**《Python基础版课件》笔记**

# 判断语句

If 条件语句：

...

If 条件语句：

...

Else:

...

If 条件语句：

...

Elif:

...

Else:

...

# 循环语句

# while循环

'''

i = 0

while i < 100:

...

I++

'''

#for 循环

'''

arr = [1, 2, 3, 4, 5]

for n in arr:

...

'''

# 字符串、列表、元祖、字典

## 字符串

#字符串常见的操作

#大部分的字符串操作函数只以返回值得方式返回结果， 不修改原来字符串

#find 如果是返回开始的索引值，否则返回-1

str = "abc def hello ha ha"

print str.find("abc")

print str.find("def")

print str.find("hello")

#index跟find()方法一样，只不过如果str不在 mystr中会报一个异常

print str.index("hell", 0, len(str))

#count 返回 str在start和end之间 在 mystr里面出现的次数

print str.count("e",0 , len(str))

# replace 把 mystr 中的 str1 替换成 str2,如果 count 指定，则替换不超过 count 次

print str.replace("ha", "Ha") #全部替换

print str.replace("ha", "Ha", 1) #替换一个

#split 以 str 为分隔符切片 mystr，如果 maxsplit有指定值，则仅分隔 maxsplit 个子字符串

#默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等

print str.split(" ")

print str.split(" ", 2)

# capitalize 把字符串的第一个字符大写

print str.capitalize()

# title 把字符串的每个单词首字母大写

print str.title()

# startswith 检查字符串是否是以 obj 开头, 是则返回 True，否则返回 False

print str.startswith("abc")

# endswith 检查字符串是否以obj结束，如果是返回True,否则返回 False.

print str.endswith("ha")

# lower 转换 mystr 中所有大写字符为小写

print str.replace("ha", "HA").lower()

# upper 转换 mystr 中所有小写字符为大写

print str.upper()

# ljust 返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串

print str.ljust(20)

# center 返回一个原字符串居中,并使用空格填充至长度 width 的新字符串

print str.center(40)

# lstrip 删除 mystr 左边的空白字符

print str.center(40).lstrip()

# rstrip 删除 mystr 右边的空白字符

print str.center(40).rstrip()

#strip 删除两边的空白字符

print str.center(40).strip()

#rfind 类似于 find()函数，不过是从右边开始查找. mystr.rfind(str, start=0,end=len(mystr) )

print str.rfind("ha")

#rindex 类似于 index()，不过是从右边开始.

print str.rindex("hell")

#partition 把mystr以str分割成三部分,str前，str和str后

print str.partition("hello")

#rpartition 和partiton 功能一样，不过是从右边开始查目标字串

print str.rpartition("e")

#splitlines 按照行分隔，返回一个包含各行作为元素的列表

print " a\n b\n c\n d".splitlines()

#isalpha 如果 mystr 所有字符都是字母 则返回 True,否则返回 False

print "abcd".isalpha()

print "1234abcd".isalpha()

# isdigit 如果 mystr 只包含数字则返回 True 否则返回 False.

print "12345".isdigit()

print "12345a".isdigit()

#isalnum 如果 mystr 所有字符都是字母或数字则返回 True,否则返回 False

#isspace 如果 mystr 中只包含空格，则返回 True，否则返回 False.

#join mystr 中每个字符后面插入str,构造 出一个新的字符串 str.join(mystr)

print "aaa aaa a a".join("\_\_\_\_\_")

## 列表

列表类似于C++中的数组，支持随机访问，支持添加元素，但元素可以是不同类型

namelist = ["aaa", "bbb", "ccc"]

print namelist

print namelist[0]

print namelist[1]

print namelist[2]

#循环遍历

for name in namelist:

print name

i = 0

while i<len(namelist):

print namelist[i]

i+=1

#增加元素 append extend insert

namelist.append("ddd")

print namelist

namelist.extend(["eee","fff"])

print namelist

#insert(index, object) 在index前插入object

namelist.insert(2,["111","222"]) #第三个元素是一个列表

print namelist

namelist.insert(-1,"333")

print namelist

#修改元素 赋值 namelist[i] = xxx

#查找元素in（存在）,如果存在那么结果为true，否则为false not in（不存在），如果不存在那么结果为true，否则false

print "aaa" not in namelist

#index和count与字符串中的用法相同

#删除元素 del, pop, remove

#del：根据下标进行删除

#pop：删除最后一个元素

#remove：根据元素的值进行删除

del namelist[2]

print namelist

namelist.pop()

print namelist

namelist.remove("333")

print namelist

# 排序(sort, reverse) sort 排序 reverse 逆置

a = [1, 4, 2, 3]

a.reverse() #[3, 2, 4, 1]

a.sort() #[1, 2, 3, 4]

a.sort(reverse=True) #[4, 3, 2, 1]

