文件

主要内容

- 文件的读写
  - 绝对路径和相对路径
  - 二进制文件
  - with语句
  - 文本文件
- 序列化和反序列化：
  - pickle、json和csv模块
  - struct模块

文件：二进制和文本文件

按照数据的组织形式，文件可分为文本文件和二进制文件

- 二进制文件
  - 存储的是一个一个的**字节**，无法用常规的文本编辑器读写
  - 包含：数据库文件、图像文件、可执行文件、音视频文件、Office文档等
- 文本文件
  - 存储文本**字符串**，由**若干行**组成，行之间以**行分隔符**(EOL, linebreak)隔开
  - 可通过文本编辑器正常显示、编辑，人类能够直接阅读和理解
  - 文本文件里面的字符采用某种**编码方式**编码后保存到存储设备上
- 文本文件的行分隔符：
  - **拓展：**不同操作系统使用的行分隔符可能各不一样
  - python解释器在进行**文本文件的读写**时统一用**\n作为行分隔符**，会自动进行相应的转换
  - 为什么有回车换行？早期英文打字机上：
    - 回车符(Carriage Return, \r): 回到carriage开始的位置（最左边)
    - 换行符(Line Feed, \n): carriage往下移动一行

拓展：行分隔符示例

系统缺省的行分隔符
- Windows系统为'\r\n'
- Linux系统为'\n'
- MacOS为'\r'

```
>>> import os
>>> os.linesep
'\r\n'
>>> os.sep
'\\'
```

```
$ cat unix.txt
line 1
line 2

$ hexdump -C unix.txt
00000000  6c 69 6e 65 20 31 0a 6c  69 6e 65 20 32 0a 0a     |line 1.line 2..|
0000000f
$ hexdump -C dos.txt
00000000  6c 69 6e 65 20 31 0d 0a  6c 69 6e 65 20 32 0d 0a  |line 1..line 2..|
00000010  0d 0a                                             |..|
00000012
$ hexdump -C mac.txt
00000000  6c 69 6e 65 20 31 0d 6c  69 6e 65 20 32 0d 0d     |line 1.line 2..|
0000000f
```

文件和目录

```
绝对路径名: C:\Users\tom\AppData\Roaming\Python\Python36\site-packages
相对路径名:  pj2.py       projects\pj1.py
```

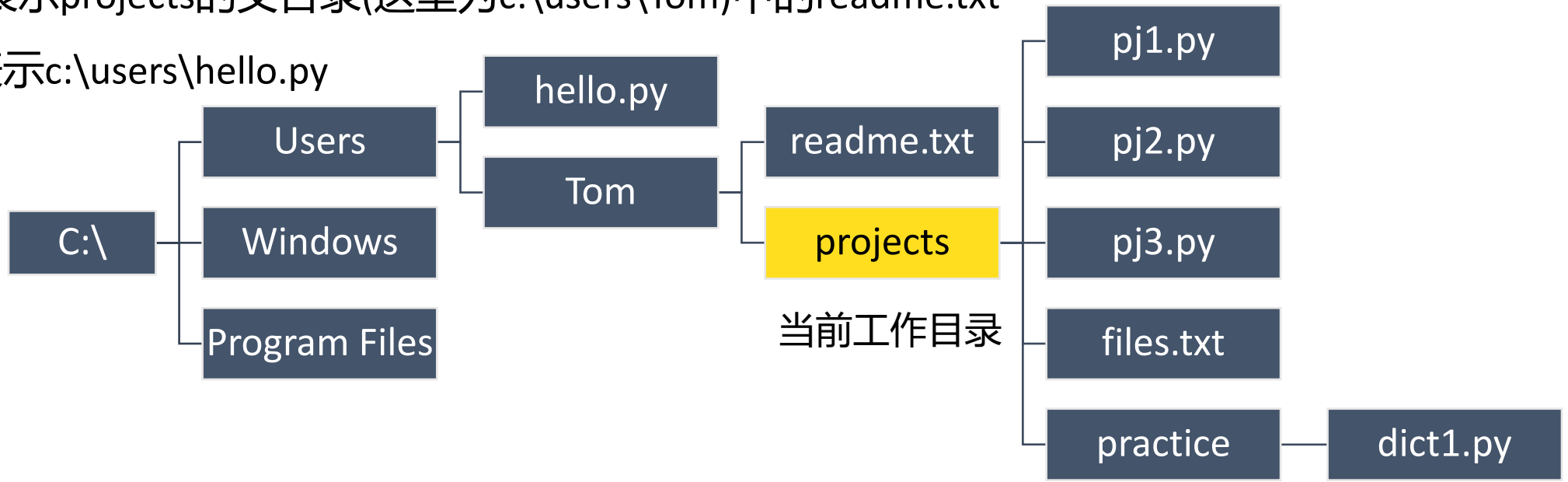
- 文件以目录（文件夹）的方式组织成目录树
  - 一个目录可包含多个文件，也可以包含子目录，这些子目录再包含文件或子目录
- Linux和MacOS系统中只有一棵目录树，而微软操作系统中对于每个驱动器维护一棵目录树(C:; D:...)
- 如何标识或者找到目录树中的文件或目录?
  - 可以从目录树中根节点开始，按照树的结构找到其子节点，如此继续直到到达指定的位置
  - 路径名(pathname): 从起点开始直到结束位置途中经过的节点名字组合在一起，之间通过**路径分隔符**隔开
    - Linux和MacOS系统的路径分隔符为'/'，而微软系统中为r'\'
    - **在实践中都可以使用/分割路径名，python会转换为操作系统所采用的路径分隔符**
  - **绝对路径名**: 以目录树中的根开始的路径名，也就是以路径分隔符开始
  - **相对路径名**：不以目录树中的根开始的路径名
    - 引入**工作目录**的概念，相对路径名从工作目录开始
  - 特殊的目录名： .(一个点)表示当前目录，..(两个点)表示当前目录的**父目录**

