**10 самых продуктивных техник для работы с файлами в Python**

15.01.2021

Какой бы проект вы ни разрабатывали, вам не избежать работы с файлами либо на компьютере, либо на сервере. И неудивительно, поскольку они являются самыми распространёнными контейнерами для хранения взаимосвязанной и обычно структурированной информации. При этом многим приходится выискивать конкретные операции, связанные с обработкой файлов. Поэтому мы решили посвятить данную статью 10 наиболее эффективным техникам для работы с файлами в Python.

**1. Показ текущей директории**

Чтобы узнать текущую рабочую директорию, мы можем просто ввести функцию getcwd() модуля *os,*как показано ниже:

>>> *# Узнаём текущую рабочую директорию*

... import os

... print("Current Work Directory:", os.getcwd())

...

Current Work Directory: /Users/ycui1/PycharmProjects/Medium\_Python\_Tutorials

>>> *# В качестве альтернативы можно использовать pathlib*

... from pathlib import Path

... print("Current Work Directory:", Path.cwd())

...

Current Work Directory: /Users/ycui1/PycharmProjects/Medium\_Python\_Tutorials

Данный код также демонстрирует возможность использования модуля pathlib для получения текущей рабочей директории. Обратите внимание, что для выполнения операций с файлами именно этот модуль является предпочтительным вариантом, и в статье вас ждёт немало примеров его употребления. Однако, если у вас устаревшая версия Python (< 3.4), то вам придётся использовать модуль os.

**2. Создание новой директории**

Для создания новой директории можно применить функцию mkdir(), которая выполнит эту операцию по конкретно заданному пути. Если вы просто укажете имя новой директории, то она будет создана в текущем каталоге:

>>> *# Создаём новую директорию в текущей*

... os.mkdir("test\_folder")

... print("Is the directory there:", os.path.exists("test\_folder"))

...

Is the directory there: True

>>> *# Создаём новую директорию в конкретно заданном каталоге*

... os.mkdir('/Users/ycui1/PycharmProjects/tmp\_folder')

... print("Is the directory there:", os.path.exists('/Users/ycui1/PycharmProjects/tmp\_folder'))

...

Is the directory there: True

Однако, если вы намерены создать новую директорию с несколькими вложенными уровнями (имеется в виду наличие одной папки внутри другой), то вам необходимо использовать функцию makedirs(). Обратимся к простому примеру:

>>> *# Создаём новую директорию с поддиректориями*

... os.makedirs('tmp\_level0/tmp\_level1')

... print("Is the directory there:", os.path.exists("tmp\_level0/tmp\_level1"))

...

Is the directory there: True

Если же у вас последние версии Python (≥ 3.4), то для решения вышеуказанной задачи можно воспользоваться преимуществом модуля pathlib. При этом он способен не только создавать поддиректории, но также при необходимости работать с каталогами, отсутствующими в пути. Рассмотрим пример:

*# Используем pathlib*

from pathlib import Path

Path("test\_folder").mkdir(parents=True, exist\_ok=True)

Имейте в виду, что попытка повторного выполнения вышеприведённого кода может вызвать проблемы  —  вы не сможете создать новую директорию, если такая уже существует. Стоит отметить, что эта проблема решается путём присвоения аргументу exist\_ok значения True, как показано выше. А вот значение False, установленное для него по умолчанию, не позволит повторно создать уже существующую директорию и приведёт к ошибке.

>>> *# Используем pathlib*

... from pathlib import Path

... Path("test\_folder").mkdir(parents=True, exist\_ok=False)

...

