安装

编辑脚本

运行第一个脚本

输入输出，print，input

# **Python 基础：**

数据类型；

编码和解码

ASII码，unicode码，utf-8，gb2313

encode()；decode()

Bytes: 由于Python的字符串类型是str，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。Python对bytes类型的数据用带b前缀的单引号或双引号表示：x = b'ABC;

如果bytes中只有一小部分无效的字节，可以传入errors='ignore'忽略错误的字节：b'\xe4\xb8\xad\xff'.decode('utf-8', errors='ignore')

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

格式化字符：

1：%；2：format

'Hi, %s, you have $%d.' % ('Michael', 1000000)

'Hello, {0}, 成绩提升了 {1:.1f}%'.format('小明', 17.125)

列表和组元

二者均为有序集合，前者可变，后者不可变。

列表：append，insert,pop

组元：初始化后不可更改

程序结构语句：

if if

else elif

else

for while

注意：continue的作用是提前结束本轮循环，并直接开始下一轮循环

break语句可以在循环过程中直接退出循环，而continue语句可以提前结束本轮循环，并直接开始下一轮循环。这两个语句通常都必须配合if语句使用。

字典和集合

字典：get, pop;

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是dict的key必须是**不可变对象**。

Set:: add, remove

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key

# 函数

概念

调用

**定义：**定义函数时，需要确定函数名和参数个数；如果有必要，可以先对参数的数据类型做检查；函数体内部可以用return随时返回函数结果；函数执行完毕也没有return语句时，自动return None。函数可以同时返回多个值，但其实就是一个tuple。

**函数参数：**

定义默认参数要牢记一点：默认参数必须指向不变对象！

**可变参数：**定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数；

**关键字参数：**可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict；

person('Jack', 24, \*\*extra)： \*\*extra表示把extra这个dict的所有key-value用关键字参数传入到函数的\*\*kw参数，kw将获得一个dict，注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra

**命名关键字：**和关键字参数\*\*kw不同，命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。

**参数组合：**在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

**递归函数：**在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

汉诺塔递归算法：有n个盘在A塔，要全部移到C塔，要求大盘不能在小盘上；

首先对1个盘，直接从A->C

对2个盘，将小盘先移到B，这是可忽略B塔上的盘，相当于只有一个盘的ABC，因此直接将大盘从A移到C，这时可以忽略C上的大盘，视C塔为空，这样BAC就又构成1个盘的情况，直接从B->C

对n个盘，因此可将A的n-1个盘视为整体移动到B，然后最大盘就可以从A移动到C，这是视AC塔上均为空，就重新构成BAC的n-1个盘的问题，依次递归上三步。

因此归为三步：1.在ACB的顺序下执行了一阶汉诺塔的移法 ；2.从A->C移动了最大盘 ；3.在BAC的顺序下执行了一阶汉诺塔的移法

# 高级特性

**切片：**

**迭代：**如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：from collections import Iterable

isinstance('abc', Iterable) *# str是否可迭代*

Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：for i, value in enumerate(['A', 'B', 'C'])

**列表生成：**列表生成式则可以用一行语句代替循环生成list

[x \* x for x in range(1, 11)]

[m + n for m in 'ABC' for n in 'XYZ']

[d for d in os.listdir('.')] *# os.listdir可以列出文件和目录*

**生成器：**在Python中，这种一边循环一边计算的机制，称为生成器：generator

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator；g = (x \* x **for** x **in** range(10))

generator是非常强大的工具，在Python中，可以简单地把列表生成式改成generator，也可以通过函数实现复杂逻辑的generator。要理解generator的工作原理，它是在for循环的过程中不断计算出下一个元素，并在适当的条件结束for循环。对于函数改成的generator来说，遇到return语句或者执行到函数体最后一行语句，就是结束generator的指令，for循环随之结束

**迭代器：**我们已经知道，可以直接作用于for循环的数据类型有以下几种：一类是集合数据类型，如list、tuple、dict、set、str等；一类是generator，包括生成器和带yield的generator function。这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。可以使用isinstance()判断一个对象是否是Iterable对象：

from collections import Iterable

isinstance([], Iterable)

生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。把list、dict、str等Iterable变成Iterator可以使用iter()函数

凡是可作用于for循环的对象都是Iterable类型；凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。Python的for循环本质上就是通过不断调用next()函数实现的.

