

# Basic file wrangling

## Chapter 20

# This chapter covers

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

# 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 example 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.
- 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

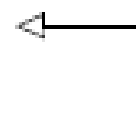
related\_items\_2017-09-16.txt

item\_info\_2017-09-17.txt

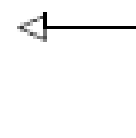
item\_attributes\_2017-09-17.txt

related\_items\_2017-09-17.txt

...



Main working folder, with  
current files for processing



Subdirectory for archiving  
processed files

# How to solve it?

- 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.

```
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)
```

← Sets the pattern to match files  
and the archive directory

← A directory named  
"archive" must exist  
for this code to run.

← Uses the date object from the  
datetime library to create a date  
string based on today's date

← Renames (and moves)  
the file as one step

Creates a new  
path from the  
current path, the  
archive directory,  
and the new  
filename  
→

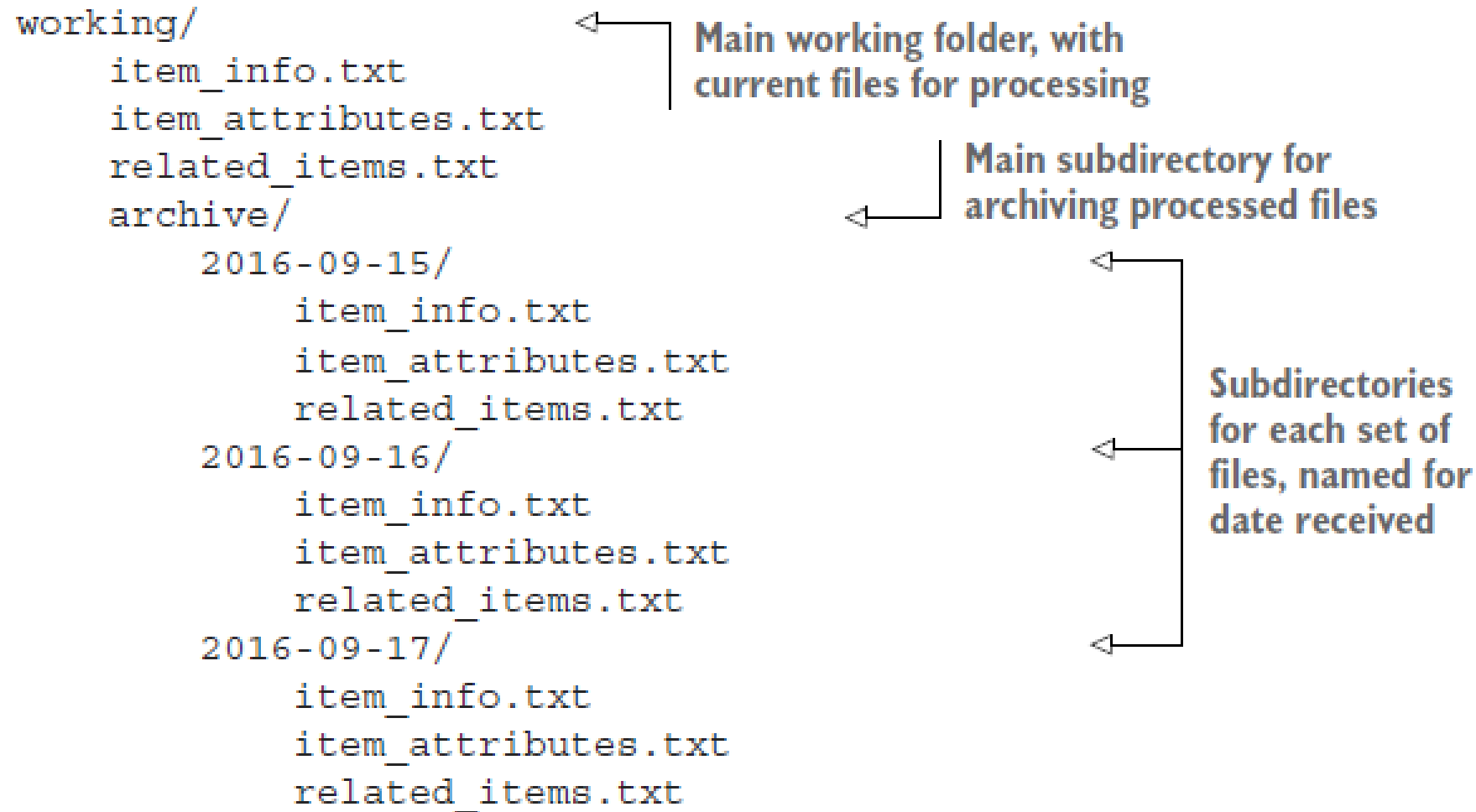
# More organization

- 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.



# A better solution

- 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 (next slide)



```
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))
```

← Note that this directory needs to be created only once, before the files are moved into it.

# Compressing files

- If the space that the files are taking up is an issue, the next approach you might consider is compressing them.

```
working/
  archive/
    2016-09-15.zip
    2016-09-16.zip
    2016-09-17.zip
```

← Main working folder, where current files are processed;  
these files are archived and removed after processing.

Zip files, each one containing that day's  
item\_info.txt, attribute\_info.txt, and  
related\_items.txt

```
import datetime
import pathlib
import zipfile
```

← Imports zipfile  
library

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

```
if __name__ == '__main__':
```

Creates the path to the zip  
file in the archive directory

```
    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")
```

← Opens the new zip file object for writing; str()  
is needed to convert a Path to a string.

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

← Removes the current file  
from the working directory

← Writes the current  
file to the zip file

# Grooming files @ Deleting

- 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.

```
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)
```

path.stem  
returns the  
filename  
without any  
extension.

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

Gets a datetime object  
for the current day

```
    for path in paths:
        name = path.stem
        path_date = datetime.strptime(name, "%Y-%m-%d")
```

strptime parses a string  
into a datetime object based  
on the format string.

Subtracting one  
date from  
another yields a  
timedelta object.

```
        path_timedelta = current_date - path_date
        if path_timedelta > timedelta(days=30) and path_date.weekday() !=
        ARCHIVE_WEEKDAY:
            path.unlink()
```

timedelta(days=30) creates a timedelta object of  
30 days; the weekday() method returns an integer  
for the day of the week, with Monday = 0.

# 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.