DevOps in Python

Infrastructure as Python

—

Moshe Zadka

DevOps in Python

Infrastructure as Python

—

Moshe Zadka

Introduction

Python was started as a language to automate an operating system: the Amoeba. Since it had an API, not just textual files representations, a typical UNIX shell would be ill suited. The Amoeba OS is now a relic. However, Python continues to be a useful tool for automation of operations – the heart of typical DevOps work. It is easy to learn and easy to write readable code in – a necessity, when a critical part of the work is responding to a 4 a.m. alert, and modifying some misbehaving program. It has powerful bindings to C and C++, the universal languages of operating systems – and yet is natively memory safe, leading to few crashes at the automation layer.

Finally, although not true when it was originally created, Python is one of the most popular languages. This means that it is relatively easy to hire people with Python experience, and easy to get training materials and courses for people who need to learr on the job.

This book will guide you through how to take advantage of Python to automate operations.

What to Expect in the Book

There are many sources that teach Python, the language: books, tutorials, and even free videos online. Basic familiarity with the language will be assumed here. However, for the typical SRE/DevOps person, there are a lot of aspects of Python that few sources cover, except for primary documentation and various blogs. We cover those early on in the book.

The first step in using Python is not writing a “hello world” program. The first step is installing it. There, already, we are faced with various choices, with various trade-offs between them. We will cover using the preinstalled version of Python, using ready-made, third-party prebuilt packages, installing Python from sources, and other alternatives. One of Python’s primary strengths, which any program slightly longer than “hello world” will take advantage of, is its rich third-party library ecosystem. We will cover

CHAPTER 1

Installing Python

Before we start using Python, we need to install it. Some operating systems, like Mac OS X and some Linux variants, have Python preinstalled. Those versions of Python, colloquially called "system Python," often make poor defaults for people who want to develop in Python.

For one thing, the version of Python installed is often behind latest practices. For another, system integrators will often patch Python in ways that can lead to surprises later. For example, Debian-based Python is often missing modules like venv and ensurepip. Mac OS X Python links against a Mac shim around its native SSL library. What those things means, especially when starting out and consuming FAQs and web resources, is that it is better to install Python from scratch.

We cover a few ways to do so and the pros and cons of each

1.1 OS Packages

For some of the more popular operating systems, volunteers have built ready-to-install packages.

The most famous of these is the "deadsnakes" PPA (Personal Package Archives).

The "dead" in the name refers to the fact that those packages are already built – with the metaphor that sources are "alive." Those packages are built for Ubuntu and will usually support all the versions of Ubuntu that are still supported upstream. Getting those packages is done simply:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
```


M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3

where one can inspect the shell script before running, or even use git checkout to pin to a specific revision.

Sadly, pyenv does not work on Windows.

After installing pyenv, it is useful to integrate it with the running shell. We do this by adding to the shell initialization file (for example, .bash_profile):

```
export PATH="~/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

This will allow pyenv to properly intercept all needed commands.

Pyenv separates the notion of *installed* interpreters from *available* interpreters. In order to install a version,

```
$ pyenv install <version>
```

For CPython, <version> is just the version number, such as 3.6.6 or 3.7.0rc1.

An *installed* version is distinct from an available version. Versions can be available either “globally” (for a user) by using

```
$ pyenv global 3.7.0
```

or locally by using

```
$ pyenv local 3.7.0
```

Local means they will be available in a given directory. This is done by putting a file python-version.txt in this directory. This is important for version-controlled repositories, but there are a few different strategies to manage those. One is to add this file to the “ignored” list. This is useful for heterogenous teams of open source projects. Another is to check this file in, so that the same version of Python is used in this repository.

Note that *pyenv*, since it is designed to install versions of Python side by side, has no concept of “upgrading” Python. In order to use a newer Python, it needs to be installed with *pyenv* and then set as the default.

1.4 PyPy

The “usual” implementation of Python is sometimes known as “CPython,” to distinguish it from the language proper. The most popular alternative implementation is PyPy. PyPy is a Python-based JIT implementation of Python in Python. Because it has a dynamic JIT (Just in Time) compilation to assembly, it can sometimes achieve phenomenal speed-ups (3x or even 10x) over regular Python.

There are sometimes challenges in using PyPy. Many tools and packages are tested only with CPython. However, sometimes spending the effort to check if PyPy is compatible with the environment is worth it if performance matters.

There are a few subtleties in installing Python from source. While it is theoretically possible to “translate” using CPython, in practice the optimizations *in* PyPy mean that translating using PyPy works on more reasonable machines. Even when installing from source, it is better to first install a binary version to bootstrap.

The bootstrapping version should be PyPy, not PyPy3. PyPy is written in the Python 2 dialect. It is one of the only cases where worrying about the deprecation is not relevant, since PyPy is a Python 2 dialect interpreter. PyPy3 is the Python 3 dialect implementation, which is usually better to use in production as most packages are slowly dropping support for Python 2.

The latest PyPy3 supports 3.5 features of Python, as well as f-strings. However, th latest async features, added in Python 3.6, do not work.

1.5 Anaconda

The closest to a “system Python” that is still reasonable for use as a development platform is the Anaconda Python. Anaconda is a so-called “meta-distribution.” It is, in essence, an operating system on top of the operating system. Anaconda has its grounding in the scientific computing community, and so its Python comes with easy-to-install modules for many scientific applications. Many of these modules are nontrivial to install from PyPI, requiring a complicated build environment.

It is possible to install multiple Anaconda environments on the same machine. This is handy when needing different Python versions or different versions of PyPI modules.

In order to bootstrap Anaconda, we can use the bash installer, available from https://conda.io/miniconda.html . The installer will also modify ~/.bash_profile to add the path to conda, the installer.

CHAPTER 2

Packaging

Much of dealing with Python in the real world is dealing with third-party packages. For a long time, the situation was not good. Things have improved dramatically, however. It is important to understand which "best practices" are antiquated rituals, which ones are based on faulty assumptions but have some merit, and which are actually good ideas

When dealing with packaging, there are two ways to interact. One is to be a "consumer," wanting to use the functionality from a package. Another is to be the "producer," publishing a package. These describe, usually, different development tasks not different people.

It is important to have a solid understanding of the "consumer" side of packages before moving to "producing." If the goal of a package publisher is to be useful to the package user, it is crucial to imagine the "last mile" before starting to write a single line of code.

2.1 Pip

The basic packaging tool for Python is pip. By default, installations of Python do not come with pip. This allows pip to move faster than core Python – and work with alternative Python implementations, like PyPy. However, they do come with the useful ensurepip module. This allows getting pip via python -m ensurepip. This is usually the easiest way to bootstrap pip.

Some Python installations, especially system ones, disable ensurepip. When lacking ensurepip, there is a way of manually getting it: get-pip.py. This is a downloadable single file that, when executed, will unpack pip.

Luckily, pip is the only package that needs these weird gyrations to install. All other packages can, and should, be installed using pip. This includes upgrading pip itself, which can be done with pip install --upgrade pip.

M. Zadka, *DevOps in Python* , https://doi.org/10.1007/978-1-4842-4433-3

2.2 Virtual Environments

Virtual environments are often misunderstood, because the concept of "environments" is not clear. A Python environment refers to the root of the Python installation. The reason it is important is because of the subdirectory lib/site-packages. The lib/site- packages directory is where third-party packages are installed. In modern times, they are often installed by pip. While there used to be other tools to do it, even bootstrapping pip and virtualenv can be done with pip, let alone day-to-day package management.

The only common alternative to pip is system packages, where a system Python is concerned. In the case of an Anaconda environment, some packages might be installed as part of Anaconda. In fact, this is one of the big benefits of Anaconda: many Python packages are custom built, especially those which are nontrivial to build.

A "real" environment is one that is based on the Python installation. This means that to get a new real environment, we must reinstall (and often rebuild) Python. This is sometimes an expensive proposition. For example, tox will rebuild an environment from scratch if any parameters are different. For this reason, *virtual* environments exist.

A virtual environment copies the minimum necessary out of the real environment to mislead Python into thinking that it has a new root. The exact details are not important, but what is important is that this is a simple command that just copies files around (and sometimes uses symbolic links).

There are two ways to use virtual environments: activated and unactivated. In order to use an unactivated virtual environment, which is most common in scripts and automated procedures, we explicitly call Python from the virtual environment.

This means that if we created a virtual environment in /home/name/venvs/my-special- env, we can call /home/name/venvs/my-special-env/bin/python to w inside this environment. For example, /home/name/venvs/my-special-env/bi -m pip will run pip but install in the virtual environment. Note that for entry-point-based scripts, they will be installed alongside Python, so we can run /home/name/venvs/my-special-env/bin/pip to install packages in the virtual environment.

The other way to use a virtual environment is to "activate" it. Activating a virtual environment in a bash-like shell means *sourcing* its activate script:

```
$ source /home/name/venvs/my-special-env/bin/activate
```

Whichever command is used to create the virtual environment, it will create the directory for the environment. It is best if this directory does not exist before that. A best practice is to remove it before creating the environment. There are also options about how to create the environment: which interpreter to use and what initial packages to install. For example, sometimes it is beneficial to skip pip installation entirely. We can then bootstrap pip in the virtual environment by using get-pip.py. This is a way to avoid a bad version of pip installed in the real environment – since if it is bad enough, it cannot even be used to upgrade pip.

2.3 Setup and Wheels

The term "third party" (as in "third-party packages") refers to a someone other than the Python core developers ("first party") or the local developers ("second party"). We have covered how to install "first-party" packages in the installation section. We used pip and virtualenv to install "third-party" packages. It is time to finally turn our attention to the missing link: local development and installing local packages, or "second-party" packages.

This is an area seeing a lot of new additions, like pyproject.toml and flit. However, it is important to understand the classic way of doing things. For one, it takes a while for new best practices to settle in. For another, existing practices are based on setup. py, and so this way will continue to be the main way for a while – possibly even for the foreseeable future.

The setup.py file describes, in code, our "distribution." Note that "distribution" is distinct from "package." A package is a directory with (usually) __init__.py that Python can import. A distribution can contain several packages or even none! However, it is a good idea to keep a 1-1-1 relationship: one distribution, one package, named the same.

Usually, setup.py will begin by importing setuptools or distutils. While distutils is built-in, setuptools is not. However, it is almost always installed first in a virtual environment, due to its sheer popularity. Distutils is not recommended: it has not been updated for a long time. Notice that setup.py cannot meaningfully, explicitly declare it needs setuptools nor explicitly request a specific version: by the time it is read, it will have already tried to import setuptools. This non-declarativeness is part of the motivation for packaging alternatives.

tedium. bumpversion is a useful tool, especially when choosing semantic versioning. Finally, versioneer supports easy integration with the git version control system, so that a tag is all that needs to be done for release.

Another popular field in setup.py, which is not marked "required" in the documentation but is present on almost every package, is install_requires. This is how we mark other distributions that our code uses. It is a good practice to put "loose" dependencies in setup.py. This is in contrast to exact dependencies, which specify a specific version. A loose dependency looks like Twisted>=17.5 – specifying a minimum version but no maximum. Exact dependencies, like Twisted==18.1, are usually a bad idea in setup.py. They should only be used in extreme cases: for example, when using significant chunks of a package's *private* API.

Finally, it is a good idea to give find_packages a whitelist of what to include, in order to avoid spurious files. For example,

```
setuptools.find_packages(include=["my_package*
```

Once we have `setup.py` and some Python code, we want to make it into a distribution. There are several formats a distribution can take, but the one we will cover here is the *wheel*. If my-directory is the one that has setup.py, running pip wheel my-directory, will produce a wheel, as well as the wheels of all of its recursive dependencies.

The default is to put the wheels in the current directory, which is seldom the desired behavior. Using --wheel-dir<output-directory> will put the wheel in the directory – as well as the wheels of any distribution it depends on.

There are several things we can do with the wheel, but it is important to note that one thing we can do is pip install <wheel file>. If we add pip install <wheel file> --wheel-dir <output directory>, then pip will use the wheels in the directory and will not go out to PyPI. This is useful for reproducible installs, or support for air-gapped modes.

2.4 Tox

Tox is a tool to automatically manage virtual environments, usually for tests and builds. It is used to make sure that those run in well-defined environments and is smart about caching them in order to reduce churn. True to its roots as a test-running tool, Tox is configured in terms of *test environments*.

The deps subsection details which packages should be installed in the test environment's virtual environment. Here we chose to specify flake8 with a loose dependency. Another option is to specify a strict dependency (e.g., flake8==1.0.0.). This helps with reproducible test runs. We could also specify -r <requirements file> and manage the requirements separately. This is useful if we have other tooling that takes the requirements file.

```
commands =
    flake8 useful
```

In this case, the only command is to run flake8 on the directory useful. By default, a Tox test run will succeed if all commands return a successful status code. As something designed to run from command lines, flake8 respects this convention and will only exit with a successful status code if there were no problems detected with the code.

```
[testenv]
```

The other two environments, lacking specific configuration, will fall back to the generic environment. Note that because of their names, they will use two different interpreters: CPython 3.6 and PyPy with Python 3.5 compatibility.

```
deps =
    pytest
```

In this environment, we install the pytest runner. Note that in this way, our tox.ini documents the assumptions on the tools needed to run the tests. For example, if our tests used Hypothesis or PyHamcrest, this is where we would document it.

```
command
    pytest useful
```

Again, the command run is simple. Note, again, that pytest respects the convention and will only exit successfully if there were no test failures.

As a more realistic example, we turn to the tox.ini of folony:

```
[tox]
envlist = {py36,py27,pypy}-{unit,func},py27-lint,py27-wheel,docs
toxworkdir = {toxinidir}/build/.tox
```

```
{py36,py27,pypy}-unit: coverage report --include ncolony* \
                                --omit */tests/*,*/interfaces*,*/_version*
                             --show-missing --fail-under=10C
py27-lint: pylint --rcfile admin/pylintrc ncolony
py27-lint: python -m ncolony tests.nitpicker
py27-lint: flake8 ncolony
{py36,py27,pypy}-func: python -Werror -W ignore::DeprecationWarnin
                                -W ignore::ImportWarning \
                                -m ncolony tests.functional_test
```

Configuring "one big test environment" means we need to have all our commands mixed in one bag and select based on patterns. This is also a more realistic test run command – we want to run with warnings enabled, but disable warnings we do not worry about, and also enable code coverage testing. While the exact complications will vary, we almost always need enough things so that the commands will grow to a decent size.

```
[testenv:py27-wheel]
skip_install = True
deps =
        coverage
        Twisted
        wheel
        gather
command
        mkdir -p {envtmpdir}/dist
        pip wheel . --no-deps --wheel-dir {envtmpdir}/dist
        sh -c "pip install --no-index {envtmpdir}/dist/*.whl"
        coverage run {envbindir}/trial \
                --temp-directory build/_trial_temp {posargs:ncolony}
        coverage report --include */site-packages/ncolony*
                        --omit */tests/*,*/interfaces*,*/_version*
                        --show-missing --fail-under=100
```

The py27-wheel test run ensures we can build, and test, a wheel. As a side effect, this means a complete test run will build a wheel. This allows us to upload a *tested* wheel to PyPI when it is release time.

