Лабораторная работа 2.19. Работа с файловой системе в Python3 с использованием модуля `pathlib`

Цель работы: *приобретение навыков по работе с файловой системой с помощью библиотеки* `pathlib` *языка программирования Python версии 3.x.*

Ход работы

До Python 3.4 работа с путями файловой системы осуществлялась либо с помощью методов строк:

```
>>> path.rsplit('\\', maxsplit=1)[0]
```

либо с помощью модуля `os.path`:

```
>>> os.path.isfile(os.path.join(os.path.expanduser('~'), 'realpython.txt'))
```

В данной лабораторной работе показаано, как работать с путями к файлам - именами каталогов и файлов - в Python. Вы изучите новые способы чтения и записи файлов, манипулирования путями и базовой файловой системой, а также увидите несколько примеров того, как составлять список файлов и выполнять их итерацию. Используя модуль `pathlib`, два приведенных выше примера можно переписать, используя элегантный, читаемый и Pythonic-код, например:

```
>>> path.parent
>>> (pathlib.Path.home()/'realpython.txt').is_file()
```

Проблема с обработкой пути к файлу Python

Работа с файлами и взаимодействие с файловой системой важны по многим различным причинам. Простейшие случаи могут включать только чтение или запись файлов, но иногда возникают более сложные задачи. Может быть, вам нужно перечислить все файлы в каталоге данного типа, найти родительский каталог данного файла или создать уникальное имя файла, которое еще не существует.

Традиционно Python представлял пути к файлам, используя обычные текстовые строки. При поддержке стандартной библиотеки `os.path` это было достаточно, хотя и немного громоздко (как второй пример во введении шоу). Однако, поскольку paths не являются строками, важные функции распространяются по всей стандартной библиотеке, включая такие библиотеки, как `os`, `glob` и `shutil`. В следующем примере нужны три оператора `import`, чтобы переместить все текстовые файлы в каталог архива:

```
import glob
import os
import shutil

for file_name in glob.glob('*.txt'):
    new_path = os.path.join('archive', file_name)
    shutil.move(file_name, new_path)
```

С путями, представленными строками, возможно, но обычно плохая идея, использовать обычные строковые методы. Например, вместо того, чтобы соединять два пути с `+`, как обычные строки, вы должны использовать `os.path.join()`, который соединяет пути, используя правильный разделитель пути в операционной системе. Напомним, что Windows использует `\`, а Mac и Linux используют `/` в качестве разделителя. Это различие может привести к трудно обнаруживаемым ошибкам, таким как наш первый пример во введении, работающий только для путей Windows.

Модуль `pathlib` был введен в Python 3.4 (PEP 428) для решения этих проблем. Он объединяет необходимые функции в одном месте и делает его доступным через методы и свойства простого в использовании объекта `Path`.

Ранее другие пакеты все еще использовали строки для путей к файлам, но начиная с Python 3.6 модуль `pathlib` поддерживается во всей стандартной библиотеке, частично из-за добавления протокол пути к файловой системе.

Создание путей

Все, что вам действительно нужно знать, это класс `pathlib.Path`. Есть несколько разных способов создания пути. Прежде всего, существуют classmethods наподобие `.cwd()` (текущий рабочий каталог) и `.home()` (домашний каталог вашего пользователя):

```
>>> import pathlib
>>> pathlib.Path.cwd()
PosixPath('/home/gahjelle/realpython/')
```

> *Примечание:* В этом уроке мы будем предполагать, что `pathlib` был импортирован, без указания `import pathlib`, как указано выше. Поскольку вы будете в основном использовать класс `Path`, вы также можете сделать из pathlib `import Path` и написать `Path` вместо `pathlib.Path`.

Путь также может быть явно создан из его строкового представления:

```
>>> pathlib.Path(r'C:\Users\gahjelle\realpython\file.txt')
WindowsPath('C:/Users/gahjelle/realpython/file.txt')
```

Небольшой совет для работы с путями Windows: в Windows разделитель пути - это обратный слеш, `\`. Однако во многих случаях обратная косая черта также используется в качестве символа *escape* для представления непечатаемых символов. Чтобы избежать проблем, используйте *raw string literals* для представления путей Windows. Это строковые литералы, перед которыми стоит `r`. В необработанных строковых литералах `\` представляет обратную косую черту: `r'C:\Users '`.

