# python的缺点

相对于编译型语言（c,c++），python和javascript一样属于解释型语言，速度慢。python的运行效率低，但开发效率高。

# 1. 安装

官网下载安装包，全选安装。python3的命令行交互工具**IDLE**。

VScode开发环境配置

插件：

python

pylint

快捷键：ctrl+~ 可以打开命令行攻击，然后输入python 文件名，执行当前编辑的python文件。

>>> python3 abc.py #使用python3命令执行abc.py命令

>>> pip3 install flake8 #使用pip3安装flake8

# 2. 基本数据类型

**1. Number:** 数字类型(大的分类，下面还有**整数int**，**浮点数float**, **布尔类型bool（python中bool类型属于Number类型的子类）**，**复数类型**（**数字j**）等子分类，**python不区分单精度和双精度浮点数，默认双精度**，**int也不细分short,long整型**)

>>> type(1*1) // class 'int'

>>> type(1*1.0) // class 'float'

>>> type(1/1) // class 'float' **两个整型相除结果类型为浮点数**

>>> type(1**//**1) //class 'int' **两个整型用双斜杠//相除结果类型为整型**

**单斜杠**除法自动转换**结果为浮点数**，**双斜杠**除法是**整除**不考虑余数，类似JS的Math.floor(), Math.ceil(), Math.round(), paserInt()

js: typeof('123')

**2. 10，2，8，16进制**

2进制（满2进1）：0,1, 10

8进制（满8进1）：0,1,2,3,4,5,6,7, 10

10进制（满10进1）：0,1,2,3,4,5,6,7,8,9, 10

16进制（满16进1）0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, 10

**进制类型，数据的记录方式**

2进制：0，1，0，1，0，1 1+1= 10

8进制：0，1，2，3，4，5，6，7，10，11

10进制：0，1，2，3，4，5，6，7，8，9，10

16进制：0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10

python进制表示方法及转换

2进制(**binary**)：**0b**10, 2进制转10进制，2进制的10，在IDLE中会自动转换成10进制数，结果为2

8进制(**octal**)：**0o**10, 8进制转10进制，8进制的10，在IDLE中会自动转换成10进制数，8

16进制(**hex**)：**0x**10, 16进制转10进制，16进制的10，在IDLE中会自动转换成10进制数，16

**bin() 转换成2进制数方法**

bin(10), bin(0x12), bin(0o10)

**int() 转换成10进制数方法**

int('123') //123

int(123) // 123

int(123.01) //123

**int('123.01') //错误**

**hex() 转换成16进制方法**

**oct() 转换成8进制方法**

js:parseInt('123'), parseFloat('123.2')

**float() 转成浮点数**

**3.bool:** 布尔类型 js:Boolean

>>> type(**True**) //class 'bool'

>>> type(**False**) //class 'bool'

**>>>int(True) // 1**

**>>>int(False) // 0**

**>>>bool(1) //True**

**>>>bool(0) //False**

**>>>bool(2) //True**

**>>>bool(0b01) //Ture**

**>>>bool(0b0) //False**

**>>>bool('abc') //True**

**>>>bool('') //False**

**>>>bool([1,2,3]) //True 列表**

**>>>bool([]) //False 这里不同于JS的空数组,JS空数组Boolean([])为true**

**>>>bool({1,2,3}) //True 元组**

**>>>bool({}) //False**

**>>>bool(None) //False**

js:Boolean('123')

js:true,flase python:**True,False**

**js数组，python列表；js:对象，python元组**

**4.str:** 字符串类型

单引号 'this is a apple'

双引号 " it's a apple "

**三引号表示多行字符串，单或双都可以**

"""

hello world

hello world

"""

//返回hello world\nhello world

**print**('hello world**\n**hello world') print函数加**转义字符**实际也会显示成两行

print(**r**'c:\na1\na2') //字符串前面加**print(r'String')**表示的是原始字符串（所见即所得）

转义字符：换行\n, 单引号\', 制表符\t

js:console.log('1234')

**str('1') 转成字符串**

**5.字符串运算**

'hello world'[0] //取字符串第0个字符

'hello world'[0:4] //取字符串0到4个字符串，不包含第4个，类似js:substring(0,4)

'hello world'[**6:** ] //从第6位截取到末尾，返回world。省略即开始或末尾

'hello world'[**:-4**] //从0位开始**往后截取**到负4位，返回'hello w'。省略即开始或末尾

'hello world'[-4:] //从倒数第4位开始**往后截取**，返回'orld'

**python中字符串也是有序数据类型**

string是不可变的序列，常用操作方法

**.replace('a','b', n)** #将字符串中的a替换成b, 修改n个匹配的字符

**.translate()**#将字符串中的每个字符根据指定的映射表进行转换。比replace方法更灵活。

**.split(',')** #将字符串以','**分割转换成list**

js:'abcde'.split('')

**.join(str1)** #将List**转换成字符串** **'-'.join(list)**

js:['a','b','c','d'].join('-')

**.splitlines() #**将字符串按行分割，并返回一个包含每行作为元素的列表

**.center(),.ljust(),.rjust()** #使用指定的字符填充出指定长度的字符串，原字符串居中或左对齐或右对齐

**.zfill()** #用0左填充出指定长度的字符串

**.startwith('a')** #是否以a开头

**.endwith('a')** #是否以a结尾

**.upper()** #全大写

**.lower()** #全小写

**.capitalize()** #首字母大写

**.title()** #每个单词首字母大写

**.swapcase()** #大小写互换

**.isnumberic()** #是否全为数字

**.isalpha()** #字符是否全为字母

**.strip()** #删除字符串头尾指定字符（默认为空格），返回新字符串

**.rstrip()** #删除字符串右侧空格

**.lstrip()** #删除字符串左侧空格

**.find()** #查找指定内容在字符串中出现的索引号，找不到返回-1

**.expandtabs()** #将字符串中的制表符\t转换城**指定数量**的空格字符

**\ 转义字符**，\n 换行, \\ 斜线

**.format()方法格式化字符串，{n}中可以指定传入参数的索引号，可以指定变量名称**

```
print("{}{}{}".format('a','b','c')) #abc
print("{0}{0}{2}{2}".format('a','b','c')) #aacc {}指定索引号占位符
print("hello {}".format("world")) #hello world
print("hello {var}".format(var="hello")) #hello hello {}中指定变量名var
print("number is {:d}".format(2)) #number is 2 指定整型数值占位符
print("string is {:s}".format('str1')) #string is str1 指定字符串占位符
print("float number is {:.2f}".format(12.50)) #float number is 12.50 指定两位小数浮点数占位
print("float number is {:.0f}".format(12.50)) #float number is 12 指定小数四舍五入占位符
print("secince number is {:.4E}".format(121234123123)) #secince number is 1.2123E+11 指定
print("number is {:.2f}%".format(45.23)) #number is 45.23% 加%自动转换成百分比
```

## f-string() 模板字符串

```
>>> num = 2
>>> f"I have {num} apples"
>>> f"They have {2+5*2} apples"
'They have 12 apples'
>>> import math
>>> f"Π的值为{math.pi}"
'Π的值为3.141592653589793'
```

## Unicode字符串和字符串

Unicode字符串通常用于表示包含非ASCII字符的字符串，比如中文或者特殊符号。Unicode字符串表示的是字符串本身，而不是它们的编码形式

```
print(u'你好, python')
```

## Unicode字符串和字节串

## 表示字符串的二进制数格式

```
print(b'hello world')
byte_string = b'hello world' #生成字节串
string2 = byte_string.decode('utf-8') #字节串转字符串
byte_string2 = string2.encode('utf-8') #字符串转字节串
```

**早期的格式化方法，使用%，目前已被format函数，f-string替代。**

**格式化字符串，用变量替换，中间有空格**

**%s** 字符串占位符

**%i** 整型数值占位符

**% i** 整型带空格的占位符，空格会显示出来

**% +i** 整型带加号和空格的占位符，但只会显示+,空格不显示

**%f** 浮点型数值占位符

**%.0f** 小数部分四舍五入

**%.2f** 浮点型数值占位符，保留两位小数

**%.2E** 科学计数法占位符，显示两位小数科学计数法，%.4E 四位小数科学计数法

**%g** 如果数据很大时自动使用科学计数法

**多个占位符替换时用括号包裹**

```
x1="abc%s a1%i a2%f"
```

```
x2="aaa"
x3=2
x4=12.5
print(x1 %(x2,x3,x4)) #空格
# abcaaa a12 a212.500000
```

**字符编码**

计算机以统一的方式存储和处理字符，无论是字母、数字、符号还是特殊字符。**字符编码通常是将字符集中每个字符映射为一个或多个字节（binary digits）**

常见的字符编码包括**ASCII 美国信息交换标准代码**，包含128个字符，英文大写小写，数字，标点，控制字符。每个字符对应一个字节长度（8bit）。

**unicode 统一字符编码**，将世界各种语言的**每个字符定义一个唯一的编码**，以满足**跨语言、跨平台**的文本信息转换。unicode通常**使用一个特定的编码方案**来实现，最常见的unicode编码方案就是**UTF-8。**常见的\u是unicode转义序列。

**UTF-8** 可变长度编码方式，UTF-8 就是在互联网上使用最广的**一种 Unicode 的实现方式，**占用**1-4个字节。**

**UTF-16占用2-4个字节**

**UTF-32始终占用4个字节。**

**GBK**

编码

```
text='你好，123abcde'
encode_text = text.encode('utf-8') --将字符串用utf-8格式编码
decode_text = encode_text.decode('utf-8') --用utf-8格式解码字符串
text2 = text1.encode('ascii', error='ignore') --忽略无法处理的字符
text3 = text1.encode('ascii', error='replace') --用?代替无法处理的字符
with codecs.open('./text.txt', encoding='utf-8') as file --codecs包读取text.txt文件，用ut
```

**6.python中的有序数据类型-list列表类型：** js:JS中的Array

List: **[1,2,2,1,'hello','abc',12.5, [2,3,22]]**

list是**有序的，而且可以嵌套list, List可以通过赋值改变成员的值**

**【1】访问列表**

List[0] 返回列表的第0个元素。如果索引号是单个数字则返回结果是List成员的字符串

如果索引号是一个区间值List**[0:4]** 返回列表的第0到第4个元素(不含第4个)组成的**新列表**。如果索引号是区间值则返回的结果是List成员组成的新列表。

【2】连接两个列表

[1,2,3,4] + ['a','a1','aa'] 返回 [1,2,3,4,'a','a1','aa'] JS:js中的concat

【3】重复列表

[1,2] * 3 返回 [1,2,1,2,1,2] 不同于JS

**列表List的常用方法**

**.append()**添加元素，改变原列表(如果参数是列表则会将参数当成一个元素插入)

**.extend()**添加元素，改变原列表(如果参数是列表则会挨个插入)

**+**不改变原列表但生成一个新的列表

**.remove(2)**删除指定的元素，第一个匹配的元素被删除

**.clear()**清空列表

**del list[0]** del是语句，删除列表list指定索引号的元素，参数可以指定区间，可以指定步长

**del list[1:6:3]** del是语句，删除列表list从第1个开始到第6个，间隔3个删除

**.insert(2,'hello')** 在指定位置插入元素

**tuple(list1)** tuple**函数**，转换list为tuple

**tuple**(range(10)) tuple**函数**，转换range为tuple

**list(tuple1)** list**函数**，转换tuple为list

**list(range(10))** list**函数**，转换range为list

**.sort()**列表的排序，默认升序，**不会生成新列表**仅修改原列表

**sorted(x)**排序并复制，对x做排序**生成新的**经过排序后的列表

```
x1=[2,3,4,5,6]
x2=['d','c','a','b']
x1.sort(reverse=True) #reverse反向排列，[6,5,4,3,2]
x2.sort() #默认升序排列 ['a','b','c','d']

x1=[6,7,8,9]
x2=sorted(x1, reverse=True) # sorted(x1) 会生成一个经过排序后的新列表
x3=x1.sort(reverse=True) # .sort()方法排序修改原列表，不会生成新列表，所以x3是none
print(x1,x2,x3) # [9, 8, 7, 6] [9, 8, 7, 6] None
```

**7.python中的有序数据类型-tuple元组类型：**

Tuple: (1,2,3,4,'hello') 元组**是不可变的List**,即元组的**元素不可以赋值**。列表有的方法元组也有。

【1】定义只有一个元素的元组type(**(1, )**)，返回tuple。如果不加逗号是(1)则python编译器默认(1)是求值运算。

【2】空的元组type(()) 返回tuple，定义空的元组 type(tuple())

**序列包含可变序列List, 和不可变序列Tuple，字符串String**，序列的特点类似JS中复杂类型的变量在内存中的特点，属于值地址引用。栈内存中存放的是堆内存中的内存地址。

**字符串、列表、元组共有的方法**

【1】切片[0:10:2] #切片方法的第**三个参数是步长**

【2】连接[1]+[2]

【3】判断元素是否**在序列**内 2 **in** [1,2,3] 返回True， JS: [1,2,3].includes(2)

判断是否**不在序列**内 2 **not in** [1,2,3] 返回False

【4】序列的长度 **len(**[1,2,3]**)**，len('hello world')，len((2,))

