00001011

CHAPTER 11

A Bash Build Script

Learning Objectives

- *Automate repetitive Python development tasks with a bash build script*
- *State the purpose and benefits of using a bash build script*
- *Define and use constants in a bash script*
- *Define and use variables in a bash script*
- *Define and call functions in a bash script*
- *Process bash script command-line arguments*
- *Pass command-line arguments to functions*
- *Organize bash script code with functions*
- *Detect the operating system*
- *Check for and verify presence of required development tools*
- *Execute a default action when calling bash script with no arguments*
- *Configure and display bash script usage help*
- *Apply the chmod command to make a bash script executable*
- *Demonstrate your ability to execute a bash script from the command line*

Introduction

As project complexity grows, so too grows the number of commands you must memorize to run your project's development tools. For example, in the previous chapter, you learned how to create virtual environments with Pipenv. You also learned how to run a Python program in a virtual environment with the `pipenv run` command. Now, add a testing framework, documentation generator, linter, and other tools, and you'll realize soon enough how easy it is to make simple mistakes when entering these commands manually on the command line. Using a bash build script solves this problem.

In this chapter, I introduce you to a baseline bash build script (build.sh) that simplifies project management and allows you to create a set of standardized commands with which you can run the various tools associated with your project. The baseline build script verifies the presence of required development tools. It's flexible, too. You can customize it to suit your needs. It's cross platform as well. The baseline build script runs on macOS, Linus, and Windows within a Git Bash terminal. It supplies usage instructions, which is the default action the script performs when you run it with no arguments.

It is not my intent to give you a complete course in bash scripting. That would require a dedicated book on the topic (*Pro Bash Programming*). Rather, I give you to a short but complete build script, show you how to run it, explain how it works, and point out how to extend it to suit your needs. Once you get the hang of using the build script, you'll wonder how you ever lived without it!

Automating repetitive development tasks with a bash build script speeds up the software development process by eliminating common mistakes you'd normally make by trying to run different development tools manually, from memory, from the command line. Bash scripting, even the small, measured dose you'll learn in this chapter, is a professional skill worth adding to your programmer's tool belt.

Perhaps the most important takeaway from this chapter, and from the book in general, is the more tasks you can automate, the less likely you are to screw something up trying to do it manually. This has an overall net positive effect on software quality, but most importantly, it reduces your stress level and helps you live a long, happy life.

1 Baseline build.sh Script

Example 11.1 gives the baseline bash build script called *build.sh*.

11.1 build.sh

```
1    #!/bin/bash
2
3    # Global Constants
4    declare -r SRC_DIR="src"
5    declare -r TESTS_DIR="tests"
6    declare -r DOCS_DIR="docs"
7
8    # TOOLS: A list of required tools. Edit as required.
9    declare -r TOOLS="git python3 pipenv"
10
11   # Global Variables
```

```
12  declare _confirm=1
13
14  check_tools( ) {
15     echo "Checking for required tools..."
16     for tool in $TOOLS
17     do
18        command -v $tool &> /dev/null && \
19        ([ $_confirm -eq 1 ] && echo "$tool: OK" || true) || \
20        (echo "$tool: MISSING"; exit 1);
21     done
22  }
23
24  display_usage( ) {
25     echo
26     echo " Usage: "
27     echo "   ./`basename $0` [ no argument | --checktools | --help | "
28     echo "                --install | --runmain | --runtests | "
29     echo "                --docstrings ] "
30     echo
31     echo " Examples: "
32     echo "    $0                 # Default: --checktools and --help"
33     echo "    $0 --checktools    # Check for required tools"
34     echo "    $0 --help          # Show this message"
35     echo "    $0 --install       # pipenv install && install --dev"
36     echo "    $0 --runmain       # pipenv run python3 $SRC_DIR/main.py"
37     echo "    $0 --runtests      # pipenv run pytest $TESTS_DIR/"
38     echo "    $0 --docstrings    # pipenv run pydocstyle $SRC_DIR/"
39     echo
40  }
41
42  default_action( ) {
43     check_tools
44     display_usage
45  }
46
47  runtests( ) {
48     pipenv run pytest $TESTS_DIR/
49  }
50
51  runmain( ) {
52     if [[ "$OSTYPE" == "linux-gnu"*  ]]; then
53         pipenv run python3 $SRC_DIR/main.py
54      elif [[ "$OSTYPE" == "darwin"*  ]]; then
55          pipenv run python3 $SRC_DIR/main.py
56      elif [[ "$OSTYPE" == "msys"* ]]; then
57          pipenv run python $SRC_DIR/main.py
58      else
59          echo "Unknown execution environment. "
60        echo "Edit build.sh and add your os type to the runmain( ) method"
61      fi
62  }
63
64  install( ) {
65     pipenv install --dev
66     pipenv install
67  }
68
69  check_doc_strings( ) {
70     pipenv run pydocstyle -v $SRC_DIR/
```

```
71  }
72
73  process_arguments( ) {
74      case $1 in
75          --help) # Display usage and help screen
76              display_usage
77              help
78              ;;
79          --checktools) # Verfy required tools are installed
80              check_tools
81              ;;
82          --runtests)  # Run all tests
83              runtests
84              ;;
85          --runmain) # Run application
86              runmain
87              ;;
88          --install) # install all packages listed in Pipfile
89              install
90              ;;
91          --docstrings) # Check source files for valid docstrings
92              check_doc_strings
93              ;;
94          *)
95              default_action
96      esac
97  }
98
99  main(){
100     process_arguments "$1"
101     exit 1
102 }
103
104 # Call main( ) with all command-line arguments
105 main "$@"
106
```

