A General Systems Handbook

Michael Marsh

A General Systems Handbook

Michael Marsh

Preface

This book is designed to be a handy reference for students taking systems courses in Computer Science. It covers a range of topics from some basic architectural details, to numeric representations in memory, to a number of useful tools. It is not intended to be exhaustive, but should be a good starting point, so that you are better-equipped to find and understand more detailed documentation.

Contents

Chapter 1

Some Architecture Basics

Let's take a brief look at how things work in a typical computer. We will focus on a single *process*, which is a program instance being run by the operating system's *kernel*.

1.1 Process Memory Layout

A typical process layout contains (from lowest memory addresses to highest):

- **Text** – this is where the actual binary instructions are stored
- **Data** – global and static data that's initialized when the program starts
- **BSS** – global and static data that is uninitialized when the program starts
- **Heap** – dymanmically allocated memory
- unallocated space
- **Stack** – function-local data

The heap and stack grow and shrink as the program runs, the heap from the bottom up through previously unallocated space, and the stack from the top down. It's important to not that **while the stack grows downwards, data strucutres in the stack still behave normally, address-wise.** That is, if we have an array in the stack, the address of the first element of the array is the *lowest* memory address of the array. Somewhat confusingly, we refer to the lowest stack memory address as the *top* of the stack.

1.2 Types of Memory

When we think of memory, we're usually thinking of Random-Access Memory (RAM). There are other types of memory, however, and typically the faster memory accesses are, the less of that type of memory we have, due to cost.

The absolute fastest memory is part of the procesor. This memory consists of a set of *registers*, which are used for computations that are currently in progress, or certain state that the processor needs to keep track of.

Between the registers and RAM there are usually between 1 and 3 layers of *cache*. These are fast RAM chips where data actively being accessed is stored. When looking up a memory address during execution, the cache is checked first. A *cache hit* means the values can be read or written very quickly. A *cache miss* means we have to go to the main memory, which is noticeably slower.

In order to support memory needs beyond what the computer physically has available, many operating systems support *virtual memory*, which includes *swap space*. This is a region of an attached disk (such as your hard drive) that the kernel can use to store data that doesn't fit in RAM. A *page fault* means that a memory address is located in a *page* of memory (a large block) that is in swap, rather than RAM. When more *working memory* (in RAM) is needed, pages are *swapped out* to disk. This is, by far, the slowest type of local memory access. If your system is making considerable use of swap, you will notice it running

substantially slower.

1.3 Segmented Virtual Memory

Virtual memory has another use beyond allowing for extra space. The common memory model is *segmented virtual memory*, which which each process is given a restricted view of the memory space. From the perspective of the process, it exists in a computer with addresses in the range (for a 32-bit architecutre) `0x0000001` through `0x7fffffff`, and those are the only addresses it can see. The processor translates memory access requests from the process from the memory segment's address to the actual address in main memory (or cache).

If a process attempts to access memory outside of this range, it generates a *segmentation violation* (SEGV) signal, which produces a *Segmentation Fault*. This prevents processes from reading or writing memory belonging to other processes (sometimes called "stomping").

1.4 Loading a Binary

When you compile (and assemble) a program, it produces a binary file. When we run the program, it loads this file into memory and sets up the data structures needed for execution. Let's consider this in a series of steps.

First, we have a C file (`foo.c`):

```c
#include <stdio.h>
int main(int argc, char** argv) {
   for ( int i = 0; i < 10; ++i ) {
      printf("i = %d\n",i);
   }
   return 0;
}
```

Now, let's compile this with **`gcc -S foo.c`**. This produces an assembly file **`foo.s`**, with the following for **`main`**:

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq        %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $32, %rsp
        movl        %edi, -20(%rbp)
        movq        %rsi, -32(%rbp)
        movl        $0, -4(%rbp)
        jmp        .L2
.L3:
        movl        -4(%rbp), %eax
        movl        %eax, %esi
        leaq        .LC0(%rip), %rdi
        movl        $0, %eax
        call        printf@PLT
        addl        $1, -4(%rbp)
.L2:
        cmpl        $9, -4(%rbp)
        jle        .L3
        movl        $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

If we then assemble this (producing **`foo`**), we can view the assembly and corresponding bytecode in **`gdb`**:

```
(gdb) disassemble/r main
Dump of assembler code for function main:
   0x0000000000001149 <+0>:   f3 0f 1e fa           endbr64
   0x000000000000114d <+4>:   55                    push   %rbp
   0x000000000000114e <+5>:   48 89 e5              mov    %rsp,%rbp
   0x0000000000001151 <+8>:   48 83 ec 20           sub    $0x20,%rsp
   0x0000000000001155 <+12>:  89 7d ec              mov    %edi,-0x14(%rbp)
   0x0000000000001158 <+15>:  48 89 75 e0           mov    %rsi,-0x20(%rbp)
   0x000000000000115c <+19>:  c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
```

```
   0x0000000000001163 <+26>:  eb 1a                    jmp    0x117f <main+54>
   0x0000000000001165 <+28>:  8b 45 fc                 mov    -0x4(%rbp),%eax
   0x0000000000001168 <+31>:  89 c6                    mov    %eax,%esi
   0x000000000000116a <+33>:  48 8d 3d 93 0e 00 00     lea    0xe93(%rip),%rdi
   0x0000000000001171 <+40>:  b8 00 00 00 00           mov    $0x0,%eax
   0x0000000000001176 <+45>:  e8 d5 fe ff ff           callq  0x1050 <printf@plt>
   0x000000000000117b <+50>:  83 45 fc 01              addl   $0x1,-0x4(%rbp)
   0x000000000000117f <+54>:  83 7d fc 09              cmpl   $0x9,-0x4(%rbp)
   0x0000000000001183 <+58>:  7e e0                    jle    0x1165 <main+28>
   0x0000000000001185 <+60>:  b8 00 00 00 00           mov    $0x0,%eax
   0x000000000000118a <+65>:  c9                       leaveq
   0x000000000000118b <+66>:  c3                       retq
End of assembler dump.
```

The middle section (between the **:** and assembly) is the bytecode corresponding to the assembly that follows it. Note that the assembly is in a different format than in **foo.s**, but it should be fairly straightforward to connect them.

Here's the important part: The bytecode for this function is (with some reformatting):

```
f30f 1efa 5548 89e5 4883 ec20 897d ec48
8975 e0c7 45fc 0000 0000 eb1a 8b45 fc89
c648 8d3d 930e 0000 b800 0000 00e8 d5fe
ffff 8345 fc01 837d fc09 7ee0 b800 0000
00c9 c3
```

If we view the program with **xxd foo**, we'll see the following:

```
00001140: f30f 1efa e977 ffff fff3 0f1e fa55 4889
00001150: e548 83ec 2089 7dec 4889 75e0 c745 fc00
00001160: 0000 00eb 1a8b 45fc 89c6 488d 3d93 0e00
00001170: 00b8 0000 0000 e8d5 feff ff83 45fc 0183
00001180: 7dfc 097e e0b8 0000 0000 c9c3 0f1f 4000
```

Notice that about halfway through the first line, we see the beginning of the bytecode as reported by **gdb**. In fact, if we look at the first line from **gdb**, it reports an address of **0x1149**, and if we count the bytes from the offset in the

file of **0x1140** (as shown above), we see that this corresponds to the location of the first byte in the file!

1.5 Processing Instructions

Because we hadn't run the program yet in **gdb**, we were only seeing addresses relative to the start of the Text section of the process, not the actual addresses. Once we start the program, we'll see the actual addresses. If we set a breakpoint in **main**, then **run**, we'll see something like the following when we disassemble:

```
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x0000560f61c4b149 <+0>:   endbr64
   0x0000560f61c4b14d <+4>:   push   %rbp
   0x0000560f61c4b14e <+5>:   mov    %rsp,%rbp
   0x0000560f61c4b151 <+8>:   sub    $0x20,%rsp
   0x0000560f61c4b155 <+12>:  mov    %edi,-0x14(%rbp)
   0x0000560f61c4b158 <+15>:  mov    %rsi,-0x20(%rbp)
   0x0000560f61c4b15c <+19>:  movl   $0x0,-0x4(%rbp)
   0x0000560f61c4b163 <+26>:  jmp    0x560f61c4b17f <main+54>
   0x0000560f61c4b165 <+28>:  mov    -0x4(%rbp),%eax
   0x0000560f61c4b168 <+31>:  mov    %eax,%esi
   0x0000560f61c4b16a <+33>:  lea    0xe93(%rip),%rdi        # 0x560f61c4c004
   0x0000560f61c4b171 <+40>:  mov    $0x0,%eax
   0x0000560f61c4b176 <+45>:  callq  0x560f61c4b050 <printf@plt>
   0x0000560f61c4b17b <+50>:  addl   $0x1,-0x4(%rbp)
   0x0000560f61c4b17f <+54>:  cmpl   $0x9,-0x4(%rbp)
   0x0000560f61c4b183 <+58>:  jle    0x560f61c4b165 <main+28>
   0x0000560f61c4b185 <+60>:  mov    $0x0,%eax
   0x0000560f61c4b18a <+65>:  leaveq
   0x0000560f61c4b18b <+66>:  retq
End of assembler dump.
```

Now we're seeing actual memory addresses, with an indicator that we're on the first instruction of the function. Running **info frame** then gives us:

```
(gdb) info frame
Stack level 0, frame at 0x7ffc71427c20:
 rip = 0x560f61c4b149 in main; saved rip = 0x7f20725020b3
 Arglist at 0x7ffc71427c10, args:
 Locals at 0x7ffc71427c10, Previous frame's sp is 0x7ffc71427c20
 Saved registers:
  rip at 0x7ffc71427c18
```

There's a lot here, but the important thing is:

```
 rip = 0x560f61c4b149 in main; saved rip = 0x7f20725020b3
```

Here, **`rip`** is the register holding the *instruction pointer*, and it's value is the address of the start of **`main`**. This register tells the CPU the address at which the *next* instruction to process resides in memory.

How does the CPU process this instruction? It begins by reading the byte at the instruction pointer. If that byte represents a complete instruction, it executes it and advances the instruction pointer to the next byte. If the first byte is only the start of an instruction, it continues reading bytes until it gets a complete instruction to execute, and then advances the instruction pointer by the number of bytes read.

