## 零碎知识点：

* “短路逻辑（short-circuit logic）”：逻辑操作符有个有趣的特性：在不需要求值的时候不进行操作。这么说可能比较“高深”，举个例子，表达式 x and y，需要 x 和 y 两个变量同时为真(True)的时候，结果才为真。因此，如果当 x 变量得知是假(False)的时候，表达式就会立刻返回 False，而不用去管 y 变量的值。
  + x and y，当x为True，返回y的值
  + x or y， 当x，y均为True，返回x的值
* not or and 的优先级是不同的：**not** > **and** > **or**
* type(input) 返回参数类型，isinstance(input, int)判断变量是否为int类型
* s 为字符串

s.isalnum() 所有字符都是数字或者字母，为真返回 True，否则返回 False。

s.isalpha() 所有字符都是字母，为真返回 True，否则返回 False。

s.isdigit() 所有字符都是数字，为真返回 True，否则返回 False。

s.islower() 所有字符都是小写，为真返回 True，否则返回 False。

s.isupper() 所有字符都是大写，为真返回 True，否则返回 False。

s.istitle() 所有单词都是首字母大写，为真返回 True，否则返回 False。

s.isspace() 所有字符都是空白字符，为真返回 True，否则返回 False。

**字符串的方法及注释**

|  |  |
| --- | --- |
| capitalize() | 把字符串的第一个字符改为大写 |
| casefold() | 把整个字符串的所有字符改为小写 |
| center(width) | 将字符串居中，并使用空格填充至长度 width 的新字符串 |
| count(sub[, start[, end]]) | 返回 sub 在字符串里边出现的次数，start 和 end 参数表示范围，可选。 |
| encode(encoding='utf-8', errors='strict') | 以 encoding 指定的编码格式对字符串进行编码。 |
| endswith(sub[, start[, end]]) | 检查字符串是否以 sub 子字符串结束，如果是返回 True，否则返回 False。start 和 end 参数表示范围，可选。 |
| expandtabs([tabsize=8]) | 把字符串中的 tab 符号（\t）转换为空格，如不指定参数，默认的空格数是 tabsize=8。 |
| find(sub[, start[, end]]) | 检测 sub 是否包含在字符串中，如果有则返回索引值，否则返回 -1，start 和 end 参数表示范围，可选。 |
| index(sub[, start[, end]]) | 跟 find 方法一样，不过如果 sub 不在 string 中会产生一个异常。 |
| isalnum() | 如果字符串至少有一个字符并且所有字符都是字母或数字则返回 True，否则返回 False。 |
| isalpha() | 如果字符串至少有一个字符并且所有字符都是字母则返回 True，否则返回 False。 |
| isdecimal() | 如果字符串只包含十进制数字则返回 True，否则返回 False。 |
| isdigit() | 如果字符串只包含数字则返回 True，否则返回 False。 |
| islower() | 如果字符串中至少包含一个区分大小写的字符，并且这些字符都是小写，则返回 True，否则返回 False。 |
| isnumeric() | 如果字符串中只包含数字字符，则返回 True，否则返回 False。 |
| isspace() | 如果字符串中只包含空格，则返回 True，否则返回 False。 |
| istitle() | 如果字符串是标题化（所有的单词都是以大写开始，其余字母均小写），则返回 True，否则返回 False。 |
| isupper() | 如果字符串中至少包含一个区分大小写的字符，并且这些字符都是大写，则返回 True，否则返回 False。 |
| join(sub) | 以字符串作为分隔符，插入到 sub 中所有的字符之间。 |
| ljust(width) | 返回一个左对齐的字符串，并使用空格填充至长度为 width 的新字符串。 |
| lower() | 转换字符串中所有大写字符为小写。 |
| lstrip() | 去掉字符串左边的所有空格 |
| partition(sub) | 找到子字符串 sub，把字符串分成一个 3 元组 (pre\_sub, sub, fol\_sub)，如果字符串中不包含 sub 则返回 ('原字符串', '', '') |
| replace(old, new[, count]) | 把字符串中的 old 子字符串替换成 new 子字符串，如果 count 指定，则替换不超过 count 次。 |
| rfind(sub[, start[, end]]) | 类似于 find() 方法，不过是从右边开始查找。 |
| rindex(sub[, start[, end]]) | 类似于 index() 方法，不过是从右边开始。 |
| rjust(width) | 返回一个右对齐的字符串，并使用空格填充至长度为 width 的新字符串。 |
| rpartition(sub) | 类似于 partition() 方法，不过是从右边开始查找。 |
| rstrip() | 删除字符串末尾的空格。 |
| split(sep=None, maxsplit=-1) | 不带参数默认是以空格为分隔符切片字符串，如果 maxsplit 参数有设置，则仅分隔 maxsplit 个子字符串，返回切片后的子字符串拼接的列表。 |
| splitlines(([keepends])) | 在输出结果里是否去掉换行符，默认为 False，不包含换行符；如果为 True，则保留换行符。。 |
| startswith(prefix[, start[, end]]) | 检查字符串是否以 prefix 开头，是则返回 True，否则返回 False。start 和 end 参数可以指定范围检查，可选。 |
| strip([chars]) | 删除字符串前边和后边所有的空格，chars 参数可以定制删除的字符，可选。 |
| swapcase() | 翻转字符串中的大小写。 |
| title() | 返回标题化（所有的单词都是以大写开始，其余字母均小写）的字符串。 |
| translate(table) | 根据 table 的规则（可以由 str.maketrans('a', 'b') 定制）转换字符串中的字符。 |
| upper() | 转换字符串中的所有小写字符为大写。 |
| zfill(width) | 返回长度为 width 的字符串，原字符串右对齐，前边用 0 填充。 |

* assert这个关键字我们称之为“断言”，当这个关键字后边的条件为假的时候，程序自动崩溃并抛出AssertionError的异常。

什么情况下我们会需要这样的代码呢？当我们在测试程序的时候就很好用，因为与其让错误的条件导致程序今后莫名其妙地崩溃，不如在错误条件出现的那一瞬间我们实现“自爆”。

一般来说我们可以用Ta在程序中置入检查点，当需要确保程序中的某个条件一定为真才能让程序正常工作的话，assert关键字就非常有用了。

* 假设有 x = 1，y = 2，z = 3，请问如何快速将三个变量的值互相交换？

答案：x, y, z = z, x, y

* 成员资格运算符：in，用于检查一个值是否在序列中，如果在序列中返回 True，否则返回 False。 str in input
* 三元操作符的实现：small = x if (x < y and x < z) else (y if y < z else z)
* break语句的作用是终止当前循环，跳出循环体，只能跳出一层循环。
* continue语句的作用是终止本轮循环并开始下一轮循环（这里要注意的是：在开始下一轮循环之前，会先测试循环条件）。
* 什么情况下我们要使循环永远为真？