Referring to example 11.1 — This script is the focus of this chapter. I will demonstrate its operation, explain how it works, and show you how to modify it to suit your needs. But first, you need to be aware of a few assumptions and take care of several preliminaries.

1.1 Assumptions

Although you can configure the baseline build.sh script to suit your needs, in its current configuration, I'm making the following assumptions:

- You're using the baseline project organizational structure presented in Chapter 9: Project Organization. Specifically, you're putting source files in the project's *src* directory, unit tests in the project's *tests* directory, and project documentation in the *docs* directory.
- You're using Pipenv to create and manage virtual environments.

That's it! At least that's all I can think of for now. OK, before running the script, you need to take care of a few preliminaries.

1.2 Preliminaries

To run the script, you need to create a virtual environment with Pipenv, install two packages, and make the script executable.

1.2.1 Create Pipenv Virtual Environment

Use Pipenv to create a virtual environment with the Python version of your choice. Be sure you're in your project folder before creating the virtual environment. I'll create a virtual environment with Python version 3.10.

```
pipenv --python 3.10
```

Figure 11-1 shows the results of running this command.

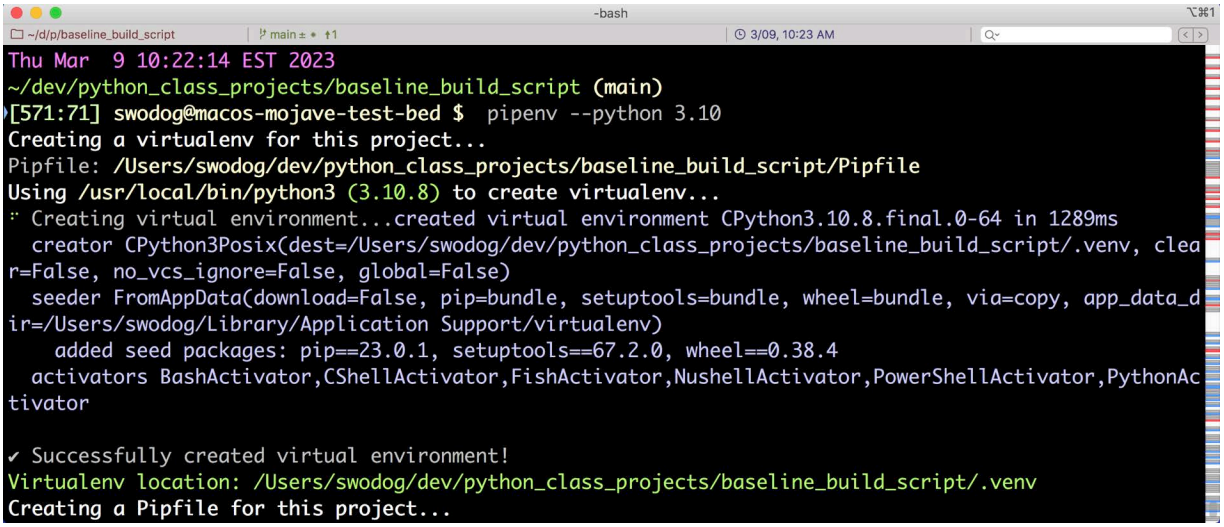

Figure 11-1: Installing a Python 3.10 Virtual Environment with Pipenv

