Chapter 17
Introduction to Files, Paths and IO

17.1 Introduction

The operating system is a critical part of any computer systems. It is comprised of elements that manage the processes that run on the CPU, how memory is utilised and managed, how peripheral devices are used (such as printers and scanners), it allows the computer system to communicate with other systems and it also provide support for the file system used.

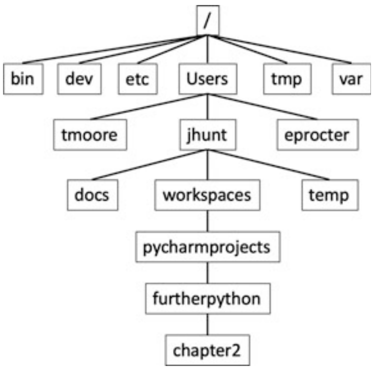
The File System allows programs to permanently store data. This data can then be retrieved by applications at a later date; potentially after the whole computer has been shut down and restarted.

The File Management System is responsible for managing the creation, access and modification of the long term storage of data in files.

This data may be stored locally or remotely on disks, tapes, DVD drives, USB drives etc.

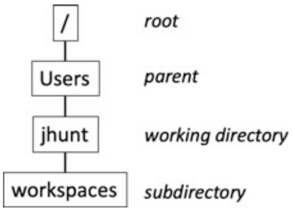
Although this was not always the case; most modern operating systems organise files into a hierarchical structure, usually in the form of an inverted tree. For example in the following diagram the root of the directory structure is shown as '/'. This `root` directory holds six subdirectories. In turn the `Users` subdirectory holds 3 further directories and so on:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_17



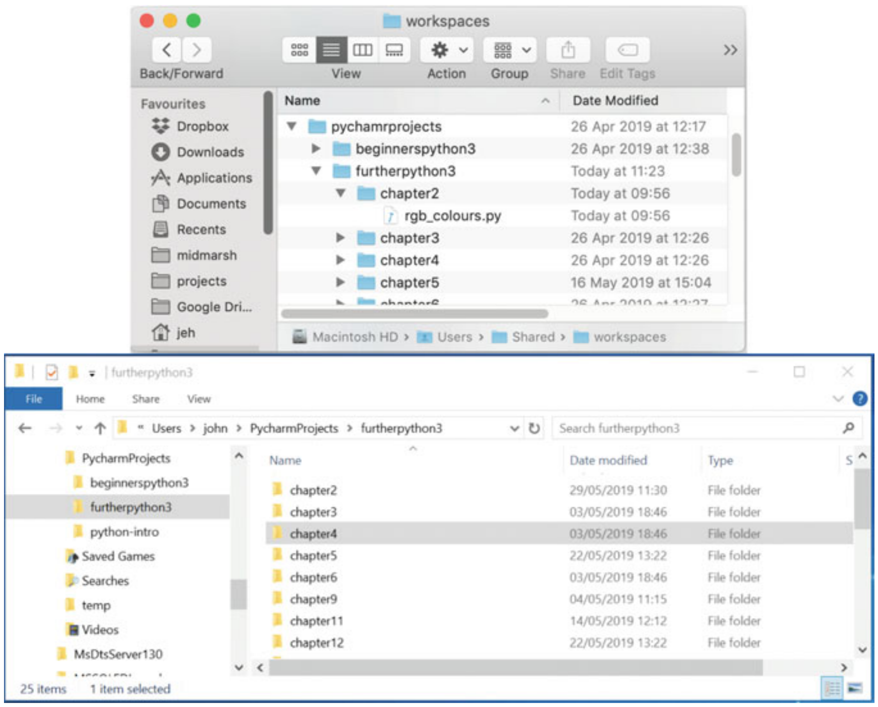
Each file is contained within a directory (also known as a folder on some operating systems such as Windows). A directory can hold zero or more files and zero or more directories.

For any give directory there are relationships with other directories as shown below for the directory `jhunt`:



The `root` directory is the starting point for the hierarchical directory tree structure. A child directory of a given directory is known as a subdirectory. The directory that holds the given directory is known as the parent directory. At any one time, the directory within which the program or user is currently working, is known as the *current working directory*.

A user or a program can move around this directory structure as required. To do this the user can typically either issue a series of commands at a terminal or command window. Such as cd to change directory or pwd to print the working directory. Alternatively Graphical User Interfaces (GUIs) to operating systems usually include some form of file manager application that allows a user to view the file structure in terms of a tree. The Finder program for the Mac is shown below with a tree structure displayed for a `pycharmprojects` directory. A similar view is also presented for the Windows Explorer program.



