00001001

CHAPTER 9

Project Organization

Learning Objectives

- *Create a project directory structure suitable for Python development*
- *State the purpose of the src directory*
- *State the purpose of the docs directory*
- *State the purpose of the tests directory*
- *List and describe the types of files found in the src, tests, and docs directories*
- *State the purpose of the .gitignore file*
- *Add entries to the .gitignore file*
- *State the purpose of the README.md File*
- *Use Markdown to create project documentation*
- *Reference the Markdown Cheat Sheet to help build project documentation*
- *State the purpose of the main.py module*
- *Run the main.py module from the project root directory*

© 2023 Pulp Free Press

Introduction

In this short, but critically important chapter, you will learn a baseline directory structure designed to help you impose order upon your projects. It places source code, test, and documentation files in separate directories while keeping the project's root directory clear for project control and configuration files. In later chapters, you will build upon the baseline directory structure and add subdirectories, and control and configuration files as required to suit a project's particular needs.

Why is this even a stand-alone chapter? As simple as the concept of *project organization* may seem at first blush, the most confounding question a novice programmer seeks to answer when they first learn how to program, especially in Python, is "*Where do I put my source code files?*" Naturally, their first instinct, if not provided with clear, understandable guidance, is to ask Google and YouTube, where they will find plenty of answers, some right, some less so, many dated and based on older versions of Python, and most not flexible enough to meet the demands of modern programming projects in which Python may play only a supporting role. Faced with conflicting recommendations and not knowing any better, they place all their source files in the root project directory, which makes things easy in the short term but a complete mess of things in the long term. Reorganizing a project is way more of a pain in the booty than starting out right the first time around.

Besides a flexible, workable, extensible project organization structure, you'll add a few more tools to your programmer's tool belt. These include how to run Python files located in the *src* directory from the project's *root* directory, how to reference modules located in the *src* directory from unit tests located in the *tests* directory, how to use Markdown to create project documentation located in a *README.md* file, and how to add entries to the *.gitignore* file.

Perhaps the best motivation I can offer you is that the project organization structure you learn here can be used for many other programming languages besides Python. Later in the book, you'll add on to the baseline structure to support different types of applications. For example, if your application uses a database, you'll want to add a *database* folder to store the database creation scripts.

1 Baseline Project Structure

For context, Figure 9-1 gives the basic *repository layout* suggested in chapter 8.

Referring to figure 9-1 — Each of the repository's project folders, (i. e., Project_1, Project_2, etc.), would contain the project structure, or some form of it, discussed in this section. Figure 9-2 gives a graphical view of the *baseline project organization structure*.

Referring to figure 9-2 — The baseline project organization structure consists of a *README.md* file, an optional *.gitignore* file, and three subdirectories: *src*, *tests*, and *docs*. The following sections discuss the contents of each of these directories in greater detail.

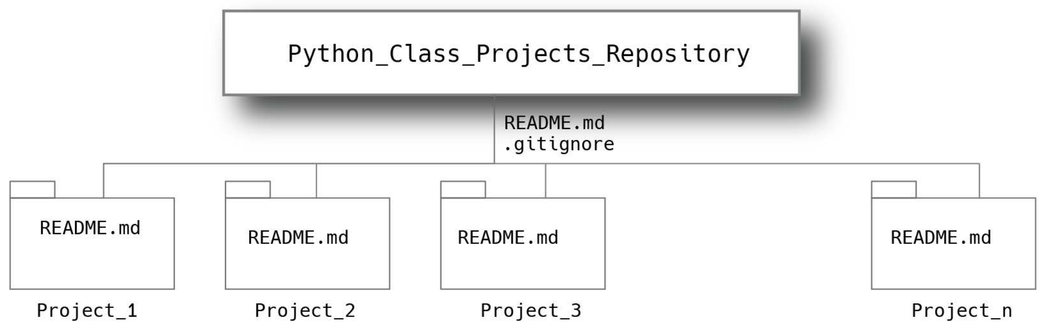


Figure 9-1: Suggested Class Projects Repository Structure

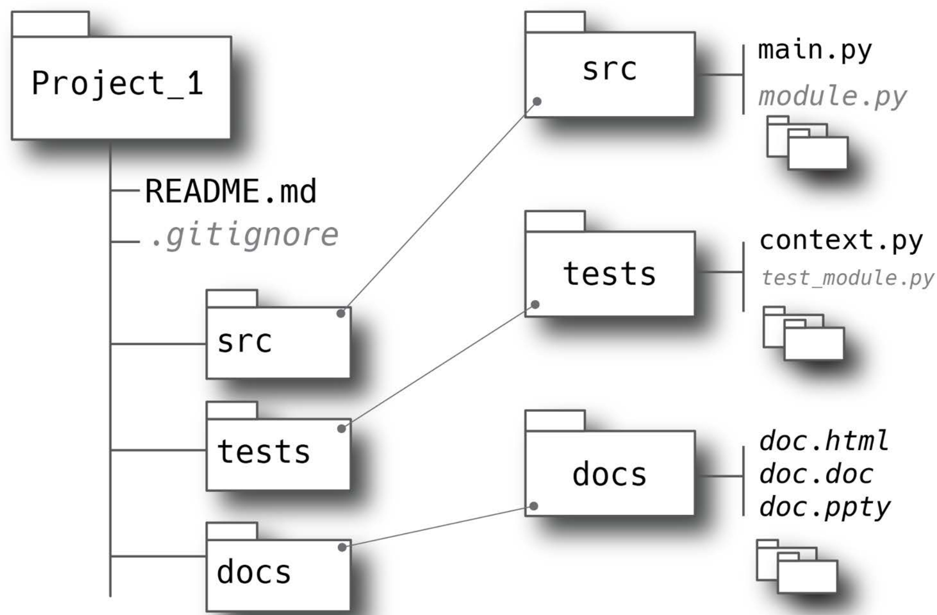


Figure 9-2: Project Organization Structure (Baseline)

1.1 SRC Directory

The project's *src* directory contains source code files and module folders. For Python projects these may be all Python files, but the *src* directory could contain a combination of programming language files. For the purposes of this book, I'll stick to Python source files and modules.

