洁一些

from operator import itemgetter
for city in sorted(metro_data,key=itemgetter(1)):
    print city
# itemgetter生成一个函数，itemgetter(1)实际上是取对象的第一个位置
# 相当于lambda fields:fields[1]

cc_name=itemgetter(1,0)
# 可以传多个参数，返回提取出的值构成元组，cc_name实际上是个函数，取1，0位置返回元组
# itemgetter使用[]运算符，支持任何使用__getitems__方法的类
```

```
from operator import attrgetter
name_lat=attrgetter('name','coord.lat')
# attrgetter与itemgetter类似，不过接受的参数是属性

from operator import methodcaller
upcase=methodcaller('upper')
# 与前两个类似，只是这个是取对象调用指定的方法,返回后还是一个函数
hiphenate=methodcaller('replace',' ','-')
# 多余的参数可以用于绑定参数，绑定部分参数，生成一个心函数
```

functools.partial 它可以基于现有函数创建一个新函数，新函数固定了部分参数，只接受一部分参数,和methodcaller类似

```
from operator import mul
from functools import partial
triple=partial(mul,3)
# 绑定了需要两个参数的mul其中一个参数3，使它固定成翻三倍的函数

picture=partial(tag,'img',cls='pic-frame')
# 用于之前tag函数上，生成一个专门做picture标签的函数
```

第六章

经典的策略模式

策略模式为利用一个公共的虚基类，实现不同方法的统一接口，如在电商打折时，利用 `Promotion` 做接口，具体的细分方法只需要继承这个虚基类，重写 `discount()` 方法。

```
from abc import ABC
# ABC 虚基类
class Promotion(ABC):
    # 新的虚基类继承ABC
    @abstractmethod
    def discount(self, order):
        """返回折扣金额"""
    # 利用了ABC中的装饰器，将方法装饰为虚方法
```

这个模式可以使用函数式编程的思想实现。

```
class Order:
    def __init__(self,customer,cart,promotion=None):
        self.customer=customer
        self.cart=list(cart)
        self.promotion=promotion
    ...

def fidelity_promo(order):
    """函数以order类作为参数，可以直接作为参数初始化Order"""

promos=[f_promo, bulk_promo,large_promo]
def best_promo(order):
    return max(promo(order) for promo in promos)
# 找到最佳的折扣方案

promos=[globals()[name] for name in globals if
        name.endswith('_promo') and name != 'best_promo']
# 用globals()获取全局下所有的命名，取出按照规则命名的，加入promos中
# 有可能有些函数没有按照规则命名

import inspect
promos=[func for name,func in inspect.getmembers(promotions,inspect.isfunction)]
# 利用inspect选取函数，并且在单独promotions模块中选取
```

第七章

装饰器基础知识

装饰器是可调用的对象，参数是另一个函数，处理这个函数，然后将它返回，包装成另一个函数。**装饰器会在导入模块时立即执行。**

```
@decorate
def target():
    print('running target()')

# 等同于
target=decorate(target)
```

使用装饰器改进策略模式

```
promos=[]

def promotion(func):
    promos.append(func)
    return func

@promotion
def fidelity(order):
    """"""

@promotion
def large(order):
    """"""
# 利用装饰器把函数注册到函数列表当中，只需遍历遍历该列表即可
```

变量作用域规则

即便是全局变量，如果是在函数内对它赋值，则认为是局部变量，需要先定义在使用。

```
b=6
def f(a):
    print(a)
    print(b)
    b=9
# 将会报错因为函数第三行对b赋值了，认为局部变量，需要先定义在使用。

def f1(a):
    global b
    print(a)
    print(b)
    b=9
# 如果想在赋值时作为全局变量，则需要事先声明global
```

闭包

```
def make_averager():

    # 从这里开始
    series=[]

    def averager(new_value):
        series.append(new_value)
        total=sum(series)
        return total/len(series)
    # 从这里结束，称为闭包
    # 闭包延伸到函数作用域之外，包含了series绑定
    # 调用make_averager()后本地作用域一去不复返
    # 但生成的函数会保存series

def make_averager():
    count = 0
    total = 0

    def averager(new_value):
```

```
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
# 必须使用nonLocal关键词，否则因为出现赋值，会被认为是局部变量
```

使用 `nonlocal` 关键词使得函数能够使用不在定义域内的变量，更好的利用闭包。

实现一个简单的装饰器

