An Illustrated Guide to Shell Magic: Standard I/O & Redirection

Created by Brennen Bearnes

https://learn.adafruit.com/basic-shell-magic

Last updated on 2021-11-15 06:24:21 PM EST

Table of Contents

Overview 5

Input & Output 6

Standard I/O & Pipes 8

- Standard Streams 8
- Building a Simple Pipeline 10

Redirection To & From Files 11

- Writing to a File 12
- Appending to the End of a File 12
- Reading from a File 12
- A Common Gotcha: Changing a File In-Place 13

Standard Error & Exit Codes 15

- The Discontents of Error Messaging 15
- Exit Status 16

Concluding Remarks: Composability 17

Overview

This guide assumes you have access to the Bash shell on a GNU/Linux computer. Most of the examples use a Raspberry Pi running Raspbian. If you haven't, you should start with What is this "Linux", anyhow? (https://adafru.it/jDZ), What is the Command Line? (https://adafru.it/sdo), and An Illustrated Shell Command Primer (https://adafru.it/CeI).

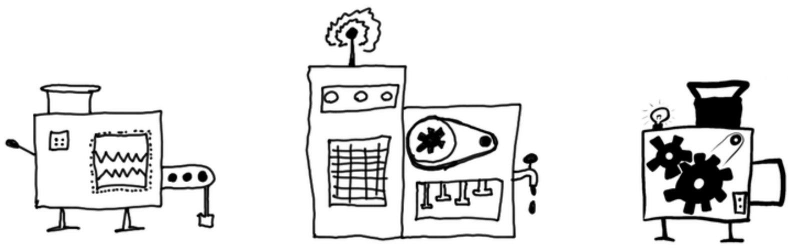
Now that we've covered some basic shell commands, let's shift focus a little and talk about some of the features that bind those commands together into a useful working environment. In this guide, we'll cover:

- standard input and output
- combining commands with pipes
- redirecting output to and from files
- standard error and exit codes

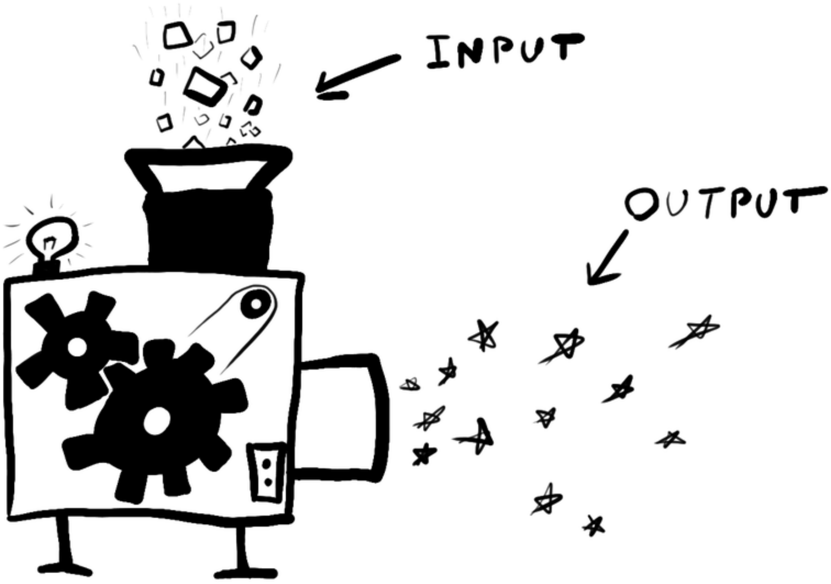
While these may sound like dry topics, it's here that the command line begins to come into its own as a tool and environment for solving problems. Pull up a terminal and read on!

Input & Output

By now, you've probably read about a number of commands available in the shell. For example, `ls` shows you a list of files (https://adafru.it/Cf4); `cat`, `head`, and `tail` look inside files (https://adafru.it/Cf5), and so on.