文件和目录：工作目录和相对路径

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Tom\\projects'
```

假设当前工作目录为projects:

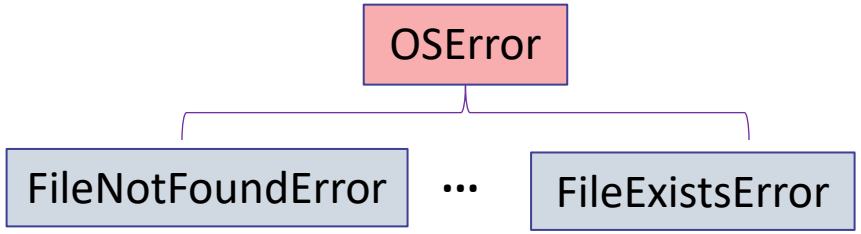
- files.txt 表示projects下的files.txt
- practice\dict1.py表示practice子目录下的dict1.py
- ..\readme.txt 表示projects的父目录(这里为c:\users\Tom)中的readme.txt
- ..\..\hello.py 表示c:\users\hello.py



- 绝对路径：你的应用程序移动到任何位置都可以访问。如果复制到另外一台机器，绝对路径有变化。
- **相对路径**：你的应用程序和文件可以**整体移动**到另外一个位置、另外一台机器，还能够访问

文件操作：打开文件

```
>>> import os
>>> os.path.exists('tmp.txt')
True
```



文件读写的第一步是打开文件

open(file, mode='r', encoding=None, errors=None, newline=None)

以指定模式打开相应的文件，返回某种类型的文件对象。出错时抛出异常OSError

- 文件名file: 可采用相对路径或绝对路径
- 访问模式(mode)：第2个参数，打开文件后的处理方式
  - 可以执行的操作：rwxa必须指定一个，且只能指定一个
  - 可选的+，表示读写模式，需要指定rwxa中的某一个
  - 可选的文件种类：文本或二进制，t或b。缺省为t
  - 缺省'r'，相当于'rt'，即打开文本文件读
  - 操作系统维护**文件指针**，给出了下一次的读写操作所对应的位置。 a模式打开时，指针为文件的尾部，否则为0
- 编码方式(encoding)：以文本方式打开时所采用的编码方式
  - 缺省为**locale. getpreferredencoding( )返回的编码方式**
  - 如果为'cp936'，表示GBK
- errors: 文本文件读写时编码或解码出现错误时怎样处理

值	说明
r	读模式，不存在时异常**FileNotFoundError**
w	写模式，文件存在时会先**截取覆盖原有文件！！！**不存在时创建
x	写模式，文件存在时抛异常**FileExistsError**，不存在时创建
a	**附加模式**，打开写，不存在时会创建，存在时会附加到文件尾部
+	读写模式，较少使用！！可使用r+ w+ x+ a+，注意w会清空原有文件的内容
t	文本模式
b	二进制模式

```
以读模式打开文本文件 f = open('a.txt')
以写模式打开文本文件 f = open('a.txt', 'w')
以读模式打开文本文件，编码方式为'utf-8'
    f = open('hello.py', encoding='utf-8')
```

文件操作：打开文件

- 以读写模式打开文件 f = open('b.csv', 'r+')
- 以读写附加模式打开文件 f = open('b.csv', 'a+')
- 以读模式打开二进制文件 f = open('c.dat', 'rb')
- 以写模式打开二进制文件 f = open('c.dat', 'wb')
- 以附加模式打开二进制文件 f = open('c.dat', 'ab')
- 文件对象在io模块中实现

