Тема 8. Модуль OS и работа с файловой системой, Модули shutil, pathlib, glob и os.path

I. Модуль OS и работа с файловой системой

Ряд возможностей по работе с каталогами и файлами предоставляет встроенный модуль **os**. Он содержит много функций, ниже рассмотрены только основные из них:

- mkdir(): создает новую папку
- rmdir(): удаляет папку
- rename(): переименовывает файл
- remove(): удаляет файл

Создание и удаление папки

Для создания папки применяется функция mkdir(), в которую передается путь к создаваемой папке:

```
import os

# путь относительно текущего скрипта
os.mkdir("hello")
# абсолютный путь
os.mkdir("c://somedir")
os.mkdir("c://somedir/hello")
```

Для удаления папки используется функция rmdir(), в которую передается путь к удаляемой папке:

```
import os

# путь относительно текущего скрипта
os.rmdir("hello")
# абсолютный путь
os.rmdir("c://somedir/hello")
```

Переименование файла

Для переименования вызывается функция rename(source, target), первый параметр которой - путь к исходному файлу, а второй - новое имя файла. В качестве путей могут использоваться как абсолютные, так и относительные.

Например, пусть в папке C://SomeDir/ располагается файл somefile.txt. Переименуем его в файл "hello.txt":

```
import os

os.rename("C://SomeDir/somefile.txt",
                        "C://SomeDir/hello.txt")
```

Удаление файла

Для удаления вызывается функция `remove()`, в которую передается путь к файлу:

```
import os

os.remove("C://SomeDir/hello.txt")
```

Существование файла

Если попытаться открыть файл, который не существует, то Python «выбросит» исключение `FileNotFoundError`.

Для «отлова» исключения можно использовать конструкцию `try...except`. Однако можно уже до открытия файла проверить, существует ли он или нет с помощью метода `os.path.exists(path)`.

В этот метод передается полное имя файла (путь и имя), существование которого необходимо проверить:

```
filename = input("Введите путь к файлу: ")
if os.path.exists(filename):
    print("Указанный файл существует")
else:
    print("Файл не существует")
```

Работа с операционной системой

Модуль os так же предоставляет множество функций для работы с операционной системой, причём их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми. Здесь будут приведены наиболее часто используемые из них (некоторые функции из этого модуля поддерживаются не всеми ОС).

os.name - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

os.environ - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

os.getlogin() - имя пользователя, вошедшего в терминал (Unix).

os.getpid() - текущий id процесса.

os.uname() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

os.access(path, mode, *, dir_fd=None, effective_ids=False, follow_symlinks=True)

- проверка доступа к объекту у текущего пользователя.

Флаги: **os.F_OK** - объект существует, **os.R_OK** - доступен на чтение, **os.W_OK** - доступен на запись, **os.X_OK** - доступен на исполнение.

os.chdir(path) - смена текущей директории.

os.chmod(path, mode, *, dir_fd=None, follow_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

os.chown(path, uid, gid, *, dir_fd=None, follow_symlinks=True) - меняет id владельца и группы (Unix).

os.getcwd() - текущая рабочая директория.

os.link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True) - создаёт жёсткую ссылку.

os.listdir(path=".") - список файлов и директорий в папке.

os.mkdir(path, mode=0o777, *, dir_fd=None) - создаёт директорию. OSError, если директория существует.

os.makedirs(path, mode=0o777, exist_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

os.remove(path, *, dir_fd=None) - удаляет путь к файлу.

os.rename(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает файл или директорию из src в dst.

os.renames(old, new) - переименовывает old в new, создавая промежуточные директории.

os.replace(src, dst, *, src_dir_fd=None, dst_dir_fd=None) - переименовывает из src в dst с принудительной заменой.

os.rmdir(path, *, dir_fd=None) - удаляет пустую директорию.

os.removedirs(path) - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

os.symlink(source, link_name, target_is_directory=False, *, dir_fd=None) - создаёт символическую ссылку на объект.

os.sync() - записывает все данные на диск (Unix).

os.truncate(path, length) - обрезает файл до длины length.

os.utime(path, times=None, *, ns=None, dir_fd=None, follow_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

os.walk(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

Пример – вывести на консоль список всех *.txt файлов, размещенных в папке l:\dist и её подпапках:

```
import os
from os.path import join
for (root, dirs, files) in os.walk('l:\\dist'):
```

```
for filename in files:
      if filename.endswith('.txt'):
            thefile = os. path.join(root, filename)
            print( os.path.getsize(thefile), thefile)
```

os.system(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

os.urandom(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

II. Пример команд работы с файлами и файловой системой

Рассмотрим 8-м крайне важных команд для работы с файлами, папками и файловой системой в целом.

Показать текущий каталог

Самая простая и вместе с тем одна из самых важных команд для Python-разработчика. Она нужна потому, что чаще всего разработчики имеют дело с относительными путями. Но в некоторых случаях важно знать, где мы находимся.

Относительный путь хорош тем, что работает для всех пользователей, с любыми системами, количеством дисков и так далее.

Так вот, для того чтобы показать текущий каталог используем встроенную в Python OS-библиотеку:

```
import os
os.getcwd()
```

Имейте в виду, что возвращаемый путь является абсолютным.

