函数：

定义函数，使用 def 关键字。

结构：

def 函数名(参数列表...):

语句...

return 返回值...

函数的参数：

1. 不定长参数： \*args
2. 关键字参数：\*\*kw
匿名函数: 关键字 **lambda** eg：qm = **lambda** x: x \*\* x
偏函数: 导入模块 **import** functools
functools.partical()就是创建一个新的函数，不需要自行定义函数，直接将结果赋值给一个变量，而这个变量就是一个函数。这个函数的目的是将默认参数给固定住。
回调函数：定义函数时,将函数名作为参数传递过来,然后在函数里边再次调用函数

常用函数：

abs(): 返回数字的绝对值
max(): 返回给定参数中的最大值
min():返回给定参数中最小的值
pow(): 求幂(求次方) [求3^4 print(3\*\*4) print(pow(3, 4))]
round():四舍五入,如果只有一个参数,默认不保留小数点.参数二: 小数点后保留的位数

**import** math
math.ceil():向上取整
math.floor():向下取整
math.sqrt():求开方
math.modf():返回参数的整数部分和小数部分

**import** random
random.choice():返回列表,元祖,字典中的某一个值
random.choices():返回列表,元祖,字典中的指定个数值个值 k 关键字参数
random.random():返回0-1之间的数, [0, 1)
random.uniform():返回指定的一个区间范围的随机数 [, )
random.randint():返回指定区间范围的随机整数 [, ]
random.shuffle():将列表中的数据进行打乱排序

变量作用域（LEGB）：

L （Local） 局部作用域

E （Enclosing） 闭包函数外的函数中

G （Global） 全局作用域

B （Built-in） 内建作用域

装饰器：

本质还是函数, 一般使用装饰器来装饰其他函数.本身还是函数的调用
**@property**
**@属性名.setter**
广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

迭代器与生成器：

迭代器

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器

生成器

yield 函数被称为生成器（generator）, 生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器 , 每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回 yield 的值, 并在下一次执行 next() 方法时从当前位置继续运行。调用一个生成器函数，返回的是一个迭代器对象。

IO操作：

IO操作，使用open() 函数，还可以使用with上下文关键字来确保打开的文件操作符被关闭。

with open(path, mode [, encoding]) as f:

语句操作...

f.read([size]) 从文件读取指定的字节数，如果未给定或为负则读取所有。
f.write() 写文件 *# 如果要写入字符串以外的数据,先将他转换为字符串*
f.close() 关闭文件
f.readline([size]) 读取整行内容,包括\n字符, 如果给定参数则从文件开始读取指定的字符个数
f.readlines([sizeint]) 读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比 sizeint 较大, 因为需要填充缓冲区。
f.tell() 返回一个整数,表示当前文件指针的位置(就是到文件头的比特数).
f.seek(偏移量,[起始位置]) 用来移动文件指针 *#偏移量:单位:比特,可正可负起始位置:0-文件头,默认值;1-当前位置;2-文件尾*

mode的模式：



异常处理：

**try**:
try语句
**except** 异常类1 **as** 变量1:
异常处理语句1

**except** 异常类2 **as** 变量2:
异常处理语句2

...

[**else**:

else语句]
[**finally**:
finally语句]

操作文件和目录：

os模块:在这个模块中给我们封装好了系统操作的功能函数(方法)

import os

os.getcwd() 返回当前的工作目录 # 获取绝对路径

os.listdir(path) 返回指定的文件夹包含的文件或文件夹的名字的列表

os.mkdir(path) 在当前目录下创建新的目录

os.rmdir(path) 删除目录, 只能删除空目录

os.rename(src, dst) 对文件进行重命名

os.stat(file) 获取文件属性

os.remove(file) 删除文件 #可以将本文件删除

os.path.join(path1, path2) 路径拼接

os.path.split(path) 拆分路径 # 返回一个元组(目录, 最后一个文件/目录名称)

os.path.splitext(path) 获取文件的或站名

os.path.exists(path) 判断文件/目录是否存在

os.path.isfile(path) 判断是否是文件

os.path.getsize(file) 获取文件的大小

os.path.dirname(path) 获取当前文件的目录

os.path.basename(path) 获取当前文件/目录名

StringIO：

很多时候，数据读写不一定是文件，也可以在内存中读写

StringIO模块主要用于在内存缓冲区中读写数据。模块是用类编写的，只有一个StringIO类，

# 所以它的可用方法都在类中。此类中的大部分函数都与对文件的操作方法类似。

**>>> from** io **import** StringIO
**>>>** f = StringIO()
**>>>** f.write('hello')
5
**>>>** f.write(' ')
1
**>>>** f.write('world!')
6
**>>>** print(f.getvalue())
hello world!

