Basic file wrangling

This lab covers

- [Moving and renaming files]
- [Compressing and encrypting files]
- [Selectively deleting files]

This lab deals with the basic operations you can use when you have an ever-increasing collection of files to manage. Those files might be log files, or they might be from a regular data feed, but whatever their source, you can't simply discard them immediately. How do you save them, manage them, and ultimately dispose of them according to a plan, but without manual intervention?

The problem: The never-ending flow of data files

Many systems generate a continuous series of data files. These files might be the log files from an e-commerce server or a regular process; they might be a nightly feed of product information from a server; they might be automated feeds of items for online advertising; historical data of stock trades; or they might come from a thousand other sources. They're often flat text files, uncompressed, with raw data that's either an input or a byproduct of other processes. In spite of their humble nature, however, the data they contain has some potential value, so the files can't be discarded at the end of the day---which means that every day, their numbers grow. Over time, files accumulate until dealing with them manually becomes unworkable and until the amount of storage they consume becomes unacceptable.

Scenario: The product feed from hell

A typical situation I've encountered is a daily feed of product data. This data might be coming in from a supplier or going out for online marketing, but the basic aspects are the same.

Consider the example of a product feed coming from a supplier. The feed file comes in once a day, with one row for each item that the business supplies. Each row has fields for the supplier's stock-keeping unit (SKU) number; a brief description of the item; the item's cost, height, width, length, and width; the item's status (in stock or back-ordered, say); and probably several other things, depending on the business.

In addition to this basic info file, you might well be getting others, possibly of related products, more detailed item attributes, or something else. In that case, you end up with several files with the same filenames arriving every day and landing in the same directory for processing.

Now assume that you get three related files every day: item_info.txt, item_attributes.txt, related_items.txt. These three files come in every day and get processed. If processing were the only requirement, you wouldn't have to worry much; you could just let each day's set of files replace the last and be done with it. But what if you can't throw the data away? You may want to keep the raw data in case there's a question about the accuracy of the process and you need to refer to past files. Or you may want to track the changes in the data over time. Whatever the reason, the need to keep the files means that you need to do some processing.

The simplest thing you might do is mark the files with the dates on which they were received and move them to an archive folder. That way, each new set of files can be received, processed, renamed, and moved out of the way so that the process can be repeated with no loss of data.

After a few repetitions, the directory structure might look something like this:

```
working/
    item_info.txt
    item_attributes.txt
    related_items.txt
    archive/
        item_info_2017-09-15.txt
        item_attributes_2017-09-15.txt
        related_items_2017-09-15.txt
        item_info_2016-07-16.txt
        item_attributes_2017-09-16.txt
        related_items_2017-09-16.txt
        item_info_2017-09-17.txt
        item_attributes_2017-09-17.txt
        related_items_2017-09-17.txt
        ...
```

Think about the steps needed to make this process happen. First, you need to rename the files so that the current date is added to the filename. To do that, you need to get the names of the files you want to rename; then you need to get the stem of the filenames without the extensions. When you have the stem, you need to add a string based on the current date, add the extension back to the end, and then actually change the filename and move it to the archive directory.

Quick Check: Consider the choices

What are your options for handling the tasks I've identified? What modules in the standard library can you think of that will do the job? If you want, you can even stop right now and work out the code to do it. Then compare your solution with the one you develop later.

You can get the names of the files in several ways. If you're sure that the names are always exactly the same and that there aren't many files, you *could* hardcode them into your script. A safer way, however, is to use the [pathlib] module and a path object's [glob] method, as follows:

```
>>> import pathlib
>>> cur_path = pathlib.Path(".")
>>> FILE_PATTERN = "*.txt"
>>> path_list = cur_path.glob(FILE_PATTERN)
>>> print(list(path_list))
[PosixPath('item_attributes.txt'), PosixPath('related_items.txt'),
     PosixPath('item_info.txt')]
```

Now you can step through the paths that match your [FILE_PATTERN] and apply the needed changes. Remember that you need to add the date as part of the name of each file, as well move the renamed files to the archive directory. When you use [pathlib], the entire operation might look like this.