Проверяем, существует файл или каталог

Прежде чем задействовать команду по созданию файла или каталога, стоит убедиться, что аналогичных элементов нет. Это поможет избежать ряда ошибок при работе приложения, включая перезапись существующих элементов с данными.

Функция os.path.exists () принимает аргумент строкового типа, который может быть либо именем каталога, либо файлом.

Проверим, существует ли каталог sample_data:

```
os.path.exists('sample_data')
```

```
[3]  os.path.exists('sample_data')

     True
```

Эта же команда подходит и для работы с файлами:

```
os.path.exists('sample_data/README.md')
```

```
[4]  os.path.exists('sample_data/README.md')

     True
```

Если папки или файла нет, команда возвращает false.

```
[5]  os.path.exists('foobar')

     False
```

Объединение компонентов пути

В предыдущем примере использовался слеш "/" для разделителя компонентов пути. В принципе это нормально, но не рекомендуется. Если вы хотите, чтобы ваше приложение было кросс платформенным, такой вариант не подходит. Так, некоторые старые версии ОС Windows распознают только слеш "\" в качестве разделителя.

Python решает эту проблему благодаря функции `os.path.join ()`.

Давайте перепишем вариант из примера в предыдущем пункте, используя эту функцию:

```
os.path.exists(os.path.join('sample_data',
                            'README.md'))
```

```
[7]  os.path.exists(os.path.join('sample_data', 'README.md'))

     True
```

Создание директории

Создадим директорию с именем `test_dir` внутри текущей директории. Для этого можно использовать функцию `os.mkdir()`:

```
os.mkdir('test_dir')
```

Давайте посмотрим, как это работает на практике.

```
[9]  print(f"test_dir existing: {os.path.exists('test_dir')}")
     os.mkdir('test_dir')
     print(f"test_dir existing: {os.path.exists('test_dir')}")

     test_dir existing: False
     test_dir existing: True
```

Если же мы попытаемся создать каталог, который уже существует, то получим исключение.

```
[11] os.mkdir('test_dir')

---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-11-48d7b58469aa> in <module>()
----> 1 os.mkdir('test_dir')

FileExistsError: [Errno 17] File exists: 'test_dir'
```

Именно поэтому рекомендуется всегда проверять наличие каталога с определенным названием перед созданием нового:

```
if not os.path.exists('test_dir'):
    os.mkdir('test_dir')
```

Еще один совет по созданию каталогов. Иногда нам нужно создать подкаталоги с уровнем вложенности 2 или более. Если мы все еще используем `os.mkdir ()`, нам нужно будет сделать это несколько раз.

В этом случае мы можем использовать `os.makedirs()`. Эта функция создаст все промежуточные каталоги:

```
os.makedirs(os.path.join('test_dir', 'level_1',
                                     'level_2', 'level_3'))
```

Вот что получается в результате.

Показываем содержимое директории

Еще одна полезная команда — `os.listdir()`. Она показывает все содержимое каталога.

Команда отличается от `os.walk ()`, где последний рекурсивно показывает все, что находится «под» каталогом. `os.listdir ()` намного проще в использовании, потому что просто возвращает список содержимого:

```
os.listdir('sample_data')
```

```
[13] os.listdir('sample_data')

['README.md',
 'anscombe.json',
 'california_housing_train.csv',
 'california_housing_test.csv',
 'mnist_train_small.csv',
 'mnist_test.csv']
```

В некоторых случаях нужно что-то более продвинутое — например, поиск всех CSV-файлов в каталоге «sample_data». В этом случае самый простой способ — использовать встроенную библиотеку glob (см. ниже):

```
from glob import
globlist(glob(os.path.join('sample_data', '*.csv')))
```

```
[14] from glob import glob

[15] list(glob(os.path.join('sample_data', '*.csv')))

['sample_data/california_housing_train.csv',
 'sample_data/california_housing_test.csv',
 'sample_data/mnist_train_small.csv',
 'sample_data/mnist_test.csv']
```

Перемещение файлов

Самое время попробовать переместить файлы из одной папки в другую. Рекомендованный способ — еще одна встроенная библиотека `shutil` (см. ниже).

Переместим все CSV-файлы из директории «sample_data» в директорию «test_dir»:

```
import shutil for file in
list(glob(os.path.join('sample_data', '*.csv'))):
    shutil.move(file, 'test_dir')
```

Кстати, есть два способа выполнить задуманное. Например, мы можем использовать библиотеку OS, если не хочется импортировать дополнительные библиотеки. Как `os.rename`, так и `os.replace` подходят для решения задачи.

Но обе они недостаточно «умные», чтобы позволить перемесить файлы в каталог.