For Python applications, I add a *main.py* module, which serves as the program main entry point. The *main.py* module imports modules required for the particular application, be they located in the *src* directory or third-party packages installed with `pip`/`pip3`. I'll talk more about installing packages in *Chapter 10: Virtual Environments with Pipenv*.

1.2 Tests Directory

The project's *tests* directory contains unit and integration test files. These are ordinary Python files that start with the string `test`. For example, referring to figure 9-2, if, in the *src* directory,

 © 2023 Pulp Free Press

00001001

you had a Python file named *module.py*, then, if you wanted to test that code, in the *tests* directory you would have a corresponding file named *test_module.py*. The testing framework employed dictates how to name test modules, classes, and methods. I talk more about unit testing in *Chapter 21: Unit Testing*, after you've been formally introduced to classes and methods.

1.3 Docs Directory

Use the project's *docs* directory to store project documentation. This includes automatically-generated documentation created by running a document generator tool like *Sphinx*. It can also include any number of file types including Microsoft Word documents, Microsoft PowerPoint slides, web pages, you name it. This does not replace the project's *README.md* file, which is used to describe the project and offer general usage or application installation instructions. (**Note:** *You can include anything you want in a README.md file. I've listed just a few of the common uses for a README.md file here.*) You can also store documentation image files in the docs directory as well.

1.4 The Project README.md and .gitignore Files

At the root of the project directory you'll have a *README.md* file and an optional *.gitignore* file.

1.4.1 README.md File

As mentioned above, the purpose of the *README.md* file is to describe the purpose of the project and provide instructions on how to run the application. GitHub will automatically load and render a *README.md* file on the project's repository page. To see examples of the many different and creative ways people use *README.md* files just poke around on GitHub. I talk more about how to add content to a *README.md* file with the *Markdown* language later in the chapter.

1.4.2 .gitignore File

As discussed in *Chapter 8: Source Code Management with Git and GitHub*, the purpose of a *.gitignore* file is to hold a list of project files and directories you **do not** want tracked or pushed to the repository. Files and directories listed in the *.gitignore* file are ignored by git when adding and committing files. Files listed in the *.gitignore* file remain untracked. I talk more about the *.gitignore* file in the following section.

A small to medium-sized repository may only need one *.gitignore* file located at the root of the repository directory as shown in figure 9-1.You can add *.gitignore* files to project subdirectories as required to tweak the ignore list.

Quick Review

The baseline project organizational structure includes a *README.md* file, an optional *.gitignore* file, and three directories: *src*, *tests*, and *docs*. Store application source files in the *src* directory, which can have any number of files and subdirectories. Use the *tests* directory to store unit and integration test files. These, too, can be organized into subdirectories. Test filenames usu-

ally begin with the string `test`. Store all project documentation with the exception of the README.md file in the *docs* directory. This includes automatically generated documentation.

2 Configuring .gitignore File

There are several ways you can add a *.gitignore* file to your project. You saw in the previous chapter that when you create a repository, you could select a template which would pre-populate the repository's *.gitignore* file with typical files and directories to ignore depending on what type of programming language used. You can also create one from scratch, or flex your Google dorks and find a good example on the Internet. In this brief section, I'll discuss each of these approaches, but before I get started, I'd like to reemphasize an important point to always keep in mind.

2.1 Think Before You Commit

Yes, back to the topic of *mindfulness* and a brief recap of the previous chapter. Configure your *.gitignore* file before you do a `git add` and `git commit`. The *.gitignore* file automatically created and populated for you when you created your GitHub repository will cover the typical things you want to omit from the repository, but not everything. Stay alert for configuration files and directories, misplaced SSH keys, along with passwords stored in the clear or in any form. The business of security delivers death by a thousand cuts. Think before you commit.

2.2 Roll Your Own

To create a *.gitignore* file from scratch, navigate to you project directory and create the file with the following command:

```
touch .gitignore
```

Alternatively, you can create it with your text editor of choice:

```
nano .gitignore
```

...or...

```
vi .gitignore
```

The previous two forms will launch the respective editor and allow you to start editing the new file. Add whatever it is you want to ignore then save and close the file. As you work on your project, you may add files and directories you want to ignore. Simply add them to the *.gitignore* file before you use `git add` and `git commit`.You can always verify the *.gitignore* file entries are indeed heeded by running `git status` and reviewing the output. The complete *.gitignore* documentation is located on the Git-SCM website: *https://git-scm.com/docs/gitignore*