```
import time

def clock(func):
    def clocked(*args):
        # 接受任意定位参数，接收的参数和原来的func接收参数相同
        t0 = time.perf_counter()
        result = func(*args) # 得出结果
        elapsed = time.perf_counter() - t0
        name = func.__name__ # 得到函数名字
        ...
        return result
    return clocked
```

这个写法的缺点在于**不支持关键字参数**，因此可以利用 `functools.wraps` 装饰器把相关属性，并且能够处理关键字参数。

```
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
    # 能够接收关键词参数
```

标准库中的装饰器

`functools.lru_cache` 实现了备忘功能，是一项优化技术，把耗时的函数的结果保存起来，避免传入相同的参数时重复计算。`lru` 即 `Least Recent Used` ，缓存大小可以自己设置，最好是2的倍数。

```
import  functools
from clockdeco import clock

@functools.lru_cache() # 装饰器工厂函数
@clock
def fibonacci(n):
    """求斐波那契数列"""
# 因为使用字典保存结果，所以函数参数必须是可散列的。
```

`functools.singledispatch` 可以用于重载方法或函数，将一个普通函数装饰为**泛型函数**，即接受不同类型的参数，并相应作出不同的操作。

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch
def htmlize(obj):
    """这是处理obj的基函数"""

@htmlize.register(str)
def _(text):
    """处理text"""

@htmlize.register(numbers.Integral)
def _(n):
    """处理整数"""
```

```
@htmlize.register(tuple)
@htmlize.register(abc.MutableSequence)
def _(seq):
    """序列"""

# 装饰之后的装饰器有了register方法，register接受的参数即为各种类型
# 装饰器装饰的函数不需要名字，可以用_()简化
# 装饰器可以叠放，使得同一函数接收不同参数
```

参数化装饰器

```
def register(active=True):
    def decorate(func):
        if active:
            """利用传入的参数"""
        else:
            """进行不同的处理"""
        return func
    return decorate

@register(active=false)
def f1():
    """函数一"""

@register()
def f2():
    """函数二"""

# 接收参数的装饰器实际上是个装饰器工厂函数，接收参数返回真正的装饰器
```

第8章

标识、相等性、别名

```
charles = {'born':1832}
lewis = charles
lewis is charles # is判断是否一样的，和__equ__不同
id(lewis), id(charles) # id()函数返回内存地址，进一步说明是否一样
```

所以 `lewis` 和 `charles` 实际上是同一对象的两个别名，对他们的操作都是对对象的操作。 `is` 和 `==` 不同，前者会比较的是标识，后者比较的是值，如果要看是不是 `None` 前面的操作符更快，因为不能重载。

元组的不可变是相对的。

```
t1 = (1, 2, [30, 40])
t1[-1].append(99)
# 元组中不可变的是对象和他的标识，并不是对象的内容不变
# 列表是可变对象，因此可以执行 `append()` 操作。
```

默认浅复制

`Python` 中的复制默认是浅复制，只复制元素引用，节省空间。这样在元素是不可变对象的时候非常节省空间，但是如果是可变对象，如列表的时候，就会出现问题。

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)
# 此时 l2==l1 但 l2 is not l1

l1.append(100) # l1=[3,[66,55,44],(7,8,9),100]
l1[1].remove(55) # l1=[3,[66,44],(7,8,9),100]
l2[1] += [33,22] # l2=[3,[66,44,33,22],(7,8,9)]
l2[2] += (10,11) # l2=[3,[66,44,33,22],(7,8,9,10,11)]
```

最终 `l1=[3,[66,44,33,22],(7,8,9),100]` 以及 `l2=[3,[66,44,33,22],(7,8,9,10,11)]` 这是因为保存了引用，`l1` 和 `l2` 虽然不是同一列表，但元素是对同一列表 `[66,55,44]` 的引用，所以对这个列表的修改，会同时影响二者，同时列表的 `+=` 运算符就是 `appen()` ，而元组的 `+=` 运算符会生成新的元素，因此导致元组不同，而列表相同的结果。

在**校车类** `bus` 中，保存一个列表，如果使用浅复制就只会复制**列表**的引用，那么对实例 `bus1` 的操作也会影响到 `bus2` ，因此需要使用深复制。

