第十章 - 文件读写

张建章

阿里巴巴商学院

杭州师范大学

2024-09

`open()` 函数是进行文件操作的基础。该函数用于打开一个文件，并返回一个文件对象，之后可对该对象进行读取或写入操作，基本语法如下：

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

其中，`file` 参数指定要打开的文件路径，`encoding` 参数指定编码格式，通常取值 `'utf-8'`，`mode` 参数指定文件的打开模式。常用的模式包括：

- `'r'`：以只读模式打开文件（默认值）。
- `'w'`：以写入模式打开文件，若文件已存在，则会覆盖其内容。
- `'a'`：以追加模式打开文件，数据将写入文件末尾。
- `'+'`：同时打开文件进行读写操作，与其他模式组合使用，如 `'r+'` 表示以读写模式打开文件。

以下是一个使用 `open()` 函数读取文件内容的示例：

```
# 打开文件 example.txt 进行读取
file = open('example.txt', 'r', encoding='utf-8')
content = file.read()
print(content)
file.close()
```

在上述代码中，`encoding='utf-8'` 参数指定文件的编码方式，以正确读取包含非 ASCII 字符的文件。`file.close()` 方法用于关闭文件对象，释放系统资源并确保数据完整性。

假设 `example.txt` 文件的内容如下：

```
Python 是一种广泛使用的高级编程语言.
它具有简洁的语法和强大的功能.
```

运行上述代码将输出文件的全部内容。

`open()` 函数还可用于写入文件。以下是一个写入文件的示例：

```
# 打开文件 example.txt 进行写入
file = open('example.txt', 'w', encoding='utf-8')
file.write('这是一个新的内容。')
file.close()
```

执行此代码后，`example.txt` 文件的内容将被替换为 `' 这是一个新的内容。'`。

注意：使用 `'w'` 模式打开文件会覆盖原有内容，若需在文件末尾追加内容，应使用 `'a'` 模式。`file.close()` 方法确保写入的数据被刷新到磁盘，避免数据丢失。

在 Python 编程中，`with` 语句用于简化对资源的管理，特别是在文件操作中。使用 `with` 语句打开文件时，系统会在代码块执行完毕后自动关闭文件，无需显式调用 `close()` 方法，从而确保资源的正确释放并提高代码的可读性。

以下示例展示了如何使用 `with` 语句读取文件内容：

```
# 使用 with 语句打开文件 example.txt 进行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

在上述代码中，`with` 语句打开名为 `example.txt` 的文件，并将文件对象赋值给变量 `file`。在 `with` 代码块内，调用 `read()` 方法读取文件内容，并输出到控制台。当代码块执行完毕后，文件会被自动关闭。

以下示例展示了如何使用 `with` 语句将内容写入文件：

```
# 使用 with 语句打开文件
with open('example.txt', 'w', encoding='utf-8') as file:
    file.write('这是一个新的内容。')
    # 无需显式调用 file.close()
```

推荐使用 `with` 语句读写文件。

文件的读取方法主要包括`read()`、`readline()`、`readlines()`和直接迭代文件对象。这些方法适用于不同的场景，选择合适的方法有助于提高代码的效率和可读性。

1. `read()`方法：

`read()`方法用于一次性读取整个文件的内容，并将其作为一个字符串返回。需要注意的是，若文件较大，使用`read()`可能导致内存占用过高。

```
# 打开文件 example.txt 进行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

2. `readline()` 方法：

`readline()` 方法用于读取文件的一行内容，包括行末的换行符。每次调用 `readline()` 都会返回文件的下一行，适用于逐行读取文件的场景。

```
# 打开文件 example.txt 逐行读取
with open('example.txt', 'r', encoding='utf-8') as file:
    line = file.readline()
    while line:
        print(line, end='')  # end='' 防止重复添加换行
        line = file.readline()
```

运行上述代码将逐行输出文件内容。

3. `readlines()` 方法：

`readlines()` 方法用于读取文件的所有行，并将其作为一个列表返回，每个元素为文件中的一行。需要注意的是，若文件较大，使用 `readlines()` 可能导致内存占用过高。

```
# 打开文件 example.txt 读取所有行
with open('example.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()
    for line in lines:
        print(line, end='')
```

运行上述代码将输出文件的全部内容。

4. 直接迭代文件对象：

文件对象本身是可迭代的，直接对文件对象进行迭代可逐行读取文件内容。这种方法内存占用较低，适用于大文件的读取。

```
# 打开文件 example.txt 直接迭代
with open('example.txt', 'r', encoding='utf-8') as file:
    for line in file:
        print(line, end='')
```