\*\*getvalue()\*\*方法用于获得写入后的str。

BytesIO：

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

BytesIO实现了在内存中读写bytes，我们创建一个BytesIO，然后写入一些bytes

**>>> from** io **import** BytesIO
**>>>** f = BytesIO()
**>>>** f.write('中文'.encode('utf-8'))
6
**>>>** print(f.getvalue())
b'\xe4\xb8\xad\xe6\x96\x87'

常用的日期和时间函数：

time(时间)

time.localtime() 获取本地时间
time.asctime() / time.ctime() 将时间转换成用户可读时间, 返回str格式 (Thu Apr 19 23:02:35 2018)
time.strftime('%Y-%m-%d %X') 字符串的格式化输出, 返回str格式 2018-04-19 23:07:51
time.strptime(timeObject, '%Y-%m-%d %X') *# 将字符串的时间格式转换成元祖类型的时间格式*

datetime(日期)

datetime.datetime.now() 获取当前时间
datetime.datetime(2020, 2, 2, 20, 20, 20) 获取指定的时间
date1.strftime('%Y-%m-%d %X') 时间字符串的格式化输出, 返回str格式

calendar(日历)

calendar.month(2018, 4) 获取指定的月份日历
calendar.calendar(2018) 获取指定的年份日历

面向对象：

面向对象的三个基本特征是：**封装、继承、多态**

**封装**：

​封装是实现面向对象程序设计的第一步，封装就是将数据或函数等集合在一个个的单元中（我们称之为类）。被封装的对象通常被称为抽象数据类型。 *隐藏复杂的现实细节，暴露简单的调用接口。*

封装的意义：封装的意义在于保护或者防止代码（数据）被我们无意中破坏。在面向对象程序设计中数据被看作是一个中心的元素并且和使用它的函数结合的很密切，从而保护它不被其它的函数意外的修改。

保护数据成员，不让类以外的程序直接访问或修改，只能通过提供的公共的接口访问==>数据封装。

方法的细节对用户是隐藏的，只要接口不变，内容的修改不会影响到外部的调用者==>方法封装。

当对象含有完整的属性和与之对应的方法时称为封装。

从对象外面不能直接访问对象的属性，只能通过和该属性对应的方法访问。

对象的方法可以接收对象外面的消息。

**继承**：

​ 从已有的类创建新类的过程 提供继承信息的称为父类(超类/基类) 得到继承信息的称为子类(派生类)。继承主要实现重用代码，节省开发时间。

在类名后边的括号里一般写的是基类的类名

子类中继承自父类(基类),则父类的成员属性和成员方法都可以被子类所继承

父类中私有的成员属性不能继承给子类的

**多态**：

“一个接口，多种表现形式”。

​ 同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。

多态的三个条件:

继承的存在(继承是多态的基础,没有继承就没有多态)

子类重写父类的方法(多态下调用子类重写的方法)

父类引用变量指向子类对象(子类到父类的类型转换)

**数据抽象** 一 找到和对象相关的数据一属性( 名词）

@property

@属性.setter

**\_*slots*\_**

​限定自定义类型的对象只能绑定某些属性，可以通过在类中定义\_\_slots\_\_变量来进行限定。需要注意的是\_\_slots\_\_的限定只对当前类的对象生效，对子类并不起任何作用。

**class** **Person**(object):
*# 限定Person对象只能绑定\_name, \_age和\_gender属性*
\_\_slots\_\_ = ('\_name', '\_age', '\_gender')

**行为抽象** 一 找到和对象相关的行为一方法（动词）

成员方法：第一个参数为self，由实例化的对象调用

类方法：使用@classmethod装饰的方法是类方法，与成员方法的区别在于所接收的第一个参数不是 self （类实例的指针），而是cls（当前类的具体类型）

静态方法：使用@staticmethod装饰的方法是类静态方法，其跟成员方法的区别是没有 self 参数，并且可以在类不进行实例化的情况下调用

魔法方法：

\_\_init\_\_()

