Содержание

1 Введение **2**

2 Основная часть **2**

2.1 Копирование деревьев каталогов 2

2.2 Сравнение деревьев каталогов 6

2.3 Поиск различий между деревьями 8

2.4 Поиск в деревьях каталогов 16

2.5 Обход каталогов 20

3 Заключение **29**

4 Список литературы **29**

1 Введение

В рамках данной курсовой работы мной был разработан программный модуль на языке Python, предоставляющий возможности работы с файловой системой. Модуль включает в себя 4 утилиты:

- Копирование деревьев каталогов
- Сравнение деревьев каталогов
- Поиск в деревьях каталогов
- Обход каталогов

Этот модуль относится к области системных приложений – сценариев, работающих с файлами, программами и окружением программ в целом. Исторически такие программы писались на непереносимых и синтаксически неочевидных языках оболочек, таких как командные файлы DOS, csh и awk. Однако в этой области ярко проявляются лучшие свойства Python. Это будет продемонстрировано на нескольких примерах.

2 Основная часть

2.1 Копирование деревьев каталогов

Копирование в Windows путем перетаскивания мышью обладает рядом недостатков: копирование прерывается, как только будет обнаружен первый испорченный файл или как только встречался файл со слишком длинным или необычным именем. Сценарий cpall.py, представленный ниже реализует один из возможных способов копирования. С его помощью можно управлять действиями, которые выполняются при обнаружении проблемных файлов, например, пропустить файл с помощью обработчика исключения. Кроме того, этот инструмент работает и на других платформах с тем же интерфейсом и таким же результатом.

```
# Порядок использования: “python cpall.py dirFrom dirTo”.
# Рекурсивно копирует дерево каталогов. Действует подобно команде Unix “cp -r
# dirFrom/* dirTo”, предполагая, что оба аргумента dirFrom и dirTo являются
# именами каталогов.
# Был написан с целью обойти фатальные ошибки при копировании файлов
# перетаскиванием мышью в Windows (когда встреча первого же проблемного файла
# вызывает прекращение операции копирования) и обеспечить возможность реализации
# более специализированных операций копирования на языке Python.

import os, sys
maxfileload = 1000000
blksize = 1024 * 500

def copyfile(pathFrom, pathTo, maxfileload=maxfileload):
    """
    Копирует один файл из pathFrom в pathTo, байт в байт;
```

```
        используется двоичный режим для подавления операций
        кодирования/декодирования и преобразований символов конца строки
    """
    if os.path.getsize(pathFrom) <= maxfileload:
        bytesFrom = open(pathFrom, 'rb').read()  # маленький файл читать целиком
        open(pathTo, 'wb').write(bytesFrom)
    else:
        fileFrom = open(pathFrom, 'rb')    # большие файлы - по частям
        fileTo   = open(pathTo,   'wb')    # режим b для обоих файлов
        while True:
        # прочитать очередной блок
            bytesFrom = fileFrom.read(blksize)
            if not bytesFrom: break        # пустой после последнего блока
            fileTo.write(bytesFrom)

def copytree(dirFrom, dirTo, verbose=0):
    """
    Копирует содержимое dirFrom и вложенных подкаталогов в dirTo,
        возвращает счетчики (files, dirs);
        для представления имен каталогов, недекодируемых на других платформах,
        может потребоваться использовать переменные типа bytes;
        в Unix может потребоваться выполнять дополнительные проверки типов файлов,
        чтобы пропускать ссылки, файлы fifo и так далее.
    """
    fcount = dcount = 0
    for filename in os.listdir(dirFrom):             # для файлов/каталогов
        pathFrom = os.path.join(dirFrom, filename)
        pathTo   = os.path.join(dirTo,   filename)  # расширить оба пути
        if not os.path.isdir(pathFrom):              # скопировать простые файлы
            try:
                if verbose > 1: print('copying', pathFrom, 'to', pathTo)
                copyfile(pathFrom, pathTo)
                fcount += 1
            except:
                print('Error copying', pathFrom, 'to', pathTo, '--skipped')
                print(sys.exc_info()[0], sys.exc_info()[1])
        else:
            if verbose: print('copying dir', pathFrom, 'to', pathTo)
            try:
                os.mkdir(pathTo)                     # создать новый подкаталог
                below = copytree(pathFrom, pathTo)  # спуск в подкаталоги
                fcount += below[0]                   # увеличить счетчики
                dcount += below[1]                   # подкаталогов
                dcount += 1
            except:
                print('Error creating', pathTo, '--skipped')
                print(sys.exc_info()[0], sys.exc_info()[1])
```

```
    return (fcount, dcount)

def getargs():
    """
    Извлекает и проверяет аргументы с именами каталогов, по умолчанию
        возвращает None в случае ошибки
    """
    try:
        dirFrom, dirTo = sys.argv[1:]
    except:
        print('Usage error: cpall.py dirFrom dirTo')
    else:
        if not os.path.isdir(dirFrom):
            print('Error: dirFrom is not a directory')
        elif not os.path.exists(dirTo):
            os.mkdir(dirTo)
            print('Note: dirTo was created')
            return (dirFrom, dirTo)
        else:
            print('Warning: dirTo already exists')
            if hasattr(os.path, 'samefile'):
                same = os.path.samefile(dirFrom, dirTo)
            else:
                same = os.path.abspath(dirFrom) == os.path.abspath(dirTo)
            if same:
                print('Error: dirFrom same as dirTo')
            else:
                return (dirFrom, dirTo)

if __name__ == '__main__':
    import time
    dirstuple = getargs()
    if dirstuple:
        print('Copying...')
        start = time.clock()
        fcount, dcount = copytree(*dirstuple)
        print('Copied', fcount, 'files,', dcount, 'directories', end=' ')
        print('in', time.clock() - start, 'seconds')
```

В этом сценарии реализована собственная логика рекурсивного обхода дерева каталогов, в ходе которого запоминаются пути каталогов источника и приемника. На каждом уровне она копирует простые файлы, создает каталоги в целевом пути и производит рекурсивный спуск в подкаталоги с расширением путей «из» и «в» на один уровень. Эту задачу можно запрограммировать и другими способами (например, в процессе обхода можно изменять текущий рабочий каталог с помощью функции os.chdir или использовать решение на основе функции os.walk, замещая пути «из» и «в» по мере их обхода), но на практике вполне достаточно использовать прием расширения имен каталогов при спуске.

Стоит обратить внимание на повторно используемую в этом сценарии функцию copyfile – на тот случай, если потребуется копировать файлы размером в несколько гигабайтов, она, исходя из размера файла, решает, читать ли файл целиком или по частям (при вызове без аргументов метода read файла он загружает весь файл в строку, находящуюся в памяти). Я выбрал достаточно большие размеры для читаемых целиком файлов и для блоков, потому что чем больший объем будет читаться за один подход, тем быстрее будет работать сценарий. Это решение гораздо эффективнее, чем могло бы показаться на первый взгляд, – строки, остающиеся в памяти после последней операции чтения, будут утилизироваться сборщиком мусора, и освободившаяся память будет повторно использована последующими операциями. Здесь используется двоичный режим доступа к файлам, чтобы подавить кодирование/декодирование содержимого файлов и преобразование символов конца строки – в дереве каталогов могут находиться файлы самых разных типов.

