Programming from Scratch

Mark Bools

March 14, 2022

Document ID: B000000001
Last Modified: 2022-11-10

Contents

Chapter 1

How to. . .

This chapter offers some guidance on getting the most from this book.

1.1 . . . read this book

This book contains mistakes. Deliberate mistakes (and no doubt some mistakes that I did not intend, that's life). Sometimes we will do things more than once. WTF? In most educational material we are presented with 'perfect' solutions, this is not realistic. The real world sucks. It changes constantly and today's ideal solution is okay but tomorrow the boss (or customer) decides new technology is desirable how do we handle this? This is were some soft skills may be required to persuade them to change their mind. Assuming we cannot persuade them to change we need to plan and execute a migration from the older solution to the new solution. Rather than avoid this sort of complexity this course takes it head on.

Don't have time to follow along from the start? No problem. At the start of each section you will find details of how to create an appropriate environment for that section (see §1.3). These 'checkpoints' also mean that if you mess up you can simply throw your environment away, recreate it using the closest checkpoint and continue with the course. In fact I encourage you to mess up your environment. You will learn much more by 'playing'. So set up and environment, mess around with your own ideas and then, when you are ready to do the next part of the course, either restore your own saved snapshots or tear down the environment and build a new one using the provided checkpoint code.

Having difficulty? No problem. Ask for help. Someone in the community may help and since I am in the community I can also help clarify things. If enough people are confused by the material then obviously I messed up and need to rewrite that material to be more clear; it shall be done.

All of these books undergo constant maintenance (hopefully improvement), my only goal is to make the material more clear and more accessible over time.

1.2 ...get the most from this book

Do the examples! You'll get much more from the material by following along and investigating for yourself.

1.3 ...manage your workspace

We are going to be doing a lot of practical work throughout this book so it is worthwhile considering how we will manage our workspace.

A lot of this section will only make sense once you have read about the tools Git, VirtualBox, and Vagrant but I'm assembling basic advice here to make it easier to refer back to later.

1.3.1 Initial setup of your workspace

If you follow this book from page one to the end you should find your workspace is always in sync with whatever the book is dealing with but practically many of you will jump to sections of particular interest, skipping many sections. Anticipating this I've put in plenty of checkpoints. All of the material (including checkpoints) is held in a single Git repository, so I recommend getting that first.

Assuming you have installed Git (see §2.4) you can create your project workspace.

```bash
mkdir pfs
cd pfs
git clone --depth 1 https://gitlab.com/saltyvagrant.classes/pfs-material.git course-material
mkdir classroom
mkdir archive
```

Your `pfs` workspace now contains three directories; `course-material` holds all this book's accompanying material, `classroom` is where you will follow along with the course, and `archive` is where we will store various backup files.

Throughout this book I use the `WSR` directory[1] to refer to the root of your workspace. Any path that does not explicitly start from the `WSR` root assumes you are following instructions from the last checkpoint and are relative to whichever directory you should be in at the time.

[1]If you are using Microsoft Windows as your host you will need to convert '/' to '\' in paths whenever working in the host workspace. (It is precisely because of this sort of "conversion" nonsense that we use the guest workspace most of the time.)

```bash
1 cd WSR
2 cd WSR/classroom
3 cd ../course-material
4 cd WSR
5 cd classroom
```

Lines 1, 2, and 4 each start with `WSR` and are therefore not really relative to your current working directory. You should take these to be absolute directories rooted at your workspace root `WSR`. If your workspace is at `/home/fred/xyz` then `WSR/classroom` should be read as `/home/fred/xyz/classroom`.

Line 3 is relative to your current working directory (in this example `WSR/classroom`) and is referring to `WSR/course-material` (since the parent of `WSR/classroom` it `WSR` and `course-material` is to be found directly under this directory).

Line 5 is again relative to your current working directory. As you just moved to `WSR` (line 4) this refers to your `classroom` directory under that root.

1.3.2 Regular host workspace activities

There are a number of actions you may want to repeat throughout this course. Rather than repeat them in full each time I present them here and simply refer to these entries as necessary.

1.3.2.1 Checkpoint Classroom

You will need to checkpoint the classroom at least once, when you start the course. If you get lost in the material you can reset your classroom to one of the checkpoints in the book. This will clean up you `classroom` directory ensuring you are ready to proceed with the book's follow-along lessons.

1. Shutdown any running classroom Virtual Machine (VM)[2] (if one is currently set up).

   ```bash
   cd WSR/classroom
   vagrant halt
   ```

2. Backup your current classroom

   ```bash
   cd WSR
   mv classroom archive/classroom_<date>
   ```

[2] A segmented presentation or emulation of a physical computer allowing multiple 'guest' machines to share the physical resources of the 'host' computer.

Replace `<date>` with the date of the backup (I recommend using a `YYYMMDD` format as this sorts properly). For example, if today where December 3rd 2020 and I wanted to backup my classroom I would us the following[3].

```bash
cd WSR
mv classroom archive/classroom_20201203
```

3. Copy the relevant material from `course-material`

```bash
cd WSR
cp course-material/<checkpoint>/ classroom
```

Replacing `<checkpoint>` with the name if the checkpoint from which you want to proceed.

4. Start up the classroom

```bash
cd WSR/classroom
vagrant up
```

This will start up any VM required for the classes.

1.3.2.2 Snapshot Classroom

If you are following the advice given above and 'playing' with your classrooms then I suggest your take a snapshot of your environment just before you start to play. This way you can quickly reset your classroom back you a point where it is ready for you to continue following along with this course.

1. Follow along with this course.

2. Decide to 'play' for a while so take a snapshot.

```bash
cd WSR/classroom
vagrant snapshot save class_<date>
```

Replace `<date>` with the date of the snapshot (I recommend using a `YYMMDD` format as this sorts properly). For example, if today where December 3rd 2020 and I wanted to backup my classroom I would use the following.