\_\_del\_\_()

\_\_str\_\_()

\_\_repr\_\_()

正则表达式：

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 解释 | 示例 | 说明 |
| . | 匹配任意字符 | b.t | 可以匹配bat / but / b#t / b1t等 |
| \w | 匹配字母/数字/下划线 | b\wt | 可以匹配bat / b1t / b\_t等  但不能匹配b#t |
| \s | 匹配空白字符（包括\r、\n、\t等） | love\syou | 可以匹配love you |
| \d | 匹配数字 | \d\d | 可以匹配01 / 23 / 99等 |
| \b | 匹配单词的边界 | \bThe\b |  |
| ^ | 匹配字符串的开始 | ^The | 可以匹配The开头的字符串 |
| |匹配字符串的结束|.exe | 可以匹配.exe结尾的字符串 |  |  |
| \W | 匹配非字母/数字/下划线 | b\Wt | 可以匹配b#t / b@t等  但不能匹配but / b1t / b\_t等 |
| \S | 匹配非空白字符 | love\Syou | 可以匹配love#you等  但不能匹配love you |
| \D | 匹配非数字 | \d\D | 可以匹配9a / 3# / 0F等 |
| \B | 匹配非单词边界 | \Bio\B |  |
| [] | 匹配来自字符集的任意单一字符 | [aeiou] | 可以匹配任一元音字母字符 |
| [^] | 匹配不在字符集中的任意单一字符 | [^aeiou] | 可以匹配任一非元音字母字符 |
| \* | 匹配0次或多次 | \w\* |  |
| + | 匹配1次或多次 | \w+ |  |
| ? | 匹配0次或1次 | \w? |  |
| {N} | 匹配N次 | \w{3} |  |
| {M,} | 匹配至少M次 | \w{3,} |  |
| {M,N} | 匹配至少M次至多N次 | \w{3,6} |  |
| | | 分支 | foo|bar | 可以匹配foo或者bar |
| (?#) | 注释 |  |  |
| (exp) | 匹配exp并捕获到自动命名的组中 |  |  |
| (?<name>exp) | 匹配exp并捕获到名为name的组中 |  |  |
| (?:exp) | 匹配exp但是不捕获匹配的文本 |  |  |
| (?=exp) | 匹配exp前面的位置 | \b\w+(?=ing) | 可以匹配I’m dancing中的danc |
| (?<=exp) | 匹配exp后面的位置 | (?<=\bdanc)\w+\b | 可以匹配I love dancing and reading中的第一个ing |
| (?!exp) | 匹配后面不是exp的位置 |  |  |
| (? | 匹配前面不是exp的位置 |  |  |
| \*? | 重复任意次，但尽可能少重复 | a.\*b  a.\*?b | 将正则表达式应用于aabab，前者会匹配整个字符串aabab，后者会匹配aab和ab两个字符串 |
| +? | 重复1次或多次，但尽可能少重复 |  |  |
| ?? | 重复0次或1次，但尽可能少重复 |  |  |
| {M,N}? | 重复M到N次，但尽可能少重复 |  |  |
| {M,}? | 重复M次以上，但尽可能少重复 |  |  |

Python提供了re模块来支持正则表达式相关操作，下面是re模块中的核心函数。

|  |  |
| --- | --- |
| 函数 | 说明 |
| compile(pattern, flags=0) | 编译正则表达式返回正则表达式对象 |
| match(pattern, string, flags=0) | 用正则表达式匹配字符串 成功返回匹配对象 否则返回None |
| search(pattern, string, flags=0) | 搜索字符串中第一次出现正则表达式的模式 成功返回匹配对象 否则返回None |
| split(pattern, string, maxsplit=0, flags=0) | 用正则表达式指定的模式分隔符拆分字符串 返回列表 |
| sub(pattern, repl, string, count=0, flags=0) | 用指定的字符串替换原字符串中与正则表达式匹配的模式 可以用count指定替换的次数 |
| fullmatch(pattern, string, flags=0) | match函数的完全匹配（从字符串开头到结尾）版本 |
| findall(pattern, string, flags=0) | 查找字符串所有与正则表达式匹配的模式 返回字符串的列表 |
| finditer(pattern, string, flags=0) | 查找字符串所有与正则表达式匹配的模式 返回一个迭代器 |
| purge() | 清除隐式编译的正则表达式的缓存 |
| re.I / re.IGNORECASE | 忽略大小写匹配标记 |
| re.M / re.MULTILINE | 多行匹配标记 |