【5】序列的最大值,最小值 **max(**[1,2,3]**)**，**min(**(1,2)**)**，

**max(**'hello world'**)** 字符串最大值、最小值是字符串**ASCII编码的最大值，最小值**

ord('param') 转换参数为ASCII编码，参数必须为字符

【6】序列的**方法**使用**.()**调用

```
x=[1,2,3,2,8,2,7,5,2]
print(x.index(5)) #序列的方法，指定元素，第一个匹配上的，在序列中的序号
print(x.count(2)) #序列的方法，指定元素，第一个匹配上的在序列中出现的次数
```

**序列的切片和步长（适用list, tuple, string）**

**x[2:-1:2]**

```
x=[1,2,3,4,5,6,7,8,9]
print(x[2:5]) #返回索引2-5的值，不包含第5个值
print(x[:5]) #从第0个值截至到第5位，不包含第5个值
print(x[2:]) #从第2个开始到最后一个值
print(x[:-1]) #从第0个值到最后一个值

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

x=[1,2,3,4,5,6,7,8,9,'hello','world']
print(x[2:9:2]) #第2个到第9个不包含9之间的，每两个间隔取值
print(x[1:10:3]) #第1个到第10个不包含10之间的，每3个间隔取值
```

```
15
16 [3, 5, 7, 9]
17 [2, 5, 8]
```

**使用.copy()方法可浅拷贝成两个独立的序列**

```
1 x1=[2,3,4]
2 x2=x1 #值地址引用，修改原列表，x1,x2列表都改变
3 x2[0]=0
4 print(x1,x2) #[0, 3, 4] [0, 3, 4]
5
6 y1=[4,5,6]
7 y2=y1.copy() #.copy()方法浅拷贝，值复制，y1,y2是两个不同的列表
8 y2[0]=0
9 print(y1,y2) #[4, 5, 6] [0, 5, 6]
```

**8. python中的无序数据类型-set集合类型**

集合是**无序的**数据类型，无法通过下标序列号获取元素。集合的元素是**没有重复**的类似JS:js中的set()无重复，但js中的set是有序的。

set: {1,2,3,4,'123',True}，定义空的集合 type(**set()**)

获取集合的长度 **len(**{1,2,3}**)**

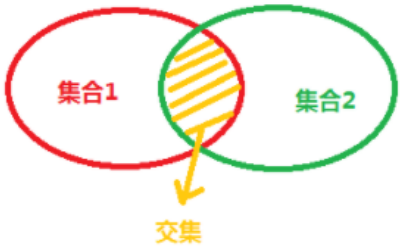
判断是否在集合内 1 **in** {1,2,3}

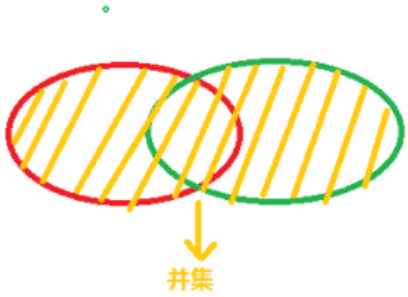
判断是否不在集合内 1 **not in** {1,2,3}

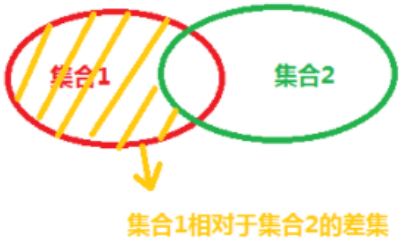
求两个集合的**差集，用减法，**{1,2,3,4,5,6} **-** {3,4} //**返回{1,2,5,6}**

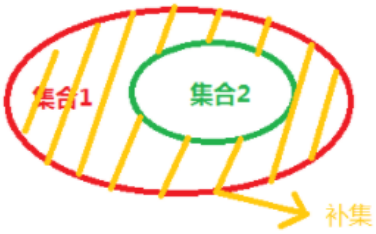
求两个集合的**交集∩，共有的元素，用&，**{1,2,3,4,5,6} **&** {3,4} //**返回{3,4}**

求两个集合的**并集∪，用 |，**{1,2,3,4,5,6} **|** {5,6,7} //**返回{1,2,3,4,5,6,7}**

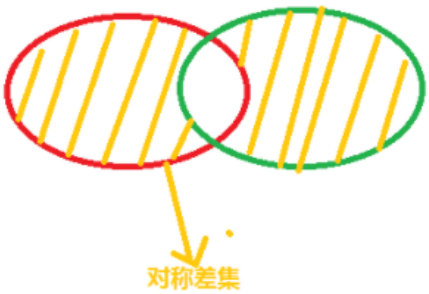



集合1是集合2的超集
集合2是集合1的子集




**9. python中的无序数据类型-dict 字典**

dict: { 'a':100, 'b':'hello' } dict是**无序列表** JS:js中的Object对象，**字典中不能有重复的key**，存在重复的key时，后添加的key会替代先添加的key。字典的键**key可以是字符串也可以是数值**,{**1**:'a','**1**':'b'} ，这里两个key1是不同的key

**dict的key 必须是不可变的类型**，比如int, str,tuple，list可变所以不能作为key

获取key对应的value值 {'a':1,'b':2}**['a']**

定义空的字典type(**dict()**)

字典的常用方法：

dic1**.update**(dic2) #合并字典，**会改变dic1**

dic3 = dic1**.copy()** #浅复制一个字典

a1 = **len(**dic3**)** #返回字典长度

'a' **in** dic1 #判断key是否在某个字典中

lst1 = [['a',1],['b',2]] dict1 = **dict(**lst1**)** #将嵌套数组lst1转成key,value格式

#lst1 dict([['a',1],['b',2]]) 或者 dict([('a',1),('b',2)])

keys1=['a','b','c'] dic1 = dict**.fromkeys(**keys1,0**)** #生成值为空只有键名的字典，fromkeys方法的第二个参数是，填充生成字典的值

dic1**.get(**'key'**)** #获取字典key对应的value
dic1**.keys()** #获取dic1的所有的key
dic1**.values()** #获取dic1的所有的value
dic1**.items()** #获取dic1的所有的key,value构成的成员，返回结果的格式是**[(**'key','value'**)]**
遍历字典

```
for keys in dic1: #遍历dic1的keys
    print(keys)

for values in dic1.values(): #遍历dic1的values
    print(values)

for (k,v) in dic1.items(): #遍历dic1的key,value 键值对
    print(k,v)
```

## 3. pyton变量

值类型 (**不可变**)：**int，str，tuple**，即便是str[0]='aa' 这种对字符串的操作也不能改变字符串str原来的值。js:'string'[0]=1 无效
引用类型（**可变**）：**list，set，dict** 引用类型的成员的值可以改变。**[1,2][0]='a'**
**id(var) 可以显示一个变量在内存中的地址**

tuple类型不能修改成员，也不能追加成员。tuple的使用场景主要是描述保持不变的数组类型的数据。但在**tuple**类型中，**嵌入了可变数据类型**的成员，则这个**可变类型的成员是可以被修改的**

```
>>>a = (1,2,3,[1,2,4])
>>>a[3][2]='4'
#返回(1,2,3,[1,2,'4'])
>>>a[3].append('d')
>>>#返回(1,2,3,[1,2,'4','d'])
```

user_account **= input()** #命令行输入时，使用input()函数

python中没有常量概念，**只有形式上的常量，变量名使用大写，**python文件的顶部，一般有一段注释，用来说明该文件模块的内容。

## 4.python运算符

算术运算符： +， - ，*，/，**， //, %

赋值运算符： +=，-=， *=， /=， %=， **=，//=

比较运算符：==，!=， >=， <=， >， <

逻辑运算符：and， or，not (js: &&, ||, !)

成员运算符： in，not in (js 没有not in 只有in)

身份运算符：is，is not (js Object.is(param1, param2) 判断两个值是否完全相等即===)

位运算符：&， |， ^，>>，<<，~ （**按位** &,|,^,>>,<<,~）

python中没有a ++, a-- 这种自增自减运算符

**比较运算符**在比较**字符串**的时候会调用**ord()**函数，将两个待比较的字符串各自分拆后逐个字符转成ASCII编码来比较。比如判断abc<abe比较时，会先比较a,a，在比较b,b，最后比较c,e

**比较运算符**在比较**列表**的时候会将两个列表的成员逐一比较，比如判断[1,2,3]<[2,3,4]比较时，会先比较1<2，再比较2<3，最后比较3<4。元组的比较也是类似

逻辑运算符可以比较bool型，数值型，字符串。在python中做逻辑运算时会自动转换数据类型：

int、float类型中的0会被转换为False，大于0或小于0的会被转换为True

str类型中，**空字符串''**会被转换成False，注意不是' '(**里面有空格不是空字符串**)

list,tuple,set,dict 类型中，空的列表[],空的元祖(),空的集合{},空的字典{'a':1} 会被转换成False. JS:这里不同于js中的空数组，js空数据会被转换成true

身份运算符:**is, not is**, 判断的是两个变量的**值和值的值类型(内存地址)**是否相同

a=1 b=1.0

判断a **==** b时返回True，比较运算符判断的是两个值是否相等，不看值类型(内存地址)。

判断a **is** b 的时候返回False，身份运算符判断值和值的类型(内存地址)是否相等

python中判断一个变量是否是某个类型的实例，使用

**isinstance： isinstance('a', str) #判断'a'是否是str类型**

**isinstance: isinstance(param1,(str, tuple, set)) #判断param1是否是str,tuple,set中的任意一种**

**JS: 判断对象是否是某个构造函数（类）的实例，str instanceof String // 这里str 必须是str = new String('a')的结果，这样才是String函数生成的实例。**

运算符优先级

赋值运算是右结合，从等号右边往左执行。如果表达式优先级顺序不太清晰，使用括号显式的标记运算优先级。

**算术运算符>比较运算符>逻辑运算符**

| | | |
|---|---|---|
| 1 | `**` | 指数(次幂)运算 |
| 2 | `~` `+` `-` | 补码，一元加减(最后两个的方法名称是 `+@` 和 `-@` ) |
| 3 | `*` `/` `%` `//` | 乘法，除法，模数和地板除 |
| 4 | `+` `-` | |
| 5 | `>>` `<<` | 向右和向左位移 |
| 6 | `&` | 按位与 |
| 7 | `^` `\|` | 按位异或和常规的“ `OR` ” |
| 8 | `<=` `<` `>` `>=` | 比较运算符 |
| 9 | `<>` `==` `!=` | 等于运算符 |
| 10 | `=` `%=` `/=` `//=` `-=` `+=` `*=` `**=` | 赋值运算符 |
| 11 | `is` `is not` | 身份运算符 |
| 12 | `in` `not in` | 成员运算符 |
| 13 | `not` `or` `and` | 逻辑运算符 |

算术运算符

a+b, a-b, a*b, a/b

**a%b #取模，求除法余数**

**a//b #除法取整数**

**a**b #幂运算，a的b次方**

比较运算符

a==b, a>=b, a!=b, a>b, a<b

逻辑运算符

True **and** False, True **or** False**, not** True

**成员运算符 in, not in**

lst = [2,3,4]

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

print(1 in lst) #返回true
print('c' in dis) #返回false

## 5.python流程控制语句

python编码规范：
行末尾不强制加分号结尾。不需要用{}包裹代码块，python使用缩进表示代码块
1. 条件控制语句 if-else
**pass** 语句在python中用来作占位，使语句能顺利执行下去不出错

```
if condition_test:
    pass
elif condition_test2:
    pass
else:
    pass
```

2.循环 **while else**

```
while condition:
    pass1
else:
    pass2 #else是condition条件为false时执行的代码块
```

示例：

```
count =1
while count <=10:
    count +=1
    print(count)
else:
    print('EOF')
```

while循环的使用场景是，设定condition为条件判断，condition条件为true时就一直执行语句块，条件判断为**false**时就执行**else**后面的语句块。while多用在递归的场景中。

3.循环 **for else**
for 循环用来遍历/循环 序列、集合、字典

for **target_list in expression_list**:

    pass1

else:

    pass2

**else**语句块在for循环**正常结束**（**非break终止**）后执行。

```
a = [['a','b','c'],(1,2,3)]
for i in a:
    print(i,end='--')
    for y in i:
      print(y,end="|")
else:
    print('EOF iterate done')
```

print(a, **end = '|'**)

print 函数使用**end参数**可以将需要打印的元素横向间隔排列。

4.**循环控制**，终止循环 使用**break、continue**

break 终止循环跳出整个循环，continue 跳过本次循环但会继续执行下一次循环。**break**会**中断**for循环遍历，使for循环结束后的else不执行，但**在循环嵌套的场景下**，**break只会中断所在层级的循环**，对**外层循环不会中断**

```
a = [1,2,3]
for x in a:
  if (x == 2):
    break
  print(x, end='|')
else:
    print('EOF')
# x==2时终止循环，整个过程只会打印出1|, break语句终止循环后
```

```
a = [1,2,3]
for x in a:
  if (x == 2):
    continue
  print(x, end='|')
else:
    print('EOF')
```

```
# x==2时跳过本次循环，整个过程会打印出1|3|EOF
```

for 循环的范围限定使用range()函数

```
// js文件
let a = [1,2,3,4,5,6,7,8,9,10]
for (let i =0;i <= a.length; i ++) {
  console.log(i+'|')
}
```

```
for i in (range(0,10)):
    print(i)
# 打印出0,1,2,3,4,5,6,7,8,9 共10个数字
```

```
for i in (range(0,10,2)):
print(i)
#打印出0，2，4，6，8 共5个数字，range()函数的第三个参数表示步长
```