```
import copy
bus2 = copy.copy(bus1) # 浅复制
bus3 = copy.deepcopy(bus1) # 深复制
```

引用传参

`Python` 的传参方式是共享传参，各个形式参数获得实参中各个引用的副本，因此内部形参实际上实参的别名。下面这个示例说明传参带来的问题。

```
def f(a, b):
    a += b
    return a

x = 1
y = 2
f(x, y) # 3
x, y # (1,2), 由于是不可变对象，因此a+b生成了一个新对象分配给a

a = [1, 2]
b = [3, 4]
f(a, b) # [1,2,3,4]
a, b # ([1,2,3,4],[3,4]), 列表是可变对象，+=会直接append到a后面

a = (1, 2)
b = (3, 4)
f(a ,b) # (1,2,3,4)
a, b # ((1,2),(3,4)), 元组不可变，+=生成新对象
```

不要使用可变类型作为参数默认值

```
class HauntedBus:
    def __init__(self, passengers=[]):
        self.passengers = passengers
```

这样写会带来问题，因为函数的默认值是在定义函数时计算的，会变成函数对象的属性，因此如果默认值是可变对象，而且修改它的值，会影响到后续的函数调用。最好的办法是将默认参数写成 `None` 然后再函数体中进行判断。

```
class TwilightBus:
    def __init__(self, passengers = None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = passengers

basketball_team = ['Sue', 'Tina']
bus = TwilightBus(basketball_team)
bus.drop('Tina')
basketball_team # ['Sue']
```

即便是这样写，还是有很大风险，因为是引用传参，对内部的修改会影响到原来的参数，可以看到原来 `basketball_team` 中的人**消失了！** 为了避免避免这种情况，只需要改成：`self.passengers = list(passengers)` 。

第9章

classmethod

`classmethod` 装饰器定义的是类方法，和实例方法不同，实例方法作用于类的实例，而类方法作用于类型本身。

```
class Demo:
    @classmethod
    def klassmeth(*args):
        return args

Demo.klassmeth()
# 是在Demo类型上使用的方法
```

可散列的类型

要是一个类的实例可以被散列，首先要求类是不能被修改的，即不可变。因此我们需要 `@property` 装饰器，把读值方法标记为特性，不能对其赋值。

```
class Vector2d:
    typecode = 'd' # 这是一个类属性

    def __init__(self, x, y):
        self._x = float(x)
        self._y = float(y)

    @property # 把读值方法标记为特性读值方法与公开属性同名
    def x(self):
        return self._x
    ...
```

上面的 `typecode` 是类属性，要修改类属性，必须在类上修改，不能在实例上修改。

```
Vector2d.typecode = 'f'

class ShortVector2d(Vector2d):
    typecode = 'f'
# 经常用于继承时定制类的数据属性
```

第10章

如果是多维的向量，我们需要同时取 `xyzt` 四个方向的分量时，按照前一章的写法，需要写四次 `@property` 非常麻烦，因此我们可以使用 `__getattr__` 特殊方法，属性查找失败后，会调用这个方法。

```
# 还在类的定义体中
shortcut_names='xyzt'

def __getattr__(Self,name):
    cls= type(self) # 获取类型

    if(len(name)) == 1:
        pos = cls.shortcut_names.find(name)
        if 0 <= pos < len(self._components):
            return self._components[pos]
```

光这样写是不够的，错误会出现在给 `v.x` 赋值的时候，因为对象本身没有 'x' 这个属性，因此赋值时，会添加这个属性，并且以后每次进行访问的时候，都会直接访问新创造的属性。因此我们要利用 `__setattr__()` 特殊方法。

```
def __setattr__(self,name,value):
    if len(name) == 1:
    """如果设置的属性为我们要存取的v.x之类的，方法不同"""
    ...

    super().__setattr__(name,value)
    # 如果不是，则使用默认行为，即在超类上调用
    # 非常重要
```

可迭代的对象、迭代器和生成器

典型的迭代器

典型的迭代器实现了 `__next__()` 和 `__iter__()` 方法。

