

Some notes on file names

Michael Lundholm*

June 14, 2011

Version 1.2

1 Introduction

That research data, methods and results at some stage of the research process are made available to others (for possible replication) is at the core of good scientific practice. To communicate the results of research, obviously, researchers have to have a common language in which they write and speak. Historically, this function was filled by Latin but today the *lingua franca* (or *vehicular language*) of science is English. However, researchers often have a need to communicate more than just research reports and speak to each other at local seminars and international conferences.

Frequently, communication in terms of computer software code is necessary. Exactly as written texts have to be authored in a common language to be understood this more basic communication in the form of *files* need to conform to some standard in order to be understood by different computer operating systems (say, Windows, Unix and GNU/Linux). This requires at least some basic knowledge about the used software and the computers' operating systems. In particular how file names are constructed under different operating systems.

At the close of the first decade of the 21st century, 36 years after the Xerox Corporation introduced the first personal computer or PC (the Xerox Alto) and 28 years after IBM introduced the IBM PC 5150 (which became the *de facto* PC standard), most individuals have at least a smattering how to use PC's. In particular PC's using some variant of the Microsoft Windows operating system (say, Windows XP or Vista or more recently Windows 7). However, the ordinary Windows user seldom has any knowledge about other operating systems. But this is not the complete story. UNIX and GNU/Linux users seldom pay any attention what the idiosyncracies of their own operating systems have when communicating with (say) Windows users.

The shallow knowledge about Microsoft Windows most individuals have therefore need to be complemented. Maybe, some insights may be gained

*Department of Economics, Stockholm University, michael.lundholm@ne.su.se.

by UNIX and GNU/Linux users as well. These notes will discuss *file names*, what they are and how they should be constructed in order to facilitate communication of computer software code between users of different operating systems; e.g., R users on GNU/Linux and some variant of Windows

2 File names

A file name is a string of characters that uniquely identifies the file. There are several logical components of a file name, the most important of which (for our purposes) are the following:

device The root mountpoint, disc or volume where the file resides on the computer.

path The directory tree, which relative to some point of reference on the device gives the location of the file.

file The base name of the file.

extension The suffix of the file name, very frequently indicating the content of the file.

The device states in which part of the computer file system the file resides. In Windows the term *volume* is used to denote this. A volume could be a physical disk named C: or D: etc. Exactly which letters that are used may vary from one individual PC running Windows to another. Sometimes physical disks are partitioned into several logical disks, where each logical disk serves as a volume. There is no common root directory in Windows, but each mount point has its own root directory denoted by \ (back slash) or sometimes / (forward slash). The root directory on the C: volume is hence denoted C:\. Where Windows systems may have many volumes only one root mount point exists on UNIX and GNU/Linux systems.

The term *path* can have at least two meanings. Here it is used in the meaning of a directory tree, which gives the location of the file relative to a point of reference. The path can be *absolute* or *relative* depending on which type of point of reference that is used. If it's the absolute path it states the file location relative to the device (i.e., volume or the root mountpoint). If it's a relative path it states the location of the file relative to the *working directory*. The meaning of working directory as well as the alternative meaning of path are discussed in section 4.

The path is a sequence of individual directories. A directory is basically a file which contains other directories and files, so what in section 3 below is said about file names is also applicable to directory names.

The directories in the path are separated by a character which is called the *directory separator*. Which character or characters that may be used

depend on the operating systems. On UNIX and GNU/Linux the directory separator is / (forward slash) but on Windows \ (back slash) is the standard although also / (forward slash) can be used.¹

Let `dir1`, `dir2` and `dir3` be directory names. On Unix and GNU/Linux computers a directory structure would therefore be `dir1/dir2/dir3/...`. On a Windows PC the standard alternative is `dir1\dir2\dir3\...`, although `dir1/dir2/dir3/...` also will work. The forward slash can also be used instead of the back slash to denote a root directory.

The file base name and the file extension in conjunction are often what we think of as the *file name*. These are written together separated only by a dot (`.`), say `filename.ext` where `filename` is the base name and `ext` is the extension.

The different operating systems are treating the base name and the extension differently. On Unix and GNU/Linux systems the extension is just a part of the regular file name and the dot has no specific meaning as a delimiter between them. Instead, the dot is a just a character in the file name. In DOS/Windows the extension is treated as a different category (another *name space*). The *single* dot between the base name and extension has a function similar to the directory separator. Double dots and dots in other positions than between base name and extension should therefore be avoided (see below).

If we want to give the complete filename with an absolute path to a file `filename.ext` in a certain directory this is done as follows on Windows and Unix and GNU/Linux systems:

- `C:\dir1\dir2\dir3\filename.ext` (Windows, standard)
- `C:/dir1/dir2/dir3/filename.ext` (Windows, alternative)
- `/dir1/dir2/dir3/filename.ext` (Unix and GNU/Linux)

One insight from this is therefore that when computer code is distributed to others for (say) replication and similar purposes it is never a good idea to use *absolute paths* to files. Instead use *relative paths*, where the point of reference is the working directory (see below) with use of the / (forward slash).