```
>>> f = open('sample.txt')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='r'
encoding='cp936'>
>>> fw = open('sample.txt','wb')
>>> fw
<_io.BufferedWriter name='sample.txt'>
>>> fr = open('sample.txt','rb')
>>> fr
<_io.BufferedReader name='sample.txt'>
```

- 这些不同类型的文件对象提供的接口类似，但具体实现有所区别
  - 文件对象有多个属性，记录了文件打开时的一些参数
  - 文件对象有读写等方法，如果打开模式不允许这些方法，调用时抛出异常io.UnsupportedOperation

属性	描述
name	文件名
mode	打开模式
encoding	编码方式，仅对文本方式有效
errors	编解码出错时的处理方式
closed	是否已经关闭

```
def print_file_property(file='sample.txt'):
    fh = open(file)
    print('file name   : %s' % fh.name)
    print('access mode: %s' % fh.mode)
    print('encoding   : %s' % fh.encoding)
    print('closed     : %s' % fh.closed)
    fh.close()
```

file_property.py

文件对象的常用操作

	方法	说明
读	f.read([size=-1)	读取最大size个字符(字节)或EOF，size为-1时表示文件中所有字符(字节)
	f.readline()	读取字符直到遇到换行符或者EOF，如果读时马上遇到EOF，则返回空字符串。注意在读时遇到\n时，\n也包括在内
	f.readlines()	读取直到EOF，返回一个列表，每个元素为读到的每行内容(包括行尾换行符\n)
	next(f)	以读方式打开的文件是一个**迭代器**，下一个元素为文件中下一行的内容(包括\n)，直到EOF为止
写	f.write((text)	将字符串(字节串)text写入文件，返回写入的字符(字节)个数
	f.writelines(lines)	把lines中的每个元素逐个写入文件，要求list的元素必须为字符串，返回None
其他	f.flush()	缓冲区的内容同步到存储设备
	f.close()	关闭文件，释放资源，写模式下还同步缓冲区内容到存储设备
	f.truncate([size])	文件截取后长度变为size，缺省为当前文件指针位置(文件打开时初始为0)，返回文件的当前长度
	f.tell()	访问打开文件的当前指针位置
	f.seek(offset, whence=0)	移动文件指针位置到基准位置的offset，缺省whence=0，表示从头部开始，whence=1,表示当前指针位置, whence=2表示文件结尾开始

二进制(binary)文件的读写

- open时mode包含字符b

```
file = open('sample.dat', 'rb')
file = open('sample.dat', 'wb')
```

- read(size=-1) **返回bytes对象**
  - 不传递参数或者size<0时表示从当前位置读直到文件结束EOF (end of file)
  - 从当前位置读指定数量的字节或文件结束EOF为止
  - 返回读取的数据(bytes对象)
  - 如果读时已经为EOF，返回**空bytes对象**
- write(bytes_obj)**传递的是bytes对象**，将该对象中的数据写入文件，返回写入的字节数

```
def write_binary_file(file='sample.dat'):
    f = open(file,'wb')
    bufs = bytes(range(256))
    f.write(bufs)
    bufs = bytes(range(255, -1, -1))
    f.write(bufs)
    f.close()
```

file_binary.py

```
def read_binary_file(file='sample.dat'):
    with open(file, 'rb') as f:
        s = f.read()
        print(s)
```

```
f = open(file)
process(f)
f.close()
```

```
with open(file) as f:
    process(f)
```

建议的方式，由解释器帮你调用close()关闭文件

- close(): 关闭文件，释放资源，写模式下还同步缓冲区内容到存储设备
- flush(): 缓冲区的内容同步到存储设备

with语句

```
with context_expr [as obj]:
    with_block
```

- with语句用于上下文管理：
  - 首先执行后面的表达式，得到一个对象obj
  - 执行obj.__enter__( )进入上下文
  - 执行with block的代码
  - 执行完成后，不管异常是否出现都会调用obj.__exit__( )结束上下文以释放资源
- 对于文件对象而言，with fileobj: body，最后会关闭该文件

with语句等价于如下代码

```
obj = context_expr
obj = obj.__enter__()
try:
    with_block
finally:
    obj.__exit__()
```

- 不仅仅是文件对象，用户也可以自己实现某个资源的上下文管理:
  - 依赖python语言的magic method，实现**上下文管理协议的__enter__()和__exit__()**方法
  - 或者通过import contextlib，并使用装饰器@contextlib.contextmanager方式实现

tee_file.py

```
def tee(in_file, out_file1, out_file2):
    with (open(in_file, 'rb') as file,
          open(out_file1, 'wb') as file1,
          open(out_file2, 'wb') as file2):
        text = file.read()
        file1.write(text)
        file2.write(text)
```

with语句：通过魔术方法实现上下文管理

拓展的内容

- Timer：记录程序的运行时间

context.py