2.3 Copy a Pre-Existing Template

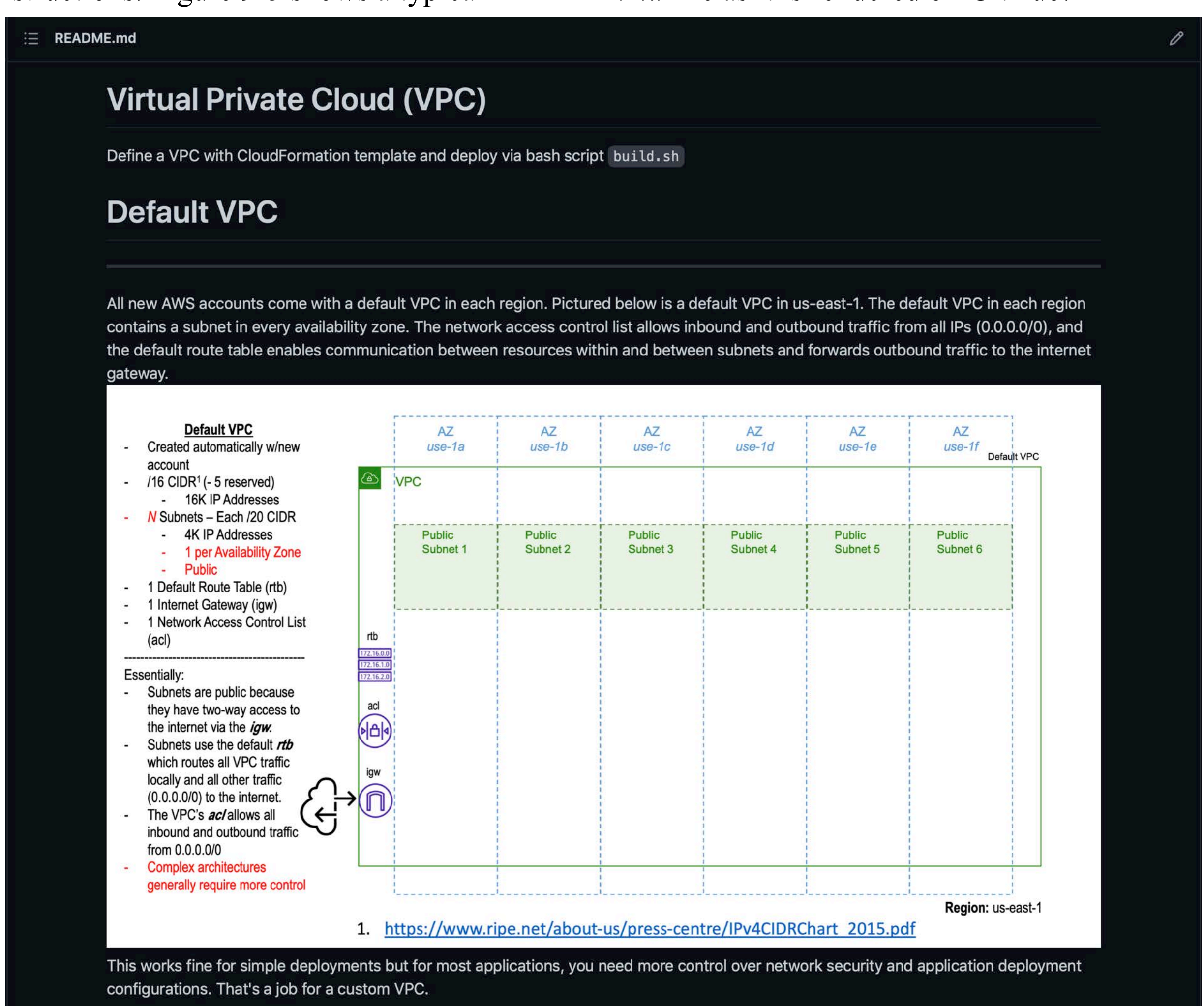
You can copy an existing template, either a good example you found on some random website that matches your particular project, or one from GitHub. The GitHub repository *.gitignore* template files are located here: *https://github.com/github/gitignore* You can create your *.gitignore* file as described above, then copy the contents of a suitable template file and paste it into yours.

Quick Review

The purpose of the *.gitignore* file is to maintain a list of files and directories you do not want to track or push to your repository. You can either create a *.gitignore* file from scratch and add ignored items as required, or start from a template. You can search for suitable *.gitignore* file examples online or use a GitHub template.

3 Documenting Your Project With README.md

At the most basic level, use the *README.md* file to provide a project description and usage instructions. Figure 9-3 shows a typical *README.md* file as it is rendered on GitHub.



README.md

Virtual Private Cloud (VPC)

Define a VPC with CloudFormation template and deploy via bash script `build.sh`

Default VPC

All new AWS accounts come with a default VPC in each region. Pictured below is a default VPC in us-east-1. The default VPC in each region contains a subnet in every availability zone. The network access control list allows inbound and outbound traffic from all IPs (0.0.0.0/0), and the default route table enables communication between resources within and between subnets and forwards outbound traffic to the internet gateway.

This works fine for simple deployments but for most applications, you need more control over network security and application deployment configurations. That's a job for a custom VPC.

Figure 9-3: Typical README.md File — Screenshot Taken from GitHub

Referring to figure 9-3 — This is only a partial view of this project's *README.md* file. You can see the entire file at the following repository: *https://github.com/pulpfreepress/it-590-aws/tree/main/vpc* This serves as documentation for one particular project, namely, the Virtual Private Cloud (VPC) project, which is one of many projects belonging to this repository. Figure 9-4 shows the repository's main *README.md* file.