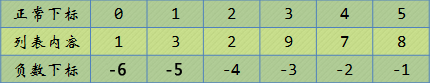
同样用于游戏实现，因为游戏只要运行着，就需要时刻接收用户输入，因此使用永远为真，确保游戏“在线”。操作系统也是同样的道理，时刻待命，操作系统永远为真的这个循环叫做消息循环。另外，许多通讯服务器的客户端/服务器系统也是通过这样的原理来工作的。所以永远为“真”虽然是“死循环”，但不一定是坏事，再说了，我们可以随时用 break 来跳出循环！

* 向列表增加元素，分别是：append()、extend() 和 insert()。

append() 方法是将参数作为一个元素增加到列表的末尾。

extend() 方法则是将参数作为一个列表去扩展列表的末尾。

* Python 的列表很聪明，竟然支持负数索引，如图：正常索引是从左到右索引，负数索引是从右到左。



* 列表顺序排序：list1.sort()
* 列表逆序排序：list1.sort(); list1.reverse() 或者 list1.sort(reverse=True)
* copy() 方法跟使用切片拷贝是一样的：list2 = list1.copy() 和 list2 = list1[:] 相同
* clear() 方法用于清空列表的元素，但要注意，清空完后列表仍然还在哦，只是变成一个空列表。
* 列表推导式（List comprehensions）也叫列表解析，灵感取自函数式编程语言 Haskell。Ta 是一个非常有用和灵活的工具，可以用来动态的创建列表，语法如：[有关A的表达式 for A in B]

list1 = [x\*\*2 for x in range(10)] 等同于 list1 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

* 列表：一个大仓库，你可以随时往里边添加和删除任何东西。当我们需要频繁修改数据，我们使用列表。
* 元组：封闭的列表，一旦定义，就不可改变（不能添加、删除或修改）。当我们希望内容不被轻易改写的时候，我们使用元组。
* 定义跨越多行的字符串的方法
  + str1 = '''待我长发及腰，将军归来可好？昔日纵马任逍遥，俱是少年英豪。'''

三引号字符串不赋值的情况下，通常当作跨行注释使用

* + str2 = '待卿长发及腰，我必凯旋回朝。\ 昔日纵马任逍遥，俱是少年英豪。'
  + str3 = ('待卿长发及腰，我必凯旋回朝。' '昔日纵马任逍遥，俱是少年英豪。')
* "{a} love {b}.{c}".format(a="I", b="FishC", c="com")中，a, b, c是关键字参数
* "{0} love {1}.{2}".format("I", "FishC", "com")中，{0}, {1}, {2}是位置参数
* '{0}{1:.2f}'.format('Pi = ', 3.1415)

|  |  |
| --- | --- |
| **字符串格式化符号** | **说     明** |
| %c | 格式化字符及其ASCII码 |
| %s | 格式化字符串 |
| %d | 格式化整数 |
| %o | 格式化无符号八进制数 |
| %x | 格式化无符号十六进制数 |
| %X | 格式化无符号十六进制数（大写） |
| %f | 格式化定点数，可指定小数点后的精度 |
| %e | 用科学计数法格式化定点数 |
| %E | 作用同%e，用科学计数法格式化定点数 |
| %g | 根据值的大小决定使用%f或者%e |
| %G | 作用同%g，根据值的大小决定使用%F或者%E |

* bin() 用于将十进制转化成二进制，hex() 用于将十进制转化成十六进制，oct() 用于将十进制转化成八进制。
* list([iterable]) 把可迭代对象转换为列表  
  tuple([iterable]) 把可迭代对象转换为元组  
  str(obj)  把对象转换为字符串
* 迭代，是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值。

## 函数

* 在函数开头写下的字符串Ta是不会打印出来的，但Ta会作为函数的一部分存储起来，这个我们称之为函数文档字符串，Ta的功能跟注释是一样的。函数的文档字符串可以按如下方式访问：

MyFirstFunction.\_\_doc\_\_ 或者 help(MyFirstFunction)

* 关键字参数，是指函数在调用的时候，带上参数的名字去指定具体调用的是哪个参数，从而可以不用按照参数的顺序调用函数。关键字参数是在函数调用的时候，通过参数名制定需要赋值的参数，这样做就不怕因为搞不清参数的顺序而导致函数调用出错。而默认参数是在参数定义的过程中，为形参赋初值，当函数调用的时候，不传递实参，则默认使用形参的初始值代替。

>>> def SaySome(name, words):

print(name + '->' + words)

>>> SaySome(words='让编程改变世界！', name='小甲鱼')

* 在函数中修改全局变量的值，使用global关键字
* 在嵌套的函数中，在内部函数修改外部函数的局部变量，使用nonlocal关键字
* 当 a = funX() 的时候，只要 a 变量没有被重新赋值，funX() 就没有被释放，也就是说局部变量 x 就没有被重新初始化。因此funX()中的内部变量并没有被初始化！！！
* 使用lambda匿名函数的好处：
  + Python 写一些执行脚本时，使用匿名函数就可以省下定义函数过程，比如说我们只是需要写个简单的脚本来管理服务器时间，我们就不需要专门定义一个函数然后再写调用，使用匿名函数就可以使得代码更加精简。
  + 对于一些比较抽象并且整个程序执行下来只需要调用一两次的函数，有时候给函数起个名字也是比较头疼的问题，使用匿名函数就不需要考虑命名的问题了。
  + 简化代码的可读性，由于普通的屌丝函数阅读经常要跳到开头 def 定义部分，使用匿名函数可以省去这样的步骤。
* 内置参数filter() 和 map()
* 在编程中有没有不得不使用递归的情况？例如汉诺塔，目录索引（因为你永远不知道这个目录里边是否还有目录），快速排序（二十世纪十大算法之一），树结构的定义等如果使用递归，会事半功倍，否则会导致程序无法实现或相当难以理解。

