The Shell and Other Tools
Statistics 650/750
Week 2 Tuesday

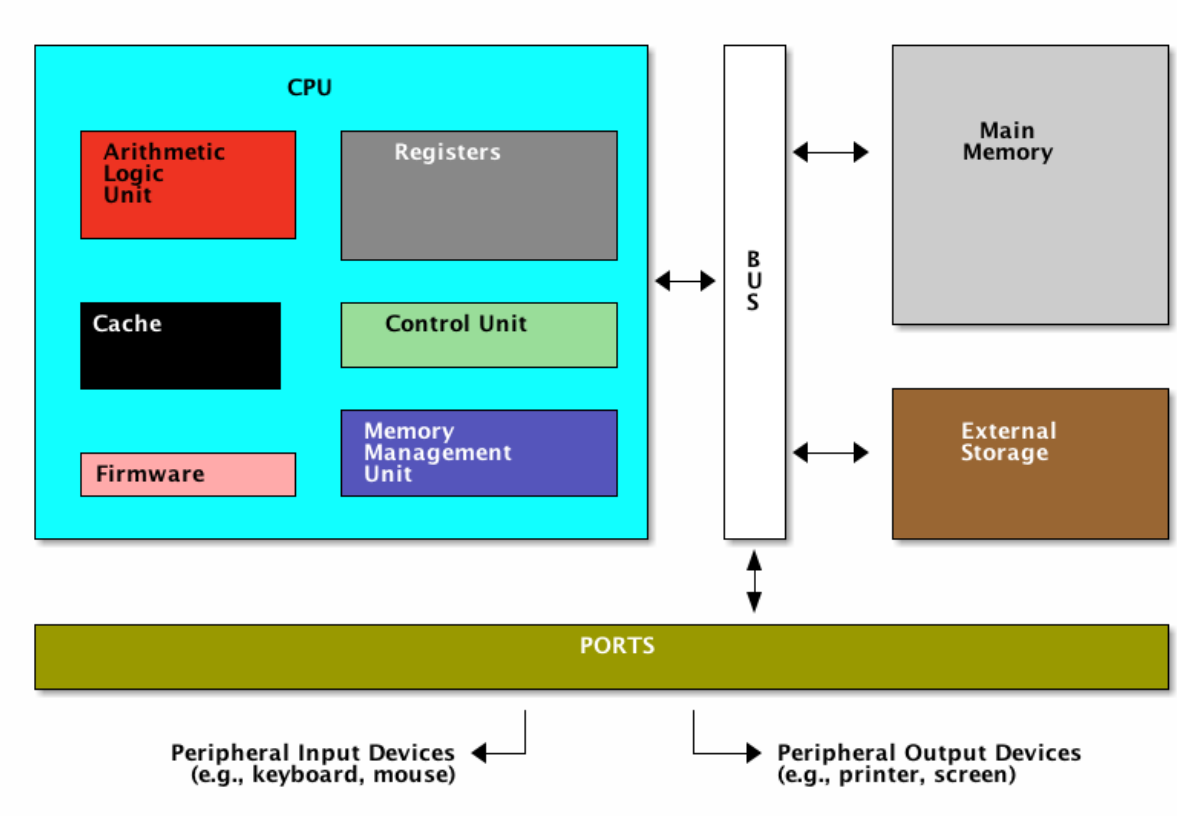
Christopher Genovese and Alex Reinhart

05 Sep 2017

A Brief Primer on Computer Architecture

Today, we will be covering how to use the command line to control your computer. For this – and more generally for thinking about programming – it is helpful to step back and consider as context how a computer works.

Here's a (somewhat simplified) picture of computer architecture:



The Components of Computer Architecture

- The Central Processing Unit (CPU)

  The CPU is the control center of the computer. It carries out the instructions given by computer programs for arithmetic, logic, system control, and input/output.

