The Most in the Shell Statistics 650/750 Week 1 Tuesday

Christopher Genovese and Alex Reinhart

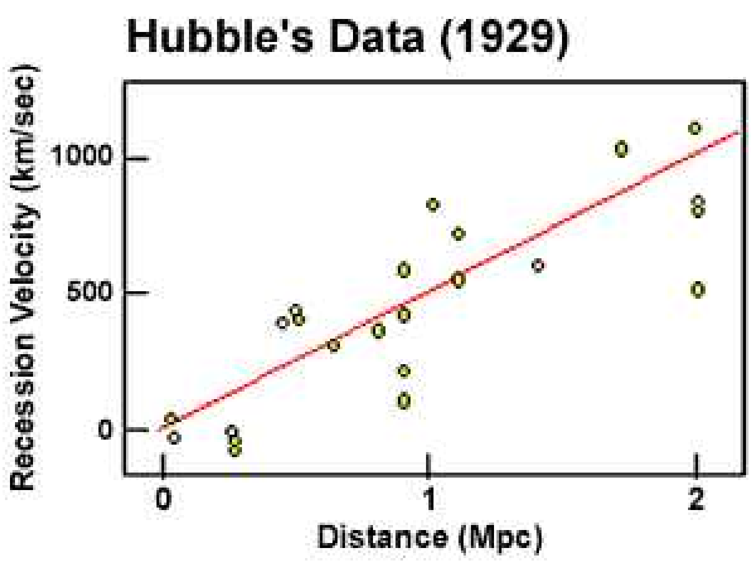
28 Aug 2018

Welcome and Introductions

Motivations

Modern data analysis can be a complex business

- Not too long ago:



- Now: cleaning and processing the data, performing analyses, and visualizing/summarizing the results can all require complicated logic and algorithms.

Creating good software to manage this complexity has become an essential skill for statisticians.

Computing is taught *at the margins* in most statistics curricula

- Typical statistical computing courses focus on the details of methods and algorithms for various concrete problems.
- Students are expected to learn the practice of computing and software engineering implicitly during their research.
- Typical feedback and incentives (e.g., get the paper done, conference deadlines, what are the results?) can obscure the benefits of building good software.

Development cycle for research software is hard to get right

1. Considering many approaches/methods, some of which are dead ends.
2. The code starts out rough and quick:
   - Assumptions about the data and algorithm get baked in
   - Meaning and documentation are sparse
   - Structure and design are secondary to "getting it working"
   - There are working examples but few distinctive tests
3. Once the paper is out, it's time to move on...
   - Build on existing code base despite flaws
   - If it's used, extensions build on top of that edifice, possibly for some time.

This limits correctness, clarity, and reusability of the code.

Building efficient, elegant, reusable software increases our productivity

Good software engineering emphasizes:

- Managing complexity
- Communicating clearly

- Finding effective abstractions
- Crafting well chosen solutions to problems
- Obtaining good performance, reuse, and generalizability

Sound familiar?

Programming well is lots of fun

It involves problem solving, design, creativity, making cool tings happen… great stuff.

Main Themes of the Course

Theme #1: Good programming practice

- Will future you (weeks or months or years from now) have a clue what your code does?
- Will a collaborator?
- Are the names of variables, functions, and classes meaningful?
- Is the code formatted in a consistent and readable manner?
- Is the code well documented? Does the documentation match the code?
- What is the role of comments? What makes a comment helpful? When should the code speak for itself?
- Can you *read* the code? Can you tell what the algorithm is?
- Does the organization of the code help the reader understand the code's purpose?
- How do we achieve the proper balance between error checking, readability, and performance? What determines where errors are "handled"?
- How do we ensure correctness of our programs?

Theme #2: Good tools and Efficient workflows

Good programming practices can make you more productive because it is easier to write correct, reusable, and generalizable code. But it also helps to use good tools.

Essentials:

- A good editor (*Emacs*, VSCode, Vim, Sublime, ...) or IDE (RStudio, Eclipse, IPython)
- Version control (e.g., git) for managing changes and collaborative development
- Debugger for finding tricky bugs
- Profilers for performance tuning
- Databases for managing complex data sets
- Testing framework to easily run and check your tests
- Documentation format for writing/disseminating documentation on your code

On Editors:

- We strongly recommend that you learn to use a good *editor*; it will make you more productive in several ways.
- We have two strong recommendations:

    1. Emacs
    2. Visual Studio Code

  Emacs is more powerful and extensible but has a steeper learning curve. VS Code has many nice built-in features and is quicker to get started with.