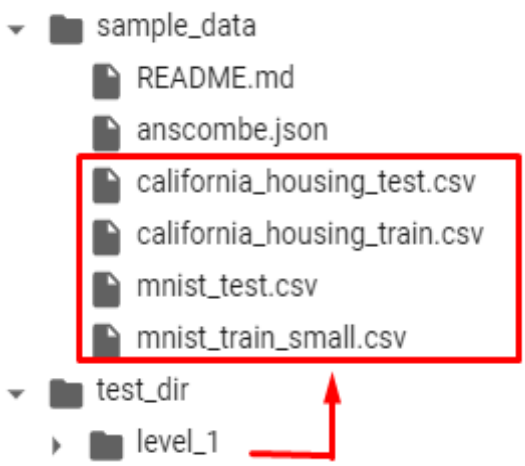
```
[21] for file in list(glob(os.path.join('test_dir', '*.csv'))):
         os.rename(file, 'sample_data')

---------------------------------------------------------------------------
IsADirectoryError                         Traceback (most recent call last)
<ipython-input-21-9675c94bbcdd> in <module>()
      1 for file in list(glob(os.path.join('test_dir', '*.csv'))):
----> 2     os.rename(file, 'sample_data')

IsADirectoryError: [Errno 21] Is a directory: 'test_dir/california_housing_train.csv' -> 'sample_data'
```

Чтобы все это работало, нужно явно указать имя файла в месте назначения. Ниже — код, который это позволяет сделать:

```
for file in list(glob(os.path.join('test_dir',
                                   '*.csv'))):
    os.rename(
        file,
        os.path.join(
            'sample_data',
            os.path.basename(file)
    ))
```



Здесь функция `os.path.basename()` предназначена для извлечения имени файла из пути с любым количеством компонентов.

Другая функция, `os.replace()`, делает то же самое. Но разница в том, что `os.replace()` не зависит от платформы, тогда как `os.rename()` будет работать только в системе Unix / Linux.

Еще один минус — в том, что обе функции не поддерживают перемещение файлов из разных файловых систем, в отличие от `shutil`.

Поэтому рекомендуется использовать `shutil.move()` для перемещения файлов.

Копирование файлов

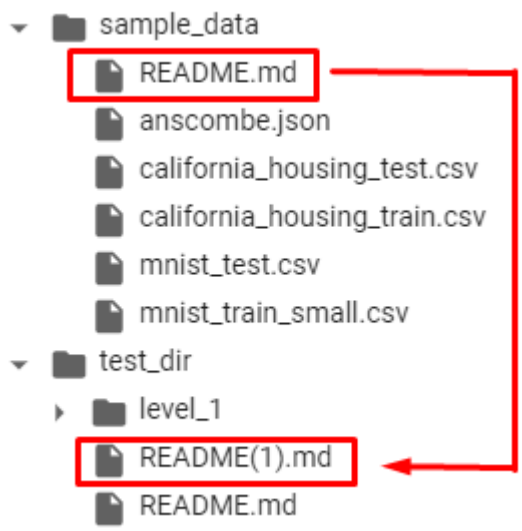
Аналогичным образом shutil подходит и для копирования файлов по уже упомянутым причинам.

Если нужно скопировать файл README.md из папки «sample_data» в папку «test_dir», поможет функция `shutil.copy()`:

```
shutil.copy(
    os.path.join('sample_data', 'README.md'),
    os.path.join('test_dir')
)
```

```
[26] shutil.copy(
         os.path.join('sample_data', 'README.md'),
         os.path.join('test_dir', 'README(1).md')
     )

     'test_dir/README(1).md'
```



Удаление файлов и папок

Когда нужно удалить файл, нужно воспользоваться командой `os.remove()`:

```
os.remove(os.path.join('test_dir', 'README(1).md'))
```

Если требуется удалить каталог, то используем `os.rmdir()`:

```
os.rmdir(os.path.join('test_dir', 'level_1', 'level_2',
                                              'level_3'))
```

Однако `os.rmdir()` может удалить только пустой каталог.

На приведенном выше рис. видим, что удалить можно лишь каталог level_3.

Что если необходимо рекурсивно удалить каталог level_1? В этом случае используем `shutil`.

```
[28] os.rmdir(os.path.join('test_dir', 'level_1'))

---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-28-352a126b9aab> in <module>()
----> 1 os.rmdir(os.path.join('test_dir', 'level_1'))

OSError: [Errno 39] Directory not empty: 'test_dir/level_1'
```

Функция `shutil.rmtree()` сделает все, что нужно:

```
shutil.rmtree(os.path.join('test_dir', 'level_1'))
```

Пользоваться ею нужно с осторожностью, поскольку она безвозвратно удаляет все содержимое каталога.

III. Модуль shutil

Модуль `shutil` содержит набор функций высокого уровня для обработки файлов, групп файлов, и папок. В частности, доступные здесь функции позволяют копировать, перемещать и удалять файлы и папки. Часто используется вместе с модулем os.

Операции над файлами и директориями

`shutil.copyfileobj``(fsrc, fdst[, length])`
- скопировать содержимое одного файлового объекта (`fsrc`) в другой (`fdst`).

Необязательный параметр `length` - размер буфера при копировании (чтобы весь, возможно огромный, файл не читался целиком в память).

При этом, если позиция указателя в `fsrc` не 0 (т.е. до этого было сделано что-то наподобие `fsrc.read(47)`), то будет копироваться содержимое начиная с текущей позиции, а не с начала файла.

`shutil.copyfile`(src, dst, follow_symlinks=True)
- копирует содержимое (но не метаданные) файла `src` в файл `dst`.

Возвращает `dst` (т.е. куда файл был скопирован). `src` и `dst` это строки - пути к файлам. `dst` должен быть полным именем файла.

Если `src` и `dst` представляют собой один и тот же файл, исключение **`shutil.SameFileError`**.

Если `dst` существует, то он будет перезаписан.