This is an example of using an *unactivated* virtual environment.

The best way to use poetry is to create a dedicated virtual environment for the project. We will build a small demo project. We will call it "useful."

```
$ mkdir useful $ cd useful $ python3 -m venv
build/useful $ source build/useful/bin/activate (useful)$
poetry init (useful)$ poetry add termcolor (useful)$
mkdir useful (useful)$ touch useful/__init__.py (useful)$
cat > useful/__main__.py import termcolor
print(termcolor.colored("Hello", "red"))
```

If we have done all this, running *python -m useful* in the virtual environment will print a red Hello. After we have interactively tried various colors, and maybe decided to make the text bold, we are ready to release:

```
(useful)$ poetry build
(useful)$ ls dist/
useful-0.1.0-py2.py3-none-any.whl useful-0.1.0.tar.gz
```

2.5.2 Pipenv

Pipenv is a tool to create virtual environments that match a specification, in addition to ways to evolve the specification. It relies on two files: Pipfile and Pipfile.lock. We can install pipenv similarly to how we installed poetry – in a custom virtual environment and add an alias.

In order to start using it, we want to make sure no virtual environments are activated. Then,

```
$ mkdir useful
$ cd useful
$ pipenv add termcolo
```

There is also a web server that allows us to search in the local package directory. Since a lot of use cases do not even involve searching on the PyPI website, this is definitely optional. Finally, there is a client command-line tool that allows configuring various parameters on the running instance. The client is most useful in more esoteric use cases.

Installing and running DevPI is straightforward. In a virtual environment, simply run:

```
(devpi)$ pip install devpi-server
(devpi)$ devpi-server --start --init
```

The pip tool, by default, goes to pypi.org . For some basic testing of DevPI, we can create a new virtual environment, playground, and run:

```
(playground)$ pip install \
                -i http://localhost:3141/root/pypi/+simple/ \
                httpie glom
(playground)$ http --body https://httpbin.org/get | glom '{"url":"url"}'
{
  "url": "https://httpbin.org/get"
}
```

Having to specify the -i ... argument to pip every time would be annoying. After checking that everything worked correctly, we can put the configuration in an environment variable:

```
$ export PIP_INDEX_URL=http://localhost:3141/root/pypi/+simple/
```

Or to make things more permanent:

```
$ mkdir -p ~/.pip && cat > ~/.pip/pip.conf << EOF
[global]
index-url = http://localhost:3141/root/pypi/+simple/

[search]
index = http://localhost:3141/root/pypi/
```

The above file location works for UNIX operating systems. On Mac OS X the configuration file is $HOME/Library/Application Support/pip/pip.conf. On Wi the configuration file is %APPDATA%\pip\pip.ini.

Note that this allows us to upload to an index that we only use explicitly, so we are not shadowing my-package for all environments that are not using this explicitly.

An even more advanced use-case, we can do this:

```
(devpi)$ devpi index root/pypi mirror_url=https://ourdevpi.local
```

This will make our DevPI server a mirror of a local, "upstream," DevPI server. This allows us to upload private packages to the "central" DevPI server, in order to share with our team. In those cases, the upstream DevPI server will often need to be run behind a proxy – and we need to have some tools to properly manage user access.

Running a "centralized" DevPI behind a simple proxy that asks for username and password allows an effective private repository. For that, we would first want to remove the root/pypi index:

```
$ devpi index --delete root/pypi
```

and then re-create it with

```
$ devpi index --create root/pypi
```

This means the root index no longer will mirror pypi. We can upload packages now directly to it. This type of server is often used with the argument `--extra-index-url` to pip, to allow pip to retrieve both from the private repository and the external one. However, sometimes it is useful to have a DevPI instance that only serves specific packages. This allows enforcing rules about auditing before using any packages. Whenever a new package is needed, it is downloaded, audited, and then added to the private repository.

2.7 Pex and Shiv

While it is currently nontrivial to compile a Python program into one self-contained executable, we can do something that is *almost* as good. We can compile a Python program into a single file that only needs an installed interpreter to run. This takes advantage of the particular way Python handles startup.

When running python /path/to/filename, Python does two things:

- Adds the directory /path/to to the module path.
- Executes the code in /path/to/filename.

Pex has a few ways to find the entry point. The two most popular ones are-m some_package, which will behave as though python -m some_pack; or -c console-script, which will find what script would have been installed as console-script, and invoke the relevant entry point.

It is also possible to use Pex as a library.

```
from pex import pex_builder
```

Most of the logic to build Pex files is in thepex_buildermodule.

```
builder = pex_builder.PEXBuilder()
```

We create a builder object.

```
builder.set_entry_point('some_package'
```

We set the entry point. This is equivalent to the-m some_packargument on the command line.

```
builder.set_shebang(sys.executable)
```

The Pex binary has a sophisticated argument to determine the right shebang line. This is sometimes specific to the expected deployment environment, so it is a good idea to put some thought into the right shebang line. One option is/usr/bin/env python, which will find what the current shell calls pythor It is sometimes a good idea to specify a version here, such as/usr/local/bin/python3.6 , for example.

```
subprocess.check_call([sys.executable, '-m', 'pip', 'wheel',
                       '--wheel-dir', 'my-wheels',
                       '--requirements', 'requirements.txt'])
```

Once again, we create wheels withpip. As tempting as it is,pip is not usable as a library, so shelling out is the only supported interface.

```
for dist in os.listdir('my-wheels'):
    dist = os.path.join('my-wheels', dist)
    builder.add_dist_location(dist)
```

We add all packages that pip built.

```
builder.build('my-file.pex')
```

Finally, we have the builder produce a Pex file.

In some cases, the --console-scripts argument is not necessary. If, as in the example above, there is only one console script entry point, then it is implied. Otherwise, if there is a console script with the same name as the package, then that one is used. This accounts for quite a few cases, which means this argument is often redundant.

2.9 Summary

Much of the power of Python comes from its powerful third-party ecosystems: whether for data science or networking code, there are many good options. Understanding how to install, use, and update third-party packages are crucial to using Python well.

With private package repositories, using Python packages for internal libraries, and distributing them in a way compatible with open source libraries, is often a good idea. It allows using the same machinery for internal distribution, versioning, and dependency management.

They can be used to calculate sales tax in the Bay area:

```
>>> rate = 9.25
>>> price = 5.99
>>> after_tax = price * (1 + rate / 100.)
>>> after_tax
6.544075
```

Or they can answer important questions about the operating environment:

```
>>> import os
>>> os.path.isfile(os.path.expanduser("~/.bashrc"))
True
```

Using the Python native console without `readline` is unpleasant. Rebuilding Python with readline support is a good idea, so that the native console will be useful. If this is not an option, using one of the alternative consoles is recommended. For example, a locally built Python with specific options might not include readline, and it might be problematic to redistribute a new Python to the entire team.

If readline support is installed, Python will use it to support line editing and history. It is also possible to save the history using readline.write_history_file. This is often after having used the console for a while, in order to have a reference for what has been done, or to copy whatever ideas worked into a more permanent form.

When using the console, the _ variable will have the value of the last expression-statement evaluated. Note that exceptions, statements that are not expressions and statements stat are expressions that evaluate to None will not change the value of _. Thi is useful during an interactive session when only after having seen the representation of the value, do we realize we needed that as an object.

```
>>> import requests

>>> requests.get("http://en.wikipedia.org")
<Response [200]>
>>> a=_
>>> a.text[:50]
'<!DOCTYPE html>\n<html class="client-nojs" lang="er
```

For the lowest-level use of code, if you want to own the UI yourself, code.compile_command(source, [filename="<input>"], symbol="single") will return a code o (that can be passed to exec), None if the command is incomplete or raise SyntaxError, OverflowError, or ValueError if there is a problem with the command.

The symbol argument should almost always be "single." The exception is if the user is prompted to enter code that will evaluate to an expression (for example, if the value is to be used by the underlying system). In that case, symbol should be set to "eval."

This allows us to manage interacting with the user ourselves. It can be integrated with a UI, or a remote network interface, to allow interactivity in any environment.

3.3 ptpython

The ptpython tool, short for “prompt toolkit Python,” is an alternative to the built-in REPL. It uses the prompt toolkit for console interaction, instead of readline.

Its main advantage is the simplicity of installation. A simple pip install ptpython in a virtual environment and regardless of readline build problems, a high-quality Python REPL appears.

ptpython supports completion suggestions, multiline editing, and syntax highlighting.

On startup, it will read ~/.ptpython/config.py. This means it is possible to locally customize ptpython in arbitrary ways. The way to configure is to implement a function, configure, which accepts an object (of type PythonRepl) and mutates it.
There are a lot of possibilities, and sadly the only real documentation is the source code. The relevant reference __init__ is ptpython.python_input.PythonInput. that config.py really is an arbitrary Python file. Therefore, if you want to distribute modifications internally, it is possible to distribute a local PyPI package and have people import a configure function from it.

3.4 IPython

IPython, which is also the foundation of Jupyter, which will be covered later, is an interactive environment whose roots are in the scientific computing community. IPython is an interactive command prompt, similar to the ptpython utility or Python’s native REPL.

IPython can be customized in a number of ways. While in an interactive session, the %config magic command can be used to change any option. For example, %config InteractiveShell.autocall = True will set the autocall option, which means expressions that are callable are called, even without parentheses. This is moot for any options that only affect startup. We can change these options, as well as any others, using the command line. For example, ipython --InteractiveShell.autocall=True, will launch into an autocalling interpreter.

If we want custom logic to decide on configuration, we can run IPython from a specialized Python script.

```
from traitlets import      config
import IPython

my_config = config.Config()
my_config.InteractiveShell.autocall =   True

IPython.start_ipython(config=my_confic
```

If we package this in a dedicated Python package, we can distribute it to a team using either PyPI or a private package repository. This allows having a homogenous custom IPython configuration for a development team.
Finally, configuration can also be encoded in profiles, which are Python snippets located under ~/.ipython by default. The profile directory can be modified by an explicit command-line parameter --ipython-dir or an environment variable IPYTHONDIR.

3.5 Jupyter Lab

Jupyter is a project that uses web-based interaction to allow for sophisticated exploratory programming. It is not limited to Python, though it does originate in Python. The name stands for "Julia/Python/R," the three languages most popular for exploratory programming, especially in data science.

Jupyter Lab, the latest evolution of Jupyter, was originally based on IPython. It now sports a full-featured web interface and a way to remotely edit files. The main users of Jupyter tend to be scientists. They take advantage of the ability to see how results were derived to add in reproducibility and peer review.

The Console is a web-based interface to IPython. All that was said about IPyth previously (for example, the In and Out arrays). The Terminal is a full-fledged terminal emulator in the browser. This is useful for a remote terminal inside a VPN: all it needs as f as connectivity needs is an open web port, and it can also be protected in the regular way that web ports are protected: TLS, client-side certificates, and more. The text editor is use for editing remote files. This is an alternative to running a remote shell, and an editor such vi in it. It has the advantage of avoiding UI lag, while still having full file-editing capabilitie The most interesting thing to launch, though, is a notebook: indeed, many a session will using nothing but notebooks. A notebook is a JSON file that records a session. As the session unfolds, Jupyter will save "snapshots" of the notebook, as well as the latest version. A notebook is made of a sequence of Cells. The two most popular cell types are "code" and "Markdown." A "code" cell type will contain a Python code snippet. It will execute it in the context of the session's namespace. The namespace is persistent from one cell execution to the other, corresponding to a "kernel" running. The kernel accepts cell content using a custom protocol, interprets them as Python, executes them, and returns both whatever was returned by the snippet as well as output from this.

When launching a Jupyter server, by default, it will use the local IPython kernel as its only possible kernel. This means that the server will, for example, only be able to use the same Python version and the same set of packages. However, it is possible to connec a kernel from a different environment to this server. The only requirement is that the environment has the ipykernel package installed. From the environment, run:

```
python -m ipykernel install \
        --name my-special-env
        --display-name "My Env
        --prefix=$DIRECTORY
```

Then, from the Jupyter server environment, run:

```
jupyter kernelspec install $DIRECTORY/jupyter/kernels/my-special-er
```

This will cause the Jupyter server in this environment to support the kernel from the special environment. This allows running one semipermanent Jupyter server and connecting kernels from any environment that is "interesting": installing specific modules, running a specific version of Python, or any other difference. One other usage of alternative kernels, which will not be covered here in details, is alternative

integrating it into a development flow takes some care, as an emergency measure to fix and continue, this is invaluable.

Last, there is a remote browser-based terminal. Between the terminal, the file editor and the file manager, a running Jupyter server allows complete browser-based remote access and management, even before thinking of the notebooks. This is important to remember for security implications, but it is also a powerful tool whose various uses we will explore later. For now, suffice it to say, the power that using a Jupyter notebook brings to remote system administration tasks is hard to overestimate.

3.6 Summary

The faster the feedback cycle, the faster we can deploy new, tested, solutions. Using Python interactively allows getting the quickest possible feedback: immediate.

This is often useful to clarify a library's documentation, a hypothesis about a running system, or just your understanding of Python.

The interactive console is also a powerful control panel from which to launch computations when the end result is not well understood: for example, when debugging the state of software systems.