- We will provide help in getting started with either.

Mac users: Installing homebrew is highly recommended.

Theme #3: Good software design

- Does your code repeat itself frequently?
- If you change one part of the code, what else needs to change?
- Does each function or object only have access to the information it needs?
- How easy is it to reuse parts of your code?
- How easy is it to adjust the code to solve a more general problem?
- How easy is it to reason about the structure of your program?

Theme #4: Good choice of representations, data structures, and algorithms.

- How does the runtime (or memory...) of your program scale as the size of the problem increases?
- Does your representation of the problem allow for an elegant/efficient solution?
- Does your data representation match well with the types of operations you are doing?
- What are the performance characteristics of the algorithm you are using?
- Can you use a well-tested library or package to run the algorithm?

Theme #5: Killer Apps!

Putting the pieces together to solve interesting and challenging statistical problems.

Examples:

- Identifying audio files from small snippets
- Estimating object velocities from images
- Spell checking and text completion
- Searching spatial and high-dimensional data sets

- Training agents to play interactive games
- Classifying images and identifying objects
- and many more…

Course Design Highlights

What we believe

- A broad and firm foundation in computing will pay off throughout your career
- The way to get better at programming is to **practice** programming
- Good software design and programming practice are skills every statistician needs
- Revision is a critical part of the development process
- Having (at least a passing) understanding of multiple languages will make you a better programmer

Learning Objectives

By the end of this course, you should be able to

- develop correct, well-structured, and readable code;
- design useful tests at all stages of development;
- effectively use development tools such as editors/IDEs, debuggers, profilers, testing frameworks, and a version control system;
- build a moderate scale software system that is well-designed and that facilitates code reuse and generalization;
- select algorithms and data structures for several common families of statistical and other problems;
- write small programs in a language new to you.

Class Activities

Classes will feature a combination of lectures, interactive discussion and problem-solving, and single/group programming activities.

You should bring your laptop to every class

Materials

- Most course materials and course work will be accessed through `github`. We also have a canvas page through CMU, primarily for announcements, grade book, and copies of some documents.

  Key Steps:

  1. Get free account `https://github.com/`
  2. Visit `https://classroom.github.com/a/MMfm3qKz`
  3. Find syllabus in the `documents` repository, under the `Info` folder. `https://github.com/36-750/documents/`
  4. Read the Syllabus!!!

  Three special repositories (besides your own homework repository) that you will have access to:

`https://github.com/36-750/documents`	Announcements, lecture notes, data, and other
`https://github.com/36-750/problem-bank`	Assignment descriptions and associated files anc
`https://36-750.github.io/`	Lecture notes and documents in easy to access

- Details on assignments are spelled out in the syllabus. **Read the Syllabus**

- Several useful language-specific books are available for you to borrow. A list of this is available in the `documents` repository and are available from the instructors. *Do not hesitate to ask for them.*

- The instructors will be available to meet to answer your questions. Scheduling of office hours will be data driven.

- You will interact with the TAs mostly through discussions on github, but they will also be available to answer questions about their feedback. They may hold fixed hours or be available by appointment depending on need.

Tips

- It's OK to make mistakes.
- Try to find ways to expand your skills and perspectives.
- Ask for help if you need it, from us and your peers.
- Pay attention to the rubric.
- Practice, revise, repeat.
- Challenge yourself.

Request

Please put `[650]` or `[750]` in your email subject lines!!

Academic Integrity

Acceptable collaboration or use of external sources includes:

- Clarifying ambiguities, errata, or vague points in class materials or assignments.
- Discussing or explaining the general class material.
- Providing assistance with system facilities, computing tools, or online interfaces.
- Discussing the assignments to better understand what is being asked.
- Looking up background material (online or in books) on general concepts discussed in class.
- Discussing general approaches to solving specific problems, though see below.

Unacceptable collaboration or use of external sources includes:

- Copying of another student's solution to a problem (in part or whole) or obtaining a solution from an outside source (including a similar or related problem in part or whole).
- Allowing someone else to copy your solution in part or whole.

- Receiving help from students who have taken this or a related course in previous years.