Третий способ построения пути - это соединение частей пути с помощью специального оператора `/`. Оператор прямой косой черты используется независимо от фактического разделителя пути на платформе:

```
>>> pathlib.Path.home()/'python'/'scripts'/'test.py'
PosixPath('/home/gahjelle/python/scripts/test.py')
```

Операция `/` может объединять несколько путей или набор путей и строк (как указано выше), если есть хотя бы один объект `Path`. Если вам не нравятся специальные обозначения `/`, вы можете сделать то же самое с помощью метода `.joinpath()`:

```
>>> pathlib.Path.home().joinpath('python', 'scripts', 'test.py')
PosixPath('/home/gahjelle/python/scripts/test.py')
```

Обратите внимание, что в предыдущих примерах `pathlib.Path` представлен либо `WindowsPath`, либо `PosixPath`. Фактический объект, представляющий путь, зависит от базовой операционной системы. (То есть пример `WindowsPath` был выполнен в Windows, в то время как примеры `PosixPath` были выполнены в Mac или Linux.)

Чтение и запись файлов

Традиционно для чтения или записи файла в Python использовалась встроенная функция `open()`. Это все еще верно, поскольку функция `open()` может напрямую использовать объекты `Path`. Следующий пример находит все заголовки в файле Markdown и печатает их:

```
path = pathlib.Path.cwd() / 'test.md'
with open(path, mode='r') as fid:
    headers = [line.strip() for line in fid if line.startswith('#')]
print('\n'.join(headers))
```

Эквивалентной альтернативой является вызов `.open()` для объекта `Path`:

```
with path.open(mode='r') as fid:
    ...
```

Фактически, `Path.open()` вызывает встроенную функцию `open()` за кулисами. Какой вариант вы используете, это в основном дело вкуса.

Для простого чтения и записи файлов в библиотеке `pathlib` есть несколько удобных методов:

- `.read_text()`: открыть путь в текстовом режиме и вернуть содержимое в виде строки.
- `.read_bytes()`: открыть путь в двоичном/байтовом режиме и вернуть содержимое в виде строки байтов.
- `.write_text()`: открыть путь и записать в него строковые данные.
- `.write_bytes()`: открыть путь в двоичном/байтовом режиме и записать в него данные.

Каждый из этих методов обрабатывает открытие и закрытие файла, делая их тривиальными, например:

```
>>> path = pathlib.Path.cwd() / 'test.md'
>>> path.read_text()
<the contents of the test.md-file>
```

Пути также могут быть указаны как простые имена файлов, и в этом случае они интерпретируются относительно текущего рабочего каталога. Следующий пример эквивалентен предыдущему:

```
>>> pathlib.Path('test.md').read_text()
<the contents of the test.md-file>
```

Метод `.resolve()` найдет полный путь. Ниже мы подтверждаем, что текущий рабочий каталог используется для простых имен файлов:

```
>>> path = pathlib.Path('test.md')
>>> path.resolve()
PosixPath('/home/gahjelle/realpython/test.md')
>>> path.resolve().parent == pathlib.Path.cwd()
False
```

Обратите внимание, что при сравнении путей сравниваются их представления. В приведенном выше примере `path.parent` не равно `pathlib.Path.cwd()`, поскольку `path.parent` представляется `'.'`, А `pathlib.Path.cwd()` представляется как `'/home/gahjelle/realpython/'`.

Выделение компонентов пути

Различные части пути удобно доступны как свойства. Основные примеры включают в себя:

- `.name`: имя файла без какого-либо каталога
- `.parent`: каталог, содержащий файл, или родительский каталог, если путь является каталогом
- `.stem`: имя файла без суффикса
- `.suffix`: расширение файла
- `.anchor`: часть пути перед каталогами

Вот эти свойства в действии:

```
>>> path
PosixPath('/home/gahjelle/realpython/test.md')
>>> path.name
'test.md'
>>> path.stem
'test'
>>> path.suffix
'.md'
>>> path.parent
PosixPath('/home/gahjelle/realpython')
>>> path.parent.parent
PosixPath('/home/gahjelle')
>>> path.anchor
'/'
```

Обратите внимание, что `.parent` возвращает новый объект `Path`, тогда как другие свойства возвращают строки. Это означает, например, что `.parent` можно объединить в цепочки, как в предыдущем примере, или даже объединить с `/` для создания совершенно новых путей:

```
>>> path.parent.parent/('new' + path.suffix)
PosixPath('/home/gahjelle/new.md')
```

Превосходный Pathlib Cheatsheet обеспечивает визуальное представление этих и других свойств и методов.

Перемещение и удаление файлов

Через `pathlib` вы также получаете доступ к базовым операциям на уровне файловой системы, таким как перемещение, обновление и даже удаление файлов. По большей части эти методы не выдают предупреждение и не ждут подтверждения, прежде чем информация или файлы будут потеряны. Будьте осторожны при использовании этих методов.