## 字典

* >>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

* 在字典中检查键（Key）是否存在 比 在序列中检查指定元素是否存在更高效。因为字典的原理是使用哈希算法存储，一步到位，不需要使用查找算法进行匹配，因此时间复杂度是O（1），效率非常高。
* Python对键的要求相对要严格一些，要求它们必须是可哈希（Hash）的对象，不能是可变类型（包括变量、列表、字典本身等）。但是Python对值是没有任何限制的，它们可以是任意的Python对象。对于 Python 来说，键（Key）必须是可哈希的，换句话说就是要可以通过散列函数计算出唯一地址的。那如果拿一个变量当键（Key）可以吗？肯定不行。因为变量随时都可能改变，不符合可哈希原则！同样的，列表、字典、集合这些都是可变的，所以都不能做为键（Key）来使用。
* Python 的哈希算法对相同的值计算得到的结果是一样的，也就是说 12315 和 12315.0 的值相同，他们被认为是相同的键（Key）。
* fromkeys方法是直接创建一个新的字典，不要试图使用它来修改一个原有的字典，因为它会直接无情的用把整个字典给覆盖掉。

## 集合

* 集合几乎所有的作用就是确保里边包含的元素的唯一性，就像世界上没有两片完全相同的树叶一样，集合内不可能存在两个相同的元素！集合是无序的，所以无法使用索引进行访问。

num\_set = set([1, 2, 3, 4, 5]) or num\_set ={1, 2, 3, 4, 5}

* 如果你希望创建的集合是不变的，使用frozenset()
* 利用哈希函数计算，相同的元素得到的哈希值（存放地址）是相同的，所以在集合中所有相同的元素都会覆盖掉，因此有了集合的唯一性。通过哈希函数计算的地址不可能是按顺序排放的，所以集合才强调是无序的！
* num1.add(6) or num1.remove(6)

## 文件

* 文件读取：f = open(r'C:\Users\Niubility\Desktop\OpenMe.mp3')

|  |  |
| --- | --- |
| **打开模式** | **执行操作** |
| 'r' | 以只读方式打开文件（默认） |
| 'w' | 以写入的方式打开文件，会覆盖已存在的文件 |
| 'x' | 如果文件已经存在，使用此模式打开将引发异常 |
| 'a' | 以写入模式打开，如果文件存在，则在末尾追加写入 |
| 'b' | 以二进制模式打开文件 |
| 't' | 以文本模式打开（默认） |
| '+' | 可读写模式（可添加到其他模式中使用） |
| 'U' | 通用换行符支持 |
| **文件对象方法** | **执行操作** |
| f.close() | 关闭文件 |
| f.read([size=-1]) | 从文件读取size个字符，当未给定size或给定负值的时候，读取剩余的所有字符，然后作为字符串返回 |
| f.readline([size=-1]) | 从文件中读取并返回一行（包括行结束符），如果有size有定义则返回size个字符 |
| f.write(str) | 将字符串str写入文件 |
| f.writelines(seq) | 向文件写入字符串序列seq，seq应该是一个返回字符串的可迭代对象 |
| f.seek(offset, from) | 在文件中移动文件指针，从from（0代表文件起始位置，1代表当前位置，2代表文件末尾）偏移offset个字节 |
| f.tell() | 返回当前在文件中的位置 |
| f.truncate([size=file.tell()]) | 截取文件到size个字节，默认是截取到文件指针当前位置 |

* 将文件f中所有语句输出：for each\_line in f: print(each\_line)
* Import os

**os模块中关于文件/目录常用的函数使用方法**

|  |  |
| --- | --- |
| **函数名** | **使用方法** |
| getcwd() | 返回当前工作目录 |
| chdir(path) | 改变工作目录 |
| listdir(path='.') | 列举指定目录中的文件名（'.'表示当前目录，'..'表示上一级目录） |
| mkdir(path) | 创建单层目录，如该目录已存在抛出异常 |
| makedirs(path) | 递归创建多层目录，如该目录已存在抛出异常，注意：'E:\\a\\b'和'E:\\a\\c'并不会冲突 |
| remove(path) | 删除文件 |
| rmdir(path) | 删除单层目录，如该目录非空则抛出异常 |
| removedirs(path) | 递归删除目录，从子目录到父目录逐层尝试删除，遇到目录非空则抛出异常 |
| rename(old, new) | 将文件old重命名为new |
| system(command) | 运行系统的shell命令 |
| walk(top) | 遍历top路径以下所有的子目录，返回一个三元组：(路径, [包含目录], [包含文件]) |
| **以下是支持路径操作中常用到的一些定义，支持所有平台** | |
| os.curdir | 指代当前目录（'.'） |
| os.pardir | 指代上一级目录（'..'） |
| os.sep | 输出操作系统特定的路径分隔符（Win下为'\\'，Linux下为'/'） |
| os.linesep | 当前平台使用的行终止符（Win下为'\r\n'，Linux下为'\n'） |
| os.name | 指代当前使用的操作系统（包括：'posix',  'nt', 'mac', 'os2', 'ce', 'java'） |

**os.path模块中关于路径常用的函数使用方法**

|  |  |
| --- | --- |
| **函数名** | **使用方法** |
| basename(path) | 去掉目录路径，单独返回文件名 |
| dirname(path) | 去掉文件名，单独返回目录路径 |
| join(path1[, path2[, ...]]) | 将path1, path2各部分组合成一个路径名 |
| split(path) | 分割文件名与路径，返回(f\_path, f\_name)元组。如果完全使用目录，它也会将最后一个目录作为文件名分离，且不会判断文件或者目录是否存在 |
| splitext(path) | 分离文件名与扩展名，返回(f\_name, f\_extension)元组 |
| getsize(file) | 返回指定文件的尺寸，单位是字节 |
| getatime(file) | 返回指定文件最近的访问时间（浮点型秒数，可用time模块的gmtime()或localtime()函数换算） |
| getctime(file) | 返回指定文件的创建时间（浮点型秒数，可用time模块的gmtime()或localtime()函数换算） |
| getmtime(file) | 返回指定文件最新的修改时间（浮点型秒数，可用time模块的gmtime()或localtime()函数换算） |
| **以下为函数返回 True 或 False** | |
| exists(path) | 判断指定路径（目录或文件）是否存在 |
| isabs(path) | 判断指定路径是否为绝对路径 |
| isdir(path) | 判断指定路径是否存在且是一个目录 |
| isfile(path) | 判断指定路径是否存在且是一个文件 |
| islink(path) | 判断指定路径是否存在且是一个符号链接 |
| ismount(path) | 判断指定路径是否存在且是一个挂载点 |
| samefile(path1, paht2) | 判断path1和path2两个路径是否指向同一个文件 |
| exists(path) | 判断指定路径（目录或文件）是否存在 |
| isabs(path) | 判断指定路径是否为绝对路径 |
| isdir(path) | 判断指定路径是否存在且是一个目录 |