Name	Date Modified
pychamrprojects	26 Apr 2019 at 12:17
beginnerspython3	26 Apr 2019 at 12:38
furtherpython3	Today at 11:23
chapter2	Today at 09:56
rgb_colours.py	Today at 09:56
chapter3	26 Apr 2019 at 12:26
chapter4	26 Apr 2019 at 12:26
chapter5	16 May 2019 at 15:04

Name	Date modified	Type
chapter2	29/05/2019 11:30	File folder
chapter3	03/05/2019 18:46	File folder
chapter4	03/05/2019 18:46	File folder
chapter5	22/05/2019 13:22	File folder
chapter6	03/05/2019 18:46	File folder
chapter9	04/05/2019 11:15	File folder
chapter11	14/05/2019 12:12	File folder
chapter12	22/05/2019 13:22	File folder

17.2 File Attributes

A file will have a set of attributes associated with it such as the date that it was created, the date it was last updated/modified, how large the file is etc. It will also typically have an attribute indicating who the owner of the file is. This may be the creator of the file; however the ownership of a file can be changed either from the command line or through the GUI interface. For example, on Linux and Mac OS X the command `chown` can be used to change the file ownership.

It can also have other attributes which indicate who can read, write or execute the file. In Unix style systems (such as Linux and Mac OS X) these access rights can be specified for the file owner, for the group that the file is associated with and for all other users.

The file owner can have rights specified for reading, writing and executing a file. These are usually represented by the symbols ‘r’, ‘w’ and ‘x’ respectively. For example the following uses the symbolic notation associated with Unix files and indicates that the file owner is allowed to read, write and execute a file:

```
-rwx------
```

Here the first dash is left blank as it is to do with special files (or directories), then the next set of three characters represent the permissions for the owner, the following set of three the permissions for all other users. As this example has `rwx` in the first group of three characters this indicates that the user can read ‘r’, write ‘w’ and execute ‘x’ the file. However the next six characters are all dashes indicating that the group and all other users cannot access the file at all.

The group that a file belongs to is a group that can have any number of users as members. A member of the group will have the access rights as indicated by the group settings on the file. As for the owner of a file these can be to read, write or execute the file. For example, if group members are allowed to read and execute a file, then this would be shown using the symbolic notation as:

```
----r-x---
```

Now this example indicates that only members of the group can read and execute the file; note that group members cannot write the file (they therefore cannot modify the file).

If a user is not the owner of a file, nor a member of the group that the file is part of, then their access rights are in the ‘everyone else’ category. Again this category can have read, write or execute permissions. For example, using the symbolic notation, if all users can read the file but are not able to do anything else, then this would be shown as:

```
-------r--
```

Of course a file can mix the above permissions together, so that an owner may be allowed to read, write and execute a file, the group may be able to read and execute the file but all other users can only read the file. This would be shown as:

```
-rwxr-xr--
```

In addition to the symbolic notation there is also a numeric notation that is used with Unix style systems. The numeric notation uses three digits to represent the permissions. Each of the three *rightmost* digits represents a different component of the permissions: owner, group, and others.

Each of these digits is the sum of its component bits in the binary numeral system. As a result, specific bits add to the sum as it is represented by a numeral:

- The read bit adds 4 to its total (in binary 100),
- The write bit adds 2 to its total (in binary 010), and
- The execute bit adds 1 to its total (in binary 001).
- This the following symbolic notations can be represented by an equivalent numeric notation:

Symbolic notation	Numeric notation	Meaning
`rwx——`	`0700`	Read, write, and execute only for owner
`-rwxrwx—`	`0770`	Read, write, and execute for owner and group
`-rwxrwxrwx`	`0777`	Read, write, and execute for owner, group and others

Directories have similar attributes and access rights to files. For example, the following symbolic notation indicates that a directory (indicated by the 'd') has read and execute permissions for the directory owner and for the group. Other users cannot access this directory:

```
dr-xr-x---
```

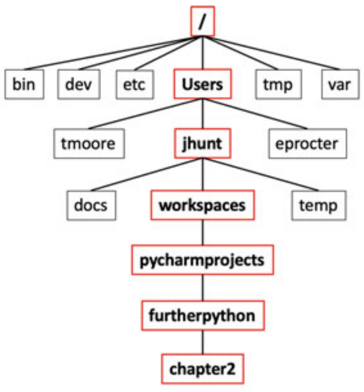
The permissions associated with a file or directory can be changed either using a command from a terminal or command window (such as chmod which is used to modify the permissions associated with a file or directory) or interactively using the file explorer style tool.

17.3 Paths

A path is a particular combination of directories that can lead to a specific sub directory or file.

This concept is important as Unix/Linux/Max OS X and Windows file systems represent an inverted tree of directories and files., It is thus important to be able to uniquely reference locations with the tree.

