文件操作

从文件的编码方式来看，文件可以分为文本文件和二进制文件：

文本文件：txt、html、json等； 二进制文件：图片、音频、视频等

从计算机物理内存来看，所有文件都是二进制形式存放的

因此文本文件也可以用二进制格式读取，而二进制文件是不能用文本格式读取的

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

- file：文件路径（相对路径或绝对路径）
- mode：文件打开的模式，默认为 'r' 模式
- encoding：编码方式（只用在文本模式下），默认依赖平台，通常设置为 'UTF-8'
- 打开 file 对应的文件，返回一个文件对象；如果该文件不能被打开，则引发 OSError

```
from typing import Iterator


file = open(r'./t01.txt', mode='a')
file.write('hello world')
file.write('\nhello China')
file.write('\nhello Baby')
file.close()
```

```
file = open(r'./t01.txt')
print(isinstance(file, Iterator))
# print(list(file))
for i in file:
    print(i)
```

mode 常用模式：

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新的文件再写入。
x	新建一个文件只用于写入，如果该文件已存在则会报错。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
+	如果要以读写模式打开，加上 + 即可，比如：r+、w+、x+、a+
b	默认为文本模式，如果要以二进制模式打开，加上 b 即可，比如：rb、rb+、wb、wb+

file 常用对象方法：

file.read(size=-1)

- 从 file 中读取至多 size 个字符并返回
- 如果 size 为负值或 None，则读取至 EOF（End Of File）

```
with open(r"./t01.txt") as file:
    print(file.read(5))
    print(file.read(2))
    print(file.read())
```

file.write(s)

- 将字符串 s 写入到流并返回写入的字符数

```
with open(r"./t01.txt", mode='a') as file:
    num = file.write('\nhello baby')
    print(num)
```

file.flush()

- 刷新缓冲区，即将缓冲区中的数据立刻写入文件，同时清空缓冲区，不需要被动的等待缓冲区写入。一般情况下，文件关闭后会自动刷新缓冲区，但有时你需要在关闭前刷新它，这时就可以使用 flush() 方法

```
import time


file = open(r"./t01.txt", mode='a')
file.write('\n123456789')
time.sleep(5)  # 文件需要等到关闭文件时才会把数据从缓冲区写入文件
file.close()  # 关闭文件，自动刷新缓冲区，数据才写入文件


file = open(r"./t01.txt", mode='a')
file.write('\n123456789')
file.flush()  # 刷新缓冲区，数据立刻写入文件
time.sleep(5)
file.close()
```

file.close()

- 刷新缓冲区并关闭该文件。如果文件已经关闭，则此方法无效
- 文件关闭后，对文件的任何操作（如：读取或写入）都会引发 ValueError

```
file = open(r'./t01.txt')
print(file.read())
file.close()
file.close()  # 多次调用该方法，只有第一个调用才会生效
file.read()  # 引发ValueError
```

file.writable()

- 判断 file 是否可写，返回 True 或 False

```
with open(r"./t02.txt", mode='x') as file:
    if file.writable():
        file.write("hello world")
    if file.readable():
        print(file.read())
```

file.readable()

- 判断 file 是否可读，返回 True 或 False

```
with open(r"./t03.txt", mode='x+') as file:
    if file.writable():
        file.write("hello world")
    if file.readable():
        print(file.read())
```

file.seek(offset, whence=SEEK_SET)

- 移动文件指针到指定位置，返回新的绝对位置
- offset：相对于 whence 的位置
- whence 可以设置为：

  SEEK_SET 或 0 -- 开头（默认值）；offset 应为零或正值

  SEEK_CUR or 1 -- 当前位置；offset 可以为负值

  SEEK_END or 2 -- 末尾；offset 通常为负值
- 注意：在文本文件中，只允许相对于文件开头搜索，二进制是可以的

```
with open(r"./t04.txt", mode='x+') as file:
    if file.writable():
        file.write("hello world")
    if file.readable():
        print(file.seek(2))
        print(file.read())
```

file.tell()