## 永久存储pickle

* pickle的实质就是利用一些算法将你的数据对象“腌制”成二进制文件，存储在磁盘上，当然也可以放在数据库或者通过网络传输到另一台计算机上。
* pickle.dump(data, file) # 第一个参数是待存储的数据对象，第二个参数是目标存储的文件对象，注意要先使用'wb'的模式open文件
* pickle.load(file) # 参数是目标存储的文件对象，注意要先使用'rb'的模式open文件

## 异常捕获

Python标准异常总结

|  |  |  |
| --- | --- | --- |
| AssertionError | 断言语句（assert）失败 | BaseException  +-- SystemExit  +-- KeyboardInterrupt  +-- GeneratorExit  +-- Exception  +-- StopIteration  +-- ArithmeticError  | +-- FloatingPointError  | +-- OverflowError  | +-- ZeroDivisionError  +-- AssertionError  +-- AttributeError  +-- BufferError  +-- EOFError  +-- ImportError  +-- LookupError  | +-- IndexError  | +-- KeyError  +-- MemoryError  +-- NameError  | +-- UnboundLocalError  +-- OSError  | +-- BlockingIOError  | +-- ChildProcessError  | +-- ConnectionError  | | +-- BrokenPipeError  | | +-- ConnectionAbortedError  | | +-- ConnectionRefusedError  | | +-- ConnectionResetError  | +-- FileExistsError  | +-- FileNotFoundError  | +-- InterruptedError  | +-- IsADirectoryError  | +-- NotADirectoryError  | +-- PermissionError  | +-- ProcessLookupError  | +-- TimeoutError  +-- ReferenceError  +-- RuntimeError  | +-- NotImplementedError  +-- SyntaxError  | +-- IndentationError  | +-- TabError  +-- SystemError  +-- TypeError  +-- ValueError  | +-- UnicodeError  | +-- UnicodeDecodeError  | +-- UnicodeEncodeError  | +-- UnicodeTranslateError  +-- Warning  +-- DeprecationWarning  +-- PendingDeprecationWarning  +-- RuntimeWarning  +-- SyntaxWarning  +-- UserWarning  +-- FutureWarning  +-- ImportWarning  +-- UnicodeWarning  +-- BytesWarning  +-- ResourceWarning |
| AttributeError | 尝试访问未知的对象属性 |
| EOFError | 用户输入文件末尾标志EOF（Ctrl+d） |
| FloatingPointError | 浮点计算错误 |
| GeneratorExit | generator.close()方法被调用的时候 |
| ImportError | 导入模块失败的时候 |
| IndexError | 索引超出序列的范围 |
| KeyError | 字典中查找一个不存在的关键字 |
| KeyboardInterrupt | 用户输入中断键（Ctrl+c） |
| MemoryError | 内存溢出（可通过删除对象释放内存） |
| NameError | 尝试访问一个不存在的变量 |
| NotImplementedError | 尚未实现的方法 |
| OSError | 操作系统产生的异常（例如打开一个不存在的文件） |
| OverflowError | 数值运算超出最大限制 |
| ReferenceError | 弱引用（weak reference）试图访问一个已经被垃圾回收机制回收了的对象 |
| RuntimeError | 一般的运行时错误 |
| StopIteration | 迭代器没有更多的值 |
| SyntaxError | Python的语法错误 |
| IndentationError | 缩进错误 |
| TabError | Tab和空格混合使用 |
| SystemError | Python编译器系统错误 |
| SystemExit | Python编译器进程被关闭 |
| TypeError | 不同类型间的无效操作 |
| UnboundLocalError | 访问一个未初始化的本地变量（NameError的子类） |
| UnicodeError | Unicode相关的错误（ValueError的子类） |
| UnicodeEncodeError | Unicode编码时的错误（UnicodeError的子类） |
| UnicodeDecodeError | Unicode解码时的错误（UnicodeError的子类） |
| UnicodeTranslateError | Unicode转换时的错误（UnicodeError的子类） |
| ValueError | 传入无效的参数 |
| ZeroDivisionError | 除数为零 |

try:

检测范围

except OSError as reason:

print('文件出错啦T\_T\n错误原因是：' + str(reason))

except (OSError, TypeError):

print('出错啦T\_T\n ')

finally:

执行必须执行的内容

try:

f = open('My\_File.txt') # 当前文件夹中并不存在"My\_File.txt"这个文件

print(f.read())

except OSError as reason:

print('出错啦：' + str(reason))

finally:

if 'f' in locals(): # 如果文件对象变量存在当前局部变量符号表的话，说明打开成功

f.close()

* 将 else 语句与循环语句（while 和 for 语句）进行搭配，那么只有在循环正常执行完成后才会执行 else 语句块的内容。
* with 语句会自动处理文件的打开和关闭，如果中途出现异常，会执行清理代码，然后确保文件自动关闭。
* with 语句处理多个项目的时候，可以用逗号隔开写成一条语句 with A() as a, B() as b:
* 使用条件语句的代码非常直观明了，但是效率不高。因为程序会两次访问字典的键，一次判断是否存在（例如 if name in contacts），一次获得值（例如 print(name + ' : ' + contacts[name])）。如果利用异常解决方案，我们可以简单避开每次需要使用 in 判断是否键存在字典中的操作。因为只要当键不存在字典中时，会触发 KeyError 异常。