Referring to figure 11-1 — You can use any version of Python you require. As I stated in Chapter 1, my only general assumption is that it's Python 3.10 or greater as many examples in the book use features found in more recent versions of Python. (***Note:*** *As a general rule, code written for older versions of Python 3 will run on newer versions of Python 3, but that's a one-way street. Code written for newer versions of Python 3 will most likely not run on older versions of Python 3, especially if the code targets specific features of the newer version of Python.*)

1.2.2 Install pydocstyle and pytests Packages

Once you have created a virtual environment with Pipenv, install the following packages for development: *pydocstyle* and *pytest*.

The pydocstyle package is used to check for valid and missing docstrings. PEP 257 - Docstring Conventions, provides guidance on how to format and apply Python docstrings. Install the package for development with the following command:

```
pipenv install --dev pydocstyle
```

The pytest package is a unit testing framework I much prefer over Python's unittest library. Install the pytest package for development with the following command:

```
pipenv install --dev pytest
```

Figure 11-2 shows the results of running these two commands.

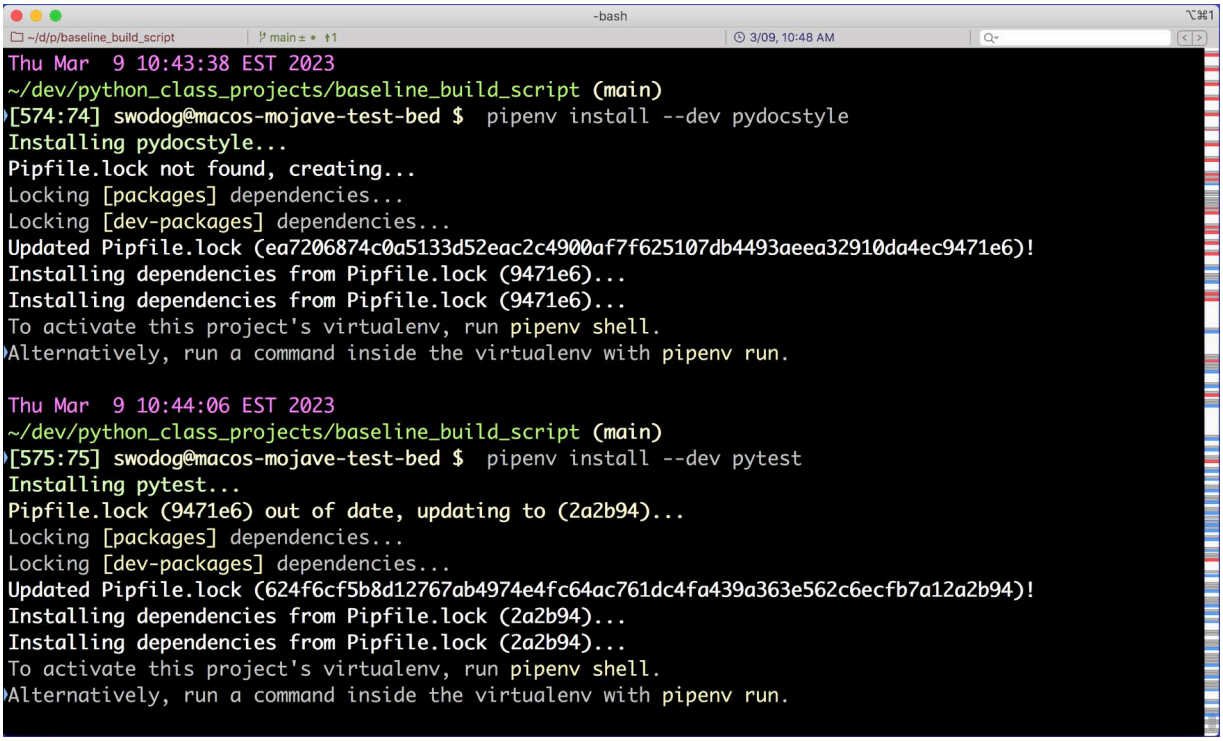

Figure 11-2: Installing pydocstyle and pytest Packages for Development

The project directory structure should now resemble that shown in figure 11-3.

