

# 文件IO操作

函数	说明
<code>open</code>	打开
<code>read</code>	读取
<code>write</code>	写入
<code>close</code>	关闭
<code>readline</code>	行读取
<code>readlines</code>	多行读取

## open方法

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

打开一个文件，返回一个文件对象(流对象)和文件描述符。打开文件失败，则返回异常

基本使用：创建一个文件test，然后打开它，用完**关闭**

```
1 f = open("test") # file对象
2 # windows <_io.TextIOWrapper name='test' mode='r' encoding='cp936'>
3 # linux <_io.TextIOWrapper name='test' mode='r' encoding='UTF-8'>
4 print(f.read()) # 读取文件
5 f.close() # 关闭文件
```

文件操作中，最常用的操作就是读和写。

文件访问的模式有两种：文本模式和二进制模式。不同模式下，操作函数不尽相同，表现的结果也不一样。

注：

windows中使用codepage代码页，可以认为每一个代码页就是一张编码表。cp936等同于GBK。

## open参数

### file

打开或者要创建的文件名。如果不指定路径，默认是当前路径

### mode模式

模式描述字符	意义
r	缺省模式，只读打开
w	只写打开
x	创建并写入一个新文件
a	只写打开，追加内容
b	二进制模式
t	缺省模式，文本模式
+	读或写打开后，使用+来增加缺失的写或读的能力

模式对于IO操作来说，其实只有读和写两种：

- 只读 r
- 只写 w、x、a
- 增加缺失能力 +

r 模式

- 只读打开文件，如果使用write方法，会抛异常
- 如果文件不存在，抛出FileNotFoundError异常

w 模式

- 表示只写方式打开，如果读取则抛出异常
- 如果文件不存在，则直接创建文件
- 如果文件存在，则清空文件内容

x 模式

- 文件不存在，创建文件，并只写方式打开
- 文件存在，抛出FileExistsError异常

a 模式

- 文件存在，只写打开，追加内容
- 文件不存在，则创建后，只写打开，追加内容

wxa模式都可以产生新文件

- w不管文件存在与否，都会生成全新内容的文件
- a不管文件是否存在，都能在打开的文件尾部追加
- x必须要求文件事先不存在，自己要造一个新文件

文本模式t

- 字符流，将文件的字节按照某种字符编码理解，按照字符操作。open的默认mode就是rt。

二进制模式b

- 字节流，将文件就按照字节理解，与字符编码无关。二进制模式操作时，字节操作使用bytes类型

+ 模式

- 为r、w、a、x提供缺失的读或写功能，但是，获取文件对象依旧按照r、w、a、x自己的特征。

- +模式不能单独使用，可以认为它是为前面的模式字符做增强功能的。

## encoding: 编码，仅文本模式使用

None 表示使用缺省编码，依赖操作系统。windows、linux下测试如下代码

```
1 f = open('test1', 'w')
2 f.write('啊')
3 f.close()
```

windows下缺省GBK (0xB0A1) , Linux下缺省UTF-8 (0xE5 95 8A)

## 文件指针

mode=r, 指针起始在0

mode=a, 指针起始在EOF

```
1 f = open('o:/test.txt', 'wb+')
2 print(f)
3 f.write(b'abc')
4 print(f.tell())
5 f.close()
6
7 f = open('o:/test.txt', 'rt+') # windows下打开
8 f.write('啊') # 从什么地方开始写几个字节?
9 print(hex(ord('啊')), '啊'.encode(), '啊'.encode('gbk'))
10 print(f.tell())
11 f.close()
```

tell、seek函数单位都是字节

## read

read(size=-1)

- size表示读取的多少个字符或字节；负数或者None表示读取到EOF

```
1 filename = 'o:/test.txt'
2 f = open(filename, 'w+')
3 f.write('马哥教育')
4 f.close()
5
6 f = open(filename)
7 print(1, f.read(1)) # 按字符
8 print(2, f.read(2))
9 print(3, f.read())
10 f.close()
11
12 f = open(filename, 'rb')
13 print(4, f.read(1)) # 按字节
14 print(5, f.read(2))
15 print(6, '马哥教育'.encode('gbk'))
16 print(7, f.read())
```