# 函数式编程

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言

Functools 模块： **reduce, filter,**

**高阶函数：**既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

变量可以指向函数；函数名也是一个变量；

**Map/reduce:**

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。（map,有for循环迭代的意味)

再看reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)（reduce，有累计迭代的意味）

**filter:** 和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

用filter求素数思路：对一个数据列表L，依次使用2，及奇数来过滤L，最后剩下的就是素数。

**sorted:** Python内置的sorted()函数就可以对list进行排序；sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序；

sorted([36, 5, -12, 9, -21], key=abs)

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True

**返回函数：**返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量

**匿名函数：**当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。关键字lambda表示匿名函数，冒号前面的x表示函数参数

**装饰器：**由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

装饰器就是能够产生函数调用日志的函数，他没有显式调用，而是在某个函数调用时，会自动运行它的装饰器。

首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。这会出错误，因此需要将其改为原始函数的名。不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的

*装饰器实际上就是为了给某程序增添功能，但该程序已经上线或已经被使用，那么就不能大批量的修改源代码，这样是不科学的也是不现实的，因为就产生了装饰器，使得其满足*：

1、不能修改被装饰的函数的源代码

2、不能修改被装饰的函数的调用方式

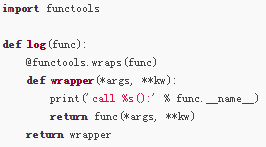
3、满足1、2的情况下给程序增添功能

《函数+实参高阶函数+返回值高阶函数+嵌套函数+语法糖 = 装饰器》

装饰器的根本目的是在不改变源代码的情况下，为源代码添加功能；

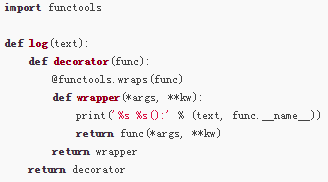
1、不带参数的装饰器

装饰器的参数是一个函数名，其返回的也是一个函数名，



2、带参数的装饰器

这种类型的装饰器，是通过在原装饰器外再加一层嵌套函数，来给装饰器添加输入参数的。



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

**import** functools

int2 = functools.partial(int, base=2)

等价于下面的函数

**def** **int2**(x, base=2):

**return** int(x, base)

简单总结functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

# 模块

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

模块是一组Python代码的集合，可以使用其他模块，也可以被其他模块使用。

创建自己的模块时，要注意：

模块名要遵循Python变量命名规范，不要使用中文、特殊字符；

模块名不要和系统模块名冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行import abc，若成功则说明系统存在此模块。

注意作用域

# 面向对象编程

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是Student这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个print\_score消息，让对象自己把自己的数据打印出来。

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法

**类和实例**

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去：

class Student(object):

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

注意：特殊方法“\_\_init\_\_”前后分别有两个下划线！！！

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；

方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；

通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

**访问限制**

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问；

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

**继承多态**

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

**当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，我们就获得了继承的另一个好处：多态。 这就是指：只要一个函数接受基类对象，那么这个函数就可以接受这个基类的所有派生类对象，并且调用对应派生类中的重载的基类方法（因为派生类中重载的方法，覆盖了基类的方法）**

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

多态的好处就是，当我们需要传入Dog、Cat、Tortoise……时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：对扩展开放：允许新增Animal子类；对修改封闭：不需要修改依赖Animal类型的run\_twice()等函数。

继承可以基于现有的类来创建新的类，新类具有基类的所有属性和方法；

而在新类中可以添加新的属性和方法；

在新类中改写基类中存在的方法，这个过程就叫做重载

**获取对象信息**

首先，我们来判断对象类型，使用type()函数：

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：dir(‘ABC’)；

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态

可以传入一个default参数，如果属性不存在，就返回默认值：

>>>getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404

>>>404

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法

我们自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法

**类属性和实例属性**

实例属性属于各个实例所有，互不干扰；

类属性属于类所有，所有实例共享一个属性；

不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。

# 面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

**使用\_\_slots\_\_：**主要用于限制实例可以添加的属性

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性



使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的；除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