运行上述代码将逐行输出文件内容。

注意

- 在读取文件时，建议使用 `with` 语句管理文件上下文。`with` 语句会在代码块执行完毕后自动关闭文件，确保资源的正确释放。

- 在读取包含非 ASCII 字符的文本文件时，需指定编码方式，如 `encoding='utf-8'`，以确保字符编码正确。

文件的读取模式决定了文件的打开方式和操作权限。常用的读取模式包括：

1. `'r'`（只读模式）：最常用。以只读方式打开文件，文件必须存在，指针位于文件开头。

2. `'rb'`（二进制只读模式）：以二进制格式只读方式打开文件，适用于非文本文件，如图片、音频等。

3. `'r+'`（读写模式）：以读写方式打开文件，文件必须存在，指针位于文件开头。

4. `'rb+'`（二进制读写模式）：以二进制格式读写方式打开文件，适用于需要读写非文本文件的情况。

以下示例展示了如何使用不同的读取模式读取文件内容。假设存在一个名为 `example.txt` 的文本文件，其内容如下：

```
Python 是一种广泛使用的高级编程语言.
它具有简洁的语法和强大的功能.
```

- 使用 `'r'` 模式读取文件：

```python
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

- 使用 `'rb'` 模式读取文件：

```python
with open('example.txt', 'rb') as file:
    content = file.read()
    print(content.decode('utf-8'))
```

- 使用 `'r+'` 模式读取并写入文件：

```
with open('example.txt', 'r+', encoding='utf-8') as file:
    content = file.read()
    print(content)
    file.write('\n这是追加的内容。')
```

- 使用 `'rb+'` 模式读取并写入二进制文件：

```
with open('example.txt', 'rb+') as file:
    content = file.read()
    print(content.decode('utf-8'))
    file.write('\n这是追加的内容。'.encode('utf-8'))
```

在上述示例中，`with` 语句用于确保文件在操作完成后自动关闭，`encoding='utf-8'` 参数指定了文件的编码方式，以正确处理包含非 ASCII 字符的内容。

文件的写入方法主要包括 `write()` 和 `writelines()`。这两种方法用于将数据写入文件，但适用场景有所不同。

1. `write()` 方法：

`write()` 方法用于将字符串写入文件。需要注意的是，`write()` 方法不会自动添加换行符，若需换行，需手动在字符串中加入\n。

```
# 打开文件 example.txt 进行写入
with open('example.txt', 'w', encoding='utf-8') as file:
    file.write('这是第一行内容.\n')
    file.write('这是第二行内容.')
```

执行上述代码后，`example.txt` 文件的内容为：

```
这是第一行内容.
这是第二行内容.
```

2. `writelines()` 方法：

`writelines()` 方法用于将字符串列表写入文件。与 `write()` 方法类似，`writelines()` 不会自动添加换行符，需在列表中的每个字符串末尾手动添加\n。

```
# 打开文件 example.txt 进行写入
with open('example.txt', 'w', encoding='utf-8') as file:
    lines = ['这是第一行内容.\n', '这是第二行内容.\n',
    ↪  '这是第三行内容.']
    file.writelines(lines)
```

执行上述代码后，`example.txt` 文件的内容为：

```
这是第一行内容.
这是第二行内容.
这是第三行内容.
```

3. `print()` 函数：

`print()` 函数不仅用于在控制台输出信息，还可用于将内容写入文本文件。通过指定 `print()` 函数的 `file` 参数，可以将输出重定向到文件对象，从而实现文件写入操作。

```
# 打开文件进行写入操作
with open('output.txt', 'w', encoding='utf-8') as file:
    # 使用 print() 函数将内容写入文件
    print("Hello, this is a sample text.", file=file)
    print("This text will be written to a file using the print
    ↪  function.", file=file)
```

注意：在使用 `print()` 函数写入文件时，默认会在每次调用后添加换行符。如需避免换行，可设置 `end` 参数，

文件的写入模式决定了文件的打开方式和操作权限。常用的写入模式包括：

1. `'w'`（写入模式）：最常用。以写入方式打开文件，若文件已存在，则清空其内容；若文件不存在，则创建新文件。

2. `'a+'`（追加读写模式）：以追加和读写方式打开文件，若文件已存在，指针位于文件末尾，可读取和追加内容；若文件不存在，则创建新文件。

3. `'a'`（追加模式）：以追加方式打开文件，若文件已存在，指针位于文件末尾，新的内容将添加到现有内容之后；若文件不存在，则创建新文件。

4. `'w+'`（读写模式）：以读写方式打开文件，若文件已存在，则清空其内容；若文件不存在，则创建新文件。

以下示例展示了如何使用不同的写入模式操作文件。假设存在一个名为 `example.txt` 的文本文件，其初始内容如下：

```
Python 是一种广泛使用的高级编程语言.
它具有简洁的语法和强大的功能.
```

- 使用 `'w'` 模式写入文件：

```
with open('example.txt', 'w', encoding='utf-8') as file:
    file.write('这是使用 w 模式写入的新内容。')
```

执行上述代码后，`example.txt` 的内容将被替换为：

```
这是使用 w 模式写入的新内容.
```

- 使用 `'a'` 模式追加内容：

```
with open('example.txt', 'a', encoding='utf-8') as file:
    file.write('\n这是使用 a 模式追加的内容.')
```

执行上述代码后，`example.txt` 的内容将变为：

```
这是使用 w 模式写入的新内容.
这是使用 a 模式追加的内容.
```

- 使用 `'w+'` 模式读写文件：

```
with open('example.txt', 'w+', encoding='utf-8') as file:
    file.write('这是使用 w+ 模式写入的新内容.')
```

执行上述代码后，`example.txt` 的内容为：

