Introduction

> *"Codes are a puzzle. A game, just like any other game." - Alan Turing*

I believe you can learn serious things through playing. This is a book of programming exercises that explores programming ideas by coding puzzles and games. Each chapter includes a description of a program you should write with examples of how the program should work. Most importantly, each program includes tests so that you know if your program is working.

When you are done with this books you be able to:

- Write command-line Python programs
- Process a variety of command-line arguments, options, and flags
- Write and run tests for your programs and functions
- Manipulate of Python data structures including strings, lists, tuples, sets, dictionaries
- Use higher-order functions like `map` and `filter`
- Write and use regular expressions
- Read, parse, and write text files
- Use and control randomness

The most fundamental idea I teach is how to test your code. How can you be sure that a function or program does what you think it does? While I will introduce guidelines for writing good code as well as tools you can use to format your code and find errors, none of these is a replacement for *testing* your code. Each exercise includes a test suite that you will run repeatedly as you write your programs. In this way, I'm using ideas from the "test-driven development" where tests are written *first* and then the programs that satisfy those tests come afterwards.

In every book, you get a dose of the author's personal biases, and this one is no different. As a matter of personal preference, I avoid writing object-oriented code and instead encourage you to break problems into small, transparent functions. I believe this leads to code that is easier to teach and test. While Python itself is an object-oriented (OO) language, it has good support for a "functional" approach to development and testing. There is so much you can do with Python's basic data types and structures like strings, numbers, lists, tuples, dictionaries. I personally never write in an OO style, preferring instead to adopt ideas from functional programming (FP) languages like Haskell.

The programming techniques in each exercise are not specific to Python. Most every language has variables, loops, functions, and more languages are incorporating FP ideas from Java to JavaScript. After you write your solutions in Python, I would encourage you to write solutions in every other language you know and compare what parts of a different language make it easier or harder to write your programs. If your programs support the same command-line options, you can even use the included tests to verify those programs!

Why Write Python?

Python is an excellent, general-purpose programming language. You can use it to write command-line programs, web servers, adventure games, and business applications. There are Python modules to help you wrangle complex scientific data, explore machine learning algorithms, and generate publication-ready graphics.

Many college-level computer science programs have moved away from languages like C and Java to Python as their introductory language because Python is a relatively easy language to learn. Often Python codes reads like English — the statement `for item in cart` is a way to visit each `item` in the collection called `cart`. While languages like C, Java, Perl, and PHP use semi-colons (`;`) to mark the ends of statements, Python uses a newline (when you hit the `Enter` key). These same languages also tend to group statements into "blocks" by enclosing them in curly brackets (`{}`), but Python groups statements by how many spaces they are indented (usually four). Many people feel that Python's syntax and lack of excessive punctuation makes it more readable than other languages.

I believe you can teach some very fundamental and powerful ideas from computer science using Python. As I show you ideas like regular expressions and higher-order functions, I hope it piques your interest in other languages and encourages you to study further.

Who Am I?

My name is Ken Youens-Clark. I work as a Senior Scientific Programmer at the University of Arizona. Most of my career has been spent working in bioinformatics using computer science ideas to study biological data. I began my undergraduate degree as a Jazz Studies on the drumset at the University of North Texas in 1990. I changed my major a few times and eventually ended up with a BA in English literature in 1995. I didn't really have a plan for my career, but I did like computers. Around 1995, I stared tinkering with databases and HTML at at first job out of college, building the company's mailing list and first website. I was definitely hooked! After that, I managed to learned VisualBasic on Windows 3.1 and, through the next few years, I programmed in several languages and companies before landing in a bioinformatics group at Cold Spring Harbor Laboratory in 2001 led by Lincoln Stein, an early advocate for open software, data, and science. In 2015 I moved to Tucson, AZ, to work at the University of Arizona where I finished my MS in Biosystems Engineering in 2019.

I was a Perl hacker for longest portion of my coding career, from 1998 to around 2016. Perl ruled the early days of the Internet and the world of genomics and bioinformatics. Over time I've also worked with bash, Python, JavaScript, Haskell, Rust, Elm, Prolog, Ruby, R, Julia, SPSS, and anything else that seems interesting. The programming style that I show blends ideas from many of these languages, and Perl's motto of "There Is More Than One Way To Do It" (TIMTOWTDI) is perhaps why I like to show so many different ways to solve problems.