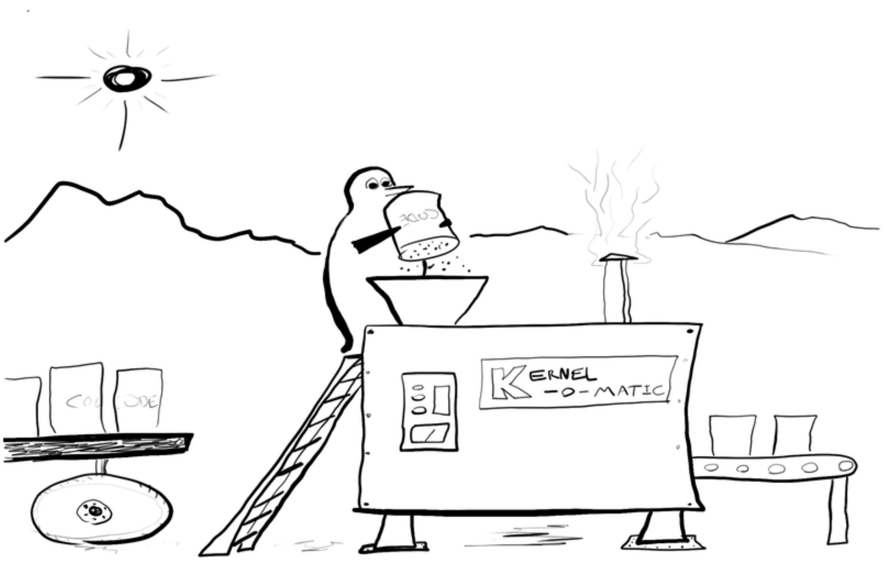


Let's imagine for a second that each of those commands is a kind of little machine in a workshop or on a factory floor. It's actually a pretty good metaphor. Shell commands are generally little self-contained programs, and a program isn't too hard to think of as a machine. Each machine takes some stuff as input, chugs along for a while, and produces some stuff as output.



So you have lots of special-purpose machines at your disposal, and they do all kinds of useful stuff. In addition to the ones we already talked about, there are commands to find files, search for text and transform it, look up words, convert units and formats, scan networks, generate pictures - most of the stuff you can think of to do with a computer, really.

If you've got machines to handle a bunch of different tasks, you can get all kinds of work done. And yet... It can get awfully tedious moving things between them.



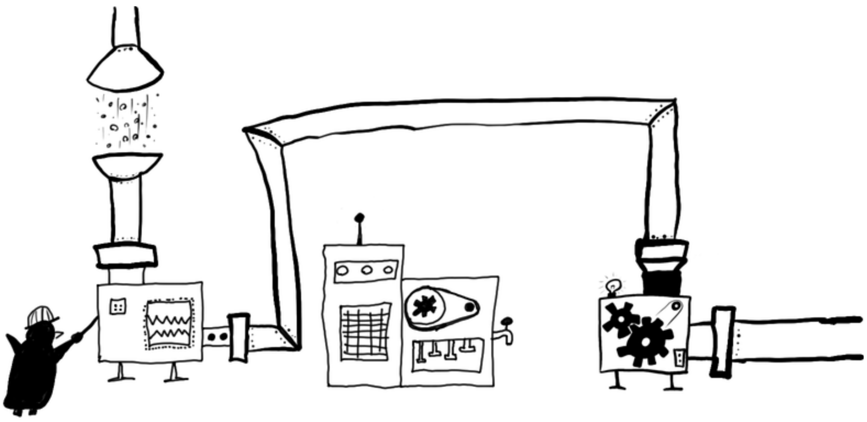
Have you ever had to save a file in one program, work on it in another, and import it into a third? Have you ever spent hours copy-and-pasting things between two programs?

I once had a temporary job that consisted of reading a name off of a sheet of paper, looking up a corresponding folder, finding a number in that folder to search for in a database program, and then using the mouse to copy and paste a different number in between two fields in the same program.

A lot of work done on computers can feel that way: Like slowly carrying buckets full of stuff in-between machines and programs.