- Communicating or having communicated (e.g., by seeing, speaking, pantomime) to you the steps of a solution.

- Reading the posted solution if you will be submitting your assignment late.

- Reviewing any course materials from this or related courses in previous years.

In general, all work must be written up individually, and no student should ask for assistance from any other student or offer assistance to any other student until that student has made a serious effort to solve the problem.

Please read the Syllabus section on academic integrity carefully.

Cheating, inappropriate collaboration, or improper use of external sources can be grounds for course failure. We may be obliged in these situations to report the incident to the appropriate University authorities. Please refer to university policies here. Feel free to come talk to us if you have any questions about this.

For Next Class:

- **Read the syllabus**

- Sign up for github and setup your class repositories

- Install R and Python 3 if needed

- Install Editor: Emacs or VSCode

Living on the Command Line

A Tale of Two Interfaces: GUI and CLI

Most computer users today interact with their computer through a **Graphical User Interface** (GUI). Such interfaces often embody a physical metaphor for the objects on the system and for the ways that users manipulate these objects. Examples: dragging a file and dropping it into the trash, sliding a panel across a touch screen.

Another approach is a **Command Line Iterface** (CLI), in which users control the system by entering a series of text *commands*.

Comparison: GUIs

Pro GUIs are easy to learn and use (when the physical metaphor is intuitive) and requires little expertise.

Pro GUIs can associate powerful operations/features to simple actions.

Con GUIs make it hard to reuse, modify, automate, or share the steps of complex/repeated tasks.

Con GUIs are relatively slow to use and usually support only limited customizability/extensibility.

Comparison: CLIs

Pro A good CLI is highly expressive and efficient to use.

Pro CLIs make it easy to reuse, modify, automate, or share the steps of complex/repeated tasks.

Pro CLIs allow combination of simple operations to handle a flexible range of complex tasks.

Con CLIs have a steeper learning curve, involving a variety of detailed concepts and some detailed patterns.

Con CLIs often use tersely named commands and obscure notation.

In this course, we will focus primarily on using CLIs to operate our tools and interact with the software we write.

Shells and Terminal Emulators

A **shell** is a program that takes text commands as input and directs the computer's operating system to carry them out. There are different shells available with slightly different features; for instance, I use `zsh`, and in this class, we will use `bash`.

When using your computer through the GUI window system, we run the shell within another program – a *terminal emulator*. For example: `xterm` on Linux, `Terminal` on Mac, `git-bash` on Windows. Starting these "terminals" opens a shell automatically.

Now, start up a shell on your computer and follow along.

You will see a **prompt** something like that below.

`bash-3.2$`

Enter a command and hit return, like

```
echo "Hello, world"
```

Next, try commands like `date`, `whoami`, or `cal`. Each command prints its output on the lines following the prompt and then a new prompt is given.

This pattern – called a **Read-Evaluate-Print Loop** or **REPL** – is the same as what you get when running R or Python interactively; only the nature and syntax of the commands is different.

Shell commands are used to control the operating system and to run other programs.

Navigating the File System (pwd, cd, ls, file, cat, more/less)

The File and Directory Tree

The **files** on your computer are arranged in a hierarchical directory structure. **Directories** (aka folders) contain files and other directories, which in turn contain files and other directories, and so on. The files are thus arranged in a *tree*, and at the *root* of that tree is the **root directory**.

(Note: directories are actually a special type of file listing information about the files "contained" in the directory.)

The Current Working Directory

At any point in time when the shell is running, it keeps track of the directory where you are currently working. Unsurprisingly, this is called the **current working directory**.

Type the command `pwd` at the shell prompt. This stands for "print working directory".

You will see output with a form something like this:

```
/Users/genovese/class/s750
```

This is called a **pathname** for the directory.

Pathnames

Since the files are arranged in a tree, we can uniquely specify a file by describing the **path** from the root directory to the file.

Let's breakdown the pathname `/Users/genovese/class/s750`.

1. The initial `/` denotes the root directory.
2. `Users` is a directory contained within the root directory.
3. `genovese` is a directory within `Users`.
4. `s750`, which is within `genovese`, is the current working directory.

This gives the path from root to the file of interest.

A pathname starting with the root `/` is called an **absolute path**. We can also specify a **relative** path, which is defined relative to the current directory.

