Python笔记

前言

本书结构，带着问题去学Python

写在正式开始学习的前面——科学学习的出发点。

学习是一门学问，编程也是其中的一个小学问。编程是由一张大网组成，一个一个基本的点组成了这个巨大的网络。每一个知识点都应该由一个一个基本模块单位组成。本书的目的便是对每一个知识点构建一个基本的模块。

比如，对于一个基本知识点，它是什么，它能做什么，怎么做，有怎么样的注意事项，又能有哪些应用拓展。具体到一个函数、表达式或语句，便是这样一些问题：基本概念，作用（表达什么、返回什么（数值、字符串、None、还是什么）），运行机制，使用有什么注意事项，最后有哪些应用。

因此，我将编程学习分为这几大模块，作为学习的基本模式。希望能给您带来一定的学习参考意义。

与君共勉。

2018.8

陈军

wan230114@qq.com

序 言

编程是什么，为什么需要去学习，要理解这种工具，我们不妨来做个比方。

...

带着问题去学习，

python语句由什么构成？变量是什么，赋值是什么，对象是什么？怎么查看对象的属性？存储数据的容器有些什么？常见的数据类型都有些什么操作方法？

目 录

[序 言 2](#_Toc13780414)

[目 录 3](#_Toc13780415)

[1 Python简介 5](#_Toc13780416)

[1.1 Python背景 5](#_Toc13780417)

[1.2 Python的安装及环境搭建 7](#_Toc13780418)

[1.3 Hello world：Python的运行 7](#_Toc13780419)

[1.4 Python程序基本介绍 8](#_Toc13780420)

[1.5 如何查看帮助? 14](#_Toc13780421)

[1.6 基本概念 10](#_Toc13780422)

[2 数据类型及表示 15](#_Toc13780423)

[2.1 数值类型及其表示 16](#_Toc13780424)

[2.2 运算及运算符 18](#_Toc13780425)

[2.3 数值相关函数 21](#_Toc13780426)

[3 数据容器 23](#_Toc13780427)

[3.1 常见三种序列 23](#_Toc13780428)

[3.2 序列操作 28](#_Toc13780429)

[3.3 字典 38](#_Toc13780430)

[3.4 集合set 41](#_Toc13780431)

[3.5 固定集合frozenset 44](#_Toc13780432)

[3.6 容器的赋值与迭代 44](#_Toc13780433)

[3.7 运行机制探索 46](#_Toc13780434)

[4 逻辑运算 53](#_Toc13780435)

[4.1 逻辑语句 53](#_Toc13780436)

[4.2 常见联用函数 58](#_Toc13780437)

[5 函数function 58](#_Toc13780438)

[5.1 基本概念 58](#_Toc13780439)

[5.2 函数的返回值 59](#_Toc13780440)

[5.3 函数参数传递 60](#_Toc13780441)

[5.4 函数名变量 65](#_Toc13780442)

[5.5 函数作用范围 65](#_Toc13780443)

[5.6 高级常用函数 69](#_Toc13780444)

[5.7 函数式编程 71](#_Toc13780445)

[5.8 函数的文档字符串 83](#_Toc13780446)

[5.9 机制研究 84](#_Toc13780447)

[6 数据的迭代与生成 85](#_Toc13780448)

[6.1 迭代器 和 生成器 85](#_Toc13780449)

[6.2 迭代工具函数: 90](#_Toc13780450)

[7 错误及异常处理 92](#_Toc13780451)

[7.1 基本概念 92](#_Toc13780452)

[7.2 语句 94](#_Toc13780453)

[7.3 应用 98](#_Toc13780454)

[8 文件file及编码 99](#_Toc13780455)

[8.1 基本概念 100](#_Toc13780456)

[8.2 内存编码和文件编码 104](#_Toc13780457)

[8.3 文件的操作流程 108](#_Toc13780458)

[8.4 python 文件方法总结 110](#_Toc13780459)

[8.5 with语句 113](#_Toc13780460)

[8.6 原理理解 113](#_Toc13780461)

[9 面向对象 115](#_Toc13780462)

[9.1 类的简介 115](#_Toc13780463)

[9.2 继承/派生 121](#_Toc13780464)

[9.3 封装 enclosure 126](#_Toc13780465)

[9.4 多态 128](#_Toc13780466)

[9.5 对象的属性管理函数 134](#_Toc13780467)

[9.6 类的属性查看方法 136](#_Toc13780468)

[9.7 研究实例 142](#_Toc13780469)

[10 模块和包 143](#_Toc13780470)

[10.1 基本概念 143](#_Toc13780471)

[10.2 模块的导入 148](#_Toc13780472)

[10.3 模块的分类: 151](#_Toc13780473)

[10.4 常用模块 153](#_Toc13780474)

[附件 164](#_Toc13780475)

[10.1 PEP8编码规范 164](#_Toc13780476)

[10.2 常用框架 165](#_Toc13780477)

# Python简介

## Python背景

Python是什么？它从何而来呢，如何诞生？以及后来有什么应用呢？它有何优劣势呢？

### 诞生

诞生时间：1989圣诞节期间

创始人：Guido van Rossum(荷兰)

Python的命名：源于一个喜剧团Monty Python

Python 的官网: [www.python.org](http://www.python.org)

Python的版本:

Python v2.7（2020年结束维护）

Python v3.5（当前学习环境）

Python v3.8（最新版本）

### 应用

应用领域：

数学（最早）、

系统运维、网络编程、科学计算、人工智能，机器人、web开发、大数据及数据库编程、云计算、教育、游戏、图像、其他

### 评价

* Python优缺点：

优点：

面向对象（Java，C++，Python）

免费，简单易学，可移植，可混合编程（C/C++/Java/.net）

开发效率高，开源

缺点：

和C/C++相比执行速度不够快（最高的是C，其次C++）

不能封闭源代码

执行效率vs开发效率，两者几乎不能兼得。

所以涉及效率的部分可以C/C++开发，这种特性使得Python被称为胶水语言。

### 版本与兼容区别

#### python3的优势

数值计算更符合数学思维：

如：“/”在python2中是地板除，python3中则是数学意义上的除法

python2: 1/2=0

python3: 1/2=0.5

速度更快：

实例：亲测迭代300M大小的文件

测试方法：

读取文件，for循环迭代每一行，再用for循环迭代每一个字符，pass填充语句

结果：

python2平均花了20秒，二进制流花了17秒

python3平均花了13秒，二进制流花了11秒

...

#### Python2和Python3的区别：

（此部分可放至最后阅读）

**1 一些程序的格式**

比如print,若要两版本都兼容，最好统一都使用括号，而且只给1个元素

python2中不加括号，可以兼容3中加括号，但仅限于只有一个元素，否则可能出错。

比如中文字符无法正常显示，多元素标准输出会带上两边括号

python3中加括号，

**2 内建函数区别**

有无区别：

比如cmp()函数在2中有，在3中无【确定为无（还需要查证）？】

使用区别：

**3 类的多继承区别**

Python3中使用的C3算法

**4 文件读取时指针问题**

python2中用r模式打开的文件可以从当前位置回退指针

python3中必须用rb模式才行。

若要考虑两者兼容性，都统一使用rb模式打开文件就好，然后统一使用bytes.decode()解码，或在str字符串前面加b转换为bytes保持一致

**5 导入模块包的区别**

Python2 不能导入文件夹，

Python3 能导入文件夹

如from dir import xxx

...

#### 不同运行平台的差别：

Linux系统下运行dict(L)，会自动乱序

Windows系统下有时候顺序保持不变。

## Python的安装及环境搭建

### Python 的安装:

windows / Mac OS X / Unix ...

### Python的解释执行器

CPython（python） ——C写的

Jython ——Java写的

IronPython ——.net

Pypy ——python

## Hello world：Python的运行

### 在Linux中Python执行的三种方式

* 方法一：python命令行交互提示模式

打开命令窗口：

$ python3

>>> print("hello")

>>> quit() #退出交互模式，exit()，或 Ctrl + D

交互模式下，返回的是表达式（对否？）【？】

* 方法二：Linux/other命令运行脚本（指定调用系统程序）

编写文本文件如下，存储为hello.py

print("hello world!")

print("today is friday!")

随后在同级目录下打开命令窗口运行命令：

$ python3 hello.py

* 方法三：脚本文件直接运行（将文本文件直接作为执行文件，第一句写入解释语句）

编写文本文件如下，存储为hello.py

#! /usr/bin/python3

print("hello world!")

print("today is friday!")

随后在同级目录下打开命令窗口运行命令：

$ chmod a+x hello.py

$ ./python.py #要求文件里必须有解释 #! /usr/bin/python3

### 输入、输出与执行

* 基本输入函数：input()

返回：str

返回输入的字符串，自动删除末尾回车。

格式及用法：

a = input("自定义输入提示语(可空)")

使用方法：交互界面输入内容，按回车结束，此时系统自动捕捉输入内容（字符串）

注：若是数字运算，需要将字符串转换为数字类型，见3.2

* 基本输出函数：print()

返回：

None

功能：

打印显示表达式结果于屏幕。如果是内容是一个表达式（如函数），

则打印其“返回结果”。见5

格式：

print(a , b , c , … , sep=" ",end="\n")

a, b, c ... 为表达式（包括数字、字符串等）

sep为输出间隔字符，默认为空格，可自定义

end为末尾默认字符，默认为回车，可自定义

如何理解print？如图



——（摘自Python小课所制作图片）

## Python程序基本介绍

### Python程序的组成

**程序**由**模块**组成

**模块**由**语句**，**函数**，**类**，**数据**等组成

**语句**包含**表达式**

**表达式**建立并处理**数据对象**

#### 语句和表达式

1. **语句statement**

（执行完整句子，如i=1+3，pass，等）

**概念：**

由表达式、运算符、变量等组成的执行某件事的完整句子

**语句换行规则：**

一般一条语句一行；

多条语句在一行中用“;”分隔；

一行语句在多行中用“\”连接，或在“括号”中进行单或多次换行。

\ 可实现显式换行，实现语句连贯

()[]{} 可实现隐式换行，实现语句连贯

1. **表达式**

（运算句子的一部分，运算返回结果，可赋值给变量，例如：1+2）

**概念：**

表达式，是由数字、运算符、数字分组符号（括号）、自由变量和约束变量等以能求得数值的有意义排列方法所得的组合。

**注意：**

表达式不是语句，表达式能赋值给变量，**语句**则不能。

表达式的最终打印内容为运算后结果，print(表达式)

**Python3中有哪些内容被称为表达式？**

简单的区别方法，在交互界面，print(表达式)，按Enter后能正常返回结果于下一行的都是表达式，表达式一般都有返回值，至少为None。【词条内容为自己定义，还有待查漏】

包括：算式，数值对象（数值、字符串、列表等），函数，类对象等。

示例：

>>> print(10+2.5) #算式

12.5

>>> print(12.5) #数值对象（数值）

12.5

>>> print("hello") #数值对象（字符串）

hello

>>> print([1,2,3]) #数值对象（列表）

[1, 2, 3]

>>> print(print(666)) #函数

666

None

>>> print(list) #类对象

<class 'list'> #Python自带的类对象

**# 表达式和语句差别**

**>>> print(1+2)**

3

**>>> print(i=1+2)**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'i' is an invalid keyword argument for print()

**>>> print(pass)**

File "<stdin>", line 1

print(pass)

^

SyntaxError: invalid syntax

#### Python的标识：

# 注释一行，开始到行尾。作用: 让注释内容不参加解释执行

\ 转义符

#### 注意事项

python里面非字符串的所有符号必须是英文的

中文符号可以存在于字符串

## 基本概念

### Python程序本质——对象

Python中一切皆对象（数值，字符串等），函数的参数是对象构成。可多个对象，逗号分隔

>>>函数（对象1，对象2，对象3，…） ## 如print

help(print)可查看函数属性，如sep="\*\*\*"

### 变量与赋值

#### 变量

* 变量基本概念：

变量是用来绑定数据对象的标识符

* 作用：

在内存中用于保存数据，变量是可变的。

**变量可以绑定任何东西**，如数据类型，函数，类实例等对象

* 数据类型：

变量的数据类型实质是绑定的对象的数据类型。

它是动态类型，可以被改变。

查看类型的函数type(a)

* 定义变量： （先定义后使用）

变量名必须是标识符:

[A-Za-z\_][A-Za-z0-9\_]\*

**1) 取名。**

**取名规则:** 1）由字母or下划线or数字构成；2）开头只能是下划线or字母；3）大小写不同；4）保留关键字。（PEP 8规则包含其他书写规则见官网文档，或附件PEP8编码规范）

**2) 赋值创建：**第一次赋值时，创建的变量与对象“绑定”（见下一节）。

合法的变量名示例：

a a1 b bbb \_aaa\_ \_Abcdvara1b2c2

不合法的变量名示例：

1a 123 $ABC +a -b #ab @ab

python 的关键字：

True ，False, None is del if elif等

#### 赋值

* 赋值格式：

变量名 = 值（对象） 或 表达式

（“=”是赋值运算符）

* 其他变量赋值方式：

1）多重赋值

a = a1 = a2 = a3 = 123

print(a, a1, a2, a3)

123 123 123 123

2）多元赋值

>>> a, a1, a2, a3 = 1, 2, 3, 4

print(a, a1, a2, a3)

123 123 123 123

多元赋值本质是元组对应位置绑定：

>>> a,b,c = 1,2,3 # 多元赋值

>>> a,b,c

(1, 2, 3)

>>> t =a,b,c

>>> t

(4, 5, 6)

>>> type(t)

<class 'tuple'>

>>> (a,b,c) = (4,5,6)

>>> a,b,c

(4, 5, 6)

>>> [a,b,c] = (4,5,6)

3）增量赋值

x+=1

x-=1

...

4）变量互换

x, y = y, x

1.变量对象绑定交换

注：该方法不会创建新对象。

>>> a,b=100,200

>>> a,b=b,a

>>> a

200

>>> b

100

2.经典交换算法：（该方法会创建新对象）

>>> a,b = 100,200

>>> temp = a

>>> a = b

>>> b = temp

### 赋值的本质——对象的“绑定”

#### “绑定”概念：

关联/绑定/引用都是变量和一个对象的关联关系。

查看本次运行所有的变量绑定的方法：

help("\_\_main\_\_")

示例：

>>> a = 2

>>> help("\_\_main\_\_") #按Q退出

结果：

Help on module \_\_main\_\_:

NAME

\_\_main\_\_

DATA

\_\_annotations\_\_ = {}

a = 2

FILE

(built-in)

#### 绑定过程：

分别创建“变量对象”和“值对象”，单向绑定。

绑定本质：建立联系，变量对象--> 对象(值)，指向最终值。

问题一：在赋值a=10，b=10的过程中，Python解释器大概做哪几个动作？

1.内存中有“10”对象

2.创建变量对象“a”，“b”

3.“a”和“10”单向绑定（联系），“b”和“10”单向绑定，a、b都绑定到10

问题二：a=10，当a=20第二次赋值时，改变了绑定联系。

原来的10会随着运行而销毁，回收。但10是永久保留于系统中的，大于256的数值对象才会被销毁（见下一小节，小整数对象池）

问题三：变量b通过变量a绑定的实质，如a绑定到10，b绑定到a，b指向哪呢？

a和b都指向了绑定的数值对象10

* 总结：

变量向另一个变量赋值，绑定是指向变量最终对应的值（对象）。

多个变量可绑定到一个对象

#### 判断对象的内存地址

* 基本概念：
* is判断方法：

（判断绑定的是否同一个对象）

is #a is b

is not #a is not b

为什么需要判断？值相同不一定对象相同。

>>> a = 257

>>> b = 257

>>> a is b

False

>>> id(a)

47962736

>>> id(b)

50304160

* id()函数

Python对象有唯一标识。函数id()可查看，id()返回对象的内存地址

小整数对象池：

-5到256的数永远存在，不会释放。

也就是说，大于或小于这个数字会创建新对象。

示例1：单个赋值

>>> a=256

>>> b=256

>>> print(a is b, id(a)==id(b), a==b)

True True True

>>> a=257

>>> b=257

>>> print(a is b, id(a)==id(b), a==b)

False False True

示例2：多重赋值:

>>> a,b=256,256

>>> print(a is b, id(a)==id(b), a==b)

True True True

>>> a,b=257,257

>>> print(a is b, id(a)==id(b), a==b)

False False True

* del函数

解除绑定，引用计数为零时销毁数值对象。

Python内存管理和**引用计数**（为零时，对象删除）

如果数据大于256并且没有其他变量绑定时则销毁

id(数字)查看的当前操作新建对象的地址，立马被销毁

>>> id(257)

139954784965552

>>> id(258)

139954784965552

>>> id(259)

139954784965552

关于更多详细信息参考附录

#### 注意事项：关于对象销毁

脚本相当于一条长命令，命令结束时对象才会销毁。

示例：

ID-test.py

#! /usr/bin/python3

a,b,c,d=255,256,257,258

print(id(255),id(256),id(257),id(258)) #单条命令257对象会再建立

print(id(a), id(b), id(c), id(d))

结果：

tarena@tedu:~/JUN code$ python3 test.py

10927552 10927584 139837766555408 139837766086448

10927552 10927584 139837766555408 139837766086448

命令行模式：

>>> a,b,c,d=255,256,257,258

>>> print(id(255),id(256),id(257),id(258)) #单条命令257对象会再建立

262621408 262621424 47962736 50304160

>>> print(id(a), id(b), id(c), id(d))

262621408 262621424 47962400 47962416

## 如何查看帮助?

* 方法一：help(x)

help(x) #可以查看关于x的信息，x可以是函数名、类名等

help(函数名) #可以查看函数帮助文档，如help(print)，help(pow)

help(类名) #可以查看类的帮助文档，如help(list)

## 对象的属性

### type(obj) 查看对象所属类型

返回对象的类

基本类型都可以用type()判断：

>>> type(123)

<class 'int'>

>>> type('str')

<class 'str'>

>>> type(None)

<type(None) 'NoneType'>

>>> type('abc')==str

True

### id(obj)查看对象内存地址编号

查看对象存储于计算机内存的地址

可用于判断是否是同一个对象

### is判断对象是否相同

* is判断是否是同一个对象

>>> a,b=[],[]

>>> a==b

True

>>> a is b

False

### dir(对象)

### 对象.\_\_dict\_\_

查看对象内的绑定关系

### globals() 返回当前全局作用域内变量的字典

### locals() 返回当前局部作用域内的变量的字典

# 数据类型及表示

茫茫大海，计算机是为处理和计算信息而服务，那么数据又如何分类存储并计算呢？

**Python3 中有六个标准的数据类型：**

* **Number（数字**：整数, 浮点数，复数，布尔型**）**
* **String（字符串）**
* **List（列表）**
* **Tuple（元组）**
* **Set（集合）**
* **Dictionary（字典）**

其中，

不可变数据对象（4 个）：

Number（数字）、String（字符串）、Tuple（元组）、frozenset固定集合

可变数据对象（3 个）：

List（列表）、Dictionary（字典）、Set（集合）。

（可变对象的区别方法为：在对象内有元素发生变化时，该对象地址不变）

**字符串、列表、元组被合并称为序列：**

常见序列类型包括字符串(普通字符串和unicode字符串),列表和元组.所谓序列,即成员有序排列,可通过下标访问。——《[引用至Python常见序列详解-weelin\_area-博客园](https://www.cnblogs.com/diaosir/p/6575891.html)》

**存储基础：**

数据最小单位1Bye(字节)=8位。

## 数值类型及其表示

**python数值类型优势：**

相比C/C++避免了数值范围限制。

### 数字型（整、浮、复）

* 整数（整形数int）

**概念：**所有非小数的数；

**计算机字面值表示方式：**

十进制表示：直接表示。（-5，0，99999）

二进制表示：0b + 0~1。（0b111，0b101）

八进制表示：0o + 0~7。（0o7，0o10，0o17）

十六进制表示：0x + 0~9，A~F。（0x11，0xff，0xFF）大小写无关

**注：**

### Python输出的数值均以十进制表达，打印和输出其他进制见本章末（3.3.2 进制与编码互相转换的函数

字符串编码(值)的转换函数：

ord(c) 返回一个字符的Unicode值

chr(i) 返回编码值对应的字符

整数转换进制字符串的函数：

hex(i) 转换为十六进制的字符串

oct(i) 转换为八进制的字符串

bin(i) 转换为二进制的字符串

### 常用运算函数

* abs()返回给定参数的绝对值

abs(数值)

>>> abs(-9)

9

* pow()指数计算

计算x的y次方

1) 内置的 pow() 方法

pow(x, y[, z]) #pow(x,y,z) 结果等效于pow(x,y) %z

>>> pow(2,3)

8

>>> pow(2,3,3)

2

2) math模块 pow() 方法:

import math

math.pow(x, y)

>>> import math

>>> math.pow(2,3)

8.0

* round()四舍五入

对数值进行四舍五入

round(number[, ndigits])

number 数值

ndigits 小数向右取整的位数，负数表示向左取整

>>> round(3.66,1)

3.7

>>> '%.1f'%3.66 # "%.1f"%num相当于round(num,1)

'3.7'

>>> round(1234,-1)

1230

>>> round(1234,-2)

1200

）

示例：

>>> 0b111

7

>>> 0o17

15

>>> 0xff

255

* 浮点数（浮点型数float）

**字面值：**

**小数形式：**3.14 3.1 3. 0.14 .14

**科学计数法：**小数 + e/E + 正负号 + 指数

>>> 3.14e-2

0.0314

>>> 3.14e8

314000000.0

**注意：**浮点型数无double类型，不同于C/C++

* 复数complex

**字面值：**1j （2j） 1+1J 3-4J (-100+100J)

**特点：**Python数据类型可以表示复数，不同于C/C++

**构成：**实部(real)和虚部(image)

>>> -1+9j

(-1+9j)

>>> 1+9j

(1+9j)

### 布尔值

**字面值：**True False

**注意：**布尔值非零既真；Ture == 1，表示数值相等，可加减

>>> bool("123")

Ture

>>> bool( 2 ) # 非零既真

Ture

>>> True

True

>>> 1

1

>>> True == 1

True

>>> True is 1

False

>>> id(True) #True也在小对象池内，运行结束对象不销毁

262398640

>>> id(1) #1在小对象池内，运行结束对象不销毁(多次运行此句id无变化)

262617344

>>> True == 6

False

>>> True + 1 #可加减

2

### 空值None对象

**概念：**None是一个表示不存在的特殊对象

**作用：**用来占位；用来变量解除绑定(我的理解是原对象消失，而被解除绑定)（？）【（day01笔记）如何解除？】

>>> a=None

>>> help('\_\_main\_\_')

Help on module \_\_main\_\_:

NAME

\_\_main\_\_

DATA

\_\_annotations\_\_ = {}

a = None

FILE

(built-in)

## 运算及运算符

### 算术运算符

所有的运算都遵循数学法则，如5/2=2.5。

算术运算符包括：

+ 加法

- 减法

\* 乘法

/ 除法后都变成浮点数，如4/2=2.0。注意python2中不遵循该规则

// 地板除（除的结果去掉小数部分向下取整）,如5//2=2，3.25//1.5=2.0

% 取余（也叫取模）

\*\* 幂运算

注：

1）运算中有混合类型自动升级。如1+2.2=3.2

2）在表达式的运算过程中，相比C/C++不损失精度，如1/3\*3=1.0

运算符优先级**：**

() #括号分组子表达式，用“()”可以将表达式分组，“()”内的表达式先进行计算

\*\* #幂运算

\* / // %

+ -

示例:

1 + 3 \* 3 \*\* 2 + 4 // 2

1 + 3 \* 9 + 4 // 2

1 + 27 + 4 // 2

1 + 27 + 2

28 + 2

30

### 复合赋值运算符

y += x ==> y = y + x

…

### 比较运算符

<

<=

>

>=

==

!=

**结果返回：**布尔类型的值True False

**运算规则：**多个比较运算符的运算规则，从左到右

例如：

>>> 3>2<3

True

了解：python2中的内建函数cmp，python3中该函数已被废弃(奥卡提姆剃须刀)

cmp(x,y) #比较x和y的大小，x<y返回-1，x=y返回0，x>y返回1

### 位运算符

（此内容初学仅作了解）

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| & | 按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0 | (a & b) 输出结果 12 ，二进制解释： 0000 1100 |
| | | 按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。 | (a | b) 输出结果 61 ，二进制解释： 0011 1101 |
| ^ | 按位异或运算符：当两对应的二进位相异时，结果为1 | (a ^ b) 输出结果 49 ，二进制解释： 0011 0001 |
| ~ | 按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1。~x 类似于 -x-1 | (~a ) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。 |
| << | 左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | a << 2 输出结果 240 ，二进制解释： 1111 0000 |
| >> | 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数 | a >> 2 输出结果 15 ，二进制解释： 0000 1111 |

以下实例演示了Python所有位运算符的操作：

示例：

a = 60 # 60 = 0011 1100

b = 13 # 13 = 0000 1101

c = a & b; # 12 = 0000 1100 &找都为真的。相加取2为1，其余为0

c = a | b; # 61 = 0011 1101 |找都为假的。相加取0为0，其余为1

c = a ^ b; # 49 = 0011 0001 ^找两两不同的，为1。相减，取绝对值

c = ~a; # -61 = 1100 0011 ~每一位取反。相当于-a-1

c = a << 2; # 240 = 1111 0000 <<左移两位，补零

c = a >> 2; # 15 = 0000 1111 >>右移两位，补零

### 运算误差问题汇总

* 1）浮点数运算误差

问题引入：出现误差。

>>> 3.14 - int(3)

0.14000000000000012

证明：

>>> int(3.14)==3 # 构造函数int()会让得到的数值产生误差吗？不会

True

>>> 3.14 - 3

0.14000000000000012

>>> 3.14 - 3.0

0.14000000000000012

>>> 3.14 - 3.00

0.14000000000000012

**误差本质其实是浮点数值的运算**，浮点数运算可能会出现误差

>>> 1/3

0.3333333333333333 #得到的是一个确定的数，非无理数

>>> 1/3 == 0.3333333333333333 #保留16位，两者竟然相等！奇怪的机制

True

>>> 7/3 - 2

0.3333333333333335 #最后一位不遵循四舍五入。

>>> 7/3

2.3333333333333335

>>> 3.25 % 1.5

0.25

>>> 3.26 % 1.5

0.2599999999999998 #结果应该为0.26，但损失了精度

小思索：计算机还存在误差，这些因素究竟为何？我们的计算还无法进行抽象概念的计算？如何解决？计算的误差，创造真正的机器智能，或是探索宇宙本质？

* 2）round()函数的误差

问题引入：round结果应该是3.65≈3.7，但返回3.6

>>> round(3.65,1)

3.6

>>> '%.1f'%3.65

'3.6'

若想进一步了解，help(round)可以查看四舍五入运算的机制

## 数值相关函数

### 数值对象的构造（创建）函数

* int()函数转换为整数

1) 将其他类型数值转换为十进制整数

**int([x])** 将数字统一转换为整数

选项：

[x] 缺省参数默认为零, x可以为整数(包括其他进制)、浮点数、布尔数，但不可为复数

示例（交互模式下）：

int() #0，缺省参数，int()==0

int(3) #3，整数

int(0b11) #3，二进制整数

int(3.14) #3，浮点数

int(bool()) #0，布尔值

int(bool(666)) #1，布尔值

2) 将字符串数值转换为十进制整数

**int("obj"[, ase])** 将进制为ase的整型数obj字符串，统一转换为十进制的数字

选项：

obj 必须为表示“整型数的字符串”，不可为浮点数、复数、布尔数(True/False)