Let's go back to our metaphorical workshop. What if all the machines had standard connectors on them for some sort of magic pipe, and the magic pipe gave you the option to move stuff between any two machines?

Standard I/O & Pipes



Standard Streams

Of course, real factories don't have magic pipes that can transport everything, and in the physical world there's no such thing as a universal input/output connector. It turns out, however, that the shell does have those things, or something pretty close to them:

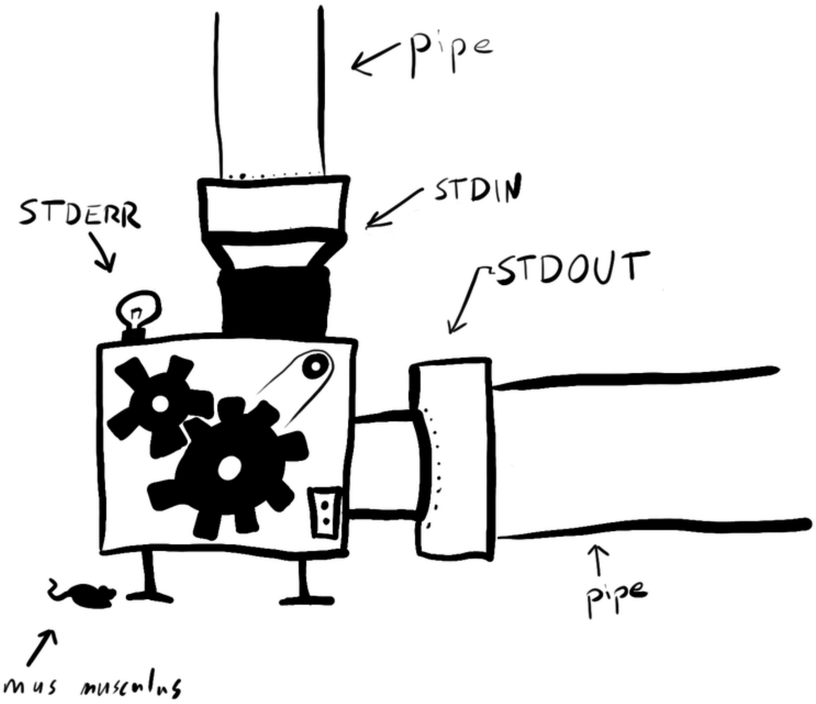
standard input or stdin	accepts stuff	file descriptor 0
standard output or stdout	spits stuff out	file descriptor 1

standard error or stderr	spits out error messages (usually to the terminal; more about that in a bit)	file descriptor 2
pipes	move stuff between streams	

Let's back up a bit. Imagine that a command like `head` or `tail` is a machine that takes input in one side and spits part of it out the other. In many of [the examples we've see so far](https://adafru.it/Cf5) (https://adafru.it/Cf5), you specify the input by giving the command an argument naming a file to use, like `tail /usr/share/dict/words`, and the output appears in your terminal. In other cases, a program runs interactively and accepts input directly from the terminal itself - that is, from what you type on your keyboard.

> The location of the dictionary may have changed, try `/usr/share/dict/wamerican` instead! or `ls /usr/share/dict` to see what's available

What you see in your terminal is like the stuff coming out one end of a machine, or like an open tap spilling water on the ground. You don't have to let it spill out directly into the terminal, though - the machine has a universal output connector of sorts, in the form of stdout, and you can connect it to the stdin of other machines with a pipe.