**range列表和生成器**

**range(num1, num2, length)**

range范围num1是起始数值，num2是终止数值但不包含num2，length是步长，步长可以为负数表示递减。

**range生成一个整数区间,不包含末尾一位，**类型为**<class range>，range不是一个列表但可以使用索引访问值，使用括号表示range，逗号表示范围,第三个参数代表步长。使用list函数可以生成由range指定区间范围的list**

```
for x in range(10,2,-2):
print(x)
#结果打印出10，8，6，4
```

```python
x= range(0,5)
y = list(x)
print(y)
#返回[0, 1, 2, 3, 4]
```

## 6.python 工程组织结构

**包、模块、类**

python项目的组织结构，最顶级的层级**包**层级（类似文件夹），然后是**模块**层级（类似文件）、最后是写在模块文件中的**类**。**包的文件夹结构是可以嵌套的。**

**__init__.py是一个包被导入的时候自动执行的文件，当导入包下面某个模块的变量时__init__.py文件也会自动执行。**文件夹下如果有**__init__.py**文件，则代表该**文件夹是一个包**。如果一个文件夹下没有__init__.py则python认为该文件夹是普通文件夹。**__init__.py本质上就是一个模块文件**，__init__.py文件可以写代码，**也可以不写只是用来标注一个包。__init__.py模块的名称就是包的名称，比如seven目录下有__init__.py，则__init__.py模块的名称就是seven。**

一个 .py 文件可以称为模块，包含了 __init__.py 文件的称为包。

不同包下同名的模块，使用命名空间来区分

```python
seven.c4
six.c4
#seven,six 是包名称，c4是模块名称
```

导入模块的方法

**【1】 import**

```python
import module_name # 导入同级目录下的某个模块
import a1.module_name #导入同级目录下a1目录下面的module_name模块
import t.b.c7 as m #导入t/b下的c7模块，并使用m表示，调用c7模块时使用m.a调用
```

python的导入无法像js语言一样导入js模块的某个变量或函数，python只能导入整个模块文件，然后使用命名空间来访问模块下的某个变量。

```javascript
// js文件
// require
```

```javascript
 const module = require('module')
```

```
// exports
export.fs = fs
module.exports = fs

// import
 import fs  from 'fs';
 import { newFs as fs } from 'fs';  // ES6语法，将fs重命名为newFs，命名冲突时常用 import {
import fs, { part } from fs;
// export
export default fs;
export const fs;
export function part;
export { part1, part2 };
export * from 'fs';
```

**【2】from import** #导入某个包/模块下的局部变量或局部函数

from t.c7 import a # 导入**t.c7模块**下的**a**变量，调用a变量时可直接使用**.方法**调用

from t import c7# 导入t模块下的c7模块，调用a变量时使用**c7.a**调用

from t.c7 import a, b, c #导入t.c7模块下的a,b,c三个变量

from t.c7 import (a, b

c) #导入t.c7模块下的a,b,c三个变量,c变量可以在括号内换行

**import方法导入的是模块，from import 导入的可以是某个变量也可以是某个模块**

from t.c7 import * #导入t模块下c7模块下的所有变量

【3】 __init__.py文件定义导入文件

导入一个包时,**__init__.py文件会自动执行**，使用__init__.py文件导入变量使用场景

在**包t**下的**__init__.py模块文件中编写需要多次重复导入的模块**，比如

```
#__init__.py文件
import sys
import datetime
import io
```

在其他目标文件中直接导入包t即可导入包t下__init__.py模块文件中所导入的变量/库

```
import t
print(t.sys)
```

导入模块注意事项：

1.包和模块不会被重复导入，多次引用同一个包或模块，但其导入过程只会执行一次。
2.当导入某个模块时，该模块文件会自动执行一次。
3.避免循环导入

**相对导入、绝对导入**
import package2.package4.m2 #绝对路径导入，根据文件路径导入某个模块
from .m3 import m #相对路径导入，导入相对于当前路径下的m3模块
from ..m4 import m #相对路径导入，导入相对于当前路径，上一级目录下的m4模块
注意，**相对路径导入时不能超过当前模块的顶级包路径**否则出现以下错误
**attempted relative import beyond top-level package**
**程序的入口文件不能使用相对路径导入模块，只能使用绝对路径导入或者将入口文件当成模块调用(加-m参数)**

添加绝对路径后再导入包

```
import sys
sys.path.append('c:\\users\\desktop') #添加一个绝对路径
import packageName1 #添加绝对路径后可以直接导入绝对路径下的包
packageName.fn1('hello world')
```

# 7.模块的内置变量

python c15.py #python 将c15.py**当作程序入口**文件直接执行
python **-m** seven**.**c15 #python 将seven目录下的c15.py文件**当作模块调用**（.py文件**当作模块调用时必须有包的环境(加.)，seven目录这时候就是包**），这种方式不同于python seven\c15.py#仍然是当入口文件调用
**main.py入口文件不是顶级包，只是入口文件**
info**=dir()**
print(info)
**带双下划线的是python的内置变量**
__annotations__
__builtins__

__cached__
**__doc__** #当前模块的注释信息
**__file__** #当前模块文件相对于**执行入口文件所在目录的**文件路径
__loader__ #引用了本模块的加载器对象，即该模块的上下文是由这个加载器加载到内存中的
**__name__** #当前模块的完整名称，包含了命名空间，比如t.c9
**__package__** #当前模块**所在的包**名称比如t，**相对于执行入口文件所在的目录**
__spec__
**__init__.py** #python检测到一个目录下存在__init__.py文件时，python就会把它当成一个模块(module)
__all__ # 在某个模块文件中指定可以被导出的变量，或在__init__中指定可以被导出的模块
**__closure__[0].cell_contents #查看对象的闭包变量**

当一个模块是程序的**入口文件**时即**执行python abc.py**时，该模块文件的 **__name__** 会显示为**__main__**，**__file__**文件路径会显示为**当前模块的文件名**（没有路径），

**__name__的经典应用**
**#make a script both importable and executable** 让一个脚本既可被当作模块导入，也可被解析执行

```
#t1.py文件
if __name__ == '__main__':
    pass1
pass2
```

#当t1.py作为**程序入口**直接**执行**的时候会执行**pass1**，当t1.py**被其他模块导入时**t1.py只会执行**pass2**，不执行pass1

# 8.python函数

1.函数具备功能性
2.函数隐藏细节
3.函数可复用，避免重复编写代码

定义函数

```
def fn(a, b=2):
    pass1
```

```
    return val
```

#参数**b的默认值是2**
调用函数fn(a)

## 可变参数

```
def fn(*parm):
    pass1
```

#参数 ***parm**，用来**收敛传入的多个参数**，是**tuple类型**

## 函数的多个返回值

```
def damage(ski1, ski2):
    val1 = ski1*2
    val2 = ski2*3
    return va11,val2
```

**返回多个值**，使用**逗号分隔开**，逗号分隔开的返回值其**类型是元组tuple**
**获取**函数的**多个返回值**的时候，用逗号分隔声明变量**解构赋值**方法，**接收结果**的变量和函数内部**返回值的先后顺序**有关

```
resule1, resule2 = damage(2,3)
```

## 声明多个变量

```
a,b,c=1,2,3 #a=1, b=2, c=3
a,b,c=[1,2,3] #a=1, b=2, c=3
a,b,c=(1,2,3) #a=1, b=2, c=3
a = 1,2,3 #a=(1,2,3)
```

a,b,c = 1,2,3 **#js let a=1,b=2,c=3 或者 let {a,b,c} = {a:1,b:2,c:3}**
a,b,c = d #d是序列类型（tuple,list），此时a,b,c会序列解包
a = 1, 2, 3 #a赋值后是tuple类型
**链式赋值**
a=b=c=1 #a=1,b=1,c=1 **#js let a=b=c=1**

**函数参数**

1. 必须参数。调用时传入的实际参数必须和形式参数个数一致且顺序一致

```
def add(x,y):
    return x + y
    pass1
```

1. 关键字参数，调用的时候可指定实参匹配哪个形参，**不用考虑传参的顺序。**前提是要知道函数定义时内部的形式参数名称。

js?

```
    c = add(x=2, y=3)
```

1. 默认参数

```
def add(x=1,y):
    return x + y
```

调用时 add(2) 传入的是，第一个参数**没传的时候**默认使用定义函数时的默认参数x=1，传入的是y=2

调用时 add(3,4) 传入的参数在函数内部是x=3　y=4

2.可变参数，收敛传入的参数，基于元组的可变参数，传入参数转成tuple格式

```
def add(*param):
    print(param)
```

调用时 add(1,2,3,4,5)，打印param是(1,2,3,4,5) 会**自动将参数列表转换成成tuple，函数内部定义的可变参数param会自动收敛传入参数，并转成tuple**

调用时 add(***(1,2,3,4,5)**) 使用*调用将传入的参数tuple平铺逐一匹配到函数内部的形式参数位置

js?

3.关键字可变参数，收敛传入的关键字参数，基于字典的可变参数，传入参数转成dict格式

```
def add(**param):
    print(param)
```

调用函数时 add(x='1', y ='2', z='3') #**指定关键字参数调用时**，函数内部定义的可变参数param会自动收敛传入参数，并转成dict {'x':1,'y':2,'z':3}

或者以传入字典的方式调用

add(****{'x':1, 'y':2，z:'3' }**) #传入字典就需要用**平铺

js? 解构赋值

for key, value in target.items():

注意：**声明函数时，必须参数靠前**，有默认值的默认参数尽量放后面。

**函数的局部变量和全局变量**

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全局上下文环境访问。函数内**声明全局变量**

```
def fn():
    global a
```

a= 'hello world'

当执行fn()时，函数会在全局环境声明全局变量a，变量的值是hello world

**变量作用域**

函数内部定义的变量（局部变量）作用域在函数内部，函数外部无法引用函数内部变量（闭包除外，还可以使用global关键字变量在函数内部定义全局变量）。但函数内部可以引用全局变量即函数外部定义的变量，或者嵌套函数，其内部变量可以引用外层变量。

c=10

demo():

print(c) #打印出10

注意：python只有全局作用域、函数作用域的概念，python没有代码块级的块作用域的概念(for循环，while循环，if..else等非函数不能形成独立的块作用域)，所以python for循环中声明的变量，在for循环外部是可以访问的，不同于js

global 在函数内部定义全局变量

def demo():

global c

c = 2

执行demo() 函数后在其内部会执行声明全局变量

print(c) #在函数外部就可以访问到函数内部使用global定义的全局变量

**某个模块中函数内部global关键字定义的全局变量也是可以被其他模块导入使用的**

# 9.python面向对象

类：用class定义，类变量**首字母大写，驼峰命**名法，类的最**基本作用是用来封装代码**。类中定义的函数即方法。类就是对现实世界的抽象定义。

```
class Student():
    name = ''
    age = 0
    #构造函数，初始化实例
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def fn1(self):
        print(self.name, self.age)
#实例化类
stu1 = Student('zhangsan',18)
#调用实例的方法fn
stu1.fn1()
```

类的构造函数和方法，必须传入self参数。类的实例化**不需要加new**关键字

导入类

```
from c1 import Student()
stu = Student('name',10)
stu.printFile()
```

**类和对象的关系**

**类**是现实世界或思维世界中的实体在**计算机中的一种抽象反映**，将抽象出的实体，其**属性**和对属性的操作即**方法**封装在一起。简单理解，类抽象出**实体的特征与行为，类基本可以理解为模板**

对象，即通过**类实例化出的一个实例**。

class Studen():

name= 'default value'

age= 0

def **__init__(self, name, age)**: #python类的构造函数，**js的constructor函数**

**self.name**=name #通过构造函数的执行来初始化对象的属性

**self.age**=age #通过构造函数的执行来初始化对象的属性

**实例化一个实例时，类的构造函数会自动执行。构造函数或实例方法内的self始终指向实例。**

**JS类的定义方法：**

```
class Student {
    consotructor(name, age) { //js构造函数
        this.name = name
        this.age = age
    }
    printFile() { //实例方法
        console.log(this.name)
    }
    static classMethod(){ //类class的静态方法
        console.log(this) //静态方法中this指向的是类而不是实例
    }
    static myClass ='grade1' //js的静态属性写法
}
```

**类变量、实例变量**

类变量即**类自身的静态属性**，类变量要和类关联在一起，体现出类的意义。

实例变量即**类实例化后的属性**，实例化后生成对象的属性。实例变量和对象关联在一起

```
class Student():
    sum=0 #类的变量，与类相关联，JS类的变量用static静态属性表示
def __init__(self, name, age): #self只是占位符，可以用this替代
    self.name=name #实例变量
    self.age=age #实例变量
def fn1(self): # 定义函数的时候参数传入self，表示该函数是实例的方法
    print(self.name)
```