Если `follow_symlinks=False` и `src` является ссылкой на файл, то будет создана новая символическая ссылка вместо копирования файла, на который эта символическая ссылка указывает.

shutil.copymode(src, dst, follow_symlinks=True)

- копирует права доступа из `src` в `dst`. Содержимое файла, владелец, и группа не меняются.

shutil.copystat(src, dst, follow_symlinks=True)

- копирует права доступа, время последнего доступа, последнего изменения, и флаги `src` в `dst`. Содержимое файла, владелец, и группа не меняются.

shutil.copy(src, dst, follow_symlinks=True) - копирует содержимое файла `src` в файл или папку `dst`. Если `dst` является директорией, файл будет скопирован с тем же названием, что было в `src`. Функция возвращает путь к местонахождению нового скопированного файла.

Если `follow_symlinks=False`, и `src` это ссылка, `dst` будет ссылкой.

Если `follow_symlinks=True`, и `src` это ссылка, `dst` будет копией файла, на который ссылается `src`

copy() копирует содержимое файла, и права доступа.

shutil.copy2(src, dst, follow_symlinks=True)

- как `copy()`, но пытается копировать все метаданные.

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

- рекурсивно копирует всё дерево директорий с корнем в `src`, возвращает директорию назначения.

Директория `dst` не должна существовать. Она будет создана, вместе с пропущенными родительскими директориями.

Права и времена у директорий копируются `copystat()`, файлы копируются с помощью функции `copy_function` (по умолчанию `shutil.copy2()`).

Если `symlinks=True`, ссылки в дереве `src` будут ссылками в `dst`, и метаданные будут скопированы настолько, насколько это возможно.

Если `False` (по умолчанию), будут скопированы содержимое и метаданные файлов, на которые указывали ссылки.

Если `symlinks=False`, если файл, на который указывает ссылка, не существует, будет добавлено исключение в список ошибок, в исключении `shutil.Error` в конце копирования.

Можно установить флаг `ignore_dangling_symlinks=True`, чтобы скрыть данную ошибку.

Если `ignore` не `None`, то это должна быть функция, принимающая в качестве аргументов имя директории, в которой сейчас `copytree()`, и

список содержимого, возвращаемый `os.listdir()`. Т.к. `copytree()` вызывается рекурсивно, `ignore` вызывается 1 раз для каждой поддиректории. Она должна возвращать список объектов относительно текущего имени директории (т.е. подмножество элементов во втором аргументе). Эти объекты не будут скопированы.

`shutil.ignore_patterns`(*patterns)

- функция, которая создаёт функцию, которая может быть использована в качестве `ignore` для `copytree()`, игнорируя файлы и директории, которые соответствуют glob-style шаблонам.

Например:

```
copytree(source, destination,
          ignore=ignore_patterns('*.pyc', 'tmp*'))
# Скопирует все файлы, кроме заканчивающихся
                      на .pyc или начинающихся с tmp
```

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

- Удаляет текущую директорию и все поддиректории; `path` должен указывать на директорию, а не на символическую ссылку.

Если `ignore_errors=True`, то ошибки, возникающие в результате неудавшегося удаления, будут проигнорированы. Если `False` (по умолчанию), эти ошибки будут передаваться обработчику `onerror`, или, если его нет, то исключение.

На ОС, которые поддерживают функции на основе файловых дескрипторов, по умолчанию используется версия `rmtree()`, не уязвимая к атакам на символические ссылки.

На других платформах это не так: при подобранном времени и обстоятельствах "хакер" может, манипулируя ссылками, удалить файлы, которые недоступны ему в других обстоятельствах.

Чтобы проверить, уязвима ли система к подобным атакам, можно использовать атрибут `rmtree.avoids_symlink_attacks`.

Если задан `onerror`, это должна быть функция с 3 параметрами: `function`, `path`, `excinfo`.

Первый параметр, `function`, это функция, которая создала исключение; она зависит от платформы и интерпретатора. Второй параметр, `path`, это путь, передаваемый функции. Третий параметр, `excinfo` - это информация об исключении, возвращаемая `sys.exc_info()`. Исключения, вызванные `onerror`, не обрабатываются.

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

- рекурсивно перемещает файл или директорию (`src`) в другое место (`dst`), и возвращает место назначения.

Если `dst` - существующая директория, то `src` перемещается внутрь директории. Если `dst` существует, но не директория, то оно может быть перезаписано.

`shutil.disk_usage`(path)

- возвращает статистику использования дискового пространства как namedtuple с атрибутами total, used и free, в байтах.

`shutil.chown`(path, user=None, group=None)

- меняет владельца и/или группу у файла или директории.

`shutil.which`(cmd, mode=os.F_OK | os.X_OK, path=None)

- возвращает путь к исполняемому файлу по заданной команде.

Если нет соответствия ни с одним файлом, то None. `mode` это права доступа, требующиеся от файла, по умолчанию ищет только исполняемые.

Архивация

Высокоуровневые функции для созданиия и чтения архивированных и сжатых файлов. Основаны на функциях из модулей `zipfile` и `tarfile`.

`shutil.make_archive`(base_name, format[, root_dir[,
base_dir[, verbose[, dry_run[,
owner[, group[, logger]]]]]]])

- создаёт архив и возвращает его имя.

`base_name` это имя файла для создания, включая путь, но не включая расширения (не нужно писать ".zip" и т.д.).