When I'm not coding, I like playing music, riding bikes, cooking, reading, and being with my wife and children.

Who Are You?

> *"You don't know the power of the command line!" — Darth Vadar (probably)*

I think my ideal reader is someone who's been trying to learn to code well but isn't quite sure how to level up. Perhaps you are someone who's been playing with Python or some other language that has a similar syntax like Java(Script) or Perl. Or maybe you've cut your teeth on something like Haskell or Scheme and you're wondering how does it go in Python Land? You are looking for interesting challenges with enough structure to help you know when you're moving in the right direction. Maybe you're interested in "test-driven development" and want to know how to write and use tests?

This is a book that will try to teach you well-structured, documented, testable code in Python. The material introduces best-practices from industry such as test-driven development. It encourages you to read documentation and Python Enhancement Proposals (PEPs) to write idiomatic code that other Python programmers would immediately recognize and understand. This book shows you why you'd want to learn about regular expressions and algorithm design and statistics and random events. This book uses simple puzzles and games you already understand and asks you to teach the rules to Python.

This is probably not an ideal book for the absolute beginning programmer. I assume no prior knowledge of the Python language specifically because I'm thinking of someone who is coming from another language. If you've never a program an *any* language at all, you might do well to come back to this material when you are comfortable with ideas like variables and loops.

Something that makes this book very different is that we write only *parameterized, command-line programs* because they are much easier to test than, say, interactive programs, web servers, or Jupyter Notebooks. This means you'll will need to access to a command-line interface. Not to fear! Programs like Microsoft's VSCode or Anaconda's Spyder can help! I hope that you'll learn the power of the command.

Why Did I Write This Book?

> *"The only way to learn a new programming language is by writing programs in it." - Dennis Ritchie*

Over the years, I've had many opportunties to help people learn programming, and I always find it rewarding. The structure of this book comes from my own experience in the classroom where I think formal specifications and tests can be useful aids in learning how to break a program into smaller problems that need to be solved to create the whole program.

The biggest barrier to entry I've found when I'm learning a new language is that small concepts of the language are usually presented outside of any useful context. Yes, we all love to write "HELLO, WORLD!", but after that, I usually struggle to write a complete program that will accept some arguments and do something *useful*. For instance, how can I get the name of a file from the user, then read the file and do something with the contents, all the while handling the various errors that arise like not getting an argument from the user, the argument isn't actually a file, the file isn't readable, etc.? This book teaches you exactly how to write Python that accepts and validates arguments, presents usage messages, handles errors, and runs to completion.

More than anything, I think you need to practice. It's like the old joke: "What's the way to Carnegie Hall? Practice, practice, practice." These coding challenges are short enough that you could probably finish each in a few hours to days. This is more material than I could work through in a semester-long university-level class, so I imagine the whole book would take you several months, perhaps even a year or more. I hope you will solve the problems, then think about them and come back later to see if you can solve them differently, maybe using a more advanced technique or making them run faster.

Using test-driven development

"Test-driven development" is described by Kent Beck in his 2002 book by that title as a method to create more reliable programs. The basic idea is that we write tests even before we write code. We run the tests and verify that our code fails. Then we write the code to make each test pass. We always run all the tests so that, as we fix new tests, we ensure we don't break tests that were passing before. When all the tests pass, we have at least some assurances that the code we've written conforms to some manner of specification.

Each program you are asked to write comes with tests that will tell you when the code is working acceptably. The first test in every exercise is whether the expected program exists. The second test checks if the program returns something like a "usage" statement when run with `-h` or `--help`.

After that, your program will be exercised with various inputs and options. You'll fail a test, and fix it. Then run the tests again. You will probably fail the next test. Fix that test such that you don't break the previously passing one. Keep fixing each test until they all pass. Then you are done.

It doesn't matter if you solved the problem the same way as in the solution I provide. All that matters is that you solve it *on your own*!