**类与对象的变量查找顺序**

stu1 = Student()

stu1.**__dict__** #**对象（包含类或者实例）的__dict__对象中保存了当前对象的所有变量**，类似js

stu1 = new Student('name1',100); stu1.name

注意：类的静态属性（类变量）和实例属性（实例变量）的查找顺序，如果查找一个实例的变量，首先在实例属性中查找，如果找不到就去类的静态属性中查找，如果类中找不到就会去类的父类中查找。**类似JS的原型对象**

**在实例方法中访问实例变量与类变量**

实例方法中访问实例变量，**self.val**

实例方法中访问类变量：**ClassName.val 或者 self.__class__.val**

**类方法即类的静态方法**

**@classmethod #定义类的静态方法时增加@classmethod装饰器**

def fn(**cls**): #传入的参数**cls表示class**，可以使用其他变量代替

    cls.val #访问类的属性，和实例方法中**self.__class__**.val 等价

类的方法中访问类的属性，**cls.val** 等价与self.__class__.val 等价与 ClassName.val

调用类方法ClassName.fn()

通过类的实例调用类的方法(不建议这么调用没意义)：

ins1 = ClassName()

ins1.__class__.fn()

**成员的可见性**

对类的变量即类的静态属性的操作，放到类的静态方法，或者实例方法中去操作。不建议在外部直接操作类的属性比如 ClassName.val = 1 这种操作不建议

类的私有方法，在一个类中定义的方法(类方法或者实例方法)名称前面加"__ "表示该方法为类的私有方法，私有方法在类的外部无法调用，比如ClassName.fn()。

私有变量，在变量名前加"__" 该变量就成为私有变量，私有变量只能在对象内部访问，无法在外部访问，比如obj.a。

def __init__(self, name, age):

    self.__name = name #私有变量

python中类的构造函数中定义私有属性后，通过__dict__查看实例的所有变量，可以发现**私有属性被改名**为_ClassName__privateVar,即在_类名后面加私有属性.

python中虽然有私有变量概念，但私有变量还是可以通过hack方法访问的，通过__dict__可以知道python的私有变量只不过通过改名实现的，所以通过**instance._ClassName__privateVar** 可以访问到私有变量

js静态方法、属性前加static

js私有方法、私有属性前加#

**python继承**

#c6文件定义Human类

```
class Human():
    sum = 0
    def __init__(self, name, age):
        self.name=name
        self.age=age
    def get_name(self):
        print(self.name)
```

from  c6 import **Human**

class Student(**Human**):  **#类的参数用来传入父类**

    def __init__(self, school, **name, age**):

        self.school = school #子类定义的实例属性

        **super(Student, self).__init__(name, age) #调用父类方法**

    **#  Human.__init__(self, name, age)  #子类中调用父类构造函数通过传入父类定义的参数执行父类的构造函数**

        self.__score = 0

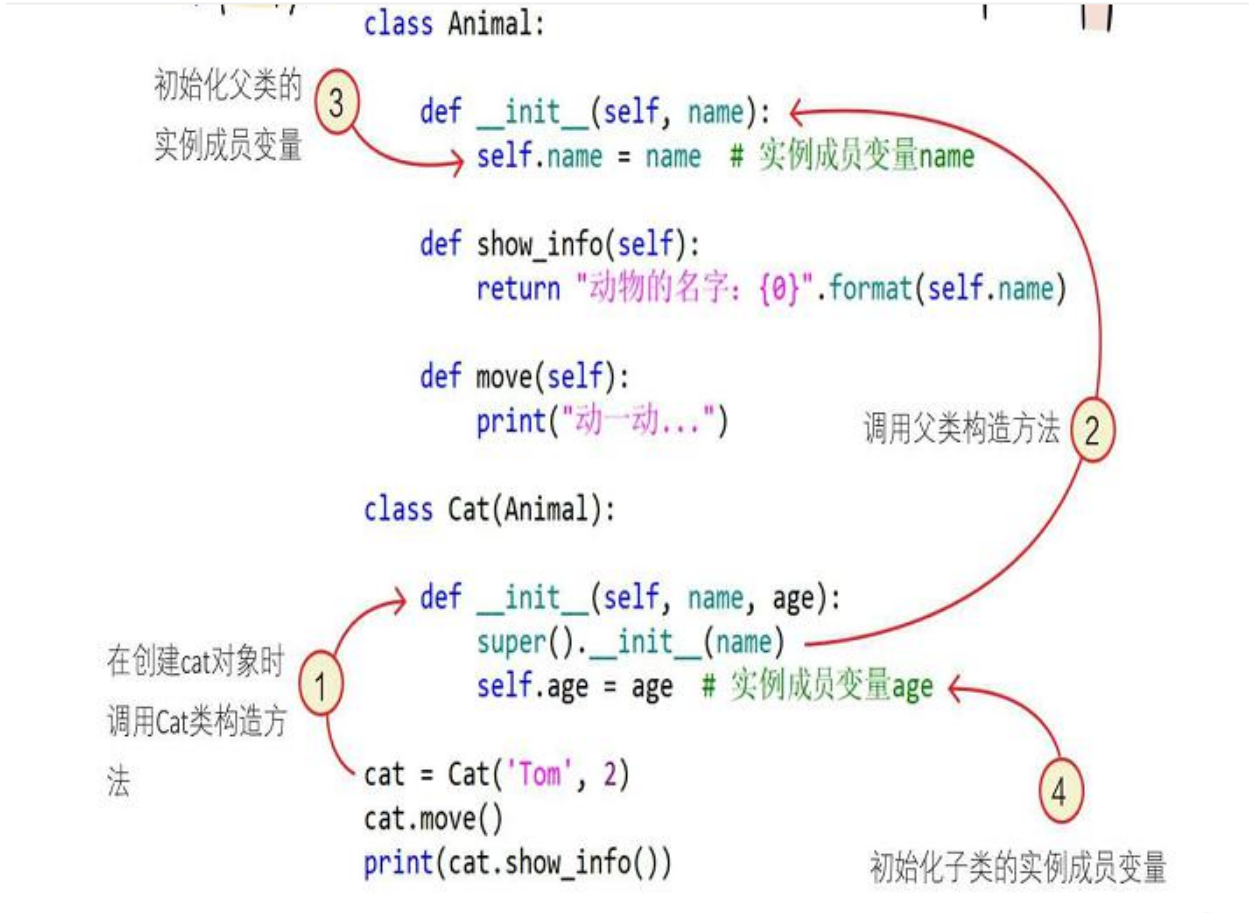
        self.__class__.sum += 1

    def do_homework(self):

**super(Student,self).get_name()** #spuer不仅可以执行父类的构造函数，还能在**子类中执行父类的实例方法**

print('do homework')

**子类继承父类的过程**



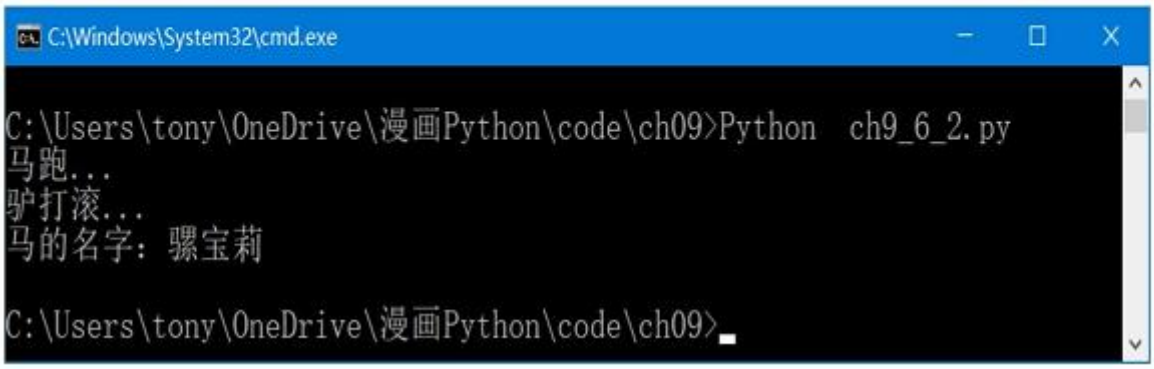

**多继承**

子类继承多个父类时，如果多个父类有相同的成员方法或成员变量，则子类优先继承左边的父类的成员方法或变量，继承时传入的父类参数从左到右其继承级从高到低。当子类方法和父类方法同名时，子类方法会重写overwrite父类方法

```python
# coding=utf-8
# 代码文件：ch09/ch9_6_2.py

class Horse:
    def __init__(self, name):
        self.name = name  # 实例变量name

    def show_info(self):
        return "马的名字：{0}".format(self.name)

    def run(self):
        print("马跑...")

class Donkey:
    def __init__(self, name):
        self.name = name  # 实例变量name

    def show_info(self):
        return "驴的名字：{0}".format(self.name)

    def run(self):
        print("驴跑...")

    def roll(self):
        print("驴打滚...")
```

```
26
27 class Mule(Horse, Donkey):
28
29     def __init__(self, name, age):
30         super().__init__(name)
31         self.age = age  # 实例变量age
32
33 m = Mule('骡宝莉', 1)
34 m.run() # 继承父类Horse方法
35 m.roll() # 继承父类Donkey方法
36 print(m.show_info()) # 继承父类Horse方法
```

通过Python指令运行文件，输出结果。




**oop思想**

封装、继承和多态是面向对象的三大特征。这三大特征与语言本身无关，这是一种面向对象的编程思想。

封装是为了提高程序的安全性；将数据（属性）和行为（方法）包装到类对象中。而在方法内部对属性进行操作，在类对象的外部调用方法。这样我们只需要用合适的方式去用它，而不用关心刚方法时如何具体实现的。

继承是为了提高代码的复用性；继承就是定义子类，子类继承父类的属性和方法。一个类没有继承任何类，默认继承object；Python支持多继承；

多态是为了提高程序的可扩展性和可维护性。Python中的多态指的是，在运行过程中根据变量所引用对象的类型，动态的决定调用哪个类对象中的方法。

```
class Student:
    provice_code = '0717'
    def __init__(self, name, age):
        self.name = name
        self.__age = age #前边加两_表示不希望再类外部被访问
    def show(self):
        print(self.name, self.__age)


stu = Student('张', 15)
```

print(stu.name)
print(stu.age) #**外部访问私有变量会提示错误**，但通过stu._Student__age可访问到 stu.show()

**Object类是所有类的父类，一个类没有继承任何类，默认继承object；**
类的浅拷贝,Python拷贝一般都是浅拷贝，拷贝时，对象包含的子对象内容不拷贝，因此，源对象与拷贝对象会引用同一个子对象
import copy
a2 = copy.copy(a1)
类的深拷贝,使用copy模块的deepcopy函数，递归拷贝对象中包含的子对象，源对象和拷贝对象所有的子对象也不相同。
import copy
a3 = copy.deepcopy(a1)

**为了让项目结构更清晰，建议一个模块只写一个类**

# 10.异常处理

异常处理 **try...except...finally**
出现异常后捕获异常，加以处理，不让程序终止并退出。常见的异常有：Exception, Error, warning，**error类型继承自Exception类**

```
try:
    pass #可能会出现错误的代码
except 异常类型 as e: #e是异常变量
    pass #出现异常后的处理代码
finally:
    pass #try、except代码块执行完之后，最终都会执行的代码块
```

except语句可指定异常类型，**如果不指定则except语句捕获try语句中的所有异常**，当指定具体异常时则except只捕获try中特定的异常。**多个异常类型的捕获可以放在一起。**
finally代码块中常用来释放资源，比如打开的文件、网络连接、打开数据库的连接、及数据结果集都会占用计算机资源，再finally中释放资源

```
try:
    pass #可能会出现错误的代码
except 异常类型1 as e:
    pass
```

except 异常类型2 as e:
    pass
except (异常类型3, 异常类型4) as e **#多个异常类型相似可以合并**
except: **#没指定异常类型，默认捕获上面未捕获到的异常**
    pass
finally:
    pass #释放资源
try...except 可以嵌套，但实际开发中尽量不要嵌套，应梳理好程序执行流程再考虑是否需要
try...except嵌套

**自定义异常，继承exception类**
class customException(**Exception**)：继承Exception类生成新的customeException异常类
    def __init__(self, message):
        super().__init__(message)
**raise 触发自定义异常**
raise customException('这是一个自定义的异常信息')

# 11.正则表达式 re模块

**import re**

**1. findall函数**

re.findall('正则规则'，目标字符串) #查找出与匹配规则的结果，返回结果是list列表

**【1】元字符**

re.findall('**\d**', a) #查找0-9的数字
\d、[0-9] #匹配0-9数字
'\D'、[^0-9] #匹配0-9非数字
'\w'、'[A-Za-z0-9_]' #匹配单词字符，即数字和字母和下划线
"\W' #匹配非单词字符符号，包含空白字符符号比如空格，回车，制表符等
'\s' #匹配空白字符符号比如空格，回车，制表符
'\S' #匹配非空白字符
'[\s\S]*?' #匹配除所有字符