3 Basic conventions for file names

In what follows I have collected some basic rules, that, if followed, will avoid a lot of trouble. These rules try to find the common denominator between the

¹Other possibilities for directory separators exists in Windows due to the special meaning that the back slash has as an escape character (i.e., a character that gives a special meaning to the character following than its ordinary meaning). Also, in some software, say \LaTeX , this means that the forward slash *must* be used as a directory separator even on Windows systems.

operating systems. Not only will communication with others be facilitated, but you will also avoid problems with just using your computer.

How should one choose file names? This question can be divided into two different but related questions:

1. How many characters are allowed in the basename and in the extension?
2. Which characters are allowed in the basename and in the extension?

3.1 How many characters are allowed?

The first question is the easiest to answer. Modern operating systems (including Windows XP and after, UNIX and GNU/Linux and Mac OS X) allow for up to 255 characters in the file name. Historically, however, several operating systems have allowed considerably fewer characters. One example is the original DOS for IBM PC's (including Windows up to version 3.1) which allowed only 12 characters.

The 12 character case is called “8.3” file names. With the base name having at the most 8 characters, the extension at the most 3 characters the user had up to 11 characters to use. The mandatory dot between base name and extension being the twelfth character. It was not until 1994 with Windows NT that Microsoft introduced LFN (Long Filenames).²

To achieve backward compatibility to older Windows versions and DOS, an algorithm for creating “8.3” file names from LFN was established. As an example the LFN `LONGFILENAME.TEXT` would become the “8.3” file name `LONGFI~1.TEX`. If additional files with the same first 6 characters and the same 3 character extension are created, then they would be given the “8.3” file names `LONGFI~2.TEX` etc.

Since this conversion algorithm is still used even on Windows Vista, it affects the choice of filenames. Some of these aspects of the conversion will be discussed in the next subsection. As a general conclusion, however, one can use filenames that are longer than “8.3” file names as long as (for obvious reasons) one do not have too many files with the same extension that share the first 6 characters.

What has been said here about LFN and “8.3” file names refer to the default behaviour. It can be altered by the user.³

3.2 Which characters are allowed?

Characters can be organised in different ways in so called character set such as ASCII (American Standard Code for Information Interchange) and Uni-

²It may be a coincidence that LFN is also the acronym for *Lingua Franca Nova*, but this vehicular language was not published on the Internet until 1998.

³See [http://msdn.microsoft.com/en-us/library/aa365247\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa365247(VS.85).aspx).

Table 1: The 95 printable ASCII characters (decimal codes 32-126).

␣	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	

code (ISO 10646 Universal Character Set). Most operating systems nowadays do not restrict the choice of character set for file names on that operating system.

However, this is not the same as that all characters can be used on all operating systems or that all characters used on one operating system can be used on another. Instead, on individual operating system some characters are reserved for special purposes and should not be used in file names. Using these characters on other operating systems where they are allowed may cause problems when files are moved to the operating systems where they are not allowed.

It turns out that good practice is to select characters for file names from the 7-bit ASCII character set (or rather a subset of it). It consists of $2^7 = 128$ different characters represented by numbers (decimal code) 0-127. Modern encodings, such as (say) 8-bit ASCII and UTF-8 would allow for more characters. However, *do not use these additional characters in file names*. It might be so that the characters easily accessible on your national keyboard are not so easy to access for user with another national keyboards even if you are using the the same operating system and therefore these characters are formally accepted on both computers. Another reason is that some software will not accept file names with characters not from the 7-bit ASCII encoding.

But you should not use all characters available in the 7-bit ASCII. Here comes a list of the characters that should be avoided:

1. Some of the 128 7-bit ASCII characters are non-printable and should therefore be excluded. These have decimal code 000-032 and 127. The remaining 95 printable characters are shown in Table 1. Note that the glyph `␣` denotes white space.
2. We then immediately see, from the discussion in section 2, that three characters have a reserved use in Windows and/or UNIX and GNU/Linux (the forward slash (/) with decimal code 047, the back slash (\) with decimal code 092 and the dot (.) with decimal code 046) as directory separators and separator between base name and extension). Also the Mac OS X has some restrictions: the dot (.) is forbidden in the begin-

ning of file names and the forward slash (/) is now allowed by some applications. The dot and the two slashes should not be part of the base name or the extension. Now remain “only” $95 - 3 = 92$ printable ASCII characters for use in file names.

3. The introduction of LFN mentioned above also had consequences for which characters that should be used. The reason is that originally case insensitive DOS and Windows interpreted all (upper and lower case) letters as upper case letter (capitals). With the LFN file names with lower case letters was allowed as LFN but transformed to “8.3” file names converted to upper case. For instance, the LFN `FileName.Txt` becomes the “8.3” `FILENAME.TXT` and the LFN `longfilename.text` becomes `LONGFI~1.TEX`.

Still, however, Windows is case-insensitive in the sense that that equal length file names, which contains less or equal to 8 plus 3 characters, with different capitalisation are regarded as the same.⁴ Therefore, users on other operating systems cannot assume that case sensitivity is honoured. See the Appendix for an example of the potential problems.