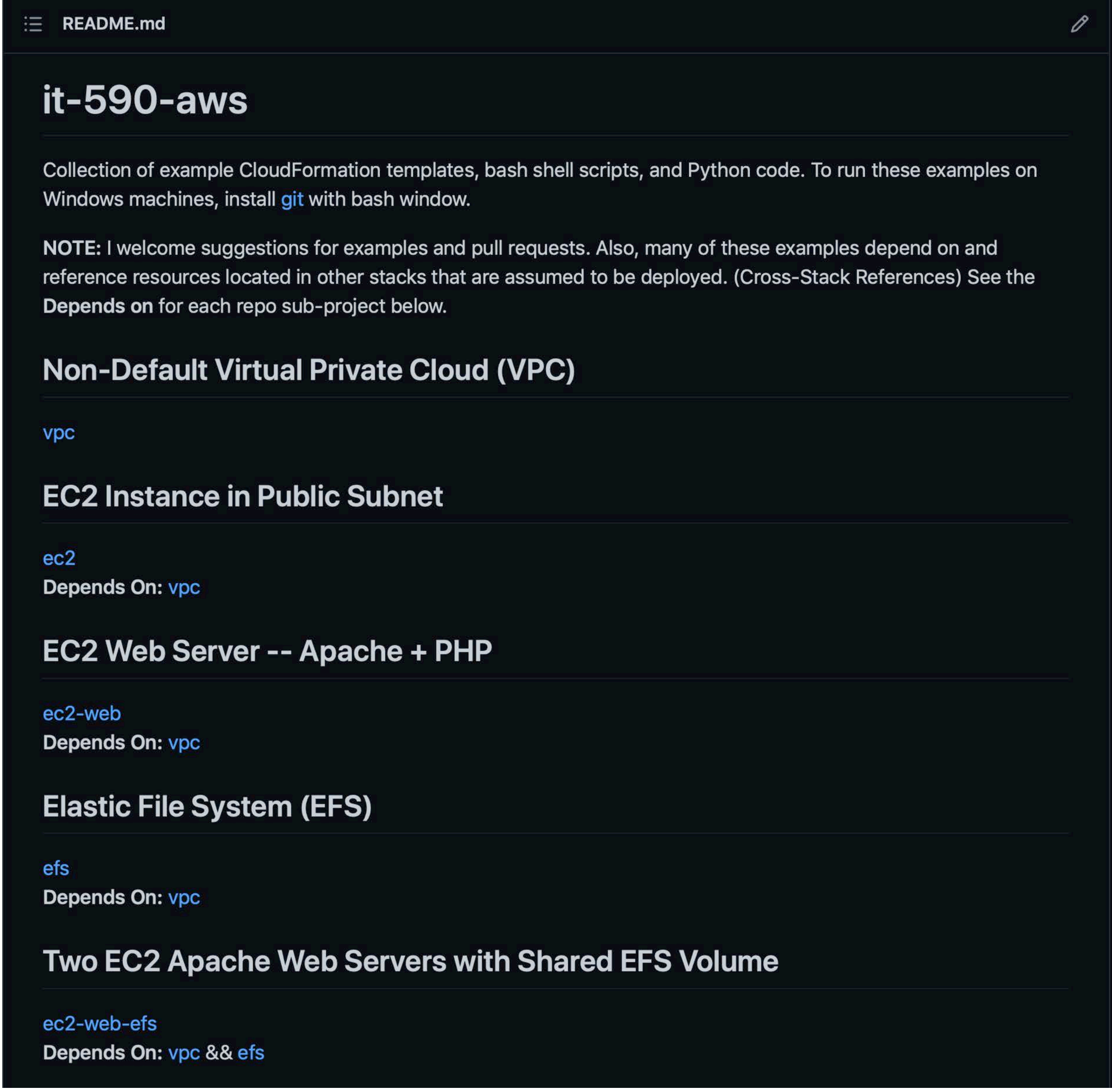


Figure 9-4: Main Repository README.md File with Hyperlinks to Subprojects — Partial View

Referring to figure 9-4 — This is the main repository *README.md*. It contains links to each of the subprojects. Example 9.1 gives the Markdown source code for this *README.md* file.