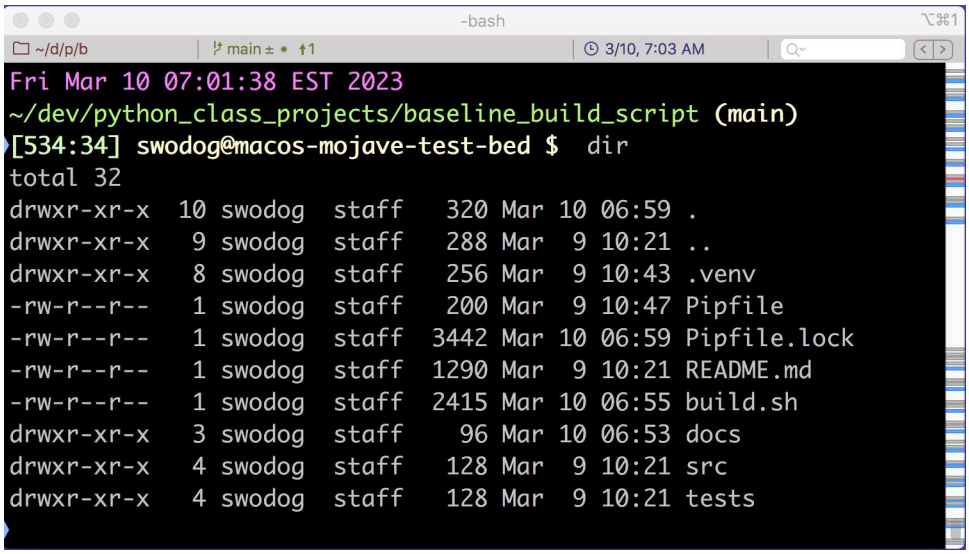

Figure 11-3: Project Directory Structure after Installing Virtual Environment and Packages

Referring to figure 11-3 — Recall from Chapter 10 that the Pipfile.lock is created when you install the first package into the project's virtual environment. Now, let's make the build.sh file executable.

1.2.3 Make The build.sh Script Executable

Before you can run a bash script you need to change its file permissions to make it an executable file. Use the `chmod` command to change a file's permissions. To make the build.sh file executable, use the `chmod` command like so:

```
chmod a+x build.sh
```

-or-

```
chmod 755 build.sh
```

Either version of the command will work. Figure 11-4 shows the directory listing after changing the build.sh file's execute permissions. Can you spot the differences between figure 11-3 and this listing?

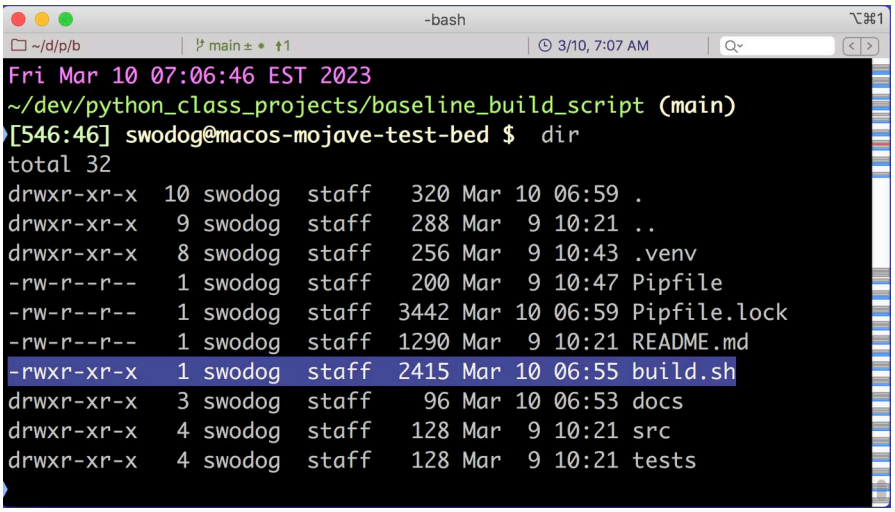

Figure 11-4: Project Directory Listing after Changing build.sh Execute Permissions.

Referring to figure 11-4 — I highlighted the build.sh file. Notice the file permission bits have changed. I suggest you write the `chmod` command down in your Engineer's Notebook. If you write bash scripts, you'll need to make them executable before you can run them. OK, let's run this puppy.

