fluent python 学习笔记

Author：Laurent Luo　Last Modity：2021.07.21

笔记Ref：https://hellowac.github.io/programing%20teach/2017/06/01/fluentpython11.html

- 强类型、弱类型语言的区别：是否存在较多隐式转化类型
  - C++，Java，python 是强类型（不会在字符串和数字之间进行类型转换）
  - Perl，JS是弱类型
- 静态类型与动态类型
  - 编译时检查类型的语言是静态语言，因此静态类型需要声明类型
  - 运行时检查类型的语言

一、python数据模型

本章主要介绍python对象中常见的特殊方法，通过自定义特殊方法，可以自定义类对象实现内置类型对象的某些功能

1）类的特殊方法方法

```
class A():

  def __new__(self):
    创建类实例对象
  def __init__(self):
    初始化类的实例对象

  def __len__(self):
    计算类的长度，由特殊方法len调用
  def __getitem__(self,item):
    模仿字典的元素获取方法，可以由dict[]调用
  def __contains__(self, item):
    判断item是否在类的属性数据中，由in运算符调用
  def __iter__(self):
    类的实例为可迭代对象，实现迭代器，生成器的属性（for）
  def __repr__(self):
    定义对象的字符串表现形式，表达如何使用代码创建出这个对象，str.format，repr 调用
  def __str__()self:
    返回的字符串对终端更加友好，str调用
```

2）常见特殊方法方法

2.1） 打印类对象的两种方式

repr / str 前者便于调试，后者用于终端打印

- repr 用于生成大型数据结构的简略字符串表示，用于调试、记录日志。必要时通过规范方法省略部分值（使用reprlib 输出有限个类实例的分量值）

2.2）类的运算符

- 加减乘除
- 增量赋值
- 位运算

2.3）自定布尔值（bool）

- 一般情况下，自定义类的实例为真

3）python中常用特殊方法（详见书本P11）

- 与运算符无关的方法
  - len 是特殊方法
  - repr
- 运算符相关方的特殊方法（实现算术运算、位运算、比较操作）

二、序列构建数组

1）常见的序列及分类标准

1.1.1）可变序列/不可变序列

可变序列类继承了不可变序列类的部分特性，不可变序列对象的值修改等于构建一个新对象。`id()` 查看不可变对象改变后内存地址是否改变

- `list/ byterarry/array.array/collections.deque/memoryview`
- `tuple/str/bytes`

1.1.2）扁平序列/容器序列

- 容器序列存放其包含对象的引用`list \ tuple \ collections.deque`
  - （体积更小，速度更快）主要存放原子性数据
- 扁平序列存放的是值（理解为一段连续的内存，一旦修改就是重新构建）`str\bytes\bytearray\memoryview\array.array`

2）序列对象的两种常用构建方法

两个便捷的方法，替代了lambda + map/ lambda + filter

2.1）列表生成式，生成列表：

- 生成笛卡尔积（）

2.2）生成器表达式，生成字典、元组：

- 笛卡尔积`itertools.product()`

3）特殊的不可变对象元组

1.元组对象是不可变类型，但是内部是可以存放可变类型数据的

2.元组的在记录数据值的同时，保存位置信息

3.1）元组常用功能

- 平行赋值 `a,b = 11,12`
- 元组的拆包
  - 函数方法的返回值是元组`a,b,c = ('1','2','3')`
  - 顺序读取参数，位置参数的入参读取利用拆包特性`a,*b,c = range(4) -> (0,[1,2],3)`
- 具名元组：常用于没有方法只有属性的类构建
  - `nametuple('Card','suit rank'`构建记录扑克牌类，记录花色和数字

3.2）元组的计算

元组作为不可变类型，不支持对自身内部元素进行增、删。但支持修改和索引

4）序列的切片/赋值

- 切片和区间的索引都不包含范围内的最后一个元素
  - 即使只有一个位置信息，也可以看出元素数量和区间
  - 快速计算去区间长度
  - 利用任意下标切片时候不重叠

5）序列的+、 *操作

当序列中元素是不可变类型时，操作构建全新的序列。因此对于不可变序列进行重复拼接，效率会很低，因为每次都要创建新的对象

5.1）利用`+`进行序列拼接

5.2）利用`*`进行序列初始化：利用`*`初始化可能导致多个对象指向引用。

6）序列的增量赋值

- 修改tuple中可变类型对象（对tuple嵌套的list进行拼接）： 返回正确结果，也会抛错

7）增量赋值

- python 字节码
- 不要在不可变对象中放入可变对象

8）list.sorted 和sorted

8.1）`list.sorted`对列表就地排序

8.2）内置函数sorted：（详见函数式编程和高阶函数）

- 创建一个新列表
- 关键参数key，高效且提供数字与字符的混合类型列表的排序
  - 两个元素比不出大小时，每次排序的结果的相对位置是固定的
- reverse

8.3）利用`bisect`库对有序序列进行插入和查找操作

- 搜索`haystack`中`needle`值的位置
  - `bisect.bisect(haystack, needle)`
  - 序列中存在与`needle`值相等的元素，可以设置左插入和右插入
- `bisect.insort(seq, item)`
  - 将变量`item`插入`seq`中

8.4）sorted 的内核算法

- Tomsort：根据原始数据的顺序特典交替使用插入排序和归并排序

9）其他数据结构

- array数组：将数组转换成为字节而不是flaot对象：
  - `array.array(typecode, sroted(list))` 现在使用pickle.dump
  - 快速读取只包含数字的列表、文件（浮点数、双精度浮点数）且占用内存小
- dequeue 双端队列
- 频繁对数据进行先进先出的操作
- Memoryview 内存视图
  - python的内置类，在不复制内容情况下，操作同一个数组的不同切片（用不同方式读写同一块内存）
  - 理解成泛化的`numpy`数组
- 双向队列 dequeue（原子操作和资源锁）
  - rotate： 队列最右边的n个元素会被移动到队列的左边
  - append（append 操作后，再添加元素会把上一次append的元素（尾部元素踢出）
  - appendleft
- 基于线程安全性的Queue
  - queue
  - multiprocessing
  - asyncio

- heapq

三、字典和集合

1）字典、集合的高效实现：散列表

标准库中所有的映射类型都是通过`dict`实现，因此映射的键必须要是可散列的(hashable)

1.1）常见可散列类型

- 可散列类型：str、bytes、frozenset

1.2）散列值获取

- 散列值通过`id()`函数返回

1.3）可散列对象要求

- 支持`__hash__`，且通过哈希函数处理的哈希值不变
- 通过`__eq__`检测相等性
- `a==b 为真，hash(a) == hash(b)为真`

1.4）字典构建

- 字典快速构建：字典推导
  - `{k:v+1 for k, v in ddd.items() }`
- 字典的比较调用`__eq__`方法比较键值

2）常见映射方法

2.1）字典更新

- `dict.update(**kwargs)`

2.2）字典元素删除

- `dict.pop(k, [default])` 返回键K对应的值，如果不存在返回None 或者 default

2.3）字典元素弹性查找（设置默认值）

- `defaultdict`初始化时，设置默认值。
  - `d = collections.defatuldict(list)`
  - `defaultdict['key']` 字典首先调用类方法`getitem`
  - `defaultdict.get() get`方法只会返回`none`
  - default_fault
- `Userdict`，重载`__missing__` 方法（getitem 调用）
  - 重载`missing`/ 增加 `isinstance` 判断
  - 重载`contains`
  - 重载`getitem/get 方法`
- 对普通字典显示`get(key, [default])`设置缺失值（效率较低）

- `dict.setdefatul(key, []).append(value)`
  - 设置字典查找不到key时候，生成list，并将value 加入字典

3）字典的变种

- `orderdict`：添加键值时会保持顺序，因此popitem 默认删除返回字典的最后一个元素
- `chainmap`：容纳数个不同映射对象，主要用于给嵌套作用域的语言做解释器
- `counter`：计算键值出现次数
  - `counter.values`

4）用于用户继承的类Userdict

`Userdict`不是dict的子类

4.1）`Userdict`的特殊属性`data`，特殊方法`setitem`：

```
class custom_dic(Userdict):
  def __missing__(key):
    pass
  def __contains__(self, kye):
    return str(key) in self. data
  def __setitem__(self, key , item):
    self.data[str(key)] = value
```

4.2）继承的是mutablemapping

- mutablmapping.update：**可以让构造方法利用传入的各类型的参数（其他映射、元素是(key,value)的可迭代元素和键值参数）来创建实例**
  - 利用setitem 来添加新的值，实现各类型参数的实例创建
- 继承`mapping.get`不需要重载

5）不可变映射类型

- 不希望用户修改映射内部的信息
- 提供只读属性`from types import MappingProxyType`

6）集合

6.1) 使用集合的优势

- 查找的高效性
- 已经实现了很多中缀运算符，处理集合交并集合
  - a+b
  - a|b
  - a-b
  - a&b

6.2）集合构造

- 利用字面量构造集合`s = {1,2,3}`
- 集合推导（字典推导的延伸）
  - { i+1 for i in }
- dis.dis 反汇编打印字节码的操作对比

6.4）集合的相关操作

- 数学运算（生成新集合/就地修改）
  - 交集、并集、异或、对称差集
- 比较运算（返回布尔）
  - `in/ > / <`
- 实用性方法
  - 增/删/清除 `add(e)/ discard(e)/pop(e)/remove(e)`

6.5）基于散列表的dict 与 set 的优势

- dict 和 set 的效率测试
  - 测试用例：1个含有1000万浮点数的数组A，从A抽取500个元素，并随机生成500不在A的元素，构建还有1000个浮点数的数组B
    - 含有相同数量元素的list/dict/set 搜索一个元素的时间

```
"""
在10000大小
    列表中花费时间0.10844278335571289,
    字典中花费时间0.000125885009765625,
    集合遍历查找花费时间0.0001442432403564453,
    集合合并查找花费时间4.00543212890625e-05
"""
```

    - 数量10倍递增的list 搜索元素花费时间/ dict 和set 的增长系数

```
"""
在10000大小
    列表中花费时间0.10844278335571289,
    字典中花费时间0.000125885009765625,
    集合遍历查找花费时间0.0001442432403564453,
    集合合并查找花费时间4.00543212890625e-05

在100000大小
    列表中花费时间1.131382942199707,
    字典中花费时间0.0002880096435546875,
    集合遍历查找花费时间0.00020313262939453125,
    集合合并查找花费时间4.696846008300781e-05
    列表的查找之间增长系数为9.961, 10.432994314509214
```

```
    字典的查找之间增长系数为1.962825278810409, 2.287878787878788
    集合的查找之间增长系数为2.1530249110320283, 1.4082644628099175
    集合的合并操作的增长系数1.1748251748251748, 1.1726190476190477

"""
```

- 使用遍历搜索和合并操作在set中搜索的效率比

```
"""
在100000大小
    列表中花费时间1.131382942199707,
    字典中花费时间0.0002880096435546875,
    集合遍历查找花费时间0.00020313262939453125,
    集合合并查找花费时间4.696846008300781e-05
    列表/字典：3928.281456953642,
    列表/集合 5569.6760563380285,
    字典/集合 1.4178403755868545,
    遍历/合并4.324873096446701
"""
```

6.6）为什么是键值是无序的

- 散列表的单元数据结构：散列表的单元结构为表元，表元分为两部分
  - 一部分存储键的引用、一部分存储值的引用
  - 表元之间不存在有序关系，通过计算哈希值的偏移量读取表元
- 键值的计算
  - 在计算键值时，数值上连续的键值使用`hash`方法处理后的哈希值完全不一样
  - 哈希值计算时候会随机“加盐”
- 散列冲突：在增加偏移量使用位数

7）字典、集合对象的特点

7.1）空间换时间

7.2）查询速度快

7.3）键的次序取决于添加的顺序，且新键的添加可能会改变已有键值的顺序（解决散列冲突）

7.4）不要对字典同时进行迭代和修改（基于7.3））

四、文本与字节序列

文本格式：

- str unicode 编码字符串，不可变变量
- bytes 字节编码，不可变
- bytesarray 字节数组，可变
- memoryview 内存内存映射，直接操作内存二进制

1）字符和字节的转换

1.1）字符在python中的标识：unicode 格式的码位

- 字符：A / unicode 码位：U+0041
- 字符：欧元符号/ unicode 码位：U+20AC

1.2）编码及解码

- 码位转换成字节序列的行为：编码
- 字节序列转换成码位的行为：解码
- A（U+0041）的码位编码成单个字节 `\x41`，而在 UTF-16LE 编码中编码成两个字节 `\x41\x00`。
- 欧元符号（U+20AC）在 UTF-8 编码中是三个字节——`M`，而在 UTF-16LE 中编码成两个字节：`\xac\x20`