'.' #匹配除换行符\n之外的其他所有字符

'a[**cf**]c' #查找a开头，c结尾，中间是**c或f**的结果

'a[**^cf**]c' #查找a开头，c结尾，中间不是**c或f**的结果

'a[**c-f**]c' #查找a开头，c结尾，中间是**c到f之间**的结果

**【2】数量词**

'[a-z]{3,6}' #匹配a-z之间的字符，最少匹配3位，最多匹配6位，贪婪模式匹配

'[a-z]{3,6}**?**' #匹配a-z之间的字符，最少匹配3位，最多匹配6位，**非贪婪模式**匹配3位即可

'pytho***' #匹配pytho**0次**或**无限多次**

'pytho**+**' #匹配pytho**1次**或**无限多次**

'pytho**?**' #匹配pytho**0次**或**1次**

**【3】边界匹配**

'^\d{3,8}$' #^即字符串开始位置，$即字符串结束位置。数字开头，数字结尾，且长度在3-8位之间

**【4】组**

r = re.findall('(Python){3}', s)

(Python){3} #将Python作为完整的一个分组，判断python是否重复3次

Python{3} #只判断Python中的单一字符n是否重复3次

小括号**(abc)**分组后，括号中的字符是且关系，中括号**[abc]**中的字符是或关系

**【5】匹配模式参数，findall的第三个参数**

r = re.findall('正则规则', 目标字符串, re.I | re.S) # re.I即不区分大小写，re.S改变元字符**.**的匹配规则。多个模式用管道符连接，是且关系

r = re.findall('c#', 'PythonC#JavaPHP', re.I)

r = re.findall('c#**.{1}**', 'PythonC#**\n**JavaPHP', re.I | re.S) #目标字符串里面有换行，元字符.是匹配除换行外所有字符，模式参数加上了re.S 就是改变元字符.的匹配特性，匹配包括换行符在内的所有字符。

**【6】正则替换，返回结果是字符串**

r = re.sub('C#', 'GO', 'PythonC#JavaPHP',0) #第二个参数是要替换成的字符串，第4个参数count设置为0是不限制替换次数

第2个参数可以定义成函数，实现更复杂的需求，**比如第一个参数定义的匹配规则匹配出的是变量的时候**

```
def convert(value):
    mached = value.group() #value是匹配到正则结果的match对象
    return 'AA' + mached + 'AA'
r = re.sub('C#', convert, 'PythonC#JavaPHP', 0) #第1个参数匹配到之后会传入到convert函数中
```

（convert函数中定义的value参数就是匹配上规则的一组match对象），返回函数执行的内容

示例：

```
def fn(value):
    matched = value.group()
```

```
    if(int(matched)>=6):
    return '9' #这里必须返回字符串，因为正则只能操作字符串
else:
    return '0'
```

re.sub('\d', fn, 'A8C3721D86', 0)

函数作为参数传入另一个函数，就是高阶函数

【7】re模块中除了findall之外的其他两个函数search, match，，只匹配一次，**返回结果是match对象**

r = re.match('\d', 'A83C7') #返回结果为None，因为match函数会从目标字符串的**首字母**开始匹配，如果首字母没匹配到就返回None

r = re.search('\d', 'A83C7') #search函数会搜索整个字符串，直到找到第一个匹配的结果就会**返回match对象**

**【8】group方法**,，match对象需要使用group()方法来获取具体的结果

r=re.search('life**(.*)**python**(.*)**python', 'life is short, i use python, i love python')

r.group(0) #获取所有分组匹配内容，即返回完整的匹配结果 'life is short, i use pyhon, i love python'

r.group(1) #获取第一个分组(.*)匹配的内容，即'is short, i use '

r.group(2) #获取第二个分组(.*)匹配的内容，即', i love'

r.group(0,1,2) #获取指定分组已元组形式返回 ('life is short, i use pyhon, i love python', 'is short, i use ',', i love')

**groups()方法**

m = re.match(r"(\d+)\.(\d+)", "24.1632")

m.groups() #返回('24','1632')

# 12. JSON

JSON是轻量级的数据交换格式

JSON是数据格式，与之对应的是XML数据格式

字符串是JSON的表现形式，符合JSON格式的字符串就是JSON

python中JSON和字典格式一样，在JS中JSON和对象格式一样，在每种语言中都有特定的数据格式与JSON对应从而实现转换。

1. JSON反序列化，转换json格式字符串为JSON，必须是标准格式的JSON字符串'{"name":"abc"}'

json转字典

import json

ditc1 = **json.loads(**jsonStr**)** #JS中JSON反序列化为对象。**JSON.parse(jsonString)**

JSON 对象数组也能反序列化为python list

list1 = json.loads('[{"name":"a1"},{"name2":"a2"}]') #JS中JSON反序列化为数组，**JSON.parse(jsonString)**

**JSON数据格式中的boolean值是小写**

| json | python |
| --- | --- |
| object | dict |
| array | list |
| string | str |
| number | int |
| number | float |
| true | True |
| false | False |
| null | None |

2. JSON序列化，转换JSON数据格式为JSON格式字符串

import json

jsonStr = **json.dumps(**jsonObject**)**

**js:JSON.stringify(jsonObject)**

JSON：一种通用的数据交换格式，支持各种语言。在每种编程语言中都有特定的数据类型和JSON格式可以转换。

JSON字符串：符合标准JSON语法格式的字符串

JSON对象：只在JS语言中存在JSON对象，在python中是dict，或list dict

# 13. 枚举

枚举其实是一个类，枚举不可实例化。**枚举就是列举出有穷集合的所有元素，枚举的成员不可修改**，且枚举类型中**没有重复成员名称name，但成员名称对应的值可以相同，后面相同值对应成员名称name相当于是别名**

```
from enum import Enum
class VIP(Enum): #继承Enum父类生成一个子类VIP，这里不同于普通的类，普通类的属性是可修改的
    YELLOW=1
    GREEN=2
    abc =2 #因为枚举值相等，所以后面的abc相当于是前面GREEN的别名
VIP.YELLOW =2 #会提示报错，因为枚举类型不能修改
VIP.YELLOW #返回VIP.YELLOW 获取的是一个枚举类型即<enum 'VIP'>，不是具体的值
VIP.YELLOW.value #获取枚举类型中标签对应的枚举值
```

VIP.YELLOW.name #获取枚举类型中**枚举名称**

**遍历枚举**

```
for v in VIP:
    print(v) #返回VIP类中的所有的枚举类型
```

枚举的比较运算，**枚举类型之间**只能判断是否相等，是否同一身份比较is，不能判断大小关系，

```
VIP.YELLOW == VIP.YELLOW #返回True
VIP.YELLOW == VIP.GREEN #返回False
VIP.YELLOW == 1 #会返回False，不是枚举类型之间比较
VIP.YELLOW  is VIP.GREEN #返回True
```

**枚举转换**

当已知某个值，要根据该值匹配出某个枚举类中的成员

```
a = 1
VIP(a) #返回的是VIP.YELLOW 枚举类型
```

# 14.python高级编程

**1.一切皆对象**

```
a=1
a='2'
a = fn
```

python中能将**函数作为参数传入到另一个函数里**，也可以把**函数作为另一个函数的返回结果。**

**2.闭包**

**函数及函数声明时所在的环境变量的集合叫闭包。可在函数外部调用函数内部的局部变量。闭包可以保存函数的环境变量，避免函数执行后环境变量被垃圾回收。**

查看某个函数的闭包使用：**__closure__[0].cell_contents**

```
def curve_pre():
    a= 25
    def fn(x):
        return a * x #fn函数内引用着curve_pre函数的环境变量
    return fn # curve_pre执行后返回的是fn函数，fn函数内引用着curve_pre的环境变量
f = curve_pre()
f.__closure__[0].cell_contents
```

记录行走步数的示例

```
def save_step(x):
```

```
    origin = x
    def walk_step(step):
     #需要修改闭包局部变量时，首先要使用nonlocal显示声明变量不是内层函数的局部变量
        nonlocal origin #声明引用的是外层函数的局部变量
        origin += step #如果不使用nonlocal显示声明，此行修改变量会提示变量不存在的错误
        print(origin)
        return origin
    return walk_step

fn = save_step(0)
fn(2)
print(fn.__closure__[0].cell_contents)
fn(6)
print(fn.__closure__[0].cell_contents)
fn(12)
print(fn.__closure__[0].cell_contents)
```

**3. lambda表达式**

定义普通函数

```
def add(x,y):
    return x+y
```

使用**lambda表达式**定义**匿名函数**

lambda parameter_list: expression

**lambda表达式后面只能是一个简单的表达式，不能是复杂的代码块，lambda表达式不需要使用return**

```
lambda x,y: x+y
```

匿名函数的调用

```
f = lambda x,y: x+y
f(1,2)
(lambda x,y: x+y)(1, 2)
```

**JS匿名函数**

```
(functon(x, y) {
    return x+ y
})(1,2)

(x,y) => x+y
```

**4.三元表达式**

**条件为真时返回的结果 if 条件判断 else 条件为假时返回的结果**

result = x if x>y else y #返回x时必须满足x>y，否则返回y

**JS三元表达式**

(x>y)? x : y

**5.函数式编程**

函数式编程与命令式编程

1. 命令式编程就是编写控制过程，包含函数，条件判断， 循环

2. 函数式编程是使用map, reduce, filter, lambda表达式替换函数，循环

声明式编程 同样是一种编程风格，它通过定义具体的规则，以便系统底层可以自动实现具体功能。主要思想是 **告诉计算机应该做什么，但不指定具体要怎么做。**

函数式编程 属于声明式编程中的一种，它的主要思想是**将计算机运算看作为函数的计算**，也就是**把程序问题抽象成数学问题**去解决。函数式编程中，我们可以充分利用数学公式来解决问题。也就是说，任何问题都可以通过函数（加减乘除）和数学定律（交换律、结合律等），一步一步计算，最终得到答案。

**【1】map函数**，返回一个map类，使用**list(**map_x**)** 转成列表

```
list_x = [1,2,3,4,5,6,7,8]
list_y = [1,2,3,4,5,6]
方法一 for...in循环
for x in list_x:
    square(x)
方法二 map函数遍历，map函数把要遍历的目标对象逐一传入前面的函数中进行处理
def square(x):
    return x * x
r= map(square, list_x) #返回结果【1，4，9，16，25，36，49，64】
list(r)
使用lambda表达式定义匿名函数，简化代码
r=map(lambda x: x*x, list_x)
r=map(lambda x, y: x*x + y, list_x, list_y)
list(r)
```

**【2】reduce** 每次的计算结果作为下一次计算的参数

```
from functools import reduce
```

```
list_x = [1,2,3,4,5,6,7,8]
r = reduce(lambda x,y:x+y, list_x, 10) #reduce函数的第三个参数10是初始值
```

**【3】filter**，返回一个过滤后的集合，结果使用**list(**filterx**)**转成列表

```
list_x = [1,0,1,0,0,1]
r = filter(lambda x: True if x ==1 else False, list_x) #挑选出x为1的元素
```

#**filter函数中的lambda表达式的返回结果必须为布尔值或者0，1类布尔值**，必须是能返回真假的表达式。

```
list(r)
```

## 15. 装饰器

开闭原则：**对修改是封闭的，对扩展是开放的。**不改变原有函数的基础上，增加新的功能

装饰器使用原则

【1】当需要对某一个封装的单元比如某个函数做出修改，可以在不修改原有函数的基础上使用装饰器的形式扩展原有函数的功能进而达到间接修改原有函数。

【2】如果需要复用某个函数的功能

示例：

需求：打印函数的同时，打印当前的时间戳。

```
import time
def f1():
    print('This is fn1')
def f2():
    print('This is fn2')
def print_current_time(fnc):
    print(time.time())
    fnc()
print_current_time(f1)
print_current_time(f2)
#以上的方式让函数f1、f2和print_current_time没有具体的关联性仍然是独立的个体，没有内聚性。
```

**使用装饰器解决问题的思路，不改变原有函数的基础上，增加新功能**

```
import time
def decorator(func): #自定义一个装饰器名称
    def wrapper(*args, **kw):  #*args表示不确定的多个参数, **kw表示关键字参数
```

```
        print(time.time()) #装饰目标函数，新增加的代码逻辑
        func(*args, **kw)   #*args收敛可变参数，**kw收敛关键字参数
    return wrapper
@decorator #不改变原有函数，增加新功能，代码没有改变f1的内容
def f1(fun_name):
    print('This is a function1'+ fun_name)
f1('a')
@decorator
def f2(fun_name1, fun_name2):
    print('This is a function2' + fun_nam1)
    print('This is a function2' + fun_nam2)
f2('a','b')
@decorator
def f3(fun_nam1, fun_nam2, **kw):
     print('This is a function2' + fun_nam1)
    print('This is a function2' + fun_nam2)
    print(kw)
f3('a','b',a=1,b=2,c='123')
```