Not all sequences of bytes represent valid instructions, however. Let's consider an analogy: We're reading a simple mathematics problem, "multiply 3 and 5". Taking this one character at a time, we can see the following sequence of events:

1. "m" – not a complete instruction
2. "mu" – not a complete instruction
3. "mul" – not a complete instruction
4. "mult" – not a complete instruction

5. “multi” – not a complete instruction
6. “multip” – not a complete instruction
7. “multipl” – not a complete instruction
8. “multiply” – not a complete instruction
9. “multiply “ – instruction type recognized, but more information needed

We would continue reading characters until we have a complete instruction, and then we would solve the math problem.

Now consider the case where our “instruction pointer” is off by one:

1. “u” – not a complete instruction
2. “ul” – not a complete instruction
3. “ult” – not a complete instruction
4. “ulti” – not a complete instruction
5. “ultip” – **no valid instruction begins with these letters!**

When this happens to the CPU, it signals that it received an *Illegal Instruction*. The same thing happens for an instruction like “multiply 3 and grapefruit”.

1.6 The Stack and Function Callings

We’re going to take a look at the structure of the *stack*. When we call a function, we add a new *frame* to the stack, containing state for that function call. This includes the arguments passed to it, local variables, some working space, and

information about *what to do when the function returns*. This information includes the next instruction pointer (as we saw previously), and where the *calling* function's stack frame is.

Because the stack grows into the unused memory between it and the heap, it can get a bit confusing. Adding to this confusion is the fact that the stack interacts with a number of CPU registers, and actually holds old values of some registers so that they can be restored.

The stack grows in two ways:

1. When a function calls another function, it adds a new stack frame *below* its own frame (in terms of addresses). This is lowest-addressed frame is referred to being at the *top* of the stack.
2. Within a function, we sometimes add new local variables or other values to the stack. These are added below the memory address of the "top" of the stack. This lowest address is stored in a register **`esp`** (on 32-bit systems) or **`rsp`** (on 64-bit systems), and is called the *stack pointer*.

We also have a *base pointer* (**`ebp`** or **`rbp`**). This provides a reference address for all of the local variables in the current (top) stack frame. While the stack pointer register just holds an address representing whatever is at the top of the stack (lowest address), the base pointer register holds the address of *another* pointer address.

So, what does the stack hold at **`rbp`**? It holds the base pointer of the *calling* function. The stack also holds another important address just above this, which is the *return address* for the function. Because our instructions are in the process's text section, the instruction pointer **`rip`** keeps track of the next instruction to execute. Function calls make this jump around, though, so when a function calls another function, it needs to store the address of the instruction to execute *after* that other function returns. That's the return address, or *saved instruction pointer*. To see all of this in action, let's consider the following code, which we've put in file **`bar.c`**:

```c
#include <stdio.h>

void g(int a) {
  int b = a*4;
  printf("%d x 4 = %d\n",a,b);
}

void f(int a) {
  int b = a*3;
  g(b);
}

int main(int argc, char** argv) {
  f(3);
  return 0;
}
```

This is a very simple sequence of function calls: **`main`** calls **`f`**, and **`f`** calls **`g`**. We'll examine the registers and stack with **`gdb`**. When we're in **`f`**, just before calling **`g`**, we see the following registers:

```
(gdb) info registers rsp rbp
rsp            0x7ffc59e797e0      0x7ffc59e797e0
rbp            0x7ffc59e79800      0x7ffc59e79800
```

This tells us that the stack pointer is **`0x7ffc59e797e0`**, and the base pointer is **`0x7ffc59e79800`**, so there are 32 bytes on the stack for local variables.

When we continue executing into **`g`**, we see:

```
(gdb) info registers rsp rbp
rsp            0x7ffc59e797b0      0x7ffc59e797b0
rbp            0x7ffc59e797d0      0x7ffc59e797d0
```

We've now added another 48 bytes (**`0x7ffc59e797e0-0x7ffc59e797b0`**) to the stack, and the current base pointer is **`0x7ffc59e797d0`**. We can examine the frame, as well:

```
(gdb) info frame
Stack level 0, frame at 0x7ffc59e797e0:
 rip = 0x555f3085f158 in g (bar.c:4); saved rip = 0x555f3085f1a2
 called by frame at 0x7ffc59e79810
 source language c.
 Arglist at 0x7ffc59e797a8, args: a=9
 Locals at 0x7ffc59e797a8, Previous frame's sp is 0x7ffc59e797e0
 Saved registers:
  rbp at 0x7ffc59e797d0, rip at 0x7ffc59e797d8
```

We know the current frame's structure, so we know that the *caller's* frame starts at address **`0x7ffc59e797e0`**, as we saw before. It's also telling us that it's saved the caller's base pointer at **`0x7ffc59e797d0`**, which we just saw is the *current* base pointer, and the instruction pointer is saved at **`0x7ffc59e797d8`**, which is 8 bytes (or 64 bits) above the base pointer.

We can verify that the current base pointer's address holds the previous base pointer by examining the memory directly:

```
(gdb) x/a 0x7ffc59e797d0
0x7ffc59e797d0: 0x7ffc59e79800
```

If we compare this with the base pointer in **`f`** before calling **`g`**, we see that they are identical.

Chapter 2

Numeric Representations

2.1 Integer Type Sizes

We are used to thinking of a byte as 8 bits (which isn't strictly true, but is *almost always* the case), but larger sizes become more ambiguous.

It used to be the case (when 32-bit processors were dominant) that an `int` in C would be 4 bytes (32 bits), a `short int` would be 2 bytes, and a `long int` would be 8 bytes. All of these are signed quantities. `unsigned int` is the corresponding non-negative 4-byte integer value.

With most processors now being 64-bit, these have shifted somewhat. Now an `int` might be 8 bytes, though `short` and `long` may or may not be twice as long. In many programs, we don't really care, but when we're encoding numbers, this becomes very important.

The header file `stdint.h` contains the following types, which you should use when you want to ensure the size of the value in bytes:

Type	Size (bytes)	Signed/Unsigned
`int8_t`	1	signed

`int16_t`	2	signed
`int32_t`	4	signed
`int64_t`	8	signed
`uint8_t`	1	unsigned
`uint16_t`	2	unsigned
`uint32_t`	4	unsigned
`uint64_t`	8	unsigned

2.2 Byte Encoding

Numbers have to be stored in memory on a host. They also have to be saved in files and sent over the network. This seems simple, but how a number is stored is more complicated than you might expect.

While a single-byte integer value is easy ("10" is "`0A`" in hex), once you have more than one byte, you have to consider the specific *architecture*. There are two main architectures commonly used: *big endian* (BE) and *little endian* (LE). In big endian encoding, the most significant byte of the number comes first in memory. In little endian encoding, the least significant byte come first.

Some examples might help:

Number	Size (bytes)	BE	LE
12	2	`00 0C`	`0C 00`
3072	2	`0C 00`	`00 0C`
4660	2	`12 34`	`34 12`
13330	2	`34 12`	`12 34`
12	4	`00 00 00 0C`	`0C 00 00 00`
201326592	4	`0C 00 00 00`	`00 00 00 0C`

2.3 Host and Network Byte Order

The host's architecture specifies the *host byte order*, but when exchanging values over the network, we can't have architecture-dependent ambiguity. Consequently, the networking community decided on big endian as the standard *network byte order*.

Because of this, if we receive a 4-byte integer value `0000000C`, we can safely assume these bytes represent the number 12, not 201326592, regardless of how our host interprets this sequence of bytes.

2.4 Converting Between Encodings

The C standard library has a number of functions to handle conversions between BE and LE encoding. Other languages have their own mechanisms, which you can look up if you need them. Here is a summary (header files might vary from system to system, such as):

Function	Size (bytes)	Input Encoding	Output Encoding	Header
`htons`	2	host	network	`arpa/inet.h`
`ntohs`	2	network	host	`arpa/inet.h`
`htonl`	4	host	network	`arpa/inet.h`
`ntohl`	4	network	host	`arpa/inet.h`
`htobe16`	2	host	big endian	`sys/types.h`
`htole16`	2	host	little endian	`sys/types.h`
`be16toh`	2	big endian	host	`sys/types.h`
`le16toh`	2	little endian	host	`sys/types.h`
`htobe32`	4	host	big endian	`sys/types.h`
`htole32`	4	host	little endian	`sys/types.h`
`be32toh`	4	big endian	host	`sys/types.h`
`le32toh`	4	little endian	host	`sys/types.h`
`htobe64`	8	host	big endian	`sys/types.h`

htole64	8	host	little endian	**sys/types.h**
be64toh	8	big endian	host	**sys/types.h**
le64toh	8	little endian	host	**sys/types.h**

Chapter 3

Git

Git has become the de-facto standard revision control system, so it's worth taking some time to familiarize ourselves with some basic, and not-so-basic, concepts. Most git commands can be done without Internet access, since they're purely local. We'll label the commands that potentially require network access, though the commands in this chapter can be run completely self-contained on any Posix-compatible host with `git` installed.

3.1 Installation

If you already have git installed, you can skip the rest of this section. We recommend installing git on any machine (laptop or desktop) on which you expect to work.

If you're on a Linux system, then it's somewhat likely that git is already installed; if not, you can use your favorite package manager to install it. Don't forget to install the documentation, if it's in a separate package!

If you're using MacOS, you have some options:

1. Download and install git from https://git-scm.com/download/mac
2. If you're using Homebrew, `brew install git git-sh` (search for "git" to see other potentially useful packages)
3. MacPorts also has a `git` package

If you're using Windows, you can download the git installer from

- https://git-scm.com/download/win

This includes both a GUI and a git bash shell.

3.2 Basic Git Operations

Here we have the basic commands that you'll use on a regular basis. You should become familiar with all of these. Before we start, we have to introduce the concept of a *repository*. This is a directory hierarchy that's managed as a single unit under source control. Generally, this is some software project, possibly the entire thing or a component (for very large projects). It can be anything that's predominantly text files, though. Many people keep documents that they're writing in git (or some other source control), especially when using LaTeX, HTML, docbook, or any other non-graphical text preparation systems.

3.2.1 init

This creates a repository, either in an empty directory, or one already containing code. Create a new directory, let's call it "testing":

```
mkdir ~/testing
```

Now, let's go into this directory and create a file:

```
cd ~/testing
date > created_on
```

So far, all we have is a directory with a single file in it, nothing special:

```
find .
```

The **find** command is an incredibly useful utility, and you'll learn new features of it for years to come, even when using it heavily. Run **man find** to read the documentation.