## 类与对象

* 对象中的属性和方法，在编程中实际是变量和函数。类和对象的关系就是抽象和具体的关系。
* 封装：对外部隐藏对象的工作细节
* 继承：子类自动共享父类之间数据和方法的机制
* 多态：可以对不同类的对象调用相同的方法，产生不同的结果
* 方法中的 self 其实就是实例对象的唯一标志，可以让python知道是哪个对象在调用方法。
* 魔法方法的写法为：\_\_init\_\_()
* 私有变量或方法的写法为：\_\_name or \_\_getName(self)
* 如果一个类 A 继承自另一个类 B，就把这个 A 称为 B 的子类，把 B 称为 A 的父类、基类或超类。继承可以使得子类具有父类的各种属性和方法，而不需要再次编写相同的代码（偷懒）。在子类继承父类的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类的原有属性和方法，使其获得与父类不同的功能。另外，为子类追加新的属性和方法也是常见的做法。
* \_\_init\_\_ 特殊方法不应当返回除了 None 以外的任何对象。所以在该方法中不需要return
* 覆盖父类方法，可以将函数体内容写 pass
* super 函数超级之处在于你不需要明确给出任何父类的名字，它会自动帮您找出所有父类以及对应的方法。由于你不用给出父类的名字，这就意味着你如果需要改变了类继承关系，你只要改变 class 语句里的父类即可，而不必在大量代码中去修改所有被继承的方法。

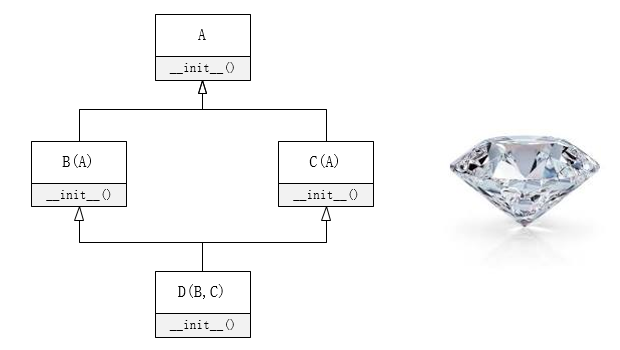


图 1 钻石继承（菱形继承）

* 钻石继承（菱形继承）会带来什么问题？

多重继承容易导致钻石继承（菱形继承）问题，上边代码实例化 D 类后我们发现 A 被前后进入了两次

这有什么危害？我举个例子，假设 A 的初始化方法里有一个计数器，那这样 D实例化，A 的计数器就跑两次（如果遭遇多个钻石结构重叠还要更多），很明显是不符合程序设计的初衷的（程序应该可控，而不能受到继承关系影响）。

* 如何避免钻石继承（菱形继承）问题？

为解决这个问题，Python 使用了一个叫“方法解析顺序（Method Resolution Order，MRO）”的东西，还用了一个叫 C3 的算法。该算法相对来说比较复杂（有兴趣深入算法的朋友可以阅读：https://www.python.org/download/releases/2.3/mro）当然我这里愿意跟你解释下 MRO 的顺序基本就是：在避免同一类被调用多次的前提下，使用广度优先和从左到右的原则去寻找需要的属性和方法。在继承体系中，C3 算法确保同一个类只会被搜寻一次。例子中，如果一个属性或方法在 D 类中没有被找到，Python 就会搜寻 B 类，然后搜索 C类，如果都没有找到，会继续搜索 B 的基类 A，如果还是没有找到，则抛出“AttributeError”异常。

其实直接使用super函数即可避免出现钻石继承。

* 类对象是当你这个类定义完的时候，类定义就变成类对象，可以直接通过“类名.属性”或者“类名.方法名()”引用或使用相关的属性或方法。
* 如果对象的属性跟方法名相同，属性会覆盖方法。

class C:

num = 0

def \_\_init\_\_(self):

self.x = 4

self.y = 5

C.count = 6

num 和 count 是类属性（静态变量），x 和 y 是实例属性。大多数情况下，你应该考虑使用实例属性，而不是类属性（类属性通常仅用来跟踪与类相关的值）。

* issubclass(class, classinfo) 函数判断一个类是否为另一个类的子类，如果第一个参数（class）是第二个参数（classinfo）的一个子类，则返回 True，否则返回 False。
  + 一个类被认为是其自身的子类
  + classinfo 可以是类对象组成的元祖，只要 class 与其中任何一个候选类的子类，则返回 True
  + 在其他情况下，会抛出一个 TypeError 异常
* isinstance(object, classinfo) 函数判断对象 a 是否为 类 A 的实例对象，如果第一个参数（object）是第二个参数（classinfo）的实例对象，则返回 True，否则返回 False。
  + 如果 object是 classinfo 的子类的一个实例，也符合条件
  + 如果第一个参数不是对象，则永远返回False
  + classinfo 可以是类对象组成的元祖，只要class与其中任何一个候选类的子类，则返回 True
  + 如果第二个参数不是类或者由类对象组成的元祖，会抛出一个 TypeError 异常
* hasattr(object, name) 函数判断属性是否存在，如果存在再访问（第一个参数（object）是对象，第二个参数（name）是属性名的字符串形式）
* getattr(object, name[, default]) 函数并设置 default 参数（返回对象指定的属性值，如果指定的属性不存在，返回default（可选参数）的值）
* property() 函数允许编程人员轻松、有效地管理属性访问

## 魔法方法

* \_\_new\_\_ 是在一个对象实例化的时候所调用的第一个方法。它跟其他魔法方法不同，它的第一个参数不是 self 而是这个类（cls），而其他的参数会直接传递给 \_\_init\_\_ 方法的。\_\_new\_\_ 方法主要任务是返回一个实例对象，通常是参数 cls 这个类的实例化对象，当然你也可以返回其他对象。
* 如果说 \_\_init\_\_ 和 \_\_new\_\_ 方法是对象的构造器的话，那么 Python 也提供了一个析构器，叫做 \_\_del\_\_ 方法。当对象将要被销毁的时候，这个方法就会被调用。但一定要注意的是，并非 del x 就相当于自动调用 x.\_\_del\_\_()，\_\_del\_\_ 方法是当垃圾回收机制回收这个对象的时候调用的。del x只是删除了指向实例的指针，而在没有指针指向实例时，则会调用\_\_del\_\_方法去进行垃圾回收。
* 从静态类型语言转向动态类型语言的用户通常试图添加一些静态的（在运行之前的）类型检查，从而影响了鸭子类型的益处和可伸缩性，并约束了语言的动态特性（Python 文档中有一句：鸭子类型应避免使用 type() 或 isinstance() 等测试类型是否合法）。