Стоит заметить также, что сценарий при необходимости создает целевой каталог, и перед началом копирования предполагает, что он пуст, – следует удалить целевой каталог перед копированием нового дерева с тем же именем, иначе к дереву результата могут присоединиться старые файлы (мы могли бы автоматически очищать целевой каталог перед копированием, но это не всегда бывает желательно). Кроме того, данный сценарий пытается определить – не являются ли исходный и конечный каталоги одним и тем же каталогом. В Unix-подобных системах, где есть такие структуры, как ссылки, функция os.path.samefile проделывает более сложную работу, чем простое сравнение абсолютных имен файлов (разные имена файлов могут означать один и тот же файл). Ниже приводится пример копирования большого дерева примеров книги в Windows. При запуске сценарию необходимо указать имена исходного и целевого каталогов, перенаправить вывод сценария в файл, если возникает слишком много ошибок, чтобы можно было прочитать все сообщения о них сразу (например, > output.Копирование деревьев каталогов 421 txt), и при необходимости выполнить команду оболочки **rm -r** или **rmdir /S** (или аналогичную для соответствующей платформы), чтобы сначала удалить целевой каталог:

```
C:\...\PP4E\System\Filetools> rmdir /S copytemp
copytemp, Are you sure (Y/N)? y
C:\...\PP4E\System\Filetools> cpall.py C:\temp\PP3E\Examples copytemp
Note: dirTo was created
Copying...
Copied 1430 files, 185 directories in 10.4470980971 seconds
C:\...\PP4E\System\Filetools> fc /B copytemp\PP3E\Launcher.py
C:\temp\PP3E\Examples\PP3E\Launcher.py
Comparing files COPYTEMP\PP3E\Launcher.py and C:\TEMP\PP3E\EXAMPLES\PP3E\
LAUNCHER.PY
FC: no differences encountered
```

Можно воспользоваться аргументом verbose функции копирования, чтобы проследить, как протекает процесс копирования. В этом примере за 10 секунд было скопировано дерево каталогов, содержащее 1430 файлов и 185 подкаталогов, – на довольно низкопроизводительном ноутбуке (для получения системного времени была использована встроенная функция time.clock). Каким же образом этот сценарий справляется с проблемными файлами следующим образом: он перехватывает и игнорирует исключения и продолжает обход. Чтобы ско- пировать все хорошие файлы, достаточно выполнить команду такого вида:

```
C:\...\PP4E\System\Filetools> python cpall.py G:\Examples C:\PP3E\Examples
```

Поскольку на моем компьютере, работающем под управлением Windows, привод CD доступен как диск «G:», эта команда эквивалентна копированию путем перетаскивания элемента, находящегося в папке верхнего уровня на компакт-диске, за исключением того, что сценарий Python восстанавливается после возникающих ошибок и копирует остальные файлы. В случае ошибки копирования он выводит сообщение в стандартный поток вывода и продолжает работу. При копировании большого количества файлов, вероятно, будет удобнее перенаправить стандартный вывод сценария в файл, чтобы позднее его можно было детально исследовать. Вообще говоря, сценарию cpall можно передать любой абсолютный путь к каталогу на компьютере, даже такой, который обозначает устройство, например привод CD. Для выполнения сценария в Linux можно обратиться к приводу CD, указав такой каталог, как */dev/cdrom*. После копирования дерева каталогов таким способом можно проверить получившийся результат. Для этого мной была написана следующая утилита. Объединив сценарий cpall, приведенный выше с универсальным инструментом сравнения деревьев, мы получаем переносимый и легко настраиваемый способ копирования и проверки наборов данных.

2.2 Сравнение деревьев каталогов

Рассмотрим, что включает в себя задача сравнить два дерева каталогов. Если оба дерева имеют одинаковую структуру ветвей и глубину, проблема сводится к сравнению соответствующих файлов в каждом дереве. Однако в общем случае деревья могут иметь произвольную различную форму, глубину и так далее.
В более общем случае каталог в одном дереве может содержать больше или меньше элементов, чем соответствующий каталог в другом дереве. Если различие обусловлено наличием других файлов, это означает отсутствие соответствующих файлов для сравнения в другом каталоге. Если различие обусловлено наличием других каталогов, это означает отсутствие соответствующей ветви, в которую нужно войти. На самом деле единственный способ выявить файлы и каталоги, которые есть в одном дереве, но отсутствуют в другом, заключается в том, чтобы выявить различия в каталогах каждого уровня. Иными словами, алгоритм сравнения деревьев должен также попутно выполнять сравнение каталогов. Для начала необходимо реализовать сравнение имен файлов для одного каталога, так как это вложенная и более простая операция.

```
"""
Порядок использования: python dirdiff.py dir1-path dir2-path
Сравнивает два каталога, пытаясь отыскать файлы, присутствующие в одном
и отсутствующие в другом.
Эта версия использует функцию os.listdir и выполняет поиск различий между двумя
списками. Обратите внимание, что сценарий проверяет только имена файлов, но не
их содержимое, - версию, которая сравнивает результаты вызова методов .read(),вы
найдете в сценарии diffall.py.
"""

import os, sys

def reportdiffs(unique1, unique2, dir1, dir2):
    """
```

```
    Генерирует отчет о различиях для одного каталога: часть вывода функции
comparedirs
    """
    if not (unique1 or unique2):
        print('Directory lists are identical')
    else:
        if unique1:
            print('Files unique to', dir1)
            for file in unique1:
                print('...', file)
        if unique2:
            print('Files unique to', dir2)
            for file in unique2:
                print('...', file)

def difference(seq1, seq2):
    """
    Возвращает элементы, присутствующие только в seq1;
Операция set(seq1) - set(seq2) даст аналогичный результат, но множества
являются неупорядоченными коллекциями, поэтому порядок следования
элементов в каталоге будет утерян
    """
    return [item for item in seq1 if item not in seq2]


def comparedirs(dir1, dir2, files1=None, files2=None):
    """
    Сравнивает содержимое каталогов, но не сравнивает содержимое файлов;
функции listdir может потребоваться передавать аргумент типа bytes, если
могут встречаться имена файлов, недекодируемые на других платформах
    """
    print('Comparing', dir1, 'to', dir2)
    files1  = os.listdir(dir1) if files1 is None else files1
    files2  = os.listdir(dir2) if files2 is None else files2
    unique1 = difference(files1, files2)
    unique2 = difference(files2, files1)
    reportdiffs(unique1, unique2, dir1, dir2)
    return not (unique1 or unique2)           # true если нет различий

def getargs():
    "Аргументы при работе в режиме командной строки"
    try:
        dir1, dir2 = sys.argv[1:]             # 2 аргумента командной строки
    except:
        print('Usage: dirdiff.py dir1 dir2')
        sys.exit(1)
    else:
```

```
        return (dir1, dir2)

if __name__ == '__main__':
    dir1, dir2 = getargs()
    comparedirs(dir1, dir2)
```

Получив списки имен для каждого каталога, этот сценарий просто выбирает уникальные имена в первом каталоге, уникальные имена во втором каталоге и сообщает о найденных уникальных именах как о расхождениях (то есть о файлах, имеющихся в одном каталоге, но отсутствующих в другом). Функция *comparedirs* возвращает значение $True$, если расхождения не были обнаружены, что полезно для обнаружения различий при вызове из других программ.
Запустим этот сценарий с несколькими каталогами – он сообщит о найденных различиях, которые представляют уникальные имена в любом из переданных каталогов. При этом сравниваются только структуры путем проверки имен в списках, но не содержимое файлов.

```
C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples copytemp
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
C:\...\PP4E\System\Filetools> dirdiff.py C:\temp\PP3E\Examples\PP3E\System ..
Comparing C:\temp\PP3E\Examples\PP3E\System to ..
Files unique to C:\temp\PP3E\Examples\PP3E\System
... App
... Exits
... Media
... moreplus.py
Files unique to ..
... more.pyc
... spam.txt
... Tester
... __init__.pyc
```

В основе сценария лежит функция $difference$: она реализует простую операцию сравнения списков. Применительно к каталогам, уникальные элементы представляют различия между деревьями, а общие элементы представляют имена файлов или подкаталогов, которые заслуживают дальнейшего сравнения или обхода. В Python 2.4 и более поздних версиях можно было бы использовать встроенные объекты типа set, если порядок следования имен в результатах не имеет значения – множества не являются последовательностями, поэтому они не сохраняют оригинальный порядок следования элементов в списках, полученных с помощью функции os.listdir. По этой причине (и чтобы не вынуждать пользователей модернизировать сценарий) вместо множеств я использовал функцию, опирающуюся на использование выражения-генератора.