- 返回文件指针当前位置

```
with open(r"./t04.txt", mode="a") as file:
    print(file.tell())
```

file.writelines(lines)

- lines：Iterable[str]
- 向文件写入一个字符串序列，不会自动添加分隔符

```
tup = ("a\n", 'bc\n', "def")
with open(r"./t01.txt", mode="w") as file:
    file.writelines(tup)
```

file.readline(size=-1)

- 从文件中读取并返回一行，如果指定了 size，将至多读取 size 个字符

```
with open("./t01.txt") as file:
    print(file.readline())
    print(file.readline(1))
    print(file.readline())
```

file.readlines(hint=-1)

- 从文件中读取并返回包含多行的列表
- hint：默认为 -1，代表读取所有行（也可以指定读取的字符数，如果要读取的字符数超过一行，则按照两行读取，超过两行则按照三行读取，依次类推）

```
with open("./t01.txt") as file:
    print(file.readlines())

with open("./t01.txt") as file:
    print(file.readlines(11))
```

with 语句

- 这种写法有一个潜在问题，如果在调用 write 的过程中出现了异常，则会导致 close 方法无法被正常调用，导致资源占用的浪费

```
file = open(r'./t01.txt', mode='w')
file.write('hello world')
file.close()
```

- 为了解决上面的问题，我们可以把程序改进

```
file = open(r'./t01.txt', mode='w')
try:
    file.write('hello world')
finally:
    file.close()
```

- 用 with 语句将会是一种更加简洁、优雅的方式

```
with open(r'./t01.txt', mode='w') as file:
    file.write('hello world')
```

路径操作

---

os.getcwd()

- 返回表示当前工作目录的字符串

```
import os

print(os.getcwd())
```

os.listdir(path)

- 返回 path 指定的文件夹包含的文件或文件夹的名字的列表

```
import os

li = os.listdir(os.getcwd())
print(li)
```

os.mkdir(path)

- 只创建 path 中的最后一级目录，所以要保证所需要的中间目录是存在的
- 如果最后一级目录已存在，则抛出 FileExistsError 异常

```
import os

os.mkdir(os.getcwd() + "/MyDir")
```

os.makedirs(name, exist_ok=False)

- 递归创建目录，并且还会自动创建到达最后一级目录所需要的中间目录
- 如果 exist_ok 为 False (默认值)，则如果目标目录已存在将引发 FileExistsError

```
import os

os.makedirs('./dir1/dir2/dir3')
```

os.remove(path)

- 移除 path 对应的文件；如果 path 是目录或者文件不存在，都会引发异常

```
import os

os.remove('./dir1/dir2/a.txt')
```

os.rmdir(path)

- 删除 path 指定的最后一级空目录，如果目录不存在或不为空，都会引发异常

os.removedirs(path)

- 递归删除空目录

```
import os

os.removedirs('./dir1/dir2/dir3')
```

os.rename(src, dst)

- 重命名目录或文件

```
import os

os.rename(os.getcwd() + "/dir1", os.getcwd() + "/dir4")
```

os.renames(old, new)

- 递归重命名目录或文件，会自动创建新路径所需的中间目录

```
import os

os.renames('./dir1/dir2/dir3/a.txt',
'./dir3/dir2/dir1/b.txt')
```

os.path.abspath(path)

- 返回路径 path 的绝对路径

```
import os

print(os.path.abspath("./dir3"))
print(os.path.abspath("./dir3/a.txt"))
```

os.path.basename(path)

- 返回路径 path 最后一级的名称，通常用来返回文件名

```
import os

print(os.path.basename('./dir3/dir2/dir1/a.txt'))
```

os.path.dirname(path)

- 返回路径 path 的目录名称

```
import os

print(os.path.dirname('./dir3/dir2/dir1/a.txt'))
```

os.path.split(path)

- 把路径分割成 dirname 和 basename，返回一个元组

```
import os

print(os.path.split('./dir3/dir2/dir1/a.txt'))
```

os.path.splitext(path)

- 把路径中的扩展名分割出来，返回一个元组

```
import os



print(os.path.splitext('./dir3/dir2/dir1/a.txt'))
```

os.path.exists(path)