For example, in the following diagram the path `/Users/jhunt/workspaces/pycharmprojects/furtherpython/chapter2` is highlighted:



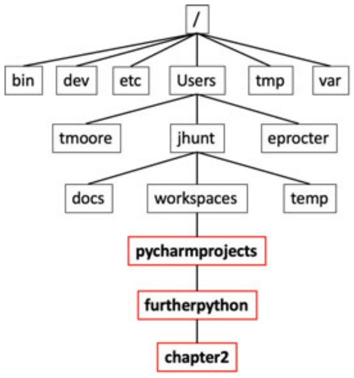
Path: /Users/jhunt/workspaces/pycharmprojects/furtherpython/chapter2

A path may be absolute or relative. An absolute path is one which provides a complete sequence of directories from the root of the file system to a specific sub directory or file.

A relative path provides a sequence from the current working directory to a particular subdirectory or file.

The absolute path will work wherever a program or user is currently located within the directory tree. However, a relative path may only be relevant in a specific location.

For example, in the following diagram, the relative path pycharmprojects/furtherpython/chapter2 is only meaningful relative to the directory workspaces:



Relative path: pycharmprojects/furtherpython/chapter2

Note that an absolute path starts from the root directory (represented by '`/`') where as a relative path starts from a particular subdirectory (such as `pychamprojects`).

17.4 File Input/Output

File Input/Output (often just referred to as File I/O) involves reading and writing data to and from files. The data being written can be in different formats.

For example a common format used in Unix/Linux and Windows systems is the ASCII text format. The ASCII format (or American Standard Code for Information Interchange) is a set of codes that represent various characters that is widely used by operating systems. The following table illustrates some of the ASCII character codes and what they represent:

Decimal code	Character	Meaning
42	*	Asterisk
43	+	Plus
48	0	Zero
49	1	One
50	2	Two
51	3	Three
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D

(continued)

(continued)

Decimal code	Character	Meaning
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d

ASCII is a very useful format to use for text files as they can be read by a wide range of editors and browsers. These editors and browsers make it very easy to create human readable files. However, programming languages such as Python often use a different set of character encodings such as a Unicode character encoding (such as UTF-8). Unicode is another standard for representing characters using various codes. Unicode encoding systems offer a wider range of possible character encodings than ASCII, for example the latest version of Unicode in May 2019, Unicode 12.1, contains a repertoire of 137,994 characters covering 150 modern and historic scripts, as well as multiple symbol sets and emojis.

However, this means that it can be necessary to translate ASCII into Unicode (e.g. UTF-8) and vice versa when reading and writing ASCII files in Python.

Another option is to use a binary format for data in a file. The advantage of using binary data is that there is little or no translation required from the internal representation of the data used in the Python program into the format stored in the file. It is also often more concise than an equivalent ASCII format and it is quicker for a program to read and write and takes up less disk space etc. However, the down side of a binary format is that it is not in an easily human readable format. It may also be difficult for other programs, particularly those written in other programming languages such as Java or C#, to read the data in the files.

17.5 Sequential Access Versus Random Access

Data can be read from (or indeed written to) a file either sequentially or via a random access approach.

Sequential access to data in a file means that the program reads (or writes) data to a file sequentially, starting at the beginning of a file and processing the data an item at a time until the end of the file is reached. The read process only ever moves forward and only to the next item of data to read.

Random Access to a data file means that the program can read (or write) data anywhere into the file at any time. That is the program can position itself at a particular point in the file (or rather a pointer can be positioned within the file) and it can then start to read (or write) at that point. If it is reading then it will read the next data item relative to the pointer rather than the start of the file. If it is writing data then it will write data from that point rather than at the end of the file. If there is already data at that point in the file then it will be over written. This type of access is also known as Direct Access as the computer program needs to know where the data is stored within the file and thus goes directly to that location for the data. In some cases the location of the data is recorded in an index and thus is also known as indexed access.

Sequential file access has advantages when a program needs to access information in the same order each time the data is read. It is also is faster to read or write all the data sequentially than via direct access as there is no need to move the file pointer around.

Random access files however are more flexible as data does not need to be written or read in the order in which it is obtained. It is also possible to jump to just the location of the data required and read that data (rather than needing to sequentially read through all the data to find the data items of interest).

17.6 Files and I/O in Python

In the remainder of this section of the book we will explore the basic facilities provided for reading and writing files in Python. We will also look at the underlying streams model for file I/O. After this we will explore the widely used CSV and Excel file formats and libraries available to support those. This section concludes by exploring the Regular Expression facilities in Python. While this last topic is not strictly part of file I/O it is often used to parse data read from files to screen out unwanted information.

17.7 Online Resources

See the following online resources for information on the topics in this chapter:

- https://en.wikipedia.org/wiki/ASCII Wikipedia page on ASCII.
- https://en.wikipedia.org/wiki/Unicode Wikipedia page on Unicode.
- https://en.wikipedia.org/wiki/UTF-8 Wikipedia page on UTF-8.