Suggestions for writing

As a general rule, I try to write many very small functions and lots of tests. Every function should be as short as possible, but no shorter — generally 50 lines is my upper limit on function length. I have no lower bound, though. Even if a function is just one line of code, I like to give it a name and some tests to ensure it does what I think. This is often called "unit testing" in the world of software engineering, and it's a very good practice. As the challenges get harder, I start suggesting specific functions and tests you should write in your programs. If we know the smaller bits work individually, then we ought to have confidence that they will work in concert.

This is the basic idea of "compositionality" where we try to compose large systems from smaller ones, so I also write tests to check that my programs as a whole do what they should. This is called "integration testing" and is found in the `test.py` programs I've provided for you in each of these exercises where your programs are run with various options and checked that they produce the expected output.

So, in short, always start a program the same way. Always process the arguments the same way. Always have the same structure to your program (e.g., start with `main`). When you think you need a function in your program, write the tests first and then write the code that will pass the tests. Then integrate your function into the greater whole. Write a test suite outside your program that checks that it fails properly as well as works properly. Whenever you make a change to your program, run your test suite to see if you accidentally broke something that used to work!

Lastly, you should always use a source code manager. If you fork and clone the GitHub repo of the exercises, then you've made an important first step in learning how to keep track of your code. You don't have to use `git` or GitHub. Mercurial is fine, too. (It's also written in Python!) Find a tool that works for you, and commit to tracking your changes.

Setting up your environment

To write the programs and run the tests, you will need to install Python and have some sort of command-line access. Windows users will probably need to to install Windows Subsystem for Linux or Cygwin in order to get a command line interface suitable for running the programs and the test suite. On a Mac, the default `Terminal` app is sufficient.

I wrote and tested the programs with the Python version 3.7 but should work with any version 3.6 or newer. Python 2 reached end-of-life at the end of 2019 and should no longer be used. To see what version of Python you have installed, type `python3 --version`. If you need to install Python, see https://www.python.org/downloads/.

You will need to use a *text editor* to write code — something that writes *plain text*. Something like Microsoft Word will not work. You author prefers `vim`, but you might like to try Notepad, Sublime, TextWrangler, or Textmate. Some people like to use an integrated development environment (IDE) that combines text editors with tools to help inspect, debug, and run code. Examples include Xcode, PyCharm, Anaconda Spyder, or Microsoft VSCode. Whatever works for you!

I demonstrate how to run the programs and tests from the command line. Each exercise includes a `Makefile` that allows you to type `make test` in the directory to run the tests. This is a shortcut to execute `pytest -xv test.py` which is how the tests are actually run. If you do not have or if you do not wish to use `make`, you can execute this command yourself to run the tests.

The command line and the Python REPL

I will show commands you can execute on the command line preceeded with the `$` (dollar sign) and using a `fixed-width font`. The `$` is a common command-line prompt, but your command line may differ. For instance, I might use the command `cat` (for "concatenate") to show you to contents of a file:

```
$ cat inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

If you want to run that command, *do not copy* the leading `$`, only the text that follows, otherwise you'll probably get an error like `$: command not found`.

Python has a really excellent tool that allows you to interact directly with the language to try out ideas. If you type `python3` on the command line, you will enter a REPL (read-evaluate-print-loop, often pronounced like "repple" in a way that sort of rhymes with "pebble"). These examples will be also shown using a `fixed-width font` with the text preceeded by the prompt `>>>`. Here is what it looks like on my system when I start the Python REPL:

```
$ python3
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can type Python statements like `x = 10` to assign the value 10 to the variable `x`:

```
>>> x = 10
```

As with the command-line prompt `$`, do not copy the leading `>>>` or Python will complain:

```
>>> >>> x = 10
  File "<stdin>", line 1
    >>> x = 10
     ^
SyntaxError: invalid syntax
```

The `ipython` REPL has a magical `%paste` mode that removes the leading `>>>` prompts so that you can copy and paste all the code examples:

```
$ ipython
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.11.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: >>> x = 10

In [2]: x
Out[2]: 10
```

You might also enjoy using Jupyter Notebooks as a REPL. Whichever way you choose to interact with Python, I suggest you manually type all the code yourself as this builds muscle memory and forces you to interact with the syntax of the language.

Code formatting and linting

Every program included has been automatically formatted with `yapf` (Yet Another Python Formatter, https://github.com/google/yapf), a tool from Google that can be customized with a configuration file. Another popular formatter is `black` (https://github.com/psf/black). I encourage you to adopt and regularly use a formatter *after every modification to your program*.