Now let's make this a git repository:

```
git init .
```

If we run our **find** command again, we see that there's now a directory called **.git** with lots of stuff in it. You can also run

```
ls -A
```

if you don't want to see the whole recursive list of files. There's one file in particular that we're going to look at later: **.git/config**

3.2.2 status

As you might guess from the name, this is going to tell you things about your repository and working directory. At this point, we need to go into terminology a little bit.

The repository is the collection of data currently under git's revision control. It's generally kept in a compressed format, for efficient storage.

The working directory, in contrast, is the set of "normal" files that you're working with. Some of these will be in the repository, and some won't be. Let's go back to our example repository and see how these relate.

Start by running

```
git status
```

You should see something like the following:

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        created_on

nothing added to commit but untracked files present (use "git add" to track)
```

This is telling us that the repository is empty (no commits – more on that later!), but the working directory contains files that aren't in the repository (Untracked files).

3.2.3 add

OK, so let's add something to the repository! We're going to shorten "repository" to "repo", because that's the term people most commonly use. The **add** command is what will tell git that you want to include a file in a commit:

```
git add created_on
```

You can specify a directory, or a wildcard, in your add command. The risk with these is that you end up with derived binary files (or log files) in your repo. These aren't useful, and binary generally can't be compressed by git, so it's wasteful of space. Try to avoid adding directories or using wildcards unless you really know what you're doing.

Now run **git status** again. You should see something like:

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   created_on
```

Note the little comment on unstaging files. If you add more than you'd intended to, this can save your bacon.

This is also how you "stage" modified files for a commit. That is, if a file is under revision control (it's in the repo), but the working directory has a newer version, you would use **git add** to include it in a commit.

3.2.4 commit

So far, we still don't have anything in our repo. That's what **`commit`** is for. Why are these separate? Let's say we have some new files to add, some to rename, and a few to delete (we'll talk about these last two later). We can run several git commands to stage the commit, without writing them to the repo yet. This gives us the chance to review what we're planning to do, and fix things as necessary. The commit is then an atomic unit of change to the repo. Consequently, we generally want to group closely related changes in a single commit. Don't be afraid to do multiple commits in a row – they're cheap!

So, what does a commit look like? There are a couple of common ways to do this (there are actually many options you can use):

```
git commit -m 'Committing my first file!'
```

or

```
git commit
```

The difference between these is this: Every commit must include a log message. This is how you know, at a high level, what the purpose of this commit is. By specifying the **`-m`** flag and a string, we're passing the log message on the command line. If we omit this, we're put into an editor, where we can add our message interactively. The first line is going to be a short summary, but the log entry itself can be as long as you need it to be. I often use the log message to keep track of things that still need to be done, or some additional information about the commit that's useful to know.

If you're using the interactive editor, just save the file and exit, and git will take care of the rest. Let's say we used the command-line message flag. Let's run **`git status`** again:

```
On branch master
nothing to commit, working tree clean
```

Ta-da!

3.2.5 log

So, we now have a repo with something in it. How do we know what the state of the repo is? The easiest way is with the **log** command:

```
git log
```

When I run this, I see the following:

```
commit 702223c70752248a5d54f16586f6501a47fd2e52 (HEAD -> master)
Author: Michael Marsh <mmarsh@cs.umd.edu>
Date:   Tue Jan 16 16:01:48 2018 -0500

    Committing my first file!
```

We can get more information with

```
git log -p
```

This gives you the log with "patches" that modify the repo from the previous commit to the one listed.

One thing you might have noticed is that the commit is a long hexadecimal number. This is a SHA-1 hash, which is what git uses to identify absolutely everything: files, directories, commits, etc.

3.2.6 clone [may require Internet access]

git lets you share a repo between users and machines. It does this very well, which is why it's so popular. The way you get a repo from elsewhere is by *cloning* it. Let's see this in action:

```
cd ~
mkdir another
cd another
git clone ~/testing
```

Take a look at **~/another/testing**, using the commands we've been using so far. Let's do even more! From **~/another**:

```
git clone ~/testing more_testing
```

Compare **~/another/testing** and **~/another/more_testing**. They should be identical! Here we've illustrated the ability to specify a destination directory for **clone**. If unspecified, the repo name of our source will be the name of the destination. This is especially useful when you want to make sure that the repository you just pushed to actually contains what you think, since you can have a second (clean) copy in another directory (see "pull", below).

Here, we've cloned a repo in a local directory. This is of limited usefulness, since generally you're going to want to clone repos stored on other machines. We'll deal with this later, but the general thing we'll see is a command like one of the following:

```
git clone https://example.com/repo_name
git clone user@example.com:repo_name
```

The latter is what we'll mostly use for repos that we're editing.

3.2.7 init –bare

We're now going to create another repo, this time slightly differently:

```
mkdir ~/testing2
cd ~/testing2
git init --bare .
```

What we've now done is create a *bare* repository. This is a repo without a corresponding working directory. Delete **~/another/testing** and **~/another/more_testi** and re-clone them from **~/testing2** instead of **~/testing**.

3.2.8 push [may require Internet access]

Go into **~/another/testing** (or **testing2**, depending on whether you provided a destination directory), and create a file. It doesn't matter what you call it, or what's in it. Add and commit it to the repo. Now run

```
git push
```

You should see something like:

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 205 bytes | 205.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /Users/mmarsh/classes/another/../testing2
 * [new branch]      master -> master
```

What we've just done is to send our commit to the repo that we cloned. In fact, this will send all local commits we may have that the cloned repo (called the "remote") does not yet have. The default remote is named "origin".

3.2.9 pull [may require Internet access]

Now go to ~/another/more_testing, and run

```
git pull
```

Take a look at the working directory and the git log. They should be identical to the repo and working directory from which we just pushed.

As with push, this will synchronize our local repo and working directory with whatever was newer at the remote.

You can also specify a source from which to pull, generally a remote. We'll talk about remotes later.

3.2.10 config

You may have noticed that your log messages have a rather generic-looking committer name and email address. You were probably also warned about this. The log message I showed you above, however, had a real name and email address. This seems like it would be really useful!

There are a couple of ways to set these. One of which is to edit the user's configuration file by hand. The other way is to run the **config** command:

```
git config --global user.name "Your Name"
git config --global user.email "your_email@example.com"
```

Any subsequent commits will now have more useful attribution. Make sure you do this anywhere you use git!

3.3 More Advanced Git Operations

You can get pretty far with the previous commands, but there's a lot you'll need to do beyond what these cover.

3.3.1 rm

Projects accumulate garbage. It happens. That means sometimes we need to get rid of a file. That's where **`rm`** comes in. Let's go back to **`~/testing`**. Now run

```
git rm created_on
```

What does **`git status`** tell us? Let's commit it now:

```
git commit -m "removed created_on"
```

The file is now gone from your working directory, and the repo! But only sort-of…

A key feature of git (or any revision control) is the ability to *revert* to previous versions of the repo. Run **`git log`**, and you'll see something like:

```
commit eef4f0ba06411f678bb741aaf6d06d580d82011a (HEAD -> master)
Author: Michael Marsh <mmarsh@cs.umd.edu>
Date:   Tue Jan 16 16:32:25 2018 -0500

    removed created_on

commit 702223c70752248a5d54f16586f6501a47fd2e52
Author: Michael Marsh <mmarsh@cs.umd.edu>
Date:   Tue Jan 16 16:01:48 2018 -0500

    Committing my first file!
```

The earlier commit is still there! Let's not worry about this just yet.

3.3.2 mv

Sometimes you need to rename a file. Let's do the following (in **~/testing**):

```
touch foobar
git add foobar
git commit -m "adding foobar"
```

Now we have a file named **foobar**. Let's say we really wanted to just call it **foo**. There are two ways we can do this. The hard way:

```
mv foobar foo
git add foo
git rm foobar
```

If you run **git status**, you'll see:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

          renamed:    foobar -> foo
```

git is smart enough to see that there used to be a file that looks extremely close (or identical) to the new file, so it was probably just renamed! We can tell git this explicitly with a single command:

```
git mv foo bar
```

Now we see essentially the same result. The file has both been renamed in the working directory, and a commit staged renaming it in the repo.

Commit this to update the repo. Don't forget to push your commit if you're working with a remote!

3.3.3 .git/config

There's a lot we can configure about a repo. All of this can be done with command-line utilities, but it's often easier to go right to the configuration file. This is often the only file in the **.git** directory you'll have to worry about. Here's what my version of ~**/testing/.git/config** looks like:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
```

This isn't very interesting. Let's look at ~**/another/more_testing/.git/config**

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = /home/vmuser/testing2
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

There's a lot more going on here! In particular, we've defined a remote and a *branch*. We already saw that a remote is another repo with which we're going to synchronize. Let's look at this a bit more.

3.3.4 remote

You can have many remotes for a local repo. In most cases, you only have one. In this course, because we're Computer Scientists, we're going to usually have two, and sometimes three!

The **`git remote`** command tells you the names of your defined remote repos. More useful is to add the **`-v`** flag:

```
git remote -v
```

should produce something like

```
origin /home/vmuser/testing2 (fetch)
origin /home/vmuser/testing2 (push)
```

You can add another remote to your **`.git/config`** by copying an existing block. Let's look at the remote defined in the config file in **`more_testing`** again:

```
[remote "origin"]
      url = /home/vmuser/testing2
      fetch = +refs/heads/*:refs/remotes/origin/*
```

Now, let's create another bare repo:

```
mkdir ~/testing3
cd ~/testing3
git init --bare .
cd ~/another/more_testing
```

We can define this as another remote, by copying and modifying the block above. Our **.git/config** file will now look like:

```
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true
      ignorecase = true
      precomposeunicode = true
[remote "origin"]
      url = /home/vmuser/testing2
      fetch = +refs/heads/*:refs/remotes/origin/*
[remote "other"]
      url = /home/vmuser/testing3
      fetch = +refs/heads/*:refs/remotes/other/*
[branch "master"]
      remote = origin
      merge = refs/heads/master
```

Note the changes we made: the remote name in the block definition, the url, and the **refs** in the **fetch** parameter. Don't worry about what these mean, just make sure that the remote name on the **fetch** line matches the name of the remote.

Now, run