Example relative pathnames (on my machine) with that same working directory:

1. `entrypoints`

   A file in the current working directory, with absolute path `/Users/genovese/class/s750/entrypo`

2. `course-materials/lectures/week1/week1T.org`

   The file for this lecture; the absolute pathname is `/Users/genovese/class/s750/course-materia`

3. `./src/scomp-exercise-mode.el`

   In pathnames, `.` (a single dot) is a special notation for the current directory, and `..` (two dots) is a special notation for the *parent* of the current directory.

   This represents the file with absolute path `/Users/genovese/class/s750/src/scomp-exercise-m`

4. `../218/info/syll.pdf`

   A file obtained by first moving up *towards the root* and then down a sibling branch of the tree.

Listing Files: The ls command

From within your current directory, type the command `ls`.

I see the following output:

```
2018-notes           documents        problem-bank-source
NOTES                entrypoints      resources
Project-Links        misc             roster
course-materials     old              src
dev                  problem-bank     style
```

You will see a similar listing. These are the files contained within your current working directory.

Now, we will add an **option** to the command to change it's behavior. Type `ls -l`. (There is a space between the ls and the -l, that is "minus ell".)

This is a long listing, indicating a variety of information about the files including access permisions, owner, modification time, and name.

Next, type `ls -lrtF`. You will see two differences: the ordering of the files – which should now be in chronological order – and a special character is attached to the name depending on file type, e.g., `/` for directories.

Note: options are specified by an initial `-`. Short (one character) option names, like `-l` or `-F`, can often be combined into one string. You will get the same thing if you type `ls -l -r -t -F`.

Next, pick the name of one directory shown in the previous listing and type `ls DIR` where you replace DIR with that directory name. For instance, I typed `ls resources`. What do you see? Here, `resources` is an *argument* to the command. The `ls` command can take any number of pathnames as arguments.

Finally, try combining some options and arguments in `ls` command.

Changing Directories: The cd command

We can change the working directory with the `cd` ("change directory") command. In its most common usage, it takes a *single directory pathname* as an argument and changes the current working directory to the given one.

Try `cd ..` to move to the parent directory. Follow that with a `pwd` and `ls` to look around at your new location.

Try a few more `cd` commands, changing your working directory up or down as you see fit. You can give `cd` either relative or absolute pathnames; try some of each.

If you type `cd` with no arguments, it moves you to a special directory called your **home directory**, which is the root of the subtree containing the files you "own". We will play with that shortly.

Examining File Contents: file, cat, and more or less.

So far, we've seen the names of the files and various metadata about them. We are usually more interested in their *contents*.

There are various commands to examine file contents; here are two that are especially good for text files.

The `cat` command (short for concatenate) prints out the contents of all files given as arguments (in order).

If I type `cat entrypoints NOTES` from my previous working directory, it prints the concatenated contents of both files in that order to the screen. But `cat` has a few other tricks up its sleeve.

The `file` command outputs a description of the file type for any pathnames given as arguments. Try it with `.` and `..` and one other non-directory file as arguments.

Use `ls -lh` and the `file` command to find one or more smallish (text) file in your directory tree, and `cat` their contents to the screen. For example, when I type `file entrypoints`, I get the output

```
entrypoints: ASCII text
```

telling me it is a text file, and `ls -lh entrypoints NOTES` displays

```
-rw-r--r-- 1 genovese staff 12K Oct 28  2017 NOTES
-rw-r--r-- 1 genovese staff 228 Oct 26  2017 entrypoints
```

telling me that the latter has size 12 kilobytes and the former 228 bytes.

If you `cat` a long file, the output will all scroll by. This has its uses, but more often you want to see the contents a little at a time. Like `cat`, the `more` and `less` commands (whichever you have) take pathnames as arguments but scroll the contents at your discretion. Try it on a longer file to see what happens; use space to scroll another page (though there are other commands).

Ex: `less ~course-materials/lectures/week1/week1T.org`.

Standard I/O: Redirection and Pipelines

"I/O" here stands for "Input/Output". The shell provides some useful tools for controlling the input and output sources of the commands we use.

By default, many of the commands we use accept *input* from our keyboard and produce *output* to our screen. For example, at the prompt, type `cat`.

Notice that nothing happens. Now type a few lines of anything, and each line you type is printed to the screen. Input to output. (Type control-D to stop the command.)