1.3 Run The Build.sh Script

To run the build.sh script, make sure you're in the project's root directory (Where you should have been all along!), and preface it with the dot slash characters `"./"` like so:

```
./build.sh
```

You should see the command usage display as shown in figure 11-5.

Referring to figure 11-5 — Running the build.sh script with no arguments triggers a default action, namely, it checks for required development tools (`git`, `python3`, and `pipenv`), and then displays help on how to use the script with usage examples.

The baseline script implements the following command-line arguments: `--checktools`, `--help`, `--install`, `--runmain`, `--runtests`, and `--docstrings`. The Examples section includes a brief comment for each command that explains what command is actually running. For example, when you run `./build.sh --runmain`, the script is actually running `pipenv run python3 src/main.py`. Where have you seen that command before?

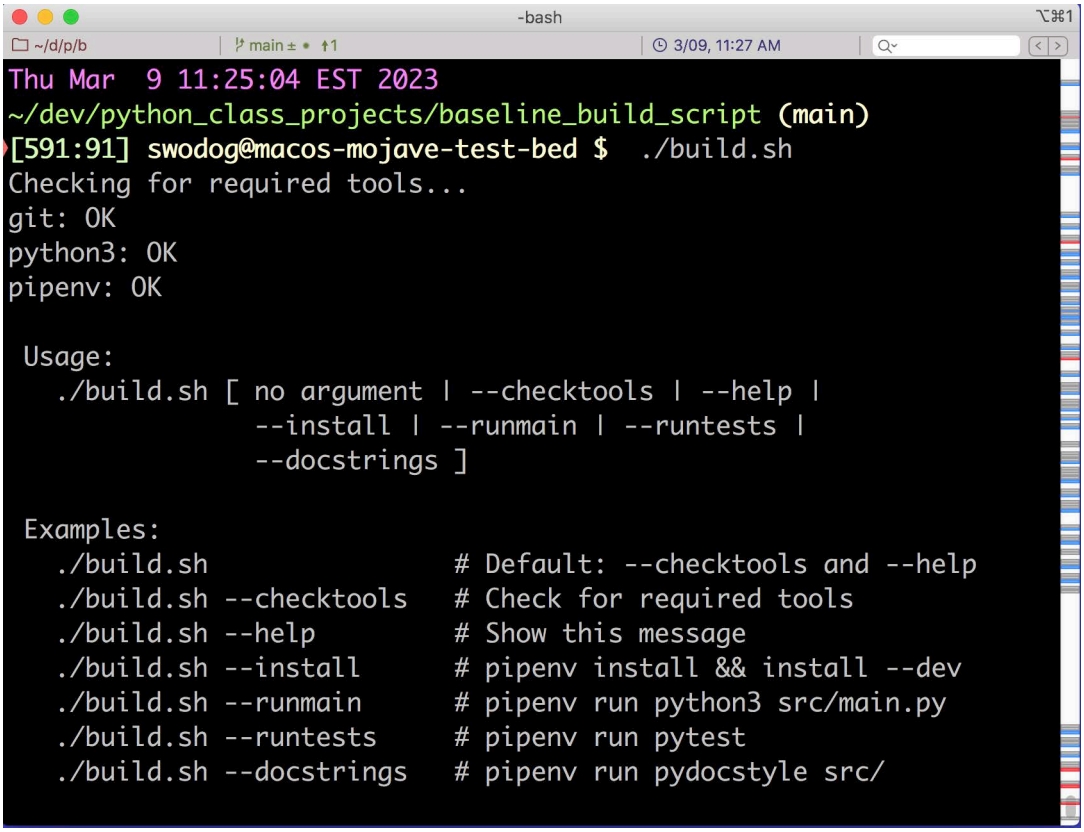

Figure 11-5: Running `./build.sh` with No Arguments

Quick Review

The baseline build.sh script is configured to work with the baseline project organizational structure presented in chapter 9. Specifically, it expects to find source files in the project's *src* directory, unit tests in the *tests* directory, and project documentation in the *docs* directory. You can, however, configure the script to suit your needs.

Before running the script, create a virtual environment with Pipenv, install the pydocstyle and pytest packages, and give the build.sh file executable permissions with the `chmod` command.