2.3 Поиск различий между деревьями

После того как мы реализовали инструмент, отбирающий уникальные имена файлов и каталогов, нам осталось реализовать инструмент обхода дерева, который будет применять

функции из модуля $dir-diff$ на каждом уровне, чтобы отобрать уникальные файлы и каталоги; явно сравнит содержимое общих файлов и обойдет общие каталоги. Эти операции осуществляет следующий сценарий

```
"""
Порядок использования: "python diffall.py dir1 dir2".
Выполняет рекурсивное сравнение каталогов: сообщает об уникальных файлах,
существующих только в одном из двух каталогов, dir1 или dir2; сообщает о файлах
с одинаковыми именами и с разным содержимым, присутствующих в каталогах dir1
и dir2; сообщает об разнотипных элементах с одинаковыми именами, присутствующих
в каталогах dir1 и dir2; то же самое выполняется для всех подкаталогов
с одинаковыми именами, находящихся внутри деревьев каталогов dir1
и dir2. Сводная информация об обнаруженных отличиях помещается в конец вывода,
однако в процессе поиска в вывод добавляется дополнительная информация об
отличающихся и уникальных файлах с метками "DIFF" и "unique". Новое: (в 3
издании) для больших файлов введено ограничение на размер читаемых блоков в 1
Мбайт, (3 издание) обнаруживаются одинаковые имена файлов/каталогов, (4 издание)
исключены лишние вызовы os.listdir() в dirdiff.comparedirs() за счет передачи
результатов.
"""

import os, dirdiff
blocksize = 1024 * 1024          # не более 1 Мбайта на одну операцию чтения

def intersect(seq1, seq2):
    """
    Возвращает все элементы, присутствующие одновременно в seq1 и seq2;
выражение set(seq1) & set(seq2) возвращает тот же результат, но множества
являются неупорядоченными коллекциями, поэтому при их использовании может
быть утерян порядок следования элементов, если он имеет значение для
некоторых платформ
    """
    return [item for item in seq1 if item in seq2]

def comparetrees(dir1, dir2, diffs, verbose=False):
    """
    Сравнивает все подкаталоги и файлы в двух деревьях каталогов;
для предотвращения кодирования/декодирования содержимого и преобразования
символов конца строки использует двоичный режим доступа к файлам,
так как деревья могут содержать произвольные двоичные и текстовые файлы;
функции listdir может потребоваться передавать аргумент типа bytes, если
могут встречаться имена файлов, недекодируемые на других платформах
    """
    # сравнить списки с именами файлов
    print('-' * 20)
    names1 = os.listdir(dir1)
    names2 = os.listdir(dir2)
```

```
    if not dirdiff.comparedirs(dir1, dir2, names1, names2):
        diffs.append('unique files at %s - %s' % (dir1, dir2))

    print('Comparing contents')
    common = intersect(names1, names2)
    missed = common[:]

    # сравнить содержимое файлов с одинаковыми именами
    for name in common:
        path1 = os.path.join(dir1, name)
        path2 = os.path.join(dir2, name)
        if os.path.isfile(path1) and os.path.isfile(path2):
            missed.remove(name)
            file1 = open(path1, 'rb')
            file2 = open(path2, 'rb')
            while True:
                bytes1 = file1.read(blocksize)
                bytes2 = file2.read(blocksize)
                if (not bytes1) and (not bytes2):
                    if verbose: print(name, 'matches')
                    break
                if bytes1 != bytes2:
                    diffs.append('files differ at %s - %s' % (path1, path2))
                    print(name, 'DIFFERS')
                    break

    # рекурсивно сравнить каталоги с одинаковыми именами
    for name in common:
        path1 = os.path.join(dir1, name)
        path2 = os.path.join(dir2, name)
        if os.path.isdir(path1) and os.path.isdir(path2):
            missed.remove(name)
            comparetrees(path1, path2, diffs, verbose)

    # одинаковые имена, но оба не являются одновременно файлами или каталогами?
    for name in missed:
        diffs.append('files missed at %s - %s: %s' % (dir1, dir2, name))
        print(name, 'DIFFERS')


if __name__ == '__main__':
    dir1, dir2 = dirdiff.getargs()
    diffs = []
    comparetrees(dir1, dir2, diffs, True)       # список diffs изменяется в
    print('=' * 40)                             # процессе обхода, вывести diffs
    if not diffs:
        print('No diffs found.')
```

```
    else:
        print('Diffs found:', len(diffs))
        for diff in diffs: print('-', diff)
```

В каждом каталоге этого дерева данный сценарий просто использует модуль *dirdiff*, чтобы обнаружить уникальные имена, а затем сравнивает общие имена, присутствующие одновременно в обоих списках содержимого каталогов. Рекурсивный спуск в подкаталоги выполняется только после сравнения всех файлов на каждом уровне, чтобы вывод сценария было удобнее воспринимать на глаз (трассировка обхода подкаталогов выводится ниже результатов сравнения файлов – они не смешиваются).
Стоит заострить внимание на списке misses. Очень маловероятно, но не невозможно, чтобы одно и то же имя в одном каталоге соответствовало файлу, а в другом – подкаталогу. Кроме того, заслуживает внимания переменная *blocksize*. Как и в сценарии копирования деревьев каталогов, приведенном выше, вместо того чтобы слепо пытаться читать файлы в память целиком, мы установили ограничение в 1 Мбайт для каждой операции чтения – на тот случай, если какие-нибудь файлы окажутся слишком большими, чтобы их можно было загрузить в память. Если бы этого ограничения не было и файлы читались бы целиком, как показано ниже, в некоторых случаях возбуждалось бы исключение:

```
MemoryError:
bytes1 = open(path1, 'rb').read()
bytes2 = open(path2, 'rb').read()
if bytes1 == bytes2: ...
```

Этот код проще, но менее практичен в ситуациях, когда могут встречаться очень большие файлы, не умещающиеся в память целиком (например, файлы образов CD и DVD). Здесь же, файл читается в цикле порциями не более 1 Мбайта, пока не будет возвращена пустая строка, свидетельствующая об окончании файла. Файлы считаются одинаковыми, если совпадают все прочитанные из них блоки и конец файла достигнут одновременно.
Помимо всего прочего, мы обрабатываем содержимое файлов в двоичном режиме, чтобы подавить операцию декодирования их содержимого и предотвратить преобразование символов конца строки, потому что деревья каталогов могут содержать произвольные двоичные и текстовые файлы. На платформах, где имена файлов могут оказаться недекодируемыми (например, с помощью dir1.encode()) необходимо передавать аргумент типа bytes в функцию *os.listdir* . На некоторых платформах может также потребоваться определять и пропускать некоторые файлы специальных типов, чтобы обеспечить полную универсальность, но в моих каталогах такие файлы отсутствовали, поэтому я не включил эту проверку в сценарий. Для каждого подкаталога результаты *os.listdir* собираются и передаются только один раз, чтобы избежать лишних вызовов функций из модуля dirdiff, это помогает выиграть в производительности, и может быть особенно актуально на медленных машинах, где каждый лишний цикл на счету.