Чтобы переместить файл, используйте `.replace()`. Обратите внимание, что если место назначения уже существует, `.replace()` перезапишет его. К сожалению, `pathlib` явно не поддерживает безопасное перемещение файлов. Чтобы избежать возможной перезаписи пути назначения, проще всего проверить, существует ли место назначения перед заменой:

```
if not destination.exists():
    source.replace(destination)
```

Тем не менее, это оставляет дверь открытой для возможного состояния гонки. Другой процесс может добавить файл по пути `destination` между выполнением оператора `if` и метода `.replace()`. Если это вызывает озабоченность, более безопасный способ - открыть путь назначения для создания exclusive и явно скопировать исходные данные:

```
with destination.open(mode='xb') as fid:
    fid.write(source.read_bytes())
```

Приведенный выше код вызовет `FileExistsError`, если `destination` уже существует. Технически это копирует файл. Чтобы выполнить перемещение, просто удалите `source` после завершения копирования.

Когда вы переименовываете файлы, полезными методами могут быть `.with_name()` и `.with_suffix()`. Они оба возвращают исходный путь, но с замененным именем или суффиксом соответственно.

Например:

```
>>> path
PosixPath('/home/gahjelle/realpython/test001.txt')
>>> path.with_suffix('.py')
PosixPath('/home/gahjelle/realpython/test001.py')
>>> path.replace(path.with_suffix('.py'))
```

Каталоги и файлы могут быть удалены с помощью `.rmdir()` и `.unlink()` соответственно.

Примеры

В этом разделе вы увидите несколько примеров использования `pathlib` для решения простых задач.

Подсчет файлов

Есть несколько разных способов перечислить много файлов. Самым простым является метод `.iterdir()`, который перебирает все файлы в данном каталоге. В следующем примере комбинируется `.iterdir()` с классом `collection.Counter` для подсчета количества файлов каждого типа в текущем каталоге:

```
>>> import collections
>>> collections.Counter(p.suffix for p in pathlib.Path.cwd().iterdir())
Counter({'.md': 2, '.txt': 4, '.pdf': 2, '.py': 1})
```

Более гибкие списки файлов могут быть созданы с помощью методов `.glob()` и `.rglob()` (рекурсивный глоб). Например, `pathlib.Path.cwd().glob('*.txt')` возвращает все файлы с суффиксом `.txt` в текущем каталоге. Следующее только подсчитывает типы файлов, начинающиеся с `p`:

```
>>> import collections
>>> collections.Counter(p.suffix for p in pathlib.Path.cwd().glob('*.p*'))
Counter({'.pdf': 2, '.py': 1})
```

Показать дерево каталогов

В следующем примере определяется функция `tree()`, которая будет печатать визуальное дерево, представляющее иерархию файлов, с корнем в данном каталоге. Здесь мы также хотим перечислить подкаталоги, поэтому мы используем метод `.rglob()`:

```
def tree(directory):
    print(f'+ {directory}')
    for path in sorted(directory.rglob('*')):
        depth = len(path.relative_to(directory).parts)
        spacer = '    ' * depth
        print(f'{spacer}+ {path.name}')
```

Обратите внимание, что нам нужно знать, как далеко от корневого каталога находится файл. Чтобы сделать это, мы сначала используем `.relative_to()`, чтобы представить путь относительно корневого каталога. Затем мы подсчитываем количество каталогов (используя свойство `.parts`) в представлении. При запуске эта функция создает визуальное дерево, подобное следующему:

```
>>> tree(pathlib.Path.cwd())
+/home/gahjelle/realpython
    + directory_1
        + file_a.md
    + directory_2
        + file_a.md
        + file_b.pdf
        + file_c.py
    + file_1.txt
    + file_2.txt
```

> *Примечание:* f-strings работают только в Python 3.6 и более поздних версиях. В старых Pythons выражение `f'{spacer} {path.name}'` можно записать как `'{0} {1}'.format(spacer, path.name)`.