9.1 README.md Markdown Source
for repository: https://github.com/pulpfreepress/it-590-aws

```
1    # it-590-aws
2
3    Collection of example CloudFormation templates, bash shell scripts, and Python
code. To run these examples on Windows machines, install <a href="https://git-scm.com/
downloads">git</a> with bash window.
4
5    **NOTE:** I welcome suggestions for examples and pull requests.
6    Also, many of these examples depend on and reference resources located in other
stacks that are assumed to be deployed. (Cross-Stack References) See the **Depends on**
for each repo sub-project below.
```

```
7
8
9    ## Non-Default Virtual Private Cloud (VPC)
10   <a href="vpc/">vpc</a>
11
12
13   ## EC2 Instance in Public Subnet
14
15   <a href="ec2/">ec2</a></br>
16   **Depends On:** <a href="vpc/">vpc</a>
17
18   ## EC2 Web Server -- Apache + PHP
19
20   <a href="ec2-web">ec2-web</a></br>
21   **Depends On:** <a href="vpc/">vpc</a>
22
23   ## Elastic File System (EFS)
24
25   <a href="efs">efs</a></br>
26   **Depends On:** <a href="vpc/">vpc</a>
27
28   ## Two EC2 Apache Web Servers with Shared EFS Volume
29
30   <a href="ec2-web-efs/">ec2-web-efs</a></br>
31   **Depends On:** <a href="vpc/">vpc</a> && <a href="efs">efs</a></br>
32
33   ## Lambda Echo Server
34
35   <a href="lambda-echo/">lambda-echo</a>
36
37   ## Lambda Echo Server with Custom RestAPI
38
39   <a href="lambda-echo-custom-api/">lambda-echo-custom-api</a>
40
41
42   ## Simple Notification Service (SNS)
43
44   <a href="sns/">sns</a>
45
46   ## Simple Queue Service (SQS)
47
48   <a href="sqs/">sqs</a>
49
50   ## Lambda Echo with SNS
51
52   <a href="lambda-echo-sns/">lambda-echo-sns</a></br>
53   **Depends On:** <a href="sns">sns</a>
54
55   ## DynamoDB
56
57   <a href="dynamodb/">dynamodb</a>
58
59   ## DynamoDB Global Table
60   <a href="dynamodb-global-table">dynamodb-global-table</a></br>
61   **NOTE:** Deploy this is you want multi-region replication for highly-available
(HA) <a href="dynamodb-global-table">Lambda Echo SQS DynamoDB</a> pipelines in multiple
regions
62
63   ## Lambda Echo with SQS, SNS, and DynamoDB
```

```
64
65   <a href="lambda-echo-sqs-dynamodb/">lambda-echo-sqs-dynamodb</a></br>
66   **Depends On:** <a href="sns">sns</a> && <a href="sqs">sqs</a>&& <a
href="dynamodb">dynamodb</a></br>
67
68   ## Relational Database Service (RDS)
69
70   <a href="rds/">rds</a></br>
71   **Depends On:** <a href="vpc/">vpc</a>
72
73   ## EC2 with Two Web Servers, EFS, and an RDS Management Server
74
75   <a href="ec2-web-rds/">ec2-web-rds</a></br>
76   **Depends On:** <a href="vpc/">vpc</a> && <a href="rds/">rds</a></br>
77
```

Referring to example 9.1 — Markdown is easy to read and master. For a complete explanation of what you see above consult the *Markdown Cheat Sheet*. I just what to touch on some highlights. Starting from the top, main headings start with a single hashtag '#'. Second-level headings start with two hashtags '##', and so on. **Bold** text is enclosed within two asterisks like so '**This text will render bold.**' On line 10, the anchor tag 'vpc' is a relative link to the repository's vpc project subdirectory.

Example 9.2 shows the Markdown code for the vpc project's *README.md* file shown in figure 9-3.

9.2 README.md file for vpc Project

```
1    # Virtual Private Cloud (VPC)
2
3    Define a VPC with CloudFormation template and deploy via bash script `build.sh`
4
5    # Default VPC
6    ---------
7    All new AWS accounts come with a default VPC in each region. Pictured below is a
default VPC in us-east-1. The default VPC in each region contains a subnet in every
availability zone. The network access control list allows inbound and outbound traffic
from all IPs (0.0.0.0/0), and the default route table enables communication between
resources within and between subnets and forwards outbound traffic to the internet
gateway.
8    <img src="diagrams/vpc/DefaultVPC.png"></img>
9    This works fine for simple deployments but for most applications, you need more
control over network security and application deployment configurations. That's a job
for a custom VPC.
10
11
12   # Custom VPC
13   ---------
14   Pictured below is a custom VPC that has both public and private subnets. A subnet
is public if it accepts inbound traffic from the internet, and a subnet is private if
it does not, or otherwise restricts inbound public traffic. A private subnet uses a NAT
gateway to enable outbound traffic to the internet. To access resources within a
private subnet, say an EC2 instance, you can use a bastion host or session manager.
15   <img src="diagrams/vpc/CustomVPC.png"></img>
16
17   # Deploying the VPC
18   ---------
19   The VPC CloudFormation template given in this project `vpc.yml` defines a custom
VPC with three public and three private subnets, an Internet Gateway, NAT Gateway,
```

```
NACL, RouteTables, Routes, and a Security Group that limits access to a known IP
address. Feel free to customize as you see fit.
20
21     * Install and configure AWS CLI
22     * On Windows -- Install Git with Unix/Linux tools for access to bash shell
23     * Edit `build.sh` and/or `vpc.yml` to customize deployment
24     * You'll need to edit the SecurityGroupAllowedIP parameter and change the IP
address to your IP address
25     * Run `./build.sh` to get help
26     * Run `./build.sh dev oh vpc` to deploy development VPC in us-east-2
27
```

Referring to example 9.2 — On line 10, an `<img>` tag is used to embed an image. In this case, a relative path is provided to the *diagrams/vpc/DefaultVPC.png* image file. By relative path, I mean the path to the image file relative to the *README.md* file. Since this *README.md* file is located in the repository's *vpc* directory, it's assumed the *diagrams* directory is located in the same directory. Alternatively, you could have given a valid web URL to some file on the Internet.

Quick Review

Include a *README.md* file in your project root directory. Its purpose is to provide a project description and instructions on how to deploy and use the application along with any other information you deem fit to include.

A *README.md* file is composed of Markdown code and rendered as HTML in a browser. GitHub will automatically render the *README.md* file on your repository page.

4 Purpose of main.py Module

The purpose of the *main.py* module is to provide a convenient, easily-identifiable point-of-entry for your application. Although I have demonstrated the use of a *main.py* module in earlier chapters, it's worth repeating.

One of the benefits of using a *main.py* module is that it keeps the application run commands consistent between different projects. The *main.py* module for each project is uniquely but predictably configured, however, the command to run different applications remains the same:

```
python3 src/main.py
```

Let me demonstrate. Figure 9-5 shows a project tree view with two files in the src folder: *main.py* and *example.py*.