There are many CPU designs with different performance trade-offs. The main components of the CPU are:

  - Control Unit – manages the execution of instructions
  - Arithmetic Logic Unit – carries out arithmetic, logic, and bitwise operations
  - Registers – specialized, very fast storage used in executing instructions Some registers have fixed uses and some are general purpose. There are only a few of these available.
  - Cache – a fast but limited memory space to speed computations
  - Memory Management Unit – helps the operating system manage memory resources (optional)
  - Firmware – fixed program for initialization and startup

  In modern CPUs, some or all of these components are on a single integrated circuit chip; at the very least, they are colocate on a common circuit board (the motherboard)

- The Bus

  Specialized communication circuitry that allows transfer of data among the various parts of the computer. A fast bus is as valuable as a fast CPU to get high performance.

- Main Memory (RAM)

  A large, fast, random access storage area. Each slot in memory has a unique **address**, by which it can be read or written. There are different types of Random Access Memory, with different cost and performance characteristics.

- External Storage (e.g., Hard Disks, USB Drives, . . .)

  A very large storage area that is relatively very *slow* compared to memory. Rotating hard disks (and even solid-state drives) impose physical constaints in the order in which information can be accessed.

- Ports

  Addressable access points to other peripheral devices, allowing expansion of the system in a general way. Special programs called *device drivers* manage the details of how these devices are controlled.

How Programs (aka Apps) Run

At the lowest level, a program is a set of instructions in **machine code**, each instruction encoded in a fixed number of bits in the following form

  opcode data-to-operate-on-if-any

The opcode is a number representing a specific operation that is hardwired into the CPU circuitry. The data operated on depends on the instruction and consists of register "names", memory addresses, or numeric constants.

For example, here is a program for the Intel x86 processor to compute the GCD of two integers via Euclid's algorithm:

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 89 b6  00 00 00 00  39 d3 7e 13  29 c3 39 c3
75 86 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

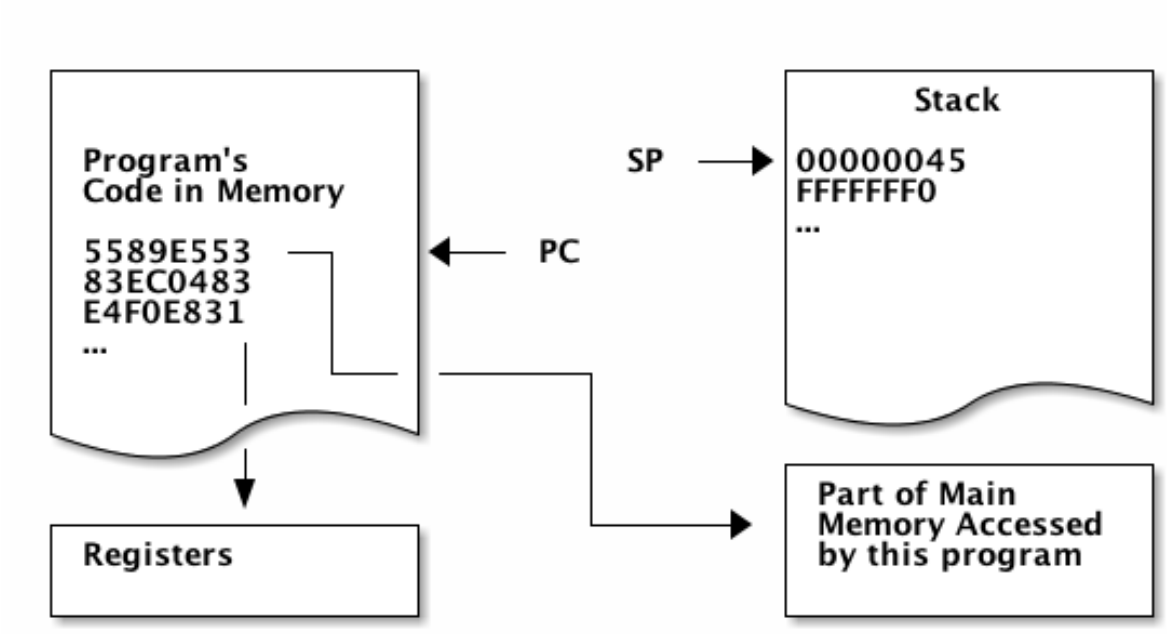
This is not easy to read or reason about, which is why we have programming languages! (To get a taste of programming "closer to the metal" I recommend you try a simple program in *assembly language*, which adds only the barest syntax to this machine code. Here's the above program in assembly language, for reference:

```
   pushl %ebp                      jle  D
   movl  %esp, %ebp                subl %eax, %ebx
   pushl %ebx                   B: cmpl %eax, %ebx
   subl  $4, %esp                  jne  A
   andl  $-16, %esp             C: movl %ebx, (%esp)
   call  getint                    call putint
   movl  %eax, %ebx                movl -4(%ebp), %ebx
   call  getint                    leave
   cmpl  %eax, %ebx                ret
   je    C                      D: subl %ebx, %eax
A: cmpl  %eax, %ebx                jmp  B
```