A simple example of a binary file is the GIMP "XCF" internal format. GIMP is an image manipulation program, and it saves files in its internal XCF format with more details than images have. For example, layers in the XCF will be separate, for easy editing.

```
>>> with
        open("Untitled.xcf", "rb")   as fp:
...         header = fp.read(100)
```

Here we open a file. The rb argument stands for "read, binary." We read the first hundred bytes. We will need far fewer, but this is often a useful tactic. Many files have some metadata at the beginning.

```
>>> header[:9].decode('ascii')
'gimp xcf '
```

The first nine characters can actually be decoded to ASCII text, and happen to be the name of the format.

```
>>> header[9:9+4].decode('ascii')
'v011'
```

The next four characters are the version. This file is the 11th version of XCF.

```
>>>header[9+4
0
```

A 0 byte finishes the "what is this file" metadata. This has various advantages.

```
>>> struct.unpack('>I', header[9+4+1:9+4+1+4
(1920,)
```

The next four bytes are the width, as a number in big-endian format. The struct module knows how to parse these. The > says it is big endian, and the I says it is an unsigned 4-byte integer.

```
>>> struct.unpack('>I', header[9+4+1+4:9+4+1+4+4])
(1080,)
```

The next four bytes are the width. This simple code gave us the high-level data: it confirmed that this is XCF, it showed what version of the format it is, and we could see the dimensions of the image.

The rename system call is wrapped in the os.rename Python function. Since rename is atomic, this can help implement operations that require a certain state.

In general, note that the os module tends to be a thin wrapper over operating system calls. The discussion here is relevant to UNIX-like systems: Linux, BSD-based systems, and, for the most part, Mac OS X. It is worth keeping in mind, but it is not worth pointing each place where we are making UNIX-specific assumptions.

For example,

```
with open("important.tmp", "w") as fou
    fout.write("The horse raced past the barn
    fout.write("fell.\n")
os.rename("important.tmp", "importa
```

This ensures that when reading the importarfile, we do not accidentally misunderstand the sentence. If the code crashes in the middle, instead of believing that the horse raced past the barn, we get nothing from important. We only rename important.tmp to important at the end, after the last word has been written to the file. The most important example of a file-which-is-not-a-blob, in UNIX, is a directory. The os.makedirfunction allows us to ensure a directory exists easily with

```
os.makedirs(some_path, exists_True)
```

This combines powerfully with the path operations from os.pathto allow safe creation of a nested file:

```
def open_for_write(fname, mode=

    os.makedirs(os.path.dirname(fname), exists_ok=T
    return open(fname, "w" + mode)

with open_for_write("some/deep/nested/name/of/file.t as fp:
    fp.write("hello world")
```

This can come in useful, for example, when mirroring an existing file layout. The os.path module has, mostly, string manipulation functions that assume strings are file names. The dirname function returns the directory name, so os.path.dirname("a/b/c") would return a/b. Similarly, the function basename returns the “file name,” so os.path.basename("a/b/c") would return c. The inverse of both is the os.path.join function, which join paths: os.path.join("some", "long/and/win "path") would return some/long/and/finding/path.

It is also a powerful alternative to calling the os.system function, which is problematic in several ways. For one, os.system spawns an *extra* process, the shell. This means that it depends on the shell, which on some weirder installation can differ with a more "exotic" system shell like ash or fish. Finally, it means that the shell will *parse* the string, which means the string has to be properly serialized. This is a hard task to do, since the formal specification for the shell parser is long. Unfortunately, it is *not* hard to write something that will work fine most of the time, so most bugs are subtle and break at the worst possible time. This sometimes even manifests as a security flaw.

While subprocess is not *completely* flexible, for most needs, this module is perfectly adequate.

subprocess itself is also divided into high-level functions and a lower-level implementation level. The high-level functions, which should be used in most circumstances, are check_call and check_output. Among other benefits, they behave like running a shell with -e, or set err – they will immediately raise an exception if a command returns with a non-zero value.

The slightly lower-level is Popen, which creates processes and allows fine-grained configuration of their inputs and outputs. Both check_call and check_output are implemented on top of Popen. Because of that, they share some semantics and arguments. The most important argument is shell=True, and it is most important in that it is almost always a bad idea to use it. When the argument is given, a string is expected, and is passed to the shell to parse it.

Shell parsing rules are subtle and full of corner cases. If it is a constant command, there is no benefit there: We can translate the command to separate arguments in code. If it includes some input, it is almost impossible to reliably escape it in a way that makes it impossible to introduce an injection problem. On the other hand, without this, creating commands on the fly is reliable, even in the face of potentially hostile inputs.

The following, for example, will add a user to the docker group.

```
subprocess.check_call(["usermod", "-G", "docker", "some-use
```

Using check_calmeans that if the command fails for some reason, such as the user not existing, this will automatically raise an exception. This avoids a common failure mode, where scripts do not report accurate status.

If we want to make it into a function that takes a username, it is straightforward:

```
def add_to_docker(usernar
    subprocess.check_call(["usermod", "-G", "docker", usernan
```

If longer input is needed, having the communicate buffer it all in memory might be problematic. While it is possible to write to the process in chunks, doing it without potentially getting deadlocks is nontrivial. The best option is often to use a temporary file:

```
with tempfile.TemporaryFile() as fp:
    fp.write(contents)        fp.write(of)
    fp.write(email)              fp.flush()
    fp.seek(0)              proc              =
    Popen(["sendmail"],       stdin=fp)
    result = proc.poll()
```

In fact, in this case, we can even use the check_call function:

```
with tempfile.TemporaryFile() as fp:
    fp.write(contents)
    fp.write(of)
    fp.write(email)
    fp.flush()
    fp.seek(0)
    check_call(["sendmail"], stdin=fp)
```

If you are used to running processes in shell, you are probably used to long pipelines:

```
$ ls -l | sort | head -3 | awk '{print $3}'
```

As noted above, it is a best practice in Python to avoid true command parallelism: in all of the cases, we tried to finish one stage before reading from the next. In Python, in general, using subprocess is only used for calling out to external commands. For preprocessing of inputs, and post-processing of outputs, we usually use Python's built-in processing abilities: in the case above, we would use sorted slices and string manipulation to simulate the logic.

The commands for text and number processing are seldom useful in Python, which has a good in-memory model for doing such processing. The general case for calling commands in scripts is for things that manipulate data in a way that is either only documented as accessible by commands – for example, querying processes via ps -ef, or where the alternative to the command is a subtle library, sometimes requiring binary binding, such as in the case of docker or git.

After the socket is connected, we can send bytes to it. Note – on sockets, only byte strings can be sent. We read back the result and do some ad hoc HTTP response parsing – and parse the actual content as JSON.

While in general, it is better to use a real HTTP client, this showcases how to write low-level socket code. This can be useful, for example, if we want to diagnose a problem by replaying exact messages.

The socket API is subtle, and the above example has a few incorrect assumptions in it. In most cases, this code will work but will fail in strange ways in the face of corner cases.

The send method is allowed to not send all the data, if not all of it can fit into the internal kernel-level send buffer. This means that it can do a "partial send." It returned 40, above, which was the entire length of the byte string. Correct code would have checked for the return value and send the remaining chunks until nothing is left. Luckily, Python already has a method to do it: sendall.

However, a more subtle problem occurs with recv. It will return as much as the kernel-level buffer has, because it does not know how much the other side intended to send. Again, much of the time, especially for short messages, this will work fine. For protocols like HTTP 1.0, the correct behavior is to read until the connection is closed.

Here is a fixed version of the code:

```
>>> import socket, json, pprint
>>>s = socket.socket()
>>>s.connect(('httpbin.org', 80))
>>>s.sendall(b'GET /get HTTP/1.0\r\n Host: httpbin.org\r\n\r\n ')
>>>resp = b''
>>> while True:
...         more = s.recv(1024)
...     ...     if more == b'':
... ...                 break
...         resp += more

>>>pprint.pprint(json.loads(resp.decode('ascii').split('  \r\n\r\n ')[1]))
{'args': {},
 'headers': {'Connection': 'close', 'Host': 'httpbin.org'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

The code above would be better written as:

```
>>> import requests, pprint
>>> session = requests.Session()
>>> res = session.get('http://httpbin.org/get')
>>> pprint.pprint(res.json())
{'args': {},
 'headers': {'Accept': '*/*',
             'Accept-Encoding': 'gzip, deflate',
             'Connection': 'close', 'Host':
             'httpbin.org', 'User-Agent': 'python-
             requests/2.19.1'},
 'origin': '73.162.254.113',
 'url': 'http://httpbin.org/get'}
```

In this example, the request is simple and session state does not matter. However, this is a good habit to get into: even in the interactive interpreter, to avoid using the get, put, and other functions directly and using only the session interface.

It is natural to use an interactive environment to prototype code, which would later make it into a production program. By keeping good habits like this, we ease the transition

4.4 Summary

Python is a powerful tool for automating operating system operations. This comes from a combination of having libraries that are thin wrappers around native operating system calls and powerful third-party libraries.

This allows us to get close to the operating systems, without any intervening abstractions, as well as to write high-level code that does not care about the details when these do not matter.

This combination often makes Python a superior alternative for writing scripts, instead of using the UNIX shell. It does require a different way of thinking: Python is not as suitable for the long pipeline of text transformers approach, but in practice, those long pipelines of text transformers turn out to be an artifact of shell limitations.

With a modern memory-managed language, it is often easier to read the entire text stream into memory, and then manipulate it without being limited to only those transformations that can be specified as pipes.

trickier flows, they are often in a position to add that test *before* such a bug makes it out into the externally-observed code change: however, such a confidence-increasing test looks exactly like a "regression test."

A final reason is to avoid incorrect future changes. This is different from the "regression test," above, in that often the case being tested is straightforward for the code as is, and the flows involved are already covered by other tests. However, it seems like some potential optimizations or other natural changes might break this case, so including it helps a future maintenance programmer.

When writing a test, it is important to think about which of those goals it is meant to accomplish.

A good test will accomplish more than one. All tests have two potential impacts:

- Make the code better by helping future maintenance work.
- Make the code worse by making future maintenance work harder.

Every test will do some of both. A good test does more of the first, and a bad test does more of the second. One way to reduce the bad impact is to consider the question: "Is this test testing something that the code promises to do?" If the answer is "no," this means it is valid to change the code in some way that will break the test, but will not cause any bugs. This means the test has to be changed or discarded.

When writing tests, as much as possible, it is important to test the actual contract of the code.

Here is an example:

```
def write_numbers(fout):
    fout.write("1\n")
    fout.write("2\n")
    fout.write("3\n")
```

This function writes a few numbers into a file.

A bad test might look like this:

```
class DummyFile:

    def __init__(self):
        self.written = []

    def write(self, thing):
        self.written.append(thing)
```

A good test is one that would only break if there was a bug in the code. However, this code still works for the users of write_numbers, meaning that the maintenance now involved unbreaking a test, pure overhead.

Since the contract is to be able to write to file objects, it is best to supply a file object. In this case, Python has one ready-made:

```
def test_write_numbers():

    fout = io.StringIO()
    write_numbers(fout)
    assert_that(fout.getvalue(), is_("1      \n2\n3\n"))
```

In some cases, this will require writing a custom fake. We will cover the concept of fakes, and how to write them, later.

We talked about the *implicit* contract of write_numbers. Since it had no documentation, we could not know what the original programmer's intent was. This is, unfortunately, common – especially in internal code, only used by other pieces of the project. Of course, it is better to clearly document programmer intent. In the face of lack of clear documentation, however, it is important to make *reasonable assumptions* on the implicit contract.

Above, we used the functions assert_that and is_ to verify that the values were what we expected. Those functions come from the hamcrest library. This library, ported from Java, allows specifying properties of structures and checks that they are satisfied.

When using the pytest test runner to run unit tests, it is possible to use regular Python operators with the assert keyword and get useful test failures. However, this binds the tests to a specific runner, as well as having a specific set of assertions that get treated especially for useful error messages.

Hamcrest is an open-ended library: while it has built-in assertions for the usual things (equality, comparisons, sequence operations, and more), it also allows defining specific assertions. Those come in handy when handing complicated data structures, such as those returned from APIs, or when only specific assertions can be guaranteed by the contract (for example, the first three characters can be arbitrary but must be repeated somewhere inside of the rest of the string).

This allows to test the *exact* contract of the function. In particular, this is another tool in avoiding testing "too much": testing implementation details that can change, requiring changing the test when no real users have been broken. This is crucial for three reasons.

We would get an error like the following:

```
Expected: (number divisible by 3 or number divisible by 7)
     but: was <17>
```

It lets us test exactly what the contract of scale_one promises: in this case, that it would scale up one of the arguments by an integer factor.

The emphasis on the importance of testing precise contracts is not accidental. This emphasis, which is a skill that is possible to learn and has principles that are possible to teach, makes unit tests into something that accelerate the process of writing code rather than make it slower.

Much of the reason people have aversion to unit tests as "something that wastes time for DevOps engineers" and leads to a lot of poorly tested code that is foundational for business processes such as deployment of software is this misconception. Properly applying principles of high-quality unit testing leads to a more reliable foundation for operational code.

5.2 Mocks, Stubs, and Fakes

Typical DevOps code has outsized effects on the operating environment. Indeed, this is almost the definition of good DevOps code: it replaces a significant amount of manual work. This means that *testing* DevOps code needs to be done carefully: we cannot simply spin up a few hundred virtual machines for each test run.

Automating operations means writing code that, run haphazardly, can have significant impact on production systems. When testing the code, it is worthwhile to have as few of these side effects as possible. Even if we have high-quality staging systems, sacrificing one every time there is a bug in operational code would lead to a lot of wasted time. It is important to remember that unit tests run on the *worst* code produced: the act of running them, and fixing bugs, means even code committed into feature branches is likelier to be in better condition.

Because of that, we often try to run unit tests against a "fake" system. Classifying what we mean by "fake," and how it impacts both unit tests and code design, is important: it is worthwhile thinking about how to test the code well before starting to write it.

The neutral term for things that substitute for the systems not under test is "test doubles." Fakes, mocks, and stubs usually have more precise meaning, though in casual conversation they will be used interchangeably.

First, filesystems tend to be robust. While bugs in filesystems are not unknown, they are rare, far between, and usually only triggered by extreme conditions or an unlikely combination of conditions.

Next, filesystems tend to be fast. Consider the fact that unpacking a source tarball, a routine operation, will create many small files (on the order of several kilobytes) in quick succession. This is a combination of fast system-call mechanisms combined with sophisticated cache semantics when reading or writing files.

Filesystems also have a curious fractal property: with the exception of some esoteric operations, a sub-sub-sub-directory supports the same semantics as the root directory.

Finally, filesystems have a very thick interface. Some of it will be built into Python, even – consider that the module system reads files directly. There are also third-party C libraries that will use their own internal wrappers to access the filesystem as well as several ways to open files even in Python: the built-in file object as well as the os.open low-level operations.

This combines to the following conclusion: for most file-manipulation code, faking out or mocking the filesystem is a low return on investment. The investment, in order to make sure we are only testing the contract of a function, is considerable; since the function could, conceivably, switch to low-level file-manipulation operations, we would need to reimplement a significant portion of Unix file semantics. The return is low; using the filesystem directly is fast, reliable, and, as long as the code merely allows us to pass an alternative "root path," almost side-effect free.

The best way to design file-manipulation code is to allow passing in such a "root path" argument, even if the default is /. Given such design, the best way to test is to create a temporary directory, populate it appropriately, call the code, and then garbage collect the directory.

If we create the temporary directory using Python's built-in tempfile module, then we can configure the Tox runner to put the temporary file inside of Tox's built-in temporary directory, thus keeping the general file system clean and, usually, being compatible with whatever version control ignorefile already ignores Tox artifacts.