建议，使用文件对象时，一定要指定编码，而不是使用默认编码

## write

- write(s)，文本模式时，从当前指针处把字符串s写入到文件中并返回写入字符的个数；二进制时将bytes写入文件并返回写入字节数
- writelines(lines)，将字符串列表写入文件

```
1 filename = 'o:/test.txt'
2 f = open(filename, 'w+')
3 lines = ['abc', '123\n', 'magedu'] # 需提供换行符
4 # for line in lines:
5 #     f.write(line)
6 f.writelines(lines)
7 f.seek(0) # 回到开始
8 print(f.read())
9 f.close()
```

## close

flush并关闭文件对象。文件已经关闭，再次关闭没有任何效果。可以查看文件对象的closed属性，判断是否关闭

## 上下文管理

文件对象这种打开资源并一定要关闭的对象，为了保证其打开后一定关闭，为其提供了上下文支持。

```
1 filename = 'o:/test.txt'
2 with open(filename) as f:
3     print(1, f.closed)
4     print(f.write('abcd')) # r模式写入失败，抛异常
5
6 print(2, f.closed) # with中不管是否抛异常，with结束时都会保证关闭文件对象
```

```
1 with 文件对象 as 标识符: # 等同于 标识符 = 文件对象
2     pass # 标识符可以在内部使用
```

### 上下文管理

1. 使用with关键字，上下文管理针对的是with后的对象
2. 使用with ... as 关键字
3. 上下文管理的语句块并不会开启新的作用域

1. 进入with时，with后的文件对象是被管理对象
2. as子句后的标识符，指向with后的文件对象
3. with语句块执行完的时候，会自动关闭文件对象

```
1 filename = 'o:/test.txt'
2 f = open(filename)
3 with f:
4     print(1, f.closed)
5     print(f.write('abcd')) # r模式写入失败
6
7 print(2, f.closed) # with中不管是否抛异常，with结束时都会关闭文件对象
```

```
1 filename = 'o:/test.txt'
2 f = open(filename)
3 with f as f2:
4     print(f is f2) # True
```

## 文件的遍历

类似于日志文件，文件需要遍历，最常用的方式就是逐行遍历。

```
1 filename = 'o:/test.txt'
2 with open(filename, 'w') as f:
3     f.write('\n'.join(map(str, range(101, 120))))
4
5 with open(filename) as f:
6     for line in f: # 文件对象时可迭代对象，逐行遍历
7         print(line.encode()) # 带换行符
```

## 路径操作

### os.path模块

```
1 # os模块常用函数
2 from os import path
3
4 p = path.join('/etc', 'sysconfig', 'network') # 拼接
5 print(type(p), p)
6 print(path.exists(p)) # 存在
7
8 print(path.split(p)) # 分割
9 print(path.splitdrive('o:/temp/test')) # windows方法
10 print(path.dirname(p), path.basename(p)) # 路径和基名
11
12 print(path.abspath('.'), path.abspath('.')) # 绝对路径
13
14 # 打印父目录
15 p1 = path.abspath(__file__)
16 print(p1)
17 while p1 != path.dirname(p1):
18     p1 = path.dirname(p1)
19     print(p1)
```

os.path模块操作的都是字符串。

## Path类

从3.4开始Python提供了pathlib模块，使用Path类操作目录更加方便。

### 初始化

```
1 p = Path() # 当前目录, Path(), Path('.'), Path('')
2 p = Path('a', 'b', 'c/d') # 当前目录下的a/b/c/d
3 p = Path('/etc', Path('sysconfig'), 'network/ifcfg') # 根下的etc目录
```