Traceback (most recent call last):

File "<input>", line 3, in <module>

File "/Users/ycui1/.conda/envs/Medium/lib/python3.8/pathlib.py", line 1284, in mkdir

self.\_accessor.mkdir(self, mode)

FileExistsError: [Errno 17] File exists: 'test\_folder'

**3. Удаление директорий и файлов**

По завершении операций с файлами или папками, возможно, потребуется их удалить, чтобы упорядочить ресурсы компьютера. Для удаления файла в модуле *os*применяется функция remove()*,*а для удаления папки  —  функция rmdir(). Попытка же удалить директорию с помощью remove()вызовет ошибку. Рассмотрим применение этих функций:

>>> *# Удаление файла*

... print(f"\* Before deleting file {os.path.isfile('tmp.txt')}")

... os.remove('tmp.txt')

... print(f"\* After deleting file {os.path.exists('tmp.txt')}")

...

\* Before deleting file True

\* After deleting file False

>>> *# Удаление директории*

... print(f"\* Before deleting directory {os.path.isdir('tmp\_folder')}")

... os.rmdir('tmp\_folder')

... print(f"\* After deleting directory {os.path.exists('tmp\_folder')}")

...

\* Before deleting directory True

\* After deleting directory False

При использовании модуля *pathlib*за удаление файла отвечает метод unlink(), а за удаление директории  —  rmdir(). Обратите внимание, что они оба являются методами экземпляра объекта Path.

**4. Получение списка файлов**

В процессе обработки данных для аналитики или проектов МО вам потребуется получить список файлов в определённой директории. Зачастую их имена соответствуют определённому шаблону. Допустим, мы хотим найти все файлы .txt в директории. Далее рассмотрим, как это можно сделать с помощью метода glob() с объектом Path. Обратите внимание, что данный метод создаёт генератор с возможностью итерации. Следующий код наглядно демонстрирует создание генератором списка путей файлов:

>>> txt\_files = list(Path('.').glob("\*.txt"))

... print("Txt files:", txt\_files)

...

Txt files: [PosixPath('hello\_world.txt'), PosixPath('hello.txt')]

Как вариант, также удобно использовать модуль *glob*напрямую, как показано ниже. Он располагает аналогичной функциональностью, создавая списки имён файлов, с которыми впоследствии можно работать. Заметьте, что Path.glob() создаёт пути. Оба метода будут работать в большинстве сценариев, таких как чтение и запись файлов.

>>> from glob import glob

... files = list(glob('h\*'))

... print("Files starting with h:", files)

...

Files starting with h: ['hello\_world.txt', 'hello.txt']

**5. Перемещение и копирование файлов**

Перемещение и копирование  —  одна из стандартных задач управления файлами, которая довольно легко решается в Python. Для перемещения вы просто переименовываете файл, заменяя его старую директорию целевой. Предположим, необходимо переместить все файлы .txt в другую папку. В следующем примере кода мы увидим, как это можно сделать с помощью модуля *pathlib:*

>>> target\_folder = Path("target\_folder")

... target\_folder.mkdir(parents=True,exist\_ok=True)

... source\_folder = Path('.')

...

... txt\_files = source\_folder.glob('\*.txt')

... for txt\_file in txt\_files:

... filename = txt\_file.name

... target\_path = target\_folder.joinpath(filename)

... print(f"\*\* Moving file {filename}")

... print("Target File Exists:", target\_path.exists())

... txt\_file.rename(target\_path)

... print("Target File Exists:", target\_path.exists(), '\n')

...

\*\* Moving file hello\_world.txt

Target File Exists: False

Target File Exists: True

\*\* Moving file hello.txt

Target File Exists: False

Target File Exists: True

Копирование же можно выполнить при помощи функциональности, доступной в *shutil,*ещё одном полезном модуле из стандартной библиотеки для операций с файлами. Здесь за это отвечает функция copy(), в которой исходный и целевой файлы указываются в виде строк. Ниже вы увидите простой пример. Конечно, вы можете объединить функции copy() и glob() для работы с группой файлов, соответствующих одному паттерну.

>>> import shutil

...

... source\_file = "target\_folder/hello.txt"

... target\_file = "hello2.txt"

... target\_file\_path = Path(target\_file)

... print("\* Before copying, file exists:", target\_file\_path.exists())

... shutil.copy(source\_file, target\_file)

... print("\* After copying, file exists:", target\_file\_path.exists())

...