```
setenv =
    TMPDIR = {envtmpd
command
    python -m 'import os;os.makedirs(sys.argv[1])' {envtmpdir}
    # rest of test commands
```

We first create a context manager for our temporary directory. This will ensure, as much as ensuring is possible, that the temporary directory will be cleaned up.

```
@contextlib.contextmanager
def get_temp_dir():

    temp_dir = tempfile.mkdtemp
    try:
        yield temp_dir
    finally:
        shutil.rmtree(temp_dir)
```

Since this test needs to create a lot of files, and we do not care about their contents too much, we define a helper method for that.

```
def touch(fname, content=''):

    with open(fname, 'a') as fpin:
        fpin.write(content)
```

Now with the help of these functions, we can finally write a test:

```
def test_javascript_to_python_simple():
    with get_temp_dir() as temp_dir:
        touch(os.path.join(temp_dir,      'foo.js'))
        touch(os.path.join(temp_dir,     'bar.py'))
        touch(os.path.join(temp_dir,    'baz.txt'))
        javascript_to_python(temp_dir)
        assert_that(set(os.listdir(temp_dir)),
                                is_({'foo.py', 'bar.py', 'baz.txt'}))
```

For a real project, we would write more tests, many of them possibly using our get_temp_dir and touch helpers above.

If we have a function that is supposed to check a specific path, we can have it take an argument to “relativize” its paths.

For example, let us say we want a function to analyze our Debian installation paths and give us a list of all domains that we download packages from.

```
def _analyze_debian_paths_from_file(fpin):

    for line in fpin:
        line = line.strip()
```

```
            with open(os.path.join(sources_dir, fname)) as fpin:
              yield from _analyze_debian_paths_from_file(fpin)
```

Now, using the same helpers as before, we can write a simple test for this:

```
def test_analyze_debian_paths():
    with get_temp_dir() as root:
        touch(os.path.join(root, 'foo.list'),
                content='deb http://foo.example.com\n'
        ret = list(analyze_debian_paths(relative_to=root))
        assert(ret, equals_to(['foo.example.com']))
```

Again, in a real project, we would write more than one test and try to make sure many more cases are covered. Those could be built using the same techniques.

It is a good habit to add a relative_to parameter to any function that accesses specific paths.

5.4 Testing Processes

Testing process-manipulation code is often a subtle endeavor, full of trade-offs. In theory, process running code has a thick interface with the operating system; we covered the subprocess module, but it is possible to use the os.spawn* functions directly, or even use code os.fork and os.exec* functions. Likewise, the standard output/input communication mechanism can be implemented in many ways, including using the Popen abstraction or directly manipulating file descriptors with os.pipe and os.dup.

Process-manipulation code can also be some of the most fragile. Running external commands depends on the behavior of those commands, as a starting point. The inter-process communication means that the flow is inherently concurrent. It is too easy to make the mistake of making the tests rely on ordering assumptions that are not always true. Those mistakes can lead to "flaky" tests: ones that pass most of the time, but fail under seemingly random circumstances.

Those ordering assumptions can sometimes be true more often on development machines, or unloaded machines, which means bugs will only be exposed in production, or possibly in production only in extreme circumstances.

```
        if args[1] != 'logs':
            raise ValueError("Can only run docker logs", args)
        if args[2] != container_name:
            raise ValueError("No such container", args[2])
        return iter(["hello\n", "error: 5 is not 6\n", "goodbye          \n"])
    ret = error_lines(container_name, runner=runner)
    assert_that(list(ret), is_(["error: 5 is not 6"))
```

Note that, in this case, we did not restrict ourselves to *only* checking the contract: error_lines could have run, for example, docker logs -- <container_name>. However, one advantage of our method is that we can slowly improve our fidelity and *only* improve the test.

For example, we can add to runner:

```
def runner(args):
    if args[0] != 'docker':
        raise ValueError("Can only run docker", args)
    if args[1] != 'logs':
        raise ValueError("Can only run docker logs", args)
    if args[2] == '--':
        arg_container_name = args[3
    else:
        arg_container_name = args[2
    if args_container_name != container_nam
        raise ValueError("No such container", args[2])
    return iter(["hello\n", "error: 5 is not 6\n", "goodbye          \n"])
```

This will *still* work with the old version of the code and will also work with post-modification code. Fully emulating the docker is not realistic or worthwhile. However, this approach would slowly improve the accuracy of the test, with no downsides.

If a significant amount of our code interfaces, for example, with docker, we can eventually factor out a mini-docker-emulator like that into its own test helper library.

Using higher-level abstractions for process running helps with this sort of approach. The seashore library, for example, separates the part that calculates the commands from the low-level runner, which allows substituting only the low-level one.

Because processes are so hard to test, it is good to use process running only when necessary. Especially when porting over shell scripts to Python – often a good idea when they grow in complexity – it is good to substitute long pipelines with in-memory data processing.

Especially if we factor the code the right way, with the data processing as a simple pure function that takes an argument and returns a value, the bulk of the code becomes a pleasure to test.

Imagine, for example, the pipeline,

```
ps aux | grep conky | grep -v grep | awk '{print $2}' | xargs kill
```

This will kill all processes that have conky in their names.

Here is a way to refactor the code to make it easier to test:

```
def get_pids(lines):
    for line in lines:
        if 'conky' not in line:
            continue
        parts = line.split()
        pid_part = parts[1]
        pid = int(pid_part)
        yield pid

def ps_aux(runner=subprocess.check_outpu
    return runner(["ps", "aux"])

def kill(pids, killer=os.kill):
    for pid in pids:
        killer(pid, signal.SIGTERM)

def main():
    kill(get_pid(ps_aux()))
```

Note how the most complicated code is now in a pure function: get_pids. Hopefully, this means most bugs will be there, and we can unit test against them.

The code that is harder to unit test, get_pids, where we have to do ad hoc dependency injection, is now in simple functions that have fewer failure modes.

The main logic is in functions that do data processing. Testing those just requires supplying simple data structure and observing the return value. *Moving* potential bugs

```
@attr.s(frozen=True):
class Response:

    content = attr.ib()

    def json(self):
        return json.loads(content)

@attr.s(frozen=True)
class FakeSession:

    _gists = attr.ib()

    def get(self, url):
        parsed = hyperlink.URL.from_text(url)
        if parsed.host == 'api.github.com':
            tail = path.rsplit('/', 1)[-1]
            gist = self._gists[tail]
            res = dict(files={name: f'http://example.com/ {tail}/{name}'
                              for name in gist.files})
            return Repsonse(json.dumps(res
        if parsed.host == 'example.com':
            _ignored, gist, name = path.split('/')
            return Response(self.gists[gist][name])
```

This is a bit long-winded. We can sometimes, if this functionality is localized and writing a whole helper library is not worth it, use the unittest.mock library.

```
def make_mock(
    gist_name = 'some_name'
    files = {'some_file': 'some_content'}
    session = mock.Mock()
    session.get.content.return_value = 'some_content'
    session.get.json.return_value = json.dumps({'files': 'some_file'})
    return session
```

This is a "quick and dirty" hack, counting on the fact (that is *not* in the contract) that the file content is retrieved using content, and the gist's logical structure is retrieved using json. However, it is often better to write a quick test using mocks that depend a little on the implementation details rather than not writing a test at all.

This allows us to control the size of "chunks." An extreme test would be to use a chunk_size of 1. This means bytes would go out one at a time, and they would be received one at a time. No real network would be this bad, but a unit test allows us to simulate more extreme conditions than any reasonable network.

This fake is useful to test networking code. For example, this code does some ad hoc HTTP to get a result:

```
def get_get(sock):

    sock.connect(('httpbin.org', 80))
    sock.send(b'GET /get HTTP/1.0\r\nHost: httpbin.org\r\n\r\n ')
    res = sock.recv(1024)
    return json.loads(res.decode('ascii').split('       \r\n\r\n ', 1)[1]))
```

It has a subtle bug in it. We can uncover the bug with a simple unit test, using the socket fake.

```
def test_get_get():

    result = dict(url='http://httpbin.org/get')
    headers = 'HTTP/1.0 200 OK\r\nContent-Type: application/jso\r\n\r\n'
    output = headers + json.dumps(result)
    fake_sock = FakeSimpleSocket(to_send=output, chunk_size=
    value = get_get(fake_sock)
    assert_that(value, is_(result))
```

This test would fail: our `get_get` assumes a good quality network connection, and this simulates a bad one. It would succeed if we changed `chunk_size` to `1024`.

We could run the test in a loop, testing chunk sizes from 1 to 1024. In a real test we would also check the sent data, and possibly also send invalid results to see the response. The important thing, however, is that none of those things need setting up clients or servers, or trying to realistically simulate bad networks.

5.6 Summary

Teams rely on DevOps code to keep systems functional and observable. The correctness of DevOps code is critical. Writing proper tests will help improve code correctness. Taking proper test-writing principles into account will help reduce the burden of modifying tests when making correct changes to the code.

ASCII only encompasses the English alphabet, used in "America." In order tc represent text in (almost) any language, we have Unicode. Unicode code points are (some of the) numbers between 0 and 2 * *32 (including 0 and not including 2 * *32). Each Unicode code point is assigned a meaning. Successive versions of the standards leave assigned meanings as is, but add meanings to more numbers. An example is the addition of more emojis. The International Standards Organization, ISO, ratifies versions of Unicode in its 10464 standards. For this reason, Unicode is sometimes callec IS O-10464.

Unicode points that are also ASCII have the same meaning – if ASCII assigns a number "uppercase A," then so does Unicode.

Properly speaking, only Unicode is "text." This is what Python strings represent. Converting bytes to strings, or vice versa, is done with an *encoding*. The most popular encoding these days is UTF-8. Confusingly, turning the bytes *to* text is "decoding." Turning the text to bytes is "encoding."

Remembering the difference between encoding and decoding is crucial in order to manipulate textual data. A way to remember it is that since UTF-8 *is* an encoding, moving from strings *to* UTF-8 encoded data is "encoding," while moving from UTF-8 encoded data to strings is "decoding."

UTF-8 has an interesting property: when given a Unicode string that happens to be ASCII, it will produce bytes with the values of the code points. This means that "visually,' the encoded and decoded form will look the same.

```
>>> "hello".encode("utf-8")
b'hello'
>>> "hello".encode("utf-16")
b'\xff\xfeh\x00e\x00l\x00l\x00o\x00'
```

We show the example with UTF-16 to show that this is not a trivial property of encodings. Another property of UTF-8 is that if the bytes are *not* ASCII, and UTF-8 decoding of the bytes succeeds, then it is unlikely that they were encoded with a different encoding. This is because UTF-8 was designed to be *self-synchronizing*: starting at a random byte, it is possible to synchronize with the string with a limited number of bytes being checked. Self-synchronization was designed to allow recovery from truncation and corruption, but as a side benefit, it allows *detecting* invalid characters reliably, and thus detect if the string was UTF-8 to begin with.

```
0 h 1
1 e 1
2 l 1
3 l 1
4 o 1
```

The string “hello” has five elements, each of which is a string of length 1. Since the string is a sequence, the usual sequence operations work on it.

We can create a slice by specifying both endpoints:

```
>>> a[2:4]
'll'
```

or just the end:

```
>>> a[:2]
'he'
```

or just the beginning:

```
>>> a[3:]
'lo'
```

We can also use negative indices to count from the end:

```
>>> a[:-3]
'he'
```

And of course, we can reverse a string by specifying an extended slice with a negative step:

```
>>> a[::-1]
'olleh'
```

However, strings also have quite a few methods that are *not* part of the genera sequence interface and are useful when analyzing text.

The startswith and endswith methods are useful, since text analysis is often around the ends.

```
>>> "hello world".endswith("world")
True
```

Finally, the join method on a string uses it as a "glue" and glues together an iterable of strings.

The simple example of ' '.join(["hello", "world"]) will return "hello world," but this is only scratching the surface of join. Since it accepts an iterable, we have the ability to pass it anything that supports iteration.

```
>>> names=dict(hello=1,world=2)
>>> ' '.join(names)
'hello world'
```

Since iterating on a dictionary objects yields the list of keys, passing it to join means that we get a string with the list of keys, joined together.

We can also pass in a generator:

```
>>> '-*-'.join(str(x)      for  x  in  range(3))
'0-*-1-*-2'
```

This allows calculating sequences on the fly and joining them, without the need to have intermediate storage for the sequence.

The usual question about join is why it is a method on the "glue" string rather than a method on sequences. The reason is exactly this: we can pass in any iterable, and the glue string will glue in the bits in it.

Note that join does nothing to single-element iterables:

```
>>> '-*-'.join(str(x)      for  x  in  range(0))
'0'
```

6.3 Regular Expressions

Regular expressions are a special DSL for specifying properties of strings, also called "patterns." They are common in many utilities, although each implementation will have its own idiosyncrasies. In Python, regular expressions are implemented by the re module. It fundamentally allows two modes of interaction – one where regular expressions are auto-parsed at the time of text analysis, and one where they are parsed in advance.

For example, (?:[a-z]{2,5}-){1,4}[0-9] will match hello-3 or hello-world-5 but not a-hello-2 (since the first part is not two characters long) or hello-world-this-is-too-long-7 since it is made up of six repetitions of the inner pattern, and we specified a maximum of four.

This allows arbitrary nesting; for example (?:(?:[a-z]{2,5}-){1,4}[0-9];)+ allows any semicolon-terminated, separated sequence of the previous pattern: for example az-2;hello-world-5; will match but this-is-3;not-good-match-6 will since it is missing the ; at the end.

This is a good example of how complex regular expressions can get. It is easy to use this dense mini-language inside Python to specify constraints on strings that are hard to understand.

Once we have a regular expression object, there are two main methods on it: match and search. The match method will look for matches at the beginning of the string, while search will look for the first match, wherever it may start. When they find a match, they return a match object.