1.3）字节（二进制序列）

- 数据类型：bytes，bytesarray
- 元素（介于0-255）之间的整数，因此也叫整数序列
  - 各个字节的值的表现方式
    - 可打印的ASCII 范围内的字节
    - 制表符、换行符、回车符、\对应的字节
    - 其他字节 ： 使用十六进制转义的序列
- 二进制序列数据对象切片返回同类型序列（与字符串序列切片不同）
  - `bytes[0] != bytes[:1]`
  - `str类型 str[0] == str[:1]`

1.4）双API模式

标准库中处理字符串方法，能够接受字节序列作为参数，展现不同行为

- os 和re 两个模块能接受 字符串和字节序列作为参数
  - re模块
    - r'\d+' 匹配泰米尔数字+ASCII数字
    - rb'\d+' 只能匹配ASCII字节中的数字
  - os模块：模块中所有的函数、文件名或者路径名参数可以使用字符串也可以使用字节序列
    - 自定义处理文件名和路径名的函数：fsencode(filename)/ fsdecode(filename)
    - 使用surrogateescape处理鬼符

3）字符与内存视图

3.1）Memoryview 类对象：目的是提供内存共享的功能，

- 二进制数据对象是不可变类型，因此我们希望在不重新复制字节序列的情况下（就地修改）
- 常用的处理内容：打包的数组和缓冲中的数据切片

3.2）struct 对象：将字节序列转换/反向转换成不同类型字段组成的元组

```
import struct
fmt = '<3s3sHH' # 设计结构体格式完成后续对memoryview 对象进行拆包
with open as f:
  data = memoryview(f.read())
struct.unpack(fm, data)
```

4）编码问题

4.1）编码、解码类型设置

- `encoding`：
  - 主要在字符串编码、字节序列编码、文件读取等方法中使用
  - `open()` 、`str.encode()` 、`bytes.decode()`
  - 常用编码方式`utf-8`
- `decoding`

4.2）码位的在内存中的存储方式（以字节方式存储）

- py3.3之前可以编译CPython时可以配置内存使用16位还是32位存储各个码位。
  - 16位是窄构建
  - 32位是宽构建
- 3.3后，在创建str对象时，解释器会检查字符然后为字符选择内存布局。如果字符都在latin1字符集中，就是用一个字节存储。或者根据具体自负选择2个或4个字节存储。

4.3）编码、解码异常

- `UnicodeEncodeError`：将文本编码成字节序列时候抛出
  - str.encode('utf-8', errors='')
  - 关于error 处理有三种常用方法：1）ignore 2）replace 将无法编码的字符替换成？3）xmlcharrefreplace 将无法编码字符替换成xml实体
- `UnicodeDecodeError`：将二进制序列转换成文本时候
  - errors = 'replace'
- `SyntaxError`：python源码文件编码错误
  - py3默认utf-8，而py2 使用ASCII，因此如果在py模块中包含utf-8之外的数据且没有生命编码，会抛出异常

4.4）编码类型判断工具

4.4.1）Chardet 工具检测未知字节序列

在某些通信协议和文件格式中，没有指明内容编码类型（可能字节流不是ASCII，字节流含有大于127的值），如果使用错误编码类型会导致乱码的情况。

5）CPU如何读取字节流（字节序列标记BOM）

- 小字节序列：码位最低有效序列在前
  - 其中 `b'\xff\xfe'` 就是BOM, 指明编码时使用 Intel CPU 的小字节序。 转换为列表：`list(u16)` => `[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]` , 其中 `E` 编码为 `69 0` , 为: `0 69`
- 大字节序列 `编码顺序是相反的`
  - UTF-8 不管使用大字节还是小字节序，生成的字节序列是一致的，因此不需要BOM
  - UTF-16 含有显示指明使用大小字节序的编码格式（UTF-16LE，UTF-16BE）

6）三明治模式

python 处理文本文件遵守unicode 三明治模式。尽早将字节序列解码成字符串，中间层只处理字符串，输出尽晚将字符串编码成字节序列。

7）Python IO 过程中使用需要设置编码类型的模块

- 标准IO文件的编码解码设置：stdout\stdin\stderr
- 打开文件、重定向到控制台（std）的默认编码： locale.getpreferredencoding()
- 二进制和字符串之间转换：sys.etdefaultencoding()
- 编解码文件名：sys.getfilesystemencoding


```
import sys, locale

expressions = """
        locale.getpreferredencoding()
        type(my_file)
        my_file.encoding
        sys.stdout.isatty()
        sys.stdout.encoding
        sys.stdin.isatty()
        sys.stdin.encoding
        sys.stderr.isatty()
        sys.stderr.encoding
        sys.getdefaultencoding()
        sys.getfilesystemencoding()
    """

# locale.getpreferredencoding() 最重要，  是打开文件的默认编码,重定向到
sys.stdout/stdin/stderr 文件的默认编码

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))
```

9）其他编码类型

- `latin1`: (即 `iso8859_1`) 一种重要的编码，是其他编码的基础，例如 cp1252 和 Unicode（注意，latin1 与 cp1252 的字节值是一样的，甚至连码位也相同）。
- `cp1252`: Microsoft 制定的 latin1 超集，添加了有用的符号，例如弯引号和€（欧元）；有些 Windows 应用把它称为“ANSI”，但它并不是 ANSI 标准。
- `cp437`: IBM PC 最初的字符集，包含框图符号。与后来出现的 latin1 不兼容。
- `gb2312`: 用于编码简体中文的陈旧标准；这是亚洲语言中使用较广泛的多字节编码之一。
- `utf-8`: 目前 Web 中最常见的 8 位编码；与 ASCII 兼容（纯 ASCII 文本是有效的 UTF-8 文本）。
- `utf-16le`: UTF-16 的 16 位编码方案的一种形式；所有 UTF-16 支持通过转义序列（称为“代理对”，surrogate pair）表示超过 U+FFFF 的码位。

8）unicode字符串规范化比较

unicode字符串比较、排序之前，要对文本进行规范化解决字符串中组合字符的情况。

8.1）规范化范式流程

- 1）将文本进行统一格式的规范化（NFC）
- 2）对不区分大小写的比较实用casefold 处理文本
- 3）对于极端规范化处理：去掉变音符号等
    - 利用unicodedata.combining 判断组合记号

8.2）使用`unicodedata.normalize`规范化字符串

- 参数：NFC是用最少的码位构成等价字符串（推荐方法）
- 参数：NFD将组合字符分解成基字符和单独组合字符
- 参数：NFKC、NFKD 更严格的规范化形式
    - NFC和NFD可能无法解决兼容字符的问题
        - NFC会将欧姆规范为大写希腊字母欧米茄，打印出的视觉效果相等，但是比较时不想等
    - 尝试将各个兼容字符替换成一个或多个兼容分解的字符，但有可能会导致格式损失。

8.3）小写折叠方法`str.casefold() / str.lower()`：

- 处理只包含latin1 字符的字符串 str.casefold() 和 str.lower()结果一致（除了微符号）
- 整理来说还有116个码位无法覆盖

8.4）归一化极端情况

- 对于拉丁字母中的组合记号
- 变音符号
- 西文印刷字符

9）文本排序

- unicode字符串利用码位排序
- 非ASCII 字符序列排序可能需要设置排序规则

9.1）常用方法

- `locale.setlocale`
- `pyuca.Collator` 设置unicode排序算法

10）unicode数据库

unicode提供了完成的数据库，可以获取字符元数据进行比对查看。

```
import unicodedata
unicodedata.numeric()
unicodedata.isdigit()
unicodedata.name()
```

五、函数对象

1）函数式编程的概念

1.1）高阶函数：接收函数作为参数，并将函数作为结果返回

- sorted
- max、min
- functools.partial
- functools.reduce
  - `reduce(fn,[1,2,3,4]) -> res1 = fn(1,2) -> fn(res1,3)`

1.2）使用列表推导和生成器推导替代`map\filter\reduct`

2）归约函数

将序列和有限的可迭代对象变成一个聚合结果。

2.1）常用归约函数

- `sum(iterable)`
- `any(iterable)`
- `all(iterable)`

2.2）operator 模块实现了所有的中缀运算符，来避免使用lambda

3）可调用对象

可调用对象能够利用协议`()/`进行调用，利用`calllable`判断是否可以调用

3.1）常见可调用对象

- 用户自定义的函数：由def 或 lambda 表达式创建
- 内置函数
- 内置方法
- 方法
- 类
- 类的实例
- 生成器函数

3.2）用户自定义可调用类型

- 关键类方法：`callable`
- 调用的本质：在类型内部维护一个状态，在调用之间可用
  - 设置类的属性
  - 利用装饰器
  - 利用闭包

3.3）函数内省

- 用于检查函数对象的属性:`dir()`
- 只读协议符 `__get__` 属性描述符
- 函数名`__name__`
- 函数签名信息

3.4）函数定位参数、关键字参数的设置与捕获

```
def tag(name, *content, cls =None , **attrs)
# name` 显示定位参数
# *content 第一个参数的后面任意个参数会被content捕获
# cls 仅限关键字
# **attr 字典中所有元素作为单个参数传入，同名键会绑定到对应的具名参数上面
```

- 设置支持仅限关键字参数
  - `def f(a, *, b)`b为仅限关键字参数

3.5）获取参数信息

- 原生函数方法
  - 获取定位参数和关键字参数`__defaults__`
  - 获取仅限关键字参数`__kedefaults`
  - 获取函数定义体中船舰的局部变量`__code__`
- `inspect`方法提取函数签名

- 构建`inspect.signature`对象

3.6）函数注解：为函数声明中的参数和返回值附加元数据

- 在定义函数体的时候，设置函数的输入输出数据格式，在注解中显示
- `func.__annotations__`
- 利用`signature`方法获取注解

4）支持函数式编程的包

4.1）operator模块

- Case1：阶乘计算
  - `functools.reduce + operator.mul`
- Case2：根据元组某个字段对元组列表排序
  - `functools.itemgetter(iterable, key=)`
- Case3：根据名称提取对象的属性，输出元组
  - `functools.attrgetter()`
- Case4：冻结部分参数
  - `functools.partial(callable_obj, *attrs, **kwargs)`

六、策略模式

1）经典策略模式

1.1）策略模式概念：在计算订单折扣功能中，利用不同的折扣计算策略计算订单的金额，具体折扣计算的策略是可共享、可复用的（享元策略）。

1.2）策略模式拆分为三部分：

- 上下文（订单）：将计算委托给实现不同算法的可互换组件（根据不同的算法计算促销折扣）
- 策略：赋予策略类通用属性，提供不同折扣计算方法的共同接口（抽象类）
- 具体策略：具体算法计算类

1.3）面向对象的实现方法：

- 商品类（计算商品金额）
- 订单类（保存折扣方法，购物车商品列表，策略选择结果）
- 策略基类（抽象折扣计算方法）
- 具体策略
  - 根据商品数量，客户星级，返回折扣金额
- 弊端：
  - 会在每一个新的上下文中使用相同策略时不断创建具体策略对象，增加开销。
  - 因此对于具体处理数据的策略，避免编写只有一个类的方法，然后在另一个类实现调用这个单类的接口

1.4）函数实现

- 商品、订单类不变
- 将具体策略重构成方法（享元策略）
    - 基于享元模式的策略：
        - 只需要维护一个函数列表、使用函数方法替代只含有单方法的类实例

2）命令模式

- 对于订单折扣问题，如果经常更新策略函数的情况，需要持续维护和检查函数列表；
- 解耦调用操作的对象（调用者）和提供实现的对象（接受者），在两者中间新增一个对象

2.1）具体操作

- 在二者之间放一个 `Command` 对象，
- 只实现一个方法（`execute`）的接口，调用接收者中的方法执行所需的操作。
- 这样，调用者无需了解接收者的接口，
- 而且不同的接收者可以适应不同的 `Command` 子类。调用者有一个具体的命令，通过调用 `execute` 方法执行。

2.2）sample

```
# MacroCommand 的各个实例都在内部存储着命令列表

class MacroCommand:
    """一个执行一组命令的命令"""

    def __init__(self, commands):
        self.commands = list(commands) # 构建一个列表，这样能确保参数是可迭代对象

    def __call__(self):
        for command in self.commands: # 调用 MacroCommand 实例时，self.commands 中的各个命令依序执行。
            command()
```


- 菜单驱动的文本编辑器：
    - 调用者：用户菜单（Menu）
    - 接受者：数据库（DB）、客户端（client）
    - 增加命令操作对象：command

七、装饰器

1）装饰器基础知识

- 装饰器是可调用对象，包装在被装饰对象外层
- 采用闭包机制，全局变量和局部变量的生命周期不同，实现函数作用域的延伸
- py程序运行和导入时与装饰器的交互（交互顺序详见第二十章类装饰器）
  - 对于类装饰器，首先会导入被装饰类的定义体，然后倒入装饰器（但不运行）