- path 路径存在则返回 True，不存在或失效则返回 False

```
import os

path = "./dir3/dir2/dir1/a.txt"
if os.path.exists(path):
    with open(path) as file:
        print(file.read())
```

os.path.isabs(path)

- 判断 path 是否是绝对路径，返回 True 或 False

```
import os

print(os.path.isabs("./dir3"))
```

os.path.isfile(path)

- 判断路径是否为文件，返回 True 或 False

```
import os

print(os.path.isfile("./dir3/dir2/dir1/a.txt"))
```

os.path.isdir(path)

- 判断路径是否为目录，返回 True 或 False

```
import os

print(os.path.isdir("./dir3/dir2/dir1"))
```

os.path.join(path, *paths)

- 智能地拼接一个或多个路径部分

```
import os

print(os.path.join("./dir3/dir2", "dir1/a.txt"))
```

json 格式

字符串可以很轻松地写入文件并从文件中读取出来，而其他类型（比如：字典）复杂的数据会变得相当麻烦，因为 read() 方法返回字符串。

Python 允许你使用 json 格式，并提供了名为 json 的标准模块，它可以将 Python 数据结构转化为 json 格式的字符串（这个过程称之为序列化），也可以将 json 格式的字符串重建成 Python 数据结构（这个过程称之为反序列化）。json 是一个文本序列化格式，可直观阅读。

注意：json 中的键值对的键，永远是 str 类型。当一个对象被转化为 json 时，字典中所有的键都会被强制转换为字符串，导致当字典被转换为 json 然后转换回字典时可能和原来不相等。

```
import json


info = {'name': 'Tom', 'age': 18, 1: 'one'}
print(type(info), info)

with open("info.json", mode="w") as f:
    info_str = json.dumps(info)  # 序列化
    print(type(info_str), info_str)
    f.write(info_str)

with open("info.json") as f:
    content = f.read()
    print(type(content), content)
    res = json.loads(content)  # 反序列化
    print(type(res), res)
```

```
import json


info = {'name': 'Tom', 'age': 18, 1: 'one'}

with open("info.json", mode="w") as f:
    json.dump(info, f)

with open("info.json") as f:
    res = json.load(f)
    print(type(res), res)
```

```
import json


with open('./TestFile.json') as file:
    content = json.load(file)

obj_list = content["outputs"]["object"]
for item in obj_list:
    print(item["name"])
    print(item["bndbox"])
```

pickle 格式

---

pickle 是一个Python专用的二进制序列化格式，不可直观阅读。

```
import pickle


info = {'name': 'Tom', 'age': 18, 1: 'one'}
print(type(info), info)

with open("info.pickle", mode="wb") as f:
    info_byte = pickle.dumps(info)
    print(type(info_byte), info_byte)
    f.write(info_byte)

with open("info.pickle", mode="rb") as f:
    content = f.read()
    print(type(content), content)
    res = pickle.loads(content)
```

```
    print(type(res), res)
```

```
import pickle


info = {'name': 'Tom', 'age': 18, 1: 'one'}

with open("info.pickle", mode="wb") as f:
    pickle.dump(info, f)

with open("info.pickle", mode="rb") as f:
    res = pickle.load(f)
    print(type(res), res)
```

xml 格式

XML文件格式是纯文本格式。XML文档的第一句是声明语句，紧接着声明后面建立的第一个元素是根元素（有且只有一个），其他元素都是这个根元素的子元素，每个XML元素包括一个开始标签，一个结束标签，以及两个标签之间的内容。

```
import xml.etree.ElementTree as ET


tree = ET.parse(r"./TestFile.xml")  # 将XML文档解析为元素树
root = tree.getroot()  # 返回该树的根元素，即 <doc>
obj = root.find("outputs").find("object")  # 找到对应的子元素
```

```
for item in obj:
    name = item.find("name")
    print(name.text)
    bndbox = item.find("bndbox")
    for item in bndbox:
        print(item.tag)  # tag：标签（尖括号之间的内容）
        print(item.text)  # text：文本（标签之间的内容）
```