[ase] 表示该字符串数代表的进制数，缺省参数为10

示例：

int("4") #4

int("4", 10) #4

int("0b100") #报错，触发ValueError异常，因为ase默认参数是10

int("0b100",2) #4

int("100",2) #4

int("100") #100，缺省参数为10，int("obj") == int("obj",10)

int("100",10) #100

错误示例：

int("3.14") #不是整型数，报错

int(3.14) #3，正确

int(100, 2) #100不是字符串，报错

int("100", 2) #4，正确

* float()

float([x]) #将数字(除复数外)统一转换为浮点数

x 可以为数值(整数，浮点数，布尔值)，以及表示整数和浮点数的字符串

float(3) #3.0

float("3") #3.0

float(3.14) #3.14

float("3.14") #3.14

float(False) #0.0

float("False") #报错

* complex()

complex(a,b) #a,b代表实部和虚部的值

* bool()

bool(x) #不为零即为真

返回：True 或 False，可直接参与数值运算，前者为1，后者为0。因为python中True==1。

bool(x)返回假值的情况：x为一切空的东西，如下

#无参数

None #空值

False #布尔假值

0.0

0j

'' #空字符串

() #空

[]

{}

set() #空集合

…

### 进制与编码互相转换的函数

字符串编码(值)的转换函数：

ord(c) 返回一个字符的Unicode值

chr(i) 返回编码值对应的字符

整数转换进制字符串的函数：

hex(i) 转换为十六进制的字符串

oct(i) 转换为八进制的字符串

bin(i) 转换为二进制的字符串

### 常用运算函数

* abs()返回给定参数的绝对值

abs(数值)

>>> abs(-9)

9

* pow()指数计算

计算x的y次方

1) 内置的 pow() 方法

pow(x, y[, z]) #pow(x,y,z) 结果等效于pow(x,y) %z

>>> pow(2,3)

8

>>> pow(2,3,3)

2

2) math模块 pow() 方法:

import math

math.pow(x, y)

>>> import math

>>> math.pow(2,3)

8.0

* round()四舍五入

对数值进行四舍五入

round(number[, ndigits])

number 数值

ndigits 小数向右取整的位数，负数表示向左取整

>>> round(3.66,1)

3.7

>>> '%.1f'%3.66 # "%.1f"%num相当于round(num,1)

'3.7'

>>> round(1234,-1)

1230

>>> round(1234,-2)

1200

# 数据容器

## 常见三种序列

### 字符串str

* 1、基本概念

Python字符串是序列的一种，是**不可改变**的字符序列

* 2、表示方法：

**1）引号：**

非注释中，凡是用引号括起来的部分都是字符串。

' 单引号

" 双引号

''' 三单引号

""" 三双引号

注：

空字符串也是字符串。

示例：

''

""

''''''

""""""

bool(空字符串)==0

单引号中可以包含双引号，双引号中可以包含单引号。

>>> '这是内容"hello"，这样表示'

三单(双)引号可以实现隐式换行

示例：

>>> print('''I

love

you''')

I

love

you

隐式字符串字面值可自动拼接：

示例：

>>> '123''12313124' '1231edasf2'

'123123131241231edasf2'

**2）格式化字符串表达式**

格式：

'字符%d字符%f字符%s'%（整数，浮点数，字符串）

占位符：

%d——整数，%f——浮点数，%s——字符串等

**注：**只有一个占位符时后面%()可省略括号，括号的实质代表的是元组

格式化字符串中的占位符和类型码（占位符类型码）：

|  |  |  |
| --- | --- | --- |
| 占位符  类型码 | 占位内容 | 意义 |
| %s | 字符串（数值也可以） |  |
| %r | 字符串 | 使用repr 而不是str |
| %c | 字符的编码值 | 返回编码值转换后的字符 |
| %d | 任意数值 | 返回十进制整数 |
| %o | 整数 | 返回八进制整数 |
| %x | 整数 | 返回十六进制整数(字符a-f小写) |
| %X | 整数 | 十六进制整数(字符A-F大写) |
| %e |  | 指数型浮点数(e小写),如2.9e+10 |
| %E |  | 指数型浮点数(E大写),如2.9E+10 |
| %f, %F |  | 浮点十进制形式 |
| %g, %G | 数值 | 十进制形式浮点或指数浮点自动转换 |
| %% | 无 | 等同于一个%字符 |

示例：

>>> "%.1e"%0.0002112

'2.1e-04'

...

**Python format 格式化函数：**

|  |  |  |  |
| --- | --- | --- | --- |
| 数字 | 格式 | 输出 | 描述 |
| 3.1415926 | {:.2f} | 3.14 | 保留小数点后两位 |
| 3.1415926 | {:+.2f} | +3.14 | 带符号保留小数点后两位 |
| -1 | {:+.2f} | -1.00 | 带符号保留小数点后两位 |
| 2.71828 | {:.0f} | 3 | 不带小数 |
| 5 | {:0>2d} | 05 | 数字补零 (填充左边, 宽度为2) |
| 5 | {:x<4d} | 5xxx | 数字补x (填充右边, 宽度为4) |
| 10 | {:x<4d} | 10xx | 数字补x (填充右边, 宽度为4) |
| 1000000 | {:,} | 1,000,000 | 以逗号分隔的数字格式 |
| 0.25 | {:.2%} | 25.00% | 百分比格式 |
| 1000000000 | {:.2e} | 1.00e+09 | 指数记法 |
| 13 | {:10d} | 13 | 右对齐 (默认, 宽度为10) |
| 13 | {:<10d} | 13 | 左对齐 (宽度为10) |
| 13 | {:^10d} | 13 | 中间对齐 (宽度为10) |

* 3、格式化字符串的灵活使用

**示例1：**手动制造组合字符串表达式：

fmt="%d岁的%s喜欢诗意和远方"

name='xiaosan'

age=20

print(fmt%(age,name))

**示例2：**占位符与类型码之间的格式语法：

%x.yf。 x代表整数部分占位数，y代表小数部分占位数

>>> '%-+10d'%10 # - 使字符居左，+显示正号，默认不显示

'+10 '

>>> '%f'%777.67 # 默认浮点数无规律补零

'777.670000'

>>> '%.3f'%777.67 # 小数点后保留3位，不够的添零补齐

'777.670'

>>> '%10.3f'%777.67 # 整数部位占位10位，小数点后添零补三位

' 777.670'

>>> '%010.3f'%777.67 # 整数部位补零占位10位，小数点后添零补三位

'000777.670'

>>> '%2.3f'%777.67 # 当位数不足时，填本身数值位数（猜测:%f==%1f）

'777.670'

**示例3：**数字前面填充0

>>> '%09d'%88

'000000088'

>>> '%9d'%88

' 88'

**示例4：**自定义格式之间的数值的方法：

实例：

自动化判断输入的值对齐到右边

s1=input('Input1：')

s2=input('Input2：')

s3=input('Input3：')

number=max(len(s1),len(s2),len(s3))

s='%'+str(number)+'s'

#其他表达形式：

#s = '%%%ds'%number ##%%d表示%d

print(s%s1)

print(s%s2)

print(s%s3)

其他方法：计算空格并填充，'空格'\*(总长度-字符长度)

**示例5：**简化元组传参，'%d与%s'%

用对象元组迭代导入%内容

>>> t=(1,'a')

>>> '%d，%s'%t

'1，a'

用tuple函数转换为元组

>>> L=[1,'a']

>>> '%d，%s' % tuple(L)

'1，a'

**示例6：**高级字典传参，'%(键)s'%{'键':值}

>>> '...%(pa)s...%(pb)s....'%{'pa':11,'pb':'haha'}

'...11...haha....'

* 4、转义字符

用转义序列代表特殊字符：

|  |  |
| --- | --- |
| 转义格式 | 意义 |
| \' | 单引号(') |
| \" | 双引号(") |
| \\ | 反斜杠(\) |
| \n | 换行 |
| \r | 返回光标至行首 |
| \f | 换页符 |
| \t | 水平制表符 |
| \v | 重直制表符 |
| \b | 倒退 |
| \0 | 空字符，字符值为零 |
| \000 | 000为三位八进制ascii表示的字符 |
| \xXX | XX为两位十六进制ascii表示的字符 |
| \uXXXX | Unicode 16的十六进制表示的字符 |
| \UXXXXXXXX | Unicode 32的十六进制表示的字符 |

使用示例：

>>> "ABC\bD"

'ABC\x08D'

>>> print("ABC\bD") # 倒退一格

ABD

>>> '\n \012 \x0A \u000A \U0000000A' # 编码

'\n \n \n \n \n'

ascii编码表：见百度百科词条ascii

* raw字符串（原始字符串）

问题陈述：

>>> s='\\'

>>> s

'\\'

>>> print(s)

\

表达式中，双反斜杠实际表示的是一个反斜杠

让原有转义字符无效：

>>> s="c:\\now\_folder\file" # \x0c的ascii对应的是换页符

'c:\\now\_folder\x0cile'

>>> print(s) #打印出来，转义字符会转换为实际值

c:\now\_folder

ile

>>> s=r"c:\\now\_folder\file" # 返回字符串及转义字符，

>>> s

'c:\\\\now\_folder\\file'

>>> print(s)

c:\\now\_folder\file

### 列表list

* 基本概念

**作用：**临时存入数据，同时方便添加，删除，修改等操作。

**特征**：

容器；一种序列；可以改变；由一系列有顺序的元素组成；元素之间可没有任何联系；

可以进行比较；是<可迭代对象>；

**字面值：**L = [] 等

列表的本质：列表中存储的是一系列变量，如X[1],X[2],X[3]。

### 元组tuple

* 基本概念
* 元组tuple定义：

元组是不可改变的序列，同list一样，元组是可以存放任意类型的容器

* 元组的表示方法：

用()括起来，单个元素括起来后加逗号。区分单个对象还是元组。

## 序列操作

包含字符串str，列表list，元组tuple，字节串bytes，字节数组bytearray等

序列满足三个共同点：

1.下标；

2.in和not in

3.切片

如字典集合不能切片。

### 运算

#### 算术运算

+ 拼接字符，如运算x+=y

\* 仅能和整数相乘，生成重复的序列，如'123'\*3，[1,2]\*2

+=

\*=

* 特殊情况：

列表中的+=格式必须为：

列表 + <可迭代对象>

>>> L=[1,2]; L+='123'; L

[1, 2, '1', '2', '3']

>>> L=[1,2]; L+=range(3); L

[1, 2, 0, 1, 2]

>>> [1,2] + '123'出错

若<可迭代对象>为不可变对象，将重新生成一个新对象，注意与拷贝的关系问题。

对于重复操作能否想到这个运算符

列表不可和元组相加

#### 比较运算

* 比较运算符。

格式：对象1 运算符 对象2

运算符： <、<=、>、>=、==、!=

返回：布尔值

**规则：**非数学比较方式。

从左往右依次比较，直到出现不同结束比较，比较的实际是ascii数值

>>> 'A'=="A"

True

>>> 'ab'<'adca'

True

>>> list('ab') < list('adca')

True

注：数值类型不同比较时会出错。但可以用!=返回true

>>> 'A'<1

TypeError...

>>> 'A'!=1

True

* in/not in比较判断

**格式：**对象 in 序列

**返回：**布尔值

**说明：（**对象、序列**）**包括**（**字符串、列表、字典等**）**

#### 索引和切片取值

**索引**(Index)**语法：**

序列[整数表达式]；

序列可以是字符串、列表、元组；

说明：

整数表达式处结果必须为整形数；

字符串每个字符对应一个下标，

正向序列从左到右从0开始，反向从-1开始；

逆序最后一个数（即为正序第一个数）等于-len(序列)。

序列获取索引值

T.index(v[, begin[, end]])

返回对应元素的索引下标，begin为开始索引，end为结束索引，v不存在时触发ValueErorr错误（序列方法）

**切片**(slice)**语法：**

序列[开始索引:结束索引:步长]

序列可为字符串、列表、字典

**说明：**都可以省略。步长从开始索引的位置从0算起。包含第一位，不包含最后一位。

**返回：**输入原对象类型。如：字符串-->字符串，列表-->列表

>>> s

'abcdef'

>>> s[1:5:2]

'bd'

>>> s[1:5]

'bcde'

>>> L

[1, 2, 3, 4, 5, 6]

>>> L[1:5]

[2, 3, 4, 5]

>>> L[1:5:2]

[2, 4]

**切片规则：**

#分析

# 0 1 2 3 4 5 6 7 8 9

# -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

#

s='0123456789'

#步长

>>> s[1:8] #切片取第一位，停止于最后一位前。[a,b)

'1234567'

>>> s[1:8:1] #默认步数为1可省略，以下展示步数1,2,3

'1234567'

>>> s[1:8:2] #步长为2，从开始索引处从0计算2步

'1357'

#逆向切片

>>> s[1:8:-1]

''

>>> s[8:1]

''

>>> s[8:1:-1] #反向切不报错，使用负步数可以打印结果

'8765432'

>>> s[-1:-8]

''

>>> s[-1:-8:-1]

'9876543'

>>> s[-1:-8:-2]

'9753'

>>> s[-8:-1]

'2345678'

#关于省略

>>> s[:5]

'12345'

>>> s[5:]

'67'

>>> s[::]

'0123456789'

>>> s[::2]

'02468'

>>> s[::-1] #此法可以看回文(看笔记，寻找老师的方法[?]

'9876543210'

#其他问题

>>> s[-100:100] #开始和结束都能越界

'0123456789'

#### 索引和切片赋值

特殊情况：

索引和切片的赋值必须是对<可变对象>。【可变对象与不可变对象区别见章尾】

* 列表切片的赋值

**语法：**列表[切片slice] = <可迭代对象>

**规则：**步长＝±1，替换连续切片值为新值；步长≠1，提供的元素个数必须对等。

注：赋值运算符=的右边必须为<可迭代对象>

实例：L部分元素倒序，并赋值给a

>>> L=[0,1,2,3,4,5,6,7,8,9]

>>> a = L[1:5] = L[1:5][::-1]

>>> L

[0, 4, 3, 2, 1, 5, 6, 7, 8, 9]

>>> a

[4, 3, 2, 1]

**技巧：**

插入元素 L[0 : 0] = [5] #第一位后

L[-1:-1] = [5] #最后一位前

反转元素 L[:] = L[::-1] #不会改变id()，

L = L[::-1] #会改变id()，

【第一个原理是对内部元素切片重新赋值，不改变外部列表对象ID】

【第二个原理是绑定于新对象，原列表对象无绑定被销毁】

#### del语句

del 可用来删除列表元素

**语法：**

del 列表[索引]

del 列表[切片]

**问题引入：**列表赋值空列表“[]”并不能删除元素

>>> L=[6, 5, 4, 3, 2, 1]

>>> L[5]=[]

>>> L

[6, 5, 4, 3, 2, []]

实例

>>> L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> del L[-1]

>>> L

[0, 1, 2, 3, 4, 5, 6, 7, 8]

>>> del L[1:8]

>>> L

[0, 8]

### 序列相关函数

#### 序列构建函数

##### 字符串构建函数

* 直接定义

S = ""

s = "abc"

* 构造函数：

str() #生成空字符串，等同于""

str(iterable) #对象iterable转换为字符串，可以为数字或其他序列

示例：

>>> str(3.14)

'3.14'

>>> str(None)

'None'

>>> str(2+3j)

'(2+3j)'

##### 列表构建

* 直接定义

L = []

L = [1,2,3]

* 构造函数：

list() 生成一个空的列表，等同于[]

list(iterable) 用<可迭代对象>创建一个列表

示例：

>>> list(range(1,5))

[1, 2, 3, 4]

>>> list('1234')

['1', '2', '3', '4']

##### 元组构建

* 直接定义

t = x,

t = (x,)

t = x,y

t = (x,y)

示例：

t = 20, # (20,)

t = (20,) # (20,)

t = 20,30 # (20,30)

t = (20,30) # (20,30)

注意：

利用逗号，逗号是关键。若无逗号则为序列赋值：

如：

t = (200) # 等同于 t=200

* 构造函数：

tuple() 生成一个空的元组，等同于()

tuple(iterable) 可用迭代对象生成一个元组

实例

>>> tuple([x\*\*2 for x in range(1,10)])

(1, 4, 9, 16, 25, 36, 49, 64, 81)

#### 常用序列函数

##### 比较判断的函数

max(x) 返回最大值

min(x) 返回最小值

any(iterable) 真值测试，判断所有元素，一个为真，返回True

all(iterable) 真值测试，判断所有元素，都为真，返回True

##### 序列计算的函数

len(x) 返回序列的长度。x可以是字符串，比较的是编码值

sum(iterable[,x]) #返回<可迭代对象>与初始值x相加的和

iterable 可以是列表、元组、字典、集合

##### 排序操作

对可迭代元素排序，生成一个排序后的列表

sorted(iterable[, reverse=False])

选项:

iterable 可以是序列

reverse 默认缺省参数reverse=True，代表正序，False代表逆序

返回：一个新的排序后的列表

示例：

L2 = sorted(L)

### 序列方法

常用序列方法

#### 字符串

* 常用字符串方法

方法和函数都是表达式，与函数不同的是必须借助对象

**格式：**对象.方法名（方法传参）

**包括：**

|  |  |  |
| --- | --- | --- |
| 方法 | 返回 | 说明 |
| S.isdigit() | 布尔值 | 判断字符串中的字符是否全为数字 |
| S.isalpha() | 布尔值 | 判断字符串是否全为英文字母 |
| S.islower() | 布尔值 | 判断字符串所有字符是否全为小写英文字母 |
| S.isupper() | 布尔值 | 判断字符串所有字符是否全为大写英文字母 |
| S.isspace() | 布尔值 | 判断字符串是否全为空白字符 |
| S.isnumeric() | 布尔值 | 判断字符串是否全为数字字符。如1，一 |
| S.center(width[,fill]) | 字符串 | 将原字符串居中，fill左右默认填充空格，width总宽度, |
| S.count(sub[, start[,end]]) | 整型数 | 获取一个字符串中子串的个数 |
| S.find(sub[, start[,end]]) | 整型数 | 获取字符串中子串sub的索引,失败返回-1 |
| S.strip([fill]) | 字符串 | 返回去掉左右空白字符的字符串,fill默认空格 |
| S.lstrip() | 字符串 | 返回去掉左侧空白字符的字符串 |
| S.rstrip() | 字符串 | 返回去掉右侧空白字符的字符串 |
| S.upper() | 字符串 | 生成将英文转换为大写的字符串 |
| S.lower() | 字符串 | 生成将英文转换为小写的字符串 |
| S.replace(old, new[, count]) | 字符串 | 将原字符串的old用new代替，生成一个新的字符串，count是替换结束的索引数 |
| S.startswith(prefix[, start[, end]]) | 布尔值 | 返回S是否是以prefix开头，如果以prefix开头返回True,否则返回False, |
| S.endswith(suffix[, start[, end]]) | 布尔值 | 返回S是否是以suffix结尾，如果以suffix结尾返回True,否则返回False |
| 以下是不常用的 |  |  |
| S.title() | 字符串 | 生成每个英文单词的首字母大写字符串 |

* 字符串的文本解析方法split和join

|  |  |  |
| --- | --- | --- |
| 方法 | 返回 | 说明 |
| str.split([sep=None[, num]]) | 列表 | 返回字符分割的列表。sep=' '，表示以空格分割，默认不填时为一切空的字符，包括空格、制表符、换行符。num为表示分割列表最多到第多少个。 |
| str.join(iterable) | 字符串 | 返回对象连接的字符串。对<可迭代对象>用字符str连接。 |
| str.splitlines() | 列表 | 返回字符分割的列表，以\r\n或\n为分隔符，  【？待实验确认】 |

str.split注意事项：

1、处理文本文件时，注意末尾的换行符。

错误示例：处理文本文件时，当部分行的列数比预计多时，直接切割并用join连接可能会因为丢失末尾换行符而导致新文件中的相邻两行未换行。

解决方案：

1）split指定切割次数

2）读取一行就去掉该行末尾的换行符，再写入新文件时手动添加。

示例：

>>> 'abc'.isalpha() #判断是否全为英文字母

True

字符串.startswith()，判断字符串的首字符是否符合要求

#!/usr/bin/python

str = "this is string example....wow!!!";

print str.startswith( 'this' );

print str.startswith( 'is', 2, 4 );

print str.startswith( 'this', 2, 4 );

以上实例输出结果如下：

True

True

False

大小写转换

str = "www.runoob.com"

print(str.upper()) # 把所有字符中的小写字母转换成大写字母

print(str.lower()) # 把所有字符中的大写字母转换成小写字母

print(str.capitalize()) # 把第一个字母转化为大写字母，其余小写

print(str.title()) # 把每个单词的第一个字母转化为大写，其余小写

print (str.swapcase()) # 大小写转换

#### 列表

* 用列表调用方法

|  |  |  |
| --- | --- | --- |
| 方法 | 返回 | 说明 |
| L.index(v [, begin[, end]]) | 数值 | 返回对应元素的索引下标。begin为开始索引，end为结束索引,当 value 不存在时触发ValueError错误 |
| L.insert(index, obj) | None | 将某个元素插放到列表中指定的位置（索引后面一位） |
| L.count(x) | 数值 | 返回列表中元素x的个数 |
| L.remove(x) | None | 从列表中删除第一次出现在列表中的值x |
| L.copy() | None | 复制此列表（只复制一层，不会复制深层对象) |
| L.append(x) | None | 向列表中追加单个元素 |
| L.extend(lst) | None | 向列表追加另一个列表lst，或其他<可迭代对象> |
| L.clear() | None | 清空列表,等同于 L[:] = [] |
| L.sort(reverse=False) | None | 将列表中的元素进行排序，默认顺序按值的小到大的顺序排列 |
| L.reverse() | None | 列表的反转，用来改变原列表的先后顺序 |
| L.pop( [index] ) | 数值 | 删除索引对应的元素，默认删除最后元素(不加索引)，同时返回删除元素的引用关系（对应删除的元素【？】 |

* 注意：

**关于返回和语句执行问题：**

>>> L = [0, 1, 2, 3, 4, 5]

>>> x=L.extend([444])

>>> print(x)

None

>>> print(L)

[0, 1, 2, 3, 4, 5, 444]

方法不一定都返回东西；方法的调用不一定要写成语句，语句中也可以执行保存

* 实例

列表LIST添加元素的几种方法与对比

1. 列表.append()方法

将“待插入对象”作为一个元素添加到原数据

>>> data = [1,2]

>>> data.append(['123','abc'])

>>> data

[1, 2, ['123', 'abc']]

1. 列表.extend()方法

将“待插入对象”作为各个元素连接到原数据

>>> data = [1,2]

>>> data.extend(['123','abc'])

>>> data

[1, 2, '123', 'abc']

>>> L = [1, 2, 3]

>>> L.extend("123") # 理解：隐藏的将对象"123"进行了一次list("123")

>>> L

[1, 2, 3, '1', '2', '3']

>>> L = [1, 2, 3]

>>> L.extend((1,2,"123"))

>>> L

[1, 2, 3, 1, 2, '123']

1. 列表.insert()方法

将“待插入对象”作为一个元素添加到原数据特定位置

>>> data = [1,2]

>>> data.insert(1,'abc')

>>> data

[1, 'abc', 2]

1. 加号

用加号，将两个list相加，会返回到一个新的list对象

注意与前三种的区别。前面三种方法（append, extend, insert）可对列表增加元素的操作，他们没有返回值，是直接修改了原数据对象。 注意：将两个list相加，需要创建新的list对象，从而需要消耗额外的内存，特别是当list较大时，尽量不要使用“+”来添加list，而应该尽可能使用List的append()方法。

>>> list1

['a', 'x', 'b']

>>> list2=['y','z']

>>> list3=list1+list2

>>> list3

['a', 'x', 'b', 'y', 'z']

#### 元组方法

只有两种

T.index(v[, begin[, end]])

T.count(x)

### 应用

#### 遍历序列

序列皆<可迭代对象>，for循环可以遍历

### 高级示范

* 寻找某个字符串之后的索引

>>> s = '0123456789'

>>> s.find(s.split('456')[1])

7

>>> s[7]

'7'

## 字典

### 基本概念

* 问题引入：

#list查找机制

L=[1,2,3,4,5,6,7]

6 in L

#字典可以加快查找？

* 作用：

可以简化循环判断操作

* 特征：

可变，可变容器，可变对象；

键-值映射储存；

键不能有重复，否则只保留后面出现的键值；

无先后顺序，存储也是无序的，不会自动排序，

### 运算

* 算术运算

键的值可以运算

>>> d={}

>>> d[1]=1

>>> d[1]+=1

>>> d[1]

2

### 字典相关函数

len(x)

max(x)

min(x)

sum(x)

any(x)

all(x)

只对键比较和运算，比较和运算也需要类型一致。

#### 构建与修改

* 创建
* 创建格式：

d = {} #空字典

d = {键(key):值(value), 键(key):值(value), ...}

**不可变对象可以作为字典的键如str,tuple等，**

**可变的对象不可以作为键，但可以作为值，如list**

* 字典构造函数：

dict() # 创建空字典，等同于 {}

dict(iterable) 用可迭代对象初始化一个字典

dict(\*\*kwargs) 关键字传参形式生成一个字典

实例：

创建实例：

>>> d={1:123,'a':123} #按照普通方式

>>> d

{1: 123, 'a': 123}

>>> dict([(1,123),('a',234)])

{1: 123, 'a': 234}

>>> dict(name='abc',age=14) #

{'name': 'abc', 'age': 14}

>>> a='werwer'

>>> dict([(1,123),(a,234)])

{1: 123, 'werwer': 234}

批量创建相同值的字典

d =dict.fromkeys((1,2,3),[])

print(d)

* 添加/修改字典

**键赋值修改操作：**d[键]=值，可用于创建与修改。

**删除操作：**del d[键]，删除键及对应值

* 查找字典

**键索引：**d[键]，返回键的值，

**迭代访问：**for i in d，得到的是键，可通过d[键]得到键值

* 判断字典

**in / not in：**判断键是否在字典内

in 判断字典实际是判断键的集合对象

示例：

>>> d={'a':1,'b':2,'c':3}

>>> 1 in d

False

>>> 'a' in d

True

>>> 1 in d.values()

True

>>> 'a' in d.keys()

True

### 方法

常用方法见文件…

|  |  |  |
| --- | --- | --- |
| 函数 | 返回 | 说明 |
| D代表字典对象 |  |  |
| D.clear() |  | 清空字典 |
| D.pop(key) |  | 移除键，同时返回此键所对应的值 |
| D.copy() |  | 返回字典D的副本,只复制一层(浅拷贝) |
| D.update(D2) |  | 将字典 D2 合并到D中，如果键相同，则此键的值取D2的值作为新值 |
| D.get(key, default) |  | 返回键key所对应的值（对象？）,如果没有此键，则返回default，default可以自己指定，不指定若没有则返回None |
| D.keys() |  | 返回可迭代的 dict\_keys 集合对象 |
| D.values() |  | 返回可迭代的 dict\_values 值对象 |
| D.items() |  | 返回可迭代的 dict\_items 对象 |

字典的排序：见6.6.1 lambda表达式（lambda匿名函数）

示例：D.items()返回可迭代的 dict\_items 对象

d = {1: 'a', 2: 'b', 'A': 'a', 'B': 'b'}

print(d.items())

for i in d.items():

print(i)

'''

dict\_items([(1, 'a'), (2, 'b'), ('A', 'a'), ('B', 'b')])

(1, 'a')

(2, 'b')

('A', 'a')

('B', 'b')

'''

### 字典的应用

字典是可迭代对象，字典只能对键进行迭代访问

示例:

d = {1: "One", 2: "Two", "三": 'Three'}

for k in d:

print(k, '对应的值是:', d[k])