```
git push other master
```

Congratulations! You've just created a fork!

3.3.5 branch

You may have noticed that git frequently refers to **`master`** (sometimes **`main`**), and sometimes refers to it as a "branch". Branches are another fundamental concept in git. It's like forking, but it's completely internal to your repo (branches can be pushed or pulled independently). Branches are also very lightweight – it's another SHA-1 hash stored somewhere that says, "this commit is the head of another branch." We haven't talked about heads yet, but they're essentially just the latest commit on a branch, whether local or remote. That is, until you push, your local head is newer than the remote's head. Once you push, they're identical (until someone else pushes or you make another commit).

When working on your own, you'll often only have one branch, by default named either **`master`** or **`main`**. Sometimes you want to create new branches, though. This can be very useful if you're experimenting with some changes, and you don't want to mess up your master branch. If the changes don't work out, you can just abandon or delete that branch, and no harm done. We'll get to merging branches in a bit.

When working with others, it's often helpful to use separate branches, either per-developer or per-feature under development. That way, you're less likely to step on each other's toes. There's also a model of development, called Git-Flow, where you have a master branch as your "production" version, a branch for each feature under development, short-term branches for bugfixes, and a "development" or "dev" branch to merge features and bugfixes back together before merging them into master. The reason for the dev branch is so that you can test the changes to make sure that nothing else broke in the process. Very occasionally, you might have "hotfix" branches that get merged directly into master; those are for critical bugs in production.

So, now that we've got the motivation behind branches, how do we create one? It's really pretty easy:

```
cd ~/another/more_testing
git branch test_branch
```

You've now created a new branch, named **test_branch**, which is currently the same as the master branch. You can verify the existence of this branch:

```
git branch
```

Note that there's an asterisk beside **master** – that means we're still on the master branch, not our new branch. We'll get to that soon.

You can also delete a branch:

```
git branch -d test_branch
git branch
```

See? The branch is now gone!

You can also specify a different starting point for a branch, whether a branch, a commit ID, or a *tag* (which we'll see later).

I rarely create branches this way, because there's a neat one-step way to create a new branch and make it active. That's our next command.

3.3.6 tag

Sometimes you don't need to create a new branch, but you *do* want to keep track of a particular commit. This commonly happens when you make a public release of your code, so that you know exactly what code was in that release. For this, we often use tags.

Where a branch is an alias for the head of a chain of commits, a tag is an alias for a particular commit. We can create tags easily with

```
git tag v1_0
```

This creates a git object named **v1_0** (for our 1.0 release) that is an alias to the current commit. You can specify a different commit after the tag name, if you like.

Once you have created the tag, you can treat it in much the same way as a branch. That is, it is something you can refer to by name, rather than having to use an explicit commit identifier. You can see all of the available tags with

```
git tag -l
```

You can update an existing tag with

```
git tag -f v1_0
```

Otherwise, creating a new tag with an existing name will fail. To delete a tag, you would run

```
git tag -d v1_0
```

Tags are not pushed to remote repositories by default. If you want your tags to be pushed, you need to run

```
git push --tags
```

What we have described so far are *lightweight* tags, which might be enough for your needs. You can also create an *annotated* tag, which looks more like a commit, in that it has an author, date, and commit message. To create an annotated tag:

```
git tag -a -m "release version 1.0" v1_0
```

As with **commit**, if you omit the **-m** option, you will be presented with an editor to enter the message.

3.3.7 checkout

This is how you control what version of the repo your working directory is configured to. Let's create **test_branch** again, and then use **checkout** to make it active:

```
git branch test_branch
git checkout test_branch
git branch
```

The asterisk should now be beside **test_branch**, indicating that we're currently working on that branch. Do the following:

```
touch bar
git add bar
git commit -m "adding bar"
git log
```

You should see your latest commit has been applied to **test_branch**, not **master**. Further, you should see that the previous commit is labelled with **master**, **origin/master**, and **other/master**. These last two indicate branches on your configured remotes.

Now run:

```
git push origin test_branch
git log
```

See? The remote **origin** now has your branch on it! Now, let's do the following:

```
ls
git checkout master
ls
git checkout test_branch
ls
```

What do you see? If we had different versions of any files in the two branches, we'd see those changes appear and disappear as we checkout one branch or the other.

I mentioned the one-liner to create and switch to another branch. We do this with **checkout**:

```
git checkout -b another_branch
```

You're now working on **another_branch**, and it's identical to the branch you were just on. This is probably the most likely way you'll create a branch, when you use them.

3.3.8 update-index and ls-files

These are two very useful commands when you're writing scripts and storing them in git. I have often seen students commit a script to git, and then ask why it's not running when I am doing a grading pass. The reason is that they haven't told git that the script file should be executable. If you set the execute bit before adding the file to a commit for the first time, git will pick this up automatically. If, however, you add and commit it before making it executable, you have to go back and tell git to record the executable bit for the file.

Fortunately, this is easy to do using **`git update-index`**. This command tells git that you want to change some of the metadata about one or more files, as opposed to modifying the contents of the file. Let's say we have **`foo.sh`**, but git doesn't (yet) know that it should be an executable bash script. We can fix this by running

```
git update-index --chmod=+x foo.sh
```

This behaves like the standard Posix **`chmod`** (change mode) command, which is used to change permissions on a file or directory. In particular, we're telling it to add the executable (x) bit. This also adds the file to a commit, so you'll then just need to do the actual commit (and push, if necessary.

Let's say you're not sure if the executable bit is set. That's where **`git ls-files`** comes in. If you run

```
git ls-files --stage foo.sh
```

it will show you the index entry for the file. The first column will be an octal number like **`100644`** or **`100755`**. The last three digits tell you (respectively) the owner, group, and other permissions. The first bit is whether the file is readable, the second writable, and the third exectuable. That means **`100644`** means the

owner can read and write the file, and everyone else can only read it. **100755** means the executable bit is set for everyone.

3.3.9 tag

Sometimes you want to mark a particular commit for later reference, and you don't want to change this reference as development continues. You can do this with a *tag*, which is just a name attached to a commit. You can use these tags any time you would use a commit or other reference. You can even create a branch off of a tag.

We're not going to go into detail about this command, but it's useful to know about. Run **man git-tag** for more information, if you're curious.

3.3.10 merge

Say we're developing on branches. Eventually, we're going to want to combine at least some of those branches back together. We do that with the **merge** command. If you're on one branch, you can easily merge in the commits from another. We're currently on **another_branch**, so let's switch to **master**, and merge **another_branch** into it.

```
git checkout master
git merge another_branch
```

Use **git status**, **git log**, and **ls** to see what's changed in the repo now. You can merge a branch, commit, tag, or any other kind of reference. See the man pages for lots of detail.

Occasionally, if two people have modified the same file, you'll have problems when trying to merge. This may happen when you pull from a remote, as well.

We'll discuss merge conflicts later.

3.3.11 fetch [may require Internet access]

This is a basic command to get the changes to the repo from your remotes, without merging them in. To fetch a single remote:

```
git fetch other
```

To fetch all remotes:

```
git fetch --all
```

The **pull** command is actually a fetch and merge rolled into one, so (on the branch **master**)

```
git pull other
```

is equivalent to

```
git fetch other
git merge other/master
```

The **fetch** command is useful if you just want to make sure you have the remote repos downloaded to your local machine. This can be important if you're going to be working without an Internet connection.

3.3.12 Dealing with conflicts

Let's make some simultaneous edits on separate branches, and then try to merge them together. When you're working on a group project, this is likely to occur at some point, unless you're extremely careful.

```
git checkout master
echo "This is a file" > file1
git add file1
git commit -m "adding file1"
git checkout test_branch
echo "This is my file" > file1
git checkout master
git merge test_branch
```

You should see a message like:

```
Auto-merging file1
CONFLICT (add/add): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

So, how do we deal with this? First:

```
cat file1
```

It should look like:

```
<<<<<<< HEAD
This is a file
=======
This is my file
>>>>>>> test_branch
```

This tells you that the current branch **HEAD** has one version of the file contents, and **test_branch** has another. You may see several of these in each file with a conflict.

The important thing you need to do is to manually resolve all of the conflicts. You can search a document for "«««" as an easy way to find them. Everything from the "«««<" line to the "»»»>" line must be replaced with whatever you determine is correct. Often, this will be removing one of the conflicting changes, leaving the other intact. Sometimes you will have to do something more complicated. Let's say the version on test_branch is the one we really wanted. We'd then replace that entire block above with

```
This is my file
```

Save that, and run **git status**. You'll see that we're in a merge conflict, with both branches having added **file1**. Helpfully, git tells us what to do:

```
(fix conflicts and run "git commit")
```

or, if we decide this was a bad idea:

```
(use "git merge --abort" to abort the merge)
```

In this case, we've already examined and resolved the conflict, so we follow the instruction:

```
(use "git add <file>..." to mark resolution)
```

and run:

```
git add file1
git commit
```

Save the log message file, and exit. We now have completed our merge! See what the log shows.

3.3.13 reset

When things stop working, sometimes starting over is easier than trying to find and fix the error. One of the nice features of git is that you can "go back" to any of your commits. For example, if you want to reset your repository to the state it was in when you last committed, run:

```
git reset --hard HEAD
```

More generally, you can restore your local repository to *any* previous commit. To do this, find the commit hash of the commit you want to restore (e.g. via **`git log`**) and run:

```
git reset --hard <hash>
```

However, since this discards changes to all files in the repository, it's not ideal if only one or two files are broken, and the rest have changes you want to keep. Fortunately, git has a way to restore individual files. To reset **`file1`**, run:

```
git checkout [<commit hash>] -- file1
```

(Note: the 'commit hash' argument is optional. If you omit it, git will default to using the previous commit)

IMPORTANT: The ability to restore old versions can be incredibly useful, but only if you make regular commits. There *will* be a time when you'll need to reset your repository – if your last working commit was five minutes ago, this will be a minor setback; if it was five hours ago, you'll be much less happy. Remember: commit early, commit often!

3.3.14 gitk

This is not a core git command, but it's often installed alongside with git. gitk is a Tk-based graphical display for git. It can be extremely useful for exploring your repo, including remotes. The basic invocation is

```
gitk
```

This will show the entire history of your current branch, including other branches that were merged into it, any tags, and any relevant remotes. Play around with this for a more complex repo, say one you've cloned from github.

Often, it's useful to see more information than this provides. Try running

```
gitk --all
```

This will not only show the current branch's history, but *all* branches in your repo.

That should be enough to get you started with git. In fact, most of your tasks will use the commands we've gone through here, with few if no options provided. The man pages have a lot more information, starting with **`man git`**. For a command **`git foo`**, the manpage would be **`man git-foo`**.

Chapter 4

Linux System Administration

If you're just getting started with administrative commands, tread carefully! There's a *lot* you can screw up accidentally. If you're reading this to fix the problem of not being able to access a shared folder, go ahead and skip to the very end, but then come back and read the rest later.