When the CPU runs a program, it is given a starting address which points to the machine code of the program and it proceeds through the instructions, modifying registers and memory, interacting with external storage and with peripheral devices, and keeping track of where in the code it is.

The CPU uses special registers to manage the control of the program's flow. For example, the **program counter (PC)** register (sometimes called instruction pointer, IP, or instruction address register, IAR) holds the next instruction to be executed. It is normally incremented with each instruction, but instructions like brances (conditionals), jumps, function calls, and function returns can change the PC register directly.

Similarly, the **stack pointer (SP)** points to an area in main memory that is used as a stack for temporary computation and storage, such as holding the arguments to a function before calling it. This is updated automatically for certain instructions.



Programs that run other programs

The process of turning a program into machine code is (loosely) called **compiling**. A C++ compiler, for instance, parses and processes the potentially complex C++ code and produces an **executable file** containing the program in machine code.

But from a programmer's perspective, we can write programs that parse and execute symbolic commands. These symbolic commands look just like programs, though written in some higher-level language.

Thus, not every program we run in practice gets converted to machine code to run, though there is always *a* program running machine code underneath.

For instance, by specification, Java programs are converted into the "machine code" of an idealized, non-physical machine called the *Java Virtual Machine (JVM)*, which is run by yet another program. A number of other languages produce JVM code as well and can be "run" by the same underlying program.

Similarly, **scripting languages** (python, R, ruby) have associated programs (python, Rscript, ruby) that run natively on the machine to process code in that particular language.

Files and Directories (aka Folders)

A **File** is persistent collection of data that is treated as a distinct and identifiable object by the operating system. Files can be accessed by the system and by users (with the appropriate permissions).

A **directory** (aka **folder**) is a special file that holds a list of other files (including directories) that are "*contained*" in that directory.

Files and directories are arranged in a *hierarchical* **file system** through which each file has a unique "address" called its **path**. This file system is essentially a *tree,* and the path tells how to access the file from the *root* of the tree.

Thus, when you see a name (more specifically a *path*) like `/Users/spock/logic-puzzles/Kolinahr`, we are referring to the file `Kolinahr` contained within the directory `logic-puzzles`, which is contained in the directory `spock`, which is contained within the directory `Users`, which finally is contained in the `root directory` of the file system.

Graphical user interfaces for file systems (e.g., as shown under OS X or Windows) provide a physical metaphor. Files and directories are represented by icons (documents or folders). You navigate the file system by clicking on directories (folders) to see the files contained within. You "open" a file by clicking on the file. When we work with files on the command line or within a program, we will typically eschew this metaphor and refer to files by a more direct means.

The Role of the Operating System

A user's primary interaction with a computer comes from running programs that operate on memory, files, and peripheral devices (e.g., printers, screens, robots).

Representing/working with programs and files at the level of the *system* is incredibly complex.

For example, while your Word file *looks like* a single stream of characters and encodings, it may be represented on disk as a collection of widely separated chunks of data that must be linked. together to read the file.

Similarly, each running process on the system gets an allocation of part of main memory. But not all of the process's storage might be kept in memory at a given time (as it can be huge). Parts are swapped in and out from disk as needed. The memory the program sees is actually *virtual memory* that is managed by the system.

The **operating system** is a special program running on the computer that acts as an interface between the system architecture that hides the system's complexity from the programs and the user.

The operating system has access to the computer's resources and manages them so that each program (and its programmer) sees a simple view of the available resources. Memory looks like a long, linear stretch of storage, files look contiguous, and the program sees itself as always running.

