**Replacing Bash Scripting with Python**

If I didn't cover something you want to know about or you find another problem, please open an issue on [github](https://github.com/ninjaaron/replacing-bash-scripting-with-python)!

Contents

* [Introduction](https://github.com/de8ug/replacing-bash-scripting-with-python#introduction)
  + [If the Shell is so great, what's the problem?](https://github.com/de8ug/replacing-bash-scripting-with-python#if-the-shell-is-so-great-what-s-the-problem)
  + [Why Python?](https://github.com/de8ug/replacing-bash-scripting-with-python#why-python)
  + [Learn Python](https://github.com/de8ug/replacing-bash-scripting-with-python#learn-python)
* [Reading and Writing Files](https://github.com/de8ug/replacing-bash-scripting-with-python#reading-and-writing-files)
* [Command-Line Interfaces in Python](https://github.com/de8ug/replacing-bash-scripting-with-python#command-line-interfaces-in-python)
  + [Working with stdin, stdout and stderr](https://github.com/de8ug/replacing-bash-scripting-with-python#working-with-stdin-stdout-and-stderr)
  + [CLI Arguments](https://github.com/de8ug/replacing-bash-scripting-with-python#cli-arguments)
  + [Environment Variables and Config files](https://github.com/de8ug/replacing-bash-scripting-with-python#environment-variables-and-config-files)
* [Filesystem Stuff](https://github.com/de8ug/replacing-bash-scripting-with-python#filesystem-stuff)
  + [Paths](https://github.com/de8ug/replacing-bash-scripting-with-python#paths)
  + [Replacing miscellaneous file operations: shutil](https://github.com/de8ug/replacing-bash-scripting-with-python#replacing-miscellaneous-file-operations-shutil)
* [Replacing sed, grep, awk, etc: Python regex](https://github.com/de8ug/replacing-bash-scripting-with-python#replacing-sed-grep-awk-etc-python-regex)
  + [How to grep](https://github.com/de8ug/replacing-bash-scripting-with-python#how-to-grep)
    - [Example: searching logs for errors](https://github.com/de8ug/replacing-bash-scripting-with-python#example-searching-logs-for-errors)
  + [How to sed](https://github.com/de8ug/replacing-bash-scripting-with-python#how-to-sed)
  + [How to awk](https://github.com/de8ug/replacing-bash-scripting-with-python#how-to-awk)
* [Running Processes](https://github.com/de8ug/replacing-bash-scripting-with-python#running-processes)
  + [Dealing with Exit Codes](https://github.com/de8ug/replacing-bash-scripting-with-python#dealing-with-exit-codes)
  + [Redirecting process IO (i.e. pipes)](https://github.com/de8ug/replacing-bash-scripting-with-python#redirecting-process-io-i-e-pipes)
    - [The stderr Parameter](https://github.com/de8ug/replacing-bash-scripting-with-python#the-stderr-parameter)
  + [Background Processes and Concurrency](https://github.com/de8ug/replacing-bash-scripting-with-python#background-processes-and-concurrency)
  + [shlex.quote: protecting against shell injection](https://github.com/de8ug/replacing-bash-scripting-with-python#shlex-quote-protecting-against-shell-injection)
* [Miscellaneous](https://github.com/de8ug/replacing-bash-scripting-with-python#miscellaneous)
  + [Getting the Time](https://github.com/de8ug/replacing-bash-scripting-with-python#getting-the-time)
  + [Interprocess Communication](https://github.com/de8ug/replacing-bash-scripting-with-python#interprocess-communication)
  + [Downloading Web Pages and Files](https://github.com/de8ug/replacing-bash-scripting-with-python#downloading-web-pages-and-files)

**[Introduction](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id4)**

The Unix shell is one of my favorite inventions ever. It's genius, plain and simple. The idea is that the user environment is a Turing-complete, declarative programming language. It has a dead-simple model for dealing with I/O and concurrency, which are notoriously difficult in most other languages.

For problems where the data can be expressed as a stream of similar objects separated by newlines to be processed concurrently through a series of filters and handles a lot of I/O, it's difficult to think of a more ideal language than the shell. A lot of the core parts on a Unix or Linux system is designed to express data in such formats.

**[If the Shell is so great, what's the problem?](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id5)**

The problem is if you want to do basically anything else, e.g. write logic, use control structures, handle complex data... You're going to have big problems. When Bash is coordinating external programs, it's fantastic. When it's doing any work whatsoever itself, it disintegrates into a pile of garbage.

For me, the fundamental problem with Bash and other shell dialects is that text is identifiers and identifiers are text -- and basically everything else is also text. In some sense, this makes the shell a homoiconic language, which theoretically means it might have an interesting metaprogramming story, until you realize that it basically just amounts to running eval on strings, which is a feature in basically any interpreted language today, and one that is frequently considered harmful. The problem with eval is that it's a pretty direct path to arbitrary code execution. This is great if arbitrary code execution is actually what you're trying to accomplish (like, say, in an HTML template engine), but it's not generally what you want.

Bash basically defaults to evaling everything. This is very handy for interactive use, since it cuts down in the need for a lot of explicit syntax when all you really want to do is, say, open a file in a text editor. This is pretty darn bad in a scripting context because it turns the entire language into an injection honeypot. Yes, it is possible and not so difficult to write safe Bash once you know the tricks, but it takes extra consideration and it is easy to forget or be lazy about it. Writing three or four lines of safe Bash is easy; two-hundred is quite a bit more challenging.

Bash has other problems. The syntax that isn't native to the Bourne Shell feels really ugly and bolted-on. For example, most modern shells have arrays. Let's look at the syntax for iterating on an array, but let's take the long way there.

$ foo='this and that' # variable assignment

$ echo $foo

this and that

$ # Oh dear. Text inside the variable was split into arguments on

$ # whitespace, because eval all the things.