# 列表中的元素可以是列表 相当于二维数组

## 元组

#元组 Python的元组与列表类似，不同之处在于元组的元素不能修改。元组使用小括号，列表使用方括号。

元组的内置函数count, index，index和count与字符串和列表中的用法相同

## 字典--键值对

#字典和列表一样，也能够存储多个数据

#列表中找某个元素时，是根据下标进行的

#字典中找某个元素时，是根据'名字'（就是冒号:前面的那个值，例如上面代码中的'name'、'id'、'sex'）

#字典的每个元素由2部分组成，键:值。例如 'name':'班长' ,'name'为键，'班长'为值

info = {'name':'班长', 'id':100, 'sex':'f', 'address':'北京'}

print info

print info['name']

print info['id']

print info['sex']

print info['address']

#若访问的键值不存在 ，会报错

#get 获取对应键值的内容和

name = info.get('name')

print name

#添加新元素

info['aa'] = 100

print info

# del info['aa'] #删除字典

# print info #字典删除之后再次打印会报错

#

# info.clear() #清除数组

# info['name'] = '组长'

# print info

#

# del info

# # info 已经不存在 不能使用print info

#字典常见操作 len() keys() values() items() has\_key()

print len(info) #长度，有多少个键值对

print info.keys() #所有key

print info.values() #所有value

print info.items() #所有键值对

print info.has\_key('name') #'name'在不在info中

#公共方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 运算符 | Python表达式 | 结果 | 描述 | 支持的类型 |
| + | [1,2]+[3,4] | [1,2,3,4] | 合并 | 字符串、列表、元组 |
| \* | ‘Hi’\*4 | ['Hi!','Hi!','Hi!','Hi!'] | 复制 | 字符串、列表、元组 |
| in | 3in(1,2,3) | true | 元素是否存在 | 字符串、列表、元组、字典 |
| Not in | 4not in(1,2,3) | true | 元素是否不存在 | 字符串、列表、元组、字典 |

# python内置函数

# Python包含了以下内置函数

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 描述 |
| 1 | cmp(item1,item2) | 比较两个值 |
| 2 | len(item) | 计算容器中元素个数 |
| 3 | max(item) | 返回容器中元素最大值 |
| 4 | min(item) | 返回容器中元素最小值 |
| 5 | del(item) | 删除变量 |

# 函数

def fun():

"打印函数"

print "hello word"

help(fun) #会打印函数名下的文字 即 "打印函数"

fun() #调用函数

#函数中调用函数

def fun1():

a = 100

fun2()

print a

def fun2():

a = 10

print a

fun1()

#除了在全局域定义的变量为全局变量外 如果在函数中修改全局变量，那么就需要使用 global 进行声明，否则修改的只是函数内部的局部变量

a = 10

def fun3():

global a

a = 100

fun3()

print a

#返回值有多个时，会返回一个元组

def fun4():

return 3, 4

x = fun4()

print x

#函数中可以使用不定长参数

# def functionname([formal\_args,] \*args, \*\*kwargs):

# 加了星号（\*）的变量args会存放所有未命名的变量参数，aargs为元组；

# 而加\*\*的变量kwargs会存 放命名参数，即形如key=value的参数， kwargs为字典。

def fun5(a, b, \*arg, \*\*kwargs):

print a

print b

print arg

print kwargs

fun5(10, 20, 30, x = 40, y = 50)

#下面第二段代码的结果 意料之外

a = 1

b = a

print a # 1

print b # 1

a = 2

print a #2

print b #1

a = [1, 2] #[1,2]

b = a

print b #[1,2]

a.append(3)

print b #[1,2,3]