装饰器的副作用：**使用装饰器后原有函数的名称会改变为wrapper，函数内的__doc__说明文档会改变，解决增加装饰器后函数名称改变的方法**

```
from functools import wraps
def  my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kw):
        print('this is fn') # 装饰目标函数，新增加的代码逻辑
        return func(*args, **kw) #执行所要装饰的目标函数
    return wrapper

@my_decorator
def example():
    """this is fn doc"""
    print('this is fn')
example.__name__ #函数的名称仍然是example,
example.__doc__#函数的注释还是原函数的注释
```

# 16. python编程技巧

**1. 使用字典代替switch case 语句，python没有switch case**

```
def get_sunday:
    return 'Sunday'
def get_monday:
    return 'Monday'
def get_tuesday:
    return 'Tuesday'
def get_default:
    return 'Unknow'
dict1 = {
    0:  get_sunday,
    1:  get_monday,
    2:  get_tuesday
}
day = 2
dict1.get(day, get_default)() #当获取不到指定day所对应的值时，默认返回get_default
```

**2. 列表推导式**

根据已经存在的列表创建一个新的列表

```
a = [1, 3, 5, 7, 9]
b = [for i in a] #返回一个新的列表
b1 = [i**2 for i in a] #返回a列表每个元素的平方，生成新的列表
```

条件筛选推导除列表

```
c = [i**2 for i in a if i>=5] #筛选出原列表中大于等于5的元素的平方，生成新的列表
```

列表、元组、集合都可以被推导

```
a = {1,2,3,4,5,6,7,8} #集合
bb = (1,2,3,4,5,6,7,8) #元组
cc = [1,2,3,4,5,6,7,8] #列表
a1 = [i**2 for i in a] #集合推导出列表
a2 = {i**2 for i in a} #集合推导出集合
a3 = [i**2 for i in bb] #元组推导出列表
a4 = {i**2 for i in cc} #列表推导出集合
```

字典推导出列表

```
students = {
    'name1': 18,
    'name2': 22,
    'name3': 24
}
```

b= **[**key for **key,value** in students.**items()]** #将原字典的key提取推导出新的列表

b2= **{key:value** for key,value in students.items()**}** #推导出新的key:value字典

### 3. iterator 与 generator 迭代器与生成器

**iterator迭代器**

可迭代对象iterable：凡是可以被for...in循环遍历的对象就是可迭代对象。比如列表、元组、集合都具有可迭代接口。

迭代器iterator即迭代接口，是一个可迭代的对象，但可迭代对象(列表、元组、集合)不一定是迭代器。

迭代器有两个内置方法**__iter__**，**__next__**，迭代器可以使用next函数控制迭代过程，**迭代器是一次性的**，第一次遍历完毕后就无法再遍历了。

**可迭代对象可以多次遍历，没有next函数控制迭代过程。**

使用迭代器创建一个可迭代的class类

```
class BookCollection:
    def __init(self):
    self.data = ['往事','只能','回味']
    self.cur = 0 #定义游标
    pass
def __iter__(self):
    return self
def __next__(self):
    if self.cur >= len(self.data):
    raise StopIteration() #如果游标值超过data列表长度即数据已全部遍历则抛出异常（迭代完成）
    result = self.data[self.cur]
    self.cur  += 1
    return result

books= BookCollection()
```

```
#迭代器可使用for...in遍历
for boo in books:
    print(book)
#通过next函数控制迭代过程
print(next(books))
print(next(books))
print(next(books))
```

**生成器函数generator**

```
n = [i for i in range(0,10001)]
def gen(max):
    n = 0
    while n <=max:
    n += 1
    yield n
g = gen(10000)
print(next(g))
print(next(g))
print(next(g))

for i in g:
    print(i)
```

**4.None**，None是单独的对象类型**NoneType**，表示此处有值但为空，**JS null值**

判断None 操作

```
if var1 is None:
if var1:      或者 if not var1:
```

类在做真假值判断的时候，会调用bool()方法将类转换成bool类型，类其定义内置的**__len__(self)**、**__bool__(self)**方法会作为判断真假值的依据。

```
class Test():
    def __len__(self):
    return False
bool(Test()) #此时判断为False
```

判断一个类型是否存在

```
a= Test()
if a:
    pass
```

**5.海象运算符 顺便声明一个由表达式计算结果赋值的变量**

**:=**

(variable_name := expression or value)

类似声明一个变量后，再来判断变量（多次用到）的场景，可以使用海象运算符简化代码

```
a=1+2
if a>0:
    pass
#使用海象运算符简化代码
if a:=(1+2) >0:
    pass
```

## 17.多线程

程序在运行时一次智能执行一个任务（单线程），让程序同时执行多个任务就要使用多线程技术。

1.程序进入执行状态后就是一个进程，每个进程有自己的独立的内存空间、系统资源，每一个进程的内部数据和状态都是完全独立的。windows中的进程局势exe或者dll程序，进程之间相互独立也可以通信。

2.一个进程中可以包含多个线程，多个线程共享一块内存空间和一组系统资源，所以系统在各线程之间切换时，系统开销比进程小得多，因此线程称为轻量级进程。

3.python程序至少有一个线程即主线程，python程序启动后由python解释器负责创建主线程，程序结束后由python解释器停止主线程。多线程中，主线程负责其他子线程的调度，启动、挂起、停止等。多线程编程时，需要给每个子线程执行分配机会，通过让当前子线程休眠暂停（**延迟当前子线程的后续执行**）执行，让其他线程有机会执行。如果当前子线程没有休眠，只能等待当前线程执行后再执行第二个线程。

多线程可以把空闲时间利用起来，比如有两个进程函数 func1、func2，func1函数里使用sleep休眠一定时间，如果使用单线程调用这两个函数，那么会顺序执行这两个函数，也就是直到第一个函数执行完后，才会执行第二个函数，这样需要很长时间；如果使用多线程，会发现这两个函数是同时执行的，这是因为多线程会把空闲的时间利用起来，在第一个函数休眠的函数就开始执行第二个函数

python多线程使用场景：如果程序时cpu密集型的，使用python的多线程是无法提升效率的，如果程序时IO密集型的，使用python多线程可以提高程序的整体效率。

CPU密集型（CPU-bound）：CPU密集型也叫计算密集型，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，CPU Loading很高。

IO密集型（I/O bound）：IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高，I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力

多线程的应用有很多，一些阻塞主线程的操作应该被放到子线程中处理，比如打开文件，网络爬虫。

多线程会产生并发问题，多个线程如果同时读取某个变量导致相互干扰产生并发问题，所以实际开发中尽量避免多个线程读取或写入相同的变量。

import _thread as thread from time import sleep, ctime def fun1():  print('开始运行func1', ctime())  # 休眠4秒  sleep(4)  print('func1运行结束', ctime()) def fun2():  print('开始运行func2', ctime())  # 休眠4秒  sleep(2)  print('func2运行结束', ctime()) def main():  print('开始运行时间', ctime())  # 启动一个线程运行func1函数  thread.start_new_thread(fun1, ())  thread.start_new_thread(fun2, ())  # 休眠6秒  sleep(6)  print('运行结束时间', ctime()) if __name__ == '__main__':  main() E:\python\python.exe E:/progect/untitled1/untitled1/urls.py 开始运行时间 Sat Feb 16 09:34:00 2019 开始运行func1 Sat Feb 16 09:34:00 2019 开始运行func2 Sat Feb 16 09:34:00 2019 func2运行结束 Sat Feb 16 09:34:02 2019 func1运行结束 Sat Feb 16 09:34:04 2019 运行结束时间 Sat Feb 16 09:34:06 2019

**线程模块 threading，线程类Thread**

线程模块相关函数：

avtive_count()  返回当前处于活动状态的线程个数

current_thread() 返回当前的Thread对象

main_thread() 返回主线程对象，主线程是python解释器启动的线程

import threading

threading.current_thread()

threading.active_count()

threading.main_thread()

创建子线程

1. 线程对象，由threading模块的Thread类或Thread的子类构建的对象

2. 线程体，即子线程要执行的代码，通常封装到一个函数中。子线程启动后会执行线程提。

实现线程体有以下两种方式

1.自定义函数中实现线程体

Thread(target=fnName, name='threadname', args=[x1,x2])

target参数指向自定义的线程体函数

name参数可自定义线程名称

args为线程体函数提供的参数，列表类型

示例

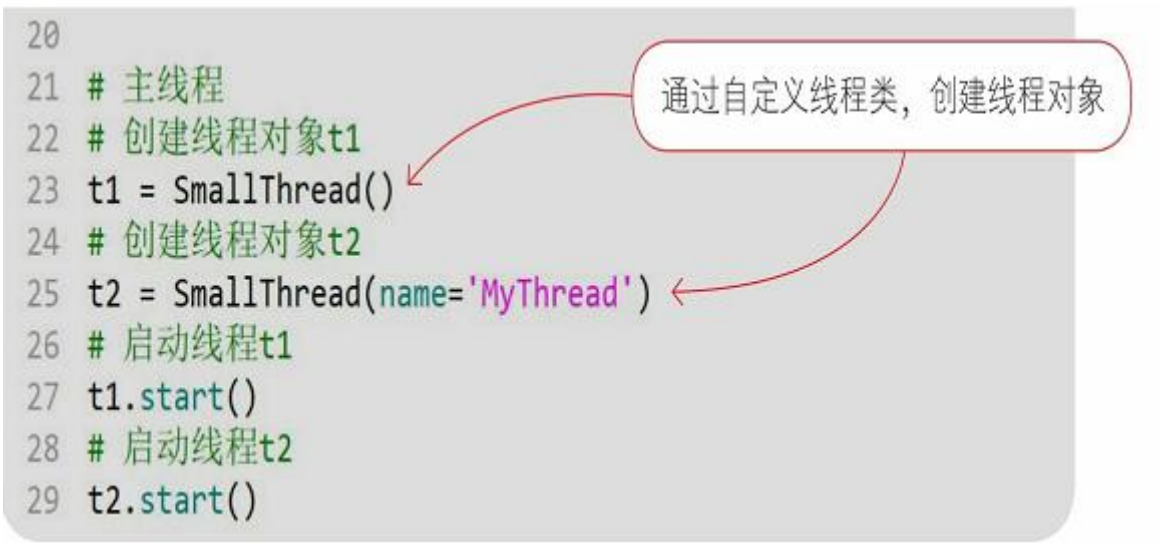
import threading

t1 = threading.Thread(target=Fn1, name ='myThread')

t1.start()

2.自定义线程类实现线程体，run()方法就是线程体函数

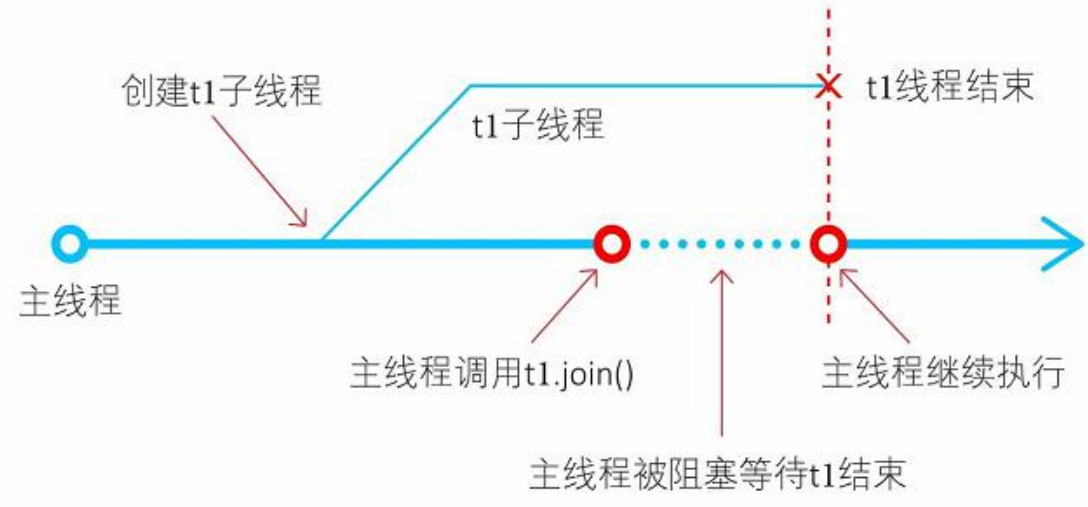

```python
# coding=utf-8
# 代码文件：ch16/ch16_3_2.py

import threading
import time

class SmallThread(threading.Thread):
    def __init__(self, name=None):
        super().__init__(name=name)
    # 线程体函数
    def run(self):
        # 当前线程对象
        t = threading.current_thread()
        for n in range(5):
            # 当前线程名
            print('第{0}次执行线程{1}'.format(n, t.name))
            # 线程休眠
            time.sleep(2)
        print('线程{0}执行完成！'.format(t.name))
```




```python

# 主线程
# 创建线程对象t1
t1 = SmallThread()
# 创建线程对象t2
t2 = SmallThread(name='MyThread')
# 启动线程t1
t1.start()
# 启动线程t2
t2.start()
```