* 用于数据查找

字典查找替换等操作

* 存储结构化数据

citys={

'北京':{

'朝阳':['国贸','CBD','天阶','我爱我家','链接地产'],

'海淀':['圆明园','苏州街','中关村','北京大学'],

'昌平':['沙河','南口','小汤山',],

'怀柔':['桃花','梅花','大山'],

'密云':['密云A','密云B','密云C']

},

'河北':{

'石家庄':['石家庄A','石家庄B','石家庄C','石家庄D','石家庄E'],

'张家口':['张家口A','张家口B','张家口C'],

'承德':['承德A','承德B','承德C','承德D']

}

## 集合set

### 基本概念：

特征：

可变容器；

无重复(数据对象具有唯一性)；

无序储存结构，无先后顺序关系（插入，删除等操作很快）；

元素必须是不可变对象（如列表字典不能作为元素）；

相当于只有键的字典。

### 构建与修改

* 创建

创建集合：

>>> s = {1, 2 ,3 ,4}

>>> s

{1, 2, 3, 4}

不能用{}创建空集合

构建函数：

set() 创建一个空的集合对象（不能用{}来创建空集合，因为{}默认创建的是字典）

set(iterable) 用<可迭代对象>创建一个集合

若是字典只取键，若是字符串则分割

示例：

>>> s=set("ABC")

>>> s

{'B', 'A', 'C'} #为什么是BAC的顺序？【？】

>>> max(s)

'C'

>>> min(s)

'A'

* 修改方法：

无索引，所以只能删了再加

### 集合的运算：

& | - ^

>>> s = {1, 2, 3}

>>> s1 = {1, 2, 3}

>>> s2 = {2, 3, 4}

>>> print(s1 & s2) # & 生成两个集合的交集

{2, 3}

>>> print(s1 | s2) # | 生成两个集合的并集

{1, 2, 3, 4}

>>> print(s1 - s2) # - 生成两个集合的补集

{1}

>>> print(s2 - s1)

{4}

>>> print(s2 ^ s1) # ^ 生成两个集合的对称补集

{1, 4}

> <

s1 = {4, 5, 6}

s1 = {4, 5}

>>> print(s1 > s2) # > 判断一个集合是否是另一个集合的超集

Ture

>>> print(s1 < s2) # < 判断一个集合是否是另一个集合的子集

False

== !=判断集合是否相同

>>> {1,2,3} == {2,1,3}

True

<= >= 判断包含与被包含关系

in / not in 判断某元素是否在集合中

### 集合相关函数

集合的内建函数操作：

len(x), max(x), min(x), sum(x), any(x), all(x)

### 集合常用方法

|  |  |  |
| --- | --- | --- |
| 方法 | 返回 | 意义 |
| S.add(e) | None | 在集合中添加一个新的元素e；如果元素已经存在，则不添加 |
| S.remove(e) | None | 从集合中删除一个元素，如果元素不存在于集合中，则会产生一个KeyError错误 |
| S.discard(e) | None | 从集合S中移除一个元素e,在元素e不存在时什么都不做; |
| S.clear() | None | 清空集合内的所有元素 |
| S.copy() | None | 将集合进行一次浅拷贝 |
| S.pop() | 数值 | 从集合S中删除一个随机元素;如果此集合为空,则引发KeyError异常。经探索是从第一个数开始删除，并返回该值 |
| S.update(s2) | 更新后的原集合 | 用 S与s2得到的全集更新变量S（相当于并和运算s1 |= s2）  注意：s1=s1|s2是赋予一个新对象，不会被浅层复制 |
| S.difference(s2) | 集合 | 用S - s2 运算，返回存在于在S中，但不在s2中的所有元素的集合。即S中的特有元素 |
| S.difference\_update(s2) | 集合 | 等同于 S = S - s2 |
| S.intersection(s2) | 集合 | 等同于 S & s2 |
| S.intersection\_update(s2) | 更新后的原集合 | 等同于S = S & s2 |
| S.symmetric\_difference(s2) | 集合 | 返回对称补集,等同于 S ^ s2 |
| S.symmetric\_difference\_update(s2) | 更新后的原集合 | 用 S 与 s2 的对称补集更新 S |
| S.union(s2) | 集合 | 生成 S 与 s2的全集 |
| S.isdisjoint(s2) | 布尔值 | 如果S与s2交集为空返回True,非空则返回False |
| S.issubset(s2) | 布尔值 | 如果S与s2交集为非空返回True,空则返回False |
| S.issuperset(...) | 布尔值 | 如果S为s2的子集返回True,否则返回False |

实例：

enter = input('enter 0 or 1：')

s1 = {1, 2, 3}

s2 = {2, 3, 4}

s = s1

print(s is s1)

if enter == '0':

s1 = s1 | s2 # 生成新对象{1,2,3,4}

print(s is s1)

print(s1) # {1,2,3,4}

print(s) # {1,2,3}

elif enter == '1':

s1 |= s2 # 将对象更新变为{1,2,3,4}，相当于方法S.update(s2)

print(s is s1)

print(s1) # {1,2,3,4}

print(s) # {1,2,3,4}

### 集合的应用

* 迭代对象

集合是可迭代对象

用for语句可以得到集合中的全部数据元素

* 1、去重复

a = [1, 1, 2, 2, 3]

a = list(set(a))

* 2、判断无序相等

a = ["a","b","c"]

a = ["a","c","b"]

a == b #Fulse

set(a) == set(b) #True

## 固定集合frozenset

固定集合可以作为字典的键，也可以作为集合的值

作用：

固定集合可作为字典的键，也可以作为集合的值【？】

创建空的固定集合：

fz = frozenset()

创建非空的固定集合：

>>> fz = frozenset()

>>> fz

frozenset()

固定集合运算，同set运算一致。参见set运算

相当于集合的方法去掉修改集合的方法。

固定集合可以和集合进行比较，如

set(1,2,3)=frozenset…

True

## 容器的赋值与迭代

关于容器的赋值：

t = (20) # t 绑定定整数，不是元组

x, y, z = 100, 200, 300 # 序列赋值等同于如下

x, y, z = (100, 200, 300) # 序列赋值

x, y, z = [100, 200, 300] # 序列赋值

x, y, z = "ABC" # 序列赋值

#### 列表推导式

是由可迭代对象依次生成带有多个元素的列表的表达式

作用：用简易方法生成列表

语法：

[表达式 for 变量 in 可迭代对象 [if 真值表达式]]

实例：

L = []

for x in range(1, 10):

L.append(x \*\* 2)

print(L)

L2 = [x \*\* 2 for x in range(1, 10)]

print(L2)

L3 = [x for x in range(1, 30) if x%2==1]

print(L3)

'''

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

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

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

'''

列表推导式的嵌套：

语法：

[表达式

for 变量1 in 可迭代对象1 [if 真值表达式1]

for 变量2 in 可迭代对象2 [if 真值表达式2]

…

示例：

>>> L=[x+y for x in 'ABC' for y in '123']

>>> L

['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3']

应用：

转换字符串类型列表为数字类型列表

>>> l= ['100', '200', '300', '400', '500']

>>> L= [int(x) for x in L]

>>> L

[100, 200, 300, 400, 500]

语法机制：

>>> t =(x\*\*2 for x in range(1,10))

>>> t

<generator object <genexpr> at 0x0000000002E97888>

>>> tuple(t)

(1, 4, 9, 16, 25, 36, 49, 64, 81)

>>> list(t)

[]

>>> [x\*\*2 for x in range(1,10)]

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

应用，

批量转换字符类型

比如str-->int

>>> S='123456789'

>>> L=[int(x) for x in S]

>>> L

[1, 2, 3, 4, 5, 6, 7, 8, 9]

#### 字典推导式

字典推导式是用可迭代的对象依次生成字典的表达式

语法:

{ 键表达式: 值表达式 for 变量 in 可迭代对象 [if 真值表达式]}

注: [] 及其中的内容代表可省略

说明:

1. 先从可迭代对象中取值

2. 用变量绑定

3. 用if 进行条件判断，如果为真值则执行下一步

4. 执行'键表达式'和 '值表达式' 然后加入到新字典中

示例:

生成一个字典，键为1~9的整数,值为键的平方

>>> {x: x\*\*2 for x in range(1, 10)}

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

#### 集合推导式

集合推导式是用可迭代对象生成集合的表达式

语法：

{表达式 for 变量 in 可迭代对象 [if 真值表达式]}

注：[]括起的部分代表可省略

推导式内的for子句可以嵌套

## 运行机制探索

### 查找速度对比

实例：

import time

from collections import OrderedDict

def f(L, xx):

t0 = time.time()

for i in L:

if i in xx:

continue

print('-->', type(xx), '遍历结束，耗时%f秒' % (time.time() - t0))

def main():

# 对比某个元素在"列表/元组/字典/集合"的查找速度

L = list(range(0, 30000))

f(L, list(L))

f(L, tuple(L))

f(L, set(L))

dic1 = dict(zip(L, [None] \* len(L)))

dic2 = OrderedDict(zip(L, [None] \* len(L)))

f(L, dic1)

f(L, dic2)

if \_\_name\_\_ == '\_\_main\_\_':

main()

结论：对于x in X的操作，集合字典在速度上有明显优势

* 字典与列表的比较

比较

1. 都是可变对象

2. 索引方式不同,列表用整数索引，字典用键索引

3. 字典的查找速度可能会快于列表(重要) （字典机制：不会一一比对）

4. 列表的存储是有序的，字典的存储是无序的

列表和字典的内部存储原理

列表是顺序存储

尾插很方便，但头插会将后面的数据都后挪一位，很麻烦。查找也是顺序逐个比对。

字典是映射存储

添加，删除，查找速度都很快

集合和字典的优点

in / not in 运算符的查找速度快

插入、删除快，特别是对应列表中间，无中间概念。

### 列表L的定义

需要弄清楚一个问题，对象与变量之间的关系，而每一个索引对象的值又是一个小对象

L=[1,2,3]

变量L，绑定一个序列对象，序列对象变化对【？待完善】

### 可变对象与不可变对象区别

* 总结：

可变对象，对原变量指向的对象改变，地址不变，说明对象不变；

不可变对象，对原变量指向的对象改变，地址改变，说明对象产生变化；

（该问题可能会对数据处理（如拷贝等）造成困扰，谨慎处理）

* 问题引入：

Lline = [1, 2, 3]

L = []

L.append(Lline)

Lline.extend([4, 5, 6])

print(L) #请问输出多少？

* 优劣：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 可变对象 | 不可变对象 | 结论 |
| 资源利用 | 对象修改时，无需创建新对象，原对象直接修改完成。 | 每次对象修改，都会创建新对象。 | 当对象数据量较大时，尽量使用可变对象。可以避免频繁创建新对象以及丢弃旧对象的增加计算资源的无用操作。 |
| 数据灵活拷贝 | 当对象修改时，赋值到该对象的所有变量的值都会发生改变。 | 当对象修改时，赋值到该对象的所有变量的值不会发生改变，会创建一个新对象可以用新变量赋值绑定。 | 在于数据应用的保护，以及一些复杂逻辑的灵活应用 |

* 问题探索：

实例：

列表对象与元组对象改变对比

t=()

print(id(t))

t+=(1,) #实际是生成了新对象，新的元组与t绑定

print(id(t))

L = []

t = (1, 2)

print('old\_id(L)=', id(L), ' old\_id(t)=', id(t), sep='')

L += [3, 4]

t += (3, 4)

print('new\_id(L)=', id(L), ' new\_id(t)=', id(t), sep='')

'''

old\_id(L)=42246472 old\_id(t)=42246728

new\_id(L)=42246472 new\_id(t)=35045192 #对象被改变

'''

L运算与t运算的发生机制不同，可能带来浅拷贝深拷贝的问题。

注意：虽然元组是不可变对象，但是其内部的元素可以是可变的对象，并且元素可以直接在原对象上修改（此处待研究【？】为何会出现这种情况）

>>> a=[1,2,3]

>>> b=[2,3,4]

>>> t=(a,b)

>>> t

([1, 2, 3], [2, 3, 4])

>>> A,B=t

>>> A

[1, 2, 3]

>>> B

[2, 3, 4]

>>> t

([1, 2, 3], [2, 3, 4])

>>> id(t)

51048712

>>> A[1]=7 #其他方式t[0].append(9)，A.append(9)都不改变对象

>>> t

([1, 7, 3], [2, 3, 4])

>>> id(t)

51048712

现象推测：不可变对象似乎只对表层管用，像这样的第二层似乎根本无效。待验证【？】【这个逻辑问题已经弄清楚，这个例子太简单了，能想象】

实例：

def f(x, l=[1, 2]):

for i in range(x):

l.append(i \* i)

print(l)

f(2) # \_\_\_\_\_\_\_

f(3, [3, 2, 1]) # \_\_\_\_\_\_\_\_\_\_

f(3) # \_\_\_\_\_\_\_\_\_\_\_\_

* 类型区别：

不可变的类型:

bool, int, float, complex, str, tuple, frozenset, bytes(后面会讲)

可变的数据类型:

list, dict, set, bytearray(后面会讲)

**可变对象支持索引赋值和切片赋值**

如：列表的索引赋值

>>> L = [1, 2, 3]

>>> L[1] = 520

>>> L

[1, 520, 3] #id()函数可以证明，改变L[1]不改变id(L)

**不可变对象不支持索引赋值和切片赋值**

如：元组不支持索引赋值和切片赋值

>>> t=(1,2,3)

>>> t[1]

2

>>> t[1]=6

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

t[1]=6

TypeError: 'tuple' object does not support item assignment

#可变对象

>>> L=[1,2,3]; id(L)

55887048

>>> L+=[4,5,6]; id(L) #说明在原地址追加

55887048

>>> L1=L+[4,5,6]; id(L1)

55888008

#不可变对象

>>> s='abc'; id(s)

31157528

>>> s+='def'; id(s) #结果说明再次重新创建了对象绑定给变量s

51147472

>>> s1=s+'def'; id(s1)

55672048

### 深拷贝和浅拷贝问题探索

结论：

凡拷贝的对象皆会产生新对象并绑定；

否则只是建立新的绑定关系，对象不变，值改变会牵连一系列变化。

浅拷贝shallow copy

深拷贝deep copy

* 问题引入：

L1赋值于L2，对象不重建，L2改变L1也改变。

而L.copy()方法重新再创建对象，L2与L1不相干。

* 区别：

深拷贝：

import copy

L2 = copy.deepcopy(L)

浅拷贝：

L2 = L.copy

L2 = L[::]

不拷贝

L2 = L

探索实例：证明赋值运算符非拷贝，copy,copy为浅拷贝

>>> import copy

>>> a = [1,2,3]

>>> b=a

>>> c=copy.copy(a)

>>> id(a)

50207000

>>> id(b)

50207000

>>> id(c)

50207360

探索实例.py

print('浅拷贝')

L = [9, 8]

L1 = [1, 2, L]

L2 = L1.copy() # 浅拷贝，只能复制一层，不会复制深层变量绑定的对象

print('L=', L, ' L1=', L1, ' L2=', L2)

# L= [9, 8] L1= [1, 2, [9, 8]] L2= [1, 2, [9, 8]]

print(id(L), id(L1), id(L2)) # 42246472 42089480 42246472

print(id(L2) - id(L)) # 两者之间相差1600？？【？】

L[0] = '~' # 验证，改变深层对象的值，发现所有深层对象都被改变

print('L=', L, ' L1=', L1, ' L2=', L2)

# L= ['~', 8] L1= [1, 2, ['~', 8]] L2= [1, 2, ['~', 8]]

L1[0]='#' # 验证，改变浅层对象的值，结果不改变，因为浅层复制产生新对象

print('L=', L, ' L1=', L1, ' L2=', L2)

# L= ['~', 8] L1= ['#', 2, ['~', 8]] L2= [1, 2, ['~', 8]]

print('深拷贝')

import copy # 导入复制模块

L = [9, 8]

L1 = [1, 2, L]

L2 = copy.deepcopy(L1) # 深拷贝，复制深层对象，重新生成对象

print('L=', L, ' L1=', L1, ' L2=', L2)

# L= [9, 8] L1= [1, 2, [9, 8]] L2= [1, 2, [9, 8]]

print(id(L), id(L1), id(L2)) # 42574024 42577288 42417160

print(id(L2) - id(L))

L[0] = '~' # 验证，改变深层对象的值，深层复制的新对象的值不被影响

print('L=', L, ' L1=', L1, ' L2=', L2)

# L= ['~', 8] L1= [1, 2, ['~', 8]] L2= [1, 2, [9, 8]]'''

* 错误示例

字典得到的所有结果都会是最后一次输入的，因为每次修改都是指向同一个对象。

L = []

d = {} #所以该步应该放到循环内部

while True:

d['name'] = input('姓名：')

if d['name'] == ' ':

break

d['age'] = input('年龄：')

d['score'] = input('成绩：')

L.append(d)

print(L)

### 临时对象探索

结论：对象只能使用一次，后面学习到这是生成器

zip()实例探索：

>>> a=zip([1,2],['a','b'])

>>> a

<zip object at 0x0000000002EA8E48>

>>> D1=dict(a)

>>> D1

{1: 'a', 2: 'b'}

>>> D2=dict(a)

>>> D2

{}

列表推导式探索：

>>> a = (x for x in range(5))

>>> a

<generator object <genexpr> at 0x0000000002E58888>

>>> L1=list(a)

>>> L1

[0, 1, 2, 3, 4]

>>> L2=list(a)

>>> L2

[]

### Python复制的机理解释

* 现象描述：

函数方法，a、b变量两者均会改变

>>> a=[1,2,3]

>>> b=a

>>> b

[1, 2, 3]

>>> b.append(4) ##或者a.append(4)，或extend([4])，a、b两者效果一致

>>> b

[1, 2, 3, 4]

>>> a

[1, 2, 3, 4]

>>> a[0]='a'

>>> a

['a', 2, 3, 4]

>>> b

['a', 2, 3, 4]

而直接赋值，a、b变量两者会单独改变

>>> a=[1,2,3]

>>> b=a

>>> b=[1,2,3,4]

>>> b

[1, 2, 3, 4]

>>> a

[1, 2, 3]

原理：浅层复制。a与b只是列表的名字等效了，并没有对其储存数据的地址进行从新分配，所以动其中一个会牵连另一个

* 正确示例

>>> a=[1,2,3,4]

>>> b=a[:]

>>> b

>>> b[0]=8

>>> b

[8, 2, 3, 4]

>>> a

[1, 2, 3, 4]

# 逻辑运算

## 逻辑语句

基本概念，语句与表达式区别见【2.3.1.1 语句和表达式】

什么为一个完整语句块？

while语句：可以用于死循环

for语句：用于<可迭代对象>的循环（字符串，range()返回对象）

break语句：用于终止当前循环

continue语句：用于开始一次新循环

注：布尔值可以直接以True和False代替

### if语句

**作用：**选择性执行语句。

语法：

if 真值表达式:

语句1

elif 真值表达式2:

语句2

elif 真值表达式3:

语句3

...

else:

语句块4

**运行机理：**本质判断是：if 布尔值。实际运行的是if bool(obj)，比如if 100为if bool(100)

### while语句

格式：

while <真值表达式1>：

语句1

else <真值表达式2>：

语句2

运行机制：

第一行while一行中，计算判断<真值表达式1>。

若为True，则执行完后面语句后又跳回while一行，重新计算判断<真值表达式1>。

若为False，则执行else语句，else语句可以省略。

### for语句

* 格式：

**作用：**循环遍历，返回可迭代“对象”元素

for <变量列表> in <可迭代对象>：

语句块1

else:

语句块2

* 示例：

for的高级用法，使用两个变量

>>> for x,y in (('x',1),('y',2),('z',3)):

print(x,y)

x 1

y 2

z 3

# 所以可迭代对象可以变成items

* for循环运行机制：

**1第一行中：**读for语句，计算判断<可迭代对象>，存于内存。

**2第一行中：**将上次赋值元素的下一个元素赋值给x。初始为第1个元素。

**3第二行中：**进入循环内部语句块参与其他计算。

循环一次结束回去重复执行2和3步骤，直到<可迭代对象>所有元素遍历完，待所有元素迭代完进入else语句(可无)循环结束。

**注意：**

<可迭代对象>在第一次运算完后，循环回来不再计算,只在for循环适用。

while中每次循环都要语句重新计算<真值表达式>判断。【？其他的真的很想都删掉，感觉都是复杂化描述】

**如：**

print('for循环探索')

i = 6

for x in range(1, i): # 迭代[1,i)，每个循环向后逐个取出元素，赋值给x

print('x=', x, ' i=', i)

i -= 1

print('while循环探索')

i = 6

x = 1

while x in range(1, i): # 判断x是否在[1,i)内

print('x=', x, ' i=', i)

x += 1

i -= 1

# for循环探索

# x = 1 i = 6

# x = 2 i = 5

# x = 3 i = 4

# x = 4 i = 3

# x = 5 i = 2

# while循环探索

# x = 1 i = 6

# x = 2 i = 5

# x = 3 i = 4

机制探索

s ={1,2,3,4,5}

for i in s:

if i == 2:

s.remove(4)

s.add(44)

# s.update({7,8}) # 但不能更改s长度，否则触发RuntimeError异常

print(i)

### pass语句

填充语法空白

如：if语句中的使用

if Ture:

pass

else:

print('输入有问题')

### break语句

退出当前语句块

注意1：不干扰外层其他语句块

如：

for i in range(2):

print(i)

for j in ['a', 'b', 'c']:

print('j:', j)

if j == 'b':

break

'''

0

j: a

j: b

1

j: a

j: b

'''

注意2：在for-else和while-else语句中break是退出整个语句块：

如：

for-else中else的语句被跳过：

|  |  |
| --- | --- |
| for i in [1,2,3,4]:  if i==2:  **continue**  print(i)  else:  print("hehe")  结果为  1  3  4  hehe | for i in [1,2,3,4]:  if i==2:  **break**  print(i)  else:  print("hehe")  结果为  1 |

### continue语句

**问题引入：**如何让程序不再向下执行，重新开始一次新的循环？

**说明：**

for循环中迭代取下一个元素。

while语句中，真值表达式中会重新判断条件。

### 其他语句

#### 条件表达式

**语法：**<表达式1，为真返回> if <真值表达式> else <表达式2，为假返回>

**作用：**灵活简洁

**再次强调：**表达式不是语句，表达式能赋值给变量，语句则不能。

# 使用条件表达式计算绝对值

b = -s if s<0 else s

b =(-s if s<0 else s)

m = n if n > 0 else -n

#### 语句嵌套

一个语句嵌套到另一个语句内部

#### 布尔运算

**返回：**布尔值或对象

**运算符：**not、and、or

布尔非操作：not x

>>> not True

False

布尔与操作：x and y

机理：从左到右运算，只要有一个不满足就返回x值，否则返回y

优先返回假值对象。

先计算x，若x为False，返回x，否则返回y值。

使用对象表示真假，不同于C/C++的“&”【？】（？？？）。

>>> 0 and 20 #优先返回假值对象

0

>>> 20 and 0

0

>>> 0 and 20 #若x为False，返回x，否则返回y值。

0

>>> 10 and 10

10

>>> 10 and 20

20

>>> 20 and 10

10

布尔或操作：x or y

机理：从左到右运算，只要有一个满足就返回x值，否则返回y

优先返回真值对象。

先计算x，若x为True，返回x，否则返回y值。

>>> 0 or 20 #优先返回真值对象

20

>>> 20 or 0

20

>>> 0 or 20 #若x为True，返回x，否则返回y值。

20

>>> 10 or 10

10

>>> 10 or 20

10

>>> 1 or 20

1

>>> 20 or 1

20

**备注：**这个过程完全满足逻辑判断。如

>>> 6>5 and 6>3

True

>>> 6>5 and 7

7

#### 正负号运算符

+（正号）

-（负号）

一元运算符

语法：+/- 表达式

## 常见联用函数

### range()函数

**格式：**range(s,e,step)，s--开始数值，e--结束数值(不包含)，step--步数(可为负数)

**返回：**返回数值序列的“对象”，非字符串（序列）

>>> print(range(3,6))

range(3, 6)

>>> for i in range(3,6):

print(i， end = ' ')

3 4 5

**常见方法示例：**

range(3,8) #3,4,5,6,7

range(3,8,2) #3,5,7

range(3,0) #无

range(3,0,-1) #3,2,1

规则类似于字符串操作中的切片

数据集合与优化【？】

# 函数function

## 基本概念

概念：

函数是可以重复执行的语句块。

可以重复调用

用于封装语句，调高代码重用性

用于定义用户级别的函数

语法：

def 函数名(形参列表):

语句块

函数名(实参)

说明：

函数的名字就是语句块的名称

函数名的命名规则与变量名相同（函数名必须是标识符）

函数名是一个变量

函数有自己的命名空间，在函数外部不可访问函数内部的变量，在函数内部可以访问外部的变量，要让函数处理外部数据需要用参数给函数传入一些数据。

参数列表可以为空

语句部分不能为空，如果为空需要用pass填充。

函数调用是一个表达式

如果没有return语句，函数执行完毕后返回None对象

如果函数需要返回其它的对象需要用到return语句

## 函数的返回值

* return语句

作用：结束函数的运行，并返回return的结果

示例：

定义函数的5种思路方法【？待总结】

def mufun(a,b,c)

* 注意事项：
* 注意事项1：函数中无return也返回一个None对象

函数中没有return语句相当于末尾加了return None

函数的运行本质，return之深入理解：print函数深度解读

>>> s = print('hello'); print(s) # 返回什么？

>>> print(print('hello')) #返回什么？

hello

None

def myfn():

def myfn2():

print('hello')

return myfn2()

print(myfn())

'''

hello

None

'''

* 注意事项2：函数中return后不再返回原函数执行

递归过程中，return fn()跳转到另一个函数后，将不会返回原函数f()中继续执行。

|  |  |
| --- | --- |
| def f(x):  if x == 0:  print('程序结束, x:', x)  return x  else:  print('开始return跳转, x:', x)  return f(x - 1) # 使用本句直接跳转  print('开始执行f(x-1)')  f(x - 1) #返回继续执行  print('结束执行f(x-1)')  print(f(3))  '''  开始return跳转, x: 3  开始return跳转, x: 2  开始return跳转, x: 1  程序结束, x: 0  0  ''' | def f(x):  if x == 0:  print('程序结束, x:', x)  return x  else:  print('开始return跳转, x:', x)  # return f(x - 1) # 使用本句直接跳转  print('开始执行f(x-1)')  f(x - 1) #返回继续执行  print('结束执行f(x-1)')  print(f(3))  '''  开始return跳转, x: 3  开始执行f(x-1)  开始return跳转, x: 2  开始执行f(x-1)  开始return跳转, x: 1  开始执行f(x-1)  程序结束, x: 0  结束执行f(x-1)  结束执行f(x-1)  结束执行f(x-1)  None  ''' |