```
class Timer(object):
    def __init__(self):
        pass

    def __enter__(self):
        self.start = time.time()
        print('执行开始时刻: ', time.strftime('%H:%M:%S', time.localtime(self.start)))

    def __exit__(self, exception_type, exception_val, trace):
        stop = time.time()
        print('执行结束时刻: ', time.strftime('%H:%M:%S', time.localtime(stop)))
        print("耗时:%.4f" % (stop - self.start))
```

```
with Timer():
    for i in range(10000000):
        pass
```

with语句：通过装饰器实现上下文管理

拓展的内容

- contextmanager是一个**装饰器**，在time_print函数外部封装了一层壳，使得其可用在with语句中
  - 在进入上下文时执行直到yield语句为止
  - 在退出上下文时从yield语句后恢复执行

context.py

```
import contextlib
import time

@contextlib.contextmanager
def time_print(task_name):
    t = time.time()
    try:
        yield
    finally:
        print('%s took %.4f seconds' % (task_name, time.time() - t))

with time_print("processes"):
    for i in range(10000000):
        pass
```

文件读写：移动文件指针

- `tell()`：返回当前文件的指针所在位置(距离文件开始多少个字节)
- `seek(offset, whence=0)`
  - 将文件指针移动到文件的某个位置（相对某个参考位置的偏移量）
  - whence=0表示从文件开始；whence=1表示当前位置； whence=2表示文件结尾
  - 缺省为从文件开始处算起的第几个字节，即whence=0
  - 对于以文本方式打开的文件对象而言，whence=1或2时，offset必须为0，不支持非0的偏移

```python
import random
def seek_file(file='sample.dat'):
    with open(file, 'rb') as f:
        f.seek(0, 2)
        size = f.tell()
        print('len:', size)
        offset = random.randint(0, size - 4)
        f.seek(offset, 0)
        print(f.read())
```

file_binary.py

主要内容

- 文件的读写
  - 绝对路径和相对路径
  - 二进制文件
  - with语句
  - 文本文件
- 序列化和反序列化：
  - pickle、json和csv模块
  - struct模块

文本文件的读写

- open时mode缺省等价于'rt'，以文本方式打开读。
  - 打开写可以采用模式w/x/a
  - rwxa之一再加上可选的+表示读写模式，比较少使用
- open时不指定编码方式时，表示encoding缺省为locale.getpreferredecoding()

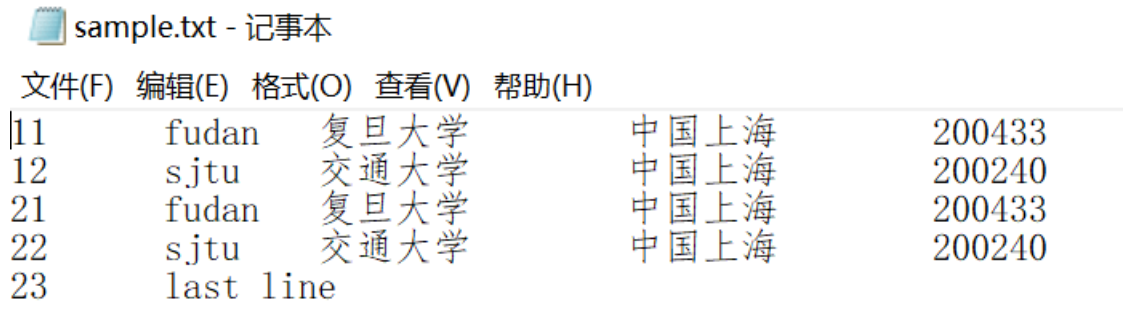
```
file = open('sample.txt')
file = open('sample.txt','w')
file = open('sample.py', encoding='utf-8')
```

- write(str_obj): 将字符串的内容str_obj按照打开文件时指定的编码方式转换为字节串后写入文件，**返回写入的字符个数**
- writelines(lines)：
  - 传递的参数为一个**可迭代对象，其中的元素为字符串**，按照顺序将所有的字符串写入文件
  - 虽然名字叫writelines，但可迭代对象的元素写入时**并不会自动添加换行符**

```
def write_lines(lines):
    for line in lines:
        f.write(line)
```

文本文件的读写

```
def write_file_2(file='sample.txt'):
    s1='11\tfudan\t复旦大学\t中国上海\t200433\n'
    s2='12\tsjtu\t交通大学\t中国上海\t200240\n'
    with open('sample.txt','w') as f:
      f.write(s1)
      f.write(s2)
```