`format` - формат архива.

`root_dir` - директория (относительно текущей), которую мы архивируем.

`base_dir` - директория, в которую будет архивироваться (т.е. все файлы в архиве будут в данной папке).

Если `dry_run=True`, архив не будет создан, но операции, которые должны были быть выполнены, запишутся в `logger`.

`owner` и `group` используются при создании tar-архива.

`shutil.get_archive_formats`()

- список доступных форматов для архивирования.

```
>>>
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"),
('gztar', "gzip'ed tar-file"),
('tar', 'uncompressed tar file'),
('xztar', "xz'ed tar-file"),
('zip', 'ZIP file')]
```

shutil.unpack_archive(filename[, extract_dir[, format]])

- распаковывает архив. `filename` - полный путь к архиву.

`extract_dir` - то, куда будет извлекаться содержимое (по умолчанию в текущую).

`format` - формат архива (по умолчанию пытается угадать по расширению файла).

shutil.get_unpack_formats() - список доступных форматов для разархивирования.

Запрос размера терминала вывода

shutil.get_terminal_size(fallback=(columns, lines))

- возвращает размер окна терминала.

`fallback` вернётся, если не удалось узнать размер терминала (терминал не поддерживает такие запросы, или программа работает без терминала). По умолчанию (80, 24).

```
>>>
>>> shutil.get_terminal_size()
os.terminal_size(columns=102, lines=29)
>>> shutil.get_terminal_size()  # Уменьшили окно
os.terminal_size(columns=67, lines=17)
```

IV. Примеры использования модуля shutil.

Копирование файла

Функция `shutil.copyfile()` копирует содержимое источника в место назначения и вызывает исключение `IOError`, если у него нет разрешения на запись в файл назначения.

```
>>> import shutil, os
>>> from glob import glob
# создадим временную директорию
>>> os.mkdir('example')
# создадим тестовый файл
>>> open('example/test_file.txt', 'w').close()
# копирование
>>> shutil.copyfile('example/test_file.txt',
'example/test_file.txt.copy')
# 'example/test-file.txt.copy'

# смотрим результат
>>> pprint.pprint(glob('example/*'))
# ['example/test_file.txt.copy',
'example/test_file.txt']
```

```
# удаляем
>>> shutil.rmtree('example')
```

Функция `shutil.copy()` интерпретирует имя выходного файла как инструмент командной строки Unix `cp`. *Если путь назначения указан как каталог, а не файл, то в каталоге создается новый файл с использованием его базового имени.*

```
>>> import shutil, os
>>> from glob import glob
# создадим тестовый файл
>>> open('shutil_copy.txt', 'w').close()
# создадим временную директорию
>>> os.mkdir('example')
>>> glob('example/*')
# []

# копируем
>>> shutil.copy('shutil_copy.txt', 'example')
# 'example/shutil_copy.txt'

# смотрим результат
>>> glob('example/*')
# ['example/shutil_copy.txt']

# удаляем
>>> shutil.rmtree('example')
```

Рекурсивное копирование каталога

Функция `shutil.copytree()` рекурсивно копирует весь каталог.

```
# для того что бы протестировать функцию copytree()
# создадим иерархию каталогов
import pathlib, itertools, glob, shutil
path = 'test/'
tmp = {'script':'.py', 'text':'.txt'}
for d, ext in tmp.items():
    comb = itertools.combinations([d, 1, 0], r=2)
    for a, b in comb:
        name = f'{a}{b}{ext}'
        pathlib.Path(path, d).mkdir(parents=True,
exist_ok=True)
        pathlib.Path(path, d,
name).touch(exist_ok=True)
```

```
f = glob.glob('test/**/*', recursive=True)
print(f)
# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']

# копируем
shutil.copytree('test', 'test_cp')
# смотрим результат
f_cp = glob.glob('test_cp/**/*', recursive=True)
print(f_cp)

# ['test/text', 'test/script', 'test/text/10.txt',
# 'test/text/text1.txt', 'test/text/text0.txt',
# 'test/script/10.py', 'test/script/script0.py',
# 'test/script/script1.py']

# удаляем
shutil.rmtree('test_cp')
shutil.rmtree('test')
```

Выборочное рекурсивное копирование файлов каталога

Пример, который использует помощник `shutil.ignore_patterns()`.

Здесь копируется все, кроме файлов `.pyc` и файлов или каталогов, чье имя начинается с `tmp`.

```
from shutil import copytree, ignore_patterns
copytree(source, destination,
             ignore=ignore_patterns('*.pyc', 'tmp*'))
```

Другой пример, который использует аргумент ignore для добавления вызова логирования копирования файлов:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

Рекурсивное удаление каталога

Работа функции `shutil.rmtree()`, которая выполняет рекурсивное удаление каталога, демонстрировалась выше. Разберем ситуации посложнее.

В этом примере показано, как удалить дерево каталогов в Windows, где для некоторых файлов установлен бит только для чтения.

Он использует обратный вызов `onerror`, чтобы очистить бит `readonly` и повторить попытку удаления.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

В функции `shutil.rmtree()`, так же можно использовать помощник `shutil.ignore_patterns()` для **выборочного рекурсивного удаления файлов** как это делается в выборочном рекурсивном копировании файлов.