The operating system also makes available special operations (called *system calls*) by which a program can request and modify its resources or interact with peripheral devices.

The Command-Line Shell

Why use the command line?

For most users of modern computers, the computer driven by touch (including mouse): we click on programs to start them, we click on or drag files to open or process them, we click on menu items to select operations to perform and the data on which to perform them, we click on buttons to answer questions, and so forth.

This has the advantage that it is easy to learn, easy to operate, and easy to visualize (e.g., through physical metaphors like documents and folders and windows and menus).

But this graphical user interface (GUI) approach to interacting with computers has several disadvantages:

- The available operations are limited to those specifically chosen by the system' or application's programmers.
- The operations cannot be combined into new operations.
- The operations cannot be parameterized for generalization.
- The operations cannot easily be recorded for later recollection, reuse, modification, or automation.
- All that clicking and dragging is **slow**!

The **command line** offers an alternative way to interact with the system. We enter textual commands run programs with a chosen configuration on specified data.

This has some disadvantages:

- It has a nontrivial learning curve.
- There are many commands and configuration options, many not particularly mnemonic.
- Many commands focus on processing text files.

But this approach has many compensating advantages:

- Operations may be easily combined to form new and useful operations.
- Very complex commands can be built from several simple pieces.
- Commands can be recorded for later recollection, reuse, modification, or automation.
- Commands can be parameterized for more general uses.
- Once mastered, working on the command line is **fast**.

Example Use Case

Suppose you have this problem: **Read a file of text, determine the `n` most frequently used words, and print out a sorted list of those words along with their frequencies.** What can you do?

1. Find an existing program to solve this problem

   Maybe it exists, but for many specialized tasks you are out of luck. You might end up moving to an environment like excel and hacking it together.

2. Write a specialized program

   Donald Knuth wrote a 10 page Pascal program for this task. It was intended as an example of *literate* programming, simultaneously easy for humans and computers to read and reason about.

3. Use the command line directly.

   Doug McIlroy wrote this simple chain of commands that does the task in a few moments:

   ```
   tr -cs A-Za-z '\n' |
   tr A-Z a-z |
   sort |
   uniq -c |
   sort -rn |
   sed ${1}q
   ```

   Each of the steps in this chain is a program that does one basic operation on its input. McIllroy's "script" configures these programs with **command-line options** and creates a **pipeline** to chain them together into a more complex command.

All three approaches are viable. But we want to convince you of the power of #3 for many needs. By combining simple components that do *one thing well,* we get a vast assortment of useful tools.

The Shell

A **shell** is a *program* that provides a command-line interface for running other programs, configuring them with options, and doing useful things with their inputs and outputs.

Shells provide poweful built-in tools for routing and transforming data and for controlling other programs. They are accompanied by a full suite of other programs for manipulating text. Shell commands can be executed interactively or stored in files (called *shell scripts*) that can be executed as programs themselves.

Mastering a shell will make you a more powerful and more efficient computer user.

Choosing a shell

There are many different shells available, differing slightly in their syntax and in the features they provide: bash, sh, ksh, zsh, fish, tcsh, ash, dash, pdksh, ...

The default shell on Linux and Macs (and with git or cygwin on Windows) is **bash**, and we recommend that you stick with that initially. (As you gain experience, you can – but need not – look at some of the other possibilities. I use **zsh**, for instance.)

Your chosen shell (which I'll assume for now is bash) starts for you whenever you open a terminal. You can also start a *new* shell explicitly by typing the corresponding command on the command line, e.g.,

```
bash
```

Automation is Your Friend

> If you do it once, smile.
>
> If you do it twice, wince.
>
> If you do it thrice, automate it.

Shell scripts – files containing a sequence of shell commands that can themselves take arguments – provide a useful tool for recording and automating the steps you take in a data analysis.

For example, in this course, we will often automate:

- Tests. Most test frameworks provide tools to run all tests in a directory or in certain files
- Data cleaning tools which need to be used on new data files

```
Rscript clean.R < big_data_dump.csv > clean_data.csv
```

- Solvers and analyses that have to run on various input files

```
python ingest_crimes.py -s 2501 data/homicides-2014.txt
```

- Plot and result generation for reproducible research

```
Rscript make_paper_plots.R
```

The structure of shell commands

The Shell Prompt

When the shell is ready to accept input from the user it displays a **prompt**. The prompt is a string, such as '% ', with a recognizable pattern.