Запускаем сценарий

При обработке идентичных деревьев во время обхода выводятся сообщения о состоянии, а в конце появляется сообщение: «No diffs found» (Расхождений не обнаружено):

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples
copytemp > diffs.txt
```

```
C:\...\PP4E\System\Filetools> type diffs.txt | more
--------------------
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Directory lists are identical
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.py matches
Launcher.pyc matches
...более 2000 строк опущено...
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
========================================
No diffs found.
```

При использовании этого сценария я устанавливаю флаг verbose в значение True и перенаправляю вывод в файл (для больших деревьев выводится слишком много информации, которую трудно воспринимать в процессе выполнения сценария). Чтобы ограничить количество сообщений, устанавливайте флаг verbose в значение False. Чтобы посмотреть, как выглядит отчет о расхождениях, нужно их создать. Если теперь изменить несколько файлов в одном из деревьев, или воспользоваться сценарием глобального поиска и замены и удалить несколько файлов, чтобы в процессе поиска можно было обнаружить уникальные элементы. Последние две команды удаления из приведенных ниже воздействуют на один и тот же каталог в разных деревьях:

```
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\README-PP3E.txt
C:\...\PP4E\System\Filetools> notepad copytemp\PP3E\System\Filetools\commands.py
C:\...\PP4E\System\Filetools> notepad C:\temp\PP3E\Examples\PP3E\__init__.py
C:\...\PP4E\System\Filetools> del copytemp\PP3E\System\Filetools\cpall_visitor.py
C:\...\PP4E\System\Filetools> del copytemp\PP3E\Launcher.py
C:\...\PP4E\System\Filetools> del C:\temp\PP3E\Examples\PP3E\PyGadgets.py
```

Теперь перезапустим сценарий сравнения, чтобы обнаружить различия, и перенаправим вывод в файл, чтобы облегчить просмотр результатов. Ниже приведена лишь часть выходного отчета, в которой сообщается о различиях. При обычном использовании я сначала смотрю на сводку в конце отчета, а затем ищу в тексте отчета строки «DIFF» и «unique», если мне нужна дополнительная информация об отличиях, указанных в сводке, – конечно,

этот интерфейс можно было бы сделать более дружественным, но мне вполне хватает и этого:

```
C:\...\PP4E\System\Filetools> diffall.py C:\temp\PP3E\Examples
copytemp > diff2.txt
C:\...\PP4E\System\Filetools> notepad diff2.txt
--------------------
Comparing C:\temp\PP3E\Examples to copytemp
Directory lists are identical
Comparing contents
README-root.txt matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E to copytemp\PP3E
Files unique to C:\temp\PP3E\Examples\PP3E
... Launcher.py
Files unique to copytemp\PP3E
... PyGadgets.py
Comparing contents
echoEnvironment.pyw matches
LaunchBrowser.pyw matches
Launcher.pyc matches
...множество строк опущено...
PyGadgets_bar.pyw matches
README-PP3E.txt DIFFERS
todos.py matches
tounix.py matches
__init__.py DIFFERS
__init__.pyc matches
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\System\Filetools to copytemp\PP3E\System\
Fil...
Files unique to C:\temp\PP3E\Examples\PP3E\System\Filetools
... cpall_visitor.py
Comparing contents
commands.py DIFFERS
cpall.py matches
...множество строк опущено...
--------------------
Comparing C:\temp\PP3E\Examples\PP3E\TempParts to copytemp\PP3E\TempParts
Directory lists are identical
Comparing contents
109_0237.JPG matches
lawnlake1-jan-03.jpg matches
part-001.txt matches
part-002.html matches
========================================
Diffs found: 5
```

```
- unique files at C:\temp\PP3E\Examples\PP3E - copytemp\PP3E
- files differ at C:\temp\PP3E\Examples\PP3E\README-PP3E.txt -
copytemp\PP3E\README-PP3E.txt
- files differ at C:\temp\PP3E\Examples\PP3E\__init__.py -
copytemp\PP3E\__init__.py
- unique files at C:\temp\PP3E\Examples\PP3E\System\Filetools -
copytemp\PP3E\System\Filetools
- files differ at C:\temp\PP3E\Examples\PP3E\System\Filetools\commands.py -
copytemp\PP3E\System\Filetools\commands.py
```

Я добавил разрывы строк и отступы кое-где, чтобы уместить листинг по ширине страницы, но отчет легко понять. В дереве, насчитывающем 1430 файлов и 185 каталогов, было найдено пять различий – три файла были изменены мною вручную, а два каталога мы рассогласовали тремя командами удаления.

Проверка резервных копий

После того как этот сценарий был написан, я начал использовать его для проверки резервных копий моих ноутбуков на внешнем жестком диске, создаваемых автоматически. Для этого я запускаю сценарий *cpall*, листинг которого представлен ранее, а затем, чтобы проверить результаты и получить список файлов, вызвавших проблемы при копировании, – сценарий сравнения, представленный в этом подразделе здесь. Когда я выполнял эту процедуру в последний раз, было скопировано и проверено 225 000 файлов и 15 000 каталогов, занимающих 20 Гбайт дискового пространства. Ниже приводятся команды, которые я вводил на моем ноутбуке с системой Windows. Здесь f: – это раздел на внешнем жестком диске.

```
C:\...\System\Filetools> cpall.py c:\ f:\ > f:\copy-log.txt
C:\...\System\Filetools> diffall.py f:\ c:\ > f:\diff-log.txt
```

Стоит заметить, что этот сценарий обнаруживает расхожде- ния только в деревьях, не сообщая никаких подробностей о различиях в отдельных файлах. На самом деле он просто загружает и сравнивает двоичное содержимое соответствующих файлов в виде строк, давая простой результат «да/нет». Если нужны дополнительные сведения о фактических различиях в двух несовпавших файлах, то можно либо открыть их в редакторе, либо выполнить команду сравнения файлов на соответствующей платформе (например, fc в Windows/DOS, diff или cmp в Unix и Linux). Этот последний шаг не является переносимым решением, но иногда просто нахождение различий в дереве из 1400 файлов является значительно более важным, чем сообщение в отчете о том, в каких строках различаются эти файлы. Конечно, поскольку в Python всегда можно вызвать команды оболочки, этот последний шаг можно автоматизировать, порождая при обнаружении различий команду diff или fc с помощью os.popen (или делать это после обхода, сканируя содержащуюся в отчете сводку). Вывод этих системных вызовов можно было бы поместить в отчет в первоначальном виде или оставить только наиболее важные его части. Можно также открывать текстовые файлы в текстовом режиме, чтобы игнорировать различия, вызванные разными комбинациями символов завершения строк при передаче файлов между платформами, но не всегда ясно – действительно ли такие отличия должны игнорироваться (что если пользователь пожелает узнать, не изменились ли сим- волы конца строки?). Например, после

загрузки файла с веб-сайта сценарий diffall обнаруживает несоответствие между локальной копией файла и оригиналом на удаленном сервере. Чтобы ответить на эти воппросы, можно выполинть несколько инструкций в интерактивном сеансе Python:

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()
>>> a == b
False
```

Эта проверка показывает, что двоичное содержимое локальной вер- сии файла отличается от содержимого удаленной версии. Чтобы выяснить, обусловлено ли это различием способов завершения строк в Unix и DOS, я попробовал выполнить то же самое, но в текстовом режиме, чтобы перед сравнением символы окончания строк были приведены к стандартному символу
n:

```
>>> a = open('lp2e-updates.html', 'r').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'r').read()
>>> a == b
True
```

Теперь, чтобы отыскать различия, я выполнил следующие инструкции, которые проверяют содержимое символ за символом, пока не наткнутся на первое несоответствие (применение двоичного режима сохраняет различия):

```
>>> a = open('lp2e-updates.html', 'rb').read()
>>> b = open(r'C:\Mark\WEBSITE\public_html\lp2e-updates.html', 'rb').read()
>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...
if ac != bc:
...
print(i, repr(ac), repr(bc))
...
break
...
37966 '\r' '\n'
```