Пример реализации функции `shutil.copytree()`

Этот пример является реализацией функции `shutil.copytree()`, с опущенной строкой документации. Он демонстрирует многие другие функции, предоставляемые этим модулем.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
```

```
                # XXX What about devices, sockets etc.?
            except OSError as why:
                errors.append((srcname, dstname, str(why)))
            # catch the Error from the recursive copytree so that
we can
            # continue with other files
            except Error as err:
                errors.extend(err.args[0])
        try:
            copystat(src, dst)
        except OSError as why:
            # can't copy file access times on Windows
            if why.winerror is None:
                errors.extend((src, dst, str(why)))
        if errors:
            raise Error(errors)
```

Архивирование каталогов

В этом примере создадим архив tar-файлов с использованием `gzip`, содержащий все файлы, найденные в каталоге `.ssh` пользователя:

```
from shutil import make_archive
import os
archive_name = os.path.expanduser(os.path.join('~',
                                               'myarchive'))
root_dir = os.path.expanduser(os.path.join('~',
                                           '.ssh'))
make_archive(archive_name, 'gztar', root_dir)

'/Users/docs-python/myarchive.tar.gz'
```

Полученный архив содержит:

```
~$ tar -tzvf /home/docs-python/myarchive.tar.gz

drwx------ docs-python/docs-python     0 2019-12-17 16:57 ./
-rw------- docs-python/docs-python  1554 2019-12-17 16:53 ./known_hosts
-rw-------     docs-python/docs-python         1554     2019-12-17     14:06
./known_hosts.old
-rw------- docs-python/docs-python  1679 2019-09-16 12:02 ./id_rsa
-rw-r--r-- docs-python/docs-python   399 2019-09-16 12:02 ./id_rsa.pub
```

V. Модуль pathlib

Python 3 включает модуль `pathlib` для манипуляции путями файловых систем независимо от операционной системы.

`pathlib` похож на модуль `os.path`, но `pathlib` предлагает более развитый и удобный интерфейс по сравнению с `os.path`.

Обычно идентифицируют файлы на компьютере с помощью иерархических путей.

Например, можно идентифицировать файл `wave.txt` на компьютере с помощью этого пути: `/Users/sammy/ocean/wave.txt`.

Операционные системы представляют пути несколько по-разному. Windows может представлять путь к файлу `wave.txt` как `C:\Users\sammy\ocean\wave.txt`.

Модуль `pathlib` может быть полезен, если в программе Python вы создаете или перемещаете файлы в файловой системе, указывая все файлы в файловой системе, совпадающие с данным расширением или шаблоном, или создаете пути файла, соответствующие файловой системе на основе наборов неформатированных строк.

Хотя вы можете использовать другие инструменты, например модуль `os.path`, для выполнения большей части этих задач, но модуль `pathlib` позволяет выполнять эти операции с большей степенью читаемости и минимальным количеством кодов.

Рассмотрим некоторые способы использования модуля `pathlib` для представления и манипуляции путями файловых систем.

Модуль `pathlib` предоставляет несколько классов, но одним из наиболее важных является класс `Path`.

Экземпляры класса `Path` представляют путь к файлу или каталогу в файловой системе вашего компьютера.

Например, следующий код получает экземпляр `Path`, который представляет часть пути к файлу `wave.txt`:

```
from pathlib import Path

wave = Path("ocean", "wave.txt")
print(wave)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/wave.txt
```

`from pathlib import Path` делает класс `Path` доступным для нашей программы.

Затем `Path("ocean", "wave.txt")` получает новый экземпляр `Path`.

Из вывода результата видно, что Python «добавил» соответствующий разделитель оперативной системы / между двумя заданными нами компонентами пути "ocean" и "wave.txt".

Примечание. В зависимости от операционной системы вывод может немного отличаться от примеров, приведенных в данном руководстве. В Windows, например, вывод для этого примера может выглядеть как ocean\wave.txt.

В примере объект Path, присвоенный переменной wave, содержит ***относительный путь***.

Можно использовать Path.home() для получения абсолютного пути к домашнему каталогу текущего пользователя:

```
home = Path.home()
wave_absolute = Path(home, "ocean", "wave.txt")
print(home)
print(wave_absolute)
```

Если запустить этот код, результат будет выглядеть приблизительно следующим образом:

```
Output
/Users/sammy
/Users/sammy/ocean/wave.txt
```

Примечание. Как упоминалось ранее, вывод будет зависеть от операционной системы

Path.home() возвращает экземпляр Path с абсолютным путем в домашний каталог текущего пользователя.

Затем мы передадим этот экземпляр Path и строки "ocean" и "wave.txt" в другой конструктор Path, чтобы создать абсолютный путь к файлу wave.txt.

Вывод показывает, что первая строка — это домашний каталог, а вторая строка — домашний каталог плюс ocean/wave.txt.

Этот пример также иллюстрирует важную функцию класса Path: конструктор Path принимает обе строки и ранее существовавшие объекты Path.

Давайте более детально рассмотрим поддержку строк и объектов Path в конструкторе Path:

```
shark = Path(Path.home(), "ocean", "animals",
                          Path("fish", "shark.txt"))
print(shark)
```

Если запустить этот код Python, результат будет выглядеть следующим образом:

```
Output
```

```
/Users/sammy/ocean/animals/fish/shark.txt
```

`shark` — это `Path` к файлу, который мы создали с помощью объектов `Path` (`Path.home()` и `Path("fish", "shark.txt")` и строк `"ocean"` и `"animals"`).

Конструктор `Path` интеллектуально обрабатывает оба типа объектов и аккуратно соединяет их с помощью соответствующего разделителя операционной системы, в данном случае `/`.

Доступ к атрибутам файла

Рассмотрим, как можно использовать экземпляры `Path` для доступа к информации о файле.

Мы можем использовать атрибуты `name` и `suffix` для доступа к именам и расширениям файлов:

```
wave = Path("ocean", "wave.txt")
print(wave)
print(wave.name)
print(wave.suffix)
```

Запустив этот код и получим вывод, аналогичный следующему:

```
Output
/Users/sammy/ocean/wave.txt
wave.txt
.txt
```

Этот вывод показывает, что имя файла в конце нашего пути — `wave.txt`, а расширение файла — `.txt`.

Экземпляры `Path` также предлагают функцию `with_name`, позволяющую беспрепятственно создавать новый объект `Path` с другим именем:

```
wave = Path("ocean", "wave.txt")
tides = wave.with_name("tides.txt")
print(wave)
print(tides)
```

Если запустить его, результат будет выглядеть следующим образом:

```
ocean/wave.txt
ocean/tides.txt
```

Код сначала создает экземпляр `Path`, указывающий на файл с именем `wave.txt`.

Затем вызывается метод `with_name` в `wave`, чтобы вернуть второй экземпляр `Path`, указывающий на новый файл с именем `tides.txt`.

Часть каталога `ocean/` остается неизмененной и оставляет финальный путь в виде `ocean/tides.txt`

Доступ к предшествующим объектам

Иногда полезно получить доступ к каталогам, содержащим определенный путь. Давайте рассмотрим пример:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/animals/fish/shark.txt
ocean/animals/fish
```

Атрибут **`parent`** в экземпляре `Path` возвращает ближайшего предшественника пути данного файла. В этом случае он возвращает каталог с файлом `shark.txt`: `ocean/animals/fish`.

Можем получать доступ к атрибуту `parent` несколько раз в команде, чтобы пройти вверх по корневому дереву данного файла:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
print(shark)
print(shark.parent.parent)
```

Если выполнить этот код, то увидим следующие:

```
Output
ocean/animals/fish/shark.txt
ocean/animals
```

Вывод будет похож на предыдущий вывод, но теперь мы перешли на уровень выше, получив доступ к `.parent` во второй раз. Два каталога от `shark.txt` — это каталог `ocean/animals`.

Использование шаблона поиска для списка файлов

Также можно использовать класс `Path` для списка файлов с помощью метода **`glob`**.

Допустим, у нас есть структура каталога, которая выглядит следующим образом:

```
└── ocean
    ├── animals
```

```
│   └── fish
│       └── shark.txt
├── tides.txt
└── wave.txt
```

Каталог `ocean` содержит файлы `tides.txt` и `wave.txt`. У нас есть файл с именем `shark.txt`, вложенный в каталог `ocean`, каталог `animals` и каталог `fish`: `ocean/animals/fish`.

Чтобы перечислить все файлы `.txt` в каталоге `ocean`, можно использовать:

```
for txt_path in Path("ocean").glob("*.txt"):
    print(txt_path)
```

Этот код произведет следующий вывод:

```
Output
ocean/wave.txt
ocean/tides.txt
```

Шаблон поиска "*.txt" находит все файлы, заканчивающиеся на .txt. Поскольку пример кода выполняет этот поиск в каталоге `ocean`, он возвращает два файла `.txt` в каталоге `ocean`: `wave.txt` и `tides.txt`.

Также можно использовать *метод `glob` рекурсивно*.

Чтобы перечислить все файлы `.txt` в каталоге `ocean` и все его подкаталоги, запишем:

```
for txt_path in Path("ocean").glob("**/*.txt"):
    print(txt_path)
```

Если запустить этот код, результат будет выглядеть следующим образом:

```
Output
ocean/wave.txt
ocean/tides.txt
ocean/animals/fish/shark.txt
```

Часть `**` шаблона поиска будет соответствовать этому каталогу и всем каталогам под ним рекурсивно.

Поэтому в выводе у нас будут не только файлы `wave.txt` и `tides.txt`, но также мы получим файл `shark.txt`, вложенный в `ocean/animals/fish`.

Вычисление относительных путей

Можно использовать метод `Path.relative_to` для вычисления путей, относящихся друг к другу. Метод `relative_to` полезен, если, например, вы хотите получить часть длинного пути файла.

Рассмотрите следующий код:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
below_ocean = shark.relative_to(Path("ocean"))
below_animals = shark.relative_to(Path("ocean",
                                       "animals"))
print(shark)
print(below_ocean)
print(below_animals)
```

Если запустить его, результат будет выглядеть следующим образом:

```
Output
ocean/animals/fish/shark.txt
animals/fish/shark.txt
fish/shark.txt
```

Метод `relative_to` возвращает новый объект `Path`, относящийся к данному аргументу. В нашем примере мы вычислим `Path` к `shark.txt`, относящийся к каталогу `ocean`, а затем относящийся к обоим каталогам `ocean` и `animals`.