```
>>> reobj = re.compile('ab+a')
>>> m = reobj.search('hello abba world')
>>> m
<_sre.SRE_Match object; span=(6, 10), match='abba'>
>>> m.group()
'abba'
```

The first method that is often used is group(), which returns the part of the string matched. This method can get a part of the match, if the regular expression contained *capturing* groups. A capturing group is usually marked with ().

```
>>> reobj = re.compile('(a)(b+)(a)')
>>> m = reobj.search('hello abba world')
>>> m.group()
'abba'
>>> m.group(1)
'a'
>>> m.group(2)
'bb'
>>> m.group(3)
'a'
```

6.4 JSON

JSON is a hierarchical file format that has the advantage of being simple to parse, and reasonably easy to read and write by hand. It has its origins on the web: the name stands for "JavaScript Object Notation." Indeed, it is still popular on the internet; one reason to care about JSON is that many web APIs use JSON as a transfer format.

It is also useful, however, in other places. For example, in JavaScript projects package.json includes the dependencies of this project. Parsing this is often useful to determine third-party dependencies for security or compliance audits, for example.

In theory, JSON is a format defined in *Unicode*, not *bytes*. When serializing, it takes a data structure and transforms it into a Unicode string, and when deserializing, it takes a Unicode string and returns a data structure. Recently, however, the standard was amended to specify a preferred encoding: utf-8. With this addition, now the format is also defined as a byte stream.

However, note that in some use cases, the encoding is still separate from the format In particular, when sending or receiving JSON over HTTP, the HTTP encoding is the ultimate truth. Even then, though, when no encoding is explicitly specified, UTF-8 should be assumed.

JSON is a simple serialization format, only supporting a few types:

- Strings
- Numbers
- Booleans
- A null type
- Arrays of JSON values
- "Objects": dictionaries mapping strings to JSON values

Note that JSON does not full specify numerical ranges or precision. If precise integers are required, usually the range -2 * *53 to 2 * *53 can be assumed to be represen precisely.

Although the Python json library has the ability to read/write directly to files, in practice we almost always separate the tasks; we read as much data as we need and pas the string directly to JSON.

Note that with Python 3.7 and above, sort_keys should be used judiciously; since all dictionaries are ordered by insertion, *not* using sort_keys will keep the original order in the dictionary.

One frequently missed type from JSON is a date-time type. Usually this is represented with strings, and is the most common need for a "schema" to parse JSON against, in order to know which strings to convert to a datetime object.

6.5 CSV

The CSV format has a few advantages. It is constrained: it always represents scalar type in a two-dimensional array. For this reason, there are not a lot of surprises that can go in. In addition, it is a format that imports natively into spreadsheet applications like Microsoft Excel or Google Sheets. This comes in handy when preparing reports.

Examples of such reports are of breaking down expenses for paying for third-party services for the financial department, or a report on incidents managed and time to recovery for management. In all these cases, having a format that is easy to produce and import into spreadsheet applications allows for easy automation of the task.

Writing CSV files is done with csv.writer. A typical example involves serializing a homogenous array, an array of things with the same type.

```
@attr.s(frozen=True, auto_attribs=True)
class LoginAttempt:
    username: str
    time_stamp: int
    success: bool
```

This class represents a login attempt by some user, at a given time, and with a record of the success of the attempt. For a security audit, we need to send the auditors an Excel file of the login attempts.

```
def write_attempts(attempts, fname):

    with open(fname, 'w') as fpout:
        writer = csv.writer(fpout)
        writer.writerow(['Username', 'Timestamp', 'Success'])
        for attempt in attempts:
            writer.writerow([
```

For example,

1,"Miami, FL","he""llo"

is properly parsed as
('1', 'Miami, FL', 'he"llo')

For the same reason, it is a good idea to avoid writing CSV files using anything other than csv.writer.

6.6 Summary

Much of the content that is needed for many DevOps tasks arrives as text: logs, JSON dumps of data structures, or a CSV file of paid licenses. Understanding what "text" is and how to manipulate it in Python allow much of the automation that is the cornerstone of DevOps, be it through build automation, monitoring result analysis, or just preparing summaries for easy consumption by others.

This allows initializing the session elsewhere, closer to the main code. This is useful because this means that decisions about which proxies to use, and when, can happen closer to the end-user requirements rather than in abstract library code.

A session object is constructed with requests.Session(). After that, the only interaction should be with the object. The session object has all the HTTP methods: s.get, s.put, s.post, s.patch, and s.options.

Sessions can be used as contexts:

```
with requests.Session()  as s:
    s.get(...)
```

At the end of the context, all pending connections will be cleaned up. This can sometimes be important, especially if a web server has strict usage limits that we cannot afford to exceed for any reason.

Note that counting on Python's reference counting to close the connections can be dangerous. Not only is that not guaranteed by the language (and will not be true, for example, in PyPy), but small things can easily prevent this from happening. For example, the session can be captured as a local variable in a stack trace, and that stack trace can be involved in a circular data structure. This means that the connections will not get closed for a potentially long time: not until Python does a circular garbage collection

The session supports a few variables that we can mutate in order to send all requests in a specific way. The most common one to have to edit is s.auth. We will touch more about the authentication capabilities of requests later.

Another variable that is useful to mutate is session.headers. Those are the default headers that are sent with every request. This can sometimes be useful for the User- Agent variable. Especially when using requests for testing our own web APIs, it is useful to have an identifying string in the agent. This will allow us to check the server logs and distinguish which requests came from tests as opposed to real users.

```
session.headers = {'User-Agent': 'Python/MySoftware ' + __version__ }
```

This will allow checking which version of the test code caused a problem. Especially if the test code crashes the server, and we want to disable it, this can be invaluable in diagnosis.

The session also holds a CookieJar in the cookies member. This is useful if we want to explicitly flush, or check, cookies. We can also use it to persist cookies to disk and recover them, if we want to have restartable sessions.

Most REST services, nowadays, use JSON as the state representation. The requests library has special support for JSON.

```
>>> pprint(s.get("https://httpbin.org/json").json())
{'slideshow': {'author': 'Yours Truly',
               'date': 'date of publication',
               'slides': [{'title': 'Wake up to WonderWidgets!', 'type': 'all'},
                          {'items': ['Why <em>WonderWidgets</em> are gre
                                     'Who <em>buys</em> WonderWidget
                           'title': 'Overview',
                           'type': 'all'}],
               'title': 'Sample Slide Show'}}
```

The return value from a request, `Response`, has a `.json()` method that assumes the return value is JSON and parses it. While this only saves one step, it is a useful step to save in a multistage process where we get some JSON-encoded response only to use it in a further request.

It is also possible to auto-encode the request body as JSON:

```
>>> resp = s.put("https://httpbin.org/put", json=dict(hello=5,world=2))
>>> resp.json()['json']
{'hello': 5, 'world': 2}
```

The combination of those two, with a multistep process, is often useful.

```
>>>res = s.get("https://api.github.com/repos/python/cpython/pulls")
>>>commits_url = res.json()[0]['commits_url']
>>>commits = s.get(commits_url).json()
>>>print(commits[0]['commit']['message'])
```

This example of getting a commit message from the first pull request on the CPython project is a typical example of using a good REST API. A good REST API includes *URLs* as resource identifiers. We can pass those URLs to a further request to get more information.

This is useful when making connections to the internet; almost all sites are tested to work with Firefox, and so have a compatible certificate chain. If the certificate fails to validate, the error CERTIFICATE VALIDATE FAILED is thrown. There is a lot of unfortunate advice on the internet, including in requests documentation, about the "solution" of passing in the flag verify=False. While there are rare cases where this flag would make sense, it almost never does. Its usage violates the core assumption of TLS: that the connection is encrypted and tamper-proof.

For example, having a verify=False on the request means that any cookies or authentication credentials can now be intercepted by anyone with the ability to modify in-stream packets. This is unfortunately common: ISPs and open access points often have operators with nefarious motivation.

A better alternative is to make sure that the correct certificates exist on the file system, and passing the path to the verify argument via verify='/full/path'.
At the very least, this allows us a form of "trust on first use": manually get the certificate from the service, and bake it into the code. It is even better to attempt some out-of-band verification, for example, by asking someone to log in to the server and verify the certificate.

Choosing what SSL versions to allow, or what ciphers to allow, is slightly more subtle. There are, again, few reasons to do it: requests is set up with good, secure, defaults. However, sometimes there are overriding concerns: for example, avoiding a specific SSL cipher for a regulatory reason.

The first important thing to know is that requests is a wrapper around the urllib3 library. In order to change low-level parameters, we need to write a customized HTTPAdapter and set the session object we are using to use the custom adapter.

```
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.poolmanager import
                                                  PoolManager

class MyAdapter(HTTPAdapter):
    pass

s = requests.Session()
s.mount('https://', MyAdapter())
```

7.4 Authentication

This will be the default authentication sent with requests. Included in requests itself, the most commonly used authentication is *basic auth*.
For basic auth, this argument can be just a tuple, (username, password). However, a better practice is to use an HTTPBasicAuth instance. This documents the intent better, and is useful if we ever want to switch to other authentication forms.

There are also third-party packages that implement the authentication interface and supply custom auth classes. The interface is pretty straightforward: it expects the object to be callable and will call the object with the Request object. It is expected that the call will mutate the Requests, usually by adding headers.

The official documentation recommends subclassing AuthBase, which is just an object that implements a __call__ that raises a NotImplementedError. There is little need for that.

For example, the following is useful as an object that will sign AWS requests with the V4 signing protocol.

The first thing we do is make the URL "canonical." Canonicalization is a first step in many signing protocols. Since often higher levels of the software will have already parsed the content by the time the signature checker gets to look at it, we convert the signed data into a standard form that uniquely corresponds to the parsed version.

The most subtle part is the query part. We parse it, and re-encode it, using the urlparse built-in library.

```
def canonical_query_string(query):
        if not query:
            return ""
    parsed = parse_qs(url.query, keep_blank_values=  True)
    return "?" + urlencode(parsed, doseq=True)
```

We use this function in our URL canonicalization function:

```
def to_canonical_url(url):
    url = urlparse(raw_url)
    path = url.path or "/"
    query = canonical_query_string(url.query)
```

give a good idea about how custom authentication schemes work: we write code that modifies the request to have the right authentication headers, and then put it in as the auth property on the session.

7.5 Summary

Saying “HTTP is popular” feels like an understatement. It is everywhere: from user-accessible services, through web-facing APIs, and even internally in many microservice architectures.

requests helps with all of these: it can help be part of monitoring a user-accessible service for health, it can help us access APIs in programs to analyze the data, and it can help us debug internal services to understand what their state is.

It is a powerful library, with many ways to fine-tune it to send exactly the right requests, and get exactly the right functions.

The key is a short string of bytes. Managing the key securely is important: cryptography is only as good as its keys. If it is kept in a file, for example, the file should have minimal permissions and ideally be hosted on an encrypted file system.

The generate_key class method takes care to generate the key securely, using an operating-system level source of random bytes. However, it is still vulnerable to operating-system level flaws: for example, when cloning virtual machines, care must be taken that when starting the clone, it refreshes the source of randomness. This is admittedly an esoteric case, and whatever virtualization system is being used should have documentation on how to refresh the randomness source in its virtual machines.

```
>>> frn = fernet.Fernet(k)
```

The fernet class is initialized with a key. It will make sure that the key is valid.

```
>>> encrypted = frn.encrypt(b"x marks the spot")
>>> encrypted[:10]
b'gAAAAABb1'
```

Encryption is simple. It takes a string of bytes and returns an encrypted string. Note that the encrypted string is *longer* than the source string. The reason is that it is also signed with the secret key. This means that tampering with the encrypted string is detectable, and the Fernet API handles that by refusing to decrypt the string. This means that the value gotten back from decryption is *trustworthy*; it was indeed encrypted by someone who had access to the secret key.

```
>>> frn.decrypt(encrypted)
b'x marks the spot'
```

Decryption is done in the same way as encryption. Fernet does contain a version marker, so if vulnerabilities in these are found, it is possible to move the standard to a different encryption and hashing system.

Fernet encryption always adds the current date to the signed, encrypted information. Because of this, it is possible to limit the *age* of a message before decrypting

```
>>> frn.decrypt(encrypted, ttl=5)
```

There are, in general, two basic operations supported with public-key cryptography. We can encrypt with the public key, in a way that can only be decrypted with the private key. We can also *sign* with the private key, in a way that can be verified with the public key.

As we have discussed earlier, modern cryptographic practice places as much value on *authentication* as it does on *secrecy*. This is because if the media the secret is transmitted on is vulnerable to eavesdropping, it is often vulnerable to modification. Secret modification attacks have had enough impact on the field that a cryptographic system is not considered complete if it does not guarantee both authenticity and secrecy Because of that, libsodium, and by extension PyNaCl, do not support encryption without signing, or decryption without signature verification.

In order to generate a private key, we just use the class method

```
>>> from nacl.public import        PrivateKey
>>> k = PrivateKey.generate()
```

The type of k is PrivateKey However, at some point, we will usually want to persist the private key.

```
>>> type(k.encode())
<class 'bytes'>
```

The encodmethod encodes the secret key as a stream of bytes

```
>>> kk = PrivateKey(k.encode())
>>> kk == k
True
```

We can generate a private key from the byte stream, and it will be identical. This means we can again keep the private key in a way we decide is secure enough: a secret manager, for example.

In order to encrypt, we need a *public key*. Public keys can be generated from private keys.

```
>>> from nacl.public import
                                PublicKey
>>> target = PrivateKey.generate()
>>> public_key = target.public_key
```

```
>>> dec_box = Box(target, source_public_key)
>>> dec_box.decrypt(result)
b'x marks the spot'
```

The decryption box decrypts with target private key and verifies the signature using source's public key. If the information has been tampered with, the decryption operation automatically fails. This means that it is impossible to access plain-text information that is not correctly signed.

Another piece functionality that is useful inside of PyNaCl is cryptographic signing. It is sometimes useful to sign *without* encryption: for example, we can make sure to only use approved binary files by signing them. This allows the permissions for *storing* the binary file to be loose, as long as we trust that the permissions on *keeping the signing key secure* are strong enough.

Signing also involves asymmetric cryptography. The private key is used to sign, and the public key is used to verify the signatures. This means that we can, for example, check the public key into source control, and avoid needing any further configuration of the verification part.

We first have to generate the private signing key. This is similar to generating a key for decryption.