[3] Windows users should use `move` rather than `mv`

```bash
cd WSR/classroom
vagrant snapshot save class_201203
```

3. Play with your classroom environment.

4. Decide to resume the course as described in this book.

5. Restore your classroom to the saved snapshot.

   ```bash
   cd WSR/classroom
   vagrant snapshot restore class_<date>
   ```

   Where `<date>` is the date of the snapshot to be restored. For example, to restore the snapshot from December 3rd 2020 we created earlier I would use the following.

   ```bash
   cd WSR/classroom
   vagrant snapshot restore class_201203
   ```

6. Resume course.

One other useful snapshot command is `list`, this can be used to show your previously saved snapshots (useful if, like me, your forget this sort of thing).

```bash
cd WSR/classroom
vagrant snapshot list
```

1.3.2.3 Update material

This book, and consequently the accompanying material, is continually being updated[4]. Most updates will be to the *guest* workspace consequently the host workspace will rarely need updating, the following procedure will update your host workspace and bring your guest systems up to date.

```bash
cd WSR/course-material
git pull
```

[4]If you have downloaded the PDF version of this book then you should download the latest version at the same time you update the course material, otherwise they will get out of sync.

This will update the course material in the host workspace. If you have created a `classroom` then you may need to re-copy the relevant course material and re-provision the virtual machine.

bash

```bash
1 cd WSR
2 cp -rf course-material/<checkpoint>/* classroom
3 cd classroom
4 vagrant up --provision
```

Line 2 copies the relevant checkpoint files (obviously replacing `<checkpoint>` with the actual checkpoint directory you want to use). Line 4 will update any guest virtual machines (even if they already exist or are running).

Chapter 2

Setting Up Your Environment

In order that we are all seeing the same environment as we progress through the following material you will need to install three applications onto your computer (the host computer):

- `VirtualBox`
- `vagrant`
- `git`

Let's take a look at each and discuss why they are required.

2.1 VirtualBox

VirtualBox is Oracle's virtual machine application. This allows us to run a virtual (guest) machine on our host computer. This in turn means that even if you are running, for example, a Windows PC you will be able to run the Linux servers required to follow along with this material.

Virtualisation also isolates our host computer from the machines that we use. This has the advantage that no matter how badly we mess up the virtual environment it will have no effect on our host computer and any change to our host computer will have no effect on the virtual machines[1].

2.2 Vagrant

Vagrant is HashiCorp's command line tool for managing virtual machines. Vagrant provided a simple consistent method for defining virtual machines as code. This means we can all easily set up the same virtual machine environment without the need to rely on following complex set up instructions.

As with many topics covered in this course, there is a more detailed book covering Vagrant *Vagrant from Scratch*[Boo20b].

[1]This is not 100% true, but close enough for our purposes here.

2.3 git

Git has become the *de facto* standard in version control tools. Git is a powerful tool, unfortunately its history means it has a bloated command line interface that is often daunting and confusing to newcomers. Fear not! We will initially use `git` commands to obtain some files and nothing more (so you can just type the commands with no need to understand them) but as we progress we will explain the `git` command line and if you are interested in learning Git in detail there is a complete book on the topic *Git from Scratch*[Boo20a].

2.4 Installing the host tools

I have prepared some brief installation videos but to get the most up-to-date instructions for installing these host tools follow the instructions on their web sites.

2.5 Setup Files for the Course

Having installed the host tools, we now need to set up a virtual machine on which we will do all of the work for this course. This approach removes many problems relating to you and I working in different environments. For example, some of you will have host computers running Microsoft Windows, others will be running MacOS, and others Linux. In addition you will all be running different versions and have different tools installed. By standardising on one environment and one set of tools we remove these differences at the cost of (perhaps) causing some issues with you not having favourite tools available.

To create the virtual class machine:

```bash
mkdir pfs
cd pfs
clone --depth 1
    https://gitlab.com/saltyvagrant.classes/programming
    course-material
cp -rf course-material/pfs010cp001 classroom
cd classroom
vagrant up
```

The `vagrant up` may take a while (several minutes even on a powerful machine) as it sets up the class machine for the first time. (The good news is that this long wait is only necessary when setting up the class virtual machine. Once set up the machine will be much quicker to start.)

Once you are returned to your command prompt you can connect to the class machine using:

```bash
vagrant ssh
```

Assuming all is well you will be connected to the new virtual machine and everything will be set up ready to start the course.

Chapter 3

What this book is about

This book is about learning to program. It is not about learning a particular style of programming, nor about learning a particular language, nor about learning programming for a particular application, nor about a particular framework. It makes no promises that you will "learn programming in ten minutes".

Along the way I will illustrate programming concepts using three languages (Lua, Python, and Bash) and we will look at many examples of code in different styles and in different applications. But all of this is simply to aid the core lessons on learning to program.

3.1 Concepts Over Specifics

As with all the 'From Scratch' series we focus on concepts rather than specifics. Learning concepts is more productive as they can be applied in many situations whereas learning specifics results in limited applicability.

3.2 Why Lua, Python, and Bash?

Lua is a simple language to learn but has all of the features needed to teach the core programming skills. Lua is also a widely used embedded language, so will likely be useful throughout your career.

Python is a wildly popular language. It has many more advanced features and extensive libraries and frameworks which we will investigate in later parts of the course. Python is also commonly used as a scripting language and as as such is popular in DevOps, so this course supports the *Devops from Scratch* material.

Bash is (almost) a *de facto* standard scripting language on Linux. If you are aiming to do DevOps in a Linux environment you will need to know Bash scripting.

3.3 Art, Craft, or Science?

The question if whether programming is an art, a craft, or a science has been around as long as I can recall and will likely persist long after I am dead. It is one of those water cooler conversations that can be both entertaining but frustrating. It can also end friendships.

Ultimately it does not matter which camp you occupy. Just for the record, I hold that *programming* is a craft, but *software engineering* (the development of software systems) is a science.

3.4 Coding, Programming, and Software Engineering

Coding is easy. To write a set if instructions in a particular computer language only requires a knowledge of that language's syntax and semantics.

Programming requires more thought. We need to first consider the problem we are being asked to solve, understand it well enough to formulate a solution, translate that solution into an algorithm, and then to code that algorithm into our chosen computer language. From this description we see that coding is only the final (and arguably the least demanding) step.