|  |  |
| --- | --- |
| **魔法方法** | **含义** |
| **基本的魔法方法** | |
| \_\_new\_\_(cls[, ...]) | 1. \_\_new\_\_ 是在一个对象实例化的时候所调用的第一个方法 2. 它的第一个参数是这个类，其他的参数是用来直接传递给 \_\_init\_\_ 方法 3. \_\_new\_\_ 决定是否要使用该 \_\_init\_\_ 方法，因为 \_\_new\_\_ 可以调用其他类的构造方法或者直接返回别的实例对象来作为本类的实例，如果 \_\_new\_\_ 没有返回实例对象，则 \_\_init\_\_ 不会被调用 4. \_\_new\_\_ 主要是用于继承一个不可变的类型比如一个 tuple 或者 string |
| \_\_init\_\_(self[, ...]) | 构造器，当一个实例被创建的时候调用的初始化方法 |
| \_\_del\_\_(self) | 析构器，当一个实例被销毁的时候调用的方法 |
| \_\_call\_\_(self[, args...]) | 允许一个类的实例像函数一样被调用：x(a, b) 调用 x.\_\_call\_\_(a, b) |
| \_\_len\_\_(self) | 定义当被 len() 调用时的行为 |
| \_\_repr\_\_(self) | 定义当被 repr() 调用时的行为 |
| \_\_str\_\_(self) | 定义当被 str() 调用时的行为 |
| \_\_bytes\_\_(self) | 定义当被 bytes() 调用时的行为 |
| \_\_hash\_\_(self) | 定义当被 hash() 调用时的行为 |
| \_\_bool\_\_(self) | 定义当被 bool() 调用时的行为，应该返回 True 或 False |
| \_\_format\_\_(self, format\_spec) | 定义当被 format() 调用时的行为 |
| **有关属性** | |
| **\_\_getattr\_\_(self, name)** | **定义当用户试图获取一个不存在的属性时的行为** |
| **\_\_getattribute\_\_(self, name)** | **定义当该类的属性被访问时的行为** |
| **\_\_setattr\_\_(self, name, value)** | **定义当一个属性被设置时的行为** |
| **\_\_delattr\_\_(self, name)** | **定义当一个属性被删除时的行为** |
| \_\_dir\_\_(self) | 定义当 dir() 被调用时的行为 |
| **描述符就是将某种特殊类型的类的实例指派给另一个类的属性** | |
| **\_\_get\_\_(self, instance, owner)** | **用于访问属性，返回属性的值**  **其中self是这个类本身，instance是调用该类的类实例，owner是调用该类的类本身** |
| **\_\_set\_\_(self, instance, value)** | **将在属性分配操作中调用，不返回任何内容** |
| **\_\_delete\_\_(self, instance)** | **控制删除操作，不返回任何内容** |
| **比较操作符** | |
| \_\_lt\_\_(self, other) | 定义小于号的行为：x < y 调用 x.\_\_lt\_\_(y) |
| \_\_le\_\_(self, other) | 定义小于等于号的行为：x <= y 调用 x.\_\_le\_\_(y) |
| \_\_eq\_\_(self, other) | 定义等于号的行为：x == y 调用 x.\_\_eq\_\_(y) |
| \_\_ne\_\_(self, other) | 定义不等号的行为：x != y 调用 x.\_\_ne\_\_(y) |
| \_\_gt\_\_(self, other) | 定义大于号的行为：x > y 调用 x.\_\_gt\_\_(y) |
| \_\_ge\_\_(self, other) | 定义大于等于号的行为：x >= y 调用 x.\_\_ge\_\_(y) |
| **算数运算符** | |
| \_\_add\_\_(self, other) | 定义加法的行为：+ |
| \_\_sub\_\_(self, other) | 定义减法的行为：- |
| \_\_mul\_\_(self, other) | 定义乘法的行为：\* |
| \_\_truediv\_\_(self, other) | 定义真除法的行为：/ |
| \_\_floordiv\_\_(self, other) | 定义整数除法的行为：// |
| \_\_mod\_\_(self, other) | 定义取模算法的行为：% |
| \_\_divmod\_\_(self, other) | 定义当被 divmod() 调用时的行为 |
| \_\_pow\_\_(self, other[, modulo]) | 定义当被 power() 调用或 \*\* 运算时的行为 |
| \_\_lshift\_\_(self, other) | 定义按位左移位的行为：<< |
| \_\_rshift\_\_(self, other) | 定义按位右移位的行为：>> |
| \_\_and\_\_(self, other) | 定义按位与操作的行为：& |
| \_\_xor\_\_(self, other) | 定义按位异或操作的行为：^ |
| \_\_or\_\_(self, other) | 定义按位或操作的行为：| |
| **反运算** | |
| \_\_radd\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rsub\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rmul\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rtruediv\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rfloordiv\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rmod\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rdivmod\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rpow\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rlshift\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rrshift\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rand\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rxor\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_ror\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| **增量赋值运算** | |
| \_\_iadd\_\_(self, other) | 定义赋值加法的行为：+= |
| \_\_isub\_\_(self, other) | 定义赋值减法的行为：-= |
| \_\_imul\_\_(self, other) | 定义赋值乘法的行为：\*= |
| \_\_itruediv\_\_(self, other) | 定义赋值真除法的行为：/= |
| \_\_ifloordiv\_\_(self, other) | 定义赋值整数除法的行为：//= |
| \_\_imod\_\_(self, other) | 定义赋值取模算法的行为：%= |
| \_\_ipow\_\_(self, other[, modulo]) | 定义赋值幂运算的行为：\*\*= |
| \_\_ilshift\_\_(self, other) | 定义赋值按位左移位的行为：<<= |
| \_\_irshift\_\_(self, other) | 定义赋值按位右移位的行为：>>= |
| \_\_iand\_\_(self, other) | 定义赋值按位与操作的行为：&= |
| \_\_ixor\_\_(self, other) | 定义赋值按位异或操作的行为：^= |
| \_\_ior\_\_(self, other) | 定义赋值按位或操作的行为：|= |
| **一元操作符** | |
| \_\_pos\_\_(self) | 定义正号的行为：+x |
| \_\_neg\_\_(self) | 定义负号的行为：-x |
| \_\_abs\_\_(self) | 定义当被 abs() 调用时的行为 |
| \_\_invert\_\_(self) | 定义按位求反的行为：~x |
| **类型转换** | |
| \_\_complex\_\_(self) | 定义当被 complex() 调用时的行为（需要返回恰当的值） |
| \_\_int\_\_(self) | 定义当被 int() 调用时的行为（需要返回恰当的值） |
| \_\_float\_\_(self) | 定义当被 float() 调用时的行为（需要返回恰当的值） |
| \_\_round\_\_(self[, n]) | 定义当被 round() 调用时的行为（需要返回恰当的值） |
| \_\_index\_\_(self) | 1. 当对象是被应用在切片表达式中时，实现整形强制转换 2. 如果你定义了一个可能在切片时用到的定制的数值型,你应该定义 \_\_index\_\_ 3. 如果 \_\_index\_\_ 被定义，则 \_\_int\_\_ 也需要被定义，且返回相同的值 |
| **上下文管理（with 语句）** | |
| \_\_enter\_\_(self) | 1. 定义当使用 with 语句时的初始化行为 2. \_\_enter\_\_ 的返回值被 with 语句的目标或者 as 后的名字绑定 |
| \_\_exit\_\_(self, exc\_type,  exc\_value, traceback) | 1. 定义当一个代码块被执行或者终止后上下文管理器应该做什么 2. 一般被用来处理异常，清除工作或者做一些代码块执行完毕之后的日常工作 |
| **容器类型** | |
| \_\_len\_\_(self) | 定义当被 len() 调用时的行为（返回容器中元素的个数） |
| \_\_getitem\_\_(self, key) | 定义获取容器中指定元素的行为，相当于 self[key] |
| \_\_setitem\_\_(self, key, value) | 定义设置容器中指定元素的行为，相当于 self[key] = value |
| \_\_delitem\_\_(self, key) | 定义删除容器中指定元素的行为，相当于 del self[key] |
| \_\_iter\_\_(self) | 定义当迭代容器中的元素的行为 |
| \_\_reversed\_\_(self) | 定义当被 reversed() 调用时的行为 |
| \_\_contains\_\_(self, item) | 定义当使用成员测试运算符（in 或 not in）时的行为 |