**线程管理**

1. 等待线程结束，某些场景可以控制主线程等待另一个子线程t1执行结束后才能继续执行




join(timeout= None) #设置超时时间单位秒，不设置默认一直等待，调用join()方法让主进程阻塞，等待t1子进程执行完毕后再继续执行。

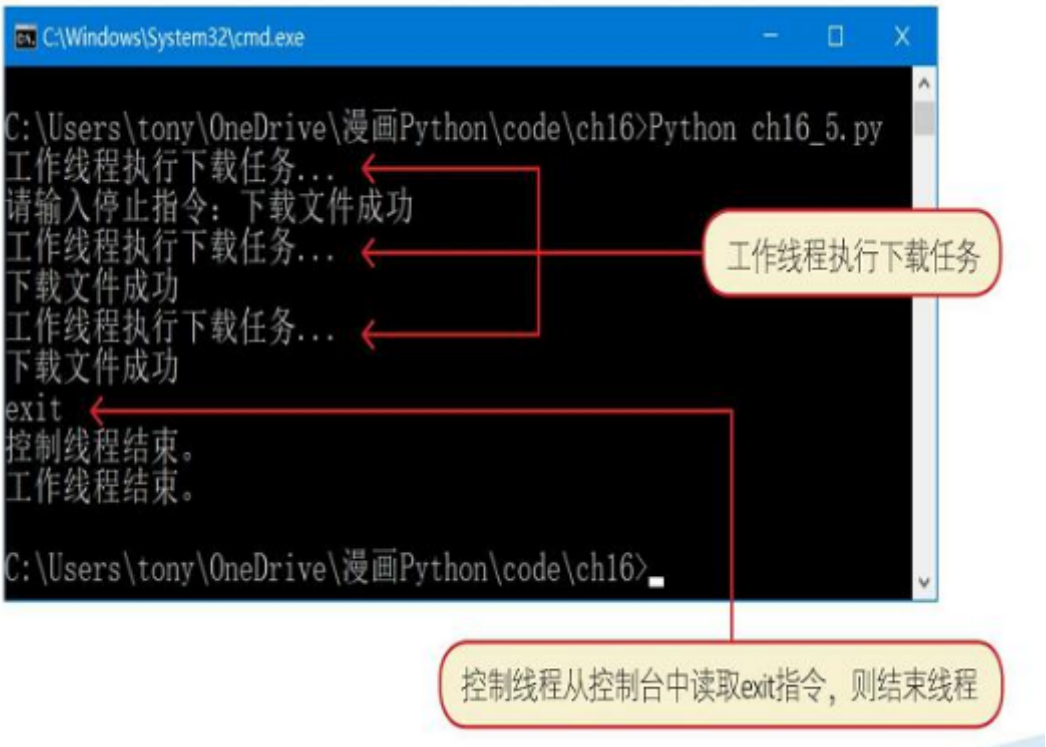

1 # coding=utf-8
2 # 代码文件：ch16/ch16_4_1.py
3
4 import threading
5 import time
6
7 # 共享变量
8 value = []
9
10 # 线程体函数
11 def thread_body():
12 # 当前线程对象
13 print('t1子线程开始...')
14 for n in range(2):
15 print('t1子线程执行...')
16 value.append(n)
17 # 线程休眠
18 time.sleep(2)
19 print('t1子线程结束。')
20
21 # 主线程
22 print('主线程开始执行...')
23 # 创建线程对象t1
24 t1 = threading.Thread(target=thread_body)
25 # 启动线程t1
26 t1.start()
27 # 主线程被阻塞，等待t1线程结束
28 t1.join()
29 print('value = {0}'.format(value))
30 print('主线程继续执行...')
定义一个共享变量value，该变量是多个线程都可以访问的变量
在子线程体中修改变量value的内容
在当前线程（主线程）中调用t1的join()方法，因此会导致当前线程阻塞，等待t1线程结束
t1线程结束，继续执行，访问并输出变量value

2.

```python
# coding=utf-8
# 代码文件：ch16/ch16_5.py

import threading
import time
import urllib.request

# 线程停止变量
isrunning = True

# 工作线程体函数
def workthread_body():
    while isrunning:
        # 线程开始工作
        print('工作线程执行下载任务...')
        download()
        # 线程休眠
        time.sleep(5)
    print('工作线程结束。')

# 控制线程体函数
def controlthread_body():
    global isrunning
    while isrunning:
        # 从键盘输入停止指令exit
        command = input('请输入停止指令：')
        if command == 'exit':
            isrunning = False
            print('控制线程结束。')

def download():
    url = 'http://localhost:8080/NoteWebService/logo.png'
    req = urllib.request.Request(url)
    with urllib.request.urlopen(url) as response:
        data = response.read()
        f_name = 'download.png'
        with open(f_name, 'wb') as f:
            f.write(data)
            print('下载文件成功')

# 主线程
# 创建工作线程对象workthread
workthread = threading.Thread(target=workthread_body)
# 启动线程workthread
workthread.start()

# 创建控制线程对象controlthread
controlthread = threading.Thread(target=controlthread_body)
# 启动线程controlthread
controlthread.start()
```

在工作线程中执行下载任务，这个下载任务每5秒调用一次

下载函数，由工作线程调用



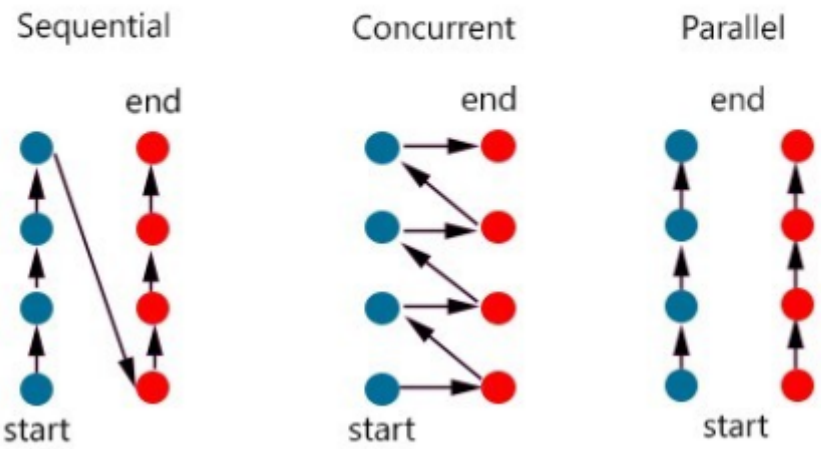

## 17.多线程与多进程

进程(process)和线程(thread)是操作系统的基本概念，是操作系统程序运行的基本单元。

**进程是执行中的程序，是资源分配的最小单位**：操作系统以进程为单位分配存储空间，进程拥有独立地址空间、内存、数据栈等。操作系统管理所有进程的执行，分配资源。可以通过fork或 spawn的方式派生新进程，新进程也有自己独立的内存空间。**多进程并行执行。**

**线程是CPU调度的的最小单位：**一个进程可以有多个线程，同进程下执行，并共享相同的上下文。线程间的信息共享和通信更加容易。**多线程并发执行**

- **并发、并行**




**并发**通常应用于 **I/O 操作频繁**的场景，**并行**则更多应用于 **CPU heavy** 的场景。

**并发(concurrency)**，指同一时刻只能有一条指令执行，多个线程的对应的指令被快速轮换地执行，线程/任务之间会互相切换。

【1】处理器先执行线程 A 的指令一段时间，再执行线程 B 的指令一段时间，再切回到线程 A，快速轮换地执行。

【2】处理器切换过程中会进行上下文的切换操作，进行多个线程之间切换和执行，这个切换过程非常快，使得在宏观上看起来多个线程在同时运行。

【3】每个线程的执行会占用这个处理器一个时间片段，同一时刻，其实只有一个线程在执行。

**并行（parallel）**,指同一时刻，有多条指令在多个处理器上同时执行

【1】不论是从宏观上还是微观上，多个线程都是在同一时刻一起执行的。

【2】并行只能在多处理器系统中存在，如果只有一个核就不可能实现并行。并发在单处理器和多处理器系统中都是可以存在的，一个核就可以实现并发。

注意：具体是并发还是并行取决于操作系统的调度。

**多线程适用场景**

多线程/多进程是解决并发问题的经典模型之一。

在一个程序进程中，有一些操作是比较耗时或者需要等待的，比如等待数据库的查询结果的返回，等待网页结果的响应。这个线程在等待的过程中，处理器是可以执行其他的操作的，从而从整体上提高执行效率。

比如网络爬虫，在向服务器发起请求之后，有一段时间必须要等待服务器的响应返回，这种任务属于 IO 密集型任务。对于这种任务，启用多线程可以在某个线程等待的过程中去处理其他的任务，从而提高整体的爬取效率。

还有一种任务叫作计算密集型任务，或者称为CPU 密集型任务。任务的运行一直需要处理器的参与。如果使用多线程，一个处理器从一个计算密集型任务切换到另一个计算密集型任务，处理器依然不会停下来，并不会节省总体的时间，如果线程数目过多，进程上下文切换会占用大量的资源，整体效率会变低。

所以，如果**任务不全是计算密集型任务，我们可以使用多线程来提高程序整体的执行效率。**尤其对于网络爬虫这种 IO 密集型任务来说，使用多线程会大大提高程序整体的爬取效率，多线程只适合IO 密集型任务。

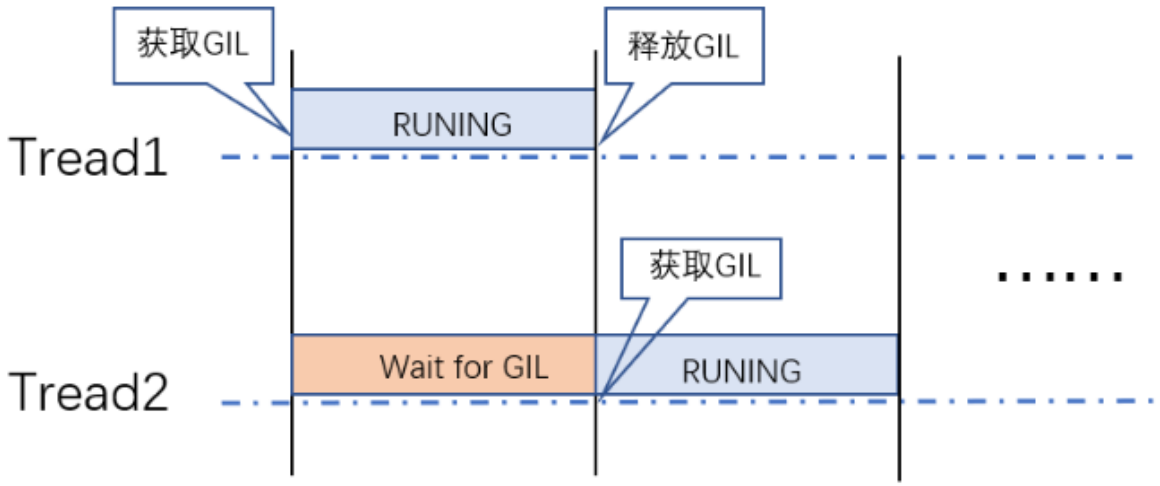
**python GIL**

由于 Python 中 GIL 的限制，导致**不论是在单核还是多核条件下，在同一时刻只能运行一个线程，导致 Python 多线程无法发挥多核并行的优势。**GIL 全称为 Global Interpreter Lock（全局解释器锁），是 Python 解释器 CPython 中的一个技术术语，是Python之父为了数据安全而设计的。

在 Python 多线程下，每个线程轮流执行:

获取 GIL-->执行对应线程的代码-->释放GIL

**某个线程想要执行，必须先拿到 GIL**，并且在一个 Python 进程中，GIL 只有一个，导致即使在多核的条件下，同一时刻也只能执行一个线程。每一个线程执行完一段后，会释放 GIL，以允许别的线程开始利用资源。




Python 由于GIL锁的存在，无法利用多进程的优势，要真正利用多核，可以重写一个不带GIL的解释器， 比如JPython（Java 实现的 Python 解释器）。某些Python 库使用C语言实现，例如 NumPy 库不受 GIL 的影响。在实际工作中，如果对性能要求很高，可以使用C++ 实现，然后再提供 Python 的调用接口。另外Java语言也没有GIL限制。

## 18.python文档化

文档化是在python代码中添加注释，提供代码的详细说明，包括参数和返回值的说明，以及代码示例。

注解不同于注释，注解有更广泛的作用，注解在python中是一种元数据机制，用于在代码中添加额外信息，可用于类型检查、函数参数、返回值等。注解在python中通常与类型提示一起使用，提供关于变量，参数，返回值的预期类型信息。python的元数据机制是用于组织、管理、存储元数据的模型，即描述数据的信息，包含数据的类型，值的范围，来源以及其他属性。在python中，注解通常使用冒号来分隔表达式和类型。注解用于提供关于变量类型等元数据的信息，而注释则是为了在代码中添加人类可读的解释和说明。注解是Python 3引入的新特性，主要用于类型提示和类型检查。

```
def greet(name: str) -> str:
    return "Hello, " + name
```

在这个例子中，name: str和-> str就是注解，它们告诉开发者greet函数接受一个字符串类型的参数，并返回一个字符串类型的值。

**注解**是Python 3引入的新特性，**主要用于类型提示和类型检查**:

使用 mypy 这样的静态类型检查工具可以在开发时提供更早的反馈

```
pip install mypy
```

```
2
# mypy_example.py
    def greet(name: str, age: int) -> str:
    return f"Hello, {name} ({age} years old)"
```