Software engineering is a broader discipline requiring an understanding of requirement solicitation, through system design, programming, to monitoring and improving operational systems.

3.5 The Essence of Programming

Programming is the craft of translating requirements into code.

Most programming courses start by introducing basic programming concepts, this course approaches programming from the perspective of translating requirements into functional software, which is more like the approach you will encounter in a professional environment.

Often, as a professional programmer you will operate on 'informal' requirements. Particularly if you are not programming as your primary function. This course will show you how to capture these 'informal' requirements in a more formal way, making them useful for controlling and documenting your projects.

Chapter 4

Core Concepts

There are several concepts that crop up across the design and management of IT that are so universally applicable that it is worth learning them regardless of your specific interests.

This chapter introduces these core concepts.

4.1 Cohesion

Cohesion refers to how closely elements are related to one another. In generally it is desirable to keep related things together and unrelated things separate.

4.2 Coupling

Coupling refers to how tightly to elements depend upon one another. In general we want elements to be as independent as practicable.

4.3 Abstraction

Abstraction is the process of extracting the essential from the incidental.

4.4 Separation of Concerns

Separation of concerns is a general principal that employs abstraction to increased cohesion and reduce coupling. The general idea is for elements to 'mind their own business', performing a well bounded function (or set of factions).

4.5 Scope

Scope is the 'range of applicability' of an element.

4.6 Context

Everything operates in a context. Most of the time in IT the context is well defined.

4.7 Contingency

Probably best summarised as ‘it depends’, contingency is the idea that we often must account for things changing.

4.8 Entropy

‘Things degrade over time’, or perhaps more accurately ‘without concerted effort to prevent it, thing get worse over time’. In software circles this is colloquially known as ‘bit rot’, the idea that without specific work to avoid it a software system’s structure will, over time, become more complex, more difficult to maintain, and more prone to error.

4.9 Parsimony

The ‘KISS’ (Keep It Simple, Stupid) principle. Do not make things more complex than required. The more parts a system has the more opportunity the more things there are to go wrong, so it makes sense to use as few things as possible.

Part I

Coding

In §3.4 I said that coding is perhaps the least important part of programming, amounting to little more than translating an algorithm into the specific computer language we have chosen. Why then is this chapter the first?

Although coding is trivial compared to the rest of the skills required for programming it is still an essential step. In this chapter you will learn the core coding skills you need. I stress 'core' because as you progress in learning to code in particular languages you will find a lot of 'syntactic sugar' that will make life easier but underlying this will be these core ideas.

Chapter 5

The Obligatory Hello World Program

It is a staple of the ‘learning to code’ movement to first write a program that prints the greeting ‘hello world’ to the screen, and who am I to fly in the face of this custom?

Setup to Follow Along

If you have been following along so far your VM will already be in the correct state for this lesson, but to be certain issue the following command.

bash

```bash
cd ~
exercise 01.01
```

Before any code is written someone must have specified what the code should do (the ‘requirements’). This has been done for you in these coding exercises.

Requirements

All programs start with requirements. These are often informal when writing code for private use but the larger the group involved in writing the program the more formally we need to specify our requirements.
Throughout this book we will use an approach generally called Test Driven Development (TDD)[a]. Although many examples used in this book are trivial compared to real-world development (and the use of formal requirements is overkill) this more disciplined approach is worth developing early in your career as adjusting to it later is more difficult (trust me, I took the hard route because the idea of TDD was developed long after I learned to program.)
We will take a more detailed look at requirements in Chapter 10.

[a]A development methodology in which tests are written before the implementation code.

5.1 Hello World in Bash

bash

```bash
cd ~/bash
shellspec
```

The `shellspec` command runs a set of tests to confirm that the requirements have been met (these tests are often called ‘acceptance tests’). Later in this course we will look at these tests and how, when, and why to write our own tests.

shellspec

```
1  Running: /bin/sh [sh]
2  F
3
4  Examples:
5    1) convert shows greeting
6       When run script bin/convert
7
8       1.1) The output should equal Hello World!
9
10             expected: "Hello World!"
11                  got: ""
12
13           # spec/bin/convert_spec.sh:4
14
15      1.2) WARNING: It exits with status non-zero but not found expectation
16
17             status: 2
18
19           # spec/bin/convert_spec.sh:2-5
20
21      1.3) WARNING: There was output to stderr but not found expectation
22
23             stderr: /bin/sh: 0: cannot open bin/convert: No such file
24
25           # spec/bin/convert_spec.sh:2-5
26
27 Finished in 0.04 seconds (user 0.03 seconds, sys 0.01 seconds)
28 1 example, 1 failure
29
30
31 Failure examples / Errors: (Listed here affect your suite's status)
32
33 shellspec spec/bin/convert_spec.sh:2 # 1) convert shows greeting FAILED
```

The `shellspec` tests are currently failing, which should be unsurprising as we have not yet written any code. The output from `shellspec` may be confusing but don't worry too much about it at the moment, the main things to notice are on line 28 (telling us we ran one example and had one failure) and line 23 (telling us that the test was trying to run the script `bin/convert` but it could not since that script does not exist yet).

On line 5 we are told that `shellspec` was trying to confirm that the `convert` script ‘shows a greeting’. Line 8 tells us that we are expecting the output to be ‘Hello World!’.

Let’s now write the code to satisfy our requirement.

bash

```bash
1  mkdir bin
2  touch bin/convert
3  chmod +x bin/convert
```

These commands create the `convert` file and make it an executable (line 3). Now we will edit that file and print out our message.

bash

```bash
vi bin/convert
```

The `vi` editor

We are using the `vi` editor (if you know another editor on Linux feel free to use it instead)—strictly we are using `neovim`, a more advanced version of `vi`. `vi` can be a bit intimidating to new users but is worth learning as it is installed on pretty much all Linux and Unix like systems. For a full introduction to `vi` see [Boo22] but I provide a brief guide in Appendix A.