Here's my prompt as I write this:

```
:documents/Lectures: 6552\%
\^{}                          \^{}\^{}
\begin{center}
\begin{tabular}{ll}
 & \\
\end{tabular}
\end{center}
+-----  Prompt  ----------+Command starts here
```

This prompt string contains special "control characters" that cause the shell to display information about its current state: in this case, a width-limited display of the current directory and the command number.

In `bash`, the primary prompt is stored in a variable called `PS1`. At the shell prompt, type the command

```
echo $PS1
```

(Here, `echo` is a command that writes its input to the terminal. The `$name` or `${name}` construct is replaced with the value of a *variable* kept by the shell.)

By default, you will see '`\s-\v\$` '. The `\s` and `\v` are control characters that give the name and version of the current shell, respectively.

There are also variables `PS2`, `PS3`, and `PS4` that represent prompts used for special purposes.

To change the prompt, set the variable `PS1` at the prompt, like so:

```
PS1='make my day: '
```

To make this permanent, put this line in the file `.bashrc` in your home directory.

Shell Commands and Arguments

Shell commands are the names of *executable files* – aka programs or apps – on the file system. Shell commands start with the **command name** followed by one or more **arguments**. The arguments are separated by **spaces**. (Some commands, like git, include a *subcommand* name after the command name, such as `git branch` or `git commit`, but otherwise look similar.)

We saw the `echo` command used above. Here are a few more examples. Try these at your shell:

- `echo Hello, World`
- `which git`
- `ls`
- `ls -A`
- `ls -F`
- `ls -lrt ..`
- `ls -1F /bin/ls /bin/bash /usr/local/bin/git`

In all of these, the first word names the command, and what follows is a mixture of options and other arguments. *Some commands require arguments, some work with none.*

By convention, arguments that begin with - set configuration options. Compare the results of `ls`, `ls -l` (minus ell), `ls -lr` (minus ell, r), and `ls -lrt` (minus ell, r, t). What do the included ell, r, and t appear to do?

Because of historical developments, there are a variety of ways that programs represent these options. The most common current conventions (POSIX) is:

- A single - starts a **short option** or **flag,** which is named by a single letter. A *short option* may itself require arguments, which must immediately follow the option (either within the same argument or as the next argument). For instance

  ```
  dvips -o foo.ps foo.dvi
  sort -k6,7 file
  sort -k 1,3 file
  perl -iorig program.pl
  ```

  A *flag*, on the other hand, represents an on or off switch. It does not take arguments, so multiple flags can be combined in one string, as in `-lrt` above.

- A double `--` starts a **long option**, which is named by a word or hyphen-separated phrase. Any arguments required by the long option are specified after an '=' in the option string or in the next argument. (Different commands and options may differ in whether they support '=' or next argument or both.)

  ```
  sort --key=6,7 --output=sorted.out file
  git branch --contains 0fcea97
  grep --ignore-case --max-depth 3 pattern file
  ```

- Options may be given in any order and may appear multiple times.
- All options should appear before any required arguments to the command. (This is sometimes violated, e.g., git.)
- Some specific options like `--help` and `--version` are always supplied to help the user understand how to use the command.

  Try `git --help` or `git --version` or `grep --help` for instance.

  Sadly, this is not uniformly supported. Try `-h` or `-help` as alternatives.

- A double `--` by itself represents the *end* of optional arguments, everything that follows is a required argument.
- A single `-` by itself usually denotes a special input/output channel, standard input or standard output, which we'll cover below.

Finding Commands

If I want to add commands to the system, how does the shell find them?

The key point is that commands are represented by *files*, so the shell just needs to look for commands in special places in the file system.

Fortunately, you can tell it where to look and how to prioritize the different locations using the shell variable `PATH`.

`PATH` is a string that contains a list of directories separated by `:`'s. (Why colons? Don't ask.)

When the shell sees a name `N` in the command position of a line, the shell looks for an executable file named `N` in the first directory on the path. If it finds it, it executes it. Otherwise, it moves on to the second directory of the path. And so on, along the entire list.