```
def write_file_writelines(file='sample.txt'):
    s = ['21\tfudan\t复旦大学\t中国上海\t200433\n',
         '22\tsjtu\t交通大学\t中国上海\t200240\n',
         '23\tlast line']
    with open(file,'a+') as f:
        f.writelines(s)
        f.seek(0)
        print(f.read())
```

file_write.py

文本文件的读写：以行为单位读

- readline(size=-1)：从文件读数据直到换行或文件结束，返回读到的字符串，换行也包括在内
  - 刚开始读遇到EOF时返回空字符串 ==> 可检查**返回值为空字符串**判断EOF
  - 如果size >0，表示最多读size个字符或者读到换行符或者读到EOF为止
- readlines()：从文件读数据直到文件结束，返回一个**字符串列表**，其元素为读到的每行字符串(包括换行符)

```
def read_file_readline(file='sample.txt'):
    with open(file, 'r') as f:
        while True:
            line = f.readline()
            if not line:
                break
            print(line, end='')
```

file_read.py

```
def read_file_readlines(file='sample.txt'):
    with open(file, 'r') as f:
        line = f.readlines()
    print(lines)
    print()
    for line in lines:
        print(line, end='')
```

文本文件的读写：以行为单位读

- 文件对象支持iterator协议，意味着调用next(file)获得下一行，即相当于file.readline()

```
def read_file_iterator(file='sample.txt'):
    with open(file, 'r') as f:
        for line in f:
            print(line, end='')
```

file_read.py

```
with open('somefile.txt') as file:
    lines = list(file)

# 相当于调用
with open('somefile.txt') as file:
    lines = file.readlines()
```

```
with open(filename) as f:
    while True:
        line = f.readline()
        if not line:
            break
        process(line)
```

```
with open(filename) as f:
    lines = f.readlines()

for line in lines:
    process(line)
```

```
with open(filename) as f:
    text = f.read()

lines = text.splitlines()
for line in lines:
    process(line)
```

splitlines: 换行符被移走

```
with open(filename) as f:
    for line in f:
        process(line)
```

文本文件的读写：编码方式

- open不传递编码方式时缺省为系统采用的编码方式（比如GBK）
- 解释器读取python源程序采用的编码方式缺省为UTF-8编码，除非
  - 前面两行中通过coding指出了相应的编码方式

```
#!/usr/bin/env python3
#  -*- coding: utf-8 -*-
```

```
def read_file_read(file='sample.txt'):
    with open(file, 'r') as f:
        text = f.read()
    print('-'*40)
    print(text)
```

UTF-8编码的py文件用GBK方式打开后，如果其中包含中文，则读到中文时会报错

```
def read_python_file(file=None, encoding=None):
    if not file:
        file = __file__
        encoding = encoding or 'utf-8'

    with open(file, encoding=encoding) as f:
        text = f.read()

    print(text)
```

file_encoding.py

```
if __name__ == '__main__':
    try:
        read_file_read(__file__)
    except Exception as e:
        print('Exception raised...', e)

    read_python_file()
```

Exception <class '**UnicodeDecodeError**'> raised... 'gbk' codec can't decode byte

标准输入输出

文件对象	描述
sys.stdout	标准输出，缺省情况下输出到屏幕上
sys.stdin	标准输入，缺省情况下从键盘输入
sys.stderr	标准错误输出，缺省情况下输出到屏幕上

print(value1, value2, …, sep=‘ ’, end=‘\n’, **file=sys.stdout**)

```
>>> import sys
>>> sys.stdout.write('hello world!\n')
hello world!
13
>>> fp = open('tmp.txt', 'a')
>>> print('hello world', file=fp)
>>> fp.close()
```

SringIO和BytesIO

拓展的内容

- io模块的StringIO和BytesIO允许通过文件的操作方式来实现对于内存缓冲区的读写
- StringIO相当于以文本方式读写： io.StringIO(initial_value='')
- BytesIO相当于以二进制方式读写： io.ByteIO(initial_bytes=b'')
- StringIO和ByteIO对象的getvalue()：获得目前内存缓冲区的值

```python
import io

def stringio_demo():
    file = io.StringIO("line 1\nline 2\n")
    while True:
        line = file.read()
        if not line:
            break
        print(line, end='')
    file.write("line 3\nline 4\n")
    print(file.getvalue())
    file.seek(0)
    for line in file:
        print(line, end='')
```

stringio.py

```python
def bytesio_demo():
    file = io.BytesIO()
    file.write(b"line 1\nline 2\n")
    print(file.getvalue())
```

文件和异常处理

```
def parse_int(file='sample.txt'):
    try:
        f = open(file)
        s = f.readline()    #如果这一行执行时出错怎么办
        f.close()
        i = int(s.strip())
        print(i)
    except (OSError, ValueError) as inst:
        print(type(inst),inst)
```

功能：打开文件，读一行内容，将其转换为整数，并且输出

parse_int.py