```
>>> from nacl.signing import SigningKey
>>> key = SigningKey.generate()
```

We will usually need to store this key (securely) somewhere for repeated use. Again, it is worthwhile remembering that anyone who can access the signing key can sign whatever data they want. For this, we can use encoding:

```
>>> encoded = key.encode()
>>> type(encoded)
<class 'bytes'>
```

The key can be reconstructed from the encoded version. That produces an identical key.

```
>>> key_2 = SigningKey(encoded
>>> key_2 == key
True
```

This is useful in case we want to save the signature in a separate place. For example if the original is in an object storage, mutating it might be undesirable for various reasons. In those cases, we can keep the signatures "on the side." Another reason is t maintain different signatures for different purposes, or to allow key rotation.

If we do want to write the whole signed message, it is best to explicitly convert the result to bytes.

```
>>> encoded = bytes(result)
```

The verification returns back the verified message. This is the best way to use signatures; this way, it is impossible for the code to handle an unverified message.

```
>>> verify_key.verify(encoded)
b'The number you shall count is three'
```

However, if it is necessary to read the object itself from somewhere else, and ther pass it into the verifier, that is also easy to do.

```
>>> verify_key.verify(b'The number you shall count is three',
...                   result.signature)
b'The number you shall count is three'
```

Finally, we can just use the result object as is to verify.

```
>>> verify_key.verify(result)
b'The number you shall count is three'
```

8.3 Passlib

Secure storage of passwords is a delicate matter. The biggest reason it is so subtle is that it has to deal with people who do not use password best practices. If all passwords were strong, and people never reused passwords from site to site, password storage would be straightforward.

However, people usually choose passwords with little entropy (123456 is stil unreasonably popular, as well as password), they have a "standard password" that they use for all websites, and they are often vulnerable to phishing attacks and social engineering attacks where they divulge the password to an unauthorized third party.

The library is storage agnostic: it does not care where the passwords are being stored. However, it does care that it is possible to update the hashed passwords. This way, hashed passwords can get updated to newer hashing schemes as the need arises. While passlib does support various low-level interfaces, it is best to use the high-level interface of the CryptContext. The name is misleading, since it does no encryption; it is a reference to vaguely similar (and largely deprecated) functionality built into Unix.
The first thing to do is to decide on a list of supported hashes. Note that not all of them have to be *good* hashes; if we have supported bad hashes in the past, they still have to be in the list. In this example, we choose argon2 as our preferred hash but allow a few more options.

```
>>> hashes = ["argon2", "pbkdf2_sha256", "md5_crypt", "des_crypt"]
```

Note that `md5` and `des` have serious vulnerabilities and are not suitable to use in real applications. We added them because there might be old hashes using them. In contrast, even though `pbkdf2_sha25` is, probably, worse than argon2, there is no urgent need to update it. We want to mark `md5` and `des` as deprecated.

```
>>> deprecated = ["md5_crypt", "des_crypt"]
```

Finally, after having made the decisions, we build the crypto context:

```
>>> from passlib.context import        CryptContext
>>> ctx = CryptContext(schemes=hashes, deprecated=deprecat
```

It is possible to configure other details, such as the number of rounds. This is almost always unnecessary, as the defaults should be good enough.

Sometimes we will want to keep this information in some configuration (for example, an environment variable or a file) and load it; this way, we can update the list of hashes without modifying the code.

```
>>> serialized = ctx.to_string()
>>> new_ctx = CryptContext.from_string(serialized)
```

When saving the string, note that it does contain newlines; this might impact where it can be saved. If needed, it is always possible to convert it to base64

On user creation or change password, we need to hash the password before storing it. This is done via the `hash` method on the context.

```
>>> res = ctx.hash("good password")
```

enable plain-text communication, was accidentally enabled (or possible to maliciously enable) in production; and furthermore, it was impossible to test that such bugs did not exist, because the testing environment *did* have plain-text communication.

For the same reason, allowing TLS connections without verification in the testing environment is dangerous. This means that the code has a non-verification flow, which can accidentally turn on, or maliciously be turned on, in production, and is impossible to prevent with testing.

Creating a certificate manually requires access to the hazmat layer in cryptography. This is so named because this is dangerous; we have to judiciously choose encryption algorithms and parameters, and the wrong choices can lead to insecure modes.

In order to perform cryptography, we need a "back end." This is because originally it was intended to support multiple back ends. This design is mostly deprecated, but we still need to create it and pass it around.

```
>>> from cryptography.hazmat.backends import
                                                default_backenc
```

Finally, we are ready to generate our private key. For this example, we will use 204ᵹ bits, which is considered "reasonably secure" as of 2019. A complete discussion of whicl sizes provide how much security is beyond the scope of this chapter.

```
>>> from cryptography.hazmat.primitives.asymmetric import          rsa
>>> private_key = rsa.generate_private_key(
...     ...     public_exponent=65537,
... ... )       key_size=2048,
                backend=default_backenc
```

As always in asymmetric cryptography, it is possible (and fast) to calculate the public key from the private key.

```
>>> public_key = private_key.public_key()
```

This is important, since the certificate only refers to the *public* key. Since the private key is never shared, it is not worthwhile, and actively dangerous, to make any assertions about it.

different path: choosing a random serial number. The probability of having the same serial number chosen twice is extremely low.

```
>>> builder = builder.serial_number(x509.random_serial_number())
```

We then add the public key that we generated. This certificate is made of assertions *about* this public key.

```
>>> builder = builder.public_key(public_key)
```

Since this is a CA certificate, we need to mark it as a CA certificate.

```
>>> builder = builder.add_extension(
...     x509.BasicConstraints(ca=True, path_length=None),
...     critical=True)
```

Finally, after we have added all the assertions into the builder, we need to generate the hash and sign it.

```
>>> from cryptography.hazmat.primitives import
                                                    hashes
>>> certificate = builder.sign(
...         private_key=private_key, algorithm=hashes.SHA256(),
...         backend=default_backend()
...     )
```

This is it! We now have a private key, and a self-signed certificate that claims to be a CA. However, we will need to store them in files.

The PEM file format is friendly to simple concatenation. Indeed, usually this is how certificates are stored: in the same file with the private key (since they are useless without it).

```
>>> from cryptography.hazmat.primitives import
                                                    serialization
>>> private_bytes = private_key.private_bytes(
...     encoding=serialization.Encoding.PEM,
...     format=serialization.PrivateFormat.TraditionalOpenSSL,
...     encryption_algorithm=serialization.NoEncrption())
>>> public_bytes = certificate.public_bytes(
...     encoding=serialization.Encoding.PEM)
>>> with  open("ca.pem", "wb")  as  fout:
```

This time, we sign the *service* public key:

```
>>> builder = builder.public_key(public_key)
```

However, we sign *with* the private key of the CA; we do not want *this* certificate to be self-signed.

```
>>> certificate = builder.sign(
...         private_key=private_key, algorithm=hashes.SHA256(),
...         backend=default_backend()
...     )
```

Again, we write a PEM file with the key and the certificate:

```
>>> private_bytes = service_private_key.private_bytes(
...     encoding=serialization.Encoding.PEM,
...     format=serialization.PrivateFormat.TraditionalOpenSSL,
...     encryption_algorithm=serialization.NoEncrption())
>>> public_bytes = certificate.public_bytes(
...     encoding=serialization.Encoding.PEM)
>>> with  open("service.pem", "wb") as fout:
...         fout.write(private_bytes + public_bytes)
```

The service.pem file is in a format that can be used by most popular web servers: Apache, Nginx, HAProxy, and many more. It can also be used directly by the Twisted web server, by using the txsni extension.

If we add the ca.crt file to the trust root, and run, say, an Nginx server, on an IP that our client would resolve from service.test.local, then when we connect clients to https://service.test.local, they will verify that the certificate is indeed valid.

8.5 Summary

Cryptography is a powerful tool but one that is easy to misuse. By using well-understood high-level functions, we reduce many of the risks in using cryptography. While this does not substitute proper risk analysis and modeling, it does make this exercise somewhat easier.

Python has several third-party libraries with well-vetted code, and it is a good idea to use them.

The SSH protocol establishes *mutual trust* – the client is assured that the server is authentic, and the server is assured that the client is authentic. There are several ways it can establish this trust, but in this explanation, we will cover the public key method. This is the most common one.

The server's public key is identified by a *fingerprint*. This fingerprint confirms the server's identity in one of two ways. One way is by being communicated by a previously-established secure channel, and saved in a file.

For example, when an AWS EC2 server boots up, it prints the fingerprint to its virtual console. The contents of the console can be retrieved using an AWS API call (which is secured using the web's TLS model) and parsed to retrieve the fingerprint.

The other way, sadly more popular, is the TOFU model – "Trust On First Use." This means that in the initial connection, the fingerprint is assumed to be authentic and stored in a secure location locally. On any subsequent attempts, the fingerprint will be checked against the stored fingerprint, and a different fingerprint will be marked as an error.

The fingerprint is a hash of the server's public key. If the fingerprints are the same, it means the public keys are the same. A server can provide proof that it knows the private key that corresponds to a given public key. In other words, a server can say "here is my fingerprint" and prove that it is indeed a server with that fingerprint. Therefore, if the fingerprint is confirmed, we have established cryptographic trust with the server.

On the other side, users can indicate to the server which public keys they trust. Again, this is often done via some out-of-band mechanism: a web API for the system administrator to put in a public key, a shared filesystem, or a boot script that reads information from the network. Regardless of how it is done, a user's directory can contain a file that means "please authorize connections which can prove they have a private key corresponding to *this* public key as coming from me."

When an SSH connection is established, the client will verify the server's identity as above, and then will provide proof that it owns a private key that corresponds to some public key on the server. If both of these steps succeed, the connection is verified in both directions and can be used for running commands and modifying files.

9.2 Client Keys

Client private and public keys are kept in files that are next to each other. Often users will already have an existing key, but if not, this is easily remedied.

For example, depending on the cloud service, code such as:

```
set_user_data(machine_id,
f"""
ssh_authorized_keys:
    - ssh-ecdsa {public_key}
""")
```

where set_user_data is implemented using the cloud API, will work on any server that uses cloudinit.
Another thing that is sometimes done is using a Docker container as a bastion.
This means we expect users to SSH into the container, and from the container into the specific machine that they need to run commands on.

In this case, a simple COPY instruction at build time (or a docker cp at runtime, as appropriate) will accomplish the goal. Note that it is perfectly fine to publish to a Docker registry an image with public keys in it – indeed, the requirement that this is a safe operation is part of the definition of public keys.

9.3 Host Identity

As mentioned earlier, the most common, first line of defense against a man-in-the-middle attack in SSH is the so-called TOFU principle – "Trust on First Use." For this to work, after connecting to a host, its fingerprint must be saved in a cache.

The location of that cache used to be straightforward – a file in the user's home directory. However, more modern setups of immutable, throwaway, environments, multiple user machines, and other complications make this more complicated.

It is hard to make a recommendation more general than "share with as many trusted sources as possible." However, to enable that guideline, Paramiko does offer some facilities:

- A client can set a MissingHostKeyPolicy, which is any instance that supports an interface. This means that we can have logic to document the key, or query an external database for it.
- An abstraction of the most common format on UNIX systems, the known_hosts file. This allows Paramiko to share experiences with keys with the regular SSH client – both by reading it and documenting new entries.

The connect method takes quite a few arguments. All of them except the hostname are optional. The more important ones are the following:

- hostname – the server to connect to.
- port is needed if we are running on a special port, other than 22. This is sometimes done as part of a security protocol; attempting a connection to port 22 automatically denies all further connections from the IP, while the real server runs on 5022 or a port that is only discoverable via API.
- username – while the default is the local user, this is true less and less Often cloud virtual machine images have a default "system" user.
- pkey – a private key to use for authentication. This is useful if we want some programmatic way to get the private key (for example, retrieving it from a secret manager).
- allow_agent – this is True by default, for good reasons. This is often a good option, since it means the private key itself will never be loaded by our process, and by extension, cannot be divulged by our process: no accidental logging, debug console, or memory dump is vulnerable.
- look_for_keys – set to False, and give no other key options, to force using an agent.

9.5 Running Commands

The "SH" in SSH stands for shell. The original SSH was invented as a telnet substitute, and its main job is still to run commands on remote machines. Note that "remote" is taken metaphorically, not always literally. SSH is sometimes used to control Virtual Machines, and sometimes even containers, which might be running close by.

After a Paramiko client has connected, it can run commands on the remote host. This is done using the client method exec_command. Note that this method takes the command to be executed *as a string*, not a list. This means that extra care must be exercised when interpolating user values into the command, to make sure that it does not give a user complete execution privileges.

9.7 Remote Files

In order to start file management, we call the client's open_sftp method. This returns an SFTPClient object. We will use methods on this object for all of our remote file manipulation.

Internally, this starts a new SSH channel on the same TCP connection. This means that even while transferring files back and forth, the connection can still be used to send commands to the remote host. SSH does not have a notion of "current directory." Though SFTPClient emulates it, it is better to avoid relying on it and instead use fully qualified paths for all file manipulation. This will make code easier to refactor, and it will not have subtle dependencies on order of operations.

9.7.1 Metadata Management

Sometimes we do not want to change the data, but merely filesystem attributes. The SFTPClient object allows us to do the normal manipulation that

The chmod method corresponds to os.chmod – it takes the same arguments. Sin the second argument to chmod is an integer that is interpreted as a permission bitfield it is best expressed in octal notation. Thus, the best way to set a file to the "regular" permissions (read/write by owner, read to world) is by this:

```
client.chmod("/etc/some_config", 0o644)
```

Note that the `0644` notation, borrowed from C, does not work in Python 3 (and is deprecated in Python 2). The `0o644` notation is more explicit and Pythonic.

Sadly, nothing protects us from passing in nonsense like this:

```
client.chmod("/etc/some_config", 644)
```

(This would correspond to -w----r-- in a directory listing, which is not insecure – but very confusing!)

Some more metadata manipulation methods are these:

- chown
- listdir_iter – used to retrieve file names and metadata
- stat, lstat – used to retrieve file metadata

CHAPTER 10

Salt Stack

Salt belongs to a class of systems called “configuration management systems,” intended to make administrating a large number of machines easier. It does so by applying the same rules to different machines, making sure that any differences in their configuration are intentional.

It is both written in Python and, more importantly, extensible in Python. For example, wherever a YAML file is used, salt will allow a Python file that defines a dictionar y.

10.1 Salt Basics

The salt (or sometimes “SaltStack”) system is a *system configuration management* framework. It is designed to bring operating systems into a specific configuration. It is based on the “convergence loop” concept. When running salt, it does three things:

- Calculates the desired configuration,
- Calculates how the system differs from the desired configuration,
- Issues commands to bring the system to the desired configuration.

Some extensions to Salt go beyond the “operating system” concept to configure some SaaS products into a desired configuration: for example, there is support for Amazon Web Services, PagerDuty, or some DNS services (those supported by *libcloud*).

Since in a typical environment not all operating systems will need to be configured exactly the same way, Salt allows detecting properties of systems and specifying which configurations apply to which systems. At runtime, Salt uses those to decide what is the complete desired state and enforce it.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_1

```
print_server:          # logical name of machi
    user: moshe        # username
    sudo: True         # boolean
    priv: /usr/local/key2   # path to private key
```

In ideal circumstances, all parameters will be identical for the machines. The user is the user to SSH as. The sudo boolean is whether sudo is needed: this is almost always True. The only exception is if we SSH as an administrative user (usually root). Since it is a best practice to avoid SSH as root, this is set to True in most environments.

The priv field is a path to the private key. Alternatively, it can be agent-forwarding to use SSH agent. This is often a good idea, since it presents an extra barrier to key leakage.

The roster can go anywhere, but by default Salt will look for it in /etc/salt/roster. Putting this file in a different location is subtle: salt-ssh will find its configuration, by default, from /etc/salt/master. Since the usual reason to put the roster elsewhere is to avoid touching the /etc/salt directory, that means we usually need to configure an explicit master configuration file using the -c option.

Alternatively, a Saltfile can be used. salt-ssh will look to a Saltfile, in the current directory, for options.