Here's my default PATH, for instance:

```
.:./bin:../bin:/Users/genovese/bin:/usr/local/priority/bin:/usr/local/bin:/Library/\TeX{}/texbin:/u
```

It's worth mentioning at this point that `.` and `..` are special directory names that represent, respectively, the **current directory** and the **parent directory** of the current directory. The special name `~` represents the users **home directory**. So `/Users/genovese/bin/foo` can be written as `~/bin/foo`.

In `bash`, you can set your path by

```
export PATH=/foo/bar:/blah/zap/bin:/usr/local/bin
```

and you can prepend or append to the path by

```
export PATH=/high/priority:$PATH
export PATH=$PATH:/fallback/place
```

To make this permanent, put this line in the file `.profile` in your home directory.

A Few Comments on Names

- Try to avoid spaces in file and directory names. Hyphens are a good alternative.
- Avoid using special characters such as `*?&:$!#()[]<>{}` in your file and directory name because these characters have other uses on the command line.
- On non-Windows systems, file names can be as long as you like.
- Macs do not distinguish case in names, though it records the case as you name it.

A Useful Tool

The site `https://explainshell.com/` provides a useful tool for understanding the structure of shell commands.

Enter a bash command into the given text field, and it will identify each of the components of the command with useful explanation and help for what the command, options, and arguments mean and expect.

Keep that handy as you learn the shell.

Let's try a couple examples from what we've seen so far...

Creating commands

Because of how commands are defined and found, we can create our own commands.

Here's a goofy example. Create a file `blah` in your current directory containing the following four lines:

```
#!/bin/bash
echo The first argument is $1.
echo The files in the directory above the current one are that start with g are:
ls -1 .. | grep --ignore-case '^g'
```

The first line is a special comment (delimited by `#`) that I'll explain in a moment. What do you make of the rest?

At the shell prompt, type

```
chmod +x blah
```

(This might not work for Windows users, but don't worry.)

Now try two commands (for Windows users only the second might work):

```
./blah first second third fourth
```

and

```
bash blah first second third fourth
```

Why did I include the `./` in the first command? What does all this mean?

The file `blah` is an example of a **shell script**, a list of shell commands that can itself be run as a command.

The first line `#!/bin/bash` is called a **shebang**. It is a special comment that tells the shell how to process the file as a command – what program will interpret these commands. In this case, it is the shell itself. And the shell essentially does a version of what we do in the second command.

The command `chmod +x blah` makes the file `blah` an *executable*, telling the shell that it is OK to use that file as a command. The second version of the command does not require that step.

We can make **scripts** using other languages too. For instance, most interactive programming languages provide a command-line tool that will run a program. For example:

```
python batch.py
Rscript analyze.R
ruby frobnicate.rb
racket anagrams.rkt
julia maxSubSum.jl
```

If you have a program in such a language, you can turn it into its own command with the same two steps as above:

1. Add a *shebang*, e.g., `#!/usr/bin/python`, to the first line of the file
2. Change the file to executable, e.g., `chmod +x foo`.

Here's a file `foo` after these steps

```
#!/usr/bin/python

def frobnicate():
    pass

frobnicate()
```

Now, you can run `./foo` as the command when in the same directory, or more conveniently, put `foo` in your path and just type `foo`, or you can type the full path `/Users/genovese/frobs/foo`.

(Again, Windows is a bit stodgier here.)

Commonly Used Shell Commands

Now that we have an idea of how the shell and shell commands work, let's look at some of the most commonly used commands.

(Note: These commonly used commands tend to have rather terse names. There is some mnemonic value here, but it takes some getting used to.)

Moving Around the Filesystem

At any point, the shell maintains a record of which directory (aka folder) you are currently working in. This is called the `working directory`.

The command `pwd` ("print working directory") tells you what that directory is. Try it! (Remember what this tells you for later.)

The command `cd` ("change directory") changes the working directory to any specified directory. If you do not supply a directory on the command line, `cd` moves to your home directory by default.

Remember that a directory is just a file on the system, so we can specify a directory by giving its **path**.

To go to my home directory, I could type `cd /Users/genovese`.