Add the following, single line, to the `bin/convert` file and save the file.

bin/convert

```bash
echo "Hello World!"
```

Running `bin/convert` will print the message ‘Hello World!’ to the screen.

bash

```bash
bin/convert
```

bin/convert

```
Hello World!
```

If we now run our tests they pass.

bash

```
shellspec
```

shellspec

```
1 Running: /bin/sh [sh]
2 .
3
4 Finished in 0.04 seconds (user 0.04 seconds, sys 0.00 seconds)
5 1 example, 0 failures
```

The only interesting line in this output is line 5 where we are told we ran one example with no failures.

Congratulations! You are now a coder. You have written a simple script that outputs a message to the screen.

Sure, not the most exciting script in the world, but outputting information to the screen is an essential part of many programs.

5.2 Hello World in Lua

bash

```
cd ~/lua
busted
```

The `busted` command runs a set of tests to confirm that the requirements have been met (these tests are often called ‘acceptance tests’). Later in this course we will look at these tests and how, when, and why to write our own tests.

busted

```
1  lua: cannot open bin/convert.lua: No such file or directory
2  -
3  0 successes / 1 failure / 0 errors / 0 pending : 0.0029 seconds
4
5  Failure → spec/convert_spec.lua @ 2
6  convert shows greeting
7  spec/convert_spec.lua:6: Expected objects to be the same.
8  Passed in:
9  (string) ''
10 Expected:
11 (string) 'Hello World!
12 '
```

The busted tests are currently failing, which should be unsurprising as we have not yet written any code. The output from busted may be confusing but don't worry too much about it at the moment, the main things to notice are on line 3 (telling us we had one failure) and line 1 (telling us that the test was trying to run the script bin/convert.lua but it could not since that script does not exist yet).

On line 6 we are told that busted was trying to confirm that the convert script 'shows a greeting'. Lines 7 through 12 tell us that we are expecting the output to be 'Hello World!' (ending with a newline character).

Let's now write the code to satisfy our requirement.

bash

```bash
mkdir bin
touch bin/convert.lua
```

These commands create the convert.lua file (unlike the bash script in §5.1 we don't need to make this script executable as it will be passed to the lua processor as a simple text file—we look at how to treat lua scripts as executables later). Now we will edit that file and print out our message.

bash

```bash
vi bin/convert.lua
```

Add the following, single line, to the bin/convert.lua file and save the file.

bin/convert.lua

```lua
print("Hello World!")
```

Running `bin/convert.lua` will print the message ‘Hello World!’ followed by a new line to the screen.

bash
```bash
lua bin/convert.lua
```

bin/convert.lua
```
Hello World!
```

If we now run our tests they pass.

bash
```bash
busted
```

busted
```
o
1 success / 0 failures / 0 errors / 0 pending : 0.002722 seconds
```

The only interesting line in this output is line 2 where we are told we ran one successful test with no failures.

Congratulations! You have written a simple lua script that outputs a message to the screen.

5.3 Hello World in Python

bash
```bash
cd ~/python
poetry run pytest
```

The `poetry run pytest` command runs a set of tests to confirm that the requirements have been met (these tests are often called ‘acceptance tests’). Later in this course we will look at these tests and how, when, and why to write our own tests.

poetry run pytest

```
1  Creating virtualenv convert-3AT6xhsc-py3.8 in /home/vagrant/.cache/pypoetry/virtualenvs
2  ============================== test session starts ===============================
3  platform linux -- Python 3.9.2, pytest-7.1.2, pluggy-1.0.0
4  rootdir: /home/vagrant/python
5  plugins: bdd-6.0.1
6  collected 1 item
7  
8  tests/test_convert.py F [100%]
9  
10 ==================================== FAILURES ====================================
11 ________________________________ test_welcome_message ____________________________
12 
13 capfd = <_pytest.capture.CaptureFixture object at 0x7f80b8694e80>
14 
15     def test_welcome_message(capfd):
16 >       import convert
17 E       ModuleNotFoundError: No module named 'convert'
18 
19 tests/test_convert.py:3: ModuleNotFoundError
20 ============================= short test summary info ===========================
21 FAILED tests/test_convert.py::test_welcome_message - ModuleNotFoundError: No module named 'convert'
22 =============================== 1 failed in 0.10s ===============================
```

The `poetry run pytest` tests are currently failing, which should be unsurprising as we have not yet written any code. The output from `poetry run pytest` may be confusing but don't worry too much about it at the moment, the main things to notice are on line 22 (telling us we had one failure) and line 21 (telling us that the problem is that the `convert` module does not yet exist).

In that line 21 message we see that the error is in `test_welcome_message`, which gives us a clue as to what we are expected to provide.

Let's now write the code to satisfy our requirement.

bash

```
vi convert.py
```

Add the following, single line, to the `convert.py` file and save the file.

convert.py

```python
print("Hello World!")
```

Running `poetry run python convert.py` will print the message ‘Hello World!’ followed by a new line to the screen[1].

bash

```
poetry run python convert.py
```

poetry run python convert.py

```
Hello World!
```

If we now run our tests they pass.

bash

```
poetry run pytest
```

poetry run pytest

```
1  ============================== test session starts ================================
2  platform linux -- Python 3.9.2, pytest-7.1.2, pluggy-1.0.0
3  rootdir: /home/vagrant/python
4  plugins: bdd-6.0.1
5  collected 1 item
6
7  tests/test_convert.py . [100%]
8
9  =============================== 1 passed in 0.03s ====================================================
```

The only interesting line in this output is line 9 where we are told we ran one passing test.

[1]In general we can run Python scripts using `python convert.py`. We use `poetry run` which runs our script via `poetry` and this takes care of managing the python environment. DON’T PANIC! All of this will be explained later.

Congratulations! You have written a simple python script that outputs a message to the screen.

5.4 Hello World Review

We've seen three different approaches to coding a simple program. Each approach shared similar characteristics.

- We started with a failing test that provided a specification for our program.
- We wrote a program to make the test pass (and thereby ensure our program fulfilled the requirement).
- We reran our tests to make sure we had met the requirement.

You may also notice how similar the programs are to one another. They all amount to a single 'print' statement. Although programming languages vary considerably in detail, you will often find many common features. When learning, try to identify the core ideas as these often translate between languages. 'print' is a good example, many languages share the idea of printing to the computers display and this is often achieved using a 'print' statement of some form.