Referring to figure 9-5 — This figure shows a tree view for the *project_2* directory. Along with the *main.py* module there is a module named *example.py*. The *example.py* module contains all the program functionality. The *main.py* module acts as the main entry point for the application. In other words, the *main.py* module supplies the code required to execute from the command-line. It also imports the *example.py* module and calls any functions or methods required to run the application.

PRO TIP: Use a main.py module to act as the main entry point to your application. Doing so allows you to standardize the run commands across different applications.

 © 2023 Pulp Free Press

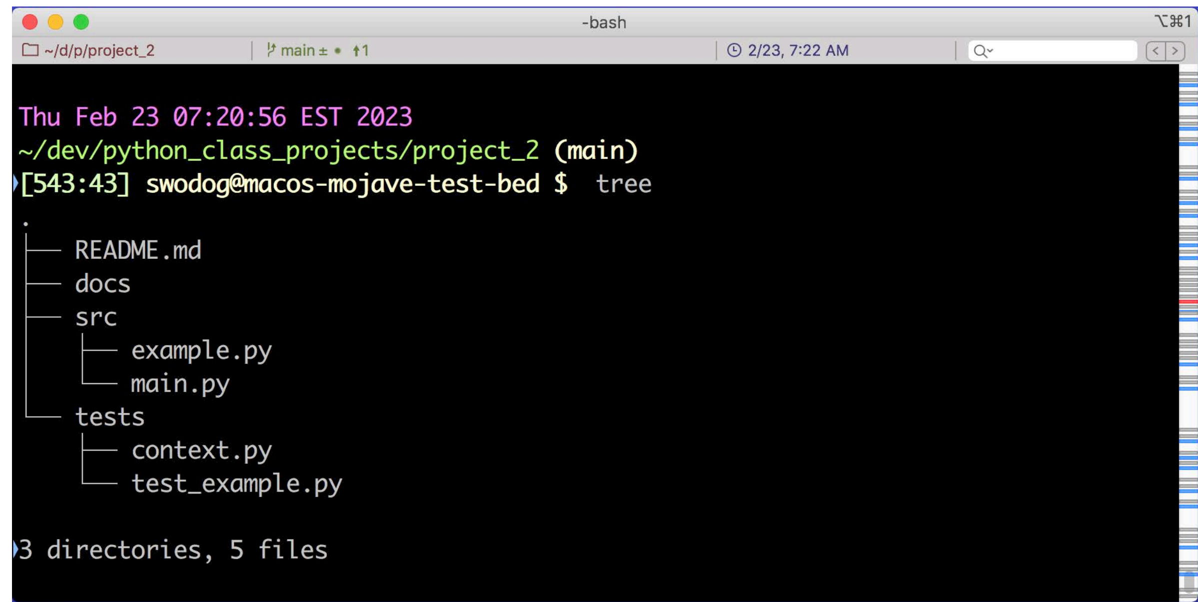


Figure 9-5: Project Tree View

Example 9.3 give the code for the *main.py* module.

9.3 main.py

```
1    """Explicit main execution module."""
2
3    from example import Example
4
5
6    def main():
7        """Execute main program."""
8        example = Example()
9        print(f'Count = { example.get_count() }')
10       example.iter_demo()
11       example.lambda_demo();
12
13
14   # Call main() if this is the main execution module
15   if __name__ == '__main__':
16       main()
17
```

Referring to example 9.3 — You should be familiar with how this code works by now. It checks to see if it's the `"__main__"` module and if so, it calls the `main()` method. The rest of the code will change depending on the project. In this case, from the *example* module it imports the *Example* class, then, in the body of the `main()` method, it creates an object of type Example by calling the `Example()` constructor, followed by several calls to methods provided by the Example object.

Example 9.4 lists the code for the *Example.py* module.

9.4 Example.py

```
1    """example module contains Example class."""
2
3    class Example:
4        """Docstrings.
5
6        Example class
```


© 2023 Pulp Free Press

```
7        """
8
9        def __init__(self):
10          """Initialize example object."""
11          self.count = 0
12
13
14       def get_count(self):
15          """Return self.count."""
16          return self.count
17
18
19       def increment_count(self):
20          """Increment and return self.count."""
21          self.count += 1
22          return self.count
23
24
25       def sum(self, arg_a, arg_b):
26          """Return sum of arg_a + arg_b."""
27          return arg_a + arg_b
28
29
30       def iter_demo(self):
31          """Demonstrate string iteration."""
32          message = 'A string of letters is also a list of chars...'
33          for s in message:
34              print(f'{s} ', end="")
35          print()
36
37
38       def lambda_demo(self):
39          """Demonstrate lambda function."""
40          print(f'{list(map(lambda x: x + x , [1,2,3,4,5] ))} ', end="")
41
```

Referring to example 9.4 — This program doesn't do much other than demonstrate a few basic Python concepts, which I cover in great detail later in the book. Just note that the *example* module defines a class named Example, and that the Example class defines a handful of methods including a constructor (`__init__(self)`) method. To run this program type the following command from the project root directory:

```
python3 src/main.py
```

Figure 9-6 shows the results.

Figure 9-6: Results of Running the `main.py` Module from the Command Line

4.1 Parting Thoughts

The guts of a *main.py* module will change depending on the nature of the application. Different applications will use different modules, create different objects, and call different methods. Using a *main.py* module is good practice because it allows you to standardize on application run commands. It also simplifies build script maintenance, a topic I'll discuss in detail in *Chapter 11: A Bash Build Script*.