### 拼接

#### 操作符/

- Path对象 / Path对象
- Path对象 / 字符串
- 字符串 / Path对象

#### joinpath

- joinpath(\*other) 在当前Path路径上连接多个字符串返回新路径对象

```
1 from pathlib import Path
2
3 p = Path()
4 p = p / 'a'
5 p1 = 'b' / p
6 p2 = Path('c')
7 p3 = p2 / p1
8 print(p1, p2, p3)
9 print(p3.parts)
10 print(p3.joinpath('d', 'e/f', Path('g/h')))
```

### 分解

parts属性，会返回目录各部分的元组

```
1 p = Path('/a/b/c/d')
2 print(p.parts) # 最左边的/是根目录
```

### 父目录

```
1 p = Path('/magedu/mysql/install/mysql.tar.gz')
2 print(p.parent)
3 for x in p.parents: # 可迭代对象
4     print(x)
```

## 目录组成部分

name、stem、suffix、suffixes、with\_suffix(suffix)、with\_name(name)

- name 目录的最后一个部分
- suffix 目录中最后一个部分的扩展名
- stem 目录最后一个部分，没有后缀
- name = stem + suffix

suffixes 返回多个扩展名列表

- with\_suffix(suffix) 有扩展名则替换，无则补充扩展名
- with\_name(name) 替换目录最后一个部分并返回一个新的路径

```
1 from pathlib import Path
2
3 p = Path('/magedu/mysql/install/mysql.tar.gz')
4 print(p.parent)
5 print(p.name)
6 print(p.stem)
7 print(p.suffix)
8 print(p.suffixes)
9 print(p.with_name('redis'))
10 print(p.with_name('redis').with_suffix('.zip'))
```

## 全局方法

- cwd() 返回当前工作目录
- home() 返回当前家目录

```
1 p = Path('/magedu/mysql/install/mysql.tar.gz')
2 print(p.cwd(), Path.cwd())
3 print(p.home(), Path.home())
```

## 判断方法

- exists() 目录或文件是否存在
- is\_dir() 是否是目录，目录存在返回True
- is\_file() 是否是普通文件，文件存在返回True
- is\_symlink() 是否是软链接
- is\_socket() 是否是socket文件
- is\_block\_device() 是否是块设备
- is\_char\_device() 是否是字符设备
- is\_absolute() 是否是绝对路径

注意：文件只有存在，才能知道它是什么类型文件

## 绝对路径\*\*

- resolve() 非Windows，返回一个新的路径，这个新路径就是当前Path对象的绝对路径，如果是软链接则直接被解析。Windows下没什么效果。
- absolute() 获取绝对路径。

## 其它操作

- `rmdir()` 删除空目录。没有提供判断目录为空的方法
- `touch(mode=0o666, exist_ok=True)` 创建一个文件
- `as_uri()` 将路径返回成URI, 例如'`file:///etc/passwd`'
- `mkdir(mode=0o777, parents=False, exist_ok=False)`  
    `parents`, 是否创建父目录, `True`等同于`mkdir -p`。`False`时, 父目录不存在, 则抛出 `FileNotFoundError`  
    `exist_ok`参数, 在3.5版本加入。`False`时, 路径存在, 抛出`FileExistsError`; `True`时, `FileExistsError`被忽略
- `iterdir()` 迭代当前目录, 不递归

```
1  from pathlib import Path
2
3  p = Path('o:/a/b/c/d')
4  p.mkdir(parents=True, exist_ok=True)
5  (p / 'test').touch()
6
7  for x in p.parents[len(p.parents) - 1].iterdir(): # 不支持负索引
8      if x.is_dir():
9          print('dir =', x)
10     elif x.is_file():
11         print('file =', x)
12     else:
13         print('other =', x)
```