$

$ # To avoid this insane behavior, do the obvious thing: use string

$ # interpolation. :-(

$ echo "$foo"

this and that

What does this have to do with iterating on arrays? Unfortunately, the answer is "something."

To properly iterate on the strings inside of an array (the only thing which an array can possibly contain), you also use variable interpolation syntax.

for item in "${my\_array[@]}"; do

stuff with "$item"

done

Why would string interpolation syntax ever be used to iterate over items in an array? I have some theories, but they are only that. I could tell you, but it wouldn't make this syntax any less awful. If you're not too familiar with Bash, you may also (rightly) wonder what this @ is, or why everything is in curly braces.

The answer to all these questions is more or less that they didn't want to do anything that would break compatibility with ancient Unix shell scripts, which didn't have these features. Everything just got shoe-horned in with the weirdest syntax you can imagine. Bash actually has a lot of features of modern programming languages, but the problem is that the syntax provided to access them is completely contrary to logic and dictated by legacy concerns.

The Bash IRC channel has a very helpful bot, greybot, written by one of the more important Bash community members and experts, greycat. This bot is written in Perl. I once asked why it wasn't written in Bash, and only got one answer: "greycat wanted to remain sane."

And really, that answer should be enough. Do you want to remain sane? Do you want people who maintain your code in the future not to curse your name? Don't use Bash. Do your part in the battle against mental illness.

**[Why Python?](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id6)**

No particular reason. [Perl](https://www.perl.org/) and [Ruby](http://rubyforadmins.com/) are also flexible, easy-to-write languages that have robust support for administrative scripting and automation. I would recommend against Perl for beginners because it has some similar issues to Bash: it was a much smaller language when it was created, and a lot of the syntax for the newer features has a bolted-on feeling [[1]](https://github.com/de8ug/replacing-bash-scripting-with-python#id2). However, if one knows Perl well and is comfortable with it, it's well suited to the task and is still a much saner choice for non-trivial automation scripts, and that is one of its strongest domains.

[Node.js](https://developer.atlassian.com/blog/2015/11/scripting-with-node/) is also starting to be used for administrative stuff these days, so that could also be an option, though JavaScript has similar issues to Perl. I've been investigating the possibility of using [Julia](https://docs.julialang.org/en/stable/) for this as well. Anyway, most interpreted languages seem to have pretty good support for this kind of thing, and you should just choose one that you like and is widely available on Linux and other \*nix operating systems.

The main reason I would recommend Python is if you already know it. If you don't know anything besides BASH (or BASH and lower-level languages like C or even Java), Python is a reasonable choice for your next language. It has a lot of mature, fast third-party libraries in a lot of domains -- science, math, web, machine learning, etc. It's also generally considered easy to learn and has become a major teaching language.

The other very compelling reason to learn Python is that it is the language covered in this very compelling tutorial.

|  |  |
| --- | --- |
| [[1]](https://github.com/de8ug/replacing-bash-scripting-with-python#id1) | I'm referring specifically to Perl 5 here. Perl 6 is a better language, in my opinion, but suffers from a lack of adoption.<https://perl6.org/> |

**[Learn Python](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id7)**

This tutorial isn't going to teach you the Python core language, though a few built-in features will be covered. If you need to learn it, I highly recommend the [official tutorial](https://docs.python.org/3/tutorial/index.html), at least through chapter 5. Through chapter 9 would be even better, and you might as well just read the whole thing at that point.

If you're new to programming, you might try the book [Introducing Python](http://shop.oreilly.com/product/0636920028659.do) or perhaps [Think Python](http://shop.oreilly.com/product/0636920045267.do). [Dive Into Python](http://www.diveintopython3.net/) is another popular book that is available for free online. You may see a lot of recommendations for [Learn Python the Hard Way](https://learncodethehardway.org/python/). I think this method is flawed, though I do appreciate that it was written by someone with strong opinions about correctness, which has some benefits.

This tutorial assumes Python 3.5 or higher, though it may sometimes use idioms from newer versions, and I will attempt to document when have used an idiom which doesn't work in 3.4, which is apparently the version that ships with the latest CentOS and SLES. Use at least 3.6 if you can. It has some cool new features, but the implementation of dictionaries (Python's hash map) was also overhauled in this version of Python, which sort of undergirds the way the whole object system is implemented and therefore is a major win all around.

Basically, always try to use whatever the latest version of Python is. Do not use Python 2. It will be officially retired in 2020. That's two years. If a library hasn't been ported to Python 3 yet, it's already dead, just that its maintainers might not know it yet.

One last note about this tutorial: It doesn't explain *so much*. I have no desire to rewrite things that are already in the official documentation. It frequently just points to the relevant documentation for those wishing to do the kinds of tasks that Bash scripting is commonly used for.

**[Reading and Writing Files](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id8)**

If you're going to do any kind of administration or automation on a Unix system, the idea of working with files is pretty central. The great coreutils like grep, sed, awk, tr, sort, etc., they are all designed to go over text files line by line and do... something with the content of that line. Any shell scripter knows that these "files" aren't always really files. Often as not, it's really dealing with the output of another process and not a file at all. Whatever the source, the organizing principle is streams of text divided by newline characters. In Python, this is what we'd call a "file-like object."

Because the idea of working with text streams is so central to Unix programming, we start this tutorial with the basics of working with text files and will go from there to other streams you might want to work with.

One handy thing in the shell is that you never really need file handles. All you have to type to loop over lines in a file would be something like:

while read line; do

stuff with "$line"

done < my\_file.txt

(Don't use this code. You actually have to do some things with $IFS to make it safe. Don't use any of my Bash examples. Don't use Bash! The proper one is while IFS= read -r line, but that just raises more questions.)