2 Build.sh script Anatomy

In this section, I will walk you through the build.sh script code and explain how it works. While this section is not a complete treatment on bash scripting, you should walk away with a fundamental understanding of what the script is doing and how to modify it to suit your needs. The best way to wrap your head around the script is to first understand how it's structured, and then trace the various execution paths. So, let's start at the top and learn what makes this a bash script. Perhaps the best way to proceed would be to print the build.sh script given in example 11.1 and take notes as we move along, that way you don't need to flip back and forth. Just a suggestion.

2.1 The Shebang or Hash-Bang

The very first line of the script, line one, contains a very important sequence of characters referred to as either the *shebang* or *hash-bang*. The shebang is a special type of comment that consists of the hash tag character plus the exclamation point `'#!'`. The purpose of the shebang is to indicate which shell should execute the script. Since this is a bash script, its shebang looks like this:

```
#!/bin/bash
```

Every bash script starts with a shebang.

2.2 Comments

Comments begin with the hashtag character `'#'`. Everything to the right of the hashtag is ignored up to the end of the line. Comments appear throughout the script to add context and explanation to what's going on in the code.

2.3 Constants and Variables

On lines 4 through 12, I've declared several *constants* and a *variable* with the help of the `declare` keyword. The difference between a constant and a variable is that constants are set to be *readonly* with the `-r` option. Also, by convention, which is nothing more than a bunch of software developers shaking hands and doing pinky swears, constants appear as `ALL_CAPS` with underscores separating words. Variables, on the other hand, consist of all lowercase characters. For more context see the *Google Shell Style Guide*.

The constants on lines 4 - 6 declare names for project artifact directories. For example, the constant on line 4 declares a constant named `SRC_DIR`, short for **s**ou**rc**e **dir**ectory, that contains the string `"src"` like so:

```
declare -r SRC_DIR="src"
```

I've declared these constants to make it easy to change the script if you use different names for these artifact directories.

To use a constant or a variable, preface it with a dollar sign character `'$'` like so `"$SRC_DIR"`. Refer to the body of the `display_usage()` function beginning on line 24 to see this in action.

You can scan the script to see where the constants appear. If one day you woke up on the wrong side of the bed and decided to put your source code in the "mycode" directory, all you'd need to do to make the script work is to change the constant to read:

```
declare -r SRC_DIR="mycode"
```

And the script would march along smartly.

The TOOLS constant declared on line 9 is set to a string containing the names of required development tool commands separated by a space like so:

```
declare -r TOOLS="git python3 pipenv"
```

The spaces that separate each command name in the string become important if you want to process the string like a list, which is what happens in the body of the `check_tools()` function beginning on line 14. More about this function later.

2.4 Processing Command-Line Arguments

Skip down to line 105 of the build.sh script. Here the `main` method is called with the special parameter`"$@"` as an argument, indicating that all of the command-line arguments present are passed to the `main()` method defined on line 99. The quotes surrounding the `$@` characters indicate the requirement to preserve arguments that contain white space.

You learned earlier that you can call the build.sh script with or without arguments like so:

```
./build.sh
./build.sh --checktools
./build.sh --help
./build.sh --install
./build.sh --runmain
./build.sh --runtests
./build.sh --docstrings
```

When passing the "$@" parameter to the `main` method, the `"$@"` parameter is expanded to the positional arguments "$1", $2", $3", etc. The special positional argument $0 indicates the path and name of the script being executed and is *not* included in the `"$@"` parameter expansion.

In normal operation, and as currently configured, the build.sh script is only concerned with the first positional argument $1. On line 100, the `main()` function passes the incoming parameter `"$1"` as an argument to the `process_arguments` function. So, if the build.sh script is called with two arguments like so:

```
./build.sh --checktools --runmain
```

The call to `main` would pass all of these arguments to the `main()` function with the `"$@"` argument. In the body of the `main()` method these arguments become `$1` and `$2`, but since `main` is only interested in the first argument, it calls the `process_arguments` function with the argument `"$1"`. Inside the body of the `process_arguments()` function on line 74, the incoming argument is accessed as the parameter name `$1`.

2.5 Translate Command-Line Arguments Into Actions

It's in the body of the `process_arguments()` function where command-line arguments get translated into script actions. This is done in the body of a `case` statement which begins on line 74.