The operating system defines three *standard channels* of I/O that any command can access:

1. **Standard input**, which by default is connected to the keyboard.
2. **Standard output**, which by default is connected to the screen.
3. **Standard error**, which is also by default connected to the screen.

Many commands read their input from standard input (unless told otherwise) and write their output to standard output (unless told otherwise), using standard error to write error or warning messages.

The shell makes it easy to change the input, output, and error channels, called **redirection**.

For any `command` (with or without options and arguments) that reads from standard input and writes to standard output, we can redirect in many ways, including

- `command < input.file`

  Gets standard input from `input.file` contents rather than the keyboard.

- `command > output.file`

  Puts standard output into `output.file` rather than the screen. It overwrites the file's contents (careful) or creates the file if it does not exist.

- `command » output.file`

  Puts standard output into `output.file` rather than the screen, but appends this output to the file's existing contents. If the file does not exist, it is created before redirecting the output.

- `command | command2`

  This is called a **pipeline** (or pipe for short). The standard output of `command` is connected to the standard input of `command2`. That is, `command2` gets as input the output of `command`.

Multiple pipes can be chained together, and the first (last) command in the chain can use (input) output redirections.

With a few new command, try

```
ls -l | grep '^d' | wc -l
echo "Hello, world" | cat | tr 'a-z' 'A-Z'
```

The first counts the number of directories in the working directory. What did the second do?

Use `ls` to make sure there are no files named FOO or BAR in your current directory. (If so, make up two names that are not used.)

Consider these commands:

```
echo "Hello, world" > FOO
cat < FOO | tr 'a-z' 'A-Z' > BAR
cat BAR
rm FOO BAR
```

What do the first three commands do? Feel free to try them. (The last command deletes the file `FOO` and `BAR`.)

Interlude: Commands, Options, and Arguments:

Shell commands have a typical format

`command OPTIONS... ARGUMENTS...`

where either or both of the options and arguments may be absent.

Options are strings that begin with a '-'. Short-form options are specified by a single letter (e.g., `-F`), and long-form options are multi-character strings that begin with another '-' (e.g., `--all`).

Options themselves can take values; the values are typically given as either the next argument (e.g., `-f name`) or with an '=' (e.g., `--file=name`).

Short-form options that take no values (sometimes called "flags") can usually be concatenated with a single '-' (e.g., `-lrt`).

Arguments (which often represent filenames) can be arbitrary strings but usually cannot start with a '-' without a special option.

Several useful conventions apply to most commands:

- Commonly used options typically have both a long and short form, e.g., `ls --almost-all` and `ls -A`.
- A bare `--` (double hyphen) indicates that no more options will follow; everything after is interpreted as an argument.
- A bare `-` (single hyphen) is used as an argument (not an option) to represent standard input or standard error.

  For example, `cat A - B` outputs the contents of file `A`, then the contents of standard input, then the contents of file `B`.
- Options `--help` and `--version` give usage and version information. (Some commands still use `-h` for the former.)
- If a command *requires* arguments (not all do), then running the command with no arguments should give a short usage summary. (Usually `--help` gives more comprehensive information.)

For assignments, you will often write shell commands to run your programs, and we will ask that you follow these conventions.

Manipulating Files and Directories (cp, mv, rm, mkdir, rmdir, chmod)

These common commands help you manage your files:

Command	What it does
cp	*copy* files to new locations
mv	*move* (rename) files
rm	*remove* (delete) files
mkdir	create a new directory
rmdir	remove an empty directory
chmod	change a file's access permissions

Common forms:

`cp file newfile` copy `file` to `newfile`, overwriting existing without `-i` option.

`cp file1 file2 ... fileN dir` copy file's to directory `dir`

`mv file newfile` rename `file` to `newfile`, overwriting existing without `-i` option.

`mv file1 file2 ... fileN dir` move file's to directory `dir`

`rm file1 file2 ... fileN` remove files (with `-i` option, will ask to confirm)

`mkdir dir1 dir2 ... dirN` create new directories with given path

`rmdir dir1 dir2 ... dirN` remove *empty* directories

`chmod +x file` tell the shell that `file` is an executable program

Interlude: Setup Activity

We will use these ideas and commands to set you up for the rest of the semester.

Windows Pre-Setup

First, Windows users running git-bash need to run a script to set up their environment.