4.1 Root

This is the normal administrative account; it has privileges which allow it to do nearly anything on the host. As such, when running as the root user, you need to be very careful about what you do. It is also called the "supervisor" or "superuser" account.

On older systems, there was typically a password for the root account, and you would open a shell as the root user (or login to the system) using this password. The `su` command is what you would use if you were already logged into the

system as a normal user, and wanted to "become root". We still use **su** occasionally, but we use it slightly differently, as we'll see later.

Now, it's more common for root not to have a password, so we have to have another way to become root. For this, we have the **sudo** command. Here's how it works (the **$** indicates a normal user command prompt):

```
$ wc -c /var/log/tallylog
wc: /var/log/tallylog: Permission denied
$ sudo wc -c /var/log/tallylog
64128 /var/log/tallylog
```

If you haven't run **sudo** recently, it will prompt you for *your* password. This is because you've been granted special permission to call sudo in the file **/etc/sudoers**

```
$ groups
vmuser adm cdrom sudo dip plugdev lpadmin sambashare wireshark docker vboxsf
$ grep %sudo /etc/sudoers
/etc/sudoers: Permission denied
$ sudo grep %sudo /etc/sudoers
%sudo   ALL=(ALL:ALL) ALL
```

Let's unpack this. First we list all of the permissions groups to which we belong. One of these is named **sudo**. If we look for this group (which is what prepending **%** denotes) in the file **/etc/sudoers**, we see that there's a matching line. However, along the way we discover that **/etc/sudoers** is itself only readable by root, so we have to use **sudo** to run **grep** over the file!

What does the line in **/etc/sudoers** mean? Here it is again:

```
%sudo   ALL=(ALL:ALL) ALL
```

If we run **man sudoers**, we can see all of the documentation, but we'll cut to the chase. First, **%sudo** means this is a rule for the permission group **sudo**, of

which we happen to be a member. If you leave off the `%`, then this would match on username instead.

Next, we have the hosts on which this is valid, since this file might be shared between a number of similarly configured hosts. In this case, we use the wildcard `ALL`. So far we have

```
%sudo   ALL
```

to indicate that on all hosts, the group `sudo` will have the specified permissions. After the `=`, we see `(ALL:ALL)`. This says that we're allowed to run as any user or group. We could have restricted this to `(man:tape)` if we wanted to grant this user or group permission to run as the `man` user (which can install or rebuild the database of manual pages) or the `tape` group (for access to a tape drive).

Finally, the last `ALL` says that when running as the provided user or group, we're allowed to run any command.

4.2 Running as Another User

The easiest way to run as another user is to use the `-u` option to `sudo`:

```
$ whoami
vmuser
$ sudo -u man whoami
man
```

Since you can run any command you like, you could also use this to start a shell:

```
$ sudo -u man /bin/bash
$ whoami
man
$ exit
exit
$
```

One thing that's important to note, however, is that each shell has an *environment*, which defines things like the directory path to search for executables, special options to pass certain programs, etc. Sometimes you don't want to carry the environment over to the new shell, but rather have the shell initialized as if the target user had just *logged in*. For this, we can use the **-i** flag. Compare the output of the following:

```
$ sudo /usr/bin/env
$ sudo -i /usr/bin/env
```

This can be very important in some circumstances, and often when starting a root shell, you'll want to include the **-i** flag. The following two commands end up being equivalent:

```
$ sudo -i /bin/bash
$ sudo su -
```

Here, the **-** option to **su** says to treat this as a login shell, just like **sudo -i**.

4.3 Managing Users and Groups

We're going to consider just a couple of things here: changing passwords and assigning users to groups. These should be the bulk of what you need to do.

To change your own password, run:

```
$ passwd
Changing password for vmuser.
(current) UNIX password:
```

You are prompted for your current password, then for the new password, and finally for the new password *again*, just to make sure you typed it correctly.

When run as root, you can change *another* user's password:

```
$ sudo passwd man
Enter new UNIX password:
```

Now you're only prompted for the new password, not the current one. This is because you're running it as the superuser.

We saw the **groups** command earlier, which lists your current groups. You can switch your currently active group (which on our VM defaults to **vmuser**) by running **newgrp**:

```
$ echo $GROUPS
1000
$ newgrp docker
$ echo $GROUPS
124
```

It is important to note that when you run **newgrp**, you open a new shell. That means your shell history will be gone, until you **exit** from that shell and return to your previous shell (and group).

Most of the time, you will not need to change your active group, but you might need to change the list of groups to which you belong. For this, you run the **vigr** command (as root):

```
$ sudo vigr
```

In particular, let's say we're on a VirtualBox VM with user **vmuser**, and consider the following line at the bottom of the file:

```
vboxsf:x:999:
```

If you see this line, it means the **vmuser** account will not be able to access shared folders. We can fix this by changing the line to:

```
vboxsf:x:999:vmuser
```

Now, when we save and quit, we'll see a message telling us to run **vigr -s**, which edits the *shadow* copy of the file. This is a security feature that hides some of the group details from the **/etc/group** file. Run:

```
$ sudo vigr -s
```

and make the same change:

```
vboxsf:!::vmuser
```

At this point, you will have to log out of your VM and back in (you don't need to restart it, though that will also work), and you'll now have access to shared folders!

Chapter 5

Network Commands

Most Posix systems will have the same set of commands you can call from the shell. Linux added the iproute2 suite of tools, which provide the same functionality with some additional bells and whistles.

Classic Posix Command	iproute2 Equivalent
`ifconfig`	`ip link`, `ip address`
`route`, `netstat -r`	`ip route`, `ip route get`

Most iproute2 subcommands can be abbreviated, such as `ip addr` instead of `ip address`. You will need to install the `iproute2` package to have access to these.

5.1 `ifconfig`

This command gives you information about all of the *network devices* on the host, whether configured or not. For example, on a VirtualBox VM:

```
$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::a00:27ff:fe9c:c99f  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:9c:c9:9f  txqueuelen 1000  (Ethernet)
        RX packets 319  bytes 195592 (195.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 258  bytes 41861 (41.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 46  bytes 3982 (3.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 46  bytes 3982 (3.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The first entry shows the **enp0s2** device. The **e** typically indicates that this is an ethernet device. The second entry shows the **lo** device, which is "loopback" or "local" (they're different names for the same thing).

Each one has a set of flags, in this case both devices are **UP** and **RUNNING**, which means they are ready to handle packets. The **mtu** is the maximum transmission unit for the connection to the next-hop through that device, which is the number of bytes a single packet can contain. 65536 is the largest size that an IP packet can be.

The address for the device is given by **inet** (or **inet6** for IPv6). The **netmask** defines the subnet size.

A device with a hardware address also has **ether** followed by that address.

The rest is mostly statistics for the device. **RX** means received data, and **TX** means transmitted data.

5.2 ip link

The iproute2 suite separates link (device) information from address information. For the former, you use the **ip link** command (in the PDF version this will be cut off, as it will for some of the later commands):

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
    state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:9c:c9:9f brd ff:ff:ff:ff:ff:ff
```

Compare these with the information from **ifconfig**. The format is slightly different, and there's a little information that **ifconfig** didn't provide, like **qdisc**. This is a Linux-specific feature that the Posix command doesn't know about.

5.3 ip address

We can get the address information, as well. In fact, this provides us with essentially everything that **ifconfig** provides, just formatted differently (and with the Linux additions):

```
$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
    state UP group default qlen 1000
```

```
    link/ether 08:00:27:9c:c9:9f brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 72430sec preferred_lft 72430sec
    inet6 fe80::a00:27ff:fe9c:c99f/64 scope link
       valid_lft forever preferred_lft forever
```

Now we have the hardware type and address first, followed by the IP and IPv6 addresses. Note that the addresses are given in CIDR notation. The only thing we don't have here are the RX and TX statistics.

5.4 Routing Tables

The **route** command prints out the routing table for the host. Note that some systems, like MacOS (which is otherwise Posix), have a **route** command that behaves a bit differently. Again, from a VM:

```
$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
```

You can also use **netstat -r**, which should provide identical output.

The iproute2 suite has **ip route** instead:

```
$ ip route
default via 10.0.2.2 dev enp0s3 proto dhcp metric 100
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15 metric 100
```

You can see that the information is all there, in different format, with the notable addition of the source address the host will use when sending a new packet based on that forwarding rule.

You can also ask what rule a particular address will use:

```
$ ip route get 10.0.2.10
10.0.2.10 dev enp0s3 src 10.0.2.15 uid 1000
    cache
```

5.5 Connecting Layers 2 and 3

We can examine the ARP cache with either of the following:

```
$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
_gateway                 ether   52:54:00:12:35:02   C                     enp0s3
```

This is the standard Posix command. The fields should be fairly self-explanatory.

```
$ ip neighbor
10.0.2.2 dev enp0s3 lladdr 52:54:00:12:35:02 DELAY
```

This is the iproute2 command (which can be abbreviated as short as **`ip n`**).

Obviously, on a VM there isn't much going on. Here's what we might see on a laptop:

```
$ arp -an
? (10.104.80.1) at 0:0:c:7:ac:0 on en0 ifscope [ethernet]
? (172.26.4.1) at 2:e0:52:40:3f:40 on en8 ifscope [ethernet]
? (172.26.5.254) at cc:4e:24:d1:b0:0 on en8 ifscope [ethernet]
? (224.0.0.251) at 1:0:5e:0:0:fb on en8 ifscope permanent [ethernet]
```

This is on MacOS, so we have to provide **`-a`**. The **`-n`** prevents addresses from being converted to hostnames, as with other network commands. The format is also slightly different, as you can see.

5.6 Existing Network Connections

The existence of **`netstat -r`** might provide a hint that **`netstat`** can do other things, as well. Here's a particularly useful command:

```
$ netstat -taun
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
udp        0      0 0.0.0.0:42966           0.0.0.0:*
udp        0      0 127.0.0.53:53           0.0.0.0:*
udp        0      0 0.0.0.0:68              0.0.0.0:*
udp        0      0 0.0.0.0:5353            0.0.0.0:*
udp6       0      0 :::50307                :::*
udp6       0      0 :::5353                 :::*
```

Here are what these flags mean:

Flag	Meaning
`-t`	match TCP sockets
`-u`	match UDP sockets
`-a`	show all, not just active, sockets
`-n`	just show numeric addresses/ports
`-p`	(not shown) the PID and process name (if permitted)

Note that on MacOS (and some other Posix platforms), **`-u`** tells **`netstat`** to display *Unix sockets*, which are for interprocess communication on the host.