\* Before copying, file exists: False

\* After copying, file exists: True

**6. Проверка директории/файла**

На самом деле, эта операция уже много раз встречалась в вышеприведённых примерах. В них для проверки того, существует ли конкретный путь, применялся метод exists(). При условии положительного ответа он возвращает True, в противном случае  —  False. Примечательно, что эта функция доступна в обоих модулях, *os* и *pathlib,*но с разными сигнатурами. Рассмотрим соответствующие примеры их применения:

*# exists() в модуле os*

os.path.exists('path\_to\_check')

*# exists() в модуле pathlib*

Path('directory\_path').exists()

В модуле *pathlib м*ожно также проверить, является ли путь директорией или файлом с готовыми к вызову функциями*.*Обратимся к следующему примеру:

*# Проверяем, является ли путь директорией*

os.path.isdir('path\_to\_check')

Path('path\_to\_check').is\_dir()

*# Проверяем, является ли путь файлом*

os.path.isfile('path\_to\_check')

Path('path\_to\_check').is\_file()

**7. Получение информации о файле**

При работе с файлами во многих сценариях возникает необходимость извлечения их имён. С объектом Path это просто как дважды два, и вы уже были свидетелями его применения. Можно просто извлечь атрибут name файлового объекта Path. Если же вам нужно узнать только имя без расширения, то извлекать следует атрибут stem. Следующий фрагмент кода демонстрирует соответствующие случаи применения:

for py\_file in Path().glob('c\*.py'):

... print('Name with extension:', py\_file.name)

... print('Name only:', py\_file.stem)

...

Name with extension: closures.py

Name only: closures

Name with extension: counter.py

Name only: counter

Name with extension: context\_management.py

Name only: context\_management

В отдельных случаях вам потребуется узнать расширение файла, с которым вы работаете. Чаще всего можно воспользоваться атрибутом suffix файлового объекта Path, как показано ниже:

>>> file\_path = Path('closures.py')

... print("File Extension:", file\_path.suffix)

...

File Extension: .py

Если необходимо получить больше информации о файле, например, его размер и время изменения, то для этого в нашем распоряжении есть метод stat(), принцип действия которого аналогичен os.stat(), знакомого тем, кто привык работать с модулем *os.*

>>> *# Получаем объект st\_stat из пути*

... current\_file\_path = Path('iterable\_usages.py')

... file\_stat = current\_file\_path.stat()

...

>>> *# Получаем информацию о размере файла:*

... print("File Size in Bytes:", file\_stat.st\_size)

File Size in Bytes: 3531

>>> *# Получаем информацию о времени последнего обращения*

... print("When Most Recent Access:", file\_stat.st\_atime)

When Most Recent Access: 1595435202.310935

>>> *# Получаем информацию о времени последнего изменения содержимого*

... print("When Most Recent Modification:", file\_stat.st\_mtime)

When Most Recent Modification: 1594127561.3204417

**8. Чтение файлов**

Одна из важнейших операций с файлами  —  считывание их данных. В конце концов, содержимое файла является, вероятно, единственной причиной его появления. Самый традиционный способ состоит в создании файлового объекта с помощью встроенной функции open(). По умолчанию она откроет файл в режиме чтения и будет работать с его данными как с текстом. Рассмотрим пример:

>>> *# Чтение всех текстов*

... with open("hello2.txt", 'r') as file:

... print(file.read())

...

Hello World!

Hello Python!

>>> *# Чтение построчно*

... with open("hello2.txt", 'r') as file:

... for i, line in enumerate(file, 1):

... print(f"\* Reading line #{i}: {line}")

...

\* Reading line *#1: Hello World!*

\* Reading line *#2: Hello Python!*

Этот код демонстрирует самые распространённые способы чтения содержимого. Если вы знаете, что ваш файл включает немного данных, можете считать их все за раз с помощью метода read(). Но если он очень крупный, то следует рассмотреть вариант с генератором, роль которого выполняет файловый объект. Он обрабатывает данные построчно и тем самым экономно расходует память, обходясь без загрузки всех данных при применении read().