2.5.1 The case Statement

The `case` statement begins with the keyword **`case`** on line 74 and ends with its reverse, **`esac`**, on line 96. Everything between the `case` and `esac` keywords is considered to be in the *body* of the `case` statement. The value of the `$1` parameter is checked against various values, or *cases*, indicated by the `')'` characters. For example, the first case, `--help)`, is defined on line 75.

You can read a case statement like a series of `if` statements like so: *"If the* `$1` *parameter contains the value* `--help` *then call the* `display_usage` *function followed by the* `help` *function, and then exit the case* `";;"`.

From here on out you can trace the execution of each case to see which functions are called for each command-line argument value. For example, you can see that when the script is run with the `--runmain` argument like so:

```
./build.sh --runmain
```

The `--runmain)` case executes, which in turn calls the `runmain` function. The `runmain()` function definition begins on line 51. It looks complicated, but what it's doing is checking the operating system with the help of the bash OSTYPE shell variable and a series of `if/then` statements.

2.5.2 The if Statement

Starting at the beginning of the `if` statement on line 52, you can read it as: "*If the OSTYPE variable contains the value "linux-gnu" then execute the main.py module using the command `pipenv run python3 src/main.py`*"

The three expected OSTYPEs include `linux-gnu`, `darwin`, and `msys`, which correspond respectively to Linux, macOS, and Windows running the Git Bash terminal. If the Git Bash terminal is detected, the script executes the *main.py* file with the `python` command.

If the script finds itself running on an unknown system, the `else` clause on line 58 executes and writes an error message to the console.

Like the `case` statement, an `if` statement begins with the keyword **`if`** and ends with its reverse **`fi`**. This is just something you need to get used to when writing bash scripts. The body of an `if` statement can contain any number of `elif` clauses, which stands for *else if*. If neither the `if` clause nor any of the `elif` clauses execute, the `else` clause executes.

2.6 Check Presence of Required Development Tools

When the build.sh script is run with the `--checktools` command-line argument, the `check_tools` method is called. It uses a `for` statement to step through each required tool in the TOOLS constant string. Note that the TOOLS constant contains a string. Each tool name is separated by a space, and the `for` statement will treat the space as a separator and treat the string as an array of strings. So, for example, the TOOLS constant contains the tool names: `"git python3 pipenv"`. The `for` statement starts like so:

```
for tool in $TOOLS
```

You can read this statement as "*For each tool listed in the TOOLS string...*". The rest of the code in the body of the `for` statement, that is the code between the **`do`** and the **`done`** keywords, attempts to execute the tool command with the following line of code:

```
command -v $tool $> /dev/null && \
```

If the tool is present and executes successfully, this line evaluates to:

```
true && \
```

The && is a logical AND connector, The `'\'` is a line continuation character that allows you to split long commands across multiple lines. Speaking of the next line, the code on line 19 is contained in parentheses:

```
([ $_confirm -eq 1 ] && echo "$tool: OK" || true) || \
```

This line starts by comparing the `_confirm` variable with the value `1`. If it evaluates to `true`, the `echo` statement prints out the name of the tool followed by "OK" (See figure 11-5). If it evaluates to `false`, the `||` will execute line 20, which prints the tool name followed by "MISSING" and then calls `exit 1`, which indicates an error condition.

Note that although I'm not doing so in this version of the script, you could call the `check_tools()` function before attempting to execute each tool, check for the exit code 1, and exit the script as appropriate.

I recommend you experiment with the tools listed in the TOOLS constant. Add a tool name that does not exist to see what happens.

2.7 Displaying Help and Usage

When the build.sh script executes with no arguments or with the `--help` argument, it displays guidance on how to use the script. This is done in a straightforward manner as shown in the `display_usage()` function. Note the use of the `$0` parameter in the echo statements to display the script name. Also note the use of the constants $SRC_DIR and $TESTS_DIR. Changing the constant values will automatically update the help text.

2.8 Parting Thoughts

If you have a good understanding what the build.sh script does and how it works, you know all the bash you need to know for the purposes of this book. I do recommend, however, that you continue to study bash, especially how to automate tasks with bash scripts. The combination of bash and Python puts an immense amount of power at your fingertips.

Quick Review

The build.sh script starts with a shebang, and consists of comments, constants, variables, and code logically organized into functions. The build.sh script processes command-line arguments with the help of a `case` statement. The `case` statement compares command-line argument values against a set of *cases* and executes the code in the body of the corresponding case.