I would also encourage you to look at code "linters" like `pylint` (https://www.pylint.org/) and `flake8` (http://flake8.pycqa.org/en/latest/) to find potential errors in your code that the Python interpreter itself will not compalain about. The `mypy` tool (http://mypy-lang.org/) will also be very helpful as we introduce type hints.

You can install all of these tools using Python's `pip` module:

```
$ python3 -m pip install yapf black pylint flake8 mypy
```

Getting the code

All the tests and solutions are available at https://github.com/kyclark/tiny_python_projects. If you like, you can simply `git clone` that repo and work out the solutions.

If you would like to make a copy of the code, you can "fork" the repository. This would allow you to commit and push your solutions to your own repo. To fork the repo:

1. Create an account on GitHub.com
2. Go to https://github.com/kyclark/tiny_python_projects
3. Click the "Fork" button to make a copy of the repository into your account.

I may update the repo on occasion. If you would like to be able to pull my changes, you should set mine as an "upstream" repo. Inside your local Git repo, execute this command:

```
$ git remote add upstream https://github.com/kyclark/tiny_python_projects.git
```

After that, use `git pull upstream master` to get updates.

Starting new programs with `new.py` or `template.py`

I wrote a program called `new.py` that will help you create new Python programs with boilerplate code that will be expected of all the programs. That is, every program is expected to respond to the `-h` and `--help` flags and produce documentation on program parameters. The generated program is over 70 lines of a well-structured and documented program that you can modify to suite your needs.

You will find `new.py` the `bin` directory of the GitHub repository. The program expects a single argument which is the name of the new program to create. It will create the new program in the directory where you execute the command. The programs you write need to exist in the same directory as their tests, so when you are ready to work on the `crowsnest` exercise you should first change to that directory:

```
$ cd ~/tiny_python_projects/crowsnest
```

You can then execute the `new.py` using an absolute path to the program:

```
$ ~/tiny_python_projects/bin/new.py crowsnest.py
Done, see new script "crowsnest.py."
```

Or you can use a *relative* path from the current directory:

```
$ ../bin/new.py crowsnest.py
Done, see new script "crowsnest.py."
```

Alternately, you may need to invoke `python3` directly:

```
$ python3 ../bin/new.py crowsnest.py
Done, see new script "crowsnest.py."
```

If you wish, you can copy `new.py` to a location in your `$PATH`, or you might prefer to alter your `$PATH` variable to include the directory where `new.py` lives.

If you don't want to use `new.py`, you can also copy `template/template.py` to start your new programs:

```
$ cd ~/tiny_python_projects/crowsnest
$ cp ../template/template.py crowsnest.py
```

Both methods are meant to save you time and make it easier to write programs that use `argparse` to handle the program's parameters.

Why Not Notebooks?

Notebooks are great for an interactive and visual exploration of data, but the downsides:

- Stored as JSON not line-oriented text, so no good `diff` tools
- Not easily shared
- Too easy to run cells out of order
- Hard to test
- No way to pass in arguments

I believe you can better learn how to create testable, *reproducible* software by writing command-line programs that always run from beginning to end and have a test suite. It's difficult to achieve that with Notebooks, but I do encourage you to explore Notebooks on your own.

A Note about the lingo

Often in programming books you will see "foobar" used in examples. The word has no real meaning, but its origin probably comes from the military acronym "FUBAR" (Fouled Up Beyond All Recognition). If I use "foobar" in an example, it's because I don't want to talk about any specific thing in the universe, just the idea of a string of characters. If I need a list

of items, usually the first item will be "foo," the next will be "bar." After that, convention uses "baz" and "quux," again because they mean nothing at all. Don’t get hung up on "foobar." It’s just a shorthand placeholder for something that could be more interesting later.

We also tend to call errors code "bugs." This comes from the days of computing before the invention of transitors. Early machines used vacuum tubes, and the heat from the machines would attract actual bugs like moths that could cause short circuits. The "operators" (the people running the machines) would have to hunt through the machinery to find and remove the bugs, hence the term to "debug."

Last updated 2020-01-15 14:06:57 -0700