In Python, you need to turn a path into a file object. The above loop would be something like this:

with open('my\_file.txt') as my\_file:

for line in my\_file:

do\_stuff\_with(line.rstrip())

## the .rstrip() method is optional. It removes trailing whitespace

## from the line (including the newline character).

Let's take that apart.

The open() function returns a file object. If you just send it the path name as a string, it's going to assume it's a text file in the default system encoding (UTF-8, right?), and it is opened only for reading. You can, of course, do my\_file = open('my\_file.txt') as well. When you use with x as y: instead of assignment, it ensures the object is properly cleaned up when the block is exited using something called a "context manager". You can do my\_file.close() manually, but the withblock will ensure that happens even if you hit an error without having to write a lot of extra code.

The gross thing about context managers is that they add an extra level of indentation. Here's a helper function you can use to open a context manager for something you want to be cleaned up after you loop.

def iter\_with(obj):

with obj:

yield from obj

and then you use it like this:

for line in iter\_with(open('my\_file.txt')):

do\_stuff\_with(line)

yield from means it's a [generator function](https://docs.python.org/3/tutorial/classes.html#generators), and it's handing over control to a sub-iterator (the file object, in this case) until that iterator runs out of things to return. Don't worry if that doesn't make sense. It's a more advanced Python topic and not necessary for administrative scripting.

If you don't want to iterate on lines, which is the most memory-efficient way to deal with text files, you can slurp entire contents of a file at once like this:

with open('my\_file.txt') as my\_file:

file\_text = my\_file.read()

## or

lines = list(my\_file)

## or with newline characters removed

lines = my\_file.read().splitlines()

## This code wouldn't actually run because the file hasn't been

## rewound to the beginning after it's been read through.

## Also note: list(my\_file). Any function that takes an iterable can

## take a file object.

You can also open files for writing with, like this:

with open('my\_file.txt', 'w') as my\_file:

my\_file.write('some text\n')

my\_file.writelines(['a\n', 'b\n', 'c\n'])

print('another line', file=my\_file) # print adds a newline.

The second argument of open() is the *mode*. The default mode is 'r', which opens the file for reading text. 'w' deletes everything in the file (or creates it if it doesn't exist) and opens it for writing. You can also use the mode 'a'. This goes to the end of a file and adds text there. In shell terms, 'r' is a bit like <, 'w' is a bit like >, and 'a' is a bit like >>.

This is just the beginning of what you can do with files. If you want to know all their methods and modes, check the official tutorial's section on [reading and writing files](https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files). File objects provide a lot of cool interfaces. These interfaces will come back with other "file-like objects" which will come up many times later, including in the very next section.

**[Command-Line Interfaces in Python](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id9)**

**[Working with stdin, stdout and stderr](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id10)**

Unix scripting is all about filtering text streams. You have a stream that comes from lines in a file or output of a program and you pipe it through other programs. Unix has a bunch of special-purpose programs just for filtering text (some of the more popular of which are enumerated at the beginning of the previous chapter). Everyone using a \*nix system has probably done something like this at one point or another:

program-that-prints-something | grep 'a pattern'

This is the "normal" way to search through the output of a program for lines containing whatever it is you're searching for. Your setting the stdout of program-that-prints-something to the stdin of grep.

Great CLI scripts should follow the same pattern so you can incorporate them into your shell pipelines. You can, of course, write your script with its own "interactive" interface and read lines of user input one at a time:

username = input('What is your name? ')

This is fine in some cases, but it doesn't really promote the creation of reusable, multi-purpose filters. With that in mind, allow me to introduce the sys module.

The sys module has all kinds of great things as well as all kinds of things you shouldn't really be messing with. We're going to start with sys.stdin.

sys.stdin is a file-like object that, you guessed it, allows you to read from your script's stdin. In Bash you'd write:

while read line; do # <- not actually safe. Don't use bash.

stuff with "$line"

done

In Python, that looks like this:

import sys

for line in sys.stdin:

do\_stuff\_with(line) # <- we didn't remove the newline char this

# time. Just mentioning it because it's a

# difference between python and shell.

Naturally, you can also slurp stdin in one go, though this isn't the most Unix-y design choice and you could use up your RAM with a very large file:

text = sys.stdin.read()

As far as stdout is concerned, you can access it directly if you like, but you'll typically just use the print() function.

print("Hello, stdout.")

# ^ functionally same as:

sys.stdout.write('Hello, stdout.\n')

Anything you print can be piped to another process. Pipelines are great. For stderr, it's a similar story:

print('a logging message.', file=sys.stderr)

# or:

sys.stderr.write('a logging message.\n')

If you want more advanced logging functions, check out the [logging module](https://docs.python.org/3/howto/logging.html#logging-basic-tutorial).

Using stdin, stdout and stderr, you can write python programs which behave as filters and integrate well into a Unix workflow.

**[CLI Arguments](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id11)**

Arguments are passed to your program as a list which you can access using sys.argv. This is a bit like $@ in Bash, or $1 $2 $3... etc. e.g.:

for arg in "$@"; do

stuff with "$arg"

done

looks like this in Python:

import sys

for arg in sys.argv[1:]:

do\_stuff\_with(arg)

Why sys.argv[1:]? sys.argv[0] is like $0 in Bash or argv[0] in C. It's the name of the executable. Just a refresher (because you read the tutorial, right?) a\_list[1:] is list-slice syntax that returns a new list starting on the second item of a\_list, going through to the end.

If you want to build a more complete set of flags and arguments for a CLI program, the standard library module for that is [argparse](https://docs.python.org/3/howto/argparse.html). The tutorial in that link leaves out some useful info, so here are the [API docs](https://docs.python.org/3/library/argparse.html). [click](http://click.pocoo.org/5/) is a popular and powerful third-party module for building even more advanced CLI interfaces.