Does this mean *all* computer languages have a 'print'? No. In Haskell, for example, we might write `putStrLn "Hello World!"` but the result is similar (display a message on the screen). Sometimes we have to be more explicit about where to print the message. In Java we might say `System.out.print("Hello World!")` to print to the screen[2].

[2]Technically we are printing to the console.

Chapter 6

Variables

Any non-trivial program will operate on some data. To keep track of this data within our program we use variables. Each variable is identified in our program by a variable name. In this section we will take our first look at variables.

In Chapter 5 we wrote a simple program to print out a standard greeting 'Hello World!' to the screen. The message was put into a print statement using quotation marks to delimit the message, this is called a string literal.

String literals in a program are a 'code smell' (something that is not necessarily wrong but may be evidence that something is in fact wrong). Consider our hello world program, the message is specified in the code, if I want to change the message I need to find the string literal and change the code. This is not a problem in this program (after all, it has only one line) but think about a larger program with many different messages spread through the code. Say I want to change my interface from English to French. I would need to search through all the program code and translate each literal string. Worse, I now have a French interface but the English messages are all gone. If I want German instead of French? Again, every message literal string must be found and translated. This is obviously not a good solution.

The first step in cleaning up our program is to gather the messages in one location. Having all messages in one place means we do not need to search the code for them. In the following sections we will rework our code to use a variable for the message. Importantly we are not changing the behaviour of the program, we expect all of the tests to continue to pass as we modify the code. This process of modifying code to improve it without changing what it does is called 'refactoring' and is one of the principal reasons for having good tests in place. Having good tests ensures that whenever we refactor the code to improve its readability, improve performance, simplify it, or make it easier to manage, we can do so confident that the passing tests ensure we are not changing what the code is designed to do.

6.1 Python Variables

We ended the last section (§5.3) in the Python version of the 'hello world' program, so let's pick up there.

We want to develop our program to make it easier to maintain by replacing the string literal in our print statement with a variable. We are modifying our code but we have not changed any requirements (a practice called 'refactoring'), so first run our tests to confirm the code is working.

bash

```
poetry run pytest
```

poetry run pytest

```
============================== test session starts ===============================
platform linux -- Python 3.9.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/vagrant/python
plugins: bdd-6.0.1
collected 1 item

tests/test_convert.py .  [100%]

=============================== 1 passed in 0.03s ====================================================
```

Now edit the program.

bash

```
vi convert.py
```

To match the following.

convert.py

```
1  greeting_message = "Hello World!"
2  print(greeting_message)
```

Save the changes.

In line 1 we assign the variable `greeting_message` the same literal string `"Hello World!"` we printed before. Then in line 2 we use the `greeting_message` variable in place of the literal string.

Re-run the tests to confirm we have broken nothing.

```bash
poetry run pytest
```

```
============================== test session starts ===============================
platform linux -- Python 3.9.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/vagrant/python
plugins: bdd-6.0.1
collected 1 item

tests/test_convert.py .                                                  [100%]

=============================== 1 passed in 0.03s ================================================
```

The program is producing the same result, so our change has broken nothing.

We now use a variable (a variable is simply a label on a piece of data) to refer to our greeting message. We will see in the following sections how to use this variable to do something more interesting.

6.2 Lua Variables

Moving on to our Lua example.

```bash
cd ~/lua
busted
```

We should see the tests pass.

```
o
1 success / 0 failures / 0 errors / 0 pending : 0.002722 seconds
```

Now edit the Lua version of our program.

```bash
vi bin/convert.lua
```

Change this program to the following.

bin/convert.lua

```lua
greeting_message = "Hello World!"
print(greeting_message)
```

This is identical to the Python version of our program but don't worry we will see differences later. What this example shows is that different languages share many features, especially with these core coding ideas like variables.

Re-run the tests to confirm we have broken nothing.

bash

```bash
busted
```

busted

```
o
1 success / 0 failures / 0 errors / 0 pending : 0.002722
  seconds
```

6.3 Bash Variables

On to our Bash example.

bash

```bash
cd ~/bash
shellspec
```

shellspec

```
Running: /bin/sh [sh]
.

Finished in 0.04 seconds (user 0.04 seconds, sys 0.00
  seconds)
1 example, 0 failures
```

Everything passing our tests. Edit the program file.

bash

```bash
vi bin/convert
```

Change its content to the following.

bin/convert

```
greeting_message="Hello World!"
echo "${greeting_message}"
```

Wow! Line 2 is very different in Bash. Not only do we use `echo` but the reference to the variable `${greeting_message}` is more complex.

Let's unpack line 2. `echo` is the instruction to output to the screen. Next we have what looks like a string literal (just like when we had `"Hello World!"` in this position) but it's not. This string contains a variable reference `${greeting_message}`. This requires some examination.

In general, in Bash, we *define* variables using their name (label) and refer to the value assigned to a variable by adding the `$` prefix. So, `greeting_message="Hello World!"` *defines* variable `greeting_message` to have the value `"Hello World!"` (the literal string 'Hello World!') and `$greeting_message` recalls the value of `greeting_message`.

In this case, we could have written the program as follows.

bin/convert

```
greeting_message="Hello World!"
echo $greeting_message
```

Try it. If you change the program and run the tests you will find it all still passes.

The `echo` command in Bash treats everything it sees as a string of characters to be output to the screen.

When run, the line `echo $greeting_message` is first expanded by replacing `$greeting_message` with its value `Hello World!`. Notice that the `""` characters are not part of the value of `greeting_message` they are only used to delimit a string (tell Bash that whatever is between them is the value to be assigned to `greeting_message`). After this expansion the line becomes `echo Hello World!` and `echo` takes anything following (`Echo World!`) as the string to be output.

Why then all the additional `"{}"` characters in the original version of the script?