网络编程

协议- protocol- 通信双方对话的规范和标准

IP.Internet Protocol- 寻址和路由 www.baidu.com---> 180.97.33.108

TCP和UDP是在IP协议之上构建的传输协议它们能够提供传输数据的服务 TCP提供了可靠传输服务 TCP- Transfer Control Protocol 握手机制+冗余校验---> 重发一次

TCP.Transfer Control Protocol I 1.可靠通信[数据不传丢也不传错] 滑动窗口机制 2.流量控制(自动调节发送数据的速度 3.拥塞控制(网络拥堵时会降低发送速度)

QQ 微信.应用级 QQ--- ICQ--- OICQ HTTP--- Hyper-Text Transfer Protocol HTTPS--- Secure SMTP-- Simple Mail Transfer Protocol POP3--- Post Office Protocol version 3 IMAP---Internet Mail Access Protocol xerox--- macintosh--- Windows

URL--- Uniform Resource Locator 协议://域名或IP地址:80/路径/资源名

网络API(应用程序编程接口) / 网络

requests.get()用于请求目标网站，类型是一个HTTPresponse类型
**import** requests
response = requests.get('http://www.baidu.com')
print(response.status\_code) *# 打印状态码*
print(response.url) *# 打印请求url*
print(response.headers) *# 打印头信息*
print(response.cookies) *# 打印cookie信息*
print(response.text) *#以文本形式打印网页源码*
print(response.content) *#以字节流形式打印*

TCP编程

Socket是网络编程的一个抽象概念。通常我们用一个Socket表示“打开了一个网络链接”，而打开一个Socket需要知道目标计算机的IP地址和端口号，再指定协议类型即可。

服务器端

from socket import socket, AF\_INET, SOCK\_STREAM, SOCK\_DGRAM

from datetime import datetime

def main():

# 1、创建套接字对象，并指定使用哪种传输服务

# AF\_INET：IPV4 AF\_INET6：IPV6 SOCK\_STREAM: TCP SOCK\_DGRAM: UDP SOCK\_RAW: 原始套接字

server\_socket = socket(family=AF\_INET, type=SOCK\_STREAM)

# 2、绑定IP地址和端口，建议使用 1024 以后的端口

print('正在绑定IP地址和端口......')

server\_socket.bind(('10.7.152.89', 9999))

print('服务器绑定IP地址和端口成功')

# 3、启动监听 监听客户端有没有连接到服务器 512：历史经验最佳值

print('正在启动服务器......')

server\_socket.listen(512)

print('服务器启动成功')

# 4、等待连接 通过循环等待客户端连接并作出相应处理

while True:

# 阻塞并等待连接

client\_socket, addr = server\_socket.accept()

print(str(addr) + '连接到了服务器.')

# 5、发送数据

client\_socket.send(str(datetime.now()).encode('utf-8'))

# 6、断开连接

client\_socket.close()

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

main()

客户端

from socket import socket, AF\_INET, SOCK\_STREAM

def main():

# 1、创建套接字 socket

client\_socket = socket(family=AF\_INET, type=SOCK\_STREAM)

# 2、连接服务器

client\_socket.connect(('10.7.152.89', 9999))

# 3、循环发送和接收信息

while True:

info = client\_socket.recv(1024).decode('utf-8')

print(info)

# client\_socket.close()

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

main()

多线程、多进程

线程：是操作系统分配CPU的基本单元

进程：是操作系统分配内存的基本单元，一个进程由一个或多个线程构成

多进程、多线程：加速程序执行，提升性能；改善用户体验

编写麻烦，调试更麻烦；对其他进程、线程不友好；占用更多资源

单进程 / 单线程模拟下载任务例子：



多进程模拟下载任务例子：



多线程模拟下载任务例子：