To get the url, you can go to `https://github.com/36-750` Navigate to documents > ClassFiles > week1 > setup-profile.py And then hit the "Raw" button. Grab that URL and insert here.

```
cd
curl https://raw.githubusercontent.com/36-750/documents/master/ClassFiles/week1/setup-p
```

Then exit and restart git-bash.

Go Home

Move to your home directory. (How?)

```
cd
```

Create a Class Directory and Move There

Create directories `s750` and `bin` and switch to `s750`.

```
mkdir s750 bin
cd s750
```

The `bin` directory is where you will keep any utility scripts/programs that you want to use regularly.

The `s750` directory is where you will keep all your work for this course throughout the semester.

Setting Up Your Repositories

When you have git installed and have a github account, do the following. Otherwise, you should do both steps before the next class, as described in an email from Alex.

If you try to do this later, *remember to do it only after doing `cd ~/s750` first.

1. Set up your configuration. Git records your name and email with each commit:

   ```
   git config --global user.name   "Alex Reinhart"
   git config --global user.email  "areinhar@stat.cmu.edu"

   git config --list       # check the configs
   git config user.name    # ...or just one
   ```

   Use the email address you used with your GitHub account, so it will recognize you.

2. Clone the course repositories

   ```
   git clone https://github.com/36-750/documents.git
   git clone https://github.com/36-750/problem-bank.git
   ```

3. Clone your assignment repository. If your github account name is `NNNN`, do

```
git clone https://github.com/36-750/assignments-NNNN.git
```

replacing the NNNN with your account name in the command.

Next Time

When you want to work on your materials, start the shell, and return to your `s750` directory by typing

```
cd ~/s750
```

Text Processing (sort, uniq, wc, head, tail, diff, grep)

There are a variety of commands for processing text files. They can be combined to produce sophisticated operations.

Command	What it does	Useful options
sort	sort lines of input	-n, -r, -k
uniq	remove or count duplicates	-c, -d
head	output first N lines of input	-N
tail	output last N lines of input	-N
diff	output differences between two files	
grep	output lines of input that match pattern	-v, -e

Working with Commands (which, type, alias, history, man, help)

Many commands help us use other commands

Command	What it does	Basic form or example
which	find pathname of command	which command
type	indicate kind of command	type command, `type diff`
alias	set alias for command string	alias ls='ls -F'
history	list previous commands in session	history
man	complete command documentation	man command
help	documentation on builtin commands	help builtin

Try:

- `which ls`
- `type diff`
- `alias ls='ls -F'`
- `man grep`
- `help type`

Patterns, Quotes, and Substitutions

The shell has a rich and powerful pattern language for matching filenames. It is extremely useful but takes some time to learn. The best way is to learn by doing. We have time only for a few highlights, but check out the documentation on bash (or whatever shell you use) for more details.

Illustrative examples:

- `ls foo*.pdf`
- `ls foo?.txt`
- `ls ./*/tests*/foo[2-3A-E]*.txt`
- `ls dir ./*/A*[0-9][0-9].*`
- `ls -d ./**/*[0-9]*`
- `ls ../foo/bar/abc{def,ghi,jkl}qrs.txt`

We can also protect our arguments from this expansion in various ways:

- Single quotes '' prevent expansion within them
- Double quotes `""` allow variable expansion and special characters
- Command substitution produces the output of a command as a string, e.g., `cat $(echo entrypoints)`.

Environments, Variables, Configuration (printenv, set, export)

The shell also maintains some state in the form of variables that you can get and set. Most important are variables that can be accessed by programs run in the shell; these are called **environment variables**.

Type `printenv` to list the names and values of the current environment variables.

The three most important for common use are the `PATH`, which determines where the shell looks for commands, `HOME` which records the pathname of your home directory, and the prompt (`PS1` in bash).

We set `PATH` and `HOME` above. Try one of these to set the prompt and see how it changes:

```
export PS1=":\W: \!$ "
export PS1=":\[\033[0;34m\]\W\[\033[0m\]: \[\033[37m\]\!\[\033[0m\]$ "
export PS1=":\[\033[0;34m\]\w\[\033[0m\]: \[\033[37m\]\!\[\033[0m\]$ "
```

The `export` command tells the shell to make the variable part of the environment. You typically use it in your .profile when setting variables such as the `PATH` and `PS1`.

```
export PS1=":\W: \!$ "
```