Этот результат означает, что в загруженном файле в байте со смещением 37 966 находится символ
r, а в локальной копии – символ
n. Эта строка в одном файле оканчивается комбинацией символов завершения строки в DOS, а в другом – символом завершения строки в Unix. Чтобы увидеть больше, можно вывести текст, окружающий несовпадение:

```
>>> for (i, (ac, bc)) in enumerate(zip(a, b)):
...
if ac != bc:
...
print(i, repr(ac), repr(bc))
```

```
...
print(repr(a[i-20:i+20]))
...
print(repr(b[i-20:i+20]))
...
break
...
37966 '\r' '\n'
're>\r\ndef min(*args):\r\n
're>\r\ndef min(*args):\n
tmp = list(arg'
tmp = list(args'
```

По всей видимости, я вставил символ завершения строки Unix в одном месте в локальной копии, там, где в загруженной версии находится комбинация символов завершения строки в DOS, – результат использования текстового режима в сценарии загрузки (который преобразует символы
n в комбинации
r
n) и многих лет использования ноутбуков и PDA, работающих под управлением Linux и Windows (вероятно, я внес это изменение, когда после редактирования этого файла в Linux я скопировал его в Windows в двоичном режиме). Такой программный код, как показано выше, можно было бы добавить в сценарий diffall, чтобы обеспечить более интеллектуальное сравнение текстовых файлов и вывод более подробной информации об отличиях в них. Поскольку Python отлично подходит для обработки строк и файлов, можно пойти еще дальше и реализовать на языке Python сценарий, эквивалентный командам fc и diff. Фактически большая часть работы в этом направлении уже выполнена – эту задачу можно было бы суще- ственно упростить, задействовав модуль difflib из стандартной библиотеки. Можно было бы поступить еще умнее и не выполнять загрузку и сравнение файлов, отличающихся размерами; выполнять чтение файлов более мелкими порциями, чтобы уменьшить потребление памяти. Для большинства деревьев каталогов такие оптимизации излишни – чтение многомегабайтных файлов в строки в Python осуществляется очень быстро, а память постоянно освобождается сборщиком мусора в процессе работы. Но эти улучшения выходят за рамки задач данного сценария.
Теперь перейдем к реализации еще одной операции, часто применяемой к деревьям каталогов: поиск.

2.4 Поиск в деревьях каталогов

Ниже приводится утилита find, реализованная мной на языке Python, которая выбирает все имена файлов в каталоге, соответствующие шаблону. В отличие от glob.glob, функция find.find автоматически выполняет поиск во всем дереве каталогов. А в отличие от структуры обхода os.walk, результаты find.find можно трактовать, как простую линейную группу строк.

```
#!/usr/bin/python
"""
```

```
Возвращает все имена файлов, соответствующие шаблону в дереве каталогов;
собственная версия модуля find, ныне исключенного из стандартной библиотеки:
импортируется как "PP4E.Tools.find"; похож на оригинал, но использует цикл
os.walk, не поддерживает возможность обрезания ветвей подкаталогов и может
запускаться как самостоятельный сценарий;
find() - функция-генератор, использующая функцию-генератор os.walk(),
возвращающая только имена файлов, соответствующие шаблону: чтобы получить весь
список результатов сразу, используйте функцию findlist();
"""

import fnmatch, os

def find(pattern, startdir=os.curdir):
    for (thisDir, subsHere, filesHere) in os.walk(startdir):
        for name in subsHere + filesHere:
            if fnmatch.fnmatch(name, pattern):
                fullpath = os.path.join(thisDir, name)
                yield fullpath

def findlist(pattern, startdir=os.curdir, dosort=False):
    matches = list(find(pattern, startdir))
    if dosort: matches.sort()
    return matches

if __name__ == '__main__':
    import sys
    namepattern, startdir = sys.argv[1], sys.argv[2]
    for name in find(namepattern, startdir): print(name)
```

Этот программный код делает немного – по сути, он лишь не- сколько расширяет возможности функции os.walk, – но его функция find позволяет получить те же результаты, что давал ранее существовавший в стандартной библиотеке модуль find и одноименная утилита в Unix. Кроме того, этот модуль является более переносимым решением, и пользоваться им намного проще, чем повторять его программный код всякий раз, когда потребуется выполнить поиск. Поскольку этот файл можно использовать и как сценарий, и как библиотечный модуль, его можно применять как инструмент командной строки и вызывать из других программ. Например, чтобы обработать все файлы с программным кодом на языке Python, находящиеся в дереве каталогов, с корнем на один уровень выше текущего рабочего каталога, достаточно просто запустить приведенную ниже команду в окне консоли. В данном примере стандартный вывод сценария передается по конвейеру команде more, обеспечивающей возможность постраничного просмотра результатов, но его точно так же можно передать любой другой программе обработки, которая читает входные данные из стандартного потока ввода:

```
C:\...\PP4E\Tools> python find.py *.py .. | more
..\LaunchBrowser.py
..\Launcher.py
..\__init__.py
```

```
..\Preview\attachgui.py
..\Preview\customizegui.py
...множество строк опущено...
\end[minted}
```
Чтобы получить еще больший контроль, можно выполнить следующий программный код на языке Python в сценарии или в интерактивной оболочке. При таком подходе к найденным файлам можно применять любые операции, доступные в языке Python:
```
\begin{minted}{bash}
C:\...\PP4E\System\Filetools> python
>>> from PP4E.Tools import find
# или просто import find, если
>>> for filename in find.find('*.py', '..'): # модуль находится в cwd
...
if 'walk' in open(filename).read():
...
print(filename)
...
..\Launcher.py
..\System\Filetools\bigext-tree.py
..\System\Filetools\bigpy-path.py
..\System\Filetools\bigpy-tree.py
..\Tools\cleanpyc.py
..\Tools\find.py
..\Tools\visitor.py
```

В данном случае отпадает необходимость во вложенных циклах, необходимых при использовании функции os.walk, когда требуется получить список имен файлов, соответствующих шаблону, – во многих случаях такой подход концептуально выглядит проще. Функция поиска является функцией-генератором, благодаря чему сценарию не приходится ждать, пока не будут выбраны все соответствующие имена файлов, – функция os.walk поставляет результаты для каждого каталога в отдельности, а функция find.find поставляет имена файлов, выбирая их из этих результатов.
Для упрощения глобального поиска на всех платформах, я написал сценарий на языке Python. В следующем примере применяются стандартные средства Python: os.walk – для обхода файлов в каталоге, os.path. splitext – для пропуска файлов с расширениями, характерными для двоичных файлов, и os.path.join – для переносимого объединения путей к каталогам с именами файлов. Поскольку он написан исключительно на языке Python, этот сценарий в равной степени может использоваться и в Linux, и в Windows. На самом деле он должен работать на любом компьютере, где установлен Python. Более того, благодаря непосредственному использованию системных вызовов он должен работать быстрее, чем при использовании приема запуска команды оболочки.

```
''''
Порядок использования: ''python ...\Tools\search_all.py dir string''.
Отыскивает все файлы в указанном дереве каталогов, содержащие заданную строку;
для предварительного отбора имен файлов использует интерфейс os.walk вместо
```

```
find.find; вызывает visitfile для каждой строки в результатах, полученных
вызовом функции find.find с шаблоном ‘‘*’’;
‘‘’’’
import os, sys
listonly = False
textexts = ['.py', '.pyw', '.txt', '.c', '.h']              # игнорировать двоичные фай

def searcher(startdir, searchkey):
    global fcount, vcount
    fcount = vcount = 0
    for (thisDir, dirsHere, filesHere) in os.walk(startdir):
        for fname in filesHere:                             # для каждого некаталога
            fpath = os.path.join(thisDir, fname)            # fname не содержит пути
            visitfile(fpath, searchkey)

def visitfile(fpath, searchkey):                            # для каждого некаталога
    global fcount, vcount                                   # искать строку
    print(vcount+1, '=>', fpath)                            # пропустить защищенные фай
    try:
        if not listonly:
            if os.path.splitext(fpath)[1] not in textexts:
                print('Skipping', fpath)
            elif searchkey in open(fpath).read():
                input('%s has %s' % (fpath, searchkey))
                fcount += 1
    except:
        print('Failed:', fpath, sys.exc_info()[0])
    vcount += 1

if __name__ == '__main__':
    searcher(sys.argv[1], sys.argv[2])
    print('Found in %d files, visited %d' % (fcount, vcount))
```