Building a Simple Pipeline

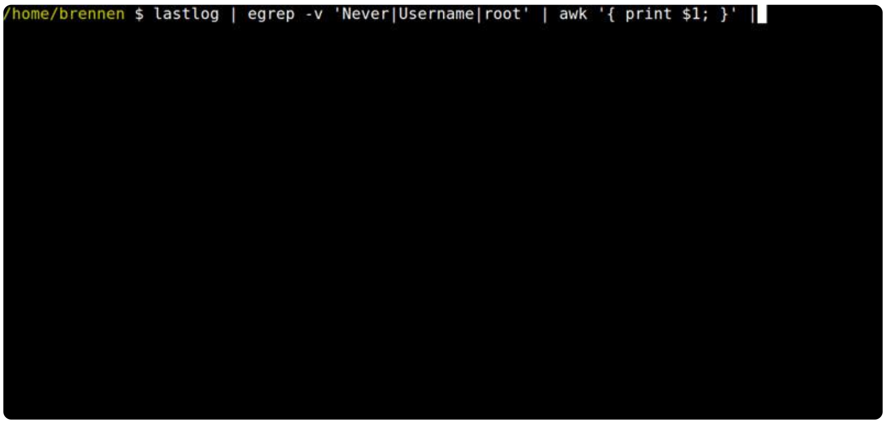
Have a look on your keyboard and see if you can find this little critter: `|`. It's often (though not always) located on the same key as the backslash (`\`). Let's try it on something simple.

Suppose you wanted to see the end of the word list, but starting from the last word and working backwards. The `sort` utility will be helpful here, especially since it optionally takes the `-r` argument to reverse its output.

This is called a pipeline, and it's one of the fundamental building blocks of the modern shell. Most traditional shell utilities will accept standard input and produce standard output.

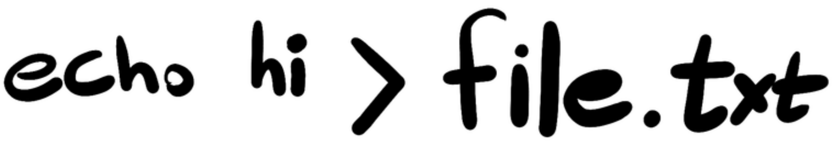
Pipelines are often simple affairs joining a few basic commands, but they can also be used to express complicated tasks combining many different operations. Here's a slightly more complicated pipeline to have an ASCII-art cow say the usernames of people who have logged into a server I run, starting with most recent login:

```
lastlog | egrep -v 'Never|Username|root' | awk '{ print $1; }' | xargs cowsay
```



Next, let's talk about what to do if you want to stash standard output somewhere.

Redirection To & From Files



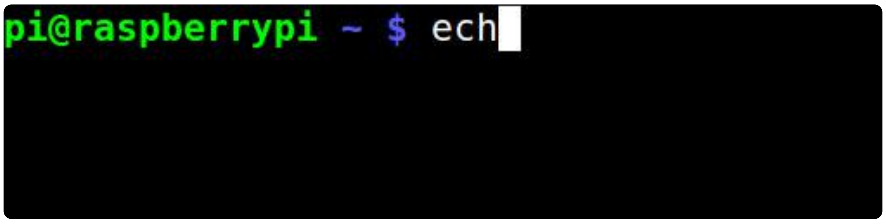
Now that you know how to use pipes in between commands, you can do a lot without worrying about storing data in files. But what if you want to keep something in a file? Let's say you want to:

- share a file with other people
- keep data for future reference
- build up a log file over time
- do further work on a file in a text editor like Nano or a spreadsheet application
- use data with some other tool which isn't very good at accepting standard input
- pull data out of a file and send it to the standard input of a command which doesn't know how to deal with files on its own

This is where `>`, `>>`, and their sibling `<` come into play.

Writing to a File

If you have output you want to store in `filename`, just tack `> filename` onto the end of your command line:

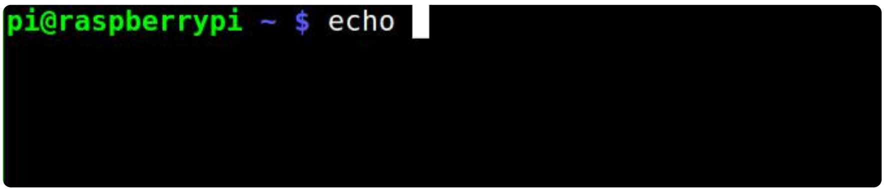


As long as you have permission to write to the specified filename, you'll wind up with a file containing the pipeline's output. I like to remember that the `>` character looks sort of like a little funnel.