```
这是使用 w+ 模式写入的新内容.
```

- 使用 `'a+'` 模式追加并读取内容：

```
with open('example.txt', 'a+', encoding='utf-8') as file:
    file.write('\n这是使用 a+ 模式追加的内容.')
```

执行上述代码后，`example.txt` 的内容为：

```
这是使用 w+ 模式写入的新内容.
这是使用 a+ 模式追加的内容.
```

在上述示例中，`with` 语句用于确保文件在操作完成后自动关闭，`encoding='utf-8'` 参数指定了文件的编码方式，以正确处理包含非 ASCII 字符的内容。

文件路径的指定方式主要分为相对路径和绝对路径。理解并正确使用这两种路径对于文件操作至关重要。

1. 相对路径：

相对路径是相对于当前工作目录（current working directory, CWD）指定的文件或目录位置。使用相对路径时，路径的起点是当前正在访问的文件夹。相对路径更灵活，易于在不同机器或文件夹结构间移植代码。**但如果更改了工作目录，可能导致路径无效。**

假设当前工作目录为 `/home/user/project`，目录结构如下：

```
/home/user/project/
├── main.ipynb
└── data/
    └── example.txt
```

在 `main.ipynb` 中，使用相对路径读取 `example.txt`：

```
with open('data/example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

此时，`data/example.txt` 即为相对路径。

注意：在相对路径中 `.` 表述当前文件夹，`..` 表示上一层文件夹。

2. 绝对路径：

绝对路径是从文件系统的根目录开始的完整路径，提供了到达指定文件或目录的完整地址。绝对路径明确无误地指向文件位置，不受当前工作目录的影响。但不够灵活，当文件系统结构变化或在不同系统之间迁移代码时可能需要修改。

基于上述目录结构，使用绝对路径读取 `example.txt`：

```
with open('/home/user/project/data/example.txt', 'r',
↪   encoding='utf-8') as file:
    content = file.read()
    print(content)
```

`/home/user/project/data/example.txt` 即为绝对路径。

获取当前工作目录：使用 `os` 模块的 `getcwd()` 函数获取当前工作目录：

```
import os

current_directory = os.getcwd()
print(f"当前工作目录: {current_directory}") # 当前工作目录:
↪  /home/user/project
```

将相对路径转换为绝对路径：使用 `os.path` 模块的 `abspath()` 函数将相对路径转换为绝对路径：

```
import os

relative_path = 'data/example.txt'
absolute_path = os.path.abspath(relative_path)
print(f"绝对路径: {absolute_path}") # 绝对路径:
↪  /home/user/project/data/example.txt
```

在 Python 中，`seek()` 和 `tell()` 方法用于控制文件指针的位置，从而实现对文件的随机读写操作。`seek()` 方法用于移动文件指针到指定位置，`tell()` 方法用于获取当前文件指针的位置，偏移量和位置是以字节 (byte) 为单位。

示例文件内容：

假设有一个名为 `example.txt` 的文本文件，内容如下：

```
Hello, this is a sample text file.
It contains multiple lines of text.
Each line serves as an example.
```

使用 `seek()` 和 `tell()` 进行文件操作的示例代码：

```
# 打开文件进行读写操作
file =  open('example.txt', 'r+')
# 读取并打印第一行
first_line = file.readline()
print(f" 第一行内容: {first_line.strip()}")

# 获取当前文件指针位置
current_position = file.tell()
print(f" 当前文件指针位置: {current_position}")
```

代码解析：

1. 使用 `with open('example.txt', 'r+')` 打开文件，模式为 `'r+'`，表示以读写模式打开文件。
2. 使用 `readline()` 方法读取并打印文件的第一行内容。
3. 使用 `tell()` 方法获取当前文件指针的位置，并打印该位置。

```python
# 移动文件指针到文件开头
file.seek(0)
print(f" 文件指针已移动到位置: {file.tell()}")

# 在文件开头插入新内容
new_content = "Inserted line at the beginning.\n"
original_content = file.read()
file.seek(0)
file.write(new_content + original_content)
```

4. 使用 `seek(0)` 将文件指针移动到文件开头，并验证指针位置。

5. 在文件开头插入新内容。为此，先读取原始内容，移动指针到开头，然后将新内容和原始内容写入文件。

```python
# 移动文件指针到文件末尾
file.seek(0, 2)
print(f" 文件指针已移动到文件末尾位置: {file.tell()}")

# 在文件末尾追加新内容
file.write("\nAppended line at the end.")

# 关闭文件
file.close()
```

6. 使用 `seek(0, 2)` 将文件指针移动到文件末尾，`0` 表示偏移量，`2` 表示从文件末尾开始计算。
7. 在文件末尾追加新内容。
8. 关闭文件。

注意

- 使用 `seek()` 方法时，第二个参数 `whence` 的取值：`0` 表示从文件开头计算（默认值），`1` 表示从当前位置计算，`2` 表示从文件末尾计算。

- 在文本模式下（如 `'r+'`），`seek()` 和 `tell()` 的偏移量和位置是以字节 (byte) 为单位的。

THE END