Найти последний измененный файл

Методы `.iterdir()`, `.glob()` и `.rglob()` отлично подходят для выражений генератора и понимания списка. Чтобы найти файл в каталоге, который был последний раз изменен, вы можете использовать метод `.stat()` для получения информации о базовых файлах. Например, `.stat().st_mtime` дает время последней модификации файла:

```
>>> from datetime import datetime
>>> time, file_path = max((f.stat().st_mtime, f) for f in directory.iterdir())
>>> print(datetime.fromtimestamp(time), file_path)
2018-03-23 19:23:56.977817/home/gahjelle/realpython/test001.txt
```

Вы даже можете получить содержимое файла, который был последний раз изменен, с помощью аналогичного выражения:

```
>>> max((f.stat().st_mtime, f) for f in directory.iterdir())[1].read_text()
<the contents of the last modified file in directory>
```

Временная метка, возвращенная из различных свойств `.stat().st_`, представляет секунды с 1 января 1970 года. В дополнение к `datetime.fromtimestamp`, `time.localtime` или `time.ctime` могут использоваться для преобразования временной метки в нечто более пригодное для использования.

Создать уникальное имя файла

Последний пример покажет, как создать уникальное нумерованное имя файла на основе шаблона. Сначала укажите шаблон для имени файла с местом для счетчика. Затем проверьте существование пути к файлу, созданного путем соединения каталога и имени файла (со значением счетчика). Если он уже существует, увеличьте счетчик и попробуйте снова:

```
def unique_path(directory, name_pattern):
    counter = 0
    while True:
        counter += 1
        path = directory/name_pattern.format(counter)
        if not path.exists():
            return path

path = unique_path(pathlib.Path.cwd(), 'test{:03d}.txt')
```

Если каталог уже содержит файлы `test001.txt` и `test002.txt`, приведенный выше код установит для `path` значение `test003.txt`.

Отличия операционной системы

Ранее мы отмечали, что когда мы создавали экземпляр `pathlib.Path`, возвращался либо объект `WindowsPath`, либо `PosixPath`. Тип объекта будет зависеть от операционной системы, которую вы используете. Эта функция позволяет довольно легко писать кроссплатформенный код. Можно явно запросить `WindowsPath` или `PosixPath`, но вы будете ограничивать свой код только этой системой без каких-либо преимуществ. Такой конкретный путь не может быть использован в другой системе:

```
>>> pathlib.WindowsPath('test.md')
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

В некоторых случаях может потребоваться представление пути без доступа к базовой файловой системе (в этом случае также может иметь смысл представлять путь Windows в системе, отличной от Windows, или наоборот). Это можно сделать с помощью объектов `PurePath`.

```
>>> path = pathlib.PureWindowsPath(r'C:\Users\gahjelle\realpython\file.txt')
>>> path.name
'file.txt'
>>> path.parent
PureWindowsPath('C:/Users/gahjelle/realpython')
>>> path.exists()
AttributeError: 'PureWindowsPath' object has no attribute 'exists'
```

Вы можете напрямую создать экземпляр `PureWindowsPath` или `PurePosixPath` во всех системах. Создание экземпляра `PurePath` вернет один из этих объектов в зависимости от используемой операционной системы.

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.
2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Проработайте примеры лабораторной работы. Создайте для них отдельные модули языка Python. Зафиксируйте изменения в репозитории.

8. Приведите в отчете скриншоты результатов выполнения примера при различных исходных данных вводимых с клавиатуры.
9. Зафиксируйте сделанные изменения в репозитории.
10. Приведите в отчете скриншоты работы программ решения индивидуальных заданий.
11. Зафиксируйте сделанные изменения в репозитории.
12. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
13. Выполните слияние ветки для разработки с веткой *master/main*.
14. Отправьте сделанные изменения на сервер GitHub.
15. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Индивидуальные задания

Задание 1

Для своего варианта лабораторной работы 2.17 добавьте возможность хранения файла данных в домашнем каталоге пользователя. Для выполнения операций с файлами необходимо использовать модуль `pathlib`.

Задание 2

Разработайте аналог утилиты tree в Linux. Используйте возможности модуля `argparse` для управления отображением дерева каталогов файловой системы. Добавьте дополнительные уникальные возможности в данный программный продукт.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется электронно в формате PDF, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. Какие существовали средства для работыс файловой системой до Pytho 3.4?
2. Что регламентирует PEP 428?
3. Как осуществляется создание путей средствами модуля `pathlib`?
4. Как получить путь дочернего элемента файловой системы с помощью модуля `pathlib`?
5. Как получить путь к родительским элементам файловой системы с помощью модуля `pathlib`?
6. Как выполняются операции с файлами с помощью модуля `pathlib`?
7. Как можно выделить компоненты пути файловой системы с помощью модуля `pathlib`?
8. Как выполнить перемещение и удаление файлов с помощью модуля `pathlib`?
9. Как выполнить подсчет файлов в файловой системе?
10. Как отобразить дерево каталогов файловой системы?
11. Как создать уникальное имя файла?
12. Каковы отличия в использовании модуля `pathlib` для различных операционных систем?