## 函数参数传递

（一）.函数传参的方式：

1.位置传参：实参按位置形参

  \*序列传参：属于位置传参的一种，用‘\*’将序列拆解后按位置一一对应传参

2.关键字传参：实参和形参通过关键字名称一一对应

 \*字典传参：属于关键字传参的一种，用‘\*\*’将字典拆解后按关键字传参

\*关键字形参要定义在位置形参之后

（二）.缺省参数：

\*在没有字典形参的情况下，缺省形参必须要定义在最后。

（三）.形参的定义：

位置形参：按位置一一对应

元组形参：属于位置形参的一种，用\*args来收集位置传参的时候多余的实参，并组成一个元组

关键字形参：只有具有相同的关键字的实参才能一一对应。（没有关键字的就是位置传参了）

字典形参：和元组形参差不多的作用，针对多出来的关键字实参进行收集并组成字典，（\*\*kwargs）

详见笔记day09.txt

* 实参传递方式
* 形参接收方式

【？待总结】

* 函数实参与形参区别

形参是指调用函数时传递的参数。

实参是指接收调用函数传入的参数。

一个实例看懂所有传参方式：

|  |  |
| --- | --- |
| def myfun(a, b, c):  print(a, b, c)  myfun(1, 2, 3)  myfun(\*(1, 2, 3))  myfun(\*[1, 2, 3])  myfun(\*\*{'a': 1, 'b': 2, 'c': 3})  myfun(a=1, c=3, b=2)  '''  1 2 3  1 2 3  1 2 3  1 2 3  1 2 3  ''' | def myfun(\*args, \*\*kwargs):  print(args, kwargs)  myfun(1, 2, 3)  myfun(\*(1, 2, 3))  myfun(\*[1, 2, 3])  myfun(\*\*{'a': 1, 'b': 2, 'c': 3})  myfun(a=1, c=3, b=2)  '''  (1, 2, 3) {}  (1, 2, 3) {}  (1, 2, 3) {}  () {'a': 1, 'b': 2, 'c': 3}  () {'a': 1, 'c': 3, 'b': 2}  ''' |

### 位置传参

* 语法：

固定位置传参。

说明

形参按位置接收；

实参按位置传入，或

def myfun(a,b,c): # 形参

print(a, b, c)

myfunc(1,2,3) # 实参

myfun(\*'123') # 等同于a='1',b='2',c='3'

myfun(\*(1,2,3))

myfun(\*[1,2,3])

### \*序列传参

* 1）序列实参传入

def fb(a, b, c): # 形参接收

print(a, b, c)

fb(1, 2, 3)

fb(\*(1, 2, 3)) # 等同于按位置传参

fb(\*[1, 2, 3])

* 2）序列形参接收
* 语法：

收集多余的位置传参。

传参后的返回值：元组序列

* 实例：

def fb(\*args): # args行业规则，一般存在用此名字，其他变量名也可以

print(args, type(args), sum(args))

fb()

fb(1, 2, 3)

fb(\*(1, 2, 3))

fb(\*[1, 2, 3])

'''

() <class 'tuple'> 0

(1, 2, 3) <class 'tuple'> 6

(1, 2, 3) <class 'tuple'> 6

(1, 2, 3) <class 'tuple'> 6

'''

### \*\*关键字传参

5.3.4.1

F(a=1, c=2

* 1）字典关键字实参
* 字典前面加“\*\*”

在传入函数前被拆解

字典键名必须与形参名一致，键名必须为字符串。

def mufun(a,b,c):

print(a,b,c)

myfun(\*\*{'a' : 1, 'b' : 2, 'c' : 3}) # 等同于(a=1,b=2,c=3)

* 强制c,d必须用关键字传参

示例1：

def fa(a, b, \*, c, d):

print(a, b, c, d)

fa(1, 2, d=400, c=300) # 1 2 300 400

# fa(1, 2, 400, 300) #报错

示例2：

def fb(a, b, **\*args**, c, d):

print(a, b, **args**, c, d)

fb(1, 2, **3, 4, 5**, \*\*{'d':400, 'c':300}) # 1 2 **(3, 4, 5)** 300 400

fb(1, 2, **3, 4**, d=400, c=200) # 1 2 **(3, 4)** 200 400

# fb(1, 2, 3, 4, 400, 200) # 报错

* 2）字典关键字形参
* 语法：

收集多余的关键字传参。

传参的返回值：字典

实例：

def fa(**xx,** \*\*kwargs):

print(**"xx =", xx** , "kwargs =", kwargs)

fa(\*\*{'a': 1, 'b': 2, 'c': 3, **'xx': 0**})

fa(a=10, b=20, c=30, **xx=0**)

'''

xx = 0 kwargs = {'a': 1, 'b': 2, 'c': 3}

xx = 0 kwargs = {'a': 10, 'b': 20, 'c': 30}

'''

* 缺省形参

在形参加入初始值，传参时便可省略一部分值。

缺省参数规则：

缺省参数可以有零个和多个，甚至全部。

若函数中有缺省参数存在，则必须遵循从右到左依次定义的规则。即缺省参数必须放在参数后面。如(x, y=0, z=0)

如：自己定义的range函数

def myrange(s, end=0, n=1):

x = s #代表数开始

y = end #代表数结束

if end == 0:

x = end

y = s

L = []

while x < y:

L.append(x)

x += n

print(L)

myrange(10)

myrange(5, 10)

myrange(5, 10, 2)

# myrange(s=5, end=10, 2) #报错

### 综合传参

位置传参可随意组合位置，传入也是位置；关键字传参必须明确定义传参变量

示例：

def myfun(a, b, \*args, c, d, \*\*kwargs):

print(a, b, c, d, args, kwargs)

myfun(1, 2, c=3, d=4)

myfun(1, d=4, b=2, c=3)

myfun(\*\*{'a': 1, 'b': 2, 'c': 3}, d=4)

myfun(\*(1, 2, 3, 4, 5, 6, 7), c=4, d=5, \*\*{'other': None})

'''

1 2 3 4 () {}

1 2 3 4 () {}

1 2 3 4 () {}

1 2 4 5 (3, 4, 5, 6, 7) {'other': None}

'''

### 函数传参

Python特点：可以传函数

函数作为另一个函数的返回值 (从一个函数中返回函数名，作为另一个函数的传参值)

示例：

灵活传max,min,sum函数进形参

def myinput(fn):

L = []

while True:

x = int(input("请输入大于0的整数:") or '-1')

if x < 0:

break

L.append(x)

return fn(L) # <<< 注意此处

print(myinput(max))

print(myinput(min))

print(myinput(sum))

# function\_return\_other\_function.py

# 此示例示意get\_op这个函数可以返回其它的函数

def get\_op():

s = input("请输入您要做的操作: ")

if s == '求最大':

return max

elif s == '求最小':

return min

elif s == '求和':

return sum

L = [1, 2, 3, 4]

fx = get\_op()

print( fx(L) )

* 函数的嵌套定义

函数的嵌套定义是指一个函数里用def语句来创建其它函数的情况。

嵌套内的执行语句不被自动执行。

执行一个总函数，内部函数也会执行。

不能直接调用嵌套内部函数，但可以外部调用？

## 函数名变量

函数名是变量，它在创建时绑定一个函数。

一个函数可以作为另一个函数实参传递

# 示例:

**def** f1():

print(**"f1被调用"**)

**def** f2():

print(**"f2被调用"**)

**def** fx(fn):

print(**"fn绑定的是: "**, fn)

fn() # 调用fn绑定的函数, 此处等同于调用谁呢?

fx(f1)

fx(f2)

函数也是一个对象，可被函数名(变量)绑定，甚至可以交换绑定f1,f2

# 示例：

# function\_variable.py

def f1():

print("hello f1")

def f2():

print("hello f2")

fx = f1 # 注意此处f1没有加()

fx() # 调用f1

f1() # 调用f1

f1 = f2 # 让f1 改变绑定关系,去绑定f2

f1() # 调用f2

f2 = fx

f2() # 调用f1 # 交换法(三次交换赋值)

## 函数作用范围

问题引入：

函数内可使用外部变量，但不可更改变量所绑定的对象。

def func(x):

print('x:', x, 'y:', y)

# y += 10 # 报错，域名空间之外变量绑定对象不可变

print(L) # [0]

L[0] += 1

print(L) # [1] # 对象可变

L = [0]

y = 0

func(9) # 同一命名空间，函数可使用，但不可更改变量和其绑定的对象(若对象是可变的，当然也可变)

注意事项：

if True:

if True:

data = 'hello'

print(data) # 正常输出'hello'

### 局部变量与全局变量

* 局部变量

定义在函数内部的变量称为局部变量（函数的形参也是局部变量）

局部变量只能在函数内部使用。

局部变量在函数调用时会创建，被调用后会被销毁

局部变量说明：

1. 在函数内首次对变量赋值是创建局部变量,再次为变量赋值是修改局部变量的绑定关系

2. 在函数内部的赋值语句不会对全局变量造成影响

3. 局部变量只能在其被声明的函数内部访问,而全局变量可以在整个模块同访问

* 全局变量

定义在函数外部,模块内部的变量称为全局变量

所有的函数都可以直接访问"全局"变量,但函数内部不能直接通过赋值语句来改变全局变量

示例：

函数内部局部变量修改全局变量

# 若函数内部无同名全局变量，则调用正常，但不能改变变量绑定对象

def f():

print(y) # 10

y = 10

f()

# 若函数内部有同名全局变量，在重新赋值后可正常运行

def f():

y = 20

print(y) # 20

y = 10

f()

# 若函数内部有同名全局变量，且调用出现在变量重新赋值语句前，则报错

def f():

print(y) # 报错：UnboundLocalError: local variable 'y' referenced before assignment

y = 20

y = 10

f()

### Python作用域

作用域也叫命名空间，是访问变量时查找变量名的范围空间。

#### Python的四个作用域：LEGB

|  |  |  |
| --- | --- | --- |
| 作用域 | 英文解释 | 英文简写 |
| 局部作用域（函数内） | Local（function） | L |
| 外部嵌套函数作用域 | Enclosing function | E |
| 函数定义所在模块（文件）的作用域 | Globals（module） | G |
| Python内置模块的作用域 | Builtin（python） | B |

说明：

包含关系：Builtin > Globals > Enclosing > Local

模块 文件 外部函数 当前函数

一般全局变量为Globals，本地变量为Local。

Builtin内建作用域变量如max,min,len等是只读的不能改变。变量和del都是在操作Globals作用域，如重定义max，优先访问全局的

变量名的查找规则：

由内到外。L --> E --> G --> B。

示例：嵌套函数调用

def print\_y(): # 属于外部嵌套函数Enclosing function

print(y)

def f():

y = 20

print\_y() # 10

print(y) # 20

y = 10

f()

示例：重定义max

def max(a, b):

return min(a, b)

print(max(10, 9)) #9

#### globals和locals函数打印作用域字典

globals() 返回当前全局作用域内变量的字典

locals() 返回当前局部作用域内的变量的字典

实例：

# globals\_locals\_function.py

a = 1

b = 2

c = 3

**def** fx(c, d):

e = 300

# 此处有几个局部变量?

print(**'locals() 返回'**, locals()) *# 3个变量：c,d,e*

print(**"globals() 返回"**, globals()) *# 4个变量：a,b,c,fx，及其他*

print(c) # 100

print(globals()[**'c'**]) *# 3*

fx(100, 200)

### 变量作用范围机制探索

不可变对象改变，会绑定新对象，作用域结束会销毁。可变对象改变则能影响到全局变量。

* 运行机制探索实例：

验证：变量指向的可变对象（如列表）的修改，不能绑定新对象，值改变会超越局部变量。

v=100

def f(v):

v=50 # v绑定新对象

f(v)

print(v) #100

v=100

def f(v):

v+=50 # v +=操作是对不可变对象，会生成新对象

f(v)

print(v) #100

v=[100]

def f(v):

v=[50] # v绑定新对象

f(v)

print(v) #[100]

v=[100]

def f(v):

v+=[50] # v为绑定为可变对象，+=操作不会生成新对象

# v.extend([50])

# v.append(50)

f(v)

print(v) #[100, 50]

再次验证：不可变对象（如字符串）的改变，会绑定新对象，作用域结束会销毁。

v='100'

def f(v):

v='50'

f(v)

print(v) #'100'

## 高级常用函数

### lambda表达式（lambda匿名函数）

引入：

除了def语句可以创建函数之外，lambda表达式也可以创建函数。

概念：

lambda表达式（又称匿名函数），用于封装有限的逻辑的函数

lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。

语法格式：

lambda [变量1，变量2，… ]:表达式

说明：

可以省略变量

只能包含一条表达式

* 示例见：

>>> myadd = lambda x,y:x+y

>>> print(myadd(10,20))

30

>>> myadd=lambda :print('hello')

>>> print(myadd())

hello

None

fun = **lambda** x: sum([x \* x **for** x **in** range(1, x + 1)])

**print**(fun(10))

### eval函数 和 exec函数

* eval函数【？疑问待解决，当存在缩进如何解决】

作用:

把一个字符串当成一个表达式来执行,返回表达式执行后的结果

格式:

eval(source [,globals=None[,locals=None]] )

运行机制：

先将参数1字符串表达式格式化预执行，再用参数2赋值，再用参数3赋值

示例：

while True:

s = input("请输入表达式 >>> ")

if not s:

break

print(eval(s))

'''

请输入表达式 >>> print('hello')

hello

None

请输入表达式 >>> 2\*3+4

10

'''

运行机制深究：

x = 100

y = 200

v = eval('x + y')

print(v) # 300

v = eval('x + y', None, None)

print(v) # 300

# v = eval('x + y', {'x':1},None)

# print(v) # 1 + y 出错,因为给定的字典内，y未定义

v = eval('x + y', None, {'y': 2})

print(v) # 102，可以单独传入，使用的x是全局变量，y是自定义变量

v = eval('x + y', {'x': 2}, {'y': 8})

print(v) # 10

v = eval('x + y', {'x': 2, 'y': 8}, {'x': 3, 'y': 4})

print(v) # 7

# 自定义传入参数示例

my\_dict = {'x': 1, 'y': 2}

v = eval('x + y', my\_dict, {'y': 20})

print(v) # 21

* exec函数:

作用:

把一个字符串当成'程序'来执行

格式:

exec(source, globals=None, locals=None)

示例:

x = 100

y = 200

s = '''z=x+y

print('z=', z)

print("hello world!")

'''

exec(s) # 执行 s这个字符串 # z= 300 # hello world!

print(z) #300

dict\_local = {'x': 1}

exec(s, None, dict\_local) # z= 201 # hello world!

print(dict\_local) # {'x': 1, 'z': 201}

## 函数式编程

* 函数式编程

**概念：**是指用一系列函数解决问题

函数是一等公民(Guido)

1. 函数本身可以赋值给变量,赋值后变量绑定函数

2. 允许将函数本身作为参数传入另一个函数

3. 允许函数返回一个函数

**好处:**

用每一个函数完成细小的功能,一系列函数在任意组合可以完成大问题

**函数的可重入性:**

当一个函数在运行时不读取和改变除局部作用域以外的变量时,此函数为可重入函数

可重入函数在每次调用时,如果参数一定,则结果必然一定

示例:

可重入函数:

def add1(x, y):

return x + y

不可重入函数示例:

y = 200

def add2(x):

return x + y

print(add2(10)) # 210

y = 300

print(add2(10)) # 310

### 高阶函数

**概念：**什么是高阶函数(high order function)？

满足下列条件中一个的函数即为高阶函数

1. 函数接收一个或多个函数作为参数传入

2. 函数返回一个函数

* python中内建的高阶函数:

map, filter, sorted

map是用函数取迭代对象的值

filter是用函数判断筛选对象的值

sorted是用函数排序对象的值

#### map 函数

格式：

map(func, \*iterables)

func是<可传参函数>，\*iterables是<可迭代对象>

用函数和对可迭代对象中的每一个元素作为参数返回新的可迭代对象;当最短的一个可迭代对象不再提供数据时迭代结束

返回：

可迭代对象

要求:

func函数接收的参数个数必须与可迭代对象的个数相同【？我不认同】

示例:

>>> def square(x) : # 计算平方数

return x \*\* 2

>>> map(square, [1,2,3,4,5]) # 计算列表各个元素的平方

[1, 4, 9, 16, 25]

>>> map(lambda x: x \*\* 2, [1, 2, 3, 4, 5]) # 使用 lambda 匿名函数

[1, 4, 9, 16, 25]

# 提供了两个列表，对相同位置的列表数据进行相加

>>> map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])

[3, 7, 11, 15, 19]

# 字符拼接

>>> list = ['abc','def','xyz']

>>> map(lambda x: 'str\_'+x, list);

['str\_abc', 'str\_def', 'str\_xyz']

**for** x **in** map(**lambda** x: x\*\*2 , range(1, 5)):

print(x, end=**' '**) *#* *1 4 9 16*

**for** x **in** map(**lambda x, y: x + y**, [1, 2, 3, 4], [5, 6, 7, 8, 9, 10]):

print(x, end=**' '**) *# 6 8 10 12*

练习 :

1. 求 1\*\*2 + 2\*\*2 + 3\*\*2 + ...+ 9\*\*2的和

(用函数式和高阶函数map实现)

s=0

for x in map(lambda x:x\*\*2,range(1,10)):

s+=x

print(s)

2. 求: 1\*\*9 + 2\*\*8 + 3\*\*7 + .... + 9\*\*1的和

s = 0

**for** x **in** map(**lambda** x, y: x \*\* y,

range(1, 10),

reversed(range(1, 10))):

s += x

print(s)

#### filter 函数:

格式:

filter(function, iterable)

作用:

筛选可迭代对象iterable中的数据,返回一个可迭代对象,此可迭代对象将对iterable提供的数据进行筛选

说明:

函数function 将对iteralbe中的每个元素进行求布尔值,返回True则保留,返回False则丢弃

示例见:

print('test1:')

for x in filter(lambda x: x % 2 == 1, range(41, 53)):

print(x, end=' ') # 41 43 45 47 49 51

print('\ntest2:')

def func1(x):

if x % 2 == 1:

return x

for x in filter(func1, range(41, 53)):

print(x, end=' ') # 41 43 45 47 49 51

练习:

1. 把之前写的is\_prime(x)判断x是否为素数的函数复制过来

1) 用filter函数把100到200的全部素数求出来,放入列表L中

2) 求300 ~ 400之间全部素数的和

#### sorted 函数

作用:

将原可迭代对象提供的数据进行排序，生成排序后的列表

格式:

sorted(iterable, key=None, reverse=False)

说明:

iterable 可迭代对象

key 函数是用来提供一个排序参考值的函数，这个函数的返回值将作为排序的依据

key的值有：

reverse 标志用来设置是否降序排序

* 简单示例:

L = [5, -2, -4, 0, 3, 1]

L2 = sorted(L) # [-4, -2, 0, 1, 3, 5]

# 要得到这样的结果该怎么办？

L3 = sorted(L, key=abs) # [0, 1, -2, 3, -4, 5]

names = ['Tom', 'Jerry', 'Spike', 'Tyke']

L = sorted(names, key=len)

# 结果 ['Tom', 'Tyke', 'Jerry', 'Spike']

* 示例：与lambda高级联用

**1、lambda与sorted联用对列表里的字典排序**

>>> L = [{'a': 1, 'b': 4}, {'a': 1111, 'b': 2}, {'a': 1111, 'b': 3}]

>>> sorted(L, key=lambda d: d['b'], reverse=False)

[{'a': 1111, 'b': 2}, {'a': 1111, 'b': 3}, {'a': 1, 'b': 4}]

2、对字典进行按key排序

>>> d={'a':25,'c':27,'b':20,'d':22}

>>> sorted(d.keys())

['a', 'b', 'c', 'd']

3、对字典进行按values排序

>>> d={'a':25,'c':27,'b':20,'d':22}

>>> sorted(d.items(),key=lambda item:item[1])

[('b', 20), ('d', 22), ('a', 25), ('c', 27)]

>>> sorted(d.items(),key=lambda item:item[1],reverse=True)

[('c', 27), ('a', 25), ('d', 22), ('b', 20)]

4、对列表多重排序

>>> sorted([2,3,4,1],key=lambda x: x)

[1, 2, 3, 4]

>>> sorted([2,3,4,1],key=lambda x: x\*-1)

[4, 3, 2, 1]

>>> sorted([['a',2], ['b',3], ['c',4], ['d',1]],key=lambda x: x[1]\*-1)

[['c', 4], ['b', 3], ['a', 2], ['d', 1]]

对sorted的key=function中lambda的理解：

>>> sorted([['a',2, 1], ['b',3,1],['bb',4,1], ['c',4,2], ['d',1,2]],key=lambda x: (x[2], x[1]\*-1))

[['bb', 4, 1], ['b', 3, 1], ['a', 2, 1], ['c', 4, 2], ['d', 1, 2]]

>>> def fn(x):

... return (x[2], x[1]\*-1)

...

>>> sorted([['a',2, 1], ['b',3,1],['bb',4,1], ['c',4,2], ['d',1,2]],key=fn)

[['bb', 4, 1], ['b', 3, 1], ['a', 2, 1], ['c', 4, 2], ['d', 1, 2]]

利用模块，编程排序（不如lambda灵活，特别是目前未找到多重排序如何实现一正排序同时一负排序）

Operator 模块函数（[python 列表排序方法sort、sorted技巧篇 - whaben - 博客园](https://www.cnblogs.com/whaben/p/6495702.html)）

上面的key参数的使用非常广泛，因此python提供了一些方便的函数来使得访问方法更加容易和快速。operator模块有itemgetter，attrgetter，从2.6开始还增加了methodcaller方法。使用这些方法，上面的操作将变得更加简洁和快速：

>>> from operator import itemgetter, attrgetter

>>> sorted(student\_tuples, key=itemgetter(2))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student\_objects, key=attrgetter('age'))

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

operator模块还允许多级的排序，例如，先以grade，然后再以age来排序：

>>> sorted(student\_tuples, key=itemgetter(1,2))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

>>> sorted(student\_objects, key=attrgetter('grade', 'age'))

[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

练习:

已知:

names = ['Tom', 'Jerry', 'Spike', 'Tyke']

排序的依据为原字符串反序的字符串

'moT', 'yrreJ', 'ekipS', 'ekyT'

结果:

['Spike', 'Tyke', 'Tom', 'Jerry']

参考：

>>> sorted(['Tom', 'Jerry', 'Spike', 'Tyke'],key=lambda x:x[::-1])

['Spike', 'Tyke', 'Tom', 'Jerry']

### 递归函数

递归函数 recursion

函数直接或间接的调用自身

说明：

递归一定要控制递归的层数，当符合一定条件时要终止递归调用

几乎所有的递归都能用while循环来代替

优点:

递归可以把问题简单化，让思路更为清晰,代码更简洁

缺点:

递归因系统环境影响大，当递归深度太大时，可能会得到不可预知的结果

示例:

**def** f():

f() # 直接调用自己，进入递归

f()

# 函数间接调用自身

**def** fa():

fb()

**def** fb():

fa()

fa()

print(**"递归完成"**)

递归的两个阶段：

递推阶段: 从原问题出发，按递归公式递推从未知到已知，最终达到递归的终止条件

回归阶段: 按递归终止条件求出结果，逆向逐步代入递归公式，回归到问题求解

示例见:

**def** fn(n):

print(**"递归进入第"**, n, **'层'**)

# 当递归进入第三层时，将不再向下走，开始回归

**if** n == 3:

**return**

fn(n + 1)

print(**'递归退出第'**, n, **'层'**)

fn(1)

print(**"程序结束"**)

'''

递归进入第 1 层

递归进入第 2 层

递归进入第 3 层

递归退出第 2 层

递归退出第 1 层

程序结束

'''

示例:

写一个函数求n的阶乘（递归实现）

见:

def myfac(n):

if n == 1:

return 1

return n \* myfac(n-1)

print('5的阶乘是:', myfac(5))

递归的实现方法

先假设函数已经实现

练习:

写一个函数mysum(n), 用递归方法求

1 + 2 + 3 + 4 + .... + n的和

def mysum(n):

....

print(mysum(100)) # 5050

**def** mysum(n):

**if** n == 1:

**return** 1

**return** n + mysum(n-1)

print(mysum(4))

### 闭包 closure 闭包函数【？待学习】

概念：

闭包是指引用了此函数外部嵌套函数作用域变量的函数

【换句话说，就是定义一次的函数变量的参数可以继续使用】

闭包必须满足三个条件:

1. 必须有内嵌函数（def一行）

2. 内嵌函数必须引用外部函数中的变量（y）

3. 外部函数返回值必须是内嵌函数.（fn）

#### 实例

示例1:

def make\_power(y):

def fn(x):

return x \*\* y

return fn

pow2 = make\_power(2)

print('5的平方是:', pow2(5))

print('5的平方是:', make\_power(2)(5)) # 还可以一句话直接生成

pow3 = make\_power(3)

print("6的立方是", pow3(6))

# 计算

# 1\*\* 2 + 2\*\*2 + 3\*\*2 + ..... + 9 \*\* 2

print(sum(map(lambda x: x\*\*2, range(1, 9))))

print(sum(map(make\_power(2), range(1, 9))))

示例2：

def get\_fx(a, b, c):

def fx(x):

return a\*x\*\*2 + b\*x\*\*1 + c

return fx

# 创建函数 f(x) = 1\*x\*\*2 + 2\*x\*\*1 + 3

f123 = get\_fx(1, 2, 3)

print(f123(20))

print(f123(50))

f654 = get\_fx(6,5,4)

print(f654(20))

print(f654(50))

我的理解：

def kuangjia(kuangjiax):

def func(x):

print('hehe')

print(kuangjiax)

print(x)

return func

do = kuangjia('mmp')

do('666')

#### 应用：

* 减少print(file=fo)的写法次数

import time

def f(foname):

def g(\*args):

print(\*args, file=fo)

fo = open(foname, 'w', buffering=1)

return g

def main():

myprint = f('my.log')

# print([1, 2], {4: 4}, 1, 2)

myprint([1, 2], {4: 4}, 1, 2)

# fo.write(str([1, 2]) + str({4: 4}))

time.sleep(20)

# print([1, 2], {4: 4}, 1, 2)

myprint([1, 2], {4: 4}, 1, 2)

if \_\_name\_\_ == '\_\_main\_\_':

main()

练习:

1. 已知:

第五个人比第四个人大2岁

第四个人比第三个人大2岁

第三个人比第二个人大2岁

第二个人比第一个人大2岁

第一个人说他10岁

编写程序算出第5个人几岁?

(思考是否可以使用递归和循环两种方式来做)

# 循环实现

x=1

age=10

while x<5:

age=age+2

x+=1

print(age)

# 递归实现

**def** f(n):

**if** n==1:

**return** 10

**return** 2+f(n-1)

print(f(5))

2. 已知有列表:

L = [[3, 5, 8], 10, [[13, 14], 15, 18], 20]

1) 写一个函数print\_list(lst)打印出列表内的所有元素

print\_list(L) # 3 5 8 10 13 14...

2) 写一个函数sum\_list(lst): 返回这个列表中所有元素的和

print(sum\_list(L)) # 86

注:

type(x) 可以返回一个变量的类型

如:

>>> type(20) is int # True

>>> type([3, 5, 8]) is list # True

**def** func(Lx):

**if** type(Lx) == typeL:

**for** i **in** Lx:

print(**'Lx'**, Lx)

print(**'i'**, i)

**return** func(i)

**else**:

print(**'Lx'**, Lx, end=**' '**)

L = [[3, 5, 8], 10, [[13, 14], 15, 18], 20]

typeL = type(L) # type([3, 5, 8]) is list # True

func(L)

# 结果如下，为何for循环不能自动向后迭代？

'''

Lx [[3, 5, 8], 10, [[13, 14], 15, 18], 20]

i [3, 5, 8]

Lx [3, 5, 8]

i 3

Lx 3

'''

3. 改写之前的学生信息管理程序

要求添加四个功能:

| 5) 按学生成绩高-低显示学生信息 |

| 6) 按学生成绩低-高显示学生信息 |

| 7) 按学生年龄高-低显示学生信息 |

| 8) 按学生年龄低-高显示学生信息 |

模块也是对象

### 装饰器 decorators(专业提高往篇)

**问题:**

函数名 / 函数名() 区别

函数名是变量，它绑定一个函数

什么是装饰器？

**概念：**

装饰器是一个函数，主要作用是用来包装另一个函数或类(后面会讲)

**作用：**

是在不改变原函数名(或类名)的情况下改变被包装对象的行为

**函数装饰器:**

函数装饰器是指装饰器是一个函数，传入的是一个函数，返回的也是一个函数

**语法:**

def 装饰器函数名(参数):

语句块

return 函数对象

@装饰器函数名<换行>

def 函数名(形参列表):

语句块

函数名()

优秀参考文档：

这是我见过最全面的Python装饰器详解！没有学不会这种说法！ - Python程序员的博客 - CSDN博客

<https://blog.csdn.net/qq_42156420/article/details/81169554>

#### 运行机制：

* 原理描述：

1. 调用顺序，遵循入栈出栈思想

2. 装饰器实际上是一个闭包函数

@mydeco

def myfunc()....