```
if __name__ == '__main__':
    setup()       # 创建测试环境，生成相应的文件
    parse_int('sample.txt')   #该文件的第一行为整数，正常运行输出结果
    parse_int('sample1.txt')  # 该文件不存在,匹配OSError
    parse_int('sample2.txt')   #该文件第一行为非整数，触发异常ValueError
    parse_int('sample3.txt')  # 该文件以utf-8编码，readline()出错，
                                # 匹配ValueError，但文件并没有关闭
```

不管是正常执行，还是异常出现时都希望能够关闭打开的文件

文件和异常处理

```
def parse_int2(file='sample.txt'):
    try:
        f = open(file)
        s = f.readline()
        # f.close()
        i = int(s.strip())
        print(i)
    except (OSError, ValueError) as inst:
        print(type(inst),inst)
    finally:
        f.close()
```

如果open时异常？f未定义，从而抛出NameError

建议采用with block打开文件，不用管文件是否关闭的问题

```
try:
    with open(file) as f:
        s = f.readline()
        i = int(s.strip())
        print(i)
except (OSError, ValueError) as inst:
    print(type(inst),inst)
```

```
def parse_int3(file='sample.txt'):
    f = None
    try:
        f = open(file)
        s = f.readline()
        # f.close()
        i = int(s.strip())
        print(i)
    except (OSError, ValueError) as inst:
        print(type(inst),inst)
    finally:
        if  f:
            f.close()
```

```
# 不要f=None
if 'f' in locals():
    f.close()
```

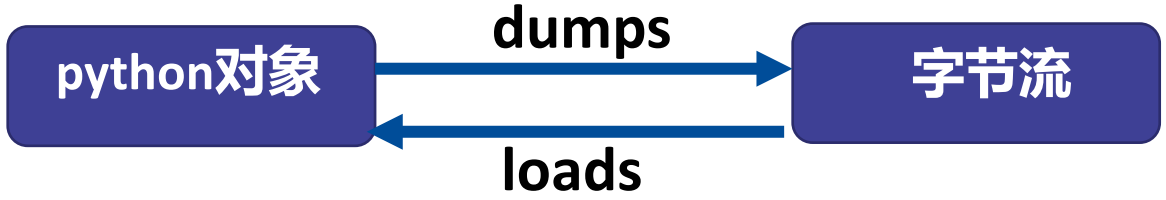
主要内容

- 文件的读写
  - 绝对路径和相对路径
  - 二进制文件
  - with语句
  - 文本文件
- **序列化和反序列化**：
  - pickle、json和csv模块
  - struct模块

序列化和反序列化

- 序列化和反序列化：将Python对象序列化成字节流，从字节流中反序列化成python对象
- pickle模块提供了序列化和反序列化的功能

```
import pickle
```



- 可以序列化的对象包括
  - None，布尔、整数、浮点数和复数
  - 字符串，bytes和bytearray
  - tuple/list/set/dictionary，但要求其元素也是可以序列化的对象
  - 模块顶层通过def定义的函数和类
  - 自定义类的实例对象，但有一些要求（略过）
- 注意**函数和类序列化时仅保存模块名+函数名或类名**，其代码和属性等并没有序列化

```
>>> import pickle
>>> s = pickle.dumps(4.15)
>>> s, pickle.loads(s)
(b'\x80\x03G@\x10\x99\x99\x99\x99\x99\x9a.', 4.15)
>>> import math
>>> pickle.dumps(math.sin)
b'\x80\x03cmath\nsin\nq\x00.'
```

方法	描述
dumps(obj)	将对象obj序列化为一个bytes对象并返回
dump(obj, file)	将对象obj序列化后写入到文件对象file中
loads(bytes_obj)	从bytes_obj中反序列化返回1个python对象
load(file)	从file中反序列化1个python对象后返回,如果文件结尾，抛出异常EOFError

persistence.py

```
import pickle
def pickle_dump(file='pickle.dat'):
    with open(file, 'wb') as f:
        n1 = 1024
        f1 = 3.14
        c1 = 1 + 2j
        s1 = 'python程序设计'
        l1 = [1,2,3]
        l2 = [l1, l1, 4, 5]
        t1 = 4, 5, 6
        st1 = {1, 2, 3}
        d1 = {'name':'idle', 'phone':['10010', '10011']}
        for obj in n1, f1, c1, s1, l1, l2, t1, st1, d1:
            pickle.dump(obj, f)
```

序列化到文件或者从文件反序列化时，文件应该以**二进制方式**打开
在反序列化时如果到文件结尾，则抛出EOFError异常

```
def pickle_load(file='pickle.dat'):
    with open(file, 'rb') as f:
        while True:
            try:
                obj = pickle.load(f)
                print(obj)
            except EOFError:
                break
```