Listing 20.1. File files_01.py

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)
    for path in paths:
        new_filename = "{}_{}{}".format(path.stem, date_string, path.suffix)
        new_path = cur_path.joinpath(ARCHIVE, new_filename)
        path.rename(new_path)
```

It's worth noting here that [Path] objects make this operation simpler, because no special parsing is needed to separate the filename stem and suffix. This operation is also simpler than you might expect because the [rename] method can in effect move a file by using a path that includes the new location.

This script is a very simple one and does the job effectively in very few lines of code. In the next sections, you consider how to handle more complex requirements.

Quick Check: Potential Problems

Because the preceding solution is very simple, there are likely to be many situations that it won't handle well. What are some potential issues or problems that might arise with the example script? How might you remedy these problems?

Consider the naming convention used for the files, which is based on the year, month and name, in that order. What advantages do you see in that convention? What might be the disadvantages? Can you make any arguments for putting the date string somewhere else in the filename, such as the beginning or the end?

More organization

The solution to storing files described in the previous section works, but it does have some disadvantages. For one thing, as the files accumulate, managing them might become a bit more trouble, because over the course of a year, you'd have 365 sets of related files in the same directory, and you could find the related files only by inspecting their names. If the files arrive more frequently, of course, or if there are more related files in a set, the hassle would be even greater.

To mitigate this problem, you can change the way you archive the files. Instead of changing the filenames to include the dates on which they were received, you can create a separate subdirectory for each set of files and name that subdirectory after the date received. Your directory structure might look like this:

```
working/
    item_info.txt
    item_attributes.txt
    related_items.txt
    archive/
        2016-09-15/
            item_info.txt
            item_attributes.txt
            related_items.txt
        2016-09-16/
            item_info.txt
            item_attributes.txt
            related_items.txt
        2016-09-17/
            item_info.txt
            item_attributes.txt
            related_items.txt
```

This scheme has the advantage that each set of files is grouped together. No matter how many sets of files you get or how many files you have in a set, it's easy to find all the files of a particular set.

Try this: Implementation of multiple directories

How would you modify the code that you developed to archive each set of files in subdirectories named according to date received? Feel free to take the time to implement the code and test it.

It turns out that archiving the files by subdirectory isn't much more work than the first solution. The only additional step is to create the subdirectory before renaming the file. This script is one way to perform this step.

Listing 20.2. File files_02.py

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")

    new_path = cur_path.joinpath(ARCHIVE, date_string)
    new_path.mkdir()

    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
        path.rename(new_path.joinpath(path.name))
```

This solution groups related files, which makes managing them as sets somewhat easier.

Quick Check: Alternate solutions
How might you create a script that does the same thing without using [pathlib]? What libraries and functions would you use?

Saving storage space: Compression and grooming

So far, you've been concerned mainly with managing the groups of files received. Over time, however, the data files accumulate until the amount of storage they need becomes a concern. When that happens, you have several choices. One option is to get a bigger disk. Particularly if you're on a cloud-based platform, it may be easy and economical to adopt this strategy. Do keep in mind, however, that adding storage doesn't really solve the problem; it merely postpones solving it.

Compressing files

If the space that the files are taking up is an issue, the next approach you might consider is compressing them. You have numerous ways to compress a file or set of files, but in general, these methods are similar. In this section, you consider archiving each day's data file to a single zip file. If the files are mainly text files and are fairly large, the savings in storage achieved by compression can be impressive.

For this script, you use the same date string with a [.zip] extension as the name of each zip file. In [listing 20.2], you created a new directory in the archive directory and then moved the files into it, which resulted in a directory structure that looks like this:

```
working/
    archive/
```

```
        2016-09-15.zip
        2016-09-16.zip
        2016-09-17.zip
```

Obviously, to use zip files you need to change some of the steps you used previously.