This path `/Users/genovese` is an **absolute path**. It tells how to find the file (in this case a directory) starting from the **root** of the file system.

We can also specify a **relative path**, which is taken relative to the current working directory.

Try this:

1. Move to your home directory by typing `cd` at the prompt.
2. List the files in that directory by typing `ls -F` at the prompt.
3. Pick a directory, one of the files listed that ends with `/`.
4. Type `cd DIRECTORY`, replacing DIRECTORY with the name you picked in step 3.
5. Type `pwd` and then `ls` to look at the contents of that directory.
6. Move to the parent directory with `cd ..`
7. Type `pwd` again to confirm that you are back.

The directory name and `..` are *relative paths*. Similarly, when you type `./foo/bar`, you are specifying a file `bar` within a directory `foo` within the current working directory.

Use `cd` to move back to the directory you were in when we started.

You can create and remove directories as well.

- `mkdir` ("make directory") creates a new empty directory
- `rmdir` ("remove directory") deletes an *empty* directory

Try `mkdir foobarzap`. Type `ls -F` to see that it has been created. Then type `rmdir foobarzap` to eliminate it.

If your directory were not empty, you would need to remove its contents before doing `rmdir`.

Looking at and Editing Files and Directories

We have seen the `ls` command to list the contents of a directory. Here are a few other useful tools.

- `ls` ("list") lists files matching specified requirements
- `cat` ("concatenate") echoes the contents of files to its output
- `more` (or even better `less`) scrolls interactively through a file at the terminal
- `head` and `tail` show the beginning or ending lines of a file
- `uniq` ("unique") removes and or counts duplicate lines
- `sort` sorts its input using specified criteria
- `cut` cuts out parts of lines in a file
- `wc` ("word count") count characters, words, lines in input
- `cp` ("copy") copies files to new locations
- `mv` ("move") renames files
- `rm` ("remove") deletes files

These simple commands are more powerful than they seem.
For example, if I have a file at path `A` and want to copy it to path `B`, I can type

```
cp A B
```

to do so. But if I have a collection of files `A1` through `A5` and a directory `D`, I can copy those files into the directory by typing

```
cp A1 A2 A3 A4 A5 D
```

Even more conveniently, the shell has special characters, called **wildcards**, that can be used to specify patterns.

If python file names, for instance, end in '.py', what do you guess following does:

```
cp *.py ../backups
```

The most often used wildcards are

- `*` – matches any string of zero or more characters
- `?` – matches any single character
- `[]` – matches any single character that is included in the brackets, which can be a range. Examples: `[xyz]`, `[a-z]`, `[-0-9_A-Z]`.

Q: What is the pattern for file names ending in `.r` or `.R` that contains a version number consisting of three single digits separated by dots just before the suffix, e.g., `foo.0.9.1.r`?

Be careful with the `rm` command, especially with wildcards. You cannot undo a delete. For this reason, I always use `rm -i`, which asks me before deleting anything.

Note: There are many different kinds of generalized patterns used, especially regular expreessions, which we will see. Many use similar special characters. Shell wildcards are a simple pattern language but distinct from these other.

Finding Things

- `grep` ("globally search a regular exprssion and print") finds lines in its input whose contents matches a special pattern
- `find` finds files matching specifed criteria

There's a lot to both of these commands, but even at their simplest, they are useful.
Let's look at `grep shell` and `grep -i shell` applied to this file.
Here's how I would find "org" files in my lectures directory containing the word shell:

```
find .. -type f -name '*.org' -exec grep -q shell '{}' \; -exec ls -ld '{}' \;
```

Input and Output

Standard Input, Standard Output, Standard Error

Command-line programs have three special input and output streams associated with them:

- **standard input** (stdin, STDIN) – by default the input the user types
- **standard output** (stdout, STDOUT) – by default echoes to the terminal
- **standard error** (stderr, STDERR) – by default echoes to the terminal

For example, when I type, `echo hello world`, the string "hello world" (with a newline) is written to the standard output (the terminal).

When I type `cat` (with no arguments), it waits for the input I type and echoes that to the terminal, passing its standard input to its standard output. Try it.