```
salt-ssh:
 config_dir: some/directory
```

If we put in the value . to config_dir , it will look in the current directory for a master file. We can set the roster_file field in the master file to a local path (for example, roster) to make sure the entire configuration is local and locally accessible. This can help if things are being managed by a version control system.

After defining the roster, it is useful to start checking that the Salt system is functioning.

The command

```
$ salt '*' test.ping
```

will send all the machines on the roster (or, later on when we use minions, all connected minions) a ping command. They are all supposed to return True. This command will fail if machines are unreachable, if SSH credentials are wrong, or there are other common configuration problems.

This will cause all connected machines to create a directory /src. More sophisticated commands are possible, and again it is possible to only target specific machines.

The technical term for the desired state in Salt is “highstate.” The name is a frequent cause of confusion, because it seems to be the opposite of a “low state,” which is described almost nowhere. The name “highstate,” however, stands for “high-*level* state”: it describes the goal of state.

The “low” states, the low-*level* states, are the steps that Salt takes to get to the goa Since the compilation of the goal to the low-level states is done internally, nothing in the user-facing documentation talks about a “low” state, thus leading to confusion.

The way to apply the desired state is the following:

```
$ salt '*' state.highstate
```

Since there was so much confusion about the name “highstate,” in an attempt to reduce the confusion, an alias was created:

```
$ salt '*' state.apply
```

Again, both of these do the *exact same thing*: they figure out what the desired state is, for all machines, and then issue commands to reach it.

The state is described in sls files. These files are, usually, in the YAML format and describe the desired state.

The usual way to configure is one file top.sls that describes which other files apply to which machines. The top.sls name is the name that salt will use by default as the top-level file.

A simple homogenous environment might be:

```
# top.sls
base:
'*':
    - core
    - monitoring
    - kubelet
```

This example would have all machines apply the configuration from core.sls (presumably, making sure the basic packages are installed, the right users are configured, etc.); from monitoring.sls (presumably, making sure that tools that monitor the machine are installed and running); and kubelet.sls, defining how to install and configure the kubelet.

A *pillar* is a way of attaching parameters to specific minions, which can then be reused by different states.

If a *pillar* filters out some minions, then these minions are *guaranteed* to never be exposed to the values in the pillar. This means that pillars are ideal for storing secrets, since they will not be sent to the wrong minions.

For better protection of secrets, it is possible to use gpg to encrypt secrets in pillars. Since gpg is based on asymmetric encryption, it is possible to advertise the public key, for example, in the same source control repository that holds the states and pillars.

This means anyone can add secrets to the configuration, but the private key is needed, on the master, to apply those configurations.

Since GPG is flexible, it is possible to target the encryption to several keys. As a best practice, it is best to load the keys into a gpg-agent. This means that when the master needs the secrets, it will use gpg, which will communicate with the gpg-agent.

This means the private keys are never exposed to the Salt master directly.

In general, Salt processes directives in states in order. However, a state can always specify require. When specifying dependencies, it is best to have the dependent state have a custom, readable, name. This makes dependencies more readable.

```
Extract archive:
 archive.extracted:
      - name: /src/some-files
      - source: /src/some-files.tgz
      - archive_format: tar
 - require:
      - file: Copy archive
Copy archive:
 file.managed:
      - name: /src/some-files.tgz
      - source: salt://some-files.tgz
```

Having explicit readable names helped us make sure we depend on the right state. Note that even though Extract precedes Copy, it will still wait for the Copy to be finished.

It is also possible to invert the relationship:

```
Extract archive:
 archive.extracted:
      - name: /src/some-files
```

Some grains, such as fqdn, are auto-detected on the minions. It is also possible to define other grains in the minion configuration file.

It is possible to push grains from the master. It is also possible to grab grains from other sources when bootstrapping the minion. For example, on AWS, it is possible to set the UserData as a grain.

Salt *environments* are directory hierarchies that each define a separate topfile. Minions can be assigned to an environment, or an environment can be selected when applying the highstate using salt '*' state.highstate saltenv=....

The Salt file_roots are a list of directories that function like a path; when looking for a file, Salt will search in them in order, until it finds it. They can be configured on a per-environment basis and are the primary thing distinguishing environments.

10.3 Salt Formats

So far, our example SLS files were YAML files. However, Salt interprets YAML files as *Jinja templates* of YAML files. This is useful when we want to customize fields based on grains or pillars.

For example, the name of the package containing the things we need to build Python packages is different between CentOS and Debian.

The following SLS snippet shows how to target different packages to different machines in a heterogenous environment.

```
{% if grains['os'] == 'CentOs' %}
python-devel: {% elif grains['os'] ==
'Debian' %} python-dev: {% endif %}
pkg:


  - installed
```

It is important to notice that the Jinja processing step is completely ignorant of the YAML formatting. It treats the file as plain text, does the formatting, and then Salt uses the YAML parser on the result.

This means that it is possible for Jinja to make an invalid file only in some cases. Indeed, we embedded such a bug in the example above; if the OS is neither CentOS or Debian, the result would be an incorrectly indented YAML file, which will fail to parse in strange ways.

The process of converting the text into such a data structure is called “rendering” in Salt parlance. This is opposed to common usage, where rendering means transforming *to* text and parsing means transforming *from* text, so it is important to note when reading Salt documentation.

A thing that can do rendering is a renderer. It is possible to write a custom renderer, but among the built-in renderers, the most interesting one is the py renderer.

We indicate that a file should be parsed with the py renderer with #!py at the top.

In that case, the file is interpreted as a Python file. Salt looks for a function run, runs it, and treats the return value as the state.

When running, __grains__ and __pillar__ contain the grain and pillar data.

As an example, we can implement the same logic with a py renderer.

```
#!py
def run():
    if __grains__['os'] == 'CentOS':
        package_name = 'python-dev
    elif __grains__['os'] == 'Debian':
        package_name = 'python-de
    else:
        raise ValueError("Unrecognized operating system",
                         __grains__['os'])
return { package_name: dict(pkg='installed') }
```

Since the pyrenderer is not a combination of two unrelated parsers, mistakes end up being sometimes easier to diagnose. If we reintroduce the bug from the first version, we get:

```
#!py
def run():

    if __grains__['os'] == 'CentOS':
        package_name = 'python-dev
    elif __grains__['os'] == 'Debian':
        package_name = 'python-de
return { package_name: dict(pkg='installed') }
```

In this case, the result will be aNameErrpinpointing the erroneous line and the missing name.

Since we want to progress one thing at a time, we are using a prewritten Salt executor: the file module, which has the find function.

```
def enforce_no_mean_files(name
    # ...continued...
    if mean_files = []:
        return dict(
            name=name,
            result=True,
            comment='No mean files detectec
            changes=[],
        )
    # ...continues below...
```

One of the things the state module is responsible for, and indeed often the most important thing, is *doing nothing* if the state is already achieved. This is what being a convergence loop is all about: optimizing for the case of having already achieved convergence.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
        old=mean_files,
        new=[],
    )
    # ...continues below...
```

We now know what the changes are going to be. Calculating it here means we can guarantee consistency between the responses in the test vs. non-test mode.

```
def enforce_no_mean_files(name):
    # ...continued...
    changes = dict(
    if __opts__['test']:
    return dict(
            name=name
            result=None,
```

10.4.2 Execution

For historical reasons, execution modules go in the file root's _modules subdirectory. Similarly to execution modules, they are also synchronized when state.highstate is applied, as well as when explicitly synchronized via saltutil.sync_all.

As an example, we write an execution module to delete several files, in order to simplify our state module above.

```
def multiremove(files): for
fname in files:

    __salt__['file.remove'](fname)
```

Note that __salt__ is usable in execution modules as well. However, while it can cross-call other execution modules (in this example, file) it cannot cross-call into state modules.

We put this code in _modules/multifile, and we can change our state module to have:

```
__salt__['multifile.mutiremove'](mean_files)
```

instead of

```
for fname in mean_files:

    __salt__['file.remove'](fname)
```

Execution modules are often simpler than state modules, as in this example. In this toy example, the execution module barely does anything at all, just coordinating calls to other execution modules.

This is not completely atypical, however. Salt has so much logic for managing machines that often all an execution module has to do is just to coordinate calls to other execution modules.

10.4.3 Utility

When writing several execution or state modules, it sometimes is the case that there is some common code that can be factored out.

This code can sit in so-called "utility modules." Utility modules sit under the file root's _utils directory and will be available as the __utils__ dictionary.

10.4.4 Extra Third-Party Dependencies

Sometimes it is useful to have third-party dependencies, especially when writing new state and execution modules. This is straightforward to do when installing a minion; we just make sure to install the minion in a virtual environment with those third-party dependencies.

When using Salt with SSH, this is significantly less trivial. In that case, it is sometimes best to bootstrap from SSH to a real minion. One way to achieve that is to have a persistent state in the SSH "minion" directory, and have the installation of the minion set a grain of "completely_disable" in the SSH minion. This would make sure that the SSH configuration does not cross-talk with the regular minion configuration.

10.5 Summary

Salt is a Python-based configuration management system. For nontrivial configurations, it is possible to *express* the desired system configuration using Python, which can sometimes be more efficient than templating YAML files. It is also possible to *extend* it with Python in order to define new primitives.

We can also give an explicit address:

```
$ ansible 10.40.32.195 -m ping
```

will try to SSH to 10.40.42.195 .

The set of hosts Ansible will try to access by default is called the "inventory." The inventory can either be specified statically in an INI or YAML files. However, the more common option is to write an "inventory script," which generates the list of

An inventory script is simply a Python file that can be run with the arguments --list and –host <hostname>. Ansible will use, by default, the same Python used to run it in order to run the inventory script. It is possible to make the inventory script a "real script" running with any interpreter, like a different version of Python, by adding a shebang line. Traditionally, the file is not named with .py. Among other things, this avoids accidental imports of the file.

When run with --list, it is supposed to output the inventory as formatted JSON. When run with --host, it is supposed to print the variables for the host. In general, it is perfectly acceptable to always print an empty dictionary in these circumstances.

Here is a simple inventory script:

```
import sys
if  '--host' in sys.argv[1:]:
      print(json.dumps({}))

print(json.dumps(dict(all='localhost')))
```

This inventory script is not very dynamic; it will always print the same thing. However, it is a valid inventory script.

We can use it with

```
$ ansible -i simple.inv all -m ping
```

This will again ping (using SSH) the localhost.

Ansible is not primarily used to run ad hoc commands against hosts. It is designed to run "playbooks." Playbooks are YAML files that describe "tasks.

```
---
- hosts: all
 tasks:

      - name: hello printer
        shell: echo "hello world"
```

Then, under roles/common/tasks/mair

```
---
- hosts: all
 tasks:
      - name: hello printer
        shell: echo "hello world" >> /etc/hello
        creates: /etc/hello
```

This will do the same thing as above, but now it is indirected through more files. The benefit is that if we have many different hosts, and we need to combine instructions for some of them, this is a convenient platform to define parts of more complicated

11.2 Ansible Concepts

When Ansible needs to use secrets, it has its internal "vault." The vault has encryptec secrets and is decrypted with a password. Sometimes this password will be in a file (ideally on an encrypted volume).

Ansible roles and playbooks are *jinja2* YAML files. This means they can use interpolation, and they support a number of Jinja2 filters.

Some useful ones are from/to_json/yaml, which allow data to be parsed and serialized back and forth. The map filter is a meta-filter that applies a filter item by item tc an iterable object.

Inside the filters, there is a set of variables defined. Variables can come from multiple sources: the Vault (for secrets), directly in the playbook or role, or in files included from it. Variables can also come from the inventory (which can be useful if different inventories are used with the same playbook). The ansible_facts variable is a dictionary that has the facts about the current host: operating system, IP, and more.

They can also be defined directly on the command line. While this is dangerous, it can be useful for quick iterations.

In playbooks, it is often the case that we will need to define both which user to *log in* as, as well as which user (usually root) to execute tasks as

All of those can be configured on a playbook and overridden on a per-task level.

The user that we *log in* as is remote_user. The user that we execute as is either remote_user if become is False, or become_user if become is True. If becor user switching will be done by become_method.

11.3 Ansible Extensions

We have seen one way to extend ansible using custom Python code: dynamic inventory. In the dynamic inventory example, we wrote an ad hoc script. The script, however, was run as a separate process. A better way to extend Ansible, and one that generalizes beyond inventory, is to use *plugins*.