The special `"$@"` parameter represents all the arguments read from the command line. The `"$@"` parameter expands into positional arguments: `"$1"`, `"$2"`, `"$3"`, ... `"$n"`. Arguments passed to functions are referenced by positional parameters. The special parameter `$0` represents the name of the script.

Summary

The baseline build.sh script is configured to work in the baseline project organizational structure presented in chapter 9. Specifically, it expects to find source files in the project's *src* directory, unit tests in the *tests* directory, and project documentation in the *docs* directory. You can, however, configure the script to suit your needs.

Before running the script, create a virtual environment with Pipenv, install the pydocstyle and pytest packages, and give the build.sh file executable permissions with the `chmod` command.

The build.sh script starts with a shebang and consists of comments, constants, variables, and code logically organized into functions. The build.sh script processes command-line arguments with the help of a `case` statement. The `case` statement compares command-line argument values against a set of *cases* and executes the code in the body of the corresponding case.

The special `"$@"` parameter represents all the arguments read from the command line. The `"$@"` parameter expands to positional arguments: `"$1"`, `"$2"`, `"$3"`, ... `"$n"`. Arguments passed to functions are referenced by positional parameters. The special parameter `$0` represents the name of the script.

Skill-Building Exercises

1. **Study Bash Scripting Fundamentals:** Procure the book *Pro Bash Programming: Scripting the GNU/Linux Shell*, by Chris F.A. Johnson, Apress. Dive deeper into the various topics discussed in this chapter. Focus on developing a firm understanding of what the build.sh script is doing and how it works.

2. **Experiment With TOOLS Constant:** Add another tool name to the TOOLS constant defined on line 9 of the build.sh script. Use a name that doesn't exist, like "my_script", to see the effects of a missing tool.

3. **Study Shell Style Guide:** Study Google's Shell Style Guide and apply its recommendations to bash scripts you create: *https://google.github.io/styleguide/shellguide.html*

4. **Process Command-Line Arguments:** Write a bash script and experiment with processing command-line arguments. Pass all the command-line arguments to a `main()` function using the special parameter `"$@"` or `$@`. Process the command-line arguments using a `case` statement. Organize your code into functions. Don't worry about what, exactly, the script will do. Seek to really understand the difference in behavior between using `"$@"` with quotes and `$@` without quotes. Start by simply echoing the command-line argument back to the console to see how spaces are treated.

5. **Bash Shell Variables:** Make a list of all the shell variables available to a base script and note their purpose.

Suggested Projects

1. **Build Script Modification:** Modify the build.sh script to automatically detect the version of Python available on the system and set a variable to that value. Replace all instances of hard-coded references to `python` or `python3` in the script with the variable. One possible name for the variable might be `python_version` and could be declared and initialized like so:

```
declare python_version="python3"
```

2. **Script Idea - Initialize Project Directory:** Write a bash script that initializes a project directory based on the project organizational structure given in chapter 9. Have the script take a command-line argument that specifies a path and project name.

Self-Test Questions

1. What's the purpose of a bash build.sh script?

2. How does the build.sh script process command-line arguments?

3. What does the special parameter `"$0"` represent?

4. What does the special parameter `"$@"` expand into?

5. How does the `"$@"` parameter behave differently if referenced with and without quotes?

6. Which constant would you need to change if you stored your source code in a directory other than *src*?

7. What value does the code snippet `command -v $tool` evaluate to if the `$tool` name can be successfully executed? What value results if execution fails?

8. What does the `-r` mean when used to declare a constant?

9. How does a `for` statement treat a string of words separated by spaces?

10. Can you call a function before it is declared in the script? Explain your answer.

11. What's does this sequence of characters `#!/bin/bash` mean on the first line of the build.sh script?

12. What are two common names given to the special comment `"#!"`?

References

Pro Bash Programming: Scripting the GNU/Linux Shell, *https://doc.lagout.org/programmation/Shell%20/Pro%20Bash%20Programming.pdf*

GNU Bash Website, *https://www.gnu.org/software/bash/*

GNU Bash Reference Manual, *https://www.gnu.org/software/bash/manual/bash.pdf*

Sed Manual, *https://www.gnu.org/software/sed/manual/sed.html*

Awk User's Guide, *https://www.gnu.org/software/gawk/manual/gawk.html*

Notes