等同于

myfunc = mydeco(myfunc)

* 运行过程

1. 将函数名myfun(x)作为一个对象传入装饰器函数形参fn，即fn=myfun(x)
2. 装饰器函数中，return返回其内部定义的子函数fx，并使用参数x
3. 子函数内部可以直接return返回fn(n)

示例见:

def mydeco(fn):

def fx(x,y=10):

print('mmp',x,y)

fn(4) #若此处调用myfunc()会导致无穷循环

return fx

@mydeco

def myfunc(x):

print('被装饰函数',x)

myfunc(5)

#结果：

'''

mmp 5 10

被装饰函数 4

'''

探索：

def mydeco1(function):

print('阶段一：程序读取mydeco1（1）')

def func(x, y=10):

print('--> 开始mydeco1(myfunc(5))')

function(x)

print('mydeco1，over, x=%d,y=%d' % (x, y))

print('阶段一：程序读取mydeco1（2）')

return func

############# 结 果 #############

'''

--------------------------

阶段一：程序读取mydeco1（1）

阶段一：程序读取mydeco1（2）

阶段一：程序读取mydeco2（1）

阶段一：程序读取mydeco2（2）

-------------

开始执行myfunc(5)

--> 开始mydeco2(myfunc(5))

--> 开始mydeco1(myfunc(5))

----> 运行myfunc(5), x=5

mydeco1，over, x=5,y=10

mydeco2，over, x=5

'''

def mydeco2(function):

print('阶段一：程序读取mydeco2（1）')

def func(x):

print('--> 开始mydeco2(myfunc(5))')

function(x)

print('mydeco2，over, x=%d' % (x,))

print('阶段一：程序读取mydeco2（2）')

return func

print('--------------------------')

@mydeco2 # 3,mydeco1( mydeco2( myfunc(5) )

@mydeco1 # 2, mydeco2( myfunc(5) ) )

def myfunc(x): # 1, myfunc(5)

print('----> 运行myfunc(5), x=%d' % x)

print('-------------\n开始执行myfunc(5)')

myfunc(5)

#### 应用实例：

# 此示例示意再加一个装饰器用来添加余额变动提醒功能

def send\_message(fn):

# 小姜写了一个装饰器函数用来发送短信 # send\_message(fn=privillage\_check(savemoney("小张", 200)))

def fy(name, x):

mod = fn(name, x) # 先办业务

if mod == "无权限":

print("%s无权限操作" % name)

elif mod:

print('发短信给%s存了%s元, 剩余%s元' % (name, x, L[0]))

else:

print('发短信给%s取了%s元, 剩余%s元' % (name, x, L[0]))

return fy

def privillage\_check(fn):

# 小赵写了一个装饰器函数: privillage\_check(fn=savemoney("小张", 200))

def fx(name, x):

print("\n正在检查权限.....")

if name in {"小张", "小赵"}:

mod = fn(name, x) # 权限通过可以调用相应函数

return mod

else:

return "无权限"

return fx

@send\_message # send\_message(privillage\_check(savemoney("小张", 200)))

@privillage\_check # privillage\_check(savemoney("小张", 200))

# 写一个操作数据的函数(此函数用来示意存钱操作)

def savemoney(name, x): # 魏老师写的函数,用于存钱

print(name, '存钱', x, '元')

L[0] += x

return 1

@send\_message

@privillage\_check

def withdraw(name, x): # 冯老师写的函数，用于取钱

print(name, '取钱', x, '元')

L[0] -= x

return 0

L = [0]

# ---- 以下是调用者小闵写的程序 -------

savemoney("小张", 2000)

withdraw("小张", 1000)

withdraw('小赵', 100)

savemoney('小赵', 400)

withdraw('小李', 500)

withdraw('小杜', 500)

# 运行结果

'''

正在检查权限.....

小张 存钱 2000 元

发短信给小张存了2000元, 剩余2000元

正在检查权限.....

小张 取钱 1000 元

发短信给小张取了1000元, 剩余1000元

正在检查权限.....

小赵 取钱 100 元

发短信给小赵取了100元, 剩余900元

正在检查权限.....

小赵 存钱 400 元

发短信给小赵存了400元, 剩余1300元

正在检查权限.....

小李无权限操作

正在检查权限.....

小杜无权限操作

'''

## 函数的文档字符串

定义：

函数内第一次未赋值给任何变量的字符串是此函数的文档字符串

说明:

1. 文档字符串通常用来说明本函数的功能和使用方法

2. 在交互模式下，输入:help(函数名) 可以查看函数的文档字符串

* 语法:

def 函数名(形参列表):

'''函数的文档字符串'''

函数语句块

* 查看方法：

help(函数名) 用于查看函数的帮助信息

函数的 \_\_doc\_\_ 属性

\_\_doc\_\_ 属性用于记录文档字符串。用来绑定函数的文档字符串

函数的 \_\_name\_\_ 属性

\_\_name\_\_ 用于记录函数的名称。用来绑定函数名(字符串)

示例:

def hello():

'''此函数用来打招呼...

这是函数的文档字符串

'''

pass

>>> help(hello)

函数的定义语法:

@装饰器1

@装饰器2

...

def 函数名(位置形参, \*元组形参(或\*), 命名关键字形参, \*\*字典形参):

'''文档字符串'''

语句块

## 机制研究

函数绑定

### 可变对象思考（函数中的）

* 现象

面试题,思考？

L = [1, 2, 3]

def f(n=0, lst=[]):

lst.append(n)

print(lst)

f(4, L) # 打印结果是什么？·[1, 2, 3, 4]

f(5, L) # 打印结果是什么？ [1, 2, 3, 4, 5]

f(100) [100]

f(200) # 打印结果是什么？为什么？ [100, 200]

如下代码的打印结果是什么？

L = [1, 2, 3]

def f(n=0, lst=None):

if lst is None:

lst = []

lst.append(n)

print(lst)

f(4, L) # 打印结果是什么？·[1, 2, 3, 4]

f(5, L) # 打印结果是什么？ [1, 2, 3, 4, 5]

f(100) # [100]

f(200) # 打印结果是什么？为什么？ [200]

* 本质理解

函数在一开始初始化时，初始形参的变量即被创建和绑定一个对象。

若调用函数不再改变，绑定的对象将在之后的调用处理中也一直是该对象。

用列表进行函数传参ID是否变化？

L是独立的空间，函数内可以直接更改L，内外都相同

示例：

>>> def f(x,L):

... print("ID",id(L))

... L+=[1]

... L.append(x)

...

>>> L

[1, 2, 3]

>>> f(9,L)

ID 21181864

>>> L

[1, 2, 3, 1, 9]

>>> id(L)

21181864

# 数据的迭代与生成

* 小结:

迭代器

用来访问可迭代对象的数据

　　iter(iterable) # 返回迭代器

next(iterator) # 获取可迭代对象的数据

生成器函数

根据需要动态创建一些数据(数据原来不存在，用yield现用现生成)

range()

生成器表达式

## 迭代器 和 生成器

iterator和generator

共同点：

1. 调用一次后，迭代对象和生成对象皆消失

示例：迭代对象消失

>>> r = range(0,5)

>>> it = iter(r)

>>> for i in it:

print(i)

0

1

2

3

4

>>> for i in it:

print(i)

1. 的

### 迭代器iterator

* 问题引入：

问题:

能否用迭代器访问 range(100, 10000) 中的前三个数?

(要求不用for 循环)

实例：

r = range(100, 10000)

it = iter(r)

print("第一个数是:", next(it)) # 100

print("第二个数是:", next(it)) # 101

print("第三个数是:", next(it)) # 102

#### 基本概念

* 概念

迭代器是访问可迭代对象的一种方式。

迭代器是访问可迭代对象的工具(对象)。

迭代器本身也是对象。

* 说明:

迭代器只能向前取值,不能后退

用iter函数可以返回一个可迭代对象的迭代器

* 疑问

for语句是该迭代器的一种封装【？】

#### 迭代器函数:

* iter( )和next( )

iter(iterable) 从可迭代对象中返回一个迭代器,iterable必须是一个能提供迭代器的可迭代对象

next(iterator) 从迭代器iterator中获取下一条记录,如果无法获取下一条记录,则触发StopIteration 异常

注意：next(iterator)，内容必须是迭代器iterator，非可迭代对象iterable。

示例：

L = [2, 3, 5]

it = iter(L) # 让可迭代对象L 提供一个迭代器

next(it) # 2

next(it) # 3

next(it) # 5

next(it) # StopIteration 异常通知

>>> next([1,2,3])

Traceback (most recent call last):

File "<pyshell#7>", line 1, in <module>

next([1,2,3])

TypeError: 'list' object is not an iterator

* reversed() 反向迭代：

reversed(lst)，一般用于for语句中反向迭代

#### 应用实例

不使用for循环的迭代：（用while 语句遍历列表L中的全部元素）

xiyou=[1,2,3,4]

it=iter(xiyou)

while True:

try:

print(next(it))

except:

print('the end')

break

### 生成器generator

问题引入：

时间 vs 空间

计算机内存有限，生成器函数能用计算的时间换取内存的空间。

概念：

生成器是能够动态提供数据的对象,生成器对象也是可迭代对象(实例)

特点：

是一次性的。数据被使用后就被删除

生成器有两种：

生成器函数

生成器表达式

#### 生成器函数

##### 基本概念：

定义

含有 yield 语句的函数是生成器函数，此函数被调用时将返回一个生成器对象

注: yield 翻译为产生（或生成)

返回：

返回一个生成器对象，生成器对象是一个可迭代对象

生成器函数说明:

1. 生成器函数的调用将返回一个生成器对象,生成器对象是可迭代对象

2. 生成器函数调用 return 会触发一个StopIteration异常【？】

语法：

def f():

...

yield 表达式1

yield 表达式2

...

f()

说明:

yield用于def函数中，目的是将此函数作为生成器函数使用

yield用来生成数据，供迭代器 next(it) 函数使用

##### 运行机制：

函数中遇到yield语句时，

1先执行yield之前语句，

2送yield之后元素出来参与各类操作,如list

示例见:

生成器函数示例：

def myyield():

print("即将生成2")

yield 2 # 生成2

print('即将生成3')

yield 3 # 生成3

print('即将生成5')

yield 5

print("myield函数返回")

print('\*----------------------') # 此处说明函数返回可迭代对象

myyield() # 无任何执行与返回

print(myyield()) # 返回迭代对象地址

print('\*----------------------')

# 说明机制：遇到yield语句时，1>执行之前语句，2>送yield元素出来

for x in myyield():

print(x)

print('\*\*---------------------') # 此处说明迭代对象只能使用一次

it = myyield() # it 绑定生成器

print(list(it)) # 可见迭代对象存在

print('\*\*\*--------------------')

print(list(it)) # 可见迭代对象已删除

结果：

"""

<generator object myyield at 0x000000000282FB48>

\*----------------------

即将生成2

2

即将生成3

3

即将生成5

5

myield函数返回

\*\*---------------------

即将生成2

即将生成3

即将生成5

myield函数返回

[2, 3, 5]

\*\*\*--------------------

[]

"""

##### 应用

使用yield生成序列

def myeven(s, e):

while s < e:

if s % 2 == 0:

yield s

s += 1

for i in myeven(10, 20):

print(i)

#### 生成器表达式

##### 基本概念

* 语法:

(表达式 for 变量　in 可迭代对象 [if 真值表达式])

说明：if 子句可以省略

* 作用:

用推导式的形式生成一个新的生成器

　　示例:

gen = (x\*\*2 for x in range(1, 4))

it = iter(gen)

next(it)　　# 1

next(it)　　# 4

next(it)　　# 9

next(it)　　# StopIteration

* 优点：

不占用内存空间

##### 列表推导式 和　生成表达式的区别:

生成表达式**不存储数据**，所有数据都是实时获取的。

而列表推导式相反，储存数据生成新列表。

实例：

改变原列表分别对结果的影响

# 看程序执行结果有什么不同:

# 1) 程序1

L = [2, 3, 5, 7]

lst = [x for x in L]

it = iter(lst)

print(next(it)) # 2

L[1] = 30

print(next(it)) # 3

#2) 程序2

L = [2, 3, 5, 7]

lst = (x for x in L)

it = iter(lst)

print(next(it)) # 2

L[1] = 30

print(next(it)) # 30

## 迭代工具函数:

迭代工具函数的作用是：生成一个个性化的可迭代对象

### zip函数

* 格式：

zip(iter1 [,iter2, iter3,...])

* 返回：

元组包装的对象enumerate。

返回一个zip对象此对象用于生成一个元组，此元组的个数由最小的可迭代对象决定，

元组由对象位置的iter1，iter组成。

* 示例:

本质探索：

zip生成的是一个个元组组成的可迭代对象。

numbers = [10086, 10000, 10010, 95588]

names = ['中国移动', '中国电信', '中国联通']

for a in zip(numbers, names):

print(a)

"""

(10086, '中国移动')

(10000, '中国电信')

(10010, '中国联通')

"""

* 应用：

常用操作：相当于序列赋值给n,a

numbers = [10086, 10000, 10010, 95588]

names = ['中国移动', '中国电信', '中国联通']

for n, a in zip(numbers, names):

print(a, '的客服号码是:', n)

"""

中国移动 的客服号码是: 10086

中国电信 的客服号码是: 10000

中国联通 的客服号码是: 10010

"""

* 应用

# 以下用zip函数生成一个字典

numbers = [10086, 10000, 10010, 95588]

names = ['中国移动', '中国电信', '中国联通']

d = dict(zip(names, numbers))

* 运行机制

# zip函数的实现机制见:

03\_myzip.py

### enumerate函数(枚举函数)

返回：

元组包装的对象enumerate。

每个元素前添加索引，并包装为元组(索引,元素)，组合为可迭代对象enumerate

作用：

从指定数（默认为0）开始向后自动生成序号

格式：

enumerate(iterable[,start=0])

生成带索引的枚举对象，返回迭代类型为索引-值对(index,value)对, 默认索引从零开始,也可以使用start绑定

示例:

names = ['中国移动', '中国电信', '中国联通']

for x in enumerate(names):

print(x) # (0, '中国移动'), (1, '中国电信')...

for x in enumerate(names, start=100): # 可省略“start=”

print(x) # (100, '中国移动'), (101, '中国电信'), (102, '中国联通')

# 错误及异常处理

问题描述：

程序语句正常，但是却因为输入值错误无法运行

若要继续运行，如何解决？

异常处理意义：

传递与处理进程异常信息，类似于公司管理。

## 基本概念

### 错误与异常比较

* 错误

错误是指由于逻辑或语法等导致一个程序无法正常执行的问题

特点：

有些错误是无法预知的

* 异常

异常是程序出错时标识的一种状态。

当异常发生时，程序不会再向下执行，而转去调用此函数的地方待处理此错误并恢复为正常状态。

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现。

### 作用

* 作用:

用作信号,通知上层调用者有错误需要处理

### 错误类型

|  |  |  |
| --- | --- | --- |
| 分类 | 错误类型 | 说明 |
| 数值 | ZeroDivisionError | 除(或取模)零 (所有数据类型) |
| 函数 | ValueError | 传入无效的参数 |
|  | AssertionError | 断言语句失败 |
|  | StopIteration | 迭代器没有更多的值 |
|  | IndexError | 序列中没有此索引(index) |
|  | IndentationError | 缩进错误 |
| 模块 | ImportError | 导入模块/对象失败 |
| 文件 | OSError | 输入/输出操作失败 |
|  | NameError | 未声明/初始化对象 (没有属性) |
|  | AttributeError | 对象没有这个属性 |
|  |  |  |
|  | BaseException | 所有异常的基类 |
|  | SystemExit | 解释器请求退出 |
|  | KeyboardInterrupt | 用户中断执行(通常是输入^C) |
|  | Exception | 常规错误的基类 |
|  | GeneratorExit | 生成器(generator)发生异常来通知退出 |
|  | StandardError | 所有的内建标准异常的基类 |
|  | ArithmeticError | 所有数值计算错误的基类 |
|  | FloatingPointError | 浮点计算错误 |
|  | OverflowError | 数值运算超出最大限制 |
|  | EOFError | 没有内建输入,到达EOF 标记 |
|  | EnvironmentError | 操作系统错误的基类 |
|  | OSError | 操作系统错误 |
|  | WindowsError | 系统调用失败 |
|  | LookupError | 无效数据查询的基类 |
|  | KeyError | 映射中没有这个键 |
|  | MemoryError | 内存溢出错误(对于Python 解释器不是致命的) |
|  | UnboundLocalError | 访问未初始化的本地变量 |
|  | ReferenceError | 弱引用(Weak reference)试图访问已经垃圾回收了的对象 |
|  | RuntimeError | 一般的运行时错误 |
|  | NotImplementedError | 尚未实现的方法 |
|  | SyntaxError Python | 语法错误 |
|  | TabError | Tab 和空格混用 |
|  | SystemError | 一般的解释器系统错误 |
|  | TypeError | 对类型无效的操作 |
|  | UnicodeError | Unicode 相关的错误 |
|  | UnicodeDecodeError | Unicode 解码时的错误 |
|  | UnicodeEncodeError | Unicode 编码时错误 |
|  | UnicodeTranslateError | Unicode 转换时错误 |

|  |  |
| --- | --- |
| 以下为警告类型 |  |
| Warning | 警告的基类 |
| DeprecationWarning | 关于被弃用的特征的警告 |
| FutureWarning | 关于构造将来语义会有改变的警告 |
| OverflowWarning | 旧的关于自动提升为长整型(long)的警告 |
| PendingDeprecationWarning | 关于特性将会被废弃的警告 |
| RuntimeWarning | 可疑的运行时行为(runtime behavior)的警告 |
| SyntaxWarning | 可疑的语法的警告 |
| UserWarning | 用户代码生成的警告 |

## 语句

* 小结:

语句:

try-except

捕获异常，偿试接收异常通知

(收通知,偿试处理错误，将状态改为正常状态)

try-finally

执行一定要执行的语句

raise

发送异常通知，将程序转为异常状态（进入异常流程)

(发错误通知，将状态改为异常状态)

assert

根据条件来触发AssertionError类型的异常

with 语句(以后再学)

### try语句

常规语法：try、except、else、finally

* 两种语法：

try-except语句

try-finally语句

* try-except语句

作用：

偿试捕获异常，将程序转为正常状态并继续执行

注:try-finally语句不会改变程序的(正常/异常)状态

* try-finally语句

作用:

通常用try-finally语句来做触发异常时必须要处理的事情,无论异常是否发生，finally子句都会被执行

#### 语法：

try:

try:

可能触发异常的语句

except 错误类型1 [as 变量1]:

异常处理语句1

except 错误类型2 [as 变量2]:

异常处理语句2

except (错误类型3, 错误类型4, ...) [as 变量3]:

异常处理语句3

...

except:

异常处理语句other

else:

末发生异常时执行的语句

finally:

一定要执行的最终语句

except 错误类型1:

异常处理语句1

...

* 语法说明:

as 子句是用于绑定错误对象的变量，可以省略不写。

except子句可以有一个或多个，但至少要有一个except或finally

else子句最多只能有一个，也可以省略不写

finally子句最多只能有一个，也可以省略不写

* 变量绑定及异常自定义说明：

as可以用于绑定变量返回错误标识。

try:

int('a')

except ValueError as e: **# as可以用于绑定变量返回错误标识。**

print("错误的值是:", e)

'''

错误的值是: invalid literal for int() with base 10: 'a'

'''

练习:

与一个函数 get\_score() 来获取学生的成绩(0~100), 如果输入出现异常，则此函数返回0,否则返回用户输入的成绩

def get\_score():

....

score = get\_score()

print("学生的成绩是:", score)

#### 运行机制：

**若无错：**

执行由try --> else --> finally

**遇错立即跳出，若在try中，则**

执行由try --> except --> finally

示例：

def make\_exception():

# err = ValueError # 可绑定变量

# raise err

raise ValueError("This is 我自己定义的一个错误")

print('该条语句会被跳过吗？yes1')

try:

make\_exception()

print('该条语句会被跳过吗？yes2')

except ValueError as e:

print("try里出现了值错误通知，已捕获!!!")

print("接收的异常通知是: ", e)

else:

print('else语句会被跳过吗？yes3')

finally:

print('该条语句会被跳过吗？No')

'''

try里出现了值错误通知，已捕获!!!

接收的异常通知是: This is 我自己定义的一个错误

该条语句会被跳过吗？No

'''

**错误会逐层向外传递**

**def** fry\_egg():

**try**:

print(**'打开天燃气....'**)

**try**:

count = int(input(**"请输入鸡蛋个数: "**))

print(**"共煎了"**, count, **'个鸡蛋'**)

**finally**:

print(**'关闭天燃气'**)

**except** ValueError:

**pass**

fry\_egg()

print("程序正常执行")

练习:

1. 一个球100米高空落下,每次落下后反弹高度是原高度的一半,再落下,

写程序

1) 算出皮球在第10次落地后反弹高度是多少,

2) 打印出球共经过了多少米的路程

2. 分解质因数,输入一个正整数，分解质因数:

如:

输入: 90

打印:

90=2\*3\*3\*5

(质因数是指最小能被原数整数的素数(不包括1))

### raise 语句

作用:

触发一个错误,让程序进入异常状态

语法:

raise 异常类型

或

raise 异常对象

示例见:

11\_raise.py

12\_raise\_get\_age.py

示例：

def make\_exception():

# raise ValueError

err = ValueError("这是我自己定义的一个错误")

raise err

print('该条语句会被跳过吗？yes')

try:

make\_exception()

print('该条语句会被跳过吗？yes')

except ValueError as e:

print("try里出现了值错误通知，已捕获!!!")

print("接收的异常通知是: ", e)

### assert 语句(断言语句)

语法:

assert 真值表达式, 错误数据(通常是字符串)

作用:

当真值表达式为False时,用错误数据创建一个 AssertionError 类型的错误,并进入异常状态

【用于抛出不满足正常条件的异常】

【检验输入的数据是否合格，后面跟满足的条件，不满足时抛出异常】

类似于:

if 真值表达式 == False:

raise AssertionError(错误数据)

示例：

# 此示例示意assert语句的用法

def get\_score():

s = int(input("请输入学生成绩: "))

# 用assert语句来断言s是否在 0~100之间

assert 0 <= s <= 100, "用户输入的整数不在0~100之间"

return s

try:

score = get\_score()

print("学生成绩为:", score)

except ValueError:

print("用户输入的成绩无法转化为整数")

except AssertionError as err:

print("发生了断言错误，原因是:", err)

## 应用

为什么要用异常处理机制？

在程序调用层数较深时,向主调函数传递错误信息需要用return语句层层传递比较麻烦,所以用异常处理机制

说明示例:

# 以建房子为例

def f1():

print("开始建房子打地基")

err = ValueError("打地基挖出古董")

raise err

print("完成打地基工作")

return "地基完成"

def f2():

print("开始建设地上部分")

# err = ValueError("规划要建高压线")

# return err

print("地上部分建完..")

return "地上完成"

def f3():

# 建地基

r1 = f1()

# 建地上部分

r2 = f2()

return r1 + r2

def built\_house():

'''接项目的人'''

return f3()

try:

h = built\_house() # 建房子的函数,此函数应当返回一个房子对象

print(h)

except ValueError as e:

print("错误原因是:", e)

print('改建博物管')

### 灵活处理

示例：

方法一：

def getnum(x):

try:

s = x

if s not in range(1, 141):

raise ValueError('你输入的数值%d不在范围内'%s)

print(s)

except ValueError as err:

print('触发错误，',err)

getnum(150)

触发错误， 你输入的数值150不在范围内

方法二：(断言)

# 文件file及编码

问题：

文件是什么，存储方式是什么？

编码是什么？为什么会有不同编码？

utf-8和gbk的文件区别是什么？

猜测：文件的头几个编码不同造成这些文件差异？

经验证并非如此，他们文件本质并无差异，都是字节串文件，差异在于文本模式打开后的汉字编码(翻译)规则。

字节串和二进制文件有什么关系？

字节串是二进制文件的存储方式。

为什么Python在Linux下能正常open并read文件，而把文件放在Windows不能？

系统的编码不同win --> gbk，linux --> utf-8。

linux的默认

## 基本概念

### 文件file

什么是文件？各种格式之间有什么区别吗？

#### 文件基本概念

我的用一句话概括：

文件是存储字节串的容器，编码和后缀名(文件格式)只是它的翻译规则。

特点：

文件是数据存储的单位

文件通常用来长期存储数据

文件中的数据是以字节为单位进行顺序存储的

#### 文件储存的信息本质

引入：

# 观察，下列数字都为十进制97

>>> 97

97

>>> 0b1100001

97

>>> 0x61

97

>>> b'\x61'

b'a'

文件的存储本质：存储字节码，被编码规则转码显示于屏幕

如文件中的

原码：

6162 6331 3233 e4b8 ade6 9687

通过utf-8转码：

abc123中文

转码验证

>>> b'\x61\x62\x63\x31\x32\x33 \xe4\xb8\xad \xe6\x96\x87'.decode('utf-8')

'abc123 中 文'

文件在外部存储都是字节串存储。

Python内部是Unicode，需要转换出去再存储。

如何理解？

文件的外部存储本身都没有任何编码信息，都是以字节串存储的。

这些字节串信息用不同的编码方法能读出不同的信息。

### 编码理解

那么，何为字节串，何为Unicode？ascII又是什么，与utf8等又是什么联系？

编码，其实相当于一套翻译规则。

#### 文件与MD5值

文件的MD5值实际上是对文件的字节码进行MD5算法加密所获得的唯一码

示例：

文件内容为：

hello

Linux下通过命令：

$md5sum 2.txt

5d41402abc4b2a76b9719d911017c592 2.txt

Python3下打开2.txt的内容读取，发现2者MD5一致

>>> import hashlib

>>> objMD5 = hashlib.md5()

>>> with open('2.txt','rb') as fi:

data=fi.read()

objMD5.update(data)

print(objMD5.hexdigest())

### 不同系统平台的编码问题

#### 换行符

* 各种操作系统的换行符:

Linux换行符: '\n'

Window换行符: '\r\n'

旧的Macintosh换行符: '\r'

新的Mac Os 换行符: '\n'

说明:

在文件文件模式下，各操作系统的换行符在读入python内部时转换为字符'\n'

这个知识有什么用呢？

因为这个你便能理解为什么有时候，Linux统计字符末尾会多1，为什么有时候文本文件编辑好后打开换行符会消失连成一片。

例1：

Linux下面创建的文件在Windows下面的换行不被读取。会连接成一行

因为其存储'\n'字节符不同。

例2：

Windows中创建的文件在Linux下读取出来能自动换行

机制是：

\n一般会操作系统被翻译成"行的结束"，即LF(Line-Feed)

\r会被翻译成"回车"，即CR(Cariage-Return)

对于文本文件的新行，在UNIX上，一般用\n(LF)来表示，Mac上用\r(CR)来表示，

Windows上是用\n\r(CR-LF)来表示。

#### 汉字编码

问题:

“十个汉字占多少个字节”这句话存储的文件占多少个字节？

不同编码，不同结果。

utf-8 32

gbk(ANSI) 20

只讲两种:

国标系列:

GB18030(二字节或四字节编码, 共27533个汉字)

GBK(二字节编码, 共21003个汉字)

GB2312(二字节编码, 共个6763汉字)

(Windows 常用)

国际标准:

UNICODE32(UNICODE16) < ---> UTF - 8

(Linux, Mac OS X, IOS, Android等常用)

有趣的是，这套标准里，3/4都是中亚文，

【问题：ASCii和这些编码是什么关系？【？】】

#### 更改文件编码的方法

Windows下，记事本 🡪 文件 🡪 另存为 🡪 选择修改编码

#### 平台兼容问题解决方案

##### 问题描述

在Windows下打开utf-8编码文件出现报错：

>>> with open('mod-run.sh') as fi:

... data = fi.read()

...