2）利用装饰器维护具体策略（第六章的折扣策略优化）

定义函数的外部状态变量promo，函数调用完外部变量仍存在

```
promos = []
def promo(promo_func):
  promos.append(promo_func)
  return promo_func

@promo
def bulk_item(order):
```

3）闭包和变量作用域

- **闭包指延伸了作用域的函数，访问定义体之外定义的非全局变量**
  - 保留定义函数时存在的自由变量的绑定，因此函数定义域不可用，仍能使用绑定
- Case1：计算移动平均值

```
def average():
  series = []
  def get_avg(value):
    series.append(value)
    return sum(series)/len(seris)
  return get_avg()
```

- 关键属性`__closure__`
- 对于不可变类型的自由变量，
  - 隐式创建在内部函数创建局部变量，此时的自由变量就不再是自由变量，也不会保存在闭包中
  - 使用`nonlocal`声明

4）定义标准装饰器

- 装饰器返回内部方法，内部方法接受从装饰器输入的参数，调用被装饰函数，并计算最终结果

```
def clock(func):
  def get_time(*args,**kwargs):
    res = func(*args)
    return res
  return get_time
```

5）标准库装饰器

- lru_cache：缓存装饰器
  - 实现备忘功能，将好事的函数结果保存起来，在一段时间内复用

```
@functools.lru_cache()
@clock()    # 叠放解析器，将lru_cache应用到@clock返回的函数上
def fibonacci(n):
  if n < 2:
    return n
  return fibonacci(n-2) + fibonacci(n-1)
```

  - 可选参数`maxisize（缓存中存放调用结果的上限）, typed（不同参数类型得到的结果分开保存）`
- singledidispatch 单分派泛函数
  - 将普通函数装饰为泛函数，根据不同的参数类型，用不同逻辑执行相同操作

```
from functools import singledispatch
@singledispatch
def himlize(obj):
  pass

@htmlize.register(str):
def _(text):
  pass

@htmlize.register(numbers.Intergral)
def _(n):
  pass
```


6.2）叠放装饰器

```
@d1
@d2
def f():
  pass

等价于
d1(d2(f))
```

5.3）参数化装饰器

装饰器会将被装饰的函数作为第一个参数（隐式）传入装饰器，如果需要在传入被装饰函数的同时，传入其他参数可以用到函数工厂

5.4）装饰器工厂

- 先创建一个装饰器工厂，将参数传入工厂，传出装饰器

```
var = []
def deco_factory():
  def decorator(func):
    var.append(func)
    return func        # 装饰器返回被装饰函数
  return decorator     # 装饰工厂函数返回装饰器
```

- Case2：设计能够接受不同打印格式的计时器

```
@clock(fmt)
def func():
  pass
```

八、对象引用、可变性和垃圾回收

1）标识、别名、相等性

- 每个变量都有标识、类型和值。
  - 对象一旦创建，标识是不会变的，通过 `id()` 进行查看；
  - 两个对象的标识是否相等，通过 `A is B` 查看
  - 变量的值是否相等，通过 `==` 进行比
- 对象只有一个标识、但对象可能被多个引用，因此有多个标志（别名）指向同一个对象。

```
a = {'1':1, '2':2, '3':3}
b = {'1':1, '2':2}
a == b -> True    # 值相同
a is b -> False   # 绑定了不同对象
```


2）可变、不可变类型的元素存储方式

- py集合内（list dict tuple）保存的是对象的引用。
  - 元组的不可变是相对的：
    - 元组的保存的对象可以是可变对象的引用 `tuple([1,2,3],1)`
    - 因此保存可变对象引用的元组是不可散列对象
- str、bytes、array 等单一类型序列保存对象是值，不是引用
  - 因此每一次修改实际是创建新对象

3）拷贝操作

3.1）浅拷贝：

常见的构造方法默认使用浅拷贝，创建一个副本。

- list可以通过`list()`、`[:]`实现浅复制
- 对于所有元素都是不可变类型的对象，浅拷贝和深拷贝效果一样
  - 因为不可变类型对象不共享：当对str、bytes、tuple 进行赋值运算时，会进行新建对象重新绑定给变量

```
t1 = {1,2,3}
t2 = tuple(t1)
t2 is t1  # True
t2 = t1[:]
t2 is t1 # True
s1 = '123'
s2 = '123'
s1 is s2 # True 驻存机制
```

- 但是一旦拷贝对象中有可变对象（列表嵌套列表），使用浅拷贝修改副本中的可变对象，会导致原始数据也被修改

```
a = [1,[3,4],(6,7)]
b = list(a)   # 获取副本
b[1].append(2)
print(a)    # -> [1,[2,3,4],(5,6)] a,b 保存的都是列表的索引，因此修改列表导致ab都被修改
b[2].add(5)
print(b)    # -> [1,[2,3,4],(5,6,7) 元组是不可变类型，因此修改元组后，等于新建元组对象绑定到b[2]，a不受到影响
print(a)    # -> [1,[2,3,4],(6,7)]
```


3.2）深复制

使用深拷贝得到的副本与原对象不共享内部对象的引用

- 深拷贝方法`copy.deepcopy()`

4）共享传参

4.1）函数传参策略：别名传参

- 函数中内部的形参数是实参的别名，获得实参中各个引用的副本。

4.2）如果实参中的元素是可变对象的引用（Case 幽灵巴士），

- 实参和形参共享一个引用，那么如果引用指向的是可变类型对象，并对这个对象的值进行修改，会导致形参和实参都被修改

4.3）在参数传入时候，防御可变参数：

- 对于可变类型参数，本地创建副本、避免把参数直接赋值给实例变量`inner_list = list(outer_list)`
- 使用不可变类型对象作参数默认值

5）del和垃圾回收

对象引用归零后，需要立即销毁

5.1）Cpython / python 的垃圾回收算法

- 利用引用计数，没个对象统计有多少引用指向自己

5.2）del

- `del` 删除指向对象的名称，不删除对象。
    - **结合变量的别名理解，变量每多一个别名，等于引用计数器+1，当该对象的引用计数器=0，表示名称都被销毁**
- `del`不删除对象，但是会导致对象因为引用归零，被销毁（对象也会因为不可获取（引用计数为0））

5.3）弱引用（缓存机制）解决“相互引用循环”

- 缓存中，避免对象由于被缓存引用而长期存活。因此使用弱引用，既不会增加对象的引用数量，也不会妨碍所指对象被垃圾回收。
- 弱引用返回被引用的对象，如果所指对象不在了，则返回None

5.4）弱引用常用集合：

- 利用weakref模块，记录所有实例的弱引用
    - weakValueDictionary 对象弱引用字典
        - key是对象，vlaue是对象的弱引用。当被引用对象被回收后，对应的键会从字典中移除
        - Case Cheese counter
    - weakSet 保存元素弱引用的集合类

5.5）弱引用限制

- 所指对象不可为tuple、str 及其子类
- list和dict的子类才可以作为所指对象
- set可以作为所指对象

九、python 风格对象

9.1）构造二维向量类，通过重构类的特殊方法，实现二维向量特性（Case vector类重载）

- 打印向量实例时，打印二维向量的分量值
- 打印类时，打印类的构造方法
- 实例中元素能够通过拆包获取
- 二维向量
- 类实例化
- 类的可散列`eq 和 hash`

```
class Vector2D():
  def __init__(self, *args):
    # 实例化
    self.x, self.y = args   # 二维

  @classmethod
  def frombytes(cls, octets):
    # 利用类方式，读取二维数据进行实例化
    type_code = chr(octest[0])
    memv = memoryview(octest[1:]).cast(type_code)
    return cls(*memv)

  def __iter__(self):
    # 为实例提供可迭代特性，被iter()调用
    # 能够提供拆包获取实例元素与顺序打印的功能
    return (i for i in (self.x, self.y))

  def __str__(self):
    # 被内置函数str()调用
    return tuple(self)    # print(v2) -> (3.0,4.0)

  def __repr__(self):
    fmt = '{} ({!r},{!r})'
    return fmt.format(type(self), self.x, self.y)  # Vector2D(3.0,4.0)

  def __abs__(self):
    # 计算向量模长
    return math.hyot(self.x, self.y)

  def __eq__(self, other):
    # 计算两个向量每一个分量相等
    return tuple(self) == tuple(other)

  def __bool__(self):
    # 模长不为0，即为真
   return bool(abd(self))

  def __hash__(self):
    # 实现类哈希方法，实例不可散列，是无法放入set集合中的
    pass
```

9.2）类常用方法总结

- 用于获取字符串和字节序列的表示形式的方法
  - `__repr__,__str__,__format__,__bytes__`
- 对象转换数字的方法
  - `__abs__,__bool__,__hash__`
- 测试数据相等于是否支持散列的方法
  - `__eq__`
- 类的备选构造方法`@classmethod`

9.3）classmethod / staticmethod 对比（参数）

- 与类绑定的方法，常用于定义备选狗砸方法
- 与类和类实例相关但不调用类属性的方法（只是声明在类定义体中）

```
# 正常方法
def a(self, args):
  pass

@classmethod
def a(cls, args):
  pass

@staticmethod(args):
  pass
```


9.4）格式化显示 format

- 通过内置函数`format()`调用format 和 str.format() 方法将各个类型的格式化方法委托给对象的**format**()方法
  - 如果自定义类没有实现`__format__`方法，则查看是否实现`__str__`，调用`str()`方法打印

```
format(var, '0.4f')   #格式规范微语言（）
'{result:0.4f}'.format(result=var)    # 代换字段表示法{字段名称:格式说明符号}
```

- 内置类型表示

```
format(43,'b')    # 二进制
fomat(43,'x')     # 十六进制
format(2/3, '.1%')    # 百分比
format(time(), '%H:%M:%S')    #与strftime()函数一致

class obj:
  def __format__(self, fmt_spec = ''):
    # 针对fmt_spec参数不同输入进行不同格式化的输出
    outer_fmt = "{} {}"
    components = (format(c, fmt_spec) for c in self)    # 使用各个分量生成可迭代对象
    return outer_fmt.format(components)
```

9.5）可散列对象构建的关键方法

- hash
  - 需要将类设置为不可变类型
  - 使用异或运算符计算实例属性的散列值
- eq

```
def Vector2D():
  # 构建私有属性
  def __init(self, x, y):
    self._x = x
    self._y = y

  # 将类属性声明为类特性，确保实例的散列值是不可改变的
  @property
  def x(self):
    return self._x
  @property
  def y(self):
    return self._y

  def __eq__(self, other):
    return tupe(self) == tuple(other)

  def __hash__(self):
    return has(self.x) ^ hash(self.y)
```


9.6）Py的私有属性与受保护属性

Py本身是没有私有属性概念的，因此通过名称规范，避免子类意外覆盖私有属性

- Py 的名称改写特性，在私有属性变量前添加双下划线`self.__x`

```
v1 = Vector2D(1.0,2.0)
v1.__dict__
{'_Vector2D_x:1.0'}    #Py会在属性名存入dict中，并添加下划线和类名
```

4）slot 类属性节省内存空间（直接使用Numpy）

- 类实例会将属性存入实例的类属性`__dict__`字典中，当处理大量实例时，字典会消耗大量内存
  - 因此引入`__slot__`类属性，让解释器在元组中存储实例属性，节省内存
  - `slot`不具有继承性，要在每个字类都实现一次
  - `slot`定义了类实例拥有的属性
  - 要对实例进行弱引用，需要将`__weakref__`加入`__slot__`中
  - Case：RAM测试实例内存占用情况
- 使用numpy进行大量数据操作，numpy已经实现了内存优化，避免自己实现`slot`

9.7）类属性的处理

- 根据子类需要，在子类中覆盖父类的属性

十、序列的修改、切片、散列

10.1）为自定义类添加序列特性

- 我们希望将原先只有两个坐标的Vector2D类拓展到多位向量的维度
  - 因此需要为类添加序列特性

10.2）py的协议和鸭子类型

基于Python的协议和鸭子类型，只需要在自定义类实现符合序列协议的方法

- 序列可迭代`__iter__,__getitem`
- 对序列类进行切片，索引获取`__getitem__,__len__`

10.3）自定义序列类的切片

- 我们希望对向量实例数据进行切片，并返回带有切片数据的向量类实例，而不是一个数组类型的切片结果
- 内置函数slice，具有`start,stop,step`属性和`indices`方法。
  - 对于`obj[a:b:c]`，创建`slice(a, b, c)`对象，交给`getitem`处理

```
class Vector:
  def __init__(self, *componets):
    self._components = [i for i in componentes]

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

  def __getitem__(self, index):
    cls = self.__class__
    if isinstance(index, slice):    # 切片
      return cls(self._components[index])
```

```
    elif isinstance(index, numbers.Intergral):
      return self._components[index]              # 索引
    else:
      err_msg = ''
      rasise TypeError('indices must be intergers')
```