Try this: Archiving to zip files pseudocode

Write the pseudocode for a solution that stores data files in zip files. What modules and functions or methods do you intend to use? Try coding your solution to make sure that it works.

One key addition in the new script is an import of the zipfile library and with it, the code to create a new zip file object in the archive directory. After that, you can use the zip file object to write the data files to the new zip file. Finally, because you're no longer actually moving files, you need to remove the original files from the working directory. One solution looks like this.

Listing 20.3. File files_03.py

```
import datetime
import pathlib
import zipfile

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)

    zip_file_path = cur_path.joinpath(ARCHIVE, date_string + ".zip")
    zip_file = zipfile.ZipFile(str(zip_file_path), "w")

    for path in paths:
        zip_file.write(str(path))
        path.unlink()
```

Grooming files

Compressing data files into zipfile archives can save an impressive amount of space and may be all you need. If you have a lot of files, however, or files that don't compress much (such as JPEG image files), you may still find yourself running short of storage space. You may also find that your data doesn't change much, making it unnecessary to keep an archived copy of every data set in the longer term. That is, although it may be useful to keep every day's data for the past week or month, it may not be worth the storage to keep every data set for much longer. For data older than a few months, it may be acceptable to keep just one set of files per week or even one set per month.

The process of removing files after they reach a certain age is sometimes called *grooming*. Suppose that after several months of receiving a set of data files every day and archiving them in a zip file, you're told that you should retain only one file a week of the files that are more than one month old.

The simplest grooming script removes any files that you no longer need---in this case, all but one file a week for anything older than a month old. In designing this script, it's helpful to know the answers to two questions:

- [Because you need to save one file a week, would it be much easier to simply pick the day of the week you want to save?]
- [How often should you do this grooming: daily, weekly, or once a month? If you decide that grooming should take place daily, it might make sense to combine the grooming with the archiving script. If, on the other hand, you need to groom only once a week or once a month, the two operations should be in separate scripts.]

For this example, to keep things clear, you write a separate grooming script that can be run at any interval and that removes all the unneeded files. Further, assume that you've decided to keep only the files received on Tuesdays that are more than one month old. Here is a sample grooming script.

Listing 20.4. File files_04.py

```
from datetime import datetime, timedelta
import pathlib
import zipfile

FILE_PATTERN = "*.zip"
ARCHIVE = "archive"
ARCHIVE_WEEKDAY = 1
if __name__ == '__main__':
    cur_path = pathlib.Path(".")
    zip_file_path = cur_path.joinpath(ARCHIVE)

    paths = zip_file_path.glob(FILE_PATTERN)
    current_date = datetime.today()

    for path in paths:
        name = path.stem
        path_date = datetime.strptime(name, "%Y-%m-%d")
        path_timedelta = current_date - path_date
        if path_timedelta > timedelta(days=30) and path_date.weekday() !=
     ARCHIVE_WEEKDAY:
            path.unlink()
```

The code shows how Python's datetime and pathlib libraries can be combined to groom files by date with only a few lines of code. Because your archive files have names derived from the dates on which they were received, you can get those file paths by using the [glob] method, extract the stem, and use [strptime] to parse it into a [datetime] object. From there, you can use [datetime]'s [timedelta] objects and the [weekday()] method to find a file's age and the day of the week, and then remove (unlink) the files you don't need.

Quick Check: Consider different parameters

Take some time to consider different grooming options. How would you modify the code in [listing 20.4] to keep only one file a month? How would you change it so that files from the previous month and older were groomed to save one a week? (Note: This is *not* the same as older than 30 days!)

Summary

- [The [pathlib] module can greatly simplify file operations such as finding the root and extension, moving and renaming, and matching wildcards.]
- [As the number and complexity of files increase, automated archiving solutions are vital, and Python offers several easy ways to create them.]
- [You can dramatically save storage space by compressing and grooming data files.]