- `stat` 相当于`stat`命令
- `lstat` 使用方法同`stat()`, 但如果是符号链接, 则显示符号链接本身的文件信息

## 通配符

- `glob(pattern)` 通配给定的模式, 返回生成器对象
- `rglob(pattern)` 通配给定的模式, 递归目录, 返回生成器对象
- `?` 代表一个字符
- `*` 表示任意个字符
- `[abc]`或`[a-z]` 表示一个字符

```
1  list(p.glob('test*')) # 返回当前目录对象下的test开头的文件
2  list(p.glob('**/*.py')) # 递归所有目录, 等同rglob
3  list(p.glob('**/*'))
4
5  g = p.rglob('*.py') # 生成器, 递归
6  next(g)
7  list(p.rglob('*.???')) # 匹配扩展名为3个字符的文件
8  list(p1.rglob('[a-z]*.???')) # 匹配字母开头的且扩展名是3个字符的文件
```

## shutil模块

文件拷贝: 使用打开2个文件对象, 源文件读取内容, 写入目标文件中来完成拷贝过程。但是这样丢失 `stat`数据信息 (权限等), 因为根本没有复制这些信息过去。

目录复制又怎么办呢?



Python提供了一个方便的库shutil（高级文件操作）。

## copy 复制

```
copyfileobj(fsrc, fdst[, length])
```

文件对象的复制，fsrc和fdst是open打开的文件对象，复制内容。fdst要求可写。  
length 指定了表示buffer的大小；

```
copyfile(src, dst, *, follow_symlinks=True)
```

复制文件内容，不含元数据。src、dst为文件的路径字符串  
本质上调用的就是copyfileobj，所以不带元数据二进制内容复制。

```
copymode(src, dst, *, follow_symlinks=True)
```

仅仅复制权限。

```
copystat(src, dst, *, follow_symlinks=True)
```

复制元数据，stat包含权限

```
copy(src, dst, *, follow_symlinks=True)
```

复制文件内容、权限和部分元数据，不包括创建时间和修改时间。  
本质上调用的是

```
copyfile(src, dst, follow_symlinks=follow_symlinks)
```

```
copymode(src, dst, follow_symlinks=follow_symlinks)
```

copy2 比copy多了复制全部元数据，但需要平台支持。

本质上调用的是

```
copyfile(src, dst, follow_symlinks=follow_symlinks)
```

```
copystat(src, dst, follow_symlinks=follow_symlinks)
```

```
copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2,  
ignore_dangling_symlinks=False)
```

**递归复制目录。**默认使用copy2，也就是带更多的元数据复制。

src、dst必须是目录，src必须存在，dst必须**不存在**

ignore = func，提供一个callable(src, names) -> ignored\_names。提供一个函数，它会被调用。src是源目录，names是os.listdir(src)的结果，就是列出src中的文件名，返回值是要被过滤的文件名的set类型数据。

```
1 # o:/temp下有a、b目录
2 def ignore(src, names):
3     ig = filter(lambda x: x.startswith('a'), names) # 忽略a开头的
4     return set(ig)
5
6 shutil.copytree('o:/temp', 'o:/tt/o', ignore=ignore)
```

## rm 删除

```
shutil.rmtree(path, ignore_errors=False, onerror=None)
```

递归删除。如同rm -rf一样危险，慎用。

它不是原子操作，有可能删除错误，就会中断，已经删除的就删除了。

ignore\_errors为true，忽略错误。当为False或者omitted时onerror生效。

onerror为callable，接受函数function、path和execinfo。

```
1 | shutil.rmtree('o:/tmp') # 类似 rm -rf
```

## move 移动

```
move(src, dst, copy_function=copy2)
```

递归移动文件、目录到目标，返回目标。

本身使用的是 os.rename 方法。

如果不支持 rename，如果是目录则 copytree 再删除源目录。

默认使用 copy2 方法。

```
1 | shutil.move('o:/a', 'o:/aaa')
2 | os.rename('o:/t.txt', 'o:/temp/t')
3 | os.rename('test3', '/tmp/py/test300')
```

shutil 还有打包功能。生成 tar 并压缩。支持 zip、gz、bz、xz。