10.4）自定义类的属性获取

- 关键类方法 `__getattr__, __setattr__` 需要一起设置，避免对象行为的不一致，调用内置方法 `getattr,setattr`

```
def Vector:
  def __getattr__(self, name):
    return getattr(self, name)
  def __setattr__(self, value, name):
    cls = type(self)
    super().__setattr__(name, value)    # 委托超类__setattr__方法，提供标准行为
```

- 实例属性索引路径
  - 首先索引实例是否存在该属性，如果不存在该属性，构建此属性。
    - 除非在字类的 `__getattr__` 中声明要委托超类提供标准行为，否则不进行超类的回溯属性查找

10.5）计算不定长度的向量实例的哈希值 `hash\eq`

- 由于向量值数量不确定，且计算结果为一个散列值
  - 调用 `map/reduce` 进行归约计算

```
functools.reduce(lambda a.b: a^b, self._components)
```

  - `xor` 异或运算（中缀运算符）则从 `operator` 库获取
  -

```
def __hash__(self)
  hashes = (i for i in self._components)
  return functools.reduct(operators.xor, hashes)
```

- 实例相等判断
  - 并列对比两个向量的每一个分量值 `itertools.zip/ itertools.ziplongest`

```
def __eq__(self, other):
  pairs = (for i in zip(self,other))
  return len(self) == (other) and all(a==b for a,b in pairs)
```

十一、协议到抽象基类

- 鸭子类型：
  - 对象类型不重要，只需要实现特定的协议即可
  - 利用动态协议（非正式接口），实现多态，给自定义类提供符合某种数据结构特性的接口
- 另一种实现方法是通过抽象基类，在实现借口时作为超类使用，可以通过抽象基类检查具体子类是否符合接口定义

1）接口与协议

1.1）接口定义

- 接口是类实现或集成的公开属性，包括特殊方法
- 即便"受保护的"属性也只是采用命名约定区分（单个前导下划线）；
  - 本质上，私有属性可以轻松地访问，按约定不要直接访问；
- 接口是对象公开方法的子集（实现特定角色方法的集合，因此一个实例可以扮演多个角色）

1.2）协议与接口的关系

- 协议是非正式接口

1.3）序列的抽象基类：

- 序列继承`Container/Size/Iterable`三个`abc`中的抽象类
- 序列接口：`iter/len/getitem/setitem/delitem/insert/contains`
- 如果没有实现iter和contains 方法，后备机制调用`getitem`方法让迭代和`in`可用；

2）猴子补丁与白鹅类型

Case：利用猴子补丁解决扑克牌类无法洗牌的情况

2.1）猴子补丁特点：

- 运行时修改类或模块，不改动源码
- 与被打补丁的程序强耦合

```
def set_card(deck, position, card):
  # 定义一个函数，参数分别为 实例对象，位置，元素（与setitem 耦合保持一致）
  deck._cards[position] = card
  # 隐含条件， 要知道 deck 对象有一个名为 _cards 的属性，而且 _cards 的值必须是可变序列
FrenchDeck.__setitem__ = set_card
# 赋值给 __setitem__ 方法， 实现接口。即 猴子补丁。
shuffle(deck)  # 可以打乱 deck 了，因为 FrenchDeck 实现了可变序列协议所需的方法。
```

2.2）白鹅类型

- 鸭子类型代表：不同类来源的对象具有相同的行为
- 白鹅类型：在鸭子类型的基础上，在类比较上作出补充
  - 只要当前类是抽象基类（继承`abc.ABCMeta`），就可以使用`isinstance(obj,cls)`

3）类的检查：

- 导入时（加载并编译模块时），Python 不会检查抽象方法的实现，在运行时实例化类时才会真正检查；
- 如果某个抽象方法没有在子类实现，py会抛出typeerror
- isinstance(obj, cls)

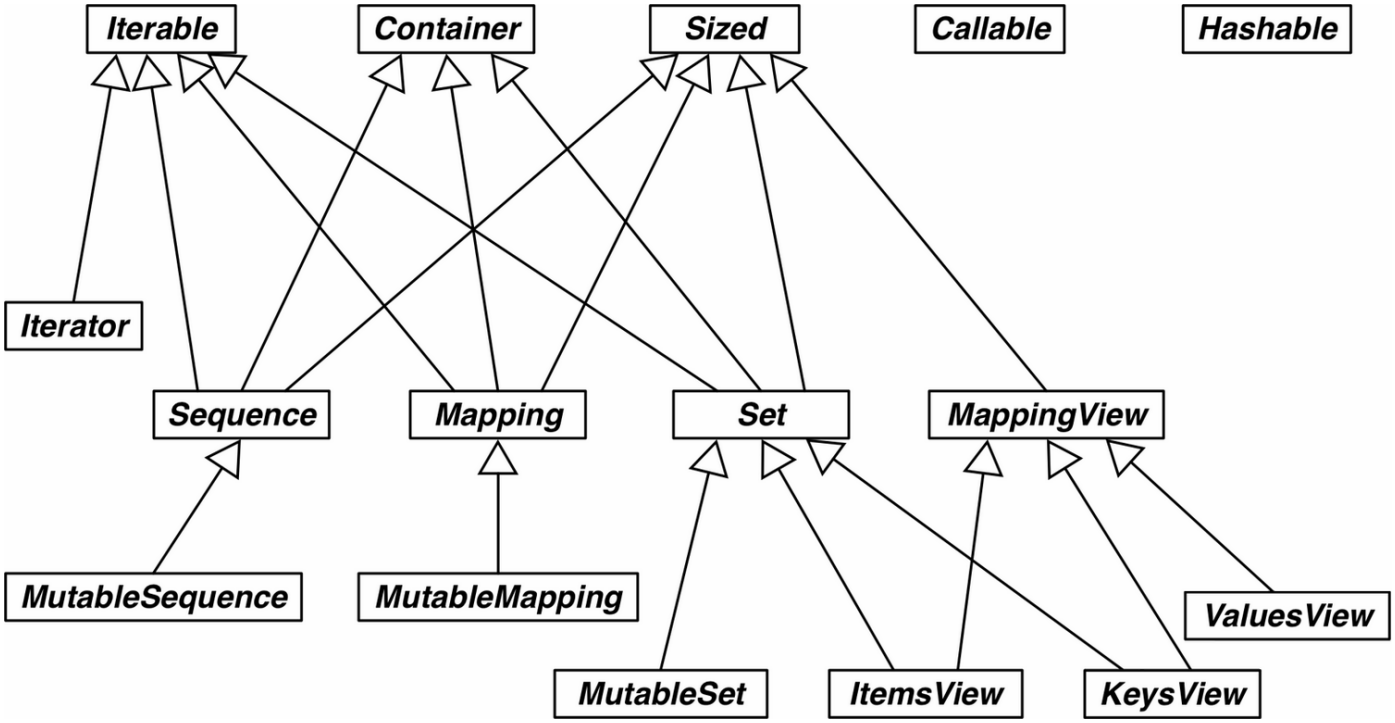
4）抽象基类的静态特性

4.1）抽象类库：

- 大多数抽象类在`collections.abc、numbers、io`；
- 抽象类的本质是几个特殊方法
  - 抽象基类的类型检查滥用`if else + isinstance`设计分派逻辑，这样会导致代码异味，
  - 建议使用多态，让解释器调用正确分派给正确方法；

4.2）`collections.abc`模块中的抽象类：

- **Case2：让扑克牌类直接继承`MutableSequence`类的相关方法**
- 1.`iterable / container/sized：`集合类都需要继承这三个类，或者实现兼容协议：
  - 通过`__iter__`支持迭代方法
  - 通过`__contains__`方法支持`in`运算符
  - `__ len__`属性支持 len() 函数
- 2.Callable / Hashable
  - 主要为`isinstance`提供安全方法，判断对象是否能够调用对象或散列
  - Isinstance(obj, hashable) / callable()
- 3.Sequence 和 mapping 和set
  - 三个不可变集合类型，延伸出可变类型子类`MutableMapping/MutableSet`
- 4.Mappingview
  - `.keys()/.items()/.values()` 是三类映射方法
  - 返回的对象分别是 `ItemsView`、`KeysView` 和 `ValuesView` 的实例。前两个类还从 `Set` 类继承了丰富的接口。
- 5.Iterator
  - itertor 迭代器的抽象基类



- 6. `number` 数字塔中的抽象类

  其中 Number 是位于最顶端的超类，随后是 Complex 子类，依次往下，最底端是 Integral 类，由抽象逐渐到具体类；

  - `Number/Complex`
  - `real`：检查是否能接受bool、int、float、fractions.Fraction、Numpy
  - `rational`
  - `integral`：检查一个数是否是整数，能否接受int，bool

4.3）子类对于抽象类方法的继承

- 抽象方法，需要在子类覆盖
- 对于部分不需要覆盖的抽象方法，使用 `__super__` 委托给 超类

5）定义抽象基类（详见代码文件）

5.1）Case：实现为用户提供随机挑选无重复的类

- 定义抽象基类 `Tombola`，继承 `abc.ABC`
  - 两个抽象方法：元素的读写
    - 抽象方法使用 `@abstractmethod` 声明

```
@abc.abstractmethod
def load(self, iterable):
# 抽象方法使用 @abstractmethod 装饰器标记，而且定义体中通常只有文档字符串
# 在抽象基类出现之前，抽象方法使用 raise NotImplementedError 语句表明由子类负责实现。

# 装饰器可以折叠，但堆叠装饰器的顺序通常很重要，@abstractmethod 的文档就特别指出：
# 与其他方法描述符一起使用时，abstractmethod() 应该放在最里层，
# 也就是说，在 @abstractmethod 和 def 语句之间不能有其他装饰器。
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```

    - 两个具体方法：容器是否为空，排序
- 具体子类
    - 子类要实现抽象基类的抽象方法
- 虚拟子类

6）抽象基类的动态特性：虚拟子类

基于白鹅类型的特性，在不声明继承的情况下，将一个类注册成为抽象基类的虚拟子类

6.1）注册虚拟子类

- 抽象基类上调用`register`（装饰器）注册虚拟子类
- `issuncalss/isinstance`都能识别
- 虚拟子类不会继承抽象基类任何方法和属性，
- 虚拟子类要实现所需的全部方法
    - 任何时候都不会检查虚拟子类是否符合抽象基类的接口（即便在实例化时也不会检查）

6.2）虚拟子类不会在方法解析顺序中显示`__mro__`

6.3）不注册实现虚拟子类

- 若抽象基类实现了特殊类方法`__subclasshook__`，可以利用鸭子类型，声明部分协议，让`subclasshook`信服，以此通过`isinstance`的类型检查
- **由于`subclasshook`不可靠，不建议自定义类去实现，了解即可**

7）类继承关系检查

7.1）真子类检查：通过类方法解析顺序属性`__mro___`，按顺序列出类及超类()

7.2）子类检查：`isinstance / issubclasses` 判断类的所属关系

- `__subclasses__`检查直接子代
- `isinstance__`检查多带

7.3）虚拟子类检查：`_abc_registry`

- 只有抽象基类具有这个数据属性，值是一个`weakset`对象，记录抽象类注册的虚拟子类的引用

十二、继承的优缺点

1）不要直接子类化C语言实现的内置类型

1.1）不建议直接子类化内置类型（list，dict 或str）

- 不会被覆盖：使用C语言实现的内置类型的方法不会调用子类覆盖的方法
  - 继承dict的子类覆盖了`getitem`，但是子类调用`get(),update()`方法时候仍会调用父类内置类型的方法

1.2）使用`collection`模块中的 `UserDict，UserList，UserString` 进行子类化

1.3）使用py语言编写的类子类化

2）多重继承（解决潜在命名冲突问题）

2.1）命名冲突（菱形问题）

- 由于不相关的祖先类实现同名方法引起

2.2）Case：PINGPONG 调用逻辑打印

2.3）多重继承的遍历逻辑

- 遍历顺序：按照方法解析顺序`mro`属性-得到类的顺序遍历继承图，避免继承冲突
  - 类一直向上直到`object`类
- 超类方法调用：
  - 正常情况下：`super`函数会按照`mro`属性给出的继承顺序调用超类方法
  - 跳级调用：显式声明要委托的超类名，传入self参数 `Father.func(self)`

2.4）多重继承的业务应用

- Tkinter 的GUI 设计
- Django

3）多重继承理论

3.1）将接口继承（定义子类是什么）与实现继承（代码复用）区分

3.2）将定义接口的类，定义为抽象类（`abc.ABC`）

3.3）声明混入类，为多个不相关的子类提供方法实现

- 功能：打包方法，封装成特定的行为，便于不相关子类复用
- 混入类不可实例化，具体类的继承不能只含有混入类
- 混入类命名、定义（详细见补充）

3.4）混入类名称需要添加`Mixin`

3.5）抽象基类可作为混入，混入类不可作为抽象基类（必须配合其他抽象基类使用）

3.6）构建子类化继承关系时，不要子类化多个具体类