```
if __name__ == '__main__':
    pickle_dump()
    pickle_load()
```

json模块

拓展的内容

- **json模块**提供类似于pickle模块的接口，只是在python对象与json格式的字符串之间进行转换
- **JSON(JavaScript Object Notation)** 是一种轻量级的数据交换格式。方便阅读和编写，也方便机器解析和生成。它基于JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集， 采用完全独立于语言的文本格式，要求采用UTF编码方式，建议UTF-8
- JSON支持的类型：
  - 字符串采用双引号定义，支持转义
  - 整数和浮点数
  - 布尔类型：true/false
  - null
  - 数组(array)：相当于python中的列表
  - 对象(object)：相当于python中的字典，但是其key必须为字符串

```
{
    "editor.minimap.enabled": true,
    "editor.wordWrap": 124,
    "python.pythonPath": "D:\\python3",
    "args": [
        "--quiet",
        "--norepeat",
        "--port",
        "1593"
    ],
    "vim.handleKeys": {
        "<C-n>": false
    }
}
```

json模块

方法	描述
dumps(obj, **kwargs)	将对象obj序列化，返回序列化后的字符串
dump(obj, file, **kwargs)	将对象obj序列化成字符串后写入(write)到文件对象file中
loads(s, **kwargs)	从字符串s中反序列化json对象，并返回相应的python对象
load(file, **kwargs)	从文件对象file(支持read)中反序列化json对象，返回相应的pythond对象。如果文件结尾，抛出异常EOFError

```
import json
def json_demo(file='demo.json'):
    text = r"""{
    "editor.minimap.enabled": true,
    "editor.wordWrap": 124,
    "python.pythonPath": "D:\\python3",
    "args": ["--quiet", "--norepeat", "--port", "1593"],
    "vim.handleKeys": {
        "<C-n>": false
    }
    }"""
    config = json.loads(text)
    print(config)

    with open(file, 'w') as f:
        json.dump(config, f, indent=4)
```

dumps和dump还包括keyword-only传递的参数
ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, **indent=None**, separators=None, default=None, sort_keys=False

- ensure_ascii: 非ASCII字符集中的字符以转义方式描述
- sort_keys: 在输出字典对象时，是否基于key排序
- indent：美化输出时缩进多少个空格
- default: 对象不能序列化时调用该函数定制如何序列化

```
{'editor.minimap.enabled': True,
'editor.wordWrap': 124, 'python.pythonPath':
'D:\\python3', 'args': ['--quiet', '--norepeat', '--
port', '1593'], 'vim.handleKeys': {'<C-n>': False}}
```

CSV(Comma-Separated Values)

拓展的内容

excel(一般为xlsx或xls格式)可将电子表格(spreadsheet)以csv格式保存。RFC 4180 定义了CSV的标准格式

- 电子表格包括多个记录(行)，每个记录包括了多个字段。一般第一条记录给出了各个字段的名字
- 记录之间通过\r\n分割；字段之间通过**逗号**分割
- 字段中如果包含了记录分隔符或字段分隔符时，需要用**双引号**括起来，如果双引号出现在字段中，则用**两个连续的双引号**代替
- 内置模块csv提供了对于csv格式的文档的读写支持

```
csvfile = open(file, newline='')
```

dialect给出了csv文件的若干配置参数的缺省值，可取值'excel'/'excel-tab'/'unix'。对于excel标准格式而言，delimiter=',', lineterminator='\r\n', quotechar='"', doublequote=True, quoting=QUOTE_MINIMAL, skipinitialspace=False (是否忽略字段前面的空格)

```
csv.reader(csvfile, dialect='excel', **fmtparams)
```

- csvfile为可迭代对象，其每个元素为字符串(一般对应一条记录)。如果csvfile为文件对象，建议newline=''
- 返回reader对象，它是1个迭代器对象，每个元素对应一条记录，每个记录以字符串列表方式呈现

csv_demo.py

```
text='''name,memo
dlmao,just a test
dlmao,just a test,others
"Dilin, Mao","""it's a test"""
'''
reader = csv.reader(text.splitlines())
for row in reader:
    print(row)
```

```
['name', 'memo']
['dlmao', 'just a test']
['dlmao', 'just a test', 'others']
['Dilin, Mao', '"it\'s a test"']
```

CSV(Comma-Separated Values)

拓展的内容

```
csv.writer(csvfile, dialect='excel', **fmtparams)
```

- csvfile为支持write的对象，建议打开文件时传递参数 newline=''
- 返回writer对象，可通过**writerow**方法写入一条记录，其参数为可迭代对象，可迭代对象中的元素对应该记录中的字段，将其转换为字符串后写入到csv文件