Quick Review

The purpose of a *main.py* module is to provide a main entry point for an application. It supports standardized application run commands, which simplifies application build script maintenance.

5 Linking Modules Located in src Directory to Tests

Refer again to the project tree view shown previously in figure 9-5. Notice in the *tests* directory a file named *context.py*. The purpose of the *context.py* file is to provide a central module from which test modules can import modules defined in the *src* directory. If you have ever been ticked off and frustrated trying to place Python unit tests in a separate directory only to be informed, when you try to run the tests, `"Module not found..."` or words to that effect. The *context.py* module solves that problem. Example 9.5 gives the listing for the *context.py* file.

9.5 context.py

```
1    import os
2    import sys
3
4    sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), \
5    '../src/')))
6
7    from example import Example
8
```

Referring to example 9.5 — In this example, the *context.py* module is adding the project *src* directory to the Python system path. Then, on line 7, it imports the Example class from the *example* module. Example 9.6 shows how the *context.py* module is used in a unit test.

9.6 test_example.py

```
1    import unittest
2    from context import Example
3
4    class Test_Example(unittest.TestCase):
5
6       def test_increment(self):
7          example = Example()
8          assert example.get_count() == 0
9          example.increment_count()
10         assert example.get_count() == 1
11
12
13      def test_sum(self):
14         example = Example()
15         assert example.sum(1, 1) == 2
16         assert example.sum(1,2) == 3
```

Referring to example 9.6 — The *text_example.py* module implements a set of unit tests using Python's built-in unittest framework. If you're unfamiliar with unit tests, don't fret — I cover unit testing in *Chapter 21: Unit Testing*. On line 1, I import the `unittext` module. On line 2, I import the Example class from the *context* module. I then use the Example class in the body of the unit tests as required. To execute these tests run the following series of commands from the project's root directory:

```
pushd tests

python3 -m unittest -v

popd
```

The `pushd` command saves the current directory and switches to the *tests* directory. The *unittest* module executes in verbose mode (`-v`), and finally, the `popd` command returns to the project root directory. This series of commands is necessary because to resolve module names located in the *tests* directory the *unittest* module must run in the *tests* directory. Figure 9-7 shows the results of running these commands.

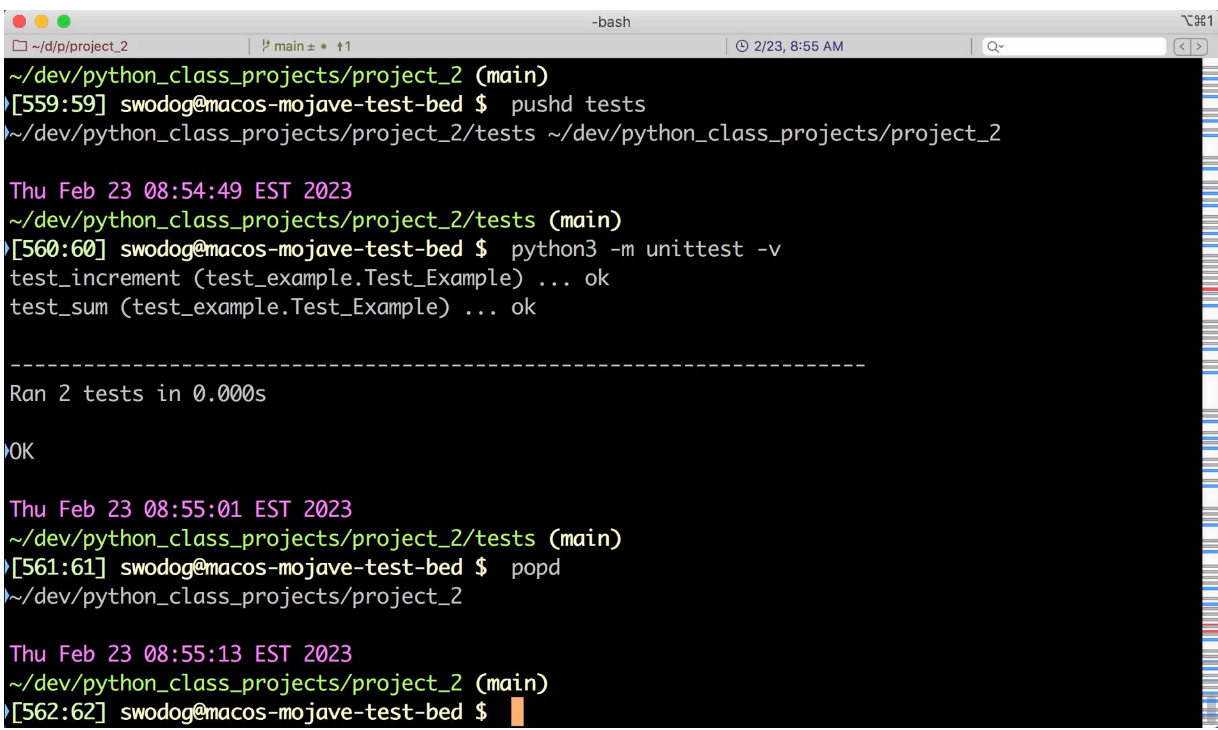
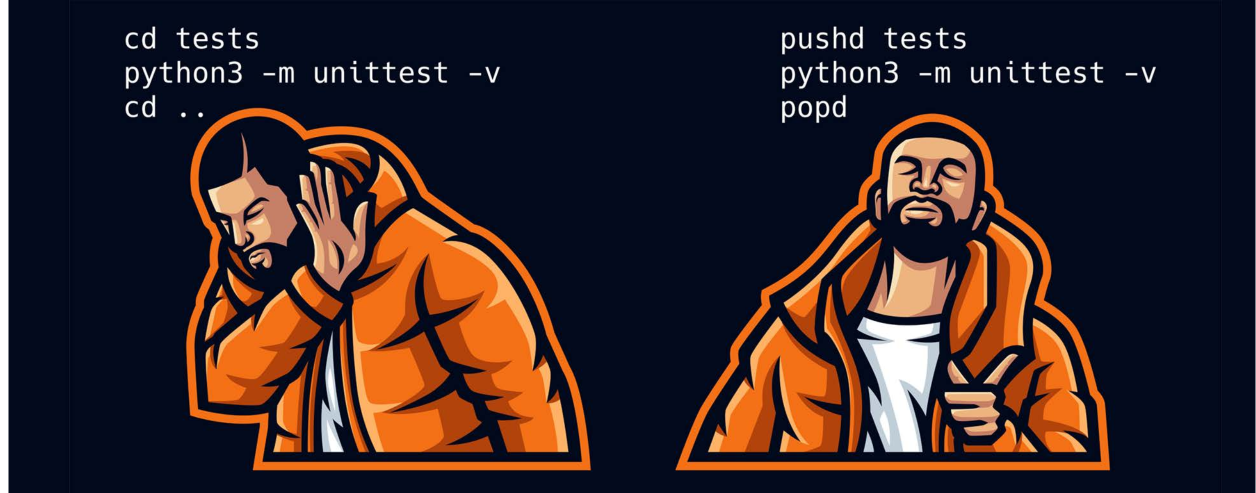