Как уже ранее упоминалось, функция open() по умолчанию работает с содержимым файла как с текстом. Однако в случае с бинарными файлами необходимо явно задать данное условие. Например, вместо ‘r’ следует ввести ‘rb’. Это требование также относится и к записи файлов, о чём мы поговорим далее. Ещё один непростой момент связан с кодировкой файла. Во многих случаях функция open() сможет выполнить эту операцию за нас, и большинство файлов, с которыми мы работаем, будут в кодировке “utf-8”. Если же вы обрабатываете файлы, применяя другие форматы кодировки, вам следует установить аргумент encoding.

**9. Запись файлов**

Для записи данных можно опять же создать файловый объект, открыв файл в режиме записи (‘w’) или дозаписи (‘a’). В первом случае при записи данных в файл его старое содержимое удаляется, а во втором  —  данные добавляются в конец файла. Рассмотрим пример работы этих двух режимов в следующем фрагменте кода.

>>> *# Запись новых данных в файл*

... with open("hello3.txt", 'w') as file:

... text\_to\_write = "Hello Files From Writing"

... file.write(text\_to\_write)

...

>>> *# Добавление данных*

... with open("hello3.txt", 'a') as file:

... text\_to\_write = "\nHello Files From Appending"

... file.write(text\_to\_write)

...

>>> *# Проверяем, содержит ли файл правильные данные*

... with open("hello3.txt") as file:

... print(file.read())

...

Hello Files From Writing

Hello Files From Appending

Этот код подтверждает, что мы можем записывать данные в двух режимах: записи и дозаписи. Обратили вы внимание или нет, но каждый раз при открытии файла использовалась инструкция *with*. Объясняется это тем, что она создаёт контекст для обработки файла и помогает закрыть файловый объект по завершении операций. Если же вовремя этого не сделать, то открытый файловый объект может повредиться.

**10. Архивирование и разархивирование файлов**

При работе с большим числом файлов может потребоваться их архивирование для долгосрочного хранения или временной передачи. Соответствующие возможности предоставляются модулем *zipfile.* Для архивирования файлов функцией ZipFile() создаётся файловый объект zip, что напоминает случай с функцией open(), поскольку обе эти функции предусматривают создание файлового объекта, управляемого контекстным менеджером (вспоминаете применение инструкции *with*?). Обратимся к фрагменту кода с простым примером:

>>> from zipfile import ZipFile

...

... *# Создание zip-файла, содержащего все текстовые файлы в директории*

... with ZipFile('text\_files.zip', 'w') as file:

... for txt\_file in Path().glob('\*.txt'):

... print(f"\*Add file: {txt\_file.name} to the zip file")

... file.write(txt\_file)

...

\*Add file: hello3.txt to the zip file

\*Add file: hello2.txt to the zip file

Вы можете получить zip-файл из внешнего источника, и вам потребуется извлечь из него файлы. Чтобы не усложнять, допустим, что мы распаковываем их в текущую директорию. Обратите внимание на то, что имена файлов в zip-файле совпадают с содержащимися в директории, вследствие чего последние будут перезаписаны без предупреждения. Поэтому вам следует рассмотреть вариант извлечения содержимого zip-файла в отдельную папку, где такой проблемы перезаписи не возникнет.

>>> *# Для разархивирования только что созданного файла*

... with ZipFile('text\_files.zip') as zip\_file:

... zip\_file.printdir()

... zip\_file.extractall()

...

File Name Modified Size

hello3.txt 2020-07-30 20:29:50 51

hello2.txt 2020-07-30 18:29:52 26

**Заключение**

Итак, в данной статье мы рассмотрели 10 наиболее полезных операций по работе с файлами. Как вы могли убедиться, все они укладываются в несколько строк кода, поэтому не представляют абсолютно никакой сложности. Если вы с ними не знакомы, то надеемся, что эта статья послужит для вас кратким руководством в процессе работы с файлами.