在命令行中运行 mypy，mypy 将会检查并报告潜在的类型错误。

```
mypy mypy_example.py
```

## 19.I/O操作

python中文件I/O可以通过内置的open()函数实现，该函数打开一个文件并返回一个对象，通过使用文件对象可以对文件进行读，写操作。

```
#打开，并读取操作
with open('./text.txt','r') as f:
    content = f.read()
    print(content)
#打开，并读取操作，按行打印，逐行读取
with open('./text.txt', 'r') as f:
    for line in f:
        print(line)
```

## 20.dotenv

dotenv库用来读取项目中的**.env文件**，将.env文件中**定义的环境变量导入到当前程序运行的环境中供程序使用。**将**敏感信息（API密钥，数据库密码等）存储在环境变量**中而不是硬编码到代码中**提高程序运行的安全性**，因为这些敏感信息不会存储在代码库中，**只存在于程序运行的环境中。**
git版本控制信息中可通过**.gitignore文件忽略.env配置文件**，避免敏感信息上传到远程仓库。
1.安装python-dotenv库

```
pip install python-dotenv
```

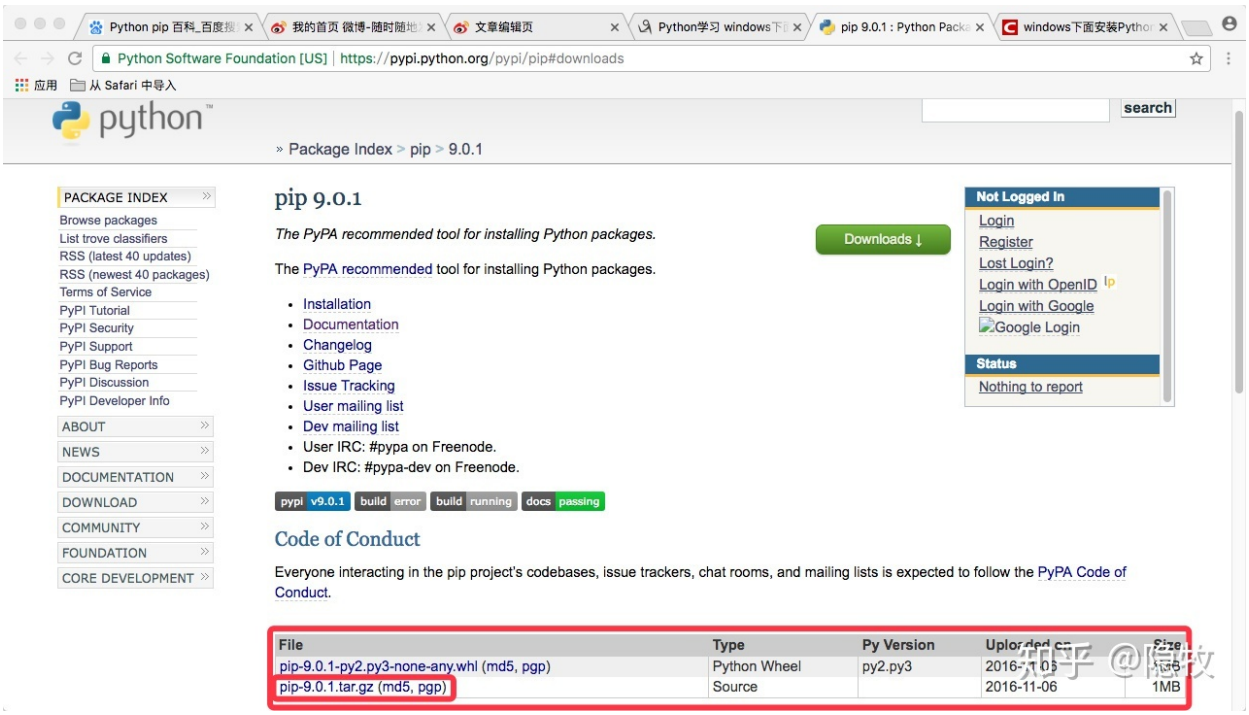
2.创建.env文件，并将环境变量写入到.env文件中，每组key=value对应一行
3.将.env文件中的环境变量挂在到系统环境中。通过使用load_dotenv函数读取.env配置文件中的环境变量挂载到系统环境中。

```
from dotenv import load_dotenv
load_dotenv() #将.env配置文件中定义的环境变量挂在到系统环境中
SECRET_KEY = os.getenv("SECRET_KEY") #调用环境变量
```

# 安装pip

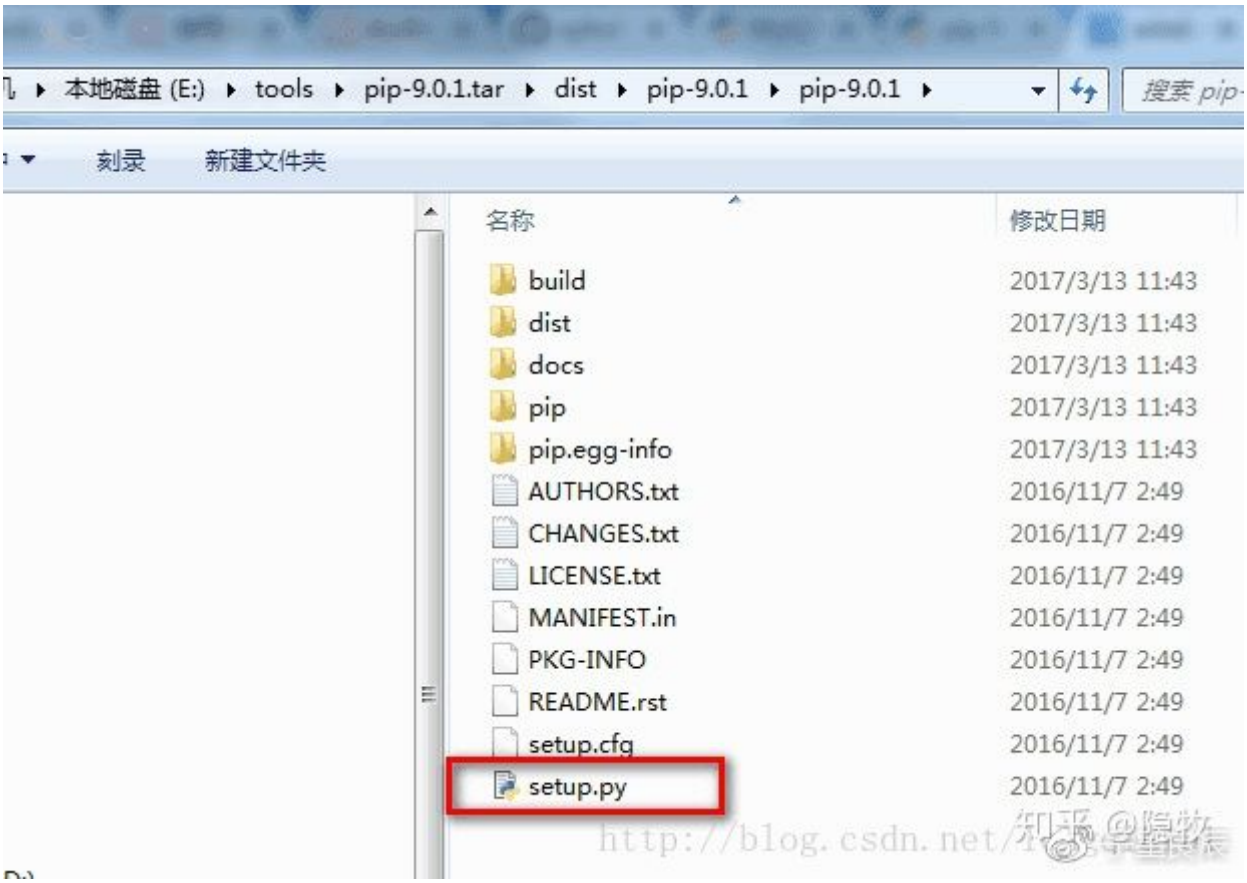
1.在Python官网上下载Windows版本pip安装包

https://pypi.org/project/pip/#downloads




下载完成后，将得到一个压缩包，将压缩包进行解压。

2.打开控制台，使用cd命令进入解压后的文件夹至可执行目录，如下图所示：




3.在控制台输入如下命令：

**python setup.py install**

回车，控制台将自动安装pip，安装完成后，在控制台输入 python -m pip --version 命令

如果显示'pip'不是内部命令，也不是可运行的程序，说明，缺少环境变量。


在此，需要在系统环境变量PATH中添加环境变量：C:\Python27\Scripts（本次安装的是Python2.7.10版本，所以需要添加环境变量C:\Python27\Scripts）;添加完成后，再次执行 python -m pip --version 命令，如果控制台输出pip的版本号，说明安装成功。

# pip python包管理工具

在cmd命令行环境下安装包

>>> pip install packName

>>> pip install packName[numpy,scipy] #安装包，同时安装包的依赖项

>>> pip install packName ==1.2 #指定安装版本
>>> pip install --upgrade packName #升级已安装的包
>>> pip install --no-dependencies packName #安装包，忽略包的依赖
>>> pip install /path/to/packName #安装本地指定目录的包
>>> pip uninstall packName
>>> pip list #列出当前python环境已安装的所有包或库
>>> pip show packName #列出指定包的详细信息
>>> pip search packName #在python package index 上查找包的信息
>>> pip #查看帮助
>>> pip freeze > requirements.txt #导出依赖库，生成依赖包列表文件
>>> pip install -r requirements.txt #根据导出的依赖包列表文件，安装依赖包
>>> pip install pipreqs #依赖包管理模块
>>> pipreqs ./ #导出当前项目目录下所依赖的包
>>> pip cache purge

## python 常用命令行

**cls** #清屏幕
**pip install**/uninstall #安装或卸载包
**pip** #查看帮助
**help(**round**)** #查看某个函数的使用方法
**import this** #查看python之禅

## python 常用全局函数

**id(**var1**)** #显示变量在内存中的位置
**dir(**obj**)** #**列表形式**返回当前模块下的**所有变量**，包含内置变量，dir(sys)返回sys包的所有变量组成的数组
1.操作相关函数
**print(**obj,**end='|')**
a=**input()**
**exec(**'print("Python")'**) #执行python语句**
**eval(**'1+1'**)** #执行一个表达式
**type(**obj**)**
**id(obj) #返回某个obj对象的唯一标识**
**globals() #返回全局变量的字典**

**help()**
**isinstance(**'a', str**) #判断一个对象是否为某个类的实例**
**issubclass(**class1,class2**) #判断一个类是否为另一个类的子类**

2.数学函数
**len(**a**)**
**max(**a**)**
**min(**a**)**
**round(**1.123, 2**)**
**abs()**
**sum()**
**sorted(**lis**) #排序，返回被排序后的list**
**reverse(**iterate**) #返回可迭代对象的反转**
**divmod(**a,b**) #获取a除以b的商和余数**
**pow(**a，b**) #a的b次方**
**range(**num1,num2, **step**)

3.类型转换函数
**str(**'a'**)** #等同于 **js toString(var)**
**int(**1.0**)**
**float(**a**)**
**bool(**1**)**
**tuple(**iterate**)** #可迭代对象转换成tuple
**list(**iterate**)** #可迭代对象转换成list
**dict(**iterate**)** #可迭代对象转换成dict
**set(**iterate**)** #可迭代对象转换成set
**iter(**iterable**)** #返回一个可迭代的对象
**enumerate(**iterable**)** #返回一个枚举对象
hex(int) 转16进制
oct(int) 转8进制
bin(int) 转2进制
chr(int) 转数字为ASCII字符
ord(str) 转字符为ASCII编码

# python对象的方法

**random**

import **random**

random.**random**() #生成0--1之间的随机浮点数

random.**randint**(a,b) #生成a,b之间的随机整数

random.**choice**(list) #对有序数列随机**取样**（list, tuple,str）

random.**sample**(list, 5) # 对有序数列随机**取片段**（对list,随机取5个元素）

random.**shuffle**(list) #随机打乱有序数列

**time**

import **time**

time.**sleep**(1) #间隔1秒休息一次

time.**ctime**() #生成本地时区时间

time.**localtime**() #生成本地时间的时间结构

time.**strftime**('%Y-%m-%d %H:%M:%S', time.localtime()) #格式化时间

str1**.format()**

**f**'str{}'

**str.isnumeric()** #判断是否为可计数类型,一二三也会返回True

str**.isdigit()** #判断是否为数值类型，一二三会返回False

lis**.copy()**

dic**.get('key')**

dic**.keys()**

dic**.values()**

dic**.items()**

dict**.fromkeys(**lis, val**) #根据lis生成一个dict,key为lis成员，用val填充**

# python流程

```
if cond1:
    pass

if cond1:
    pass1
else:
    pass2
```

```
while cond1:
    pass1
else:
    pass2
```

```
for target in expression_list:
    pass1
else:
    pass2
```