```
class MyClass(Alpha, Beta, Gamma): # Alpha 可以是具体类，但是Beta，Gamma 只能是抽象基类或混入类
```

- 设计具体类超类，而不是在子类化时继承多个具体类

3.7）如果多个抽象基类和混入类能够组合成为一个特定行为

- 使用聚合类将多个超类进行结合（用户无需记住多个混入类）

3.8）对象组合替代类继承：

- 利用组合委托代替混入，将行为提供给不同的类

https://zhuanlan.zhihu.com/p/59519955

4）混入类补充

4.1）混入类定义

- Mix-in class 本质上是**代码片段**，不能独立存活。
- 增强“兄弟” [1] 类中的同名函数功能，来达到**可插拔**的效果。
  - 这里的 “兄弟”关系指的是多继承中的两个类的关系。比如 `class Child(A, B)：pass` 中类 `A` 和 `B` 就是兄弟关系
- 定义子类时，基类继承顺序不能乱，可以通过mro属性进行查询
- 混入类定义在被混入类的前面，使混入类可以通过 `super()` 方法调用被混入类的同名函数
  - 使用同名函数来替换原函数。而Mixin强调插件作用，即在原有函数实现上，增加额外功能。如 `class Strong(Week, PowerMixIn)` ，`Week` 即是 `PowerMixin` 的被混入类。

十三、重载运算符

1）运算符概念广泛：

- 中缀运算符
- 一元运算符
- 内置类型运算符
- 函数调用`()`，属性访问`.`，序列切片`[]`

2）重载运算符目的：

- 解决不同类型的操作数的运算
- 中缀运算符对于无法处理的操作数的异常处理

3）运算符详细分类

3.1）一元运算符

生成新的操作数作为返回

- +取正
  - 对应的特殊方法 `__pos__`
  - 一般情况下： `x = +x`
- -取负
  - 对应特殊方法 `__neg__`
  - 一般情况下 `x = -x`
- ~ 取反
  - 对应特殊方法`__invert__`
  - 一般情况下 `x = -(x+1)`

3.2）中缀运算符重载

- py中的运算符分为正向和反向（后备机制）
  - 正向方法只处理与实例类型相同的对象`__add__`。
  - 反向方法处理一般处理混合类型的操作数`__radd__`
    - 正向方法返回Notimplmented 时候，激活反向函数，如果仍返回Notimplemented，则抛出Typeerror，在错误消息中指出操作数类型不支持。
- 运算符中的增量赋值返回左操作数对象，而不是新创建一个操作数对象

3.3）标量加法运算符重载 +

- 常见case：
  - 向量按维度相加`itertools.zip_longest`
  - 列表拼接
- 利用正向和反向配合NotImplemented 实现后备机制

```
def __add__(self, other):
  try:
  except TypeError:
    raise NotImplemented
def __radd__(self,other):
  return self + other
```

- 解释器执行流程
  - 检查左操作数是否有`__add__`，如果有没有，会抛出TypeError，返回NotImplemented
  - 调用`__radd__`，检查右操作数是否有`__add__`，如果仍没有抛出TypedError 则指明类型错误
- 区别 Notimplemented 和 NotimplementedError，增加异常捕获
  - notimplemented 作为特殊值返回，通知启动后备方法
  - Error 是在子类继承抽象基类时，没有覆盖抽象方法时候抛出异常

3.4）重载标量乘法运算符 *

- Case：标量积和点积的区别
  - `__mul__`，乘以标量
  - `__matmul__` 点积
- 如何保证左右操作数，有对应的操作符方法?
  - 利用鸭子类型（正向、逆向），实现协议，完成运算符重载
  - 利用白鹅类型对操作数的类型进行检查，限制数据类型

3.5）其他中缀运算符及就就地方法

- 就地方法: iadd / imul

4）比较运算符

4.1）比较运算符的后备机制

- `gt`调用的反向运算符是`lt`
- `ne`是`eq`取反
- 对于`==/~=`不抛出TypeError
  - 根据左操作数`eq`方法，比较两个对象是否失败
    - 失败（左右操作数，都没有`__eq__`方法）：再对类型进行判断
    - 失败（比较`id`）

4.2）`== / !=`比较值

- 对于值相等的`list`，`tuple`是相等的

4.3）`is`比较对象标识

4）增量赋值运算符

- 不可变对象（只是语法糖）：
  - 始终创建新对象，然后重新绑定（常规运算符加上赋值）
- 可变对象
  - 使用就地修改方法（修改左操作数）
- Case：
  - BIngo类拓展成为 AddableBingo 满足与可迭代类型的计算
- 增量赋值运算符必须返回`self`：

```
class Addable:
  def __iadd(self,other):
    return self
```

十四、可迭代对象、迭代器、生成器

目标：实现惰性计算

迭代器用于支持：

- for 循环
- 构建和拓展集合类型
- 逐行遍历文本文件
- 列表推导、字典推导和集合推导
- 元组拆包
- 调用函数，使用*拆包实参

1）可迭代对象

1.1）可迭代对象的迭代器实例化方法

- 由于可迭代对象，内部实现了`__iter__`方法或实现了具有迭代特性的协议，可以调用`iter(obj)`实例化迭代器

1.2）`iter()`方法实例化迭代器的过程

- 是否实现`__iter__`方法
  - 实现了，获取迭代器
  - 没有，检查是否实现了`__getitem__`
    - 实现了，创建一个迭代器，顺序获取元素（从0开始）
    - 仍然失败的话抛出Typeerror ， not iterable

1.3）判断对象是否可以迭代

- 调用`iter()`如果抛出`TypeError`，则不可迭代
  - 只要实现了`iter/getitem`都认为是可迭代
- `isinstance(obj, abc.Iterable)`，类型检查
  - 更加严格，只检查`__iter__`方法是否实现

1.4)`iter(iterable_obj, flag)`函数的参数

- 第一个参数是可迭代对象
- 第二个值是哨符，当可调用对象返回这个值时，抛出StopIteration异常

2）迭代器

2.1）迭代器的超类

- 抽象基类`Iterable`
  - 抽象方法`__iter__`
- 抽象基类`class iterator(Iterable)`
  - 继承了`Iterable`，`__iter__`方法返回Iterator实例
  - 新增抽象方法`__next__`，由具体类实现
  - 新增类方法`sunclasshook(cls,c)`可迭代对象类型检查（白鹅类型）

2.2）迭代器定义：

- 因此迭代器可迭代，实现了`__iter__`方法，返回实例本身
- 实现了无参数的`__next__`方法，返回序列中的下一个元素
  - 当后续没有元素时候，抛出StopIteration异常

2.3）迭代器与可迭代对象的区别

- 通过`iter()`方法实例化可迭代对象，返回迭代器，**因此可迭代对象不是迭代器**
- 避免将可迭代对象构造成迭代器，生成器对象替代单独定义迭代器类

2.4）迭代器的常见用途

- 访问聚合对象内容无需暴露内部表示
- 支持聚合对象多种遍历：从同一个可迭代实例获取多个独立的迭代器，且每个迭代器维护自身内部状态。（itertools）
- 为遍历不同聚合结构提供统一接口（多态迭代）

3）生成器（Case Sentence 类）

3.1）生成器函数

- 生成器函数是生成器的工厂，定义体中包含`yield`关键字
- `yield`关键字
- 创建生成器对象，包装生成器函数的定义体

3.2）生成器与迭代器的异同

- 从接口定义来看：生成器也实现了迭代器的关键函数，因此生成器是迭代器
- 实现方式，
  - yield 是生成器的关键字，隐式区别生成器和迭代器
  - 若用显式函数声明，利用生成器函数得到的生成器对象实现了迭代器接口，因此可以理解为生成器是迭代器
- 返回内容：
  - 生成器在yield关键字处暂停，产出值
  - 迭代器`__next__`，返回下一个值
- 异常处理：
  - 生成器在函数定义体返回时，外层生成器对象会抛出StopIteration

3.3）生成器对象的运行方式

- 生成器函数（工厂）会创建创建一个生成器对象，包装生成器函数的定义体
- 把生成器传给next，函数向前执行定义体中的yeild语句，返回产出值，并在当前位置暂停
- 最终定义体返回时，生成器抛出StopIteration异常（与迭代器一致）

3.4）Case：构造生成器类 Senetence，实现惰性求值原则

-

```
def __iter__(self): # 显示yield 关键字
  for match in RE_WORD.finiter(self.text)
    yield match.group()

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

- 打印列表推导与生成器推导构造的类的数据生成顺序

3.5）使用生成器表达式和生成器函数的时机：

当函数处理逻辑代码无法通过单行实现时候，使用生成器函数

4）Case2：等差数列生成器

4.1）主要功能：

- 处理有穷与无穷等差数列
- 保持数字类型的一致（int，float，decimal，Fraction）

4.2）实现方式：

- 手动构造`start/step/end`记录属性
- 利用`itertools.count`计算等差间隔

5）标准库中常见生成器 `itertools` 库

5.1）设置 `predicte` 断言（判定条件），从输入的可迭代对象中产出子集

- `dropwhile` 跳过真值的，产出剩下的
- `filter(predicate, it)`，返回经过predicte 为真值的it的元素
- `filterfalse` 返回为假的
- `islice` 设置起点、终点和步长产出片段，配合拓展行的生成器工具进行切片
- `takewhile(predicte, it)` 返回predicate 函数得到假立即停止，返回为真的结果

模块	函数	说明
`itertools`	`compress(it, selector_it)`	并行处理两个可迭代的对象；如果 `selector_it` 中的元素是真值，产出 `it` 中对应的元素
`itertools`	`dropwhile(predicate, it)`	处理 `it` ，跳过 `predicate` 的计算结果为真值的元素，然后产出剩下的各个元素（不再进一步检查)
（内置）	`filter(predicate, it)`	把 `it` 中的各个元素传给 `predicate` ，如果 `predicate(item)` 返回真值，那么产出对应的元素；如果 `predicate` 是 `None`，那么只产出真值元素
`itertools`	`filterfalse(predicate, it)`	与 `filter` 函数的作用类似，不过 `predicate` 的逻辑是相反的：`predicate` 返回假值时产出对应的元素
`itertools`	`islice(it, stop)` 或 `(it, start, stop[, step])`	产出 `it` 的切片，作用类似于 `s[:stop]` 或 `s[start:stop:step]`，不过 `it` 可以是任何可迭代的对象，而且这个函数实现的是惰性操作

5.2）映射类：输入单个迭代对象，在各个元素上做计算，保存每一次计算的结果

- `accumulate(it, [func])`
  - 配合`sum\max\min`记录每一次求和，比较的结果
  - 如果有func 函数，则顺序将两个元素输入进行计算，将每次计算结果保存，并将结果与第三个元素继续计算。
- `enumerate(iterable, start = )`
  - 返回每一个元素与编号（`参数start设置起始`）
- `map`
- `startmap`
  - 配合`enumerate`使用，对迭代器每一个元素做可以不同处理

模块	函数	说明
（内置）	`enumerate(iterable, start=0)`	产出由两个元素组成的元组，结构是 `(index, item)`，其中 `index` 从 `start` 开始计数，`item` 则从 `iterable` 中获取
（内置）	`map(func, it1, [it2, ..., itN])`	把 `it` 中的各个元素传给 `func`，产出结果；如果传入 `N` 个可迭代的对象，那么 `func` 必须能接受 `N` 个参数，而且要并行处理各个可迭代的对象
`itertools`	`accumulate(it, [func])`	产出累积的总和；如果提供了 `func`，那么把前两个元素传给它，然后把计算结果和下一个元素传给它，以此类推，最后产出结果
`itertools`	`starmap(func, it)`可以理解为包装装成器，替代for循环+lambda+迭代器组合	把 `it` 中的各个元素传给 `func`，产出结果；输入的可迭代对象应该产出可迭代的元素 `iit`，然后以 `func(*iit)` 这种形式调用 `func`

5.3 ）合并多个可迭代对象的生成器函数

模块	函数	说明
`itertools`	`chain(it1, ..., itN)`	先产出 `it1` 中的所有元素，然后产出 `it2` 中的所有元素，以此类推，无缝连接在一起 [('0','A'),('2','B'),('3','C')]
`itertools`	`chain.from_iterable(it)` 将chain方法得到的序列，展平成一维	产出 `it` 生成的各个可迭代对象中的元素，一个接一个，无缝连接在一起；`it` 应该产出可迭代的元素，例如可迭代的对象列表 ['0','A','2','B','3','C']
`itertools`	`product(it1, ..., itN, repeat=1)` 笛卡尔积的全量组合	计算笛卡儿积：从输入的各个可迭代对象中获取元素，合并成由 `N` 个元素组成的元组，与嵌套的 `for` 循环效果一样；`repeat` 指明重复处理多少次输入的可迭代对象
（内置）	`zip(it1, ..., itN)` 多个对象同时迭代	并行从输入的各个可迭代对象中获取元素，产出由 `N` 个元素组成的元组，只要有一个可迭代的对象到头了，就默默地停止
`itertools`	`zip_longest(it1, ..., itN, fillvalue=None)`	并行从输入的各个可迭代对象中获取元素，产出由 `N` 个元素组成的元组，等到最长的可迭代对象到头后才停止，空缺的值使用 `fillvalue` 填充