**[Environment Variables and Config files](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id12)**

Ok, environment variables and config files aren't necessarily only part of CLI interfaces, but they are part of the user interface in general, so I stuck them here. Environment variables are in the os.environ mapping, so you get to $HOME like this:

>>> import os

>>> os.environ['HOME']

'/home/ninjaaron'

As far as config files, in Bash, you frequently just do a bunch of variable assignments inside of a file and source it. You can also just write valid python files and import them as modules or eval them... but don't do that. Arbitrary code execution in a config file is generally not what you want.

The standard library includes [configparser](https://docs.python.org/3/library/configparser.html), which is a parser for .ini files, and also a [json](https://docs.python.org/3/library/json.html) parser. I don't really like the idea of human-edited json, but go ahead and shoot yourself in the foot if you want to. At least it's flexible.

[PyYAML](http://pyyaml.org/wiki/PyYAMLDocumentation), the YAML parser, and [TOML](https://github.com/uiri/toml) are third-party libraries that are useful for configuration files.

**[Filesystem Stuff](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id13)**

**[Paths](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id14)**

So far, we've only seen paths as strings being passed to the open() function. You can certainly use strings for your paths, and the os and os.path modules contain a lot of portable functions for manipulating paths as strings. However, since Python 3.4, we have [pathlib.Path](https://docs.python.org/3/library/pathlib.html#basic-use), a portable, abstract type for dealing with file paths, which will be the focus of path manipulation in this tutorial.

>>> from pathlib import Path

>>> # make a path of the current directory

>>> p = Path()

>>> p

PosixPath('.')

>>> # iterate over directory contents

>>> for i in p.iterdir():

... print(repr(i))

PosixPath('.git')

PosixPath('out.html')

PosixPath('README.rst')]

>>> # use filename globbing

>>> for i in p.glob('\*.rst'):

... print(repr(i))

PosixPath('README.rst')

>>> # get the full path

>>> p = p.absolute()

>>> p

PosixPath('/home/ninjaaron/doc/replacing-bash-scripting-with-python')

>>> # get the basename of the file

>>> p.name

'replacing-bash-scripting-with-python'

>>> # name of the parent directory

>>> p.parent

PosixPath('/home/ninjaaron/doc')

>>> # split path into its parts.

>>> p.parts

('/', 'home', 'ninjaaron', 'doc', 'replacing-bash-scripting-with-python')

>>> # do some tests about what the path is or isn't.

>>> p.is\_dir()

True

>>> p.is\_file()

False

>>> # more detailed file stats.

>>> p.stat()

os.stat\_result(st\_mode=16877, st\_ino=16124942, st\_dev=2051, st\_nlink=3, st\_uid=1000, st\_gid=100, st\_size=4096, st\_atime=1521557933, st\_mtime=1521557860, st\_ctime=1521557860)

>>> # create new child paths with slash.

>>> readme = p/'README.rst'

>>> readme

PosixPath('/home/ninjaaron/doc/replacing-bash-scripting-with-python/README.rst')

>>> # open files

>>> with readme.open() as file\_handle:

... pass

>>> # make file executable with mode bits

>>> readme.chmod(0o755)

>>> # ^ note that octal notation is must be explicite.

Again, check out the documentation for more info. [pathlib.Path](https://docs.python.org/3/library/pathlib.html#basic-use). Since pathlib came out, more and more builtin functions and functions in the standard library that take a path name as a string argument can also take a Path instance. If you find a function that doesn't, or you're on an older version of Python, you can always get a string for a path that is correct for your platform by using str(my\_path). If you need a file operation that isn't provided by the Path instance, check the docs for [os.path](https://docs.python.org/3/library/os.path.html) and [os](https://docs.python.org/3/library/os.html) and see if they can help you out. In fact, [os](https://docs.python.org/3/library/os.html) is always a good place to look if you're doing system-level stuff with permissions and UIDs and so forth.

If you're doing globbing with a Path instance, be aware that, like ZSH, \*\* may be used to glob recursively. It also (unlike the shell) will include hidden files (files whose names begin with a dot). Given this and the other kinds of attribute testing you can do on Path instances, it can do a lot of the kinds of stuff find can do.

>>> [p for p in Path().glob('\*\*/\*') if p.is\_dir()]

Oh. Almost forgot. p.stat(), as you can see, returns an [os.stat\_result](https://docs.python.org/3/library/os.html#os.stat_result) instance. One thing to be aware of is that the st\_mode, (i.e. permissions bits) is represented as an integer, so you might need to do something like oct(p.stat().st\_mode) to show what that number will look like in octal, which is how you set it with chmod in the shell.

**[Replacing miscellaneous file operations: shutil](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id15)**

There are certain file operations which are really easy in the shell, but less nice than you might think if you're using python file objects or the basic system calls in the os module. Sure, you can rename a file with os.rename(), but if you use mv in the shell, it will check if you're moving to a different file system, and if so, copy the data and delete the source -- and it can do that recursively without much fuss. [shutil](https://docs.python.org/3/library/shutil.html) is the standard library module that fills in the gaps. The docstring gives a good summary: "Utility functions for copying and archiving files and directory trees."

Here's the overview:

>>> import shutil

>>> # $ mv src dest

>>> shutil.move('src', 'dest')

>>> # $ cp src dest

>>> shutil.copy2('src', 'dest')

>>> # $ cp -r src dest

>>> shutil.copytree('src', 'dest')

>>> # $ rm a\_file

>>> os.remove('a\_file') # ok, that's not shutil

>>> # $ rm -r a\_dir

>>> shutil.rmtree('a\_dir')

>>> # $ tar caf 'my\_archive.tar.gz' 'my\_folder'

>>> shutil.make\_archive('my\_archive.tar.gz', 'gztar', 'my\_folder')

>>> # $ tar xaf 'my\_archive.tar.gz'

>>> shutil.unpack\_archive('my\_archive.tar.gz')

>>> # chown user:ninjaaron a\_file.txt

>>> shutil.chown('a\_file.txt', 'ninjaaron', 'user')

>>> # info about disk usage, a bit like `df`, but not exactly.