Если `relative_to` не сможет вычислить ответ, поскольку мы даем ему не связанный путь, он выдаст `ValueError`:

```
shark = Path("ocean", "animals", "fish", "shark.txt")
shark.relative_to(Path("unrelated", "path"))
```

Мы получим исключение `ValueError`, возникшее из этого кода, которое будет выглядеть следующим образом:

```
Output
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/Python3.8/pathlib.py", line 899, in
                                                    relative_to
    raise ValueError("{!r} does not start with {!r}"
ValueError: 'ocean/animals/fish/shark.txt' does not start with
                                                  'unrelated/path'
```

`unrelated/path` не является частью `ocean/animals/fish/shark.txt`, поэтому Python не сможет вычислить относительный путь.

Модуль `pathlib` представляет дополнительные классы и утилиты, Можно воспользоваться ссылкой на документацию модуля `pathlib` для получения дополнительной информации.

VI. Модуль glob

Модуль glob находит все пути, совпадающие с заданным шаблоном в соответствии с правилами, используемыми оболочкой Unix.

Обрабатываются символы "*" (произвольное количество символов), "?" (один символ), и диапазоны символов с помощью [].

Для использования тильды "~" и переменных окружения необходимо использовать `os.path.expanduser()` и `os.path.expandvars()`.

Для поиска спецсимволов, заключайте их в квадратные скобки. Например, [?] соответствует символу "?".

`glob.glob`(pathname)

- возвращение список (возможно, пустой) путей, соответствующих шаблону pathname. Путь может быть как абсолютным (например, /usr/src/Python-1.5/Makefile) или относительный (как ../../Tools/*/*.gif).

`glob.iglob`(pathname)

- возвращает итератор, дающий те же значения, что и glob.glob.

`glob.escape`(pathname)

- экранирует все специальные символы для glob ("?", "*" и "["). Специальные символы в имени диска не экранируются (так как они там не учитываются), то есть в Windows escape("//?/c:/Quo vadis?.txt") возвращает "//?/c:/Quo vadis[?].txt".

Рассмотрим, например, каталог, содержащий только следующие файлы: 1.gif, 2.txt и card.gif.

glob.glob() вернёт следующие результаты. (обратите внимание, что любые ведущие компоненты пути сохраняются):

```
>>>
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

Если каталог содержит файлы, начинающиеся с ".", они не будут включаться по умолчанию. Рассмотрим, например, каталог, содержащий card.gif и .card.gif:

```
>>>
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

VII. Модуль os.path

os.path является вложенным модулем в модуль os, и реализует некоторые полезные функции на работы с путями.

os.path.abspath(path) - возвращает нормализованный абсолютный путь.

os.path.basename(path) - базовое имя пути (эквивалентно os.path.split(path)[1]).

os.path.commonprefix(list) - возвращает самый длинный префикс всех путей в списке.

os.path.dirname(path) - возвращает имя директории пути path.

os.path.exists(path) - возвращает True, если path указывает на существующий путь или дескриптор открытого файла.

os.path.expanduser(path) - заменяет ~ или ~user на домашнюю директорию пользователя.

os.path.expandvars(path) - возвращает аргумент с подставленными переменными окружения ($name или ${name} заменяются переменной окружения name). Несуществующие имена не заменяет. На Windows также заменяет %name%.

os.path.getatime(path) - время последнего доступа к файлу, в секундах.

os.path.getmtime(path) - время последнего изменения файла, в секундах.

os.path.getctime(path) - время создания файла (Windows), время последнего изменения файла (Unix).

os.path.getsize(path) - размер файла в байтах.

os.path.isabs(path) - является ли путь абсолютным.

os.path.isfile(path) - является ли путь файлом.

os.path.isdir(path) - является ли путь директорией.

os.path.islink(path) - является ли путь символической ссылкой.

os.path.ismount(path) - является ли путь точкой монтирования.

os.path.join(path1[, path2[, ...]]) - соединяет пути с учётом особенностей операционной системы.

os.path.normcase(path) - нормализует регистр пути (на файловых системах, не учитывающих регистр, приводит путь к нижнему регистру).

os.path.normpath(path) - нормализует путь, убирая избыточные разделители и ссылки на предыдущие директории. На Windows преобразует прямые слеши в обратные.

os.path.realpath(path) - возвращает канонический путь, убирая все символические ссылки (если они поддерживаются).

os.path.relpath(path, start=None) - вычисляет путь относительно директории start (по умолчанию - относительно текущей директории).

os.path.samefile(path1, path2) - указывают ли path1 и path2 на один и тот же файл или директорию.

os.path.sameopenfile(fp1, fp2) - указывают ли дескрипторы fp1 и fp2 на один и тот же открытый файл.

os.path.split(path) - разбивает путь на кортеж (голова, хвост), где хвост - последний компонент пути, а голова - всё остальное. Хвост никогда не начинается со слеша (если путь заканчивается слешем, то хвост пустой). Если слешей в пути нет, то пустой будет голова.

os.path.splitdrive(path) - разбивает путь на пару (привод, хвост).

os.path.splitext(path) - разбивает путь на пару (root, ext), где ext начинается с точки и содержит не более одной точки.

os.path.supports_unicode_filenames - поддерживает ли файловая система Unicode.