- `zip`并行处理任意数量的可迭代对象

5.4）把输入的各个元素拓展成多个输出元素的生成器函数

模块	函数	说明
`combinations(it, out_len)`	out_len长度的组合	把 `it` 产出的 `out_len` 个元素组合在一起，然后产出
`combinations_with_replacement(it, out_len)`		把 `it` 产出的 `out_len` 个元素组合在一起，然后产出，包含相同元素的组合
`count(start=0, step=1)`		从 `start` 开始不断产出数字，按 `step` 指定的步幅增加
`cycle(it)`		从 `it` 中产出各个元素，存储各个元素的副本，然后按顺序重复不断地产出各个元素
`permutations(it, out_len=None)`	含有顺序信息的相同元素的组合	把 `out_len` 个 `it` 产出的元素排列在一起，然后产出这些排列；`out_len` 的默认值等于 `len(list(it))`
`repeat(item, [times])`		重复不断地产出指定的元素，除非提供 `times` ，指定次数

- `cycle`配合`islice`生成列表
- `repeat([1,2],2)-> repeat([1,2],[1,2])`限制生成器生成的元素数量
- 组合学生成器：
  - `combinations` 按照`out_len`设置的元素个数进行组合

```
list(itertools.combinations('ABC',2))
# [(A,B),(A,C),(B,C)]
```

  - `combinations_with_replacement` 含有相同元素的组合

```
list(itertools.combinations('ABC',2))
# [(A,A),(A,B),(A,C),(B,B),(B,C),(C,C)]
```

  - `permutations`：带有位置顺序的组合学生成器，不含相同元素

```
list(itertools.combinations('ABC',2))
# [(A,B),(A,C),(B,C),(B,A),(C,A),(C,B)]
```

  - `product`：笛卡尔积，全量组合，`permuation`+含重复

5.5）重新排序

模块	函数	说明
`itertools`	`groupby(it,key=None)`	产出由两个元素组成的元素，形式为 (`key`, `group`)，其中 `key` 是分组标准，`group` 是生成器，用于产出分组里的元素
（内置）	`reversed(seq)`	从后向前，倒序产出 `seq` 中的元素；`seq` 必须是序列，或者是实现了 `__reversed__` 特殊方法的对象
`itertools`	`tee(it, n=2)` 生成多个迭代器	产出一个由 `n` 个生成器组成的元组，每个生成器用于单独产出输入的可迭代对象中的元素

5.6）可迭代的归约函数

- 读取迭代器，返回单个值【只能处理最终会停止的可迭代对象】
- 配合`accumlate/map``

模块	函数	说明
（内置）	`all(it)`	`it` 中的所有元素都为真值时返回 `True`，否则返回 `False`；`all([])` 返回 `True`
（内置）	`any(it)`	只要 `it` 中有元素为真值就返回 `True`，否则返回 `False`；`any([])` 返回 `False`
（内置）	`max(it, [key=,] [default=])`	返回 `it` 中值最大的元素；`key` 是排序函数，与 `sorted` 函数中的一样；如果可迭代的对象为空，返回 `default`.
（内置）	`min(it, [key=,] [default=])`	返回 `it` 中值最小的元素；`key` 是排序函数，与 `sorted` 函数中的一样；如果可迭代的对象为空，返回 `default`.
（内置）	`sum(it, start=0)`	`it` 中所有元素的总和，如果提供可选的 `start`，会把它加上（计算浮点数的加法时，可以使用 `math.fsum` 函数提高精度）
`functools`	`reduce(func, it, [initial])`	把前两个元素传给 `func`，然后把计算结果和第三个元素传给 `func`，以此类推，返回最后的结果；如果提供了 `initial`，把它当作第一个元素传入

- 常用于排序，产生子集的迭代器函数

6）yield from 关键字

- 用于解决嵌套的生成器，获取内层生成器产出结果
- 利用生成器构造协程

十五）上下文管理器和esle

1）else 与 try / while / for 联合使用

- for 循环运行完毕，执行else
- while正常退出后，执行else
- 需要对else 子句单独做异常检测

2）上下文管理对象

2.1）上下文管理管理对象

- 背景：利用`with`语句，简化`try/finally`模式，保证一段代码运行完毕后执行某项操作
  - 即时代码异常、`return`语句和`sys.exit()`终止都会执行指定操作
- 主要任务：管理`with`语句：类似迭代器管理`for`语句
- 常见使用场景：
  - 确保文件对象关闭
  - 协程
  - 并发
- 关键方法
  - `__enter__`，返回上下文管理对象或其他对象
  - `__exit__`，`with`语句结束后，在**上下文管理器对象**上调用此方法（类似`finally`）
  - `as`可选子句，对于`open`用于获取文件的引用，但有些上下文管理器返回none

```
with open('ftp') as f:
  # 将fp绑定到打开的文件上，open类的实例的enter方法返回self
  src - f.read()

# 无论流程如何退出with 模块，都会在上下文管理器调用exit方法，而不是在enter 方法返回的对象上调用
```

- Case：利用with + 上下文管理器 + 猴子补丁，临时控制系统输出流格式
  - `__enter__`调用时不传入其他参数
  - `__exit__(exc_type, exc_value, traceback)`控制异常处理逻辑，是否向上抛出
    - 这三个参数与`try/finally`调用的`sys_exc_info()`接收的参数一致
      - `exc_type`异常类
      - `exc_value`异常实例，有时候会传参数（错误消息）给异常构造方法
      - `traceback` `traceback`对象

3）标准库中的上下文管理器模块`contextlib`

3.1）常用工具

- `closing`：调用对象的`__close__`方法，实现上下文管理器（替代`__enter__/__exit__`协议）
- `suppress`：构建临时忽略指定异常的上下文管理器
- `@contextmanager` 常用装饰器，将生成器函数变成上下文管理器
- `ContextDecorator` 上下文管理器，用于进入多个上下文管理器

3.2）`@contenxtmanager`利用生成器实现上下文管理器

- `yield`关键字将函数定义体分隔为两部分
  - 第一部分在`with`开始时候执行(`__enter__`)
  - 后一部分在`with`结束时执行(`__exit__`)
- 装饰器将函数包装成具有`__enter/__exit__`方法的类
- `__enter__`执行逻辑
  - 利用yield语句，保存生成器对象，
  - Next(obj) 执行到yield关键字所在的值，产出并返回值
  - 将返回的值绑定到with / as 语句的目标变量上面
- `__exit__`执行逻辑
  - 检查是否有异常传给exc_type ，
    - 如果有调用`throw()`，在生成器定义体中包含yield 关键字抛出异常（因此在yeild 处增加try/except 对一场进行捕获）
    - 否则，继续执行`yield`后续代码

十六、协程

使用协程在单线程中管理并发

1）作协程的生成器的基本行为

1.1）协程的主要状态：

- `GEN_CREATED`协程等待开始执行
- `GEN_RUNNING`解释器正在执行
- `GEN_SUSPENDED`在`yield`表达式处暂停
- `GEN_CLOSED`执行结束

1.2）定义体中的关键字：

- `yield`
  - 对于没有产出值的协程（产出值为None），关键字右边没有表达式 `x = yield`

1.3）调用协程的关键方法

- `gen_obj.send(value)`
  - 由调用方给协程传递值`send`
  - 生成器在`yeild`关键字处暂停，将控制权移交给调用方，`send`方法的岑书作为`yield`表达式的值
    -

```
x = yield
gen.send(43)    # x=43
```

- `next(gen_obj)`：驱动协程
  - 首次调用，预激协程`prime`
  - 后续调用，驱动协程产出值
- `gen_obj.close()`：终止协程
  - 调用方在协程上调用此方法，终止协程
  - 在暂停的`yield`表达式处抛出`GeneratorExit`异常，生成器不产出值
  - 携程状态转变为`GEN_CLOSED`
- `gen_obj.throw(exc_type[, exc_valuep, traceback])`：异常处理
  - 在暂停的`yield`表达式处抛出指定异常，如果生成器内部设置了该异常的处理逻辑，则激活该逻辑，并执行到下一个`yield`表达式。
  - 产出值会成为调用`genertor.throw`方法得到的返回值
  - 对于未设置处理逻辑的异常，向上冒泡`GEN_CLOSED`

1.4）协程预激方法

许多框架已经提供预激机制，不需要手动预激

- 显示`next`预激
- 装饰器激活
  -

```
def coroutine(func):
  @wraps(func)
  def primer(args, **kwargs):
    gen = func(*args, **kwargs)
    next(gen)
    return func
  return primer
```

- 在嵌套的协程中`yield from`自动预激

1.5）协程值的获取

- 从异常中获取：
  - 原理：
  - 值传递格式

2）`yield from`关键字

2.1）解决问题

- 联通并发异步调用的多层嵌套生成器
- 通过建立调用方和子生成器的通道，直接发送和获取产出值，同时不需要在中间过程设置异常处理模版解决不同异常

2.2）调用方、委派生成器、子生成器

- 调用方（客户端）：
  - 创建委派生成器，并通过委派生成器与子生成器通信
- 委派生成器（管道）：
  - 作为子生成器和调用方的管道
  - 在委派生成器中创建子生成器对象
  - `yield from`连接嵌套的子生成器和子委派生成器
- 子生成器（具体业务代码）
  - 生成器或其他可迭代对象
  - 通过`yield`产出值，通过`return`在子生成器结束时返回值

2.3）值的传入和传出

- 调用方值传递：调用委派生成器对象`send()`方法
  - 当委派生成器`send()`方法传输的值非None，时候，调用子生成器对象`send()`方法，传入具体值
  - 当值为None时，委派生成器调用`next()`方法

```
def client(data):
    """构建客户端
    对数据的平均统计
    :return:
    """
    results = {}        #创建空字典用于协议结果传递
    for key, values in data.items():
        group = grouper(results, key)    # 构建新的委派生成器
        next(group)          # 协程预激
        for value in values:
            group.send(value)
        group.send(None)
    print(results)
    return report(results)

def grouper(results, key):
    # 构建生成器委派器协程
```

```
    while True:
        results[key] = yield from average_v2()
```

- 子生成器产出的值都会直接传递给委派生成器的调用方
  - `yield`传出，case中产出值为None
  - 子生成器终止：`return expr`表达式会触发`StopIteration`异常抛出，结果值同时传出

```
Reuslt = namedtuple('Result',"turn avg")
def average_v2():
    # 构建子生成器
    total = 0
    turn = 0
    avg = None
    while True:
        term = yield
        if term is None:
            break
        total += term
        turn += 1
        avg = total / turn
    return Reuslt(turn, avg)
```


- 委派生成器获取子生成器产出的值
  - 正常情况下，委派生成器阻塞等待子生成器产出值
  - `yield from`表达式会获取子生成器终止时抛出`StopIteration`异常的第一个参数，作为结果值
  - Case中：如果子生成器没有因为`send(None)`终止，则委派生成器无法获取`return Result`的结果

2.4）流程结束信号

- 调用方，通过委派生成器对象`group.send(None)`终止子生成器
  - Case中：委派生成器调用子生成器对象`sub_gen.send(None)`给`term`赋值，然后调用`next(sub_gen)`（目的是结束子生成器）

2.5）异常处理

- `StopIteration`
  - 当子生成器终止时会抛出`StopIteration`
  - 委派生成器不会向上冒泡，通过`yield from`获取异常中携带的子生成器传出的结果`return expr`，委派生成器恢复运行

```
except StopIteration as _e:
  _r = e.value
```

- 其他异常：向上冒泡，传给调用方中的委派生成器对象

2.6）异常传ming入

- 当调用方，通过委派生成器传入异常时：
  - `GeneratorExit`，与在委派生成器对象调用`close()`方法一致
    - 如果子生成器有，`close()`直接调用
    - 没有`close()`方法，则抛出`GenertorExit`异常
    - 如果调用`close()`导致异常，则向上抛出
  - 其他异常：都会直接传递给子生成器对象，调用`sub_obj.throw(Exception)`
    - 如果调用`throw()`时抛出`StopIteration`，则委派生成器停止阻塞，恢复运行
    - 其他异常，向上冒泡，传给委派生成器