5.7 Examining Connectivity

Our two workhorses here are **`ping`** and **`traceroute`**. Both take a number of options, but generally are called as

```
ping <destination>
```

Here's an example of **ping**:

```
$ ping -c 3 gizmonic.cs.umd.edu
PING gizmonic.cs.umd.edu (128.8.130.3) 56(84) bytes of data.
64 bytes from gizmonic.cs.umd.edu (128.8.130.3): icmp_seq=1 ttl=63 time=0.928 ms
64 bytes from gizmonic.cs.umd.edu (128.8.130.3): icmp_seq=2 ttl=63 time=1.57 ms
64 bytes from gizmonic.cs.umd.edu (128.8.130.3): icmp_seq=3 ttl=63 time=1.33 ms

--- gizmonic.cs.umd.edu ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 0.928/1.280/1.575/0.268 ms
```

This shows you we have a working network path to gizmonic, as well as the round-trip times. We lost no packets, and also get to see the statistics for the round-trip times.

traceroute lets us examine the path between us and a destination:

```
$ traceroute www.google.com
traceroute to www.google.com (216.58.217.100), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.2)  0.240 ms  0.152 ms  0.084 ms
 2  router-604.cs.umd.edu (172.26.4.1)  2.840 ms  4.756 ms  6.638 ms
 3  csnat03.priv.cs.umd.edu (172.26.127.103)  0.791 ms  0.851 ms  1.129 ms
 4  128.8.127.190 (128.8.127.190)  9.209 ms  12.075 ms  15.525 ms
 5  * * *
 6  129.2.0.178 (129.2.0.178)  2.105 ms  1.412 ms  1.492 ms
 7  128.8.0.160 (128.8.0.160)  1.617 ms  1.676 ms  1.676 ms
 8  128.8.0.13 (128.8.0.13)  2.628 ms  2.510 ms  2.507 ms
 9  206.196.177.200 (206.196.177.200)  2.887 ms  2.982 ms  3.128 ms
10  mbp-t1-1720.maxgigapop.net (206.196.177.109)  4.431 ms  4.330 ms  4.147 ms
11  * * *
12  216.239.54.106 (216.239.54.106)  4.098 ms  4.033 ms 72.14.235.32 (72.14.235.32)  5.498 ms
13  iad23s42-in-f4.1e100.net (216.58.217.100)  4.172 ms 108.170.246.3 (108.170.246.3)  4.860 ms
```

You can suppress the conversion of IP addresses to names with the **-n** flag. The way **traceroute** works is that it sends a small packet with a very small TTL

flag, and waits to see the ICMP Time Exceeded message, the sender of which is TTL hops away. Note that this is not reliable, as paths can vary from packet to packet.

5.8 Finding Other Hosts

Your simplest go-to for looking up a host's IP address given its name is to use the **host** command:

```
$ host gizmonic.cs.umd.edu
```

On older systems, you would use

```
$ nslookup gizmonic.cs.umd.edu
```

This has been deprecated on newer systems, so you should always use **host** when available. If you want more detail about the query and response, you can use **dig** instead:

```
$ dig gizmonic.cs.umd.edu
```

All of these have a number of options available to you to control what kinds of queries you perform. They are also able to perform *reverse lookups*. That is, given an IP address, they will tell you the (canonical) hostname:

```
$ host 128.8.130.3
$ dig -x 128.8.130.3
```

You can also find out who registered a domain with **whois**:

```
$ whois umd.edu
```

You can even find out who owns the subnet an address is in:

```
$ whois 128.8.130.3
```

Chapter 6

Docker

What is docker? You can think of it like a lightweight VM. It's really considerably different, because it uses the host processor, memory, network stack, etc., without creating virtual hardware. We can throw around terms like user-level filesystems, process groups, and network namespaces, but the important part is that you can run a self-contained guest Linux OS within another host Linux OS, with applications and all of their dependencies. The guest can only see the resources given to it by the host, so it provides some (minimal) level of security. It also means we can start a process from a known-clean state, so we have repeatability.

6.1 Installation

The first thing we need to do is install docker. If you're running Linux, there's a good chance that your package manager already has docker available (don't confuse it with a KDE package of the same name!), but for the most up-to-date version, you can download it from https://docker.com. One slight complication is if you're running Red Hat Enterprise Linux; Fedora and CentOS are just fine. There's a special version of docker that works with RHEL, but it doesn't work

as easily. At this point, you can ignore the rest of this section.

If you're running MacOS, then there's a download available from https://docker.com called Docker Desktop. It installs and runs easily. At this point, you can ignore the rest of this section.

If you're running Windows, life becomes more complicated. We're going to restrict ourselves to Docker Desktop under Windows 10. The next thing you need to do is ensure you're running at least version 2004, which supports Windows Subsystem for Linux version 2 (WSL 2).

- Go to https://aka.ms/wslstore and get a WSL Linux distribution. Ubuntu is a good choice.
- Install https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi
- In an Admin PowerShell, run the following:

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

  - You *might* need to restart at this point.
  - **`wsl --set-default-version 2`**
  - **`wsl --set-version Ubuntu 2`**

- There's a docker service icon at the bottom right (it's a whale) – right-click on it and select "Settings:

  - Enable WSL 2 as the engine, instead of Hyper-V. This allows docker to take advantage of the Windows/Linux integration in the OS.
  - Expose the TCP daemon on localhost without TLS.

- For convenience, I suggest doing the following in the Ubuntu shell:

```
ln -s "/mnt/c/Users/<your username>" winhome
```

That will allow you to access your Windows home directory from Ubuntu as **~/winhome/**.

You should now be able to run all docker commands from either PowerShell or the WSL Ubuntu (or other distribution) shell.

6.2 Docker Images

Let's start with the concept of an *image*. This is the self-contained guest Linux OS, which is configured to automatically run some process when it starts. Nothing is running in it – you can think of it like a hard drive.

The easiest way to get an image is to *pull* it from a *registry*. Docker has a default registry built in. We have, at times, used a course VM that is running Ubuntu 16.04 for a common baseline, and it turns out there's an image available with this OS on it! Here's the command to run:

```
docker pull ubuntu:16.04
```

Let's go through this command. **docker** is, of course, the utility we're using. The **pull** command tells us that we want to get something from a registry. In this case, we're getting the **ubuntu** image from the default registry. If we just left it at this, we'd get *all* of the ubuntu variants. Instead, we add **:16.04**. That tells docker we only want one image, and it's the one with the *tag* "16.04".

When the command completes, try running

```
docker images
```

You should see something like:

```
REPOSITORY        TAG                 IMAGE ID          CREATED           SIZE
ubuntu            16.04               2a4cca5ac898      28 hours ago      111MB
```

Most of this should be fairly self-explanatory. The image ID is another hexadecimal number, like with git, but it's clearly not a SHA-1 hash. It really doesn't matter what it is, other than a unique identifier for this image.

We can do a few things with this image, aside from running it. Try the following:

```
docker tag ubuntu:16.04 my_ubuntu
docker images
```

Note that we now see the same image ID twice, but with different names. By default, a repository (the tagless part of an image name) is tagged as **`latest`** if you don't specify one. Let's try specifying a tag, though:

```
docker tag ubuntu:16.04 foo:bar
docker images
```

The results should not be surprising.

We can quickly build up a lot of images we don't want anymore, so it's good to know how to clean these up. Let's get rid of our new tagged images:

```
docker rmi my_ubuntu:latest foo:bar
docker images
```

A common problem is that we'll end up reusing an old tag, leaving an image with no repository:tag name. These show up as **<none>:<none>**. We can get rid of all of these with the following bash one-liner:

```
docker images -a | grep none | awk '{print $3}' | xargs docker rmi
```

For the curious, feel free to read the man pages for **awk** and **xargs**.

The commands here are largely from an older version of docker. Now they're aliases to new-style commands. Here's the mapping:

Old Command	New Command
docker images	**docker image list**
docker pull	**docker image pull**
docker rmi	**docker image rm**
docker tag	**docker image tag**

6.3 Running an Image in a Container

Images are all fine and good, but we actually want to use docker to *do* something, which means we have to run these images. An image runs in a *container*. The container has system resources allocated to it, and runs a program or programs that exist in the image. A container runs a single image, but an image may be running in multiple containers.

Containers can also be started with various options, such as elevated privileges, mounted volumes, environment variables, and so on. The most basic invocation

is

```
docker run ubuntu:16.04
```

If you run this, you'll find that it pauses for a second or so, and then returns to the command line. If you want to see running containers, run

```
docker ps
```

You see headings, but probably no actual containers. Now, try

```
docker ps -a
```

Now we have something! Here's an example of what you might see:

```
CONTAINER ID    IMAGE           COMMAND         CREATED
  STATUS                          PORTS             NAMES
1b937126d5bc    ubuntu:16.04    "/bin/bash"     About a minute ago
  Exited (0) About a minute ago                     upbeat_archimedes
```

Let's parse this out:

- The container ID is a unique ID, like the image ID we saw before
- The image should be self-explanatory
- The command is what the container ran. In this case, it's just bash
- The created time is when the container was started
- The status tells us that this container exited, and is no longer running

- We have no ports bound, but if we did these would map from local network ports to network ports on the container
- The names are symbolic names used to refer to this container, and are synonyms for the container ID

By default, names are assigned randomly according to the pattern **<adjective>_<scientist>**

We can assign a name to the container, which is often useful:

```
docker run --name=bash_test ubuntu:16.04
```

This will behave similarly to the previous command, but if we run

```
docker ps -a
```

We'll now see our container named **bash_test** along with whatever random name our first container was assigned.

Usually, an image is defined to do something useful when run non-interactively. We can get interactive access to the container, though, as follows:

```
docker run -ti ubuntu:16.04
```

We've passed two new options to docker run. The **-t** option allocates a pseudo-TTY, and the **-i** option makes the container interactive. You should now have a shell on the container running as root! If you run **docker ps** in another terminal, you will see that the container status is "Up

When you're done playing around in this shell, exit to stop the container.