> Careful! If you redirect output to an existing file, any existing contents will be overwritten. Use >> for tacking things on to the end of a file.

Appending to the End of a File

Sometimes, you want to stash your output at the end of an existing file. For that, just use `>> filename`:



This is often handy for accumulating log files over time, or for building up a collection of data with multiple pipelines.

Reading from a File

Lastly, suppose you want to take the contents of a file and send them to the standard input of some command? Just use `command < filename`.

The little funnel is going the other direction now, from the file back into the command.

This one can be harder to remember, because it doesn't come up nearly as often as the first two. Most utilities you'll encounter are written so that they can read files on their own. Still, it can be a powerful trick that saves you a bunch of contortions, especially once you get into writing your own scripts and utilities.

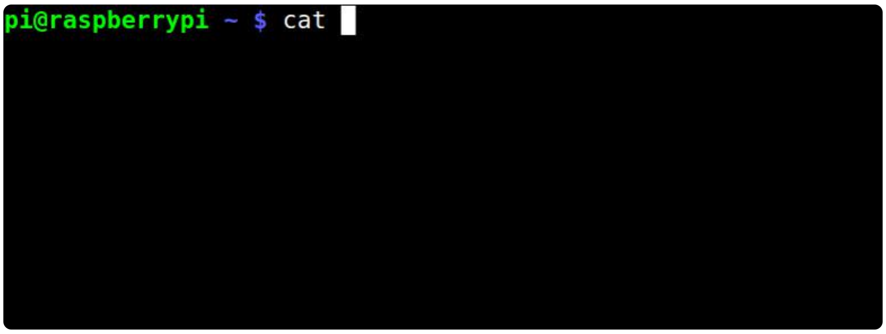
A Common Gotcha: Changing a File In-Place

Before long, you might find it natural to try and overwrite the contents of some file by running a command on it and redirecting the output of that command to the same file.

For example, `nl` is a command to number the lines of its input. Suppose I decided that the version of a file I've already saved would be better with line numbers on it? I might be tempted to try something like the following:

Ouch. The whole file seems to be gone. What's happening here? It turns out that the shell opens `file.txt` for writing before it ever executes the command pipeline! This makes a kind of sense once you know about it, but it's not especially obvious at first.

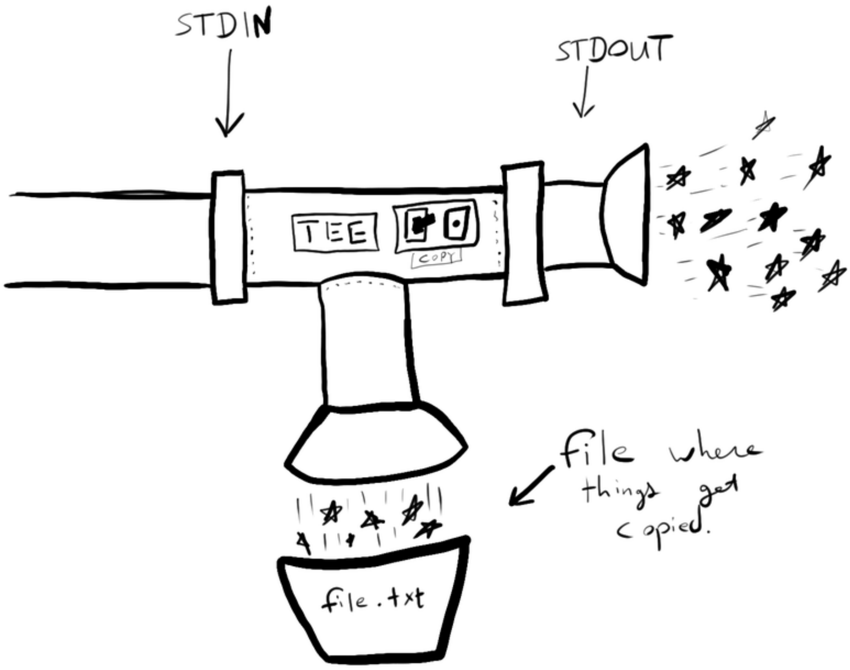
Fortunately, there's an easy workaround in this case - use `tee`:



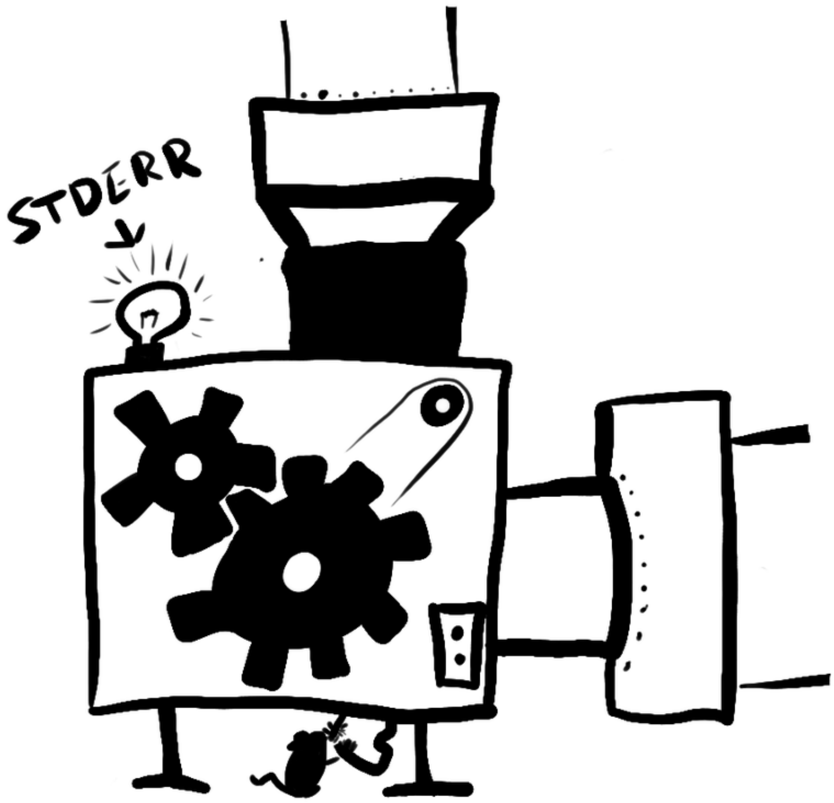
This does two things:

1. print its stdin back to stdout
2. write its stdin to a given filename

Since `tee file.txt` isn't executed until after `nl file.txt` has already run, you wind up with the desired outcome.



Standard Error & Exit Codes



The Discontents of Error Messaging

I mentioned stderr earlier, but I glossed over its actual function.

There are some problems with pipelines. One is that if you stream all the output of a command down a pipe to other commands, and part of that output is an error message, you may never discover that a command reported an error at all.

Another is that [lots of things that might look like an error message could be perfectly valid output](https://adafru.it/exP) (https://adafru.it/exP). Consider, for example, working with a log file containing the string "ERROR" on some line.

Unix systems have, for a long time, dealt with these problems by exposing a third standard stream. Programs print error messages and diagnostics to standard error, which (usually) shows up in your terminal even if you've redirected standard output elsewhere.

You can redirect stderr like you can stdout (and knowing this will come in handy sometimes), but the syntax is kind of fiddly and weird, and you shouldn't feel guilty if you have to look it up every time (or at least I don't):

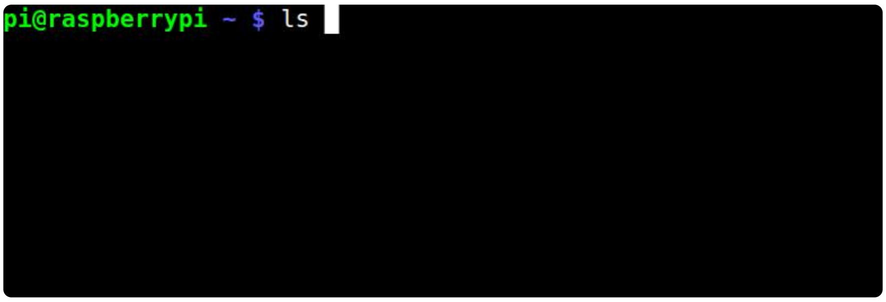
Redirect stderr (file descriptor 2) to a file	`command 2> filename`
Redirect standard stderr (file descriptor 2) to stdout (file descriptor 1) and pipe both to another commmand	`command 2>&1 | other_command`