**@property：重要作用是把一个方法变为属性，@property将为这个方法自动添加一个装饰器，从而可以在装饰器中对这个方法形式的属性进行检查，避免出错。**

**装饰器既可以给函数添加功能，也可以给对象添加功能**

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的；

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作；

@property广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

**多重继承：指可以同时继承多个类**

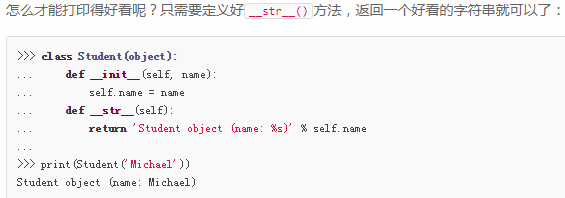
通过多重继承，一个子类就可以同时获得多个父类的所有功能。由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用MixIn的设计。

**定制类:通过这些定制属性，可以使得自己创建的类和标准类具有相同的调用方式**

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

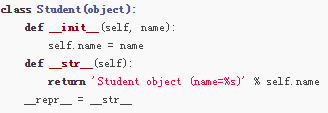
\_\_str\_\_：



\_\_repr\_\_:

\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

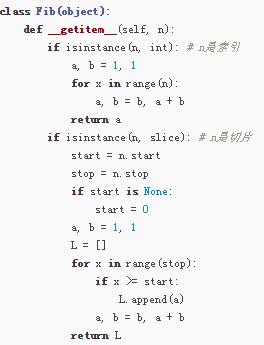
解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：



\_\_iter\_\_：如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环；

\_\_getitem\_\_：定义这个方法可以是类具有与列表类似的索引方法

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法；\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断



此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

\_\_getattr\_\_:通过这个方法可以对类中不存在的属性，添加到实例中

如果调用的属性不存在，就会报错，要避免这个错误，除了可以加上一个score属性外，Python还有另一个机制，那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。注意，只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

\_\_call\_\_:任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象

**枚举：**Enum可以把一组相关常量定义在一个class中，且class不可变，而且成员可以直接比较。

**使用元类：**

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义

要创建一个class对象，type()函数依次传入3个参数：

1、class的名称；

2、继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；

3、class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

**metaclass：**

# 错误、调试、测试

**错误**

高级语言通常都内置了一套try...except...finally...的错误处理机制，Python也不例外；

由于没有错误发生，所以except语句块不会被执行，但是finally如果有，则一定会被执行（可以没有finally语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误

 出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。

**记录错误：**如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的logging模块可以非常容易地记录错误信息，同样是出错，但程序打印完错误信息后会继续执行，并正常退出，通过配置，logging还可以把错误记录到日志文件里，方便事后排查。

**抛出错误：**因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例

**调试**

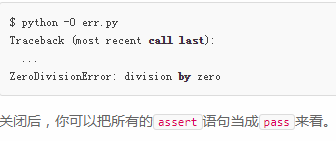
**1、Print:** 第一种方法简单直接粗暴有效，就是用print()把可能有问题的变量打印出来看看

print('>>> n = %d' % n)

**2、assert:** 凡是用print()来辅助查看的地方，都可以用断言（assert）来替代

**assert** n != 0, 'n is zero!'

程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert



**3、logging：**把print()替换为logging是第3种方式，和assert比，logging不会抛出错误，而且可以输出到文件

**import** logging

s = '0'

n = **int**(s)

logging.info('n = %d' % n)

print(10 / n)

logging.info()就可以输出一段文本。运行，发现除了ZeroDivisionError，没有任何信息。怎么回事？别急，在import logging之后添加一行配置再试试：

**import** logging

logging.basicConfig(level=logging.INFO)

这就是logging的好处，它允许你指定记录信息的级别，有debug，info，warning，error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

**4、pdb:** 第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令l来查看代码；输入命令n可以单步执行代码；任何时候都可以输入命令p 变量名来查看变量；输入命令q结束调试，退出程序。

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

这个方法也是用pdb，但是不需要单步执行，我们只需要import pdb，然后，在可能出错的地方放一个pdb.set\_trace()，就可以设置一个断点；运行代码，程序会自动在pdb.set\_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行；

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。**虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。**

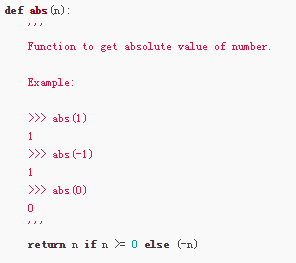
单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug

**文档测试**



无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

注意到最后3行代码。当模块正常导入时，doctest不会被执行。只有在命令行直接运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行

# I/O

两个重要概念：同步I/O、异步I/O，前者简单效率低，后者复杂效率高。

**文件读写：**

要以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符；如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示；最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的；由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

try:

f = open('/path/to/file', 'r')

print(f.read())

finally:

if f:

f.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

**with** open('/path/to/file', 'r') **as** f:

print(f.read()

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()最方便

**for** line **in** f.readlines():

print(line.strip()) *# 把末尾的'\n'删掉*

**file-like object**

像open()函数返回的这种有个read()方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个read()方法就行。StringIO就是在内存中创建的file-like Object，常用作临时缓冲。

**二进制文件：**前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可；

**字符编码：**要读取非UTF-8编码的文本文件，需要给open()函数传入encoding参数，例如，读取GBK编码的文件：f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')

遇到有些编码不规范的文件，你可能会遇到UnicodeDecodeError，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，open()函数还接收一个errors参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

**写文件：**写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件

可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险

**with** open('/Users/michael/test.txt', 'w') **as** f:

f.write('Hello, world!')

要写入特定编码的文本文件，请给open()函数传入encoding参数，将字符串自动转换成指定编码。

细心的童鞋会发现，以'w'模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入'a'以追加（append）模式写入。

**StringIO 和ByteIO**

很多时候，数据读写不一定是文件，也可以在内存中读写。StringIO顾名思义就是在内存中读写str; 要把str写入StringIO，我们需要先创建一个StringIO，然后，像文件一样写入即可。getvalue()方法用于获得写入后的str。

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes；请注意，写入的不是str，而是经过UTF-8编码的bytes。

StringIO和BytesIO是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

**操作文件和目录：**主要通过os模块

**序列化：**我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

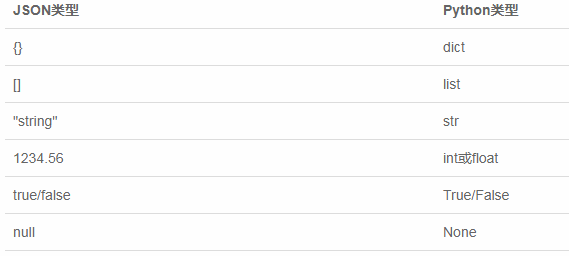
Python提供了pickle模块来实现序列化。pickle.dumps()方法把任意对象序列化成一个bytes，然后，就可以把这个bytes写入文件。或者用另一个方法pickle.dump()直接把对象序列化后写入一个file-like Object；

当我们要把对象从磁盘读到内存时，可以先把内容读到一个bytes，然后用pickle.loads()方法反序列化出对象，也可以直接用pickle.load()方法从一个file-like Object中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象。

**Json:** JSON表示的对象就是标准的JavaScript语言的对象

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON和Python内置的数据类型对应如下



Python内置的json模块提供了非常完善的Python对象到JSON格式的转换:

json模块专门处理这种转换

json.dumps()方法返回一个str，内容就是标准的JSON。类似的，dump()方法可以直接把JSON写入一个file-like Object。要把JSON反序列化为Python对象，用json.loads()或者对应的json.load()方法，前者把JSON的字符串反序列化，后者从file-like Object中读取字符串并反序列化;

Python的dict对象可以直接序列化为JSON的{}，不过，很多时候，我们更喜欢用class表示对象，比如定义Student类，然后序列化;

因为通常class的实例都有一个\_\_dict\_\_属性，它就是一个dict，用来存储实例变量。也有少数例外，比如定义了\_\_slots\_\_的class。

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把Student类实例序列化为JSON，是因为默认情况下，dumps()方法不知道如何将Student实例变为一个JSON的{}对象。可选参数default就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为Student专门写一个转换函数，再把函数传进去即可：

json.dumps(s, **default**=student2dict)) #序列化

同样的道理，如果我们要把JSON反序列化为一个Student对象实例，loads()方法首先转换出一个dict对象，然后，我们传入的object\_hook函数负责把dict转换为Student实例：

json.loads(json\_str, object\_hook=dict2student) #反序列化

**小结**

Python语言特定的序列化模块是pickle，但如果要把序列化搞得更通用、更符合Web标准，就可以使用json模块。

json模块的dumps()和loads()函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。