>>> shutil.disk\_usage('.')

usage(total=123008450560, used=86878904320, free=36129546240)

>>> # ^ sizes in bytes

>>> # which vi

>>> shutil.which('vi')

'/usr/bin/vi'

>>> # info about the terminal you're running in.

>>> shutil.get\_terminal\_size()

os.terminal\_size(columns=138, lines=30)

That's the thousand-foot view of the high-level functions you'll normally be using. The module documentation is pretty good for examples, but it also has a lot of details about the functions used to implement the higher-level stuff I've shown which may or may not be interesting.

I should probably also mention os.link and os.symlink at this point. They create hard and soft links respectively (like linkand link -s in the shell). Path instances also have .symlink\_to() method, if you want that.

**[Replacing sed, grep, awk, etc: Python regex](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id16)**

This section is not so much for experienced programmers who already know more or less how to use regexes for matching and string manipulation in other "normal" languages. Python is not so exceptional in this regard, though if you're used to JavaScript, Ruby, Perl, and others, you may be surprised to find that Python doesn't have regex literals. The regex functionally is all encapsulated in the [re](https://docs.python.org/3/library/re.html) module. (The official docs have a [regex HOWTO](https://docs.python.org/3/howto/regex.html), which is a good place to start if you don't know anything about regular expressions. If you have some experience, I'd recommend going straight for the [re](https://docs.python.org/3/library/re.html) API docs.)

This section is for people who know how to use programs like sed, grep and awk and wish to get similar results in Python, though short explanations will be provided of what those utilities are commonly used for.

I admit that writing simple text filters in Python will never be as elegant as it is in Perl, since Perl was more or less created to be like a super-powered version of the sh + awk + sed. The same thing can sort of be said about awk, the original text-filtering language on Unix. The main reason to use Python for these tasks is that the project is going to scale a lot more easily when you want to do something a bit more complex.

Another thing to keep in mind is that python has built-in operations that you can use if you just need to match a string, rather than a regular expression. Simple string operations are much faster than regular expressions, though not as powerful.

Note

One thing to be aware of is that Python's regex is more like PCRE (Perl-style -- also similar to Ruby, JavaScript, etc.) than BRE or ERE that most shell utilities support. If you mostly do sed or grep without the -E option, you may want to look at the rules for Python regex (BRE is the regex dialect you know). If you're used to writing regex for awk or egrep (ERE), Python regex is more or less a superset of what you know. You still may want to look at the documentation for some of the more advanced things you can do. If you know regex from either vi/Vim or Emacs, they both use their own dialect of regex, but they are supersets of BRE, and Python's regex will have some major differences.

**[How to grep](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id17)**

grep is the Unix utility that goes through each line of a file, tests if it contains a certain pattern, and then prints the lines that match. If you're a programmer and you don't use grep, start using it! Retrieving matching lines in a file is easy with Python, so we'll start there.

If you don't need pattern matching (i.e. something you could do with fgrep), you don't need regex to match a substring. You can simply use built-in syntax:

>>> 'substring' in 'string containing substring'

True

Otherwise, you need the regex module to match things:

>>> import re

>>> re.search(r'a pattern', r'string containing a pattern')

<\_sre.SRE\_Match object; span=(18, 27), match='a pattern'>

>>> re.search(r'a pattern', r'string without the pattern')

>>> # Returns None, which isn't printed in the Python REPL

I'm not going to go into the details of the "match object" that is returned at the moment. The main thing for now is that it evaluates to True in a boolean context. You may also notice I use raw strings r''. This is to keep Python's normal escape sequences from being interpreted, since regex uses its own escapes.

So, to use these to filter through strings:

>>> ics = an\_iterable\_containing\_strings

>>> # like fgrep

>>> filtered = (s for s in ics if substring in s)

>>> # like grep (or, more like egrep)

>>> filtered = (s for s in ics if re.search(pattern, s))

an\_iterable\_containing\_strings here could be a list, a generator or even a file/file-like object. Anything that will give you strings when you iterate on it. I use [generator expression](https://docs.python.org/3/tutorial/classes.html#generator-expressions) syntax here instead of a list comprehension because that means each result is produced as needed with lazy evaluation. This will save your RAM if you're working with a large file. You can invert the result, like grep -v simply by adding not to the if clause. There are also flags you can add to do things like ignoring the case (flags=re.I), etc. Check out the docs for more.

**[Example: searching logs for errors](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id18)**

Say you want to look through the log file of a certain service on your system for errors. With grep, you might do something like this:

$ grep -i error: /var/log/some\_service.log

This will search through /var/log/some\_service.log for any line containing the string error:, ignoring case. To do the same thing in Python:

with open('/var/log/some\_service.log') as log:

matches = (line for line in log if 'error:' in line.lower())

# line.lower() is a substitute for -i in grep, in this case

The difference here is that the bash version will print all the lines, and the python version is just holding on to them for further processing. If you want to print them, the next step is print(\*matches) or for line in matches: print(line, end=''). However, this is in the context of a script, so you probably want to extract further information from the line and do something programmatically with it anyway.

**[How to sed](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id19)**

sed can do a LOT of things. It's more or less "text editor" without a window. Instead of editing text manually, you give sedinstructions about changes to apply to lines, and it does it all in one shot. (The default is to print what the file would look like with modification. The file isn't actually changed unless you use a special flag.)

I'm not going to cover all of that. Back when I wrote more shell scripts and less Python, the vast majority of my uses for sedwere simply to use the substitution facilities to change instances of one pattern into something else, which is what I cover here.