6.3.1 Separating variable names

Suppose I want to print an `X` immediately after the `greeting_message`? I might try `echo $greeting_messageX` but this will fail because we are now referencing a variable called `greeting_messageX`, which is undefined in this program so Bash replaces the reference with nothing and `echo` prints nothing[1]. Having failed, we might try `echo $greeting_message X` but this outputs the `greeting_message`, a space, and then an `X`. Not what we want. Finally, we

[1]Actually it prints a newline with no message.

can use the special syntax surrounding the variable name with `{}`. This allows us to place the `X` immediately after the variable `echo ${greeting_message}X`, getting us what we want (the message `Hello World!X`, with the `X` immediately after our message).

That explains the `{}` characters, but why surround the variable with `""` characters? We do this to prevent two, fairly subtle, causes of bugs in Bash scripts; 'globbing' and 'word splitting'.

6.3.2 Globbing and word splitting

'Globbing' is a feature of Bash that replaces certain characters on a line with entries from the file system. If, for example, you enter the command `echo *` on the Bash command line you will not see a `*` output. Instead you will see a list of files and directories (in fact the same list as you would see by typing `ls .`). This is because of 'globbing'. Before a line is executed (either on the command line or in a script) the `*` is expanded to a space separated list of files and directories that match the wildcard pattern `*`. The problem with this is that if a variable contained one of these globbing patterns then it will be expanded. Try the following at the Bash command line.

```bash
x="*"
echo $x
```

Line 1 assigns a string literal containing one character (`*`) to variable `x`. The output from the echo on line 2 will not be `*` (the value we assigned to `x`) but `bin spec`, the two directories in our current working directory. Bash first expanded `echo $x` to `echo *` and it then matched `*` to the content of the current working directory and expanded the command line to `echo bin spec`. Globbing expansions like this are a source of often subtle and difficult to find bugs in Bash scripts, you can avoid them by simply enclosing all strings (or things you want to treat as a single string) in `""`. On the Bash command line try the following.

```bash
x="*"
echo "$x"
```

This time you see the `*` as intended. Enclosing the variable expansion in `""` prevents Bash from globbing the expanded string.

What about 'word splitting'[2]?

To understand the problem with word splitting we need to understand a bit more about how Bash processes commands. To keep things simple, and relevant to the current example, we will consider the `echo` command.

[2] Also called 'field splitting'.

The echo command can be followed by a white space separated list of fields, it will then print to the screen a line of text constructed from these fields, each field separated by a single space[3]. The 'word splitting' problem arises when outputting variables containing strings with multiple spaces. Try the following on the command line.

bash

```bash
x="magic     disappearing    spaces"
echo $x
```

It does not matter how may spaces are between the words in the string assigned to `x`, so long as there are more than one to demonstrate the splitting issue.

echo $x

```
magic disappearing spaces
```

All the space in the string is reduced to single space. This is because Bash first expands `echo $x` to `echo magic     disappearing    spaces` then outputs each 'field' (in this example the words 'magic', 'disappearing', and 'spaces') separated by a single space.

Now try using quotes.

bash

```bash
echo "${x}"
```

I've also included the `{}` because, although unnecessary here, it is a good habit to develop.

echo "${x}"

```
magic     disappearing    spaces
```

Notice that this time the space in the string is preserved, this is because Bash expands `echo "${x}"` to `echo "magic     disappearing    spaces"` and `echo` sees the literal string as one field so it is printed out literally.

This is 'defensive programming'; programming to avoid problems that may occur if we take a more relaxed approach. Developing the habit of always using the longer form `"${variable}"` you will avoid many painful debugging sessions.

Before moving on make sure `bin/convert` contains the correct code.

[3]This is not a complete description of `echo`, for a complete description refer to the `man echo` page.

bin/convert

```
greeting_message="Hello World!"
echo "${greeting_message}"
```

And remember to run the tests to check your changes!

6.4 Variables Review

In this chapter we have learned that variables are labels by which we refer to data in our program.

Chapter 7

Special Variables

Most programming languages have special variables. These might be referencing data from outside the program, or data from a command line, or even referencing data defined as part of the language itself. We will investigate many of these special variables later, for now we are going to focus on getting data from the command line.

What do we mean 'get information from the command line'?

When a command is entered on the command line it starts with the command (in this case the program we want to run) and is often followed by one or more arguments that changes the way the command works. For example, if we enter `ls` at the bash command prompt we see the list of files and directories in the current directory. If we enter `ls ..` the two dots tell the `ls` command to list the files and directories in the parent directory to the current one. The `..` is a value provided to the `ls` program as a 'command line argument'.

Now we will change our program to use some information from the command line and greet the user by name.

7.1 Command Line Arguments in Bash

We left off in the Bash version of the program, so let's pick up there. We want to be able to enter a name by which the program will greet us. If we supply a name on the command line `bin/convert "Mark"` the program should greet us with `Hello Mark!`. If no name is supplied on the command line we will expect `Hello !` for now (after we learn about controlling a program's flow (Chapter 9) we will improve our code to do something more sensible).

With new requirements we need new tests. Update your environment.

```bash
exercises 01.02
```

Run the tests, which we expect to fail as we have not yet written the code.

```bash
shellspec
```

Sure enough the tests fail.

```shellspec
Running: /bin/sh [sh]
FF

Examples:
  1) convert shows greeting #1
     When run script bin/convert

     1.1) The output should equal Hello !

            expected: "Hello !"
                 got: "Hello World!"

          # spec/bin/convert_spec.sh:8

  2) convert shows greeting #2
     When run script bin/convert Mark

     2.1) The output should equal Hello Mark!

            expected: "Hello Mark!"
                 got: "Hello World!"

          # spec/bin/convert_spec.sh:8

Finished in 0.06 seconds (user 0.05 seconds, sys 0.01
 seconds)
2 examples, 2 failures


Failure examples / Errors: (Listed here affect your
 suite's status)

shellspec spec/bin/convert_spec.sh:6 # 1) convert shows
 greeting #1 FAILED
shellspec spec/bin/convert_spec.sh:6 # 2) convert shows
 greeting #2 FAILED
```

Modify the `bin/convert` program as follows.

bin/convert

```
1  name="${1}"
2  greeting_message="Hello ${name}!"
3  echo "${greeting_message}"
```

Re-run the tests.