The conclusion is that the upper case letters A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y and Z (decimal codes 065-090) should not be used. “Do not assume case sensitivity.”⁵ This leaves $92 - 26 = 66$ characters left to use in file names.

4. The following characters are prohibited for various reasons. These reasons varies, but basically the characters have some special meaning in some operating systems (say, *wild cards* as `?`, `*` and `%`). By excluding them we avoid problems when files are transferred between operating systems:

Windows, Unix and GNU/Linux: `" > < |` (decimal codes 034, 060, 062 and 124)

Windows and OS X: `:` (decimal code 058)

Windows: `* ?` (decimal codes 042 and 063)

Unix and GNU/Linux: `_ ! # $ & ' () + , ; = @ [] ^ ` { } ~`
(decimal codes 032-033, 034-035, 038-041, 043-044, 059, 061, 064, 091, 093-094, 096, 123, 125-126)

Other OS's: `%` (decimal code 037)

This means that we have $66 - 28 = 38$ characters left to use in file names.

⁴See for instance http://blogs.msdn.com/brian_dewey/archive/2004/01/19/60263.aspx and <http://msdn.microsoft.com/en-us/library/aa365247.aspx>.

⁵[http://msdn.microsoft.com/en-us/library/aa365247\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa365247(VS.85).aspx).

The characters that remain for use in file names are then the following:

Numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (decimal codes 048-057)

Lower case letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z (decimal codes 097-122)

Other characters: - _ (decimal codes 045 and 095).⁶

4 Search path, working directory and parent directory

As noted above (see page 2) the term *path* has two meanings. There we used the term in the meaning of a directory tree. Another meaning is as a set of directories which the operating system searches for executables that the user tries to invoke; i.e., *search path*. This has importance when the user of computer executes a command.

Consider a computer user who is issuing a command from the command prompt. Most frequently this is done by providing a command, that is the filename of the executable file, with no directory tree. This means that the command is given *as if* the executable file was residing in the directory from which the command was given. This directory is called the *working directory*.

If the computer does not find the executable in the working directory it searches some parts of the file system to find the executable. The set of directories where it will search is called the (search) path. If the executable is not found in either the working directory or in the search path there will be some error message.

The same will not necessarily occur when some executable is looking for code or data file. If the location of a code file or data file is given without its directory tree, then it is presumed that the file resides in the working directory. Normally, the search path is not searched for the file.

When software with a graphical user interfaces are used one has to check how they behave in terms of working directory. Text editors usually change the working directory to the directory of the edited file. Editing a new file then automatically changes the working directory. Other software do not change the working directory unless explicitly told to do so. One example of this is R when run under Windows.⁷

All directories except the root directory have a parent directory. Given that the user is in some directory the general notation for the parent directory

⁶- (hyphen–minus) is also called minus, hyphen and dash. _ (underscore) is also called understrike, low line and low dash.

⁷See R Development Core Team (2008). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL <http://www.R-project.org>.

is a double dot (`..`). This is always relative to the working directory and explains why double dots should be used as part of a file name. A single dot (`.`) normally indicates the working directory.

Appendix

Consider the following example: A student is using a Windows XP system and is creating a data file with name `data.r` as a home assignment. The student is running an R session where the data file is called at three different instances. A little bit careless, or maybe just uninformed, the student uses a different capitalisation at each instance (other code is omitted):

```
> source(file="data.r")
> source(file="Data.r")
> source(file="daTa.r")
```

On this Windows XP system the student does not get any error messages, although only the first time that the file was called there was a “perfect” match between the LFN transformed to “8.3” file name on the computer and the capitalisation of the file name in the call from R. The reason was that in all three instances Windows interpreted the requested file names and also the existing file as `DATA.R`. They (the three calls and the LFN transformed to “8.3” file name) were all regarded as equivalent by the operating system. Happy with the result the student now submits the code and the data file to the teacher. Using R on GNU/Linux, the teacher runs the code with the following disastrous result:

```
> source(file="data.r")
> source(file="Data.r")
Error in file(file, "r", encoding = encoding) :
  cannot open the connection
In addition: Warning message:
In file(file, "r", encoding = encoding) :
  cannot open file 'Data.r': No such file or directory
> source(file="daTa.r")
Error in file(file, "r", encoding = encoding) :
  cannot open the connection
In addition: Warning message:
In file(file, "r", encoding = encoding) :
  cannot open file 'daTa.r': No such file or directory
```

This example shows that the R code called for seemingly three different files (due to different capitalisations of file names). On the Windows system these calls were interpreted as if calling for the same file and since such a file also existed there was no error messages. But on the GNU/Linux system the calls

were interpreted as if calling for three different files, only one of which existed (i.e., the file submitted by the student with only lower case letters). The problem for the teacher is of course to know whether the student actually intended to call for three different files or not.

The example could be reversed: The teacher using GNU/Linux supplies Windows using students with three files: `data.r`, `Data.r` and `daTa.r`. On the teacher's GNU/Linux PC everything works fine, but on the students' Windows PCs it will be impossible to save the three files under these names.