>>> # sed 's/a string/another string/g' -- i.e. doesn't regex

>>> replaced = (s.replace('a string', 'another string') for s in ics)

>>> # sed 's/pattern/replacement/g' -- needs regex

>>> replaced = (re.sub(r'pattern', r'replacement', s) for s in ics)

[re.sub](https://docs.python.org/3/library/re.html#re.sub) has a lot of additional features, including the ability to use a *function instead of a string* for the replacement argument. I consider this to be very useful. If you're new to regex, note especially the section about backreferences in replacements. You may wish to check the section in the [regex HOWTO](https://docs.python.org/3/howto/regex.html) about [Search and Replace](https://docs.python.org/3/howto/regex.html#search-and-replace) as well.

**[How to awk](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id20)**

The sed section needed a little disclaimer. The awk section needs a bigger one. AWK is a Turing-complete text/table processing language. I'm not going to cover how to do everything AWK can do with Python idioms.

However, inside of shell scripts, it's most frequently used to extract fields from tabular data, such as tsv files. Basically, it's used to split strings.

>>> # awk '{print $1}'

>>> field1 = (f[0] for f in (s.split() for s in ics))

>>> # awk -F : '{print $1}'

>>> field1 = (f[0] for f in (s.split(':') for s in ics))

>>> # awk -F '[^a-zA-Z]' '{print $1}'

>>> field1 = (f[0] for f in (re.split(r'[^a-zA-Z]', s) for s in ics))

As is implied in this example, the [str.split](https://docs.python.org/3/library/stdtypes.html#str.split) method splits on sections of contiguous whitespace by default. Otherwise, it will split on whatever is given as a delimiter. For more on splitting with regular expressions, see [re.split](https://docs.python.org/3/library/re.html#re.split) and [Splitting Strings](https://docs.python.org/3/howto/regex.html#splitting-strings).

**[Running Processes](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id21)**

I come to this section at the end of the tutorial because one *generally should not be running a lot of processes inside of a Python script*. However, there are plenty of times when this "rule" should be broken. Say you want to do some automation with packages on your system; you'd be nuts not to use apt or yum (spelled dnf these days) or whatever your package manager is. Same applies if you're doing mkfs or using a very mature and featureful program like rsync. My general rule is that any kind of filtering utility should be avoided, but specialized programs for manipulating the system are fair game -- However, in some cases, there will be a 3rd-party Python library that provides a wrapper on the underlying C code. The library will, of course, be faster than spawning a new process in most cases. Use your best judgment. Be extra judicious if you're trying to write re-usable library code.

There are a number of functions which shall not be named in the [os](https://docs.python.org/3/library/os.html) module that can be used to spawn processes. They have a variety of problems. Some run processes in subshells (c.f. injection vulnerabilities). Some are thin wrappers on system calls in libc, which you may want to use if you implement your own processes library, but are not particularly fun to use. Some are simply older interfaces left in for legacy reasons, which have actually been re-implemented on top of the new module you're supposed to use, [subprocess](https://docs.python.org/3/library/subprocess.html). For administrative scripting, just use subprocess directly.

This tutorial focuses on using the [Popen](https://docs.python.org/3/library/subprocess.html#popen-constructor) constructor and the [run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function, the latter of which was only added in Python 3.5. If You are using Python 3.4 or earlier, you need to use the [old API](https://docs.python.org/3/library/subprocess.html#call-function-trio), though a lot of what is said here will still be relevant.

The [Popen](https://docs.python.org/3/library/subprocess.html#popen-constructor) API (over which the [run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function is a thin wrapper) is a very flexible, securely designed interface for running processes. Most importantly, it doesn't open a subshell by default. That's right, it's completely safe from shell injection vulnerabilities -- or, the injection vulnerabilities are opt-in. There's always the shell=True option if you're determined to write bad code.

On the other hand, it is a little cumbersome to work with, so there are a lot of third-party libraries to simplify it. [Plumbum](https://plumbum.readthedocs.io/en/latest/) is probably the most popular of these. [Sarge](http://sarge.readthedocs.io/en/latest/) is also not bad. My own contribution to the field is [easyproc](https://github.com/ninjaaron/easyproc) (though the documentation needs to be completely rewritten).

There are also a couple of Python supersets that allow inlining shell commands in python code. [xonsh](http://xon.sh/) is one, which also provides a fully functional interactive system shell experience and is the program that runs every time I open a terminal. I highly recommend it!

Anyway, on with the show.

>>> import subprocess as sp

>>> sp.run(['ls', '-lh'])

total 104K

-rw-r--r-- 1 ninjaaron users 69K Mar 21 16:40 out.html

-rw-r--r-- 1 ninjaaron users 32K Mar 23 11:11 README.rst

CompletedProcess(args=['ls', '-lh'], returncode=0)

As you see, the first and only required argument of the run function is a list (or any other iterable) of command arguments. stdout is not captured, it just goes wherever the stdout of the script goes. What is returned is a CompletedProcess instance, which has an args attribute and a returncode attribute. More attributes may also become available when certain keyword arguments are used with run.

**[Dealing with Exit Codes](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id22)**

Unlike most other things in Python, a process that fails doesn't raise an exception by default.

>>> sp.run(['ls', '-lh', 'foo bar baz'])

ls: cannot access 'foo bar baz': No such file or directory

CompletedProcess(args=['ls', '-lh', 'foo bar baz'], returncode=2)

This is the same way it works in the shell. However, you usually are going to want your script to stop if your command didn't work, or at least try something else. You could, do this manually:

>>> proc = sp.run(['ls', '-lh', 'foo bar baz'])

ls: cannot access 'foo bar baz': No such file or directory

>>> if proc.returncode != 0:

... # do something else

This would be most useful in cases where a non-zero exit code indicates something other than an error. For example, grepreturns 1 if no lines were matched. Not really an error, but something you might want to check for.