Reading from **standard input** in a program is easy, though the details vary from language to language
In python:

```
import fileinput

# a bunch of code goes here

for line in fileinput.input():
    do_stuff(line)
```

Or in R:

```
#!/usr/bin/Rscript

## do stuff here

f <- file("stdin")
open(f)

lines <- readLines(f)
do_stuff_with(lines)
```

Writing to **standard output** is easier: just write output like you normally do, with `cat` or `print` or `println` et cetera.

The third stream, **standard error** is used to separate warnings and other special messages from a programs output. It is by default directed to the terminal, but by changing where that points, we can keep our results and our diagnostics from interferring with each other.

In R, the `message` and `warning` functions print to standard error by default. For example,

```
print(a_bunch_of_data)

message("Processed ", nrows(data), " data entries, ", num_errors, " failed")
```

The result of `print` will go to whereever we direct our output; the result of message to where we direct our errors. This is very useful.

In Python 3, we can similarly write

```
import sys

print("A big important warning message", file=sys.stderr)
```

What makes these streams so useful in the shell is that we can **redirect** each of them to other sources.

Redirection and Pipelines

You can arbitrarily redirect standard input, output, and error to other files or devices.

For example,

```
./maze-solver < maze.txt > solution.txt
```

runs the program `maze-solver`, reading input from the file `maze.txt` *as though it were typed at the keyboard* and directing its output to the file `solution.txt` instead of the terminal (overwriting that file in the process.

The most common redirection methods are:

- `> file` – direct standard output to file `file` (overwrites file)
- `< file` – read from `file` as standard input
- `2> file` – direct standard error to `file`
- `» file` – direct standard output to append to `file` (not overwriting)
- `« STRING` – read input from the following lines until one beginning with `STRING`

There are other combinations as well, like `2>&1` (stderr to stdout),

In addition, command-line programs are *composable* through **pipelines**. You can plumb them together in a chain, like functions in your code, to achieve complicated results *without changing your code*. The character `|` denotes a **pipe**, which maps the standard output of a program to the standard input of the next.

```
./analyze.R data.txt | grep "statistically significant" | less
```

These processes run in parallel, one passing output to the next.

We will talk about writing small, composable functions that can be fit together and reused in different ways. A corollary is to write small, composable *programs* which can be combined in different ways by using the shell.

Executing in the Background

If you end a command with `&`, the command will execute quietly in the background and you can keep working. Such commands should not read or write from the terminal.

Iteration, Conditionals, and Other Programming Constructs

Besides commands, the shell also provides a variety of programming constructs including iteration (`for`), conditionals (`if`), functions, nested commands, and a variety of simple data structures.

This makes it possible to create quite elaborate shell scripts, which can be useful for automation and other tasks.

In general, above a certain level of complexity, I prefer to write programs in a scripting language, but your mileage may vary.

Configuration

We have seen `PS1` and `PATH` as ways to customize the shell's behavior. But much more is possible.

Each shell maintains a collection of variables that describe its environment. You can customize these variables, define new commands, define aliases to making typing more convenient, and do much more so that every time you start the shell you have it configured as you prefer.

For bash, the `.profile` and `.bashrc` files in your home directory are automatically read for such customizations and configurations. (The latter is specifically for interactive shells; the former is ready whenever the shell starts.)

Remote Shells

The Secure Shell (SSH) protocol lets you log in to a remote computer and use its shell. The Statistics Department maintains a set of servers for use by students and faculty to do their work on – if you need 32GB of memory and eight CPUs for your analysis, a department server may be right for you.

We'll be using the department servers to learn about databases, starting this Thursday.

For Mac and Linux users, an SSH client is already provided. Just type

```
ssh yourusername@hydra1.stat.cmu.edu
```

and you'll be connected to the shell on hydra1.

Windows users will need PuTTY, a free client for Windows. If you're on a Windows computer, you should download `putty.exe` before next class, and get it set up. Make sure you can connect to the department servers.

Resources

If you are new to the command line, you should try one of these:

- Software Carpentry has a Unix Shell Tutorial
- LearnShell.org is an interactive tutorial you can run in your browser
- ExplainShell is a useful tool for understanding bash command lines

There is also a `cli-murders` homework exercise to grow your understanding of the command line and the tools it provides.