You will often want to have access to your computer's files in a container. You can do this with the volume mount option, **-v**. It takes two directories, separated by a colon. The first directory is an *absolute path* to a host directory, while the second is where this directory should be "mounted" in the container. For example,

```
docker run -ti -v "$(pwd):/mnt" ubuntu:16.04
```

will run the same interactive container as before, but now the **/mnt** directory in the container contains the directory in which you ran the docker command (it is the *exact* directory, so changes in the container appear on the host, and vice-versa). It is worth dissecting the argument a bit more. The double quotes are in case the host directory contains any spaces in it, which has become very common. The other thing that double quotes do, in contrast to single quotes, is process any shell expansions. Here, **$(pwd)** runs the command **pwd** (print working directory) and substitutes it into the string. Most Posix systems (such as Linux) define an empty **/mnt** directory so that you can mount remote directories or temporarily connected devices (though these often now appear under **/media**).

At this point, you probably want to get rid of these stopped containers. Run:

```
docker rm bash_test
docker ps -a
```

You'll still have the two randomly-named containers, but the one named **bash_test** should no longer be present. Remove the other two, as well.

We don't have to run the configured program in a container; we can run any command that's present on the image. Let's see this in action:

```
docker run ubuntu:16.04 /bin/date
```

That should print the date in the container. It's probably in UTC, while running **`/bin/date`** (or the equivalent) on your computer should print the date in your local time zone. You can also specify options:

```
docker run ubuntu:16.04 ls /var
```

Another very useful option is **`--rm`**, which will get rid of the container once it stops:

```
docker run --rm --name="rm_test" ubuntu:16.04 ls /var
```

We've once again been using old-style commands, which are aliases:

Old Command	New Command
`docker run`	**`docker container run`**
`docker ps`	**`docker container ls`**

6.4 Stopping a Running Container

A container might become unresponsive, or it might be a long-running service that you want to terminate. You can do this with either of the following:

```
docker kill <container>
docker stop <container>
```

stop is more graceful, trying SIGTERM first, and then SIGKILL. **kill** sends SIGKILL by default, but this can be overridden on the command line.

Old Command	New Command
docker kill	**docker container kill**
docker stop	**docker container stop**

6.5 Removing Stopped Containers

As with images, you'll tend to accumulate lots of stopped containers, unless you've run them all with the **--rm** option. Fortunately, we can get rid of these with

```
docker rm <container>
```

which is now an alias for

```
docker container rm <container>
```

6.6 Other Options for Running Containers

Here are some useful options you might want to use:

Option	Argument	Effect
--rm		removes container after exit
-ti		run interactively with a pTTY
-e	**<vars>**	set environment variables
-h	**<hostname>**	set the container's hostname

-p	**<hport>:<cport>**	map host's **<hport>** to container's **<cport>**
-v	**<hdir>:<cdir>**	mount host's **<hdir>** on **<cdir>**

6.7 Executing Commands in a Running Container

Sometimes you need to examine what's going on inside a container. That's where the **exec** command can come in handy. It's a lot like **run** but for a container, rather than an image. Here's a common thing you might want to do:

```
docker run --name=svc_instance my_service:latest
docker exec -ti svc_instance /bin/bash
```

What this does is to first start a container using the latest version of the image **my_service**, and name the container **svc_instance**, and then to execute an interactive bash shell on that container. You don't have to exec an interactive command, though. There may be times when you want to run something like:

```
docker exec svc_instance touch /var/cache/magic_file
```

in order to change the behavior of a running process. As with the other commands we've looked at, **docker exec** is now an alias for **docker container exec**.

6.8 Getting Process Output

Many processes send their output to STDOUT or STDERR. Since there's no TTY available to the process in a container, this output would generally be

lost. Docker saves this output for you, however, and you can retrieve these by running

```
docker logs <container>
docker container logs <container>
```

The first command is now an alias for the second command. There are a number of options, such as **`--since`** to limit the timeframe of the logs returned, **`-f`** to continue to follow the logs rather than just dumping their current contents and exiting, and **`-t`** to show timestamps at the beginnings of lines.

Chapter 7

Python

Python has two major version in current use: 2.7 and 3. In many cases, these behave identially, but there are some differences. Try things out in the interactive interpreter, which you can start by typing `python` with no arguments.

For functions, classes, and modules, there's a built-in help system. Simply type `help(item)` for the documentation on *item*.

An important note about python: *scope is indentation-based!* A change of indentation is a change of scope, and you cannot mix spaces and tabs. You should probably ensure that your editor uses spaces for indentation in python, and be careful of reflexively using the tab key for indentation unless you're sure your editor will replace tabs with spaces. You can break up long statements on multiple lines, but only if it's unambiguous that you're in the middle of a statement (such as within parenetheses or braces) or you use a line continuation character. I'm not going to tell you what the continuation character is, since it makes code ugly.

7.1 Terminal Output

print is a statement in python2.7, and a function in python3. You can treat it like a function in both, and it will generally do what you expect:

```
print('Hello world')
```

This will print a string to STDOUT.

Printing to STDERR, which is what you should do for debug messages, is more complex. There are two portable ways to do this:

```
import sys
sys.stderr.write('Hello world\n')
```

This uses the filehandle directly.

```
import sys
from __future__ import print_function
print('Hello world', file=sys.stderr)
```

This will ensure that the python3-style print function is present, even in python2.7.

7.2 Terminal Input

A simple way to read from STDIN is

```
import sys

while True:
    line = sys.stdin.readline()
    if '' == line:
        break
    print(line)
```

You can also use the **raw_input** or **input** functions, but this is where things get messy. In python2.7, **raw_input** reads a line from STDIN, while **input** calls **raw_input** and then evaluates the result as a python expression. In python3, **input** behaves like python2.7's **raw_input**, and there is no **raw_input** function. We can hack our way around this, though:

```
if 'raw_input' not in dir(__builtins__):
    raw_input = input

try:
    while True:
        line = raw_input()
        print(line)
except:
    pass
```

Note that both of these require some special way to handle the end of file: either testing against an empty string or handling an exception. A third way is:

```
import sys

for line in sys.stdin:
    print(line)
```

Note how much simpler this is! This is, in general, how we would read from any file.

7.3 Files

Python has an **open** function, that opens files for reading or writing, potentially in binary mode, and possibly (for writing) in append mode. See **help(open)** for details. We generally want to use these with the **with** keyword, which provides automatic file closing and other cleanup:

```
with open('input_file.txt') as in_file:
    for line in in_file:
        pass # This is a no-op

with open('output_file.txt', 'w') as out_file:
    out_file.write('Hello world!\n')

with open('output_file.txt', 'w') as out_file:
    print('Hello world!', file=out_file)
```

7.4 Scalar Types

Python has integers and floating-point numbers. Unlike many languages, all integers are arbitrary-length, as long as you have enough memory to represent them.

Python2.7 has strings, which do double-duty as byte arrays. In python3, there is a separate bytes type, and strings are utf-8 encoded by default. Most of the time, you can ignore these differences. Strings can be single-quoted or double-quoted. There isn't much reason to prefer one over the other, though if you want to use the quote character in the string, you'll have to escape it:

```
s1 = "How's it goin'?"
s2 = 'How\'s it goin\'?'
```

7.5 Iterable Types

Python also has lists and tuples. The difference is that a list can be modified, a tuple cannot:

```
list1 = [ 1, 2, 3, 4 ]  # Initialize a list with elements
list2 = []              # Create an empty list
list2.append(1)         # Append an item to the list
list3 = list()          # Another way to create an empty list
list3.extend([1,2,3])   # Add multiple items to the list

tuple1 = ( 1, 2, 3, 4)  # Create a tuple with explicit entries
tuple2 = tuple(list1)   # Create a tuple from another iterable
```

There are other iterable types, defined by particular methods they have.

Lists and tuples can be accessed by indexes:

```
list1[0]   # first element
list1[-1]  # last element
```

They also support *slicing*:

```
list1[1:3]    # returns [ list1[1], list1[2] ]
list1[2:]     # returns [ list1[2], ..., list1[-1] ]
list1[:3]     # returns [ list1[0], list1[1], list1[2] ]
list1[0:3:2]  # returns [ list1[0], list1[2] ]
list1[0::2]   # returns all even-indexed entries
list1[::2]    # same
```

You can iterate over the elements of a list or tuple:

```
for v in list1:
    print(v)
```

Note that a string (or byte string) can also be indexed like a list and iterated over.

If you want to create an iterable of integers, you can use the **range** function:

```
stop_val  = 10
start_val = 1
step_val  = 2
range(stop_val)                     # range of ints from 0 through 9
range(start_val,stop_val)           # range of ints from 1 through 9
range(start_val,stop_val,step_val)  # odd ints from 1 through 9
```

7.6 Dictionaries

A python dictionary, or dict, is a map type.

```
d = {}                  # Create an empty dict
d = dict()              # Create an empty dict
d = { 'a': 1, 'b': 2 }  # Create a dict with initial values
```

Keys and values can be of any type, and python does not require keys or values to be uniform in type:

```
d = dict()
d[1] = 'a'
d['a'] = 2
```

Dictionaries are also iterable, though the iterator will be the keys, in some order:

```
for k in d:
    print(d[k])
```

dict has a number of useful methods (see **help(dict)** for more):

Method	*Returns*
`keys()`	an iterable containing the keys
`values()`	an iterable containing the values
`items()`	an iterable containing (key,value) tuples
`get(k)`	value for key k, or None if not present
`get(k,x)`	value for key k, or x if not present
`pop(k)`	value for key k, removing entry from dict

7.7 None

Python has a special type called `NoneType`, which has a single instance, named `None`. This is roughly python's equivalent of null.

7.8 List Comprehensions

Python has some functional programming elements, one of which is list comprehensions. Here's a simple example:

```
[ x**2 for x in xs ]
```

This takes a list of values named `xs` and returns a list of the values squared. These can be combined extensively:

```
[ x*y for x in xs for y in ys ]
```

The order can matter:

```
d = dict()
d['a'] = [ 1,2,3 ]
d['b'] = [ 4,5,6 ]
d['c'] = [ 7,8,9 ]

print([ x for k in d for x in d[k] ])
```

7.9 Formatted Strings

The simple way to construct a formatted string is to use the string class's format function:

```
s1 = 'This is {} test'.format('a')
s2 = 'The square of {} is {}'.format(2,4)
s3 = '{1} is the square of {0}'.format(2,4)
s4 = 'The first 5 powers of {x} are {pows}'.format(
         x=2,
         pows=[2**e for e in range(1,6)]
        )
```

7.10 Control Flow

Like any good language, python has a number of control flow expressions. Here are a few:

```
if boolean_expression:
    do_something
elif boolean_expression_2:
    do_something_else
else:
    do_default_thing
```

Both **elif** and **else** are optional.

```
for x in xs:
    do_something
```

We've seen this before; it's a simple for loop

```
while boolean_expression:
    do_something
```

In all of these, **`boolean_expression`** is just something that evaluates to **`True`** or **`False`** (python's boolean constants). Python will coerce things:

- **`0`** is **`False`**
- any other number is **`True`**
- **`''`** (empty string) is **`False`**
- any other string is **`True`**
- **`None`** is **`False`**
- **`[]`** (empty list) is **`False`**
- any other list is **`True`**

7.11 Combining Lists

We've already seen **`list.extend()`** as a way to append an entire list to another. Sometimes we want to do other things, though. Say we have a list of items, and we'd like to do something that involves their list index. We could do this:

```
for i in len(xs):
    print('item {} of xs is {}'.format(i,xs[i]))
```

There's another way we can do this, though:

```
for (i,v) in zip(range(len(xs)), xs):
    print('item {} of xs is {}'.format(i,v))
```

In this case, it doesn't seem to buy us much, but if we've read in two sequences from two different sources, but we know they should correlated, then we could use:

```
for (a,b) in zip(a_list, b_list):
    do_something
```

The **zip** function can take multiple sequences, and will truncate the resulting tuple to the length of the shortest sequence.