Инструмент поиска хорошо подходит для выбора файлов определенного типа, при этом преимущество данного сценария состоит в возможности произвести определенные действия непосредственно в процессе обхода. При запуске в виде самостоятельного сценария ключ поиска передается в командной строке, а при импортировании клиент вызывает функцию searcher непосредственно. Например, чтобы найти все вхождения строки в дереве каталогов, достаточно в команду в оболочке DOS или Unix, как показано ниже:

```
C:\\PP4E> Tools\search_all.py . mimetypes
1 => .\LaunchBrowser.py
2 => .\Launcher.py
3 => .\Launch_PyDemos.pyw
4 => .\Launch_PyGadgets_bar.pyw
5 => .\__init__.py
6 => .\__init__.pyc
Skipping .\__init__.pyc
```

```
7 => .\Preview\attachgui.py
8 => .\Preview\bob.pkl
Skipping .\Preview\bob.pkl
...множество строк опущено: ожидает нажатия клавиши Enter после
обнаружения каждого совпадения...
Found in 2 files, visited 184
```

Сценарий выводит список всех проверяемых им файлов, сообщает о пропущенных файлах (имена с расширениями, отсутствующими в перемен- ной textexts, которые, как предполагается, являются двоичными файлами) и останавливается, ожидая нажатия клавиши Enter после вывода сообщения о нахождении в файле искомой строки. Точно так же сценарий search_all работает и при импортировании, но не выводит итоговой строки со статистикой (функции fcount и vcount находятся в модуле, и их также можно импортировать, чтобы получить итоговые сведения):

```
C:\...\P4E\dev\Examples\P4E> python
>>> import Tools.search_all
>>> search_all.searcher(r'C:\temp\P3E\Examples', 'mimetypes')
... множество строк опущено: останавливается 8 раз в ожидании
нажатия клавиши Enter...
>>> search_all.fcount, search_all.vcount
# совпадений, файлов
(8, 1429)
```

Каким бы образом ни запускался этот сценарий, он находит все вхожде- ния искомой строки в целом дереве каталогов – например, изменившееся имя файла, объекта или каталога в коде. Многие редакторы кода и IDE (Integrated development evnironment) предоставляют возможность переименовать объект (переменную, функцию, класс). При этом редактов найдет все упоминания данного имени во всех файлах проекта, и произведет замену старого имени на новое.

2.5 Обход каталогов

Если нужно отредактировать все найденные файлы можно одном окне запускить search_all, чтобы отобрать нужные файлы, и вручную редактировсть каждый из них в другом окне. Но вводить вручную имена файлов в командах запуска редактора неоптимально, особенно если нужно отредактировать много файлов, и кроме того всегда есть вероятность допустить ошибку в названии файла. Вместо того чтобы десятки раз запускать редактор вручную, лучше автоматически запускать редактор для каждого найденного файла. Но сценарий search_all просто выводит полученные результаты на экран. Хотя этот текст можно перехватить и проанализировать с помощью другой программы, запускаемой функцией os.popen, проще может оказаться подход, когда редактор запускается прямо во время поиска, но для этого могут потребоваться большие изменения в реали- зации сценария. При этом стоит обратить внимание на три вещи:

Избыточность

Нужно стремиться избегать написания однотипного программного кода. Обход можно упростить еще больше, если скрыть детали под оболочкой и тем самым упростить повторное использование решения. Инструмент os.walk позволяет избежать необходимости

писать рекурсивные функции, но при его использовании выполняются лишние действия (например, присоединение имен каталогов, вывод трассировочной информации).

Расширяемость

Очевидно, что в долгосрочной перспективе легче добавлять новые возможности в универсальный механизм поиска в каталогах в виде внешних компонентов, чем менять программный код исходного сценария. Редактирование файлов могло быть одним из возможных расширений, поэтому предпочтительнее выглядит более обобщенное и настраиваемое решение, допускающее возможность многократного использования. Функция os.walk доста- точно проста в использовании, но прием, основанный на циклах, не так хорошо поддается настройке, как использование классов.

Инкапсуляция

Всегда желательно стараться максимально скрывать детали реализации инструментов от программ. Функция os.walk скрывает свою рекурсивную природу, тем не менее она предлагает весьма специфический интерфейс, который вполне может измениться в будущем. Подобные изменения имели место в прошлом – например из версии Python 3.X был исключен один из инструментов обхода деревьев, что сразу же привело к нарушениям в работе програмного кода, использующего его. Было бы лучше скрыть подобные зависимости за более нейтральным интерфейсом, чтобы клиентский программный код не приходил в негодность, как только нам потребуется внести изменения в реализацию нашего инструмента.

Все эти цели указывают на необходимость использования объектно-ориентированного подхода к реализации обхода и поиска. Далее приводится одна из возможных реализаций этих целей. Этот модуль экспортирует универсальный класс FileVisitor, который в основном служит лишь оболочкой для os.walk, облегчающей использование и расширение, а также базовый класс SearchVisitor, обоб- щающий идею поиска в каталоге.

Сам по себе класс SearchVisitor делает то же самое, что делал сценарий search_all, но кроме этого, он открывает новые возможности по настройке процедуры поиска – какие-то черты его поведения могут модифицироваться путем перегрузки методов в подклассах. Более того, его базовая логика поиска может быть использована везде, где требуется поиск: достаточно просто определить подкласс, в котором будут добавлены специфические для поиска расширения. То же относится и к классу FileVisitor – переопределяя его методы и используя его атрибуты, можно внедряться в процесс обхода деревьев, используя приемы ООП.

```
‘‘””
Тест: ‘‘python ...\Tools\visitor.py dir testmask [строка]’’. Использует
классы и подклассы для сокрытия деталей использования функции os.walk при
обходе и поиске; testmask - битовая маска, каждый бит в которой определяет
тип самопроверки; смотрите также: подклассы visitor_*/.py; вообще подобные
фреймворки должны использовать псевдочастные имена вида __X, однако в данной
реализации все имена экспортируются для использования в подклассах и клиентами;
переопределите метод reset для поддержки множественных, независимых объектов-
обходчиков, требующих обновлений в подклассах;
‘‘””
import os, sys
```

```
class FileVisitor:
    """
    Visits all nondirectory files below startDir (default '.');
    override visit* methods to provide custom file/dir handlers;
    context arg/attribute is optional subclass-specific state;
    trace switch: 0 is silent, 1 is directories, 2 adds files
    """
    def __init__(self, context=None, trace=2):
        self.fcount   = 0
        self.dcount   = 0
        self.context  = context
        self.trace    = trace

    def run(self, startDir=os.curdir, reset=True):
        if reset: self.reset()
        for (thisDir, dirsHere, filesHere) in os.walk(startDir):
            self.visitdir(thisDir)
            for fname in filesHere:                    # для некаталогов
                fpath = os.path.join(thisDir, fname) # fname не содержит пути
                self.visitfile(fpath)

    def reset(self):                                 # используется обходчиками,
        self.fcount = self.dcount = 0                # выполняющими обход независимо

    def visitdir(self, dirpath):                     # вызывается для каждого каталога
        self.dcount += 1                             # переопределить или расширить
        if self.trace > 0: print(dirpath, '...')

    def visitfile(self, filepath):                   # вызывается для каждого файла
        self.fcount += 1                             # переопределить или расширить
        if self.trace > 1: print(self.fcount, '=>', filepath)


class SearchVisitor(FileVisitor):
    """
    Выполняет поиск строки в файлах, находящихся в каталоге startDir и ниже;
в подклассах: переопределите метод visitmatch, списки расширений, метод
candidate, если необходимо; подклассы могут использовать testexts, чтобы
определить типы файлов, в которых может выполняться поиск (но могут также
переопределить метод candidate, чтобы использовать модуль mimetypes для
определения файлов с текстовым содержимым: смотрите далее)
    """

    skipexts = []
    testexts = ['.txt', '.py', '.pyw', '.html', '.c', '.h']  # допустимые расш.
   #skipexts = ['.gif', '.jpg', '.pyc', '.o', '.a', '.exe']  # или недопустимые
```

```
    def __init__(self, searchkey, trace=2):
        FileVisitor.__init__(self, searchkey, trace)
        self.scount = 0

    def reset(self):                                # в независимых обходчиках
        self.scount = 0

    def candidate(self, fname):                     # переопределить, если желательно

        ext = os.path.splitext(fname)[1]
        if self.testexts:
            return ext in self.testexts             # если допустимое расширение
        else:                                       # или, если недопустимое
            return ext not in self.skipexts

    def visitfile(self, fname):                     # поиск строки
        FileVisitor.visitfile(self, fname)
        if not self.candidate(fname):
            if self.trace > 0: print('Skipping', fname)
        else:
            text = open(fname).read()               # 'rb' для недекодируемого текста
            if self.context in text:                # или text.find() != -1
                self.visitmatch(fname, text)
                self.scount += 1

    def visitmatch(self, fname, text):                  # обработка совпадения
        print('%s has %s' % (fname, self.context))      # переопределить


if __name__ == '__main__':
    # self-test logic
    dolist   = 1
    dosearch = 2    # 3 = список и поиск
    donext   = 4    # при добавлении следующего теста

    def selftest(testmask):
        if testmask & dolist:
           visitor = FileVisitor(trace=2)
           visitor.run(sys.argv[2])
           print('Visited %d files and %d dirs' % (visitor.fcount, visitor.dcount))

        if testmask & dosearch:
           visitor = SearchVisitor(sys.argv[3], trace=0)
           visitor.run(sys.argv[2])
           print('Found in %d files, visited %d' % (visitor.scount, visitor.fcount))

    selftest(int(sys.argv[1]))    # например, 3 = dolist | dosearch
```