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

UnicodeDecodeError: 'gbk' codec can't decode byte 0xb4 in position 183: illegal multibyte sequence

##### 探索解析

* Python脚本运行的平台兼容性问题：

windows下正常读取“gbk编码(ANSI)”，却不能正常读取“utf-8编码”文件

linux下不能正常读取“gbk编码(ANSI)”，却能正常读取“utf-8编码”文件

都返回UnicodeDecodeError错误

* 探索实例：

文本内容：

abc123中文

ABC英文

hello world

另存为myfile1.txt（“utf-8编码”），myfile2.txt（“gbk编码(ANSI)”）

Windows下运行：

打开文件1报错：

f = open('myfile1.txt', 'rt')

print("打开文件成功!")

s = f.read()

print("读取文件成功!")

print("文件中的内容是:", s)

报错：

UnicodeDecodeError: 'gbk' codec can't decode byte 0xad in position 8: illegal multibyte sequence

打开文件2正常读取：

f = open('myfile2.txt', 'rt')

print("打开文件成功!")

s = f.read()

print("读取文件成功!")

print("文件中的内容是:", s)

打开文件成功!

读取文件成功!

文件中的内容是: abc123中文

ABC英文

hello world

linux下运行结果则相反，实例不列举

##### 解决方案

* 1 二进制读取
* 2 转码

示例：解码打开

f = open('hanzi\_gbk.txt', 'rb')

b = f.read() # 读取字节串

s = b.decode('gbk') # 将gbk字节串解码为UNICODE字符串

# s = f.read()

print(s)

f.close()

##### 操作模式（文件的读取方法）

文本文件操作的两种模式:

'b' 二进制模式

't' 文本模式

文本模式(textmode)和二进制模式(binarymode)有什么区别?

流可以分为两种类型：文本流和二进制流。

文本流是解释性的，最长可达255个字符。

二进制流是非解释性的，一次处理一个字符，并且不转换字符。

Unicode/UTF/UCS格式的文件，必须用二进制方式打开和读写。

* 二进制模式的文件操作

默认的文件中存储的都是以字节为单位的数据，通常有人为规则的格式，需要以字节为单位进行读写

以十六进制方式查看文件内容的命令:

$ xxd 文件名

示例见:

file\_write\_binary.py【？】

file\_read\_binary.py

## 内存编码和文件编码

### python 的内部(内存)编码

't' 文本模式(默认)

1. 默认文件中存储的数据为字符数据，以行为单位分隔，在 python 内部统一用'\n'作为换行符进行分隔

2. 对文本文件的读写需要用字符串(str)进行读取和写入数据

说明:

python3的字符串内部都是用UNICODE来存储字符的

### python文件的编码注释

python 编码(encode) 字符串:

'gb2312'

'gbk'

'gb18030'

'utf-8'

'ascii'

（在python 源文件第一行或第二行写入如下内容是告诉解释执行器此文件的编码类型是什么）

如:

# -\*- coding: gbk -\*-

# 设置源文件编码格式为gbk

或

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

# 设置源文件编码格式为utf-8

示例见:

hello\_gbk.py

### 字节串 bytes

* 引入：

回顾问题:

之前学的容器类型:

str, list, tuple, dict, set, frozenset

序列:

list, str, tuple, bytes, bytearray

* 意义：

所有字符转码为字节作为通信传输。

存储以字节为单位的数据

* 总结

python:字符串转换成字节的三种方式

str='zifuchuang'

第一种 b'zifuchuang'

第二种bytes('zifuchuang',encoding='utf-8')

第三种('zifuchuang').encode('utf-8')

#### 概念

何为字节串？

区别

B...byte

b...bit

说明:

字节串是不可变的字节序列【？】

字节是0~255之间的整数【？】

* 创建字面值:

创建空字节串的字面值

b = b'' b 绑定空字节串

b = b"" b 绑定空字节串

b = b'''''' b 绑定空字节串

b = b"""""" b 绑定空字节串

b = B""

创建非空字节串的字面值:

b'ABCD'

b'\x41\x42'

b'hello tarena'

说明：

字面值只能是字母、数字、英文符号【对否，待验证【？】】

#### 字节串的构造函数 bytes

格式：

bytes() 生成一个空的字节串 等同于 b''

bytes(整型可迭代对象) 用可迭代对象初始化一个字节串

bytes(整数n) 生成n个值为零的字节串

bytes(字符串, encoding='utf-8') 用字符串的转换编码生成一个字节串

总结：整形数int，整型数int构成的可迭代对象，'字符串' +'utf-8'

说明：中文等会被转换为十六进制码

示例：

>>> bytes(1) # b'\x00'解码是空格

b'\x00'

>>> bytes(2)

b'\x00\x00'

>>> bytes(10)

b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> bytes([1,2])

b'\x01\x02'

>>> bytes(range(1,3))

b'\x01\x02'

>>> bytes('a')

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

bytes('a')

TypeError: string argument without an encoding

>>> bytes('a',encoding='utf-8')

b'a'

>>> bytes('aa',encoding='utf-8')

b'aa'

>>> bytes('aaaa',encoding='utf-8')

b'aaaa'

>>> bytes('你好吗',encoding='utf-8')

b'\xe4\xbd\xa0\xe5\xa5\xbd\xe5\x90\x97'

>>> len(bytes('你好吗',encoding='utf-8'))

9

>>> bytes('你',encoding='utf-8')

b'\xe4\xbd\xa0'

#### bytes、str的区别与转换

区别本质:

bytes

存储字节(0-255)

【可被传输【？】】

str

存储Unicode字符(0-65535)

转换：

b = s.encode('utf-8')

编码(encode)

str --------------> bytes

s = b.decode('utf-8')

解码(decode)

bytes-------------> str

示例：

>>> print('妈耶'.encode('utf-8'))

b'\xe5\xa6\x88\xe8\x80\xb6'

>>> print(b'\xe5\xa6\x88\xe8\x80\xb6'.decode('utf-8'))

妈耶

#### 字节串的运算

+ += \* \*=

< <= > >= == !=

in / not in

索引/切片

示例:

b = b'abc' + b'123' # b=b'abc123'

b += b'ABC' # b=b'abc123ABC'

b'ABD' > b'ABC' # True

b = b'ABCD'

65 in b # True

b'A' in b # True

#### 用于序列函数:

len(x)

max(x)

min(x)

sum(x) # 字节串存储的是数值。可以

all(x)

any(x)

#### 不同编码字节串长度区别

>>> '给'.encode('gbk')

b'\xb8\xf8'

>>> '给'.encode('utf8')

b'\xe7\xbb\x99'

>>> '给'.encode() # 不与平台相关，默认'utf8'。

b'\xe7\xbb\x99'

>>> len(b'\xe7\xbb\x99')

3

>>> len(b'\xe7\xbb\x99'**.decode('utf8')**)

1

### 字节数组 bytearray

可变的字节序列

#### 创建字节数组的构造函数:

bytearray() 创建空的字节数组

bytearray(整数)

bytearray(整型可迭代对象)

bytearray(字符串,encoding='utf-8')

注: 以上参数等同于字节串

#### 字节数组的运算:

+ += \* \*=

比较运算: < <= > >= == !=

in / not in

索引　/ 切片(字节数组支持索引和切片赋值，规则与列表相同)

>>> ba=bytearray(b'ABCDEF')

>>> ba[::2]=[48,49,50] **# 内容必须是整数，因为代表编码值**

>>> ba

bytearray(b'0B1D2F')

#### bytearray的方法:

B.clear() 清空字节数组

B.append(n) 追加一个字节(n为0-255的整数)

B.remove(value) 删除第一个出现的字节，如果没有出现，则产生ValueError错误

B.reverse() 字节的顺序进行反转

B.decode(encoding='utf-8') # 解码

B.find(sub[, start[, end]]) 查找

字节数组的方法详见: help(bytearray)

## 文件的操作流程

1. 打开文件

2. 读/写文件

3. 关闭文件

任何的操作系统，一个应用程序同时打开文件的数量有最大数限制

### 文件的操作方法

#### 打开open( )（创建文件对象）

* 打开方式：

b

t

* 打开方法：

函数 open

open(file, mode='rt') 注：mode的rt是缺省参数。

用于打开一个文件，返回此文件对应的文件流对象，如果打开失败，则会触发OSError错误！

open返回的文件流对象是可迭代对象

文件的迭代读取:

示例:

f = open('myfile.txt')

for line in f:

print(line)

#### 读取read( )

读取方法：（按光标读取之后的内容）

file.read()

file.read(n) # n代表读取第几位

file.readline() # 逐次取出文件一行，空行时空字符串''，读取会带最后的换行符“\n”

file.readlines() # 读取所有航并以“行”为分隔符，生成返回列表

f.read()的返回类型:

1. 对于文本模式('t')打开的文件，返回字符串(str)

2. 对于二进制模式('b')打开的文件，返回字节串(bytes)

注：

file.readline() 读取的内容等同于 for s in file: print(s) 的第一句

但不同之处在于，它可以精确控制光标

#### 写入write( )

文本文件的写操作:

写文件模式有:

'w'

'x'

'a'

写入方法：

f.write(x) 对于文本模式,x必须为字符串; 对于二进制模式,x必须为字节串

写入的末尾不会带最后的换行符“\n”

file\_write\_text.py

* 写入文件的编码问题探索

经探索：

文本模式写入：

Python在Windows下默认写入的txt文件编码为gbk

Python在 Linux 下默认写入的txt文件编码为utf-8

二进制模式下：

使用二进制模式写入的文件，能保持其文件原有的编码。

示例：

myfile2.txt --编码为gbk

abc123中文

ABC英文

hello world

f = open('myfile2.txt', 'rb')

f2=open('myfile\_cp.txt','wb')

print("打开文件成功!")

s = f.read()

print("读取文件成功!")

print("文件中的内容是:", s)

f2.write(s)

print("写入文件成功!")

结果：写入的新文件myfile2.txt --编码也为gbk

#### 关闭

F.close()

关闭，释放系统资源

示例见:

01\_fileopen.py

## python 文件方法总结

### 常用方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 返回 | 传参 | 说明 |
| F.close() |  |  | 关闭文件(关闭后文件不能再读写会发生ValueError错误) |
| F.readline() |  |  | 读取一行数据, 如果到达文件尾则返回空行 |
| F.readlines(max\_chars=-1) |  |  | 返回每行字符串的列表,max\_chars为最大字符(或字节)数 |
| F.writelines(lines) |  | 列表 | 每行字符串的列表 |
| F.flush() |  |  | 把写入文件对象的缓存内容写入到磁盘 |
| F.read(size = -1) |  |  | 从一个文件流中最多读取size个字符，并将光标从文件流后移 |
| F.write(text) |  |  | 写一个字符串到文件流中，返回写入的字符数 |

|  |  |  |  |
| --- | --- | --- | --- |
| 二进制文件操作方法： |  |  |  |
| F.tell() |  |  | 返回当前文件流的绝对位置，即读到第几位了，读取的光标位置 |
| F.seek(offset, whence=0) |  |  | 改变数据流的位置，返回新的绝对位置（简言之，使光标归位） |
| F.readable() |  |  | 判断这个文件是否可读,可读返回True,否则返回False |
| F.writable() |  |  | 判断这个文件是否可写,可写返回True,否则返回False |
| F.seekable() |  |  | 返回这个文件对象是否支持随机定位 |
| F.truncate(pos = None) |  |  | 剪掉 自pos位置之后的数据，返回新的文件长度(字节为单位) |

* F.tell()返回文件当前位置。

F.tell() 方法:

作用:

返回当前文件读写位置

* F.seek()设置文件当前位置

F.seek(offset[, whence])

应用

file.seek(0) # 光标归位

F.seek(偏移量, 相对位置)

偏移量:

大于0的数代表向文件尾方向移动

小于0代表向文件头方向移动

相对位置:

0 代表从文件头开始偏移

1 代表从当前位置开始偏移

2 代表从文件尾开始偏移

作用:

改变当前文件的读写位置

### 模式(mode)字符的含义

|  |  |
| --- | --- |
| 字符 | 含义 |
| 'r' | 以只读方式打开(默认) |
| 'w' | 以只写方式打开，删除原有文件内容(如果文件不存在，则创建该文件并以只写方式打开) |
| 'x' | 创建一个新文件, 并以写模式打开这个文件,如果文件存在则会产生"FileExistsError"错误 |
| 'a' | 以只写文件打开一个文件，如果有原文件则追加到文件末尾 |
| 'b' | 用二进制模式打开 |
| 't' | 文本文件模式打开 (默认) |
| '+' | 为更新内容打开一个磁盘文件 (可读可写) |

缺省参数模式是 'rt'

'w+b' 可以实现二进制随机读写，当打开文件时，文件内容将被清零

'r+b' 以二进制读和更新模式打开文件,打开文件时不会清空文件内容

'r+' 以文本模式读和更新模式打开文件,打开文件时不会清空文件内容

### 应用实例

* 示例1：F.read()读取文本内容

#data.bin

e4b8 ad00 00

#file\_read\_binary.py

file = open('data.bin', 'rb')

b = file.read(1) # 读取一个字节

print("第一个字节是:", b)

b = file.read() # 读取所有的字节

print('其它所有字节是: ', b)

file.close()

结果：

第一个字节是: b'\xe4'

其它所有字节是: b'\xb8\xad\x00\x00'

* 示例2：while + readline()读取文本内容

# 此示例示意以文本文件模式读取myfile.txt中的内容

try:

f = open("myfile.txt")

print('打开文件成功!')

# 读取文件内容

while True:

s = f.readline()

if s == '':

print('已经读取到了文件末尾')

break

print("读到这一行是:", s)

f.close()

except OSError:

print("打开文件失败")

* 示例3：F.readlines()用法

## with语句

简化try open文件

注意事项：

1、with语句命名空间

with open(file) as fi:

test = 'hello'

print(test) # 正常显示

## 原理理解

### 读取的原理

问题引入：

for迭代文件对象，与next()方法读取下一行，以及readline()读取下一行都有区别吗？

【无区别，原理本质都是通过光标读取下一行，在深入一点为指针】

四字总结：

光标移位

【无论是单独语句，还是表达式中（读取，条件判断等）只要往下面走，光标都会移位！！】

* 示例：

文件：t.txt

1

2

3

4

5

6

脚本：t.py

# t.py

fi = open('t.txt')

for i in fi:

print('--'+i)

if i.startswith('3'):

next(fi)

data=fi.readline()

print(data)

运行结果：

--1

--2

--3

5

--6

另外，还发现一些问题：

使用for和使用next()迭代文件对象，都会直接让光标直接移至文件末尾，因为无法在文件读取中获取其当前光标位置。

而使用F.readline()则会让光标按行移动。

### 二进制模式下写入文件的理解

>>> f = open('t.txt','wb')

>>> f.write(b'asdf嗨，你好')

File "<stdin>", line 1

SyntaxError: bytes can only contain ASCII literal characters.

# 面向对象

——面向对象编程: Object - Oriented Programing

研究意义：打包分类，耦合性低，思维明确；可以少一点全局变量；

研究方法A.\_\_dict\_\_，dic(A)

问题集锦：

\_\_new\_\_是什么来头？？构造函数？与\_\_init\_\_关系是什么？

## 类的简介

### 基本概念

#### 基本名词解释

* **类(Class):**用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
* **方法：**类中定义的函数。
* **类变量：**类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
* **数据成员：**类变量或者实例变量用于处理类及其实例对象的相关的数据。
* **方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重写。
* **实例变量：**定义在方法中的变量，只作用于当前实例的类。
* **继承：**即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）。
* **实例化：**创建一个类的实例，类的具体对象。
* **对象：**通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

#### 面向对象简介

* 什么是对象:

对象是指现实中的物体或实体

* 什么是面向对象

把一切看成对象(实例), 用各种对象之间的关系来描述事务

* 对象都有什么特征:

对象有很多属性(名词)

姓名, 性别, 年龄, ...

对象有很多行为(动作，动词)

学习，吃饭，睡觉，工作, ....

示意:

车(类) - -----> > BYD E6(京A.88888)(实例，对象)

\

\-----> > BWM X5(京B.66666)(对象)

狗(类) - -----> > 京巴(户籍号: 000001)

\

\-----> > 导盲犬(户籍号: 000002)

int(类) - -----> > 100 (对象, 实例)

\

\-----> > 200 (对象, 实例)

#### 类概念探索

* 什么是类？什么是实例？

拥有相同属性和行为的对象分为一组，即为一个类

类是用来描述对象的工具，用类可以创建此类的对象(实例)

* 类属性？实例属性？
* 类的本质和实例的本质区别和联系是什么？

### 类的创建

语法：

class 语句

语法:

class 类名(继承列表):

''' 类文档字符串'''

类变量定义

实例方法定义

类方法定义(@classmethod)

静态方法定义(@staticmethod)

作用:

创建一个类

用于描述对象的行为和属性

用于创建此类的一个或多个同类对象(实例)

说明:

继承列表可以省略，省略继承列表表示类继承自object

类名必须为标识符

类名实质上是变量，它绑定一个类

示例见:

class.py

### 类的基本结构

#### 构造函数

构造函数调用表达式：

类名([创建传参列表])

如 a = A()

作用:

创建这个类的实例对象，并返回此实例对象的引用关系

实例说明:

1. 实例有自己的作用域和名字空间，可以为该实例添加实例变量(也叫属性)

2. 实例可以调用类方法和实例方法

3. 实例可以访问类变量和实例变量

#### 类变量（类属性）

* 类变量

类变量有一个特点，在类中定义的每一个类不管在任何一个位置（比如某个类函数，某个循环嵌套中），最终都会成为类空间的“全局变量”！实例验证：

* 实例属性 attribute(也叫实例变量)

每个实例可以用自己的变量，称为实例变量(也叫属性)

使用语法:

实例.属性名

属性的赋值规则:

1. 首次为属性赋值则创建此属性

2. 再次为属性赋值则改变属性的绑定关系

作用:

记录每个对象自身的数据

示例见:

atribute.py

* 删除属性方法：del 语句

语法:

del 对象.属性名

示例:

class Dog:

pass

dog1 = Dog()

dog1.color = '白色' # 添加属性

del dog1.color 删除属性

* del 语句总结:

1) 删除变量 del a

2) 删除列表中的元素 del L[0]

3) 删除字典中的键 del d['name']

4) 删除对象的属性 del dog1.color

#### 实例方法(method):

语法:

class 类名(继承列表):

def 实例方法(self, 形参1, 形参2, ...):

'''方法的文档字符串'''

语句块

作用:

用于描述一个对象的行为，让此类型的全部对象都拥有相同的行为

说明:

实例方法的实质是函数，是定义在类内的函数

实例方法至少有一个形参，第一个形参代表调用这个方法的实例，一般命名为'self'

实例方法的调用语法:

实例.实例方法名(调用传参)

或

类名.实例方法名(实例, 调用传参)

示例1:

instance\_method.py

#### 初始化方法:

作用:

对新创建的对象添加属性等初始化操作

语法格式:

class 类名(继承列表):

def \_\_init\_\_(self[, 形参列表])

语句块

注: [] 里的内容代表可省略

说明:

1. 初始化方法名必须为\_\_init\_\_ 不可改变

2. 初始化方法会在构造函数创建实例后自动调用，且将实例自身通过第一个参数self 传入 \_\_init\_\_ 方法

3. 构造函数的实参将通过 \_\_init\_\_方法的参数列表 传入到 \_\_init\_\_ 方法中

4. 初始化方法内如果需要return 语句返回，则只能返回None

示例见:

init\_method.py

#### 析构方法

语法:

class 类名(继承列表):

def \_\_del\_\_(self):

语句块

作用:

通常用来释放此对象占用的资源

说明:

1. 析构方法会在对象被销毁时自动调用

2. python语句建议不要在对象销毁时做任何事情，因为对象销毁的时间难以确定

示例:

del\_method.py

示例：

不绑定变量也可以实例，只不过语句结束会被销毁

>>> class Test:

... def prt(self):

... print('start')

... def \_\_del\_\_(self):

... print('over')

...

**>>> Test().prt()**

start

over

#### 类方法 @classmethod

问题:

类方法会自动 a --> a.\_\_class\_\_ 【？】

1. 类方法属于类

2. 实例方法属于该类的实例

3. 请问: 类内能不能有函数, 这个函数不属于类, 也不属于实例

概念：

类方法是描述类的行为的方法, 类方法属于类, 不属于该类创建的实例

说明:

1. 类方法需要用@classmethod装饰器定义

2. 类方法至少有一个形参, 第一个形参用于绑定类, 约定写为'cls'

3. 类实例和对象实例都可以调用类方法

4. 类方法不能访问此类创建的实例的属性(只能访问类变量)

示例:

* # 此示例示意类方法的定义方法和用法

class A:

v = 0 # <<<---类变量

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

self.my\_v = 10000

@classmethod

def get\_v(cls):

'''此方法为类方法，cls用于绑定调用此方法的类

此方法用于返回类变量v的值

'''

return cls.v

# return cls.my\_v # 出错

@classmethod

def set\_v(cls, value):

cls.v = value

print(A.get\_v()) # 0 调用类方法返回值

A.set\_v(100)

print(A.get\_v()) # 100

a = A()

print(a.get\_v()) # 100

a.set\_v(200)

print(a.get\_v()) # 200

print(A.get\_v()) # 200

print(a.my\_v) # 10000

# 无法用类方法访问调用此对象的a的my\_v实例变量

print(a.get\_v())

#### 静态方法@staticmethod

静态方法 @staticmethod

静态方法是定义在类内的函数，此函数的作用域是类的内部

静态方法不属于类,也不属于类的实例,它相当于定义在类内普通函数,只是它的作用域属于类

说明:

静态方法需要使用staticmethod装饰器定义

静态方法与普通函数定义相同，不需要传入self实例参数和cls类参数

静态方法只能凭借该类或类的实例调用

静态方法不能访问类变量和实例变量(属性)

示例见:

static\_method.py

小结 ：

实例方法, 类方法, 静态方法, 函数

### 类的初始化过程

dir(对象)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_']

对象.\_\_dict\_\_

{'\_\_module\_\_': '\_\_main\_\_',

'\_\_init\_\_': <function Person.\_\_init\_\_ at 0x00000000025D4840>,

'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>,

'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Person' objects>,

'\_\_doc\_\_': None}

实例传参

初始化函数\_\_init\_\_

生成\_\_dict\_\_

...

## 继承/派生

继承/派生的概念：

继承是从已有的类中派生出新的类，新类具有原类的行为，并能扩展新的行为

派生类就是从一个已有的类衍生出新的类，在新的类上可以添加新的属性和行为

作用:

1. 用继承派生机制，可以将一些共有功能加在基类中，实现代码共享.

2. 在不改变超类的代码的基础上改变原有的功能

名词

基类(base class) / 超类(super class) / 父类(father class)

派生类(derived class) / 子类

继承说明:

任何类都直接可间接的继承自object类

object类是一切类的超类

### 单继承

语法:

class 类名(基类名):

语句块

说明:

单继承是指派生类由一个基类衍生出来

示例见:

inherit.py

显式，初始化。

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

pass

### 调用父类

#### 通过类对象属性A.\_\_base\_\_

>>> class Aaa:

pass

>>> Aaa.\_\_base\_\_

<class 'object'>

>>> object.\_\_base\_\_ is None

True

* 方法覆盖 override

什么是覆盖

覆盖是指在有继承关系的类中，子类中实现了与基类同名的方法，在子类实例调用该方法时，实际调用的是子类中的覆盖版本的方法的现象叫覆盖

示例见：

class A:

'''A类'''

def work(self):

print("A.work被调用!")

class B(A):

'''B类'''

def work(self):

'''work 方法覆盖了父类的work'''

print("B.work被调用!")

b = B()

b.work() # B.work

a = A()

a.work() # A.work

b.work() # B.work

b.\_\_class\_\_.\_\_base\_\_.work(b) # A.work

问题:

在override.py中 ，b能否调用到父类的work方法?

#### super 函数:

* 格式：

super(class, cls) 返回绑定超类的实例(要求cls必须为class的实例)

类中使用时可以省略参数

super() 返回绑定超类的实例，等同于 super(self.\_\_class\_\_, self), 必须用在方法内调用。如： super().work()等同于super(B, self).work()

* 作用：

返回绑定超类的实例，

用超类的实例来调用其父类的覆盖方法

* 说明：

先去找上一层

super(类，对象)。是按照查找顺序去调用的

* 示例:

# 此示例示意用super构造函数来间接调用父类的覆盖版本的方法

class A:

def work(self):

print("A.work()")

class B(A):

def work(self):

print("B.work()")

def super\_work(self):

'''此方法先调用一下子类的work,

再调用一下父类的work'''

self.work() # 调用自己的work

# **super(B, self).work()**  # 调用父类的work

**super().work()** # 调用父类的work

b = B()

b.work() # B.work(),

super(B, **b**).work() # A.work()

b.super\_work() # B.work() # A.work()

# super().work() # 错的!!! super() 不能在方法外使用

* 应用

显式调用基类的构造方法

当子类中实现了\_\_init\_\_方法，基类的构造方法并不会被调用，此时需要显式调用基类的构造方法

示例见:

# super\_init.py

class Human:

def \_\_init\_\_(self, n, a):

self.name = n

self.age = a

print("Human.\_\_init\_\_被调用")

def infos(self):

print("姓名:", self.name)

print("年龄:", self.age)

class Student(Human):

def \_\_init\_\_(self, n, a, s):

**super().\_\_init\_\_(n, a)** # 显式调用父类的初始化方法

self.score = s

print("Student.\_\_init\_\_被调用")

def infos(self):

super().infos()

print("成绩:", self.score)

# h1 = Human('小张', 18)

# h1.infos()

s1 = Student("魏老师", 35, 60)

s1.infos()

### 类继承判断函数

#### issubclass()判断实例与对象的继承关系

* 概念

判断一个类的实例是否是另一个类的子类。

* 语法：

issubclass(cls, class\_or\_tuple)

判断一个类是否继承自其它的类,如果此类cls是class或 tuple中的一个派生子类则返回True,否则返回False

* 探索实例：

class A:

pass

class B(A):

pass

class C(B):

pass

issubclass(C, (A, B)) # True

issubclass(C, (int, str)) # False

探索对象绑定

class Animal(object):

def run(self):

print('Animal is running...')

class Dog(Animal):

def run(self):

print('Dog is running...')

class Cat(Animal):

def run(self):

print('Cat is running...')

def run\_twice(animal):

animal.run()

animal.run()

a = Animal()

d = Dog()

c = Cat()

print('a is Animal?', isinstance(a, Animal))

print('a is Dog? ', isinstance(a, Dog))

print('a is Cat? ', isinstance(a, Cat))

print('d is Animal?', isinstance(d, Animal))

print('d is Dog? ', isinstance(d, Dog))

print('d is Cat? ', isinstance(d, Cat))

run\_twice(c)

结果：

a is Animal? True

a is Dog? False

a is Cat? False

d is Animal? True

d is Dog? True

d is Cat? False

Cat is running...