# 匿名函数

sum = lambda a, b: a + b

print sum(1,2)

stus = [

{"name":"zhangsan", "age":18},

{"name":"lisi", "age":19},

{"name":"wangwu", "age":17}

]

key = lambda x:x['name']

stus.sort(key = lambda x:x['name'])

print stus

stus.sort(key = lambda x:x['age'])

print stus

# 文件操作

open(‘filename’, ‘flag’)

wirte(string)

read()

readline() #读取单独一行，结尾自动加上\n

f.tell() 返回一个整数，代表文件对象在文件中的指针位置

close()

f = open('test.txt', 'a+')

buf = f.read()

print (buf)

f.close()

f = open('test.txt', 'w')

f.write('\_\_aaaaaaaaaaaa')

f.close()

f = open('test.txt', 'r')

content = 1

i = 1

while content:

content = f.readline()

print ('%s'%content)

i += 1

# 在读写文件的过程中，如果想知道当前的位置，可以使用tell()来获取

# 如果在读写文件的过程中，需要从另外一个位置进行操作的话，可以使用seek()

# seek(offset, from)有2个参数

# offset:偏移量

# from:方向

# 0:表示文件开头

# 1:表示当前位置

# 2:表示文件末尾

# rename(需要修改的文件名, 新的文件名)

# remove(待删除的文件名)

# mkdir("张三")创建文件夹

# getcwd()获取当前目录

# listdir("./")获取目录列表

# rmdir("张三")删除文件夹

# 面向对象

class Car:

name = 'aaa' #属性名可隐藏

def \_\_init\_\_(self): #相当于C++中的默认构造函数，在对象创建时，被自动调用

#self.name = 'xxx'

#print ('创建对象')

pass

# \_\_init\_\_()方法，在创建一个对象时默认被调用，不需要手动调用

# \_\_init\_\_(self)中，默认有1个参数名字为self，如果在创建对象时传递了2个实参

# 那么 \_\_init\_\_(self)

# 中出了self作为第一个形参外还需要2个形参，例如\_\_init\_\_(self, x, y)

# \_\_init\_\_(self)中的self参数，不需要开发者传递，

# python解释器会自动把当前的对象引传递进去

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

msg = 'hello world!'

return msg

def \_\_del\_\_(self): #相当于c++的析构函数

#print ('删除对象')

pass

def move(self):

print ('move')

def stop(self):

print ('stop')

def print\_name(self):

print self.name

**\_\_call\_\_**

一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法\_\_call\_\_()。

# 私有的属性，不能通过对象直接访问，但是可以通过方法访问私有的方法，不能通过对象直接访问

# 私有的属性、方法，不会被子类继承，也不能被访问

# 一般情况下，私有的属性、方法都是不对外公布的，往往用来做内部的事情，起到安全的作用

在大多数情况下，在最简单的情况下，你能想到的搜索属性从父类继承的深度优先，左到右，而不是搜索两次在同一个类层次结构中，其中有一个重叠。。实际上，super() 可以动态的改变解析顺序。

'''

类属性 和 实例属性

类属性：表示这个类的属性

实例属性：表示某个实例的属性

通过类名修改类属性：那么这个类的属性就会变，之后创建的所有实例的这个属性也会变,通过对象修改类属性：相当于只在这个对象中修改了这个类属性，这个类属性拷贝到实例中,之后不管怎么用类名修改类属性，这个实例中的这个属性不会改变（通过对象修改类属性 使得 对这个对象来说，这个类属性变成了实例属性）

'''

BWM = Car()

BWM.name = 'xxx'

BWM.print\_name()

BYD = Car()

BYD.print\_name()

Car.name = 'www'

BYD.print\_name()

BWM.print\_name()

#静态方法和类方法

'''

静态方法和类方法

类方法：需要用修饰器 @classmethod 来标识其为类方法 第一个参数必须是类对象

静态方法：需要通过修饰器 @staticmethod 来进行修饰 静态方法不需要多定义参数

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，

那么通过cls引用的必定是类对象的属性和方法；而实例方法的第一个参数是实例对象self，那么通

过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的

类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态

方法中引用类属性的话，必须通过类对象来引用

'''

class Stu:

name = 'hello world'

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

name = 'aaa'

@classmethod

def print\_name(cls):

print cls.name

@staticmethod #静态方法

def getCountry():

return People.country

tmp = Stu()

tmp.print\_name()

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，

那么通过cls引用的必定是类对象的属性和方法；而实例方法的第一个参数是实例对象self，那么通

过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的

类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态

方法中引用类属性的话，必须通过类对象来引用

new\_\_和\_\_init\_\_ 的作用

\_\_new\_\_ 至少要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供

\_\_new\_\_ 必须要有返回值，返回实例化出来的实例，这点在自己实现 \_\_new\_\_ 时要特别注

意，可以return父类 \_\_new\_\_ 出来的实例，或者直接是object的 \_\_new\_\_ 出来的实例

\_\_init\_\_ 有一个参数self，就是这个 \_\_new\_\_ 返回的实例， \_\_init\_\_ 在 \_\_new\_\_ 的基础上可以完成一些其它初始化的动作， \_\_init\_\_ 不需要返回值