Этот модуль служит в основном для экспорта классов, используемых другими программами, но и при запуске в виде самостоятельного сценария делает кое-что полезное. Если вызвать его как сценарий с одним аргументом 1, он создаст и запустит объект FileVisitor и выведет полный список всех файлов и каталогов, начиная с того каталога, откуда он вызван, и ниже:

```
C:\...\PP4E\Tools> visitor.py 1 C:\temp\PP3E\Examples
C:\temp\PP3E\Examples ...
1 => C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E ...
2 => C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
3 => C:\temp\PP3E\Examples\PP3E\LaunchBrowser.pyw
4 => C:\temp\PP3E\Examples\PP3E\Launcher.py
5 => C:\temp\PP3E\Examples\PP3E\Launcher.pyc
...множество строк опущено (передайте по конвейеру команде more или
перенаправьте в файл)...
1424 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread-count.py
1425 => C:\temp\PP3E\Examples\PP3E\System\Threads\thread1.py
C:\temp\PP3E\Examples\PP3E\TempParts ...
1426 => C:\temp\PP3E\Examples\PP3E\TempParts\109_0237.JPG
1427 => C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Visited 1429 files and 186 dirs
```

Если же вызвать этот сценарий с 2 в первом аргументе, он создаст и запустит объект SearchVisitor, используя третий аргумент в качестве ключа поиска. Эта форма напоминает запуск сценария search_all.py, но в данном случае при обнаружении совпадений сценарий не останавливается:

```
C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples mimetypes
C:\temp\PP3E\Examples\PP3E\extras\LosAlamosAdvancedClass\day1-system\data.txt
has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailParser.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Email\mailtools\mailSender.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\downloadflat_modular.py has
mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\ftptools.py has mimetypes
C:\temp\PP3E\Examples\PP3E\Internet\Ftp\mirror\uploadflat.py has mimetypes
C:\temp\PP3E\Examples\PP3E\System\Media\playfile.py has mimetypes
Found in 8 files, visited 1429
```

Технически при передаче сценарию числа 3 в первом аргументе он вы- полнит оба объекта, FileVisitor и SearchVisitor (осуществив два отдельных обхода). Первый аргумент в действительности используется в качестве битовой маски для выбора одной или более поддерживаемых само-проверок – если бит для какого-либо теста установлен в двоичном значении аргумента, этот тест будет выполнен. Поскольку 3 представляется в двоичном виде, как

011, выбираются одновременно поиск (010) и вывод списка (001). В более дружественной системе можно было бы определить символические параметры (например, искать аргументы -search и -list), но для целей данного сценария достаточно битовых масок. Этот модуль можно также использовать в интерактивном сеансе. Ниже приводится один из способов определения количества файлов и каталогов внутри определенного каталога. Последняя команда выполняет обход всего жесткого диска (и выводит результаты после заметной задержки). Такие проблемы, как повторное посещение подкаталогов, не обрабатываемые данной реализацией:

```
C:\...\PP4E\Tools> python
>>> from visitor import FileVisitor
>>> V = FileVisitor(trace=0)
>>> V.run(r'C:\temp\PP3E\Examples')
>>> V.dcount, V.fcount
(186, 1429)
>>> V.run('..')
>>> V.dcount, V.fcount
(19, 181)
# независимый обход (сброс счетчиков)
>>> V.run('..', reset=False) # накопительный обход (счетчики сохраняются)
>>> V.dcount, V.fcount
(38, 362)
>>> V = FileVisitor(trace=0) # новый независимый обходчик (свои счетчики)
# весь диск: в Unix попробуйте '/'
>>> V.run(r'C:\\')
>>> V.dcount, V.fcount
(24992, 198585)
```

Модуль visitor удобно использовать как самостоятельный сценарий, чтобы получить список файлов и выполнить поиск в дереве каталогов, но в действительности он создавался, чтобы служить основой для расширения. Ниже приведены некоторые клиентами этого модуля, которые добавляют свои операции с деревьями каталогов, используя приемы ООП.

Редактирование файлов в деревьях каталогов (Visitor)

Теперь, после обобщения обхода деревьев и поиска, легко сделать следующий шаг и добавить отдельный, совершенно новый компонент автоматического редактирования файлов.Здесь приводится определение нового класса EditVisitor, который просто переопределяет метод visitmatch класса SearchVisitor, новая версия которого открывает найденный файл в текстовом редакторе. Да, это законченная программа – что-либо особое нужно делать только при обработке найденных файлов, и только это поведение должно обеспечиваться. Все остальное, касающееся логики обхода и поиска, остается неизменным и приобретается по наследству.

```
""
Порядок использования: "python ...\Tools\visitor_edit.py string rootdir?".
Добавляет подкласс класса SearchVisitor, который автоматически запускает
текстовый редактор. В процессе обхода автоматически открывает в текстовом
```

```
редакторе файлы, содержащие искомую строку; в Windows можно также использовать
editor='edit' или 'notepad'; чтобы воспользоваться текстовым редактором,
реализация которого будет представлена далее в книге, попробуйте r'python Gui\
TextEditor\textEditor.py'; при работе с некоторыми редакторами можно было бы
передать
''''
import os, sys
from visitor import SearchVisitor

class EditVisitor(SearchVisitor):
    """
    открывает для редактирования файлы, содержащие искомую строку и
находящиеся в каталоге startDir и ниже
    """
    editor = r'C:\cygwin\bin\vim-nox.exe'   # ymmv!

    def visitmatch(self, fpathname, text):
        os.system('%s %s' % (self.editor, fpathname))

if __name__  == '__main__':
    visitor = EditVisitor(sys.argv[1])
    visitor.run('.' if len(sys.argv) < 3 else sys.argv[2])
    print('Edited %d files, visited %d' % (visitor.scount, visitor.fcount))
```