Cat is running...

#### Isinstance()判断对象与对象间继承关系

isinstance(class, class\_or\_tuple)

返回这个对象class 是否是某个类的对象，或者某些类中的一个类的对象，如果是则返回True, 否则返回False

例子，如果继承关系是：

object -> Animal -> Dog -> Husky

>>> a = Animal()

>>> d = Dog()

>>> h = Husky()

然后，判断：

>>> isinstance(h, Husky)

True

示例:

isinstance(3.14, float) # True

isinstance('hello', str) # True

class Dog:

pass

dog1=Dog()

isinstance(dog1, Dog) # True

### 应用

* 实例：

思考下列表代码做什么事儿?

继承list类应用其方法

class MyList(list):

def insert\_head(self, n):

self.insert(0, n)

myl=MyList(range(3, 6))

myl.insert\_head(2)

myl.append(6)

print(myl) # [2, 3, 4, 5, 6]

### 继承和派生研究实例

* 查看内建类的继承

查看python内建类的继承关系的方法:

>>> help(\_\_builtins\_\_)

>>> \_\_builtins\_\_.\_\_dict\_\_

探索：

内建类的继承关系

object --> int --> bool

一切皆对象啊！！！

* 探索继承实质【？可删】

## 封装 enclosure

封装 enclosure

封装是指隐藏类的实现细节, 让使用者不用关心这些细节

封装的目的是让使用者尽可能少的实例变量(属性)进行操作

私有属性:

"\_\_name"，私有类型，

python类中, 以双下划线'\_\_'开头, 不以双下划线结尾的标识符为私有成员, 在类的外部无法直接访问

"\_name"，保护类型，protected 类型

单下划线表示，\_foo: 以单下划线开头的表示的是 protected 类型的变量，即保护类型只能允许其本身与子类进行访问，不能用于 from module import \*

示例:

# 此示例示意使用私有属性和私有方法:

class A:

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

self.\_\_p1 = 100 #\_\_p1为私有属性,在类的外部不可访问

def test(self):

print(self.\_\_p1) # 可以访问

self.\_\_m1() # A类的方法可以调用A类的私有方法

def \_\_m1(self):

'''我是私有方法,只有我自己的类中的方法才能调用我哦'''

print("我是A类的\_\_m1方法!")

a = A() # 创建对象

# print(a.\_\_p1) # 在类外看不到\_\_p1属性,访问失败!

a.test()

# a.\_\_m1() # 出错.无法调用私有方法

* 深度探索：

实例：

#私有变量深度探索：

**class** Student(object):

# \_xxx 是普通变量, 可直接访问, 人为约定俗成视为私有变量

# \_\_xxx 是私有变量(private变量), 一般不能直接访问

# \_\_xxx\_\_ 是特殊变量, 可直接访问, 特殊变量不是private变量。

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

self.\_\_name = name

self.\_\_score = score

**def** print\_score(self):

print(**'%s: %s'** % (self.\_\_name, self.\_\_score))

**def** get\_name(self):

**return** self.\_\_name

**def** get\_score(self):

**return** self.\_\_score

# 为什么要定义一个方法大费周折？

# 因为在方法中，可以对参数做检查，避免传入无效的参数

**def** set\_score(self, score):

**if** 0 <= score <= 100:

self.\_\_score = score

**else**:

**raise** ValueError(**'bad score'**)

print('（一）实例化')

bart = Student(**'Bart Simpson'**, 59)

print('（二）访问私有变量')

# （1）正常访问

print(bart.get\_name()) # 'Bart Simpson'

# （2）错误访问

try:

print(bart.\_\_name) # 访问报错

**except** AttributeError **as** e:

print('不能直接访问\_\_name, 报错！！错误是：', e)

# 私有变量机制：因为Python解释器对外把\_\_name变量改成了\_Student\_\_name。

# 所以还可以通过漏洞访问。强烈建议不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

print('（三）验证私有变量机制')

print(bart.\_Student\_\_name) # 'Bart Simpson'

print(bart.get\_name()) # 'Bart Simpson'

print('（四）修改内部私有变量')

# （1）正常修改

bart.set\_score(89)

print(bart.get\_score()) # 'Bart Simpson'

# （2）非正常修改（利用机制）

bart.\_Student\_\_name = **'New Name'**

print(bart.get\_name()) # 'New Name'

# （3）错误修改

bart.\_\_name = **'hello'**

print(bart.\_\_name) # 'New Name'

print(bart.get\_name()) # 'Bart Simpson'

# 设置\_\_name变量！

# 实质是外部代码给bart新增了一个\_\_name变量，并没有改变私有变量

# 内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name

## 多态

静态与动态

动态语言：可以在运行的过程中，修改代码

静态语言：编译时已经确定好代码，运行过程中不能修改

* 多态 polymorphic
* 字面意思: "多种状态"
* 多态是指在继承 / 派生关系的类中, 调用基类对象的方法, 实际能调用子类的覆盖版本方法的现象叫多态

说明:

多态调用的方法与对象相关, 不与类型相关

Python的全部对象都只有"运行时状态(动态)", 没有"C++/Java"里的"编译时状态(静态)"

面向对象的编程语言的特征:

继承

封装

多态

如: C + + / Java / Python / Swift / C

* 应用：

实例1：动态的指令输入函数执行

class Point:

def draw(self):

print('正在画一个点')

class Circle:

def draw(self):

print("正在画一个圆")

def my\_draw(s):

s.draw() # 此处显示出多态中的动态

L = [Circle(), Point(), Point(), Circle()]

for s in L:

my\_draw(s)

### 多继承

* 多继承 multiple inheritance

概念

多继承是指一个子类继承自两个或两个以上的基类

语法:

class 类名(基类名1, 基类名2, ....):

语句块

说明:

1. 一个子类同时继承自多个父类, 父类中的方法可以同时被继承下来

2. 如果两个父类中有同名的方法, 而在子类中又没有覆盖此方法时, 调用结果难以确定

* 多继承的问题(缺陷)

标识符(名字空间冲突的问题)

要谨慎使用多继承

话外：为了解决多继承的命名冲突，C++不支持多继承，引入了虚函数等概念，Java则…

命名冲突时执行的先后顺序：

实例1：命名冲突，优先执行\_\_mro\_\_列表中排更前面的语句

class A:

**def** work(self):

print(**"A.work被调用!"**)

class B:

**def** work(self):

print(**"B.work被调用!"**)

**class** AB(A, B):

pass

a = A()

b = B()

ab = AB()

a.work() *# A.work*

b.work() *# B.work*

print(AB.\_\_mro\_\_) # (<class ... <class 'object'>)

ab.work() # A.work，默认谁在前调用谁

A.work被调用!

B.work被调用!

(<class '\_\_main\_\_.AB'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>)

A.work被调用!

* \_\_mro\_\_的查找规则

使用C3算法

语法：

类名.\_\_mro\_\_ 返回查找对象排列的元组

作用：

类内的\_\_mro\_\_属性用来记录继承方法的查找顺序

实例探索：

对于如下继承关系，如何查找？

obj

\_\_ \_\_|\_\_ \_\_

/ | | \

/ | | \

A1 A2 A3 A4

\ | | / |

\ | | / |

B1 B2 B3

/ | | \ |

/ | | \ |

C1 C2 C3 C4

\ | | /

\\_\_|\_\_\_\_ \_\_\_\_|\_\_/

|

S

class A1: pass

class A2: pass

class A3: pass

class A4: pass

class B1(A1, A2): pass

class B2(A3, A4): pass

class B3(A4): pass

class C1(B1): pass

class C2(B1): pass

class C3(B2): pass

class C4(B2, B3): pass

class S(C1,C2,C3,C4): pass

print(S.\_\_mro\_\_)

'''

(<class '\_\_main\_\_.S'>,

<class '\_\_main\_\_.C1'>, <class '\_\_main\_\_.C2'>,

<class '\_\_main\_\_.B1'>,

<class '\_\_main\_\_.A1'>,

<class '\_\_main\_\_.A2'>,

<class '\_\_main\_\_.C3'>, <class '\_\_main\_\_.C4'>,

<class '\_\_main\_\_.B2'>,

<class '\_\_main\_\_.A3'>,

<class '\_\_main\_\_.B3'>,

<class '\_\_main\_\_.A4'>,

<class 'object'>)

'''

### 函数重写

函数重写 override

重写是在自定义的类内添加相应的方法,让自定义的类生成的对象(实例)像内建对象一样进行内建的函数操作

对象转字符串函数重写

repr(obj) 返回一个能代表此对象的表达式字符串,通常:

eval(repr(obj)) == obj (通常用于计算机通信)

str(obj) 通过给定的对象返回一个字符串(这个字符串通常是给人看的)

#### repr()和str()区别

探索实例

>>> a=1

>>> b=2

>>> str(a+b)

'3'

>>> repr(a+b)

'3'

#### 对象转字符串函数重写方法

repr() 函数的重写方法:

def \_\_repr\_\_(self):

return 能够表达self内容的字符串

str() 函数的重写方法:

def \_\_str\_\_(self):

return 人能看懂的字符串

说明:

1. str(obj) 函数优先调用obj.\_\_str\_\_()方法返回字符串

2. 如果obj没有\_\_str\_\_()方法, 则调用obj.\_\_repr\_\_()方法返回的字符串

3. 如果obj没有\_\_repr\_\_()方法, 则调用object类的 \_\_repr\_\_() 实例方法显示 < xxxx > 格式的字答鼓足

示例：

class MyNumber:

pass

# def \_\_len\_\_(self):

# return 100

n1 = MyNumber()

x = len(n1) # 重写了\_\_len\_\_方法才可以使用，否则报错

print('x =', x)

内建函数的重写

# 此示例示意一个自定义的数字类型重写 repr和 str的方法

#重写前：

class Mytest:

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

self.data = value

n1 = Mytest(100)

print('问题1:')

print(str(n1)) # 调用 n1.\_\_str\_\_(self)

print(n1) # 等同于print(str(n1))，与上句执行结果一致

print('问题2:')

print(repr(n1))

print('问题3:')

n = int(n1) # 普通的int方法无法做事情

print(type(n1))

print(type(n))

print(n)

问题1:

<\_\_main\_\_.Mytest object at 0x0000000001DEE160>

<\_\_main\_\_.Mytest object at 0x0000000001DEE160>

问题2:

<\_\_main\_\_.Mytest object at 0x0000000001DEE160>

问题3:

Traceback (most recent call last):

File "07\_str\_repr.py", line 34, in <module>

n = int(n1)

TypeError: int() argument must be a string, a bytes-like object or a number, not 'Mytest'

#重写后：

class Mytest:

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

self.data = value

def \_\_str\_\_(self):

print("\_\_str\_\_被调用")

return "数字: %d" % self.data

def \_\_repr\_\_(self):

print("\_\_repr\_\_被调用")

return 'Mytest(%d)' % self.data

def \_\_int\_\_(self):

print("\_\_int\_\_被调用")

return int(self.data)

n1 = Mytest(100)

print('问题1:')

print(str(n1)) # 调用 n1.\_\_str\_\_(self)

print(n1) # 等同于print(str(n1))，与上句执行结果一致

print('问题2:')

print(repr(n1))

print('问题3:')

n = int(n1)

print(type(n1))

print(type(n))

print(n)

问题1:

\_\_str\_\_被调用

数字: 100

\_\_str\_\_被调用

数字: 100

问题2:

\_\_repr\_\_被调用

Mytest(100)

问题3:

\_\_int\_\_被调用

<class '\_\_main\_\_.Mytest'>

<class 'int'>

100

其他问题深究

>>> i = -100

>>> abs(i)

100

>>> i.\_\_abs\_\_()

100

>>> help(int)

...

| Methods defined here:

|

| \_\_abs\_\_(self, /)

| abs(self)

...

## 对象的属性管理函数

详见文档

对象属性管理函数:

详见:

python\_base\_docs\_html/内建函数(builtins).html

|  |  |
| --- | --- |
| 函数 | 说明 |
| getattr(obj, name[, default]) | 从一个对象得到对象的属性；getattr(x, 'y') 等同于x.y; 当属性不存在时,如果给 出default参数,则返回default,如果没有给出default 则产生一个AttributeError错误 |
| hasattr(obj, name) | 用给定的name返回对象obj是否有此属性,此种做法可以避免在getattr(obj, name)时引发错误 |
| setattr(obj, name, value) | 给对象obj的名为name的属性设置相应的值value, set(x, 'y', v) 等同于 x.y = v |
| delattr(obj, name) | 删除对象obj中的name属性, delattr(x, 'y') 等同于 del x.y |

### 获取对象信息

>>> class MyObject(object):

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

self.x = 9

def power(self):

return self.x \* self.x

>>> obj = MyObject()

>>> hasattr(obj, 'x') # 有属性'x'吗？

True

>>> obj.x

9

>>> hasattr(obj, 'y') # 有属性'y'吗？

False

>>> setattr(obj, 'y', 19) # 设置一个属性'y'

>>> hasattr(obj, 'y') # 有属性'y'吗？

True

>>> getattr(obj, 'y') # 获取属性'y'

19

>>> obj.y # 获取属性'y'

19

>>> getattr(obj, 'z') # 获取属性'z'

Traceback (most recent call last):

File "<pyshell#28>", line 1, in <module>

getattr(obj, 'z') # 获取属性'z'

AttributeError: 'MyObject' object has no attribute 'z'

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

404

>>> hasattr(obj, 'power') # 有属性'power'吗？

True

>>> getattr(obj, 'power') # 获取属性'power'

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x00000000030670B8>>

>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn

>>> fn # fn指向obj.power

<bound method MyObject.power of <\_\_main\_\_.MyObject object at 0x00000000030670B8>>

>>> fn() # 调用fn()与调用obj.power()是一样的

81

## 类的属性查看方法

### 类对象属性

#-\*-coding:UTF-8-\*-

class A:

\_\_slots\_\_ = ['a','b']

pass

# 类属性

print(A.\_\_base\_\_) # 返回元组，包含其所有父类继承对象（按照多继承查找规则\_\_mro\_\_）

print(A.\_\_dict\_\_) # 返回字典，查看类中实例变量

print(A.\_\_mro\_\_) # 返回元组，查看多继承查找顺序

# 实例属性

a = A()

print(a.\_\_class\_\_) # 返回实例对象绑定的类对象

print(A)

print(A.\_\_class\_\_) # 返回实例对象绑定的类对象

print(type(A))

print(A.\_\_slots\_\_) # 返回被限制的类属性的列表，必须提前定义

print(a.\_\_slots\_\_) # 返回被限制的类属性的列表，必须提前定义

### 预置的实例属性

类在实例化时，会自动生成一个字典，存储实例变量

#### 类的\_\_dict\_\_属性

* 作用

\_\_dict\_\_属性绑定一个存储此实例自身变量的字典

* 示例:

class Dog:

pass

dog1=Dog()

print(dog1.\_\_dict\_\_) # {}

dog1.color="白色"

print(dog1.\_\_dict\_\_) # {'color': '白色'}

* 应用：
* 示例1：字典对象传参妙用

# 传入字典参数，减少代码

# 原代码：

**class** Person:

**def** \_\_init\_\_(self, \_obj):

self.name = \_obj[**'name'**]

self.age = \_obj[**'age'**]

self.energy = \_obj[**'energy'**]

self.gender = \_obj[**'gender'**]

self.email = \_obj[**'email'**]

#修改代码：

**class** Person:

**def** \_\_init\_\_(self, \_obj):

self.\_\_dict\_\_.update(\_obj)

d = {'name': 1, 'age': 2, 'energy': 3, 'gender': 4, 'email': 5}

s1 = Person(d)

print(s1.\_\_dict\_\_) *# 实例对象的字典*

print(**'s1.name'**, s1.name)

print('------------------------------')

print(Person.\_\_dict\_\_) *# 类对象的字典*

{'name': 1, 'age': 2, 'energy': 3, 'gender': 4, 'email': 5}

s1.name 1

------------------------------

{'\_\_module\_\_': '\_\_main\_\_', '\_\_init\_\_': <function Person.\_\_init\_\_ at 0x0000000001DD4840>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Person' objects>, '\_\_doc\_\_': None}

#### 类的\_\_class\_\_属性

\_\_class\_\_属性绑定创建此实例(类实例)的类

* 作用:

可以借助于此属性来访问创建此实例的类

* 示例:

class Dog:

pass

dog1 = Dog()

dog2 = dog1.\_\_class\_\_()

print(Dog) # 类对象Dog，结果<class '\_\_main\_\_.Dog'>

print(dog1.\_\_class\_\_) # 类对象Dog，结果<class '\_\_main\_\_.Dog'>

print(dog2.\_\_class\_\_) # 类对象Dog，结果<class '\_\_main\_\_.Dog'>

print(dog1) # 命名为dog1的实例对象，结果<\_\_main\_\_.Dog object at 0x02FD5F10>

print(dog1) # 命名为dog1的实例对象，结果<\_\_main\_\_.Dog object at 0x02FF0590>

print(Dog()) # 未绑定变量的实例对象，结果<\_\_main\_\_.Dog object at 0x02FF0650>

print(Dog) # 类对象，结果<class '\_\_main\_\_.Dog'>

#### 类的\_\_base\_\_属性

\_\_base\_\_属性用来记录此类的基类

* 示例:

class Human:

pass

class Student(Human):

pass

class Teacher(Human):

pass

Student.\_\_base\_\_ is Human # True

内建类的继承关系见:

>>> help(\_\_builtins\_\_)

注意，返回的是其所有继承对象的元组

class ParentClass1:

pass

class ParentClass2:

pass

class SubClass1(ParentClass1):

pass

class SubClass2(ParentClass1,ParentClass2):

pass

print(type(SubClass1.\_\_bases\_\_))

print(SubClass1.\_\_bases\_\_)

print(SubClass2.\_\_mro\_\_)

### 类的\_\_slots\_\_属性

* 问题描述：

在创建类实例对象时，一般会初始化生成字典，记录类属性。

所有类属性都是用字典记录的，通过 实例名.\_\_dict\_\_ 可查看访问。

* \_\_slots\_\_作用:

限定一个类创建的实例只能有固定的属性(实例变量)，不允许对象添加列表以外的属性

访止用户因错写属性的名称而发生程序错误

说明:

含有\_\_slots\_\_属性的类所创建的实例没有\_\_dict\_\_属性,即此实例不用字典来存储对象的属性

\_\_slots\_\_列表作用：

1. 定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性
2. 阻止实例化类时为实例分配字典dict。
3. 减少内存

网络原话：

在python新式类中，可以定义一个变量\_\_slots\_\_，它的作用是阻止在实例化类时为实例分配字典dict

使用slots可以让内存使用减少3.5倍！！# 通过 (200 - 4) / ((60 - 4) \* 1.0) 计算得来【来源：

<https://www.jb51.net/article/118024.htm>

】

说明:

含有\_\_slots\_\_列表的类创建的实例对象没有\_\_dict\_\_属性, 即此实例不用字典来保存对象的属性(实例变量)

* 示例:

示例1：限制添加实例属性

**class** Person(object):

\_\_slots\_\_ = (**"name"**, **"age"**) # 即便是[]，也不能再添加属性

P = Person()

P.name = **"老王"**

P.age = 20

# P.score = 100 *# 此句报错*

# Traceback (most recent call last):

# File "<pyshell#3>", line 1, in <module>

# AttributeError: Person instance has no attribute 'score'

注意：可以正常添加类属性，\_\_slots\_\_只是限制添加实例属性

...

P.\_\_class\_\_.score = 100

print(Person.score)

Person.score=10

print(Person.score)

注意：添加的类属性，名字不能与实例属性相同，否则会发生冲突

示例2：此示例示意 类的变量 \_\_slots\_\_列表的作用

class Student:

\_\_slots\_\_ = ['name', 'score']

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

self.name = name

self.score = score

s1 = Student('小张', 58)

print(s1.score)

# s1.socre = 100 # 此处错写了属性名,但在运行时不会报错!

# print(s1.\_\_dict\_\_) # 报错

s1.score = 100

print(s1.score) # 请问打印的值是多少?

### self代表类的实例，而非类

self代表类的实例，而非类本身

类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称, 按照惯例它的名称是 self。

self不是python关键字，若将self替换为其他单词一样正常运行。

实例：

class A:

def prt1(self):

print(self)

def prt2(self):

print(self.\_\_class\_\_)

a = A()

# 调用实例对象: <\_\_main\_\_.Test object at 0x7feddc55e2b0>

print(a) # 类外调用实例对象

A().prt1() # 类中调用实例对象

# 调用类对象: <class '\_\_main\_\_.Test'>

print(a.\_\_class\_\_) # 类外通过实例去调用类对象

print(A().\_\_class\_\_) # 类外通过实例去调用类对象

print(A) # 类外调用类对象

A().prt2() # 类中调用类对象

print(a.\_\_class\_\_ is A) # True

### 类变量（类属性）

类变量class variable(也叫类属性)

#### 问题引入：

* 全局的变量有哪些？

>>> class Human:

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

self.name = n

>>> h1 = Human('小张')

>>> h2 = Human('小李')

>>> dir() #可以看到有3个变量，h1,h2,Human，分别绑定实例1，实例2，类对象

结论：dir()查看有3个变量，类名变量绑定类

实例变量 不同于 类变量

问题:

1. 对象内可以有:

实例变量

实例方法

2. 类内可以有:

类变量

类方法 # 这个可以有

#### 基本概念

类变量：

类变量是的类的属性，此属性属于类，不属于类的实例

作用：

通常用来存储该类对象共有的数据

说明：

类变量可以通过类直接访问

类变量可以通过类的实例直接访问

类变量可以通过此类的实例的\_\_class\_\_属性间接访问

语法：

class 类名(继承列表):

类变量名 = 表达式

...

#### 应用

# <类的实例对象> 不同于 <类的对象> ，但前者可以访问和修改后者变量

# 类变量的定义和使用

class Human:

count = 0 # 创建类变量

print("Human的类变量count=", Human.count) # 0

Human.count = 100

print(Human.count) **# 100**

# Human类的实例可以访问和修改count类变量

class Human:

count = 0 # 创建类变量

h1 = Human()

print("用h1对象访问Human的count变量", h1.count) # 0

h1.count = 100 # 此做法是为实例添加一个变量，并不是修改类变量

print(h1.count) **# 100**

print(Human.count) **# 0**

h1.\_\_class\_\_.count = 200 # 此做法修改类变量

print("h1.count=", h1.count) # 100

print('Human.count=', Human.count) **# 200**

# 此示例示意用类变量来记录对象的个数

class Car:

count = 0 # 创建类变量, 用来记录汽车对象的总数

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

print(info, "被创建")

self.data = info # 记录传入数据

self.\_\_class\_\_.count += 1 # 让车的总数加1

def \_\_del\_\_(self):

print(self.data, '被销毁')

# 当车被销毁时总数自动减1

self.\_\_class\_\_.count -= 1

print('当前汽车总数是:', Car.count)

b1 = Car("BYD 京A.88888")

print(Car.count)

b2 = Car('TESLA 京B.00000')

b3 = Car('Audi, 京C.66666')

print('当前汽车总数是:', Car.count)

del b1

del b2

print("当前汽车数是:", Car.count)

### 类的文档字符串:

类内第一个没有赋值给任何变量的字符串是类的文档字符串

类的文档字符串由类的\_\_doc\_\_属性绑定

说明:

类的文档字符串用类的\_\_doc\_\_属性可以访问

类的文档字符串可以用help()函数查看

示例:

class Dog:

'''这是类的文档字符串'''

pass

>>> help(Dog)

>>> dog1 = Dog()

>>> help(dog1)

## 研究实例

### 实例属性和类属性区别

**class** Student(object):

name = **'Student'**

s = Student() *# 创建实例s*

print(**'s.name:'**, s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性

# Student

print(**'Student.name:'**, Student.name) *# 打印类的name属性*

# Student

s.name = **'Michael'** *# 给实例绑定name属性*

print(" --> 修改s.name = 'Michael'" )

print(**"s.name:"**, s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性

# Michael

print(**'Student.name:'**, Student.name) *# 但是类属性并未消失，用Student.name仍然可以访问*

# Student

**del** s.name # 如果删除实例的name属性

print(' --> 删除del s.name ')

print(**'s.name:'**, s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了

# Student

s.name: Student

Student.name: Student

--> 修改s.name = 'Michael'

s.name: Michael

Student.name: Student

--> 删除del s.name

s.name: Student

# 模块和包

## 基本概念

### 模块Module

#### 概念

模块是一个包含有一系列数据，函数，类等组成的程序组

模块是一个文件，模块文件名通常以'.py'结尾

作用:

1. 让一些相关的数据，函数,类等有逻辑的组织在一起，使逻辑结构更加清晰

2. 模块中的变量，函数和类等可提供给其它模块或程序使用

### 模块的文档字符串

模块内第一个没有赋值给任何变量的字符串为文档字符串

模块的\_\_doc\_\_属性:

用于绑定模块文档字符串

\_\_file\_\_ 属性

绑定模块对应的文件路径

### 模块的属性

模块内第一个没有赋值给任何变量的字符串为文档字符串

#### 分类

* \_\_doc\_\_属性:

用于绑定模块文档字符串

* \_\_file\_\_属性:

绑定模块对应的文件路径

* \_\_name\_\_属性:

用来记录模块的自身的名字

\_\_name\_\_作用:

1. 记录模块名
2. 判断是否为主模块

说明:

1. 当此模块作为主模块(也就是第一个运行的模块)运行时,\_\_name\_\_绑定'\_\_main\_\_'
2. 当此模块不是主模块时，\_\_name\_\_绑定模块名(文件名去掉.py后缀)

**if \_\_name\_\_ == '\_\_main\_\_'作用：**

if \_\_name\_\_ == '\_\_main\_\_'的意思是：

当.py文件被直接运行时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块将被运行；

当.py文件以模块形式被导入时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块不被运行。

如何简单地理解Python中的if \_\_name\_\_ == '\_\_main\_\_' - CSDN博客

<https://blog.csdn.net/yjk13703623757/article/details/77918633>

示例：

查看name属性

# mymod3.py

print("载入mymod3")

print("\_\_name\_\_ =", \_\_name\_\_)

print("mymod3模块内的:\_\_name\_\_属性绑定是：", \_\_name\_\_)

# test\_mymod3.py 主模块

import mymod3

print('------------------------------------')

print('<进入主模块test\_mymod3.py>')

print("\_\_name\_\_ =", \_\_name\_\_)

print("主模块内的:\_\_name\_\_属性绑定是：", \_\_name\_\_)

print("可访问载入模块，mymod3.\_\_name\_\_ =", mymod3.\_\_name\_\_)

主模块运行结果：

载入mymod3

\_\_name\_\_ = mymod3

mymod3模块内的:\_\_name\_\_属性绑定是： mymod3

------------------------------------

<进入主模块test\_mymod3.py>

\_\_name\_\_ = \_\_main\_\_

主模块内的:\_\_name\_\_属性绑定是： \_\_main\_\_

可访问载入模块，mymod3.\_\_name\_\_ = mymod3

#### 属性查看

* 模块访问方法：

载入模块后：

print(模块名.\_\_doc\_\_)

print(模块名.\_\_file\_\_)

print(模块名.\_\_name\_\_)

* help函数

help(已导入的模块)

* dir 函数

dir([对象])

返回绑定的变量的列表。

返回当前对象或作用域内变量名的列表。

* 作用:

1. 如果没有参数调用，则返回当前作用域内所有变量的列表

2. 如果给定一个对象作为参数,则返回这个对象的所在变量(属性)列表

1) 对于一个模块，返回这个模块的全部变量

2) 对于一个类对象，返回类对象的所有变量,并递归基类对象的所有变量

3) 对于其它对象，返回所有变量、类变量和基类变量

### \_\_all\_\_列表

#### 模块的\_\_all\_\_列表

模块中的\_\_all\_\_列表是一个用来存放可导出属性的字符串列表

* 模块的隐藏属性

模块中以'\_'开头的属性,

在from xxx import \*导入时将不被导入,通常称这些属性为隐藏属性

作用：

限定当用from xxx import \*语句导入时，只导入\_\_all\_\_列表内的属性

**注意：只对 from import \* 语句起作用**

#### \_\_init\_\_.py内的 \_\_all\_\_ 列表

作用:

用来记录此包中有哪些包或模块需要在from import \*语句导入时被导入

说明:

\_\_all\_\_列表只在from xxx import \*语句中起作用

包的相对导入

是指包内模块的相互导入

示例见:

# mymod4.py

# \_\_all\_\_ 列表限定其它模块在用from mymod4 import \*导入时

# 只导入 'myfun1', 'myfun3', 'name1'

\_\_all\_\_ = ['myfun1', 'myfun3', 'name1']

def myfun1():

print("myfun1被调用")

def myfun2():

print("myfun2被调用")

def myfun3():

print("myfun3被调用")

name1 = 'aaaaa'

name2 = 'bbbbbbb'

### 包(模块包) package

#### 概念

包是将模块以文件夹的组织形式进行分组管理的方法

* 作用:

将一系列模块进行分类管理，有利于访问命名冲突

可以在需要时加载一个或部分模块，而不是全部模块

换言之，导入包时，并不导入下面的所有模块【待验证【？】】

* \_\_init\_\_.py 文件

\_\_init\_\_.py 是常规包内必须存在的文件（内容可空）

\_\_init\_\_.py 会在包加载时被自动调用

文件作用:

编写此包的内容

在内部填写包的文档字符串

* 包的加载路径：

同模块的加载路径搜索

1. 搜索当前路径

2. 搜索sys.path给定的路径

* 包示例:

mypack/

\_\_init\_\_.py

menu.py

games/

\_\_init\_\_.py

contra.py

supermario.py

tanks.py

office/

\_\_init\_\_.py

excel.py

word.py

powerpoint.py

* 创建命令:

mkdir mypack

cd mypack

touch \_\_init\_\_.py menu.py

mkdir games office

cd games

touch \_\_init\_\_.py contra.py supermario.py tanks.py

cd ../office

touch \_\_init\_\_.py excel.py word.py powerpoint.py

#### 包的导入语法:

* 包的导入:

import 语句

from import 语句

from import \* 语句

# 同模块的导入规则相同

import 包名 [as 包别名]

import 包名.模块名 [as 模块新名]

import 包名.子包名.模块名 [as 模块新名]

from 包名 import 模块名 [as 模块新名]

from 包名.子包名 import 模块名 [as 模块新名]

from 包名.子包名.模块名 import 属性名[as 属性新名]

from 包名 import \*

from 包名.子包名 import \*

...