```
csv.DictReader(csvfile,...)        csv.DictWriter(csvfile, fieldnames=...,)
```

- 每一列通过字段名描述，传递的记录不是以列表等形式，而是字典形式，其key为字段名

```
reader = csv.DictReader(file)
for row in reader:
    print(row)

writer = csv.DictWriter(file, fiel
dnames=('f1', 'f2'))
writer.writeheader()
writer.writerow({'f1': 1, 'f2': 2})
```

```
import csv
def csv_writer(file='eggs.csv'):
    with open(file, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Spam'] * 5 + ['Baked Beans'])
        writer.writerow(['Spam','Lovely Spam','Wonderful Spam'])

def csv_reader(file='eggs.csv'):
    with open(file, newline='') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            print(', '.join(row))
```

csv_demo.py

```
Spam,Spam,Spam,Spam,Spam,Baked Beans
Spam,Lovely Spam,Wonderful Spam
```

struct模块(9.3.2)

拓展的内容

- 在网络编程、与采用C语言编写的应用程序交互时进行序列化和反序列化
- values--> bytes：pack(fmt, v1, v2, ...)：将v1、v2等值按照fmt给出的格式进行转换，返回一个bytes对象
- bytes --> values: unpack(fmt, buffer)：按照fmt给出的格式将bytes对象buffer中的内容进行转换，返回包含了多个值的元组
- 字节顺序、大小和对齐方式：基于fmt的第1个字符的取值：
  - 网络字节或大端：高字节在先的顺序
  - 小端：高字节在后的顺序
  - 对齐：比如整数对应的地址为4的倍数的边界

字符	字节顺序	大小和对齐
@或非@=<>!字符	native	native
=	native	标准大小，无对齐
!或>	大端(big-endian)	标准大小，无对齐
<	小端(little-endian)	标准大小，无对齐

数值类型：小写/大写，对应有符号和无符号类型)

格式	C类型	Python	标准字节数
x	填充字节	无对应	
c	char	长度1的bytes	1
?	_Bool (C99)	bool	1
bB	signed/unsigned char	integer	1
hH	[unsigned] short	integer	2
iI	[unsigned] int	integer	4
lL	[unsigned] long	integer	4
qQ	[unsigned] long long	integer	8
nN	ssize_t, size_t	integer	native
f	float	float	4
d	double	float	8
p	char[]	bytes	
s	char[]	bytes	
P	void *	integer	

struct模块

拓展的内容

- calcsize(fmt)：根据fmt，计算如果要pack需要的字节数
- pack_into(fmt, buffer, offset, v1, v2, ...): 类似于pack，只是不是返回bytes对象，而是写入到bytearray对象buffer中，从offset开始写入
- unpack_from(fmt, buffer, offset=0)：类似于unpack，只是从buffer的偏移量offset处开始unpack
- 字符串：
  - 首先转换为字节串
  - 计算字节串的长度。如果要与C语言的NULL字符结尾的字符串char[]对应，应该为字节串的长度+1

struct_demo.py

```
i, f, b, s = 1234567, 3.14, True, 'python程序设计'
s2 = s.encode()
s2_len = len(s2)
packed_buffer = struct.pack(f'if?h{s2_len}s', i, f, b, s2_len, s2)
print(packed_buffer)
```

```
offset = struct.calcsize('if?h')
values = struct.unpack('if?h', packed_buffer[:offset])
print(*values)
s = struct.unpack_from(f'{values[-1]}s', packed_buffer, offset)
print(s[0].decode())
```

struct模块

字符	字节顺序	大小和对齐
@或非@=<>!字符	native	native
=	native	标准大小，无对齐
!或>	大端(big-endian)	标准大小，无对齐
<	小端(little-endian)	标准大小，无对齐

拓展的内容

```
>>> import sys
>>> sys.byteorder
'little'
>>> struct.pack('3i', 1, 2, 3)
b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> struct.pack('@3i', 1, 2, 3)
b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'
>>> struct.pack('@bhi', 1, 2, 3)
b'\x01\x00\x02\x00\x03\x00\x00\x00'
>>> struct.pack('<bhi', 1, 2, 3)  # 采用小端字节顺序
b'\x01\x02\x00\x03\x00\x00\x00'
>>> struct.pack('>bhi', 1, 2, 3)  # 采用大端或网络字节顺序
b'\x01\x00\x02\x00\x00\x00\x03'
>>> struct.pack('!bhi', 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'

>>> struct.calcsize('bhi')
8
# why? 在实验主机的native实现时，short要以地址为2的
# 倍数的边界开始。 int要以地址为4的倍数的边界开始
>>> struct.pack('<bxhi', 1, 2, 3)
b'\x01\x00\x02\x00\x03\x00\x00\x00'
```