我们可以将类比作制造商， \_\_new\_\_ 方法就是前期的原材料购买环节， \_\_init\_\_ 方法就是在有原材料的基础上，加工，初始化商品环节

\_\_init\_\_其实不是实例化一个类的时候第一个被调用 的方法。当使用 Persion(name, age) 这样的表达式来实例化一个类时，最先被调用的方法 其实是 \_\_new\_\_ 方法。

\_\_new\_\_方法接受的参数虽然也是和\_\_init\_\_一样，但\_\_init\_\_是在类实例创建之后调用，而 \_\_new\_\_方法正是创建这个类实例的方法。

# 单例模式

1. 单例是什么

举个常见的单例模式例子，我们日常使用的电脑上都有一个回收站，在整个操作系统中，回收站只能有一个实例，整个系统都使用这个唯一的实例，而且回收站自行提供自己的实例。因此回收站是

单例模式的应用。

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，单例模式是一种对象创建型模式

创建单例-保证只有1个对象

# 实例化一个单例

class Singleton(object):

\_\_instance = None

def \_\_new\_\_(cls, age, name):

#如果类属性\_\_instance的值为None，

#那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法时

#能够知道之前已经创建过对象了，这样就保证了只有1个对象

class Singleton(object):

\_\_instance = None

def \_\_new\_\_(cls, age, name):

#如果类属性\_\_instance的值为None，

#那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法时

#能够知道之前已经创建过对象了，这样就保证了只有1个对象

if not cls.\_\_instance:

cls.\_\_instance = object.\_\_new\_\_(cls)

return cls.\_\_instance

a = Singleton(18, "dongGe")

b = Singleton(8, "dongGe")

print(id(a))

print(id(b))

a.age = 19 #给a指向的对象添加一个属性

print(b.age)#获取b指向的对象的age属性

4391023224

4391023224

19

创建单例时，只执行1次\_\_init\_\_方法

# 实例化一个单例

class Singleton(object):

\_\_instance = None

\_\_first\_init = False

def \_\_new\_\_(cls, age, name):

if not cls.\_\_instance:

cls.\_\_instance = object.\_\_new\_\_(cls)

return cls.\_\_instance

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

if not self.\_\_first\_init:

self.age = age

self.name = name

Singleton.\_\_first\_init = True

a = Singleton(18, "dongGe")

b = Singleton(8, "dongGe")

print(id(a))

print(id(b))

print(a.age)

print(b.age)

a.age = 19

print(b.age)

# 异常

当Python检测到一个错误时，解释器就无法继续执行了，反而出现了一些错误的提示，这就是所谓的"异常"

捕获异常 try...except...

try ... except 语句可以带有一个 else子句，该子句只能出现在所有 except 子句之后，使用 else 子句比在 try 子句中附加代码要好，因为这样可以避免 try ... except 意外的截获本来不属于它们保护的那些代码抛出的异常。

raise 语句允许程序员强制抛出一个指定的异常。例如:

>>> raise NameError('HiThere')

Traceback (most recent call last):

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

NameError: HiThere

当打开一个不存在的文件时会抛异常

try:

print('-----test--1---')

open('123.txt','r')

print('-----test--2---')

except IOError:

pass

Try语句内执行的是可能抛异常的代码

Except放的是产生错误时，处理的方法

except捕获多个异常

但是要注意 上面代码捕获的只是IO异常，如果有别的异常，并不能捕获

如果想捕获多种异常，可以这样写 expect(IOError, NameError)

获取异常的信息描述

Try:

Print(a)

Except NameError as result

Print(result)

捕获所有异常

Try:

Print(a)

Except

Print(‘产生了一个异常’)

Try：

Open(“a.out”)

Except Exception as result:

Print(“捕获到了异常”)