* 包的相对导入

是指包内模块的相互导入

语法:

from 相对路径包或模块 import 属性或模块

或

from 相对路径包或模块 import \*

说明:

包的相对导入不能用于import xxx 语句中

相对路径:

. 代表当前目录

.. 代表上一级目录

... 代表上二级目录

.... 以此类推

**注: 相对导入时不能超出包的外部**

示例：

from ..menu import \*

from .games import \*

## 模块的导入

导入实质：

导入语句的实质是 在本地创建变量来绑定模块/函数/数据

### 模块导入的三种语句

三种语句:

import 语句

from import 语句

from import \* 语句

* import 语句

作用:

将一个模块整体导入到当前模块中

语法:

import 模块名1 [as 模块新名1], 模块名2[as 模块新名2], ....

示例:

import math # 导入数学模块

import sys, os # 导入sys和os模块

import copy as cp

属性用法:

模块名.属性名

help(obj) 可以查看模块的文档字符串

练习:

1. 输入一个圆的半径，打印出这个圆的面积

2. 输入一个圆的面积，打印出这个圆的半径

(要求用math模块内的函数和变量)

* from import 语句

语法:

from 模块名 import 模块属性名 [as 属性新名1], 模块属性名2 [as 属性新名2]

作用:

将某模块的一个或多个属性导入到当前模块的作用域

示例:

from math import pi

from math import sin

from math import factorial as fac

* from import \* 语句

语法:

from 模块名 import \*

作用:

将某模块的所有属性导入到当前的模块

示例:

from math import \*

s = sin(pi/2)

print(factorial(10))

### 模块导入和执行的机制

模块被导入和执行的过程:

先搜索相关路径找模块(.py文件)

判断是否有此模块对应的.py文件，如果.pyc文件比.py文件新，则直接加载.pyc文件

否则用模块.py 文件生成.pyc并加载执行

* 编译模块

pyc模块的编译 compile

编译 解释执行

mymod.py -----> mymod.pyc -------> python3

* 导入模块步骤：

搜索模块 --> 编译模块 --> 加载执行模块

* 搜索模块

import语句搜索<模块.py>的路径顺序(查找.py的顺序)

1. 本地路径 ：搜索程序运行时的路径(当前路径)

2. sys.path ：sys.path 提供的路径

3. 内置模块 ：搜索内置模块

当模块文件不能被搜索到时怎么办？

使用append方法添加路径

示例：

* 加载执行模块

模块的加载过程:

在模块导入时，模块所有语句会执行

如果一个模块已经被导入，则再次导入时不会重新执行模块内的语句

### 模块的重新导入

模块的重新加载

* 语法：

import mymod

import imp

imp.reload(mymod) # 在运行时重新加载mymod 模块

* 模块的重新加载

>>> import math

>>> import imp

>>> imp.reload(math) # 在运行时重新加载math模块

<module 'math' (built-in)>

* 应用
* 现象及问题：

# mymod.py

name1 = 'tesla'

>>> import mymod

>>> mymod.name1

'tesla'

>>> mymod.name1='a'

>>> import mymod

>>> mymod.name1

'a'

解释：

模块一旦加载，此后不会重新加载。即使再次使用import mymod

解决方案：

对象.reload()

## 模块的分类:

1. 内置模块(builtins) 在解析器的内部可以直接使用

2. 标准库模块,安装python时已安装具可直接使用

3. 第三方模块（通常为开源), 需要自己安装

4. 用户自己编定的模块(可以作为其它人的第三方模块)

### 内建模块

什么是内建模块？

builtins是内建模块

Python启动时相当于执行from builtins in \*，识别max，len，等

### 自定义模块

#### 模块中载入的变量作用范围

载入模块并不是将模块的内容直接替换到主模块，

再次验证了global的全局变量作用域是在文件范围

示例：

# mymod6.py

var = 100

def print\_var():

print('载入模块var:', var)

def set\_var(n):

global var

var = n

# test\_mymod6.py

from mymod6 import var

var = 200 # 赋值只能改变本模块内的全局变量

from mymod6 import print\_var

print\_var() # 100

from mymod6 import set\_var

set\_var(300)

print\_var() # 300

print('主模块var:',var)

主模块运行结果

载入模块var: 100

载入模块var: 300

主模块var: 200

示例：

主模块中使用sys.path.append添加需要载入模块的路径

给自定义模块添加载入搜索路径

import sys

# 为sys.path 列表添加一个路径

sys.path.append('/home/tarena')

import mymod

from mymod import name1, name2

mymod.myfac(5)

mymod.mysum(100)

文件名(模块名).py

# mymod.py

''' 此示例示意自定义模块

此模块中有二个函数:

myfac(n), mysum(n)

此模块中有两个数据:

name1, name2

'''

def myfac(n):

'''我是myfac 的文档字符串'''

print("正在计算%d的阶乘!" % n)

def mysum(n):

print("正在计算%d的和!" % n)

name1 = "audi"

name2 = "tesla"

print("mymod 模块被导入")

# test\_mymod.py

import mymod

from mymod import name1, name2

mymod.myfac(5)

mymod.mysum(100)

print(mymod.name1)

print(mymod.name2)

print('name1=', name1)

name1 = "魏老师"

print('name1=', name1)

## 常用模块

### sys模块

——系统模块

此模块都是运行时系统的信息，与系统相关的信息

#### sys模块的方法

|  |  |  |
| --- | --- | --- |
| 变量 | 返回 | 描述 |
| sys.path | 列表 | 返回路径的列表，模块搜索路径 path[0] 是当前脚本程序的路径名，否则为 '' |
| sys.modules | 字典 | 已加载模块的字典 |
| sys.version | 字符串 | 版本信息字符串 |
| sys.version\_info | 元组 | 版本信息的命名元组 |
| sys.platform | 字符串 | 操作系统平台名称信息 |
| sys.argv | 列表 | 返回关于当前路径的列表。命令行参数 argv[0] 代表当前脚本程序路径名:[路径,数值,数值] |
| sys.copyright | 字符串 | 获得Python版权相关的信息 |
| sys.builtin\_module\_names | 元组 | 获得Python内建模块的名称（字符串元组） |
| 标准输入输出时会用到 |  |  |
| sys.stdin | 【？】 | 标准输入文件对象，多用于input() |
| sys.stdout |  | 标准输出文件对象,多用于print() |
| sys.stderr |  | 标准错误输出文件对象, 用于输出错误信息 |

**sys模块的方法**

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回 | 描述 |
| sys.exit([arg]) |  | 退出程序，正常退出时sys.exit(0) |
| sys.getrecursionlimit() | 整型数 | 得到递归嵌套层次限制（栈的深度） |
| sys.setrecursionlimit(n) | None | 得到和修改递归嵌套层次限制（栈的深度） |

#### 应用

* 1、 sys.version检测程序是否是在python3环境下运行：

import sys

print("i'm in python%s" % sys.version[0])

print("当前程序正运行在:", sys.platform, '上')

if sys.version\_info[0] < 3:

print("can not run in python2")

sys.exit(0) # exit programe

print("-------------------")

print("programe running normally")

i'm in python3

当前程序正运行在: win32 上

programe running normally

* 2、sys.argv从Linux命令窗口传入参数

文件：test.py

import sys

print(sys.argv)

命令行输入运行：

tarena@tedu:/mnt/hgfs/虚拟机/AID1805$ python3 get\_sys.py 1 2 3

['get\_sys.py', '1', '2', '3']

tarena@tedu:/mnt/hgfs/虚拟机/AID1805$ python3 get\_sys.py

['get\_sys.py']

* 3、捕获标准输入函数的末尾换行符

**输出函数**

等价情况：

print('hello')

sys.stdout.write('hello' + '\n')

**输入函数**

sys.stdin.readline( )会将标准输入全部获取，包括末尾的'\n'，不同于input自动去掉末尾'\n'

等价情况：

input( )

sys.stdin.readline( )[:-1]

sys.stdin.readline( ).strip('\n')

* 4、解决python2里面的编码问题

# UnicodeDecodeError: 'ascii' codec can't decode byte 0xe4 in position 8: ordinal not in range(128)

if sys.version[0] == '2':

import sys

reload(sys)

sys.setdefaultencoding('gb18030')

导入其他路径下的模块：

sys.path.append('/ifs/TJPROJ3/Plant/chenjun/mytools')

import fas

### os模块

——系统模块

#### 应用

实例：设置程序工作目录

import os

执行

os.system() #执行Linux语句，返回执行执行脚本的最后一句执行状态，成功返回0

os.popen().read() #执行Linux语句，返回语句的标准输出结果

目录获取

os.pardir #获取当前目录的父目录名称，字符串('..')

os.getcwd() #返回当前工作路径的字符串

os.curdir #返回当前目录 (为'.')

路径操作

print(os.path) # 【返回调用模块的路径】

os.path.isdir('/home/test') #判断路径是否存在，返回布尔值

os.path.exists()

os.path.join('a','aa','aaa') #得到'a/aa/aaa'

os.chdir('/home/test') #切换当前工作路径(切换到'/home'路径下)

os.sep #返回当前使用系统的分隔符 (windows '\';linux '/')

文件(夹)列表获取

os.listdir() #列表形式列出指定目录下的所有文件以及其子文件

os.walk() #列表形式列出指定目录下的所有文件以及其子文件的文件

文件名操作

os.path.splitext(filename) # 返回filename的后缀名

os.getpid() # 获取当前进程PID

示例：

1. 获取文件名的后缀名

>>> import os

>>> os.path.splitext('01.tonglu.py')

('01.tonglu', '.py')

应用：

# 检测输出文件夹是否存在，若不存在则创建文件夹

def mkdir(mkpath):

isExists = os.path.exists(mkpath)

if not isExists:

os.makedirs(mkpath)

mkpath = "WGCNA\_Module\_Gene\_ID--goseq" # 输出文件夹

mkdir(mkpath)

# 复制文件到另一个文件夹

### random随机模块

* 说明：

random模块是用于模拟或生成随机输出的模块.

import random as R

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回 | 描述 |
| R.random() | 浮点数 | 返回一个[0, 1) 之间的随机实数 |
| R.uniform(a,b) | 浮点数 | 返回[a,b) 区间内的随机实数 |
| R.randrange([start,] stop[, step]) | 整型数 | 返回range(start,stop,step)中的随机数 |
| R.choice(seq) | 元素对象 | 从序列中返回随意元素 |
| R.shuffle(seq[, random]) | 【？】 | 随机指定序列的顺序(乱序序列） |
| R.sample(seq,n) |  | 从序列中选择n个随机且不重复的元素 |
| R.getrandbits(nbit) |  | 以长整型的形式返回用nbit位来表示的随机数 |
| R.seed(a=None) |  | 用给定的数a设置随机种子,不给参数a则用当前时间设置随机种子 |

练习:

猜数字游戏:

随机生成一个0~100的整数,用变量x绑定

让用户输入一个数y,输出猜数字的结果.

1) 如果y等于x,则提示"恭喜您猜对了!", 退出程序

2) 如果y大于x,同提示"您猜大了"

3) 如果y小于x,同提示"您猜小了"

让用户循环输入，直到猜对为止，同时显示用户猜数字的次数后退出程序

### math数学模块

文档参见:

python\_base\_docs\_html/数学模块math.html

* 数学模块 math

模块名: math

* 注：

linux下为内建模块

Mac OS下为标准库模块

* 数学模块用法：

import math

# 或

from math import \*

|  |  |
| --- | --- |
| 变量 | 描述 |
| math.e | 自然对数的底e |
| math.pi | 圆周率pi |

|  |  |
| --- | --- |
| 函数名 | 描述 |
| math.ceil(x) | 对x向上取整，比如x=1.2，返回2 |
| math.floor(x) | 对x向下取整，比如x=1.2，返回1 |
| math.sqrt(x) | 返回x的平方根 |
| math.factorial(x) | 求x的阶乘 |
| math.log(x[, base]) | 返回以base为底x的对数, 如果不给出base,则以自然对数e为底 |
| math.log10(x) | 求以10为底x的对数 |
| math.pow(x, y) | 返回 x\*\*y (x的y次方) |
| math.fabs(x) | 返回浮点数x的绝对值 |
| 角度和弧度degrees互换 |  |
| math.degree(x) | 将弧度x转换为角度 |
| math.radians(x) | 将角度x转换为弧度 |
| 三角函数 |  |
| math.sin(x) | 返回x的正弦(x为弧度) |
| math.cos(x) | 返回x的余弦(x为弧度) |
| math.tan(x) | 返回x的正切(x为弧度) |
| math.asin(x) | 返回x的反正弦(返回值为为弧度) |
| math.acos(x) | 返回x的反余弦(返回值为为弧度) |
| math.atan(x) | 返回x的反正切(返回值为为弧度) |

### time时间模块

#### PC时间简介

* 时间简介

公元纪年是从公元 0000年1月1日0时开始的

计算机元年是从1970年1月1日0时开始的,此时时间为0,之后每过一秒时间+1

UTC 时间 (Coordinated Universal Time) 是从Greenwich时间开始计算的.

UTC 时间不会因时区问题而产生错误

DST 阳光节约时间(Daylight Saving Time)，又称夏令时, 是一个经过日照时间修正后的时间

* 时间元组

时间元组是一个9个整型元素组成的,这九个元素自前至后依次为:

四位的年(如: 1993)

月 (1-12)

日 (1-31)

时 (0-23)

分 (0-59)

秒 (0-59)

星期几 (0-6, 周一是 0)

元旦开始日 (1-366)

夏令时修正时间 (-1, 0 or 1).

* 注：

如果年份值小于100,则会自动转换为加上1900后的值

#### 模块内容

模块名: time

此模块提供了时间相关的函数，且一直可用

* 时间模块导入：

import time # 或

from time import xxx # 或

from time import \*

##### 变量

|  |  |
| --- | --- |
| 变量 | 描述 |
| time.altzone | 夏令时时间与UTC时间差(秒为单位) |
| time.daylight | 夏令时校正时间 |
| time.timezone | 本地区时间与UTC时间差(秒为单位) |
| time.tzname | 时区名字的元组， 第一个名字为未经夏令时修正的时区名,  第一个名字为经夏令时修正后的时区名 |

注：CST为中国标准时间(China Standard Time UTC+8:00)

##### 函数

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回 | 描述 |
| time.time() | 浮点数 | 返回从计算机元年至当前时间的秒数的浮点数(UTC时间为准)。时间戳 |
| time.sleep(secs) | None | 让程序按给定秒数的浮点数睡眠一段时间 |
| time.gmtime([secs]) | 元组 | 用给定秒数转换为用UTC表达的时间元组  (缺省返回当前时间元组) |
| time.asctime([tuple]) | 字符串 | 将时间元组转换为日期时间字符串 |
| time.mktime(tuple) | 浮点数  (保留1位) | 将本地日期时间元组转换为新纪元秒数时间(UTC为准) |
| time.localtime([secs]) |  | 将UTC秒数时间转换为日期元组（以本地时间为准) |

#### 应用

实例1：

格式化输出日期

>>> import time *#* 格式化成*2016-03-20 11:45:39*形式

>>> time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())

2018-06-25 08:57:34

>>> time.strptime('2018-9-23 18:45:23',"%Y-%m-%d %H:%M:%S")

time.struct\_time(tm\_year=2018, tm\_mon=9, tm\_mday=23, tm\_hour=18, tm\_min=45, tm\_sec=23, tm\_wday=6, tm\_yday=266, tm\_isdst=-1)

#### 实例练习

写一个程序，输入您的出生日期

1) 算出你已经出生多少天?

2) 算出你出生的那天是星期几

1. 写一个程序，以电子时间格式显示时间:

HH:MM:SS

(要求：不停显示当前时间即可)

2. 编写一个闹钟程序，启动时设置定时时间，到时间后打印"时间到" 然后退出

3. 编写函数fun,其功能是求下列多项式的和:

Sn = 1 + 1/1! + 1/2! + 1/3! + 1/4! +...+ 1/n!

建议用数学模块中的math.factorial来求

当n为50时，Sn的值

### string字符串模块

python string模块

import string

string.ascii\_letters

'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

string.ascii\_lowercase 小写字母

'abcdefghijklmnopqrstuvwxyz'

string.ascii\_uppercase 大写字母

'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

string.digits 数字

'0123456789'

set(string.digits) 集合

{'0', '7', '3', '9', '2', '1', '8', '4', '6', '5'}

string.hexdigits 16进制

'0123456789abcdefABCDEF'

string.octdigits 8进制

'01234567'

string.punctuation 符号

'!"#$%&\'()\*+,-./:;<=>?@[\\]^\_`{|}~'

string.printable

'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()\*+,-./:;<=>?@[\\]^\_`{|}~ \t\n\r\x0b\x0c'

string.whitespace 空白符

' \t\n\r\x0b\x0c'

应用：

生成正则字符串

>>> string.digits

'0123456789'

>>> type(string.digits)

<class 'str'>

特殊联用：

>>> t = "1231asdfa123123"

>>> t.lstrip(string.digits) #去除字符串数字开头的数字

'asdfa123123'

>>> t.lstrip("0123456789")

'asdfa123123'

### shutil文件操作

Python中复制文件：

import shutil

shutil.copyfile(fileold, filenew)

### traceback模块捕获异常

捕获所有异常，并使用traceback输出所有异常信息：

import traceback

import sys

print('-------------start-------------')

try:

print(a)

a = 1

except Exception as e:

print('\n方法1：traceback')

traceback.print\_exc()

print('\n方法2：sys')

print(sys.exc\_info())

print('\n方法3：')

print('错误信息：', e)

print('\n-------------over-------------')

### argparse解析参数

(转) argparse — 解析命令参数和选项 - liujiacai - 博客园

<https://www.cnblogs.com/liujiacai/p/9890495.html>

* 标准框架

import sys

import argparse

def fargv():

parser = argparse.ArgumentParser(description='Process introduction.')

parser.add\_argument('file\_sh', type=str,

help='输入需要运行的sh文件')

parser.add\_argument('-w', '--win', type=int,

help='-w/-win的使用帮助')

parser.add\_argument('-t', '--thread', type=int,

help='-t/--thread的帮助')

parser.add\_argument('-k', '--iskeep', action='store\_true',

default=False,

help='是否怎样怎样')

args = parser.parse\_args()

return args.file\_sh, args.win, args.thread, args.iskeep

def main():

sys.argv = ['', 'file', '--win', '10', '--thread', '4']

test = fargv()

print(test)

if \_\_name\_\_ == '\_\_main\_\_':

main()

* 示例1：

def fargv(self):

parser = argparse.ArgumentParser(description='Process some integers.')

# parser.add\_argument('integers', metavar='N', type=int, nargs='+',

# help='an integer for the accumulator')

# parser.add\_argument('--sum', dest='accumulate', action='store\_const',

# const=sum, default=max,

# help='sum the integers (default: find the max)')

# fprint(args.accumulate(args.integers))

parser.add\_argument('file\_sh', type=str,

help='输入需要运行的sh文件')

parser.add\_argument('-l', '--line', type=int,

help='line 每几行做一个运行的分割')

parser.add\_argument('-t', '--thread', type=int,

help='thread 用多少线程跑')

parser.add\_argument('-c', action='store\_true', default=False)

#python test.py -c => c是true（因为action）

#python test.py => c是false（default）

if len(sys.argv) < 5:

parser.parse\_args(['', '--help'])

sys.exit()

args = parser.parse\_args()

self.Largs = [args.file\_sh, args.line, args.thread]

fprint(self.folog, "输入参数是:\n 1、文件: %s\n 2、每个进程运行的行数: %s\n 3、进程数: %s" % tuple(self.Largs))

* 示例2

import argparse

def fargv():

parser = argparse.ArgumentParser(

formatter\_class=argparse.RawDescriptionHelpFormatter,

description='''本程序用于可视化tree文件

使用方法：

python3 itol.py fipath fopath [--option OPTION]''',

epilog="""可选参数--option <str>：

str部分需用引号引起来

--format 转换格式（默认svg，可多选中间用逗号隔开，可选：svg, eps, pdf, png, newick, nexus, phyloxml）

--display\_mode 显示模式，1 =正常，2 =圆形，3 =无根（1，2，3）

--tree\_x 水平移动像素距离（int）

--tree\_y 垂直移动像素距离（int）

--branchlength\_label\_size 分支长度标签的字体大小，像素为单位（int）

--branchlength\_display 是否显示支长（0, 1）

--bootstrap\_display 是否显示元数据值（0, 1）

...

更多参数请参考官网: https://itol.embl.de/help.cgi#bExOpt

实例：

# 实例1：转换ceshi.tre为myname.svg输出至"out/"下

python3 itol.py ceshi.tre out/myname

# 实例2：转换ceshi.tre为myname.svg、myname.pdf和myname.png输出至"out/"下，设置参数：选择输出格式，使用圈形，显示支长

python3 itol.py ceshi.tre out/myname --option "--format svg,pdf,png --display\_mode 2 --branchlength\_display 1"

""")

parser.add\_argument('fipath', type=str,

help='输入需要转换的树文件路径, 如/home/test/test.tre')

parser.add\_argument('fopath', type=str,

help='输入转换的树文件路径(无需后缀), 如out/Mdom会在目录out下生成Mdom.svg或其他指定格式')

parser.add\_argument('--option', type=str, default=None,

help='可选参数，参数部分为字符串，需要引号引起来')

args = parser.parse\_args()

Doption = dict()

if args.option:

args.option = args.option.replace('\n', ' ').replace('--', '\n--')

Ldic = [x.split() for x in args.option.strip().split('\n')]

Ldic = [[x[0].lstrip('--'), x[1]] for x in Ldic]

Doption = dict(Ldic)

# print(Doption)

Targs = (args.fipath, args.fopath, args.option)

print("--------------------------")

print("输入参数是:\n1、输入路径: %s\n2、输出路径: %s.xxx\n3、可选参数: %s" % Targs)

print("--------------------------\n")

return Targs[0], Targs[1], Doption

* 示例3：接受多个参数

import argparse

def fargv():

parser = argparse.ArgumentParser(description='merge.')

parser.add\_argument('-i', '--input', type=str, nargs='\*',

help='输出文件名字')

parser.add\_argument('-o', '--out', type=str,

help='输出文件名字')

args = parser.parse\_args()

return args.input, args.out

def fmain(finames, foname):

print(finames, foname)

# Lopen = [gzip.open(x) for x in finames]

def main():

sys.argv = ['', '-i', 'a', 'b', 'c', '-o', 'a']

fi, fo = fargv()

fmain(fi, fo)

if \_\_name\_\_ == '\_\_main\_\_':

main()

* 总结：

创建对象

import argparse

parser = argparse.ArgumentParser(description='Process some integers.')

添加参数

parser.add\_argument()

可用选项：

type int/str/

help 自定义字符串

action 'store\_true', 默认为False，当该参数存在时为Ture

default

获取参数值

args = parser.parse\_args()

Largs = [args.file\_sh, args.line, args.thread]

# 附

## PEP8编码规范

### 代码编排

1. 使用4空格缩进，不使用Tab,更不允许用Tab和空格混合缩进
2. 每行最大长度最大79字节，超过部分使用反斜杠折行
3. 类和全局函数定义间隔两个空行,类内方法定义间隔一个空行.其它地方可以不加空行。

### 文档编排

1. 其中import部分，又按标准、三方和自己编写顺序依次排放，之间空一行。
2. 不要在一句import中导入多个模块，比如不推荐import os, sys。
3. 尽可能用import XX 而不采用from XX import YY引用库,因为可能出现名字冲突。

### 空格的使用

1. 各种右括号前不用加空格
2. 逗号、冒号、分号前不要加空格。
3. 函数的左括号前不要加空格。如func(1)。
4. 序列的左括号前不要加空格。如list[2]。
5. 操作符左右各加一个空格，不要为了对齐增加空格。
6. 函数默认参数使用的赋值符左右省略空格。
7. 不要将多条语句写在同一行，尽管使用‘;’允许。
8. if/for/while语句中，即使执行语句只有一句，也必须另起一行

原则：避免不必要的空格

## 常用框架

import sys

def fmain(finame, foname):

with open(finame) as fi, open(foname, 'w') as fo:

pass

def main():

finame, foname = sys.argv[1:3]

fmain(finame, foname)

if \_\_name\_\_ == '\_\_main\_\_':

main()