* Python 什么时候会调用到反运算的魔法方法？例如 a + b，如果 a 对象的 \_\_add\_\_ 方法没有实现或者不支持相应的操作，那么 Python 就会自动调用 b 的 \_\_radd\_\_ 方法。
* 在继承的类中调用基类，使用 super() 这个 BIF 函数。

|  |  |
| --- | --- |
| 如果我要继承的基类是动态的（有时候是 A，有时候是 B），以便基类可以随意改变。  可以先为基类定义一个别名，在类定义的时候，使用别名代替你要继承的基类。如此，当你想要改变基类的时候，只需要修改给别名赋值的那个语句即可。顺便说一下，当你的资源是视情况而定的时候，这个小技巧很管用。  BaseAlias = BaseClass # 为基类取别名  class Derived(BaseAlias):  def meth(self):  BaseAlias.meth(self) # 通过别名访问基类 | 类的静态属性很简单，在类中直接定义的变量（没有 self.）就是静态属性。引用类的静态属性使用”类名.属性名”的形式。  类的静态属性应用（计算该类被实例化的次数）：  class C:  count = 0 # 静态属性  def \_\_init\_\_(self):  C.count = C.count + 1 # 类名.属性名的形式引用  def getCount(self):  return C.count |

* 静态方法是类的特殊方法，静态方法只需要在普通方法的前边加上 @staticmethod 修饰符即可。最大的优点是：不会绑定到实例对象上，换而言之就是节省开销。静态方法并不需要 self 参数，因此即使是使用对象去访问，self 参数也不会传进去。

def \_\_setattr\_\_(self, name, value):

**×** self.name = value + 1

**√** self.\_\_dict\_\_[name] = value + 1

**√** super().\_\_setattr\_\_(name, value+1)

这段代码试图在对象的属性发生赋值操作的时候，将实际的值 +1赋值给相应的属性。但这么写法是错误的，因为每当属性被赋值的时候， \_\_setattr\_\_() 会被调用，而里边的 self.name = value + 1 语句又会再次触发 \_\_setattr\_\_() 调用，导致无限递归。

* 描述符是什么：

有时候，某个应用程序可能会有一个相当微妙的需求，需要你设计一些更为复杂的操作来响应（例如每当属性被访问时，你也许想创建一个日志记录）。最好的解决方案就是编写一个用于执行这些“更复杂的操作”的特殊函数，然后指定它在属性被访问时运行。那么一个具有这种函数的对象被称之为描述符。

往再简单了说，描述符就是一个类，一个至少实现 \_\_get\_\_()、\_\_set\_\_() 或 \_\_delete\_\_() 三个特殊方法中的任意一个的类。

>>> class MyDes:

def \_\_init\_\_(self, value = None):

self.val = value

def \_\_get\_\_(self, instance, owner):

return self.val \*\* 2

>>> class Test:

def \_\_init\_\_(self):

self.x = MyDes(3)

>>> test = Test()

>>> test.x **#访问实例层次上的描述符 x，只会返回描述符本身。为了让描述符能够正常工作，它们必须定义在类的层次上。如果你不这么做，那么 Python 无法自动为你调用 \_\_get\_\_ 和 \_\_set\_\_ 方法。<\_\_main\_\_.MyDes object at 0x1058e6f60>**

* **迭代**是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值。迭代器就是实现了\_\_next\_\_()方法的对象（用于遍历容器中的数据）。
* **迭代器**性质决定没有办法回退，只能往前进行迭代。但这并不是什么很大的缺点，因为我们几乎不需要在迭代途中进行回退操作。迭代器通过 \_\_next\_\_() 方法每次返回一个元素，并指向下一个元素。如果当前位置已无元素，通过抛出 StopIteration 异常表示。对于原生支持随机访问的数据结构（如tuple、list），可以使用迭代器或者下标索引的形式访问，但对于无法随机访问的数据结构 set 而言，迭代器是唯一的访问元素的方式。
* 所谓的协同程序就是可以运行的独立函数调用，函数可以暂停或者挂起，并在需要的时候从程序离开的地方继续或者重新开始。Python 是通过**生成器**来实现类似于协同程序的概念：生成器可以暂时挂起函数，并保留函数的局部变量等数据，然后在再次调用它的时候，从上次暂停的位置继续执行下去。生成器只需要一个 yield 语句即可，但它内部会自动创建 \_\_iter\_\_() 和 \_\_next\_\_() 方法并且是基于迭代器来实现的，使得函数可以“保留现场”，当下一次执行该函数是从上一次结束的地方开始，而不是重头再来。