The tests now pass. In the program, line 1 assigns the value in variable `1` to a new variable `name`. In a Bash script variables `1`, `2`, through `9` are assigned values from arguments supplied on the command line[1]. Variable `0` is assigned the name of the program being called[2]. If we invoke our script with `bin/convert "Mark"` then `$1` will be assigned the literal string 'Mark'.

Why assign this to variable `name` rather than use it directly? The name `1` does not provide any useful information to someone reading our code. At least not once we use it in our greeting. The only thing `greeting_message="Hello ${1}!"` tells us is that the greeting should contain the first argument from the command line, which, while accurate tells us nothing about the program's intended use for this argument. The variable `name` tells us that the author of the program intends this argument to be a name.

A variable name should tell someone reading your code what data it refers to. That is it in a nutshell. There are several heuristics we can use make effective variable names and we will consider these in detail in Appendix B.

As your scripts become more complex you will find that clear variable names become increasingly necessary.

7.2 Command Line Arguments in Lua

Moving to our Lua example and run the tests.

bash

```
cd ~/lua
busted
```

These tests fail because we updated the exercise setup and we have not yet written the new Lua code.

[1]This is not the complete picture as we shall see in §**??**.
[2]Again, this is not the complete picture but will do for now.

busted

```
--
0 successes / 2 failures / 0 errors / 0 pending : 0.005102 seconds

Failure → spec/convert_spec.lua @ 2
convert shows greeting
spec/convert_spec.lua:6: Expected objects to be the same.
Passed in:
(string) 'Hello World!
'
Expected:
(string) 'Hello !
'

Failure → spec/convert_spec.lua @ 8
convert shows greeting
spec/convert_spec.lua:12: Expected objects to be the same.
Passed in:
(string) 'Hello World!
'
Expected:
(string) 'Hello Mark!
'
```

We fix this by editing the Lua program to contain the following.

bin/convert.lua

```lua
1 name = arg[1]
2 greeting_message = "Hello " .. name .. "!"
3 print(greeting_message)
```

As in our Bash example we are assigning the command line argument to the variable `name`. Unlike Bash, Lua creates a special variable called `arg`. This `arg` variable refers to data in a ‘table’. Briefly (we look at tables in more detail later) a table is like a list of other data where each item in the list can be referred to using a key. In this case the table contains a list of the command line arguments, each argument is referred to by its position on the command line (so the key `1` in this example refers to the first argument passed on the command line).

Line 2 looks very different to the Bash version too. Here we are concatenating (the two dots `..` tell Lua to take the string to the left, make a copy and then append a copy of the string to the right) three strings; the literal string "Hello " (note the space), the string held in `name`, and the literal string `"!"`.

The resulting string is then assigned to the `greeting_message` variable.

Line 3 is unchanged.

7.3 Command Line Arguments in Python

Moving on to the Python example and running the new tests[3].

```bash
cd ~/python
poetry run pytest --tb=no -v
```

Again, these tests will fail because we have new requirements, hence new tests, which require new code.

```
poetry run pytest -tb
============================= test session starts
 ==============================
platform linux -- Python 3.9.2, pytest-7.1.2,
 pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/vagrant/python
plugins: bdd-6.0.1
collected 2 items

tests/test_convert.py::test_welcome_message[-Hello !\n]
 FAILED           [ 50%]
tests/test_convert.py::test_welcome_message[Mark-Hello
 Mark!\n] FAILED   [100%]

=========================== short test summary info
 ============================
FAILED tests/test_convert.py::test_welcome_message[-Hello
 !\n] - AssertionErr...
FAILED
 tests/test_convert.py::test_welcome_message[Mark-Hello
 Mark!\n] - Asse...
============================== 2 failed in 1.37s
 ===============================
```

We can fix these failing tests by changing the program to the following.

[3]I use the `--tb=no -v` flags to produce more compact output for this book. If you want to leave these flags off the result will be the same but with more verbose output.

convert.py

```
1  import sys
2
3  name = sys.argv[1]
4  greeting_message = f"Hello {name}!"
5  print(greeting_message)
```

Python deals with command line arguments a bit differently to Bash and Lua. Rather than built in special variables like `$1` and `argv`, Python relies on a package (a library of code[4]) that is loaded into the script using the `import sys` directive on line 1. Once the `sys` package is loaded we have access to `sys.argv` which works in a similar way to Lua's `argv`; each command line argument is placed into the corresponding `sys.argv` entry (so the first command line argument is placed in `sys.argv[1]`). As with the other versions of this program we set create the `name` variable to give this first argument a more readable name.

Line 4 assigns the greeting string to `greeting_message`. This is a Python 'format' string, which work in a similar way to Bash strings in that the variable in `{}` is expanded into the string[5].

7.4 Special Variables Review

In this chapter we have learned that each language provides certain 'special' variables. These special variables come from various sources but all share the common feature that we do not declare them directly in our program, they 'just appear'.

We will encounter more of these special variables as we progress.

[4]We discuss these packages when we cover organising code.

[5]This is doing Python format strings a disservice, they are more powerful than this suggests but for now this simplification will serve.

Chapter 8

Functions

One of the core concepts in IT is abstraction (§4.3). This chapter introduces one of the most basic forms of abstraction; the function.

A variable is a label that allows us to refer to data in our program, a function is a label that allows us to refer to one or more program statements in our program. As with variables we use the names of functions to make it clear what the enclosed statements do.

Functions have another neat trick, we can pass data to the function for it to act on (and get information back).

8.1 Parameters and Arguments

Some terminology. When we define a function we can specify parameters that will become variables in the function's code. When we use a function we can provide the data to be assigned to these parameters.

For example, suppose we define a function as follows.

Simple function

```
1  def add (a,b):
2    return a+b
```

This says, define a function labelled `add` that has two parameters `a` and `b`. When called this function expects two values to be provided, these will be assigned to variables `a` and `b` and the statements in the function (in this case the single `return` statement on line 2) will be executed.

We call this function in our code as follows.

Calling the add function

```
1  add(1,2)
```

This will call the `add` function and assign `1` to `a` and `2` to `b`. The `return` statement is run and the function returns `a+b` (so, 3 in this example).