An *inventory plugin* is a Python file. There are several places for this file so that Ansible can find it: often the easiest is plugins/inventory_plugins in the same directory as the playbook and roles.

This file should define a class called InventoryModule that inherits from BaseInventoryPlugin. The class should define two methods: verify_file and parse. The verify_file function is mostly an optimization; it is meant to quickly skip the parsing if the file is not the right one for the plugin. It is an optimization since parse can (and should) raise AnsibleParserError if the file cannot be parsed for any reason. Ansible will then try the other inventory plugins.

The parsefunction signature is

```
def parse(self, inventory, loader, path, cache=    True):
    pass
```

A simple example parsing JSON:

```
def parse(self, inventory, loader, path, cache=True):
    super(InventoryModule, self).parse(inventory, loader, path, cache)
    try:
        with open(path) as fpin:
            data = json.loads(fpin.read())
    except ValueError as exc:
        raise AnsibleParseError(exc)
    for host in data:
        self.inventory.add_host(server['name'])
```

The inventoryobject is how to manage the inventory; it has methods foradd_grou add_childand set_variable, which is how the inventory is extended.

The loaderis a flexible loader that can guess a file's format and load it. Thepathis the path to the file that has the plugin parameters. Notice that in some cases, if the plugin is specific enough, the parameters and the loader might not be needed.

CHAPTER 12

Docker

Docker is a system for application-level virtualization. While different Docker *containers* share a *kernel*, they will usually share little else: files, processes, and more can all be separate. It is commonly used for both testing software systems and running them in production.

There are two main ways to automate Docker. It is possible to use the subprocess library and use the docker command line. This is a popular way and does have some advantages.

However, an alternative is to use the dockerpy library. This allows doing some things that are completely impossible with the docker command, as well as some things that are merely impossible or annoying with the command.

One of the advantages is installation; getting DockerPy installed is just pip install docker in a virtual environment, or any of the other ways to install Python packages. Installing the Docker binary client is often more involved. While it does come when installing the Docker daemon, it is not uncommon to need just a client, while the server runs on a different host.

When using the docker Python package, the usual way is to connect using

```
import docker
client = docker.from_env()
```

This will, by default, connect to the local Docker daemon. However, in an environment that has been prepared using docker-machine env, for example, it will connect to the relevant remote Docker daemon.

In general, from_env will use an algorithm that is compatible with the one the docker command-line client uses, and so is often useful in a drop-in replacement.

For example, this is useful in Continuous Integration environments that allocate a Docker host per CI session. Since they will set up the local environment to be compatible with the docker command, from_env will do the right thing.

M. Zadka, *DevOps in Python*, https://doi.org/10.1007/978-1-4842-4433-3_

This can come in handy, for example, when creating an image from wheels; we can download the wheels into in-memory buffers, create the container, tag it, and push it, all without needing any temporary files.

12.2 Running

The containers attribute on the client allows managing running containers.

The containers.run() method will run a container. The arguments are much the same as the docker run command line, but there are some differences in the best way to use them.

It is almost always a good idea, from Python, to use the detach=True option. This will cause run() to return a Container object. If you need to wait until it exits, for some reason, call .wait, explicitly, on the container object.

This allows timeouts, which are useful for killing runaway processes. The return value of the container object also allows retrieving the logs, or inspecting the list of processes inside the container.

The containers.create method will create a container, but not run it, like docker create.

Regardless of whether a container is running or not, it is possible to interact with its file system. The get_archive method will retrieve a file or recursively a directory from the container. It will return a tuple. The first element is an iterator that yields raw bytes objects. Those can be parsed as a tar archive. The second element is a dictionary containing metadata about the file or the directory.

The put_archive command injects files into the container. This is sometimes useful between create and start to fine-tune the container: for example, injecting a configuration file for a server.

It is even possible to use this as an alternative to the build command; a combination of container.put_archive and container.commit with containers.run and containers.create allows building containers incrementally, without a Dockerfile. One advantage of this approach is that the layer division is orthogonal to the number of steps: you can have several logical steps be the same layer.

Note, however, that deciding which “layers” to cache becomes our responsibility, in this case. Also, in this case, the “intermediate layers” are fully fledged images. This has its advantages: for example, cleaning up becomes more straightforward.

CHAPTER 13

Amazon Web Services

Amazon Web Services, AWS, is a cloud platform. It allows using computation and storage resources in a data center, paying by usage. One of the central principles of AWS is that all interactions with it should be possible via an API: the web console, where computation resources can be manipulated, is just another front end to the API. This allows automating configuration of the infrastructure: so-called "infrastructure as code," where the computing infrastructure is reserved and manipulated programmatically.

The Amazon Web Services team supports a package on PyPI, boto3, to automate AWS operations. In general, this is one of the best ways to interact with AWS

While AWS does support a console UI, it is usually best to use that as a read-only window into AWS services. When making changes through the console UI, there is no repeatable record of it. While it is possible to log actions, this does not help to reproduce them.

Combining boto3 with Jupyter, as we have discussed in an early chapter, makes for a powerful AWS operations console. Actions taken through Jupyter, using the boto3 API, can be repeated, automated, and parameterized as needed.

When making ad hoc changes to the AWS setup to solve a problem, it is possible to attach the notebook to the ticket tracking the problems, so that there is a clear record of what was done to address the problem. This serves both to understand what was done in case this caused some unforeseen issues, and to easily repeat this intervention in case this solution is needed again.

As always, notebooks are not an *auditing* solution; for one, when allowing access via boto3, actions do not have to be performed via a notebook. AWS has internal ways to generate *audit* logs. The notebooks are there to document intent and allow repeatability.

M. Zadka, *DevOps in Python* , https://doi.org/10.1007/978-1-4842-4433-3_

13.1.2 Creating Short-Term Tokens

AWS supports something called “Short-Term Tokens” or STS. Short-term tokens can be used for several things. They can be used to convert alternative authentication methods into tokens that can be used with any boto3-based program, for example, by putting them in an environment variable.

For example, in an account that has been configured with SSO-based authentication based on SAML, boto3.client('sts').assume_role_with_saml can be called to generate a short-term security token. This can be used in boto3.Session in order to get a session that has those permissions.

```
import boto3
response = boto3.client('sts').assume_role_with_saml(

    RoleArn=role_arn,
    PrincipalArn=principle_arn,
    SAMLAssertion=saml_assertior
    DurationSeconds=120
) credentials = response['Credentials']
session = boto3.Session(

    aws_access_key_id=credentials['AccessKeyId'],
    aws_secret_access_key=credentials['SecretAccessKey'],
    aws_session_token=credentials['SessionToken'],
) print(session.client('ec2').describe_instances())
```

A more realistic use case would be in a custom web portal that is authenticated to an SSO portal. It can perform actions on behalf of the user, without *itself* having any special access privileges to AWS.

On an account that has been configured with cross-account access, assume_toker can return credentials for the granting account.

Even when using a single account, sometimes it is useful to create a short-term token. For example, this can be used to limit permissions; it is possible to create an STS with a limited security policy. Using those limiting tokens in a piece of code that is more prone to vulnerabilities, for example, because of direct user interactions, allows limiting the attack surface.

13.2.2 Amazon Machine Images

In order to start an EC2 machine, we need an "operating system image." While it is possible to build custom Amazon Machine Images (AMIs), it is often the case we can use a ready-made one.

There are AMIs for all major Linux distributions. The AMI ID for the right distribution depends on the AWS *region* in which we want to run the machine. Once we have decided on the region and on the distribution version, we need to find the AMI ID.

The ID can sometimes be nontrivial to find. If you have the *product code*, for example, aw0evgkw8e5c1q413zgy5pjce we can use describe_image

```
client = boto3.client(region_name='us-west-2')
description = client.describe_images(Filters=[{
    'Name': 'product-code',
    'Values': ['aw0evgkw8e5c1q413zgy5pjce']
}])
print(description)
```

The CentOS wiki contains product codes for all relevant CentOS versions.

AMI IDs for Debian images can be found on the Debian wiki. The Ubuntu website has a tool to find the AMI IDs for various Ubuntu images, based on region and version. Unfortunately, there is no centralized automated registry. It is possible to search for AMIs with the UI, but this is risky; the best way to guarantee the authenticity of the AMI is to look at the creator's website.

13.2.3 SSH Keys

For ad hoc administration and troubleshooting, it is useful to be able to SSH into the EC2 machine. This might be for manual SSH, using Paramiko, Ansible, or bootstrapping Salt. Best practices for building AMIs that are followed by all major distributions for default images use cloud-init to initialize the machine. One of the things cloud-init will do is allow a preconfigured user to log in via an SSH public key that is retrieved from the so-called "user data" of the machine.

Public SSH keys are stored by region and account. There are two ways to add an SSH key: letting AWS generate a key pair, and retrieving the private key, or generating a key pair ourselves and pushing the public key to AWS.

```
        InstanceType='t2.micro',
        KeyName=ssh_key_name,
        SecurityGroupIds=['sg-03eb2567']
)
```

The API is a little counterintuitive – in almost all cases, both MinCount and MaxCount need to be 1. For running several identical machines, it is much better to use an AutoScaling Group (ASG), which is beyond the scope of the current chapter. In general, it is worth remembering that as AWS's first service, EC2 has the oldest API, with the least lessons learned on designing good cloud automation APIs.

While in general the API allows running more than one instance, this is not often done. The SecurityGroupIds imply which VPC the machine is in. When running a machine from the AWS console, a fairly liberal security group is automatically created. For debugging purposes, using this security group is a useful shortcut, although in general it is better to create custom security groups.

The AMI chosen here is a CentOS AMI. While KeyName is not mandatory, it is highly recommended to create a key pair, or import one, and use the name.

The InstanceType indicates the amounts of computation resources allocated to the instance. t2.micro is, as the name implies, is a fairly minimal machine. It is useful mainly for prototyping but usually cannot support all but the most minimal production workloads.

13.2.5 Securely Logging In

When logging in via SSH, it is a good idea to know beforehand what is the public key we expect. Otherwise, an intermediary can hijack the connection. Especially in cloud environments, the "Trust-on-First-Use" approach is problematic; there are a lot of "first uses" whenever we create a new machine. Since VMs are best treated as disposable, the TOFU principle is of little help.

The main technique in retrieving the key is to realize that the key is written to the "console" as the instance boots up. AWS has a way for us to retrieve the console output

```
client = boto3.client('ec2')
output = client.get_console_output(InstanceId=sys.argv[1])
result = output['Output']
```

One way to prepare machines is to use a configuration management system. Both Ansible and Salt have a "local" mode that runs commands locally, instead of via a server/client connection.

The steps are:

- Launching an EC2 machine with the right base image (for example, vanilla CentOS).
- Retrieve the host key for securely connecting.
- Copy over Salt code.
- Copy over Salt configuration.
- Via SSH, run Salt on the EC2 machine.
- At the end, call client("ec2").create_image in order to save the current disk contents as an AMI.

```
$ pex -o salt-call -c salt-call salt-ssh
$ scp -r salt-call salt-files $USER@$IP:/
$ ssh $USER@$IP /salt-call --local --file-root /salt-files
(botovenv)$ python
...
>>> client.create_image(....)
```

This approach means a simple script, running on a local machine or in a CI environment, can generate an AMI from source code.

13.3 Simple Storage Service (S3)

The simple storage service (S3) is an object storage service. Objects, which are byte streams, can be stored and retrieved. This can be used to store backups, compressed log files, video files, and similar things.

S3 stores objects in *buckets*, by *key* (a string). Objects can be stored, retrieved, or deleted. However, objects cannot be modified in place.

S3 buckets names must be globally unique, not just per account. This uniqueness is often accomplished by adding the account holder's domain name, for example, large- videos.production.example.com.

When a there are more objects that qualify than MaxKeys, the response will be truncated. In that case, the IsTruncated field in the response will be True, and the NextMarker field will be set. Sending another list_objects with the Marker set to the returned NextMarker will retrieve the next MaxKeys objects. This allows pagination through responses that are consistent even in the face of mutating buckets, in the limited sense that we will get at least all *objects* that were not mutated while paginating.

In order to retrieve a single object, we use get_object:

```
response = boto3.client("s3").get_object(
    Bucket='string',
    Key='string',
)
value = response["Body"].read()
```

The value will be a *bytes* object.

Especially for small- to medium-sized objects, say up to several megabytes, this is a way to allow simple retrieval of all data.

In order to push such objects into the bucket we can use this:

```
response = boto3.client("s3").put_object(
    Bucket=BUCKET,
    Key=some_key,
    Body=b'some content
)
```

Again, this works well for the case where the body all fits in memory.

As we have alluded to earlier, when uploading or downloading larger files (for example, videos or database dumps) we would like to be able to upload incrementally, without keeping the whole file in memory at once.

The boto3 library exposes a high-level interface to such functionality using the *_fileobj methods.

For example, we can transfer a large video file using:

```
client = boto3.client('s3')
with open("meeting-recording.mp4", "rb") as fpin:
    client.upload_fileobj(
        fpin,
```

An even more interesting use case is allowing pre-signed *uploads*. This is especially interesting because uploading files sometimes requires subtle interplays between the web server and the web application server to allow large requests to be sent in. Instead, uploading directly from the client to S3 allows us to remove all the intermediaries. For example, this is useful for users who are using some document sharing applications.

```
post = boto3.client("s3").generate_presigned_post(
    Bucket=my_bucket,
    Key='meeting-recording.avi',
)  post_url  =  post["url"]
post_fields = post["fields"]
```

We can use this URL from code with something like:

```
with open("meeting-recording.avi", "rb"):
    requests.post(post_url,
                  post_fields,
                  files=dict(file=file_contents))
```

This lets us upload the meeting recording locally, even if the meeting recording device does not have S3 access credentials. It is also possible to limit the maximum size of the files via generate_presigned_post, to limit the potential harm from an unknown device uploading these files.

Note that pre-signed URLs can be used multiple times. It is possible to make a pre-signed URL only valid for a limited time, to mitigate any risk of potentially mutating the object after uploading. For example, if the duration is one second, we can avoid checking the uploaded object until the second is done.

13.4 Summary

AWS is a popular Infrastructure-as-a-Service platform, which in general is used on a pay-as-you-go basis. It is suitable to automation of infrastructure management tasks, and boto3, maintained by AWS itself, is a powerful way to approach this automation.

D

E

F

G

H

I

J, K, L

R

S