```
# 这是在sentenceIterater的定义体里
def __next__(self):
    try:
        word = self.words[self.index]
    except IndexError:
        raise StopIteration()
    self.index += 1
    return word

def __iter__(self):
    return self
```

迭代器第三版

在这个版本中，可以不单独在前一个版本中实现迭代器，并写 `__next__()` 方法，而是通过生成器函数，或者是生成器表达式完成。如：

```
# 注意这是在sentence的定义体里
def __iter__():
    for word in self.words:
        yield word
    return
    # 这样定义__iter__()方法，就可以了
```

这样就不需要单独顶提供一个迭代器类了！

生成器工厂

上面的生成器函数返回一个生成器对象，生成器函数就会变成一个生成器工厂。

```
def f():
    yield 1
    yield 2
    yield 3

it = f() # it即为生成器
next(it) # 1
for i in f():
    # 可以放到for循环里
    pass
```

惰性实现

惰性与急切相对，即只在需要的时候返回值，比如生成器，调用next方法每次返回一个值。在之前的sentence的实现里，一次性先把文本全部处理，再传入迭代器函数中，这不符合惰性原则。 `re.finditer` 是 `re.findall` 的惰性版本，返回的不是列表，而是一个生成器，按照需求生成 `re.matchObject` 因此可以节省大量内存，我们可以让类变得懒惰，仅在需要时生成下一个单词。

```
# 这是在sentence的定义体中
def __init__(self, text):
    self.text = text

def __iter__(self):
    for match in RE_WORD.finditer(self.text):
        yield match.group()
```

用生成器函数产出等差数列

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin +step * index
```

除了用以上生成器函数产出生成器外，还可以使用 `itertools` 模块，这个模块提供了19个生成器函数，结合起来使用能实现很多有趣的用法。比如 `itertools.count` 能生成无穷多个数，我们提供可选的 `start` 和 `step` 值。不过 `itertools.takewhile` 函数则不同，会生成一个使用另一个生成器的生成器，在制定条件计算结果为false时停止 。 `gen = itertools.takewhile(lambda n: n<3, itertools.count(1, .5))`

标准库提供的生成器函数

```
# 用于过滤
itertools.compress(it, select_it) # 并行处理两个可迭代对象，后者中元素为真，则产出前者中对应的元素
itertools.dropwhile(predicate, it) # 按照predicate丢掉满足要求的元素
filter(predicate, it) # 留下满足要求的元素
itertools.takewhile(predicate, it) # 返回真值则产出，否则停止

# 用于映射
itertools.accumulate(it, [func]) # 类似于reduce但是在生成器水平上的，并且返回生成器
enumerate(iterable, start=0) # 产生有两个元素组成的元组，结构是index,item，index从start开始计数
map(func, [it1,it2...]) # 后面的it都是func的参数
itertools.starmap(func, it) # 后者的每个元素以 *it的形式传入func

# 用于合并
itertools.chain(it1,it2) # 无缝链接
itertools.product(it1,it2) # 笛卡儿积
itertools.zip(it1,it2) # 并行从输入的可迭代对象中获取元素，直到最短的为止

# 将输入的各个元素扩展成多个输出元素的生成器
itertools.combinations(it, out_len)
itertools.count(start=0, step=1)
itertools.cycle(it) # 按顺序重复不断
itertools.repeat(item) # 一直永续的产出同一个元素，除非提供数量

# 分组
itertools.groupby(it, key=None) # 默认按值相等分组
```

iter函数不为人知的用法

```
def  d6():
    return randint(1,6)

d6_iter = iter(d6, 1)
# 持续调用d6，知道到标记值1为止
```

第十五章 上下文管理器和else块

无处不在的else语句块

除了有经典的 `if/else` ，其实还有 `for/else` `while/else` `try/else` ，他们的行为如下：

- for，仅当循环运行完毕才执行
- while，仅当因为条件为假而退出时，才执行
- try，仅当没有异常抛出时才执行

上下文管理器和with

上下文管理器对象的存在目的是管理with语句，上下文管理器协议包含 `__enter__` 和 `__exit__` 两个方法。with语句开始调用时在上下文管理器对

象上调用 `__enter__` 方法，运行结束时调用 `__exit__` 方法。

```
with open('mirror.py') as fp:
    src = fp.read(60)
```

`fp` 实际上是个上下文管理器对象。