However, in the majority of cases, you probably want a non-zero exit code to crash the program, especially during development. This is where you need the check parameter:

>>> sp.run(['ls', '-lh', 'foo bar baz'], check=True)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/lib/python3.6/subprocess.py", line 418, in run

output=stdout, stderr=stderr)

subprocess.CalledProcessError: Command '['ls', '-lh', 'foo bar baz']' returned non-zero exit status 2.

Command '['ls', '-lh', 'foo bar baz']' returned non-zero exit status 2.

Much better! You can also use normal Python [exception handling](https://docs.python.org/3/tutorial/errors.html) now, if you like.

**[Redirecting process IO (i.e. pipes)](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id23)**

If you want to capture the output of a process, you need to use the stdout parameter. If you wanted to redirect it to a file, it's pretty straight-forward:

>>> with open('./foo', 'w') as foofile:

... sp.run(['ls'], stdout=foofile)

Pretty similar with input:

>>> with open('foo') as foofile:

... sp.run(['tr', 'a-z', 'A-Z'], stdin=foofile)

...

FOO

OUT.HTML

README.RST

If you want to do something with input and output text inside the script itself, you need to use the special constant, subprocess.PIPE.

>>> proc = sp.run(['ls'], stdout=sp.PIPE)

>>> print(proc.stdout)

b'foo\nout.html\nREADME.rst\n'

What's this now? Oh, right. Streams to and from processes default to bytes, not strings. You can decode your string, or you can use the flag to ensure the stream is a python string, which, in their infinite wisdom, the authors of the subprocess module chose to call universal\_newlines, as if that's the most important distinction between bytes and strings in Python. *Update: as of Python 3.7, `universal\_newlines` is aliased to `text`*

>>> proc = sp.run(['ls'], stdout=sp.PIPE, universal\_newlines=True)

>>> print(proc.stdout)

foo

out.html

README.rst

So that's awkward. In fact, this madness was one of my primary motivations for writing [easyproc](https://github.com/ninjaaron/easyproc).

If you want to sen a string to the stdin of a process, you will use a different run parameter, input (again, requires bytes unless universal\_newlines=True).

>>> sp.run(['tr', 'a-z', 'A-Z'], input='foo bar baz\n')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/lib/python3.6/subprocess.py", line 405, in run

stdout, stderr = process.communicate(input, timeout=timeout)

File "/usr/lib/python3.6/subprocess.py", line 828, in communicate

self.\_stdin\_write(input)

File "/usr/lib/python3.6/subprocess.py", line 781, in \_stdin\_write

self.stdin.write(input)

TypeError: a bytes-like object is required, not 'str'

a bytes-like object is required, not 'str'

>>>

>>> ## Makes nothing but sense...

>>>

>>>

>>> sp.run(['tr', 'a-z', 'A-Z'], input='foo bar baz\n', universal\_newlines=True)

FOO BAR BAZ

CompletedProcess(args=['tr', 'a-z', 'A-Z'], returncode=0)

**[The stderr Parameter](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id24)**

Just as there is an stdout parameter, there is also an stderr parameter for dealing with messages from the process. It works as expected:

>>> with open('foo.log', 'w') as logfile:

... sp.run(['ls', 'foo bar baz'], stderr=logfile)

...

>>> sp.run(['ls', 'foo bar baz'], stderr=sp.PIPE).stderr

b"ls: cannot access 'foo bar baz': No such file or directory\n"

However, another common thing to do with stderr in administrative scripts is to combine it with stdout using the oh-so-memorable incantation shell incantation of 2>&1. subprocess has a thing for that, too, the STDOUT constant.

>>> proc = sp.run(['ls', '.', 'foo bar baz'], stdout=sp.PIPE, stderr=sp.STDOUT)

>>> print(proc.stdout.decode())

ls: cannot access 'foo bar baz': No such file or directory

.:

foo

foo.log

out.html

README.rst

You can also redirect stdout and stderr to /dev/null with the constant subprocess.DEVNULL.

There's a lot more you can do with the [run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function, but that should be enough to be getting on with.

**[Background Processes and Concurrency](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id25)**

subprocess.run starts a process, waits for it to finish, and then returns a CompletedProcess instance that has information about what happened. This is probably what you want in most cases. However, if you want processes to run in the background or need to interact with them while they continue to run, you need the the [Popen](https://docs.python.org/3/library/subprocess.html#popen-constructor) constructor.

If you simply want to start a process in the background while you get on with your script, it's a lot like run.

>>> ## Time for popcorn...

>>> sp.Popen(['mpv', 'Star Trek II: The Wrath of Kahn.mkv'])

<subprocess.Popen object at 0x7fc35f4c0668>

>>> ## and the script continues while we enjoy the show...

This isn't quite the same as backgrounding a process in the shell using &. I haven't looked into what happens technically, but I can tell you that the process will keep going even if the terminal it was started from is closed. It's a bit like nohup. However, if not redirected, stdout and stderr will still be printed to that terminal.

Other reasons to do this might be to kick off a process at the beginning of the script that you need output from, and then come back to it later to minimize wait-time. For example, I use a Python script to generate my ZSH prompt. Among other things, this script checks the git status of the folder. However, that can take some time and I want the script to do as much work as possible while it's waiting on those commands.

## somewhere near the top of the script:

branch\_proc = sp.Popen(['git', 'branch'], stdout=sp.PIPE,

stderr=sp.DEVNULL, universal\_newlines=True)

status\_proc = sp.Popen(['git', 'status', '-s'], stdout=sp.PIPE,

stderr=sp.DEVNULL, universal\_newlines=True)

## ... somewhere further down:

branch = [i for i in branch\_proc.stdout if i.startswith('\*')][0][2:-1]

color = 'red' if status\_proc.stdout.read() else 'green'

Notice that stdout in this case is not a string. It's a file-like object. This is perfect for dealing with output from a program line-by-line, as many system utilities do. This is particularly important if the program produces a lot of lines of output and reading the whole thing into a Python string could potentially use up a lot of RAM. It's also useful for long-running programs that may produce output slowly, but you want to process it as it comes. e.g.:

>>> # don't actually use `find` in Python. Path.glob and os.walk

>>> # are better.