3）`yield from`的链条驱动模式

- 委派生成器统称管道`yield from`关键字
- 链条末端以一个只使用`yeild`表达式的简单生成器或可迭代对象结束
- 调用方处初始化、预激、调用委派生成器对象

4）Case：协程做离散事件仿真

十七、使用future处理并发

1）内容概要

本章节会构建本地服务器，客户端，顺序请求/并发请求下载国旗图片，并通过对比下载耗时验证并发下载的效率

2）future对象

2.1）常用并发框架

- `concurrent.futures.Future`
- `asyncio.Future`

2.2）future 类对象概念：

- future类对象只能通过并发框架进行实例化
- future类的实例本质是对现有可调用对象进行封装，给并发框架提供统一的接口（状态查询，结果获取）

2.3）future类对象的实例化

- 不同并发框架下实例化的future对象提供相同接口，但处理逻辑可能不同“
  - `.done()`
    - 指明`future`封装的可调用对象是否已经执行（Bool值）
  - `.add_done_callback()`
    - 指定`future`运行结束后，调用指定的可调用对象
  - `.result()`
    - 如果是`future`实例对象运行结束后调用，作用相同，返回可调用对象结果
    - 如果未结束时调用
      - `concurrency.futures.Future`类实例会阻塞调用方所在线程，到有结果返回（可以设置等待时间）
      - `asyncio.Future`会直接抛出异常
- 多线程
  -

```
with futures.ThreadPoolExecutor(max_work=2) as executor:
  todo= []    # 保存future实例对象集合
  for task in task:
    future = executor, submit(func, args)    # 封装func为future实例对象，表示待执行操作
    todo.append(future)

  for future in futures.as_completed(to_do):    # as_completed() 方法的参数是future列表，返回值是迭代器，迭代器元素是运行结束的future对象
    res = future.result()
```

- 多进程

```
with futures.ProcessPoolExecutor(max_work=2) as executor:
  todo= []    # 保存future实例对象集合
  for task in task:
    future = executor.submit(func, args)    # 封装func为future实例对象，表示待执行操作
    todo.append(future)

  for future in futures.as_completed(to_do):    # as_completed() 方法的参数是future列表，返回值是迭代器，迭代器元素是运行结束的future对象
    res = future.result()
```

- 多线程多进程处理器选择
  - GIL（全局解释器锁）
    - 一个python进程通常不能使用多个CPU核心，GIL锁只允许一次使用一个线程执行字节码
    - 对于I/O密集型程序，可以使用多线程（因为线程等待网络响时，会释放GIL）
    - 对于CPU密集型处理，使用多进程

3）Future类实例化范式（推荐第一种，避免出现结果获取阻塞的情况）

- 构造`concurrent.futures.Executor`对象（）
  - 使用`executor.submit()`实例化`future`类对象：能够处理不同的可调用对象和参数
  - 使用`futures.as_completed`驱动执行并获取结果：
    - `future`集合可以来自多个`Executor`实例
    - `as_compelted`只返回已经完成的`future`因此不会出现结果获取阻塞的情况
    - 构建`key`为`future`实例对象的字典（见Case）
- 使用`Executor.map`方法：只能处理参数不同的同一个可调用对象

```
executor = futures.ThreadPoolExecutor(max_workers=3)
results = executor.map(func, obj_list)
for i, result in enumerate(results):    # 等待result结果期间，可能存在阻塞
  print(i,result)
```

4）Case：顺序、两种多线程并发实现国旗下载

- 常用方法，`futures.as_compelted()`构建字典，将`future`映射到其他数据，用于中间数据的保存

```
with futures.ThreadPoolExecutor(max_worerkers=) as executor:
  todo_map = {}
  for cc in sorted(cc_list):
    future = executor.submit(download_one,cc, base_url, verbose):
    todo_map[future] = cc
  do_iter = futrures.as_compelted(to_map)
  for future in do_iter:
    res = future.result()
```

- 关注下载过程中，不同类型异常的处理层级

十八、asyncio处理并发

`asyncio`库目前支持TCP和UDP服务的并发编排

1）多线程和异步调用的关系：

- Case： Spinner
  - 多线程之间不停交换解释器控制权实现异步行为
- 利用多线程/异步调用的方法解决阻塞函数的影响

2）`asyncio` API 定义的协程特点：

2.1）调用方、委派生成器、子生成器

- 协程声明`@asyncio.coroutine`使用装饰器`asyncio.corotine`封装协程 `async def func`声明协程函数
  - 协程本质是委派生成器，协程链条本质是嵌套结构的委派生成器
- 调用方通过`yield from` `await`驱动协程，调用方不能够是协程
  - 协程链条最终把最外层委派生成器传给`asyncio`包中的API中的某个函数
  - `loop.run_until_complete()`驱动
- 协程链条末端的异步业务（真正的IO操作不通过我们自定义的业务逻辑完成），通过`yield from` `await`委托给`asyncio`包中的某个协程函数或协程方法
  - `aiohttp.request()`
  - `asycio.sleep()`

2.2）`asyncio`编排并发作业范式

- Case：spinner_asyncio.py

```
async def spin(msg):
    write, flush = sys.stdout.write, sys.stdout.flush
    for c in cycle(r"|/-|\\"):
        status = c + ' ' + msg
        write(status)
        flush()
        write('\x08'*len(status))
        try:
            await asyncio.sleep(.1)
        except asyncio.CancelledError:
            break
    write(" " * len(status) + "\x08" * len(status))

async def thinking():
    await asyncio.sleep(3)
    return 42

async def supervisor():

    spinner = asyncio.create_task(spin('thinking'))
    # spinner = asyncio_18(spinner('thinking'))
    print("spinner", spinner)
    result = await thinking()
    spinner.cancel()
    return result
```


- 构建事件循环：`loop = asyncio.get_event_loop()`
  - 利用事件调度协程（1）：`loop.run_until_compelete(supervisor())`
    - 对于协程（1）中嵌套的协程（2）
      - 使用Task类对象（`future`类子类）封装协程（2）：`spinner = asynico.async(spin('thinking'))`

        - 使用`yield from`驱动：`yield from slow_function()`
            - 协程（2）调用`yield from`驱动嵌套的`asyncio`库协程（链条末端业务逻辑模块仍通过调用`asyncio`库现有函数进行实现）
    - 终止协程：`spinner.cancel()`
    - 获取协程返回结果：`result = loop.run_until_complete(supervisor())`

2.3）`task类`对象与`thread`线程的区别

- `asyncio`通过将协程封装成`Task`类对象。
    - Task对象是实现协作式多任务的绿色线程
- 驱动对象不同：
    - task对象驱动携程
    - 线程驱动可调用对象
- 实例化方式：
    - Task类对象通过将携程传给`asyncio.async()`或`loop.creat_task()`获取
    - Thread实例可自行创建
- 驱动指令：
    - Task类对象在实例化时已经排定了运行时间
    - Thread对象需要显示驱动`thread.starr()`
- 数据保护
- 协程只能在
- 多线程则需要使用资源锁，防止多步操作执行中断导致数据无效
    - 协程无需保留锁，同一时刻只有一个协程在运行，在交出控制权时候会进行同步操作，实现数据保护
        - 通过`yield /yield from`交换控制权
        - 协程结束通过`cancelledError`异常处理，保证了安全取消与资源清理
    - 线程需要建立锁机制，防止执行过程中断，数据无效的情况

2.4）区别`asyncio.Future`类与`concurrent.futures.Future`

详见（十八）中的两者在相同方法调用的区别

3）future、任务和协程的关系

3.1）驱动方式

- `asyncio.Future`类（task类是future的子类）通过`yield from`驱动（绑定使用）
- 一般无需调用`future.result()/future.add_done_callback()/future.done()`等方法

3.2）协程与future类对象的关系

- 协程被封装成Task类实例
- `asyncio`API接收的参数对象是task/future对象
    - `asyncio.async(coro_or_future)`如果是`future`对象直接返回，如果是协程进行封装
    - `creat_task(coro)`构建Task对象
    - `loop.run_until_completed(coro)` 将参数指定的协程包装在Task对象中

- `asyncio.wait()`协程函数，同时处理多个协程，将多个协程包装到一个`task`对象中
  - 产出协程或生成器对象，交给`loop/rim_until_complete`驱动产出结果

4）协程中控制权的切换方式

4.1）`yield from foo()/ future`利用关键字驱动协程

- 当前协程暂停后，控制权交还给事件循环，再去驱动其他协程。
- 当`foo 或 future`运行完毕后，将结果返回给暂停的协程，协程从事件获得控制权，恢复运行

5）使用异步策略的原因

- CPU读取不同介质中的数据的速率不同。不同存储介质中读取数据存在不同量级的延迟
  - Min：L1 缓存
  - Max：网络
- 阻塞型调用在不同存储介质的读取延迟导致整个应用程序进程的中止，解决方案：
  - 单独线程运行各个阻塞型操作
  - 将阻塞型操作转换成非阻塞的异步调用

5.1）非阻塞的异步调用

- 多线程
  - 内存开销过大
- 基于回调实现异步调用
  - 回调概念：利用硬件中断，不等待响应而是注册一个函数，在发生某件事时调用
  - 异步应用程序底层事件循环依靠基础设施的中断、线程、轮询和后台进程等，确保多个并发请求能够完成
- 基于协程的异步调用：
  - 避免回调地狱：实现嵌套循环（三次异步调用=三层嵌套回调）
  - 协程暂停带来的内存开销远小于线程消耗的内存
  - 同在单线程中，Case中的国旗下载任务，异步执行比顺序执行快5倍

6）优化国旗下载Case

- 异常处理：
  - 自定义异常类，将运行过程中的国旗编号保存到异常信息中
  - HTTP 请求：`status == 200/ status==404/ 其他异常`
- 利用上下文管理器+同步装置`Semaphore`限制并发请求数量
  - `Semphore`内部维护一个计数器，通过`.acquire()`提供协程，`.release()`释放协程，实现并发请求数量的限制
    - `with await semaphore:`
- 进度监控（Case 18-8）：
  - 构建协程的列表对象，封装成`future`对象（返回迭代器）
    - `to_do_iter = asyncio.as_compelete(coro_list)`
  - 驱动迭代器，执行`future`

- 对IO操作构建事件循环，实现多线程异步调用

```
loop = asyncio.get_event_loop()
loop.run_in_executor(save_flag)
```

7）同时发送多个下载请求

- 单独构建http下载请求的协程，用于处理不同需求的下载请求（国家代码，国家名称）
- 根据下载内容构建调用http下载请求的协程

8）利用`asyncio`编写服务器（Case：Charfinder）

建议直接跑demo代码

- `asyncio`编写TCP服务器
- `aiohttp`：编写Web服务器
  - 启动与关闭事件循环与HTTP服务器

十九、动态属性和特性

1）Case：动态属性访问Json格式嵌套字典

```
clasa FrozenJson():
  def __init__(dic):
    self._data = dict(dic)    # 将字典数据保存到实例的属性

  def __getattr__(self, item): # 以属性进行字典值获取的关键方法
    if has(self, attr):
      return getattr(self, item)    # 调用内置函数
    else:
      return FrozenJson.build(self._data[item])       # 查看实例data属性存储的字典中是否存在同名键值

  @classmethod
  def build(cls, item):           # 构造类方法实现备选实例化方法
    if isinstance(item, abc.MutableMapping):
      return cls(item)
    elif isinstance(item, abc.MutableSequence):
      return [cls(i) for i in item]
    else:
      return item
```

1.1）以属性获取方式读取字典信息

- `__getattr__`:
- 实现属性获取方式的键值查找功能

1.2）构造类方法动态生成类实例的嵌套

- `build(cls, item)`
  - 对不同类型的输入数据进行类型判断
- 利用`__new__`方法实现类实例的初始化

```
class __new__(cls, args):
  if isinstance(item, abc.MutableMapping):
    super().__new__(cls, args)
  elif isinstance(item, abc.MutableSequence):
    return [cls(arg) for arg in args]
  else:
    return args
```

  - `new`方法的第一个参数是类，返回类的实例
    - 对于类实例的`__class__`属性存储的是类的引用
    - 并将实例作为参数`self`输入`__init__`方法进行实例初始化；

1.3）解决自定义类属性与类的关键属性重名问题

```
def __init__(self, dic):
  for k,v in dic.items():
    if iskeyword(k):
      k += '_'    # 对于用_区分自定义类名
    self._data[k] = v
```