Figure 9-7: Running `unittest` Module Starting from Project Root Directory

Referring to figure 9-7 — Starting from the *project_2* root directory, the `pushd tests` command saves the current working directory, prints it to the console, and changes to the *tests* directory. Now in the *tests* directory, the `python3 -m unittest -v` command runs all the unit tests it finds via *autodiscovery* and writes a verbose report of test results to the console. Finally, the `popd` command returns to the previous working directory.

In the interest of full disclosure, you could have simply changed to the *tests* directory with the `cd tests` command, run the tests, and changed back to the project root directory with `cd ..`, but using the `pushd` and `popd` commands is really the way to go.



Quick Review

The *tests/context.py* module provides a central module from which unit tests located in the project's *tests* directory can import modules located in the *src* directory. Favor the use of the `pushd` and `popd` commands over the `cd` command when you need to make a temporary directory change and return to the original directory.

6 Baseline Project Template Download

You can find the *baseline project template* in the book's repository: *https://github.com/pulp-freepress/cst_with_python_1st_ed*, in the chapter09 directory.

Summary

The baseline project organizational structure includes a *README.md* file, an optional *.gitignore* file, and three directories: *src*, *tests*, and *docs*. Store application source files in the *src* directory, which can have any number of files and subdirectories. Use the *tests* directory to store unit and integration test files. Test filenames usually begin with the string `test`. Store all project documentation with the exception of the *README.md* file in the *docs* directory. This includes automatically generated documentation.

The purpose of the *.gitignore* file is to maintain a list of files and directories you **do not** want to track or push to your repository. You can either create a *.gitignore* file from scratch and add ignored items as required, or start from a template. You can search for suitable *.gitignore* file examples online or use a GitHub template.

Include a *README.md* file in your project root directory. Its purpose is to provide a project description and instructions on how to deploy and use the application along with any other information you deem fit to include.

A *README.md* file is composed of Markdown code and rendered as HTML in a browser. GitHub will automatically render the README.md file on your repository page.

The purpose of a *main.py* module is to provide a main entry point for an application. It supports standardized application run commands, which simplifies application build script maintenance.

The *tests/context.py* module provides a central module from which unit tests located in the project's *tests* directory can import modules located in the *src* directory. Favor the use of the `pushd` and `popd` commands over the `cd` command when you need to make a temporary directory change and return to the original directory.

Skill-Building Exercises

1. **Download Baseline Project Template:** Download the baseline project template from the book's repository and adopt it for use with you projects. I recommend you modify it to suit your needs and push it to your personal repository for future use.

2. **Terminal Command Practice:** Practice using the `pushd` and `popd` commands. Start in any directory and change to a different directory with the `pushd` command. Return to your starting directory with the `popd` command.

Suggested Projects

1. **Sphinx Documentation Generator:** Research the Sphinx documentation generator tool and use it to generate documentation for your projects. *https://www.sphinx-doc.org*

Self-Test Questions

1. What's the purpose of the *src* directory?

2. What's the purpose of the *docs* directory?

3. What's the purpose of the *tests* directory?

4. What types of project files are found in the *src* directory?

5. What types of project files are found in the *docs* directory?

6. What types of project files are found in the *tests* directory?

7. What's the purpose of the *main.py* module?

8. What's the purpose of the *tests/context.py* module?

9. What do the commands `pushd` and `popd` do?

10. What are some advantages of using the `pushd` and `popd` commands over the `cd` command?

References

Git-SCM .gitignore Page, *https://git-scm.com/docs/gitignore*

GitHub .gitignore Template Repository, *https://github.com/github/gitignore*

MarkdownGuide.org, *https://www.markdownguide.org*

Sphinx Documentation Generator, *https://www.sphinx-doc.org*

Notes

0 0 0 0 1 0 0 1

© 2023 Pulp Free Press