It is important to understand that the code inside the function is not run when defining the function. It it only run when the function is called and values are provided to `a` and `b`.

Once defined a function can be called as often as we like.

Repeated calls

```
add(1,2)
add(15,3)
add(5.3,6.2)
```

These return 3, 18, and 11.5 respectively.

8.2 Functions in Python

We left variables (Chapter 7) in the Python example so let's pick up there by adding a function.

First, our test should be passing so check that now.

bash

```bash
poetry run pytest
```

Once we have passing tests we are ready to refactor our code to introduce a function (remember that during refactoring we are aiming to improve our code without changing what it does; in other words we don't need new tests, but all existing tests must continue to pass).

Modify the `convert.py` code as follows.

convert.py

```python
import sys

name = sys.argv[1]
greeting_message = f"Hello {name}!"


def display_greeting(greeting):
    print(greeting)


display_greeting(greeting_message)
```

Once you have saved these changes, rerun the tests to make sure we have broken nothing.

```bash
poetry run pytest
```

What has this change achieved?

Firstly, and most obviously, it shows the basic form of a function in Python. On line 7 we use the `def` keyword[1] to tell Python we are about to declare a function. Next we tell Python that we want to label this function `display_greeting` and this is followed by a list of parameters this function expects (in this case we expect just one and we tell Python that any value provided for this parameter should be labelled `greeting`). The function declaration is ended with a colon character.

Following the function declaration we provide the code that is to be labelled `display_greeting`. In Python the function's code is indented with respect to the declaration. In this example the function contains a single line of code, line 8. This is just the `print` statement from our previous version of the `convert.py` program but we have replaced the `greeting_message` variable with the parameter label `greeting`.

So that's the function defined. Defining the function will not print any message though, to do this we must 'call' the function, which is what we do on line 11. When we call the `display_greeting` function we 'pass' the content of variable `greeting_message` as the argument (in Python arguments appear as a comma separated list between parentheses immediately after the name of the function we want to call). This argument value is assigned to the parameter `greeting` (just as we might assign a variable, in fact when `display_greeting` is executed the parameter `greeting` *is* a variable within the function code). The code within the function is run and the `print` statement outputs the value in variable `greeting`.

That may seem like a lot of work and all we seem to have done is make our code longer and more complex. However we have added some useful information, specifically that the `print` statement is not simply a generic 'output this string' but in fact is more specifically displaying a greeting (as indicated by our function's name `display_greeting`). In a short program like this example this may seem unnecessary but trust me this sort of clarity in our code will be vital later.

We can go further. We are currently creating our message using a format string on line 4, which is fine but we can do better. What is this line really doing within our program? It is 'composing' the greeting, so let's make this clear by putting it into a function. Once again, edit `convert.py` to contain the following.

[1]A 'keyword' is a reserved sequence of characters that has special meaning in the programming language.

convert.py

```python
import sys

name = sys.argv[1]


def compose_greeting(name):
    return f"Hello {name}!"


def display_greeting(greeting):
    print(greeting)


greeting_message = compose_greeting(name)
display_greeting(greeting_message)
```

Examples

We are working with examples that emphasis particular coding features! We will be learning how to write better code when we start 'programming'.

As always, rerun the tests to ensure we have broken nothing.

bash

```bash
poetry run pytest
```

The new `compose_greeting` function shows how Python functions can return values using the `return` keyword. When `compose_greeting` is called it will return a value (the greeting string), behaving a little like a variable.

There is one more thing we could do to tidy up our program using functions. Edit `convert.py` as follows.

convert.py

```python
import sys


def compose_greeting(name):
    return f"Hello {name}!"


def display_greeting(greeting):
    print(greeting)


def main(command_line_arguments):
    name = command_line_arguments[1]
    greeting_message = compose_greeting(name)
    display_greeting(greeting_message)


main(sys.argv)
```

Hopefully you are now getting used to this next step; rerun the tests to ensure we have broken nothing.

bash

```bash
poetry run pytest
```

The new `main` function gathers together all of the code for our new program. The name `main` is a common convention across programming languages, so if you are looking at unfamiliar code for the first time looking for `main` is a good starting point[2].

[2] Although common the use of `main` is not universal.

Chapter 9

Flow Control

Part II

Programming

Intro to progrmming.

Chapter 10

Requirements

Appendix A

`vi`

This is not a book on using `vi` (refer to [Boo22] for a detailed introduction) but for convenience this appendix offers sufficient instruction for basic editing.

Appendix B

Naming Things

Choosing variable names

There is a huge amount of advice, guidance, and even prescription, about naming variables. Much of it contradictory. At the risk of adding to this cacophony here is my advice.
In a professional environment you will often be required to follow a project coding standard and this will often mandate a particular approach to naming variables (you must abide by these rules for the sake of consistency within the project—even if these rules seem stupid).
Beyond simply choosing a good name you may need to create the name in a particular format. Common naming formats include:

lowercase Use only lowercase letters. Words with the variable name may be separated by an underscore.

UPPERCASE Use only uppercase letters. Words with the variable name may be separated by an underscore.

CamelCase Start each word with a capital letter, otherwise use lower case.

These various formats may be used to represent different sorts of data. For example, UPPERCASE for constant values, CamelCase for classes, lowercase for local variables, and so on.
There is some merit to this practice, but with modern Integrated Development Environment (IDE)[a] the advantages are slight. That said, don't swim upstream, some of these conventions are so widely used (like CamelCase for class names) that it seems obtuse to ignore them.
Some naming standards include additional information, such as the type of the data referred to; end variable with `_i` for integer values, `_s` if the data is a string.
In my opinion adding information like variable type is almost always a problem in the long-term. Avoid this practice.

[a]A tool used by developers to simplify navigating complex software.

Bibliography

[Boo20a] Mark Bools. *Git from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/git/git.html`.

[Boo20b] Mark Bools. *Vagrant from Scratch*. From Scratch. 2020. URL: `https://saltyvagrant.com/members/books/vagrant/vagrant.html`.

[Boo22] Mark Bools. *NeoViM from Scratch*. From Scratch. 2022. URL: `https://saltyvagrant.com/members/books/neovim/neovim.html`.

Index