2）数据源格式调整

2.1）数据库构造、载入

- `shelve`库实现数据库的构造

2.2）自定义类存储入数据库

- 对数据格式不同信息字段进行类自定义：Record、DbRecord、Event
  - Record：数据存储基类
  - DbRecord：继承Record、获取数据库、从数据库值搜索的方法
  - 链接每一条数据记录中`venue\spearker`属性的

```
class Record:
  def __init(**kwargs):
    self.__dict__.update(kwargs)
```

```
class DbRecord(Record):
  _db = None

  @staticMethod
  def get_db():
    return DbRecord._db

  @ staticmethod
  def set_db(db):
    DbRecord = db

  @classmethod
  def fetch(cls, intend)
    db = cls.get(db)
    return db[intend]
```

```
class Event(Record):
  @propery
  def speakers(self):    #以属性获取方式访问，类能够使用fetch 方法对实例的属性进行搜索
    pass

  @property
  def venue(self):
    pass
```


2.3）类公开属性同名处理，避免无限递归

- 当类实例公开属性与类特性同名，如果要获取实例的同名属性，为了避免无限递归。
  - 从实例的`__dict__`公开属性字典中直接获取属性，跳过特性。
  - 此时不使用fetch方法

3）类的特性：

类特性的是对类属性存取方法进行定义与管理，特性本质是是覆盖型描述符

3.1）Case：LineItem 类构造

3.2）`property` 构造方法完整签名

- `propery(fget, fset, fdel, doc)`
- 显示声明类特性`weight = propery()`
- 构造类属性的读值、写值和删除方法

3.3）类私有属性与特性设值

- 利用特性管理LineItem类实例的`weight`属性

```
class LineItem():
  def __init__(weight)
    self._weight = weight
```

```
@property          # 值读取
@weight(self):
  return self._weight

@weight.setter     # 值写入
def weight(self, value):
  if value > 0:
    self._weight = value
  else:
    raise
```

- 使用`@property`包装特性对象`obj`
- 使用`@obj.setter`自定义取值方法
- 使用`@obj.deleter`自定义特性删除方法（python支持属性和特性删除，但一般情况下不这么做）
- 特性构造类私有属性的读值与写值

3.4）类特性对类实例属性的遮盖情况

- 类实例的公开属性会覆盖类的数据属性
  -

```
class Test():
  data = sss
  @property
  def p():
    pass
t1 = Test()
t1.__dict__      # None
t1.data          # 向上索引到类数据属性 sss
t1.data = 1      #
t1.__dict__      # {data:1}
t1.data          # 1 实例属性遮盖类属性
```


- 类特性不会被实例属性遮盖
  - 特性是覆盖型描述符的一种，无法遮盖的原因是解释器对类实例的属性获取顺序（先类再实例）
    - 先查找`obj.__class__ has attr`
    - 当类特性没有搜索到时，再查找`obj.attr has attr`

3.5）定义特性工厂

- 通过定义特性工厂函数，实现对类属性的读值和写值
- 特性工厂范式

```
def quantity(storage_name):
  def qty_getter(instance):
    return instance.__dict__[storage_name]

  def qty_setter(instance, value):      # instance 指代要把存储属性的类实例
    if value > 0:
      instance.__dict__[storage_name] = value
    else:
      raise ValueError

  return property(qty_getter, qty_setter)
```

4）类属性的属性管理

4.1）三个特殊属性

- `__class__`：对象所属类的引用
  - `__getattr__`方法只在所属类寻找属性，不在类实例中
- `__dict__`：映射，存储对象或者类的可写属性
  - 如果有`__slot__`属性，那么实例可能没有`__dict__`属性
- `__slots__`：限制实例能有的属性
  - 值是一个字符串元组，指明允许有的属性

4.2）内置函数，对对象的属性做读、写和内省操作

- `dir([obj])`
  - 功能：审查并列出对象是否有`__dict__`属性的对象
  - 作用域：列出`__dict__`属性中的键
  - 异常：如果没有obj参数，则列出当前作用域名称
- `getattr[obj, name[, default]]`
  - 功能：获取对象属性
  - 作用域：对象所属的类和超类获（向上传递，不会查看到实例层级）
  - 异常：抛出AttributeError
- `hasattr(obj, name)`
  - 功能：是否有指定属性（返回True/False），本质是使用`getattr`方法
  - 作用域：obj对象、obj对象，及能够通过obj获取的属性（继承）
  - 异常：AttributeError
- `setattr(obj, name, value)`
  - 功能：把obj对象的指定属性值设置为value
  - 作用域：覆盖对于已有属性，或者创建新属性
  - 异常
- `vars(object)`
  - 功能：返回对象的`__dict__`属性，
  - 作用域：无法处理定义了`__slot__`且没有`__dict__`属性的实例
  - 异常：没有obj参数，返回本地作用域字典

4.3）处理属性的特殊方法：使用4.2）中的内置方法，会触发类属性的特殊方法

- `__delattr__(self, name)`
  - 描述：删除属性
  - 触发：`del obj.attr`调用此特殊方法
- `__dir__(self)`：
  - 描述：将对象传给`dir`函数调用，列出属性
- `__getattr__(self name)`：
  - 描述：获取指定属性
  - 异常：当获取指定属性失败，且搜索过obj、class 和超类后调用
- `__getattribute(self, name)`
  - 描述：获取指定属性时调用的常用方法
  - 异常：当抛出`attributeError`异常时，下一步才调用`__getattr__`方法
- `__setattr(self, name, value)`：
  - 触发：由`.`和`setattr`内置函数触发

二十、属性描述符

1）利用覆盖型描述符实现对LineItem类的属性管理

1.1）Case 利用Quality描述符对LineItem类属性进行管理

将描述符类实例声明为托管类的类属性，因此对此类属性的读、写由描述符类实例进行管理。此类属性成为托管属性（`weight\price`），属性的值保存在储存属性中。因此访问顺序是，托管类实例(instance)->托管类的托管属性(描述符实例）->调用描述符类实例中的读写方法->修改托管类实例中的储存属性

- 描述符类
  - 实现描述符协议的类（quantity）
- 托管类
  - 将描述符实例声明为类的属性
- 描述符类实例
  - 描述符类的各个实例，声明为托管类的类属性
- 托管类实例
  - 托管类实例，参照访问顺序对实例中的储存属性进行值读取
- 储存属性
  - 托管类实例中存储自身托管属性的属性 (instance.weight)
- 托管属性
  - 托管类中由描述符实例公开处理的属性（quantity.storage_name）

1.2）描述符实例作为类属性被托管类实例之间共享

- 对属于同一个类的存储属性运用相同的存取逻辑
- 将描述符实例作为类（托管类）的属性，由于类实例化的对象共享类的属性，因此可以理解为托管实例共享操作符实例（多对一关系）
- 描述符实例参与到托管实例的实例化操作中，操作结果作为实例的存储属性保存（理解为字典 key value）。实例有单独的存储属性集合（不同实例之间不共享）
- 实例的托管属性（理解为key），可以通过托管属性的key 去 实例的储存属性字典中查找value；
  - 当托管属性和储存属性的key 同名时
    - 索引和赋值为了避免递归，`obj.__dict__[key]`
  - 不同名时
    - 使用内置函数 `getattr/ setattr`

2）LineItem迭代优化

2.1）`bulkfood_v3`使用`Quantity`描述符管理LineItem属性

- 描述符实例中含有`storage_name`实例属性
- 利用对描述符类实例属性赋值时`Quantity.storage_name`，实际是对托管类实例中同名属性进行赋值管理`instance.__dict__[storage_name] = value`

```
class Quantity():
  def __init(self, storage_name):      # 描述符
    self.storage_name = storage_name

  def __set__(self, instance, value):
    if value > 0:
      instance.__dcit__[self.storage_name] = value
    else:
      raise
```

2.2）利用闭包对每一个描述符实例设计不同名属性

- 利用闭包实现每一个托管类实例的`storage_name`字段是不同的
  - `instance0.__dict__[_quantity#0]` -》 `instance0.weight`
  - `instance1.__dict__[_quantity#1]` -》 `instance1.weight`
- 索引LineItem实例(it.weight)的时候，实际是先去LineItem类属性找到描述符实例，调用`get`方法从描述符实例获取`storage_name`作为key，再到托管类实例`instance.__dict__[storage_name]`取值

```
class Quantity:
  __counter = 0
  def __init__(self):     # 不需要将托管类实例的weight，price作为参数
    cls = self.__class__
    prefix = cls.__name__
    index = cls.__counter
    self.storage_name = '_{}#{}'.format(prefix, index)
    cls.__counter +=1

  def __get__(self, instance, owner):   # instance
    return getattr(instance, self.storage_name)
```

2.3）解决无法直接通过类访问托管属性抛出的异常

- `owner`参数是托管类（`Lineitem`）的引用，如果直接从托管类中获取托管属性（`weight`）会抛出异常，因为此时并没有托管类的实例化对象（`instance 是None`）

```
def __get__(self, instance, owner):
  if instance is None:
    return self
  else:
    return getattr(instance, self.storage_name)
```


2.4）将描述符类进行抽象管理

- 通过模版方法设计模式构造`AutoStorage/Validated/Quantity/NoneBlank`
  - `AutoStorage`：自动管理存储属性的描述符类
    - `storage_name`属性
    - `init/get/set`属性读写方法
  - `validated`拓展`AutoStorage`类的抽象子类
    - 重载`set`方法
    - 声明`validate`抽象方法
  - `Quantity`/`NonBlank`
    - 声明`validate`具体方法
- Case：20-6

3）特性工厂与描述符类对比

- 创建局部变量`storage_name`并利用闭包保持值，供`qty_getter/qty_setter`使用

```
def quantity():
  try:
```

```
    quantity.counter += 1
  except:
    quantity.counter = 0

  storage_name = '_{}#{}'.format('quantity', quantity.counter)

  def qty_getter(instance):
    return getattr(instance, storage_name)

  def qty_setter(instance, value):
    pass
  return property(qty_getter, qty_setter)
```

4）覆盖型与非覆盖型描述符

- 内置的`property`特性类本质是覆盖型描述符
  - 已经自动实现了`get/set`
- 覆盖型描述类方法中自定义了`set\get`，覆盖了托管类实例的同名方法

4.1）定义了`get/set`读写方法的覆盖型描述符

类托管属性的读写都通过描述符对象处理

```
def __get__(self, instance, owner):
  pass

def __set__(self, instance, value):
  pass
```


- `property`特性属于覆盖型描述符
- `__set__`会覆盖对实例属性的赋值操作
  - 如果对没有提供设值的函数，使用`set`方法会抛出AtrributeError 异常
- `__get__`

4.2）只实现了`set`方法的覆盖型描述符

- 写操作由描述符对象处理
- 对托管实例的新增属性，读取该属性会从实例中返回新赋予的值，不会返回描述符对象
  - 读操作时，实例属性会覆盖描述符

4.3）没有实现`set`方法的非覆盖型描述符

- 描述符会被遮盖，导致描述符无法处理该实例的托管属性

4.4）上述4.1）- 4.2）覆盖针对的是类实例的属性

- 读类属性的操作可以由托管类定义了`get`方法的描述符实例处理
- 写类属性的操作不会被`set`描述符实例处理（无法覆盖）

5）方法与描述符

5.1）绑定方法

- 由于用户定义的函数都有`__get__`方法，因此类中定义方法时，相当于绑定了描述符
  - 类中函数的`__get__`方法（函数都是非覆盖型描述符）
    -
  - 如果对类方法进行赋值，会遮盖该方法
  - `__get__`方法
    - 通过托管类访问时，函数`get`方法返回自身引用
    - 通过实例访问，返回绑定方法的具体实例的包装对象（作为中间状态的可调用对象）
      - 并把托管实例绑定给函数的第一个参数`self`

5.2）实例的类方法包装对象的属性

- `__self__` 调用这个方法绑定的实例的引用
- `__func__`实例中类方法的包装对象所包装的函数的引用（原始类函数的引用）
- `__call__`处理调用过程
  - 首先调用`__func__`获取原始函数
  - 把函数第一个参数`self`设置为通过`__self__`绑定的实例（形参self的隐式绑定方法）

6）描述符用法原则

- 使用property类创建覆盖型描述符
- 只读描述符也需要实现`set`方法，避免同名属性覆盖描述符
  - `set`方法
- 用于验证的描述符可以只有`set`方法
  - 直接在托管类实例`__dict__`属性设置值，key就是描述符实例名称
- 区别只读和只有`get`方法的描述符
  - 利用非覆盖描述符，处理耗费资源的计算，并在计算完成后为实例设置同名属性，实现快速读取
- 函数和方法都只实现了`__get__`
  - 是会被实例属性覆盖
  - 特殊方法不会覆盖
- 描述符的常见方法
  - `get`
  - `set`

- `delete`

二十一、类元编程

1）python程序导入和运行状态

2）动态创建类

2.1）类工厂函数

2.2）类装饰器

3）py文件的导入与运行

- 导入过程的操作
  - 对类
  - 对函数

4）object 类和type类的关系

- obejct 是type类的实例
- type类是object类的子类
- 所有类都间接或直接的是type的实例
- 但元类还是type的子类。因此可以作为制造类的工厂

5）实例化元类替代类装饰器