For a concise and well-researched breakdown of the problem, check out Jesse Storimer's When to use STDERR instead of STDOUT (https://adafru.it/exQ).

Exit Status

As if multiple output streams didn't make for enough extra complexity, there's another way commands signal error conditions: Every command actually has a numeric exit status, ranging from `0` to `255`, where anything other than `0` signals a failure.

In Bash, you can check the most recent exit status by inspecting the special variable `$?`.

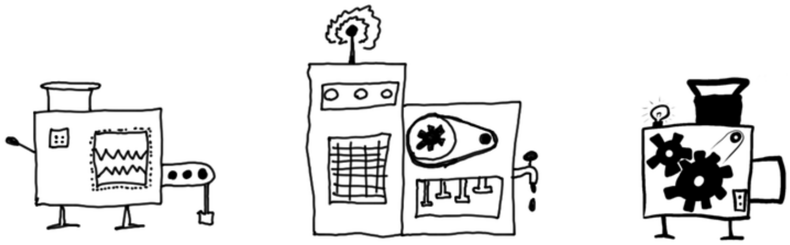


Exit statuses don't come up directly all that often in interactive use of the shell, but they become interesting once you start writing shell scripts. We won't get into scripting just now, but if you want to jump right in at the deep end, have a look at Mendel Cooper's Advanced Bash-Scripting Guide (https://adafru.it/exR).

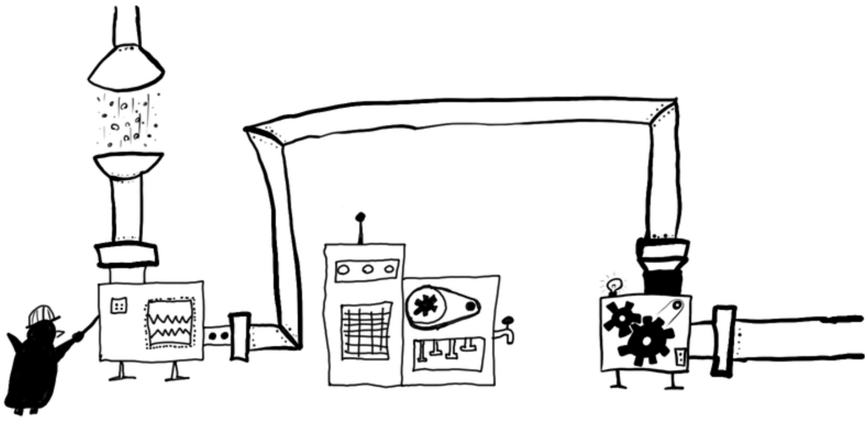
Concluding Remarks: Composability

The classic Unix utilities are, traditionally, designed to do one or two things well, and fit together with other utilities to solve problems. This leads to a property known as composability, which is another way of saying that the utilities are small pieces which can be put together and rearranged in many different ways.

The present-day GNU versions of the utilities (and lots of the other programs that have accumulated in the era of Linux) relax the only do one thing idea, out of pragmatism and simple feature creep (https://adafru.it/exY), but the basic idea holds.



Along with text files and the filesystem, standard streams, pipes, and redirection provide the necessary plumbing of a composable system. Pipelines can be written to explore and solve a huge range of problems by connecting simple operations together, and most tools can share data with other tools.



There's other shell magic to be learned, but the most important pieces are in place.

Next up, we look at aliases, wildcards, and the basics of treating the shell as a full-featured scripting language: An Illustrated Guide to Shell Magic: Typing Less & Doing More (https://adafru.it/sds).