7.12 Functions

Python has functions. It even has anonymous functions. Let's start with normal functions:

```
def my_func():
    pass
```

This defines a function named **my_func** that takes no arguments and does nothing, returning None.

```
def my_func(a):
    pass
```

Now we've added an argument to our function. Arguments can be passed based on position or name:

```
my_func(1)
my_func(a=1)
```

The latter is nice, because you don't have to worry about argument order:

```
def my_func(a,b):
    pass
my_func(b=1,a=2)
```

A common python idiom is to define very flexible functions like:

```
def my_func(a,b, *args, **kwargs):
    pass
```

This means we can provide additional positional parameters, which are then captured by **`*args`**, as well as named (keyword) arguments, which are captured by **`**kwargs`**. In this case, **`args`** is a tuple, and **`kwargs`** is a dict.

A function can return a value. If there is no return statement, the return value is None:

```
def my_func():
    return 'a'
```

What about anonymous functions? We define these with the **`lambda`** keyword:

```
f = lambda x: x**2
f(2)
```

This doesn't look like it gives us a lot of advantages over named functions, but it can be extremely handy:

```
num_output = map(lambda x: int(x,16), output)

def my_func(a,b):
    return a*b
f = lambda a: my_func(a,2)
```

7.13 Classes

Without going into a lot of detail, python has a rich type system. Here's a simple class:

```
class Foo(object):
    def __init__(self,a):
        self.a = a
```

This defines a type **`Foo`** and a constructor that takes two values. Here's how we create an instance:

```
foo = Foo(1)
```

By calling the class name as a function, python automatically makes this a call to the **`__init__`** method, with the newly allocated instance as the first argument, named **`self`** by convention.

Other methods can be defined similarly. Any instance method should have **self** as the first argument. Methods are otherwise almost identical to normal functions.

Python is duck-typed. That is, if it looks like a duck and acts like a duck, it's a duck. When you use a value, if it conforms to the expected interface, you're good.

You can query an object for its methods and data elements:

```
dir(foo)
foo.__dict__
```

7.14 Modules

A module is a python library. We've already used the **sys** and **__future__** modules. To use a module **foo**, you need to import it:

```
import foo
```

Now anything defined in **foo**, say a method **bar**, can be accessed through **foo**'s namespace:

```
foo.bar
```

We can also import things into the current namespace:

```
from foo import bar
from foo import *
```

The first line means we can reference **bar** without **foo.**, but the second means we can reference *everything* in **foo** without the namespace. This is generally a bad idea, because it makes it less clear where a function or class comes from. Some packages work much better with this type of import, however, like **scapy**:

```
from scapy.all import *
```

How do we create a module? We'll keep it easy, and only consider single-file modules. Feel free to look up more complex modules. If you want to create a module named **foo**, you would simply create a file named **foo.py**, and define functions, classes, and variables in it as normal. Now, when you import **foo**, all of those will exist within **foo**'s namespace.

foo.py:

```
def bar(a):
    print('foo: {}'.format(a))
```

top-level script:

```
import foo

foo.bar(3)
```

7.15 Useful Modules

The **sys** module is the one you're most likely to import. It has a lot of functions, but one of its most useful elements is the **sys.argv** list. This contains the positional parameters to the script, in the order provided on the command line. The first element is the script name.

```
import sys
for arg in sys.argv:
    print('We were called with argument {}'.format(arg))
```

The **random** module is also very useful; it provides random numbers:

```
import random
r = random.Random()
r.choice(['a','b','c']) # choose a random element from the list
r.sample(range(100),5)  # choose 5 unique elements from [0,100)
r.randint(5,10)         # choose an integer in the range [5,10]
r.uniform(5,10)         # choose a float in the range [5,10)
r.gauss(75,10)          # choose a gaussian-distributed float with mean
                        # 75 and standard deviation 10
```

The **subprocess** module lets you call other processes, potentially capturing their output. We won't go into how to use it here. See

- https://docs.python.org/2/library/subprocess.html or
- https://docs.python.org/3/library/subprocess.html,

depending on which version of python you're using.

Finally, the **argparse** module is a great way to process command-line arguments, if you need something fancier than just **sys.argv**. Here's an example to illustrate:

```
from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument('-s', '--students',
                    dest='students',
                    default='enrollments.json',
                    help='JSON file containing the student enrollments'
                    )
```

```
parser.add_argument('-g', '--groups',
                    dest='groups',
                    default='teams.json',
                    help='JSON file containing the student groups'
                    )
args = parser.parse_args()

students = list()
with open(args.students) as f:
    students = json.load(f)
```

See the documentation for details; there's a lot to see here.

7.16 A Complete Script

This doesn't do much useful, but it should work, when saved to a file and made executable:

```
#! /usr/bin/env python

import random
import sys

count = int(sys.argv[1])
min = int(sys.argv[2])
max = int(sys.argv[3])

r = random.Random()

nums = [ r.uniform(min,max) for _ in range(count) ]

print('Generated {n} random numbers between {a} and {b}'.format(
   n=count,
   a=min,
   b=max ))
print('Values: {}'.format(nums))
```

Chapter 8

Python Scapy

The **scapy** module is extremely useful, but the documentation is somewhat lacking. Consequently, here is a simple cookbook of handy **scapy** recipes. Note that this chapter assumes you are familiar with networking, the OSI model, and packet formats.

8.1 Importing Scapy

Unlike most modules, **scapy** requires a global import to be useful:

```
from scapy.all import *
```

This imports all exported symbols from the **scapy.all** submodule into the global namespace. The rest of our examples assume your program has done this.

8.2 Reading and Writing Packet Capture Files

Your life will be easiest if all of your captures are in pure pcap format, not a format like pcap-ng. Wireshark, tshark, and dumpcap will *generally* produce pcap, unless you capture on all interfaces, in which case you will get pcap-ng files. That is, unless you override the default behavior. For **dumpcap**, which is our recommended way to capture packets (when feasible), the **-P** option will force normal pcap output.

Having said all that:

```
frames = rdpcap('file.cap')
```

This opens a pcap file named **file.cap** for reading, and returns an iterable, which we've called **frames**, because it potentially contains layer-2 frames, rather than layer-3 packets. That will depend on the capture file, however.

We can iterate over these as follows:

```
for f in frames:
    pass
```

This loop does noting (**pass** is a nop in python).

To write packets to a file, we would call:

```
wrpcap('outfile.pcap', pkts)
```

pkts may be packets or frames, and should be an iterable, such as a list.

8.3 Dissecting Packets and Frames

Scapy stores everything in **`dict`**-like objects, which is handy. The objects are actually built as a series of *layers*. Consider:

```
for f in frames:
    if IP not in f:
        continue
    pkt = f[IP]
```

First, we verify that there's an IP layer in this object, and if not we skip to the next one. Then we get the IP layer of **`f`**, which may be identically **`f`** or it may be a layer (at any depth).

We can also do more complex things, skipping over layers we don't care about:

```
for f in frames:
    if DNS not in f:
        continue
    d = f[DNS]
```

This might be a DNS layer within a UDP layer within an IP layer within an Ether layer. The nice thing is that we don't have to care.

At a given layer, there are a number of *fields* that we can access:

```
for f in frames:
    if DNS not in f:
        continue
    d = f[DNS]
    d.opcode
```

The easiest way to get a feel for what's in a scapy object is to call the **`display`** method:

```
for f in frames:
    f.display()
    if DNS in f:
        f[DNS].display()
```

The first call will print (to stdout) all of the layers, including their fields, while the second will only print information about the DNS layer.

8.4 Creating Scapy Objects

Scapy has fairly normal constructors:

```
pkt = IP(dst='1.2.3.4')
```

It also has a layering operator:

```
pkt = IP(dst='1.2.3.4')
udp = UDP(dport=123)
p = pkt/udp
pkt.display()
udp.display()
p.display()
```

We can simplify this:

```
pkt = IP(dst='1.2.3.4')
udp = UDP(dport=123)
pkt /= udp
pkt.display()
```

We can even simplify it further:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)
pkt.display()
```

Here are some layers that might interest you:

- **Ether**
- **IP**
- **ICMP**
- **UDP**
- **TCP**
- **DNS**
- **DNSQR**
- **DNSRR**
- **Raw**

What's this **Raw** layer? It's literally a raw sequence of bytes:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/Raw('This is a test')
pkt.display()
```

This just wraps the bytes in an appropriate scapy layer object, and we can shorten this:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/'This is a test'
pkt.display()
```

We can also nest things further:

```
pkt = IP(dst='1.2.3.4')/UDP(dport=123)/IP(
       src='2.3.4.5',dst='3.4.5.6')/ICMP()/'This is a test'
pkt.display()
```

8.5 Sending and Receiving Packets

Finally, how do we connect to the actual network, rather than just working with files?

If you're creating packets at layer 3 (that is, starting from the IP layer), you can just call:

```
send(pkt)
```

You can send and then wait for a response, as well:

```
new_pkt = sr1(pkt)
```

Here, **`new_pkt`** is the response received.

To just receive packets from an interface:

```
def my_callback(pkt):
    pass

sniff(iface=None, count=0, prn=my_callback)
```

Specifying **None** for iface (the default) captures on all interfaces. A count of 0 (the default) captures forever; nonzero will stop after that number of packets have been received. See **help(sniff)** for more parameters, including filters.