Print(result)

else咱们应该对 else 并不陌生，在if中，它的作用是当条件不满足时执行的实行；同样在

try...except...中也是如此，即如果没有捕获到异常，那么就执行else中的事情

try:

num = 100

print num

except NameError as errorMsg:

print('产生错误了:%s'%errorMsg)

else:

print('没有捕获到异常，真高兴')

try...finally...语句用来表达这样的情况：

在程序中，如果一个段代码必须要执行，即无论异常是否产生都要执行，那么此时就需要使

用finally。 比如文件关闭，释放锁，把数据库连接返还给连接池等

import time

try:

f = open('test.txt')

try:

while True:

content = f.readline()

if len(content) == 0:

break

time.sleep(2)

print(content)

except:

#如果在读取文件的过程中，产生了异常，那么就会捕获到

#比如 按下了 ctrl+c

pass

finally:

f.close()

print('关闭文件')

except:

print("没有这个文件")

如果try嵌套，那么如果里面的try没有捕获到这个异常，那么外面的try会接收到这个异常，然

后进行处理，如果外边的try依然没有捕获到，那么再进行传递。。。

如果一个异常是在一个函数中产生的，例如函数A---->函数B---->函数C,而异常是在函数C中产

生的，那么如果函数C中没有对这个异常进行处理，那么这个异常会传递到函数B中，如果函

数B有异常处理那么就会按照函数B的处理方式进行执行；如果函数B也没有异常处理，那么这

个异常会继续传递，以此类推。。。如果所有的函数都没有处理，那么此时就会进行异常的默

认处理，即通常见到的那样

# 模块

Import module1, module2

相当于c语言中的

#include<xxx.h>

调用时必须这样引种 模块名.函数名

因为可能存在这样一种情况：在多个模块中含有相同名称的函数，此时如果只是通过函数名来调用，解释器无法知道到底要调用哪个函数。所以如果像上述这样引入模块的时候，调用函数必须加上模块名

import math

#这样会报错

print sqrt(2)

#这样才能正确输出结果

print math.sqrt(2)

from 模块名 import 函数名1,函数名2....

有时候我们只需要用到模块中的某个函数，只需要引入该函数即可，此时可以用下面方法实现：

from modname import \*

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

定位模块

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

1. 当前目录

2. 如果不在当前目录，Python则搜索在shell变量PYTHONPATH下的每个目录。

3. 如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/

4. 模块搜索路径存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录

# 其他

Python文件中的\_\_name\_\_变量

在测试是，当test.py import 了 test1.py时，假如test1.py中有测试代码，那么两个py文件的测试代码就会混到一起

怎么解决呢？

Python文件中都会有一个\_\_name\_\_变量，这个变量在他所在的文件中时，print一下 打印出来的是\_\_main\_\_,但被别的文件import是，还会打印，但是打印的内容就会变为它所在文件的文件名。

所以，在测试代码前加上 if \_\_name\_\_ == ‘\_\_main\_\_’很有必要。

**包**

1.1 有2个模块功能有些联系

1.2 所以将其放到同一个文件夹下

1.3 使用import 文件.模块 的方式导入

1.4 使用from 文件夹 import 模块 的方式导入 （这时并不会引入）

1.5 在msg文件夹下创建 \_\_init\_\_.py 文件

1.6 在 \_\_init\_\_.py 文件中写入 \_\_all\_\_ = [“py文件1”,”py文件2”]

1.7 重新使用from 文件夹 import 模块 的方式导入（引入成功）

总结：

包将有联系的模块组织在一起，即放到同一个文件夹下，并且在这个文件夹创建一个名字为 \_\_init\_\_.py 文件，那么这个文件夹就称之为 包有效避免模块名称冲突问题，让应用组织结构更加清晰

2. \_\_init\_\_.py 文件有什么用\_\_init\_\_.py 控制着包的导入行为

2.1 \_\_init\_\_.py 为空仅仅是把这个包导入，不会导入包中的模块

2.2 \_\_all\_\_在 \_\_init\_\_.py 文件中，定义一个 \_\_all\_\_ 变量，它控制着 from 包名 import \*时导入的模块

2.3 可以在 \_\_init\_\_.py 文件中编写内容可以在这个文件中编写语句，当导入时，这些语句就会被执行

使用pip来安装模块

pip 是 python 自带的一个软件，相当于手机里的应用市场，可以用来安装、卸载、搜索 python 的常见模块。

直接输入 pip 后回车，可以查看 pip 命令的所有可用参数。

1.安装模块

在命令行执行

pip install 模块名

2.卸载模块

在命令行执行

pip uninstall 模块名

3.搜索模块

在命令行执行

pip search 模块名