## 模块

* 导入模块的三种方法：

第一种：import 模块名

第二种：from 模块名 import 函数名

第三种：import 模块名 as 新名字

* 如果你不想模块中的某个属性被 from…import \* 导入，那么你可以给你不想导入的属性名称的前边加上一个下划线（\_）。不过需要注意的是，如果使用 import … 导入整个模块，或者显式地使用 import xx.\_oo 导入某个属性，那么这个隐藏的方法就不起作用了。
* 所有模块都有一个 \_\_name\_\_ 属性，\_\_name\_\_ 的值取决于如何应用模块，在作为独立程序运行的时候，\_\_name\_\_ 属性的值是 '\_\_main\_\_'，而作为模块导入的时候，这个值就是该模块的名字了。
* 如果你不想将相关的模块文件放在当前文件夹内，那最好的选择是放在 site-packages 文件夹，因为它就是用来存放你的模块文件的。
* Python 如何区分一个文件夹是普通文件夹还是包？
* 看文件夹中是否有 \_\_init\_\_.py 文件。必须在包文件夹中创建一个 \_\_init\_\_.py 的模块文件，内容可以为空。可以是一个空文件，也可以写一些初始化代码。这个是 Python 的规定，用来告诉 Python 将该目录当成一个包来处理。

## 爬虫

* URI 是统一资源标识符（Universal Resource Identifier），URL 是统一资源定位符（Universal Resource Locator）。用一句话概括它们的区别：URI 是用字符串来标识某一互联网资源，而 URL 则是表示资源的地址（我们说某个网站的网址就是 URL），因此 URI 属于父类，而 URL 属于 URI 的子类。
* 设计爬虫时要注意：不要重复爬取同一个 URL 的内容。假设你没做这方面的预防，如果一个 URL 的内容中包含该 URL 本身，那么就会陷入无限递归。
* 在网站的根目录下创建并编辑 robots.txt 文件，用于表明您不希望搜索引擎抓取工具访问您网站上的哪些内容。此文件使用的是 Robots 排除标准，该标准是一项协议，所有正规搜索引擎的蜘蛛均会遵循该协议爬取。既然是协议，那就是需要大家自觉尊重，所以该协议一般对非法爬虫无效。
* urlopen() 方法的 timeout 参数用于设置连接的超时时间，单位是秒。
* 从 urlopen() 返回的对象中获取 HTTP 状态码
  + response = urllib.request.urlopen(url)
  + code = response.getcode()
* 客户端和服务器之间进行请求-响应时，最常用的是GET和POST。HTTP 是基于请求-响应的模式，发出请求的永远是客户端，做出响应的永远是服务端。
* User-Agent 属性通常是记录所使用的浏览器类型、操作系统、浏览器内核等信息的标识。
* urlopen 函数有一个 data 参数，如果给这个参数赋值，那么 HTTP 的请求就是使用 POST 方式；如果 data 的值是 NULL，也就是默认值，那么 HTTP 的请求就是使用 GET 方式。
* decode 的作用是将其他编码的字符串转换成 unicode 编码，相反，encode 的作用是将 unicode 编码转换成其他编码的字符串。
* JSON 是一种轻量级的数据交换格式，说白了这里就是用字符串把 Python 的数据结构封装起来，便与存储和使用。
* 代理步骤：
  + 参数是一个字典{'类型' : '代理ip:端口号'} ; proxy\_support = urllib.request.ProxyHandler({})
  + 定制、创建一个opener; opener = urllib.request.build\_opener(proxy\_support)
  + 安装opener; urllib.request.install\_opener(opener)
  + 调用opener; opener.open(url)
* 服务器端通过发送的 HTTP 头中的 User-Agent 来进行识别浏览器与非浏览器，还能区分各个浏览器。
* 客户端首先发出 Request，服务器收到后返回 Response，通过add\_header()方法向Request对象中添加headers。
* 代理服务器：将信息传给代理服务器，代理服务器替你向你要访问的服务器发送请求，然后在将服务器返回的内容返回给你。因为有“丢包”现象发生，所以多了一个中间人就意味着会多一层发生“丢包”的几率，且大多数代理并不只为一个人服务，尤其是免费代理。PS：大家想做“坏坏”的事情时可以考虑多几层代理，一般来说路由器日志并不会保存很长时间，几层代理后，基本很难查到是谁请求的。
* cookie 可以分成两类：
  + 一类是即时过期的 cookies，称为“会话” cookies，当浏览器关闭时（这里是 python 的请求程序）自动清除
  + 另一类是有期限的 cookies，由浏览器进行存储，并在下一次请求该网站时自动附带（如果没过期或清理的话）

## 正则表达式

* 重复使用某个正则表达式，可以先将正则表达式编译成模式对象。使用re.compile()方法来编译。

## Scrapy爬虫

* 使用Scrapy抓取一个网站需要四个步骤：
  + 创建一个Scrapy项目
    - scrapy startproject tutorial
  + 定义Item容器
    - Item是保存爬取到的数据的容器，其使用方法和python字典类似，并且提供了额外保护机制来避免拼写错误导致的未定义字段错误。
  + 编写爬虫类Spider
    - Spider是用户编写用于从网站上爬取数据的类，其包含了一个用于下载的初始URL，然后是如何跟进网页中的链接以及如何分析页面中的内容，还有提取生成item的方法。
  + 存储内容
* Scrapy使用一种基于XPath和CSS的表达式机制：Scrapy Selectors
* Selector是一个选择器，有四种基本方法：
  + xpath()：传入xpath表达式，返回该表达式所对应的所有节点的selector list列表
  + css()：传入CSS表达式，返回该表达式所对应的所有结点的selector list列表
  + extract()：序列化该节点为unicode字符串并返回list
  + re()：根据传入的正则表达式对数据进行提取，返回unicode字符串list列表
* xpath部分示例：
  + /html/head/title：选择HTML文档中<head>标签内的<title>元素
  + /html/head/title/text()：选择上面提到的<title>元素的文字
  + //td：选择所有的<td>元素
  + //div[@class="mine"]：选择所有具有class="mine"属性的div元素