>>> with sp.Popen(['find', '/'], stdout=sp.PIPE,

... universal\_newlines=True) as proc:

... for line in proc.stdout:

... do\_stuff\_with(line)

You can also use this mechanism to pipe processes together, though the cases when you need to do this in python should be rare, since text filtering is best done in python itself. A case where you might want to pipe processes together could be extracting the content of an rpm package:

>>> # rpm2cpio a\_package.rpm | cpio -idm

>>> r2c = sp.Popen(['rpm2cpio', 'a\_package.rpm'], stdout=sp.PIPE)

>>> sp.run(['cpio', '-idm'], stdin=r2c.stdout)

**[shlex.quote: protecting against shell injection](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id26)**

The subprocess module, as mentioned earlier, is safe from injection by default, unless shell=True is used. However, there are some programs that will give arguments to a shell after they are started. SSH is a classic example. Every argument you send with ssh gets parsed by a shell on the remote system.

As soon as a process gets a shell, you're giving up one of the main benefits of using Python in the first place. You get back into the realm of injection vulnerabilities.

Basically, instead of this:

>>> sp.run(['ssh', 'user@host', 'ls', path])

You need to do something like this:

>>> import shlex

>>> sp.run(['ssh', 'user@host', 'ls', shlex.quote(path)])

[shlex.quote](https://docs.python.org/3/library/shlex.html#shlex.quote) will ensure that any spaces or shell metacharacters are properly escaped. The only trouble with it is that you actually have to remember to use it.

The shlex module also has a split function which will split a string into a list the same way the shell would split arguments. This is useful if you have a string that looks like a shell command and you want to send it to subprocess.run or subprocess.Popen.

**[Miscellaneous](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id27)**

This is where all the stuff goes that doesn't really need detailed coverage in this tutorial, but it's something you need to do often enough in shell scripts that it deserves pointers to additional resources.

**[Getting the Time](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id28)**

In administrative scripting, one frequently wants to put a timestamp in a file name for naming logs or whatever. In a shell script, you just use the output of date for this. Python has two libraries for dealing with time, and either is good enough to handle this. The [time](https://docs.python.org/3/library/time.html) module wraps time functions in libc. If you want to get a timestamp out of it, you do something like this:

>>> import time

>>> time.strftime('%Y.%m.%d')

'2018.08.18'

This can use any of the format spec you see when you run $ man date. There is also a time.strptime function which will take a string as input and use the same kind of format string to parse the time out of it and into a tuple.

The [datetime](https://docs.python.org/3/library/datetime.html) module provides classes for working with time at a high level. It's a little cumbersome for very simple things, and incredibly helpful for more sophisticated things like math involving time. The one handy thing it can do for our case is to give us a string of the current time without the need for a format specifier.

>>> import datetime

>>> # get the current time as a datetime object

>>> datetime.datetime.now()

datetime.datetime(2018, 8, 18, 10, 5, 56, 518515)

>>> now = \_

>>> str(now)

'2018-08-18 10:05:56.518515'

>>> now.strftime('%Y.%m.%d')

'2018.08.18'

This means that, if you're happy with the default string representation of the datetime class, you can just do str(datetime.datetime.now()) to get the current timestamp. There is also a datetime.datetime.strptime() to generate a datetime instance from a timestamp.

**[Interprocess Communication](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id29)**

I'm not sure if IPC is really part of bash scripting, but sometimes administrators might need to write a daemon or whatever that runs in the background, but is still able to receive communication from the user via a client.

The simplest way to do this is with a fifo, a.k.a. a named pipe.

import os

myfifo = '/tmp/myfifo'

os.mkfifo(myfifo)

try:

while True:

with open(myfifo) as fh:

do\_something(fh.read())

except:

os.remove(myfifo)

raise

That's your server that you start with your init system. The simplest client could just be echo; echo some text > /tmp/myfifo. Of course, you can do a lot more with the client if you like. The limitation of a fifo is that it's one-way communication. If you want two-way, you need two fifos. Alternatively, use a TCP socket.

Python has a dead-simple library for making a socket server, aptly named [socketserver](https://docs.python.org/3/library/socketserver.html). Scroll down to the examples and they have basically everything you need to know for implementing your server and client. For a daemon that you're just interacting with over localhost, you're going to get better performance using the UnixStreamServer class, and you won't use up a port. Plus, Unix sockets will make your Unix beard grow better.

The problem with either of these is that they just block until they get a message (unless you use the threaded socket server, which might be fine in some cases). If you want your daemon to do work while simultaneously listening for input, you need threads or asyncio. Unfortunately for you, this tutorial is about replacing Bash with Python, and I'm not about to try to teach you concurrency.

Note

I'll just say that the python threading module is fine for IO-bound multitasking on a small scale. If you need something large-scale, use asyncio. If you need real concurrent execution, know that Python threads are a lie, and asyncio doesn't do that. You need multiprocessing. If you need concurrent execution, but processes are too expensive, use another programming language. Python has limitations in this area.

**[Downloading Web Pages and Files](https://github.com/de8ug/replacing-bash-scripting-with-python" \l "id30)**

If you're doing any kind of fancy http requests that require things like interacting with APIs, shooting data around, doing authentication, or basically anything besides downloading static assets, use [requests](http://docs.python-requests.org/en/master/). In fact, you should probably even use it for the simple case of downloading things. However, this is also possible with the standard library, and not particularly painful.

For that, you need [urllib.request](https://docs.python.org/3/library/urllib.request.html).