При использовании объекта EditVisitor текстовый редактор запускается посредством передачи командной строки функции os.system, которая обычно блокирует вызывающий программный код до момента, когда завершится порожденная программа. При каждом обнаружении сценарием совпадения во время обхода запускается текстовый редактор vi в том окне консоли, где был запущен сценарий. При выходе из редактора обход дерева возобновляется. Найдем и отредактируем несколько файлов. При запуске этого файла как самостоятельного сценария мы передаем ему искомую строку в аргументе командной строки (здесь используется строка «mimetypes»). Корневой каталог всегда передается методу run как «.» (текущий рабочий каталог). Сообщения о состоянии обхода выводятся на консоль, но каждый файл, в котором обнаружено совпадение с искомой строкой, тут же автоматически открывается в текстовом редакторе. В данном случае редактор запускается восемь раз.

```
C:\...\PP4E\Tools> visitor_edit.py mimetypes C:\temp\PP3E\Examples
C:\temp\PP3E\Examples ...
1 => C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E ...
2 => C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
3 => C:\temp\PP3E\Examples\PP3E\LaunchBrowser.pyw
4 => C:\temp\PP3E\Examples\PP3E\Launcher.py
5 => C:\temp\PP3E\Examples\PP3E\Launcher.pyc
Skipping C:\temp\PP3E\Examples\PP3E\Launcher.pyc
...множество строк опущено...
```

```
1427 => C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
Skipping C:\temp\PP3E\Examples\PP3E\TempParts\lawnlake1-jan-03.jpg
1428 => C:\temp\PP3E\Examples\PP3E\TempParts\part-001.txt
1429 => C:\temp\PP3E\Examples\PP3E\TempParts\part-002.html
Edited 8 files, visited 1429
```

При таком подходе все еще приходится вручную изменять файлы в редакторе, но это зачастую безопаснее, чем вслепую выполнять глобальную замену.

Глобальная замена в деревьях каталогов (Visitor)

Имея общий класс для обхода дерева, легко написать и подкласс для глобального поиска и замены. Ниже приводится определение класса ReplaceVisitor, насле- дующего класс FileVisitor, который переопределяет метод visitfile так, чтобы глобально заменять все вхождения одной строки другой строкой во всех текстовых файлах, находящихся в корневом каталоге и ниже. Он также составляет список всех изменившихся файлов, чтобы их можно было просмотреть и проверить автоматически сделанные изме- нения (можно, например, автоматически вызывать текстовый редактор для каждого измененного файла).

```
‘“”’
Использование: ‘‘python ...\Tools\visitor_replace.py rootdir fromStr toStr’’.
Выполняет глобальный поиск с заменой во всех файлах в дереве каталогов: заменяет
fromStr на toStr во всех текстовых файлах; это мощный, но опасный инструмент!!
visitor_edit.py запускает редактор, чтобы дать возможность проверить и внести
коррективы, и поэтому он более безопасный; чтобы просто получить список
соответствующих файлов, используйте visitor_collect.py; режим простого вывода
списка здесь напоминает SearchVisitor и CollectVisitor;
‘“”’
import sys
from visitor import SearchVisitor

class ReplaceVisitor(SearchVisitor):
    """
    Заменяет fromStr на toStr в файлах в каталоге startDir и ниже;
имена изменившихся файлов сохраняются в списке obj.changed
    """
    def __init__(self, fromStr, toStr, listOnly=False, trace=0):
        self.changed  = []
        self.toStr    = toStr
        self.listOnly = listOnly
        SearchVisitor.__init__(self, fromStr, trace)

    def visitmatch(self, fname, text):
        self.changed.append(fname)
        if not self.listOnly:
            fromStr, toStr = self.context, self.toStr
            text = text.replace(fromStr, toStr)
```

```
            open(fname, 'w').write(text)

if __name__  == '__main__':
    listonly = input('List only?') == 'y'
    visitor  = ReplaceVisitor(sys.argv[2], sys.argv[3], listonly)
    if listonly or input('Proceed with changes?') == 'y':
        visitor.run(startDir=sys.argv[1])
        action = 'Changed' if not listonly else 'Found'
        print('Visited %d files'  % visitor.fcount)
        print(action, '%d files:' % len(visitor.changed))
        for fname in visitor.changed: print(fname)
```

Чтобы применить этот сценарий к определенному дереву каталогов, следует выполнить команду, как показано ниже, указав соответствующую искомую строку и строку замены. Обработка дерева с 1429 файлами, из которых 101 потребовалось изменить, заняла примерно три секунды реального времени, когда система была не слишком занята другими задачами:

```
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?y
Visited 1429 files
Found 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
...большое количество имен файлов, соответствующих критерию поиска,
опущено...
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 101 files:
C:\temp\PP3E\Examples\README-root.txt
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw
C:\temp\PP3E\Examples\PP3E\Launcher.py
...большое количество имен изменившихся файлов опущено...
C:\...\PP4E\Tools> visitor_replace.py C:\temp\PP3E\Examples PP3E PP4E
List only?n
Proceed with changes?y
Visited 1429 files
Changed 0 files:
```

Естественно, проверить работу этого сценария можно с помощью сценария visitor (и суперкласса SearchVisitor):

```
C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP3E
Found in 0 files, visited 1429
C:\...\PP4E\Tools> visitor.py 2 C:\temp\PP3E\Examples PP4E
```

```
C:\temp\PP3E\Examples\README-root.txt has PP4E
C:\temp\PP3E\Examples\PP3E\echoEnvironment.pyw has PP4E
C:\temp\PP3E\Examples\PP3E\Launcher.py has PP4E
...большое количество имен файлов, соответствующих критерию поиска,
опущено...
Found in 101 files, visited 1429
```

Это одновременно очень мощный и опасный сценарий. Если заменяемая строка может обнаружиться в неожиданных местах, запуск определенного здесь объекта ReplaceVisitor может разрушить все дерево файлов. С другой стороны, если строка является чем-то очень специфическим, этот объект поможет избежать необходимости вручную редактировать подозрительные файлы. Например, адреса веб-сайтов в файлах HTML достаточно специфичны, чтобы случайно появиться в других местах.

3 Заключение

Простота использования Python и обширное многообразие встроенных библиотек упрощают использование развитых системных инструментов, таких как потоки выполнения, сигналы, ветвление процессов, сокеты и аналогичные им; такие инструменты намного сложнее использовать в неясном синтаксисе языков оболочек и в многоэтапных циклах разработки на компилируемых языках. Поддержка в Python таких идей, как ясность программного кода и объектно-ориентированное программирование, способствует созданию таких инструментов оболочки, которые можно читать, сопровождать и повторно использовать.

При использовании Python нет необходимости начинать с нуля каждый новый сценарий. Более того, в Python не только есть все интерфейсы, необходимые для разработки системных инструментов, но он также обеспечивает переносимость сценариев. При использовании стандартной библиотеки Python большинство системных сценариев, написанных на языке Python, автоматически становятся переносимыми на все основные платформы. Например, сценарий для обработки каталогов, написанный под Windows, обычно может выполняться и под Linux безо всякой правки исходных текстов: достаточно просто скопировать сценарий. Для разработки сценариев, обеспечивающих такой уровень переносимости, необходимо прикладывать некоторые усилия, тем не менее при разумном использовании Python может стать единственным средством, которым необходимо владеть для создания системных сценариев.

4 Список литературы

1. Марк Лутц - Программирование на Python (4 издание)
2. `https://docs.python.org/3.5/tutorial/index.html` - The Python Tutorial
3. `https://docs.python.org/3/library/pathlib.html` - pathlib documentation
4. `https://docs.python.org/3/library/os.html#module-os` - os documentation