The Rhodes Local Edition (RLE) (Version of August, 2012)

By Peter Wentworth

A word of thanks ...

We switched from Java to Python in our introductory courses in 2010. So far we think the results look positive. More time will tell.

This predecessor to this book was a great starting point for us, especially because of the liberal permission to change things. Having our own in-house course notes or handouts allows us to adapt and stay fresh, rearrange, see what works, and it gives us agility. We can also ensure that every student in the course gets a copy of the handouts — something that doesn't always happen if we prescribe costly textbooks.

Many thanks to all the contributors and the authors for making their hard work available to the Python community and to our students.

A colleague and friend, Peter Warren, once made the remark that learning introductory programming is as much about the environment as it is about the programming language.

I'm a big fan of IDEs (Integrated Development Environments). I want help to be integrated into my editor, as a first-class citizen, available at the press of a button. I want syntax highlighting. I want immediate syntax checking, and sensible autocompletion. I'd like an editor that can fold function bodies or regions of code away, because it promotes and encourages how we build mental abstractions.

I'm especially keen on having a single-stepping debugger and breakpoints with code inspection built in. We're trying to build a conceptual model of program execution in the student's mind, so I find most helpful for teaching to have the call stack and variables explicitly visible, and to be able to immediately inspect the result of executing a statement.

My philosophy, then, is not to look for a language to teach, but to look for a combination of IDE and language that are packaged together, and evaluated as a whole.

I've made some quite deep changes to the original book to reflect this (and various other opinionated views that I hold), and I have no doubt that more changes will follow as we mature our course.

Here are some of the key things I've approached differently:

- Our local situation demands that we have a large number of service course students in an introductory course of just 3 weeks, and then we get another semester of teaching with those going into our mainstream program. So the book is in two parts: we'll do the first five chapters in the big "get your toes wet" course, and the rest of the material in a separate semester.
- We're using Python 3. It is cleaner, more object oriented, and has fewer ad-hoc irregularities than earlier versions of Python.
- We're using PyScripter as our IDE, on Windows. And it is hardwired into parts of these notes, with screenshots, etc.
- I've dropped GASP.
- For graphics we start with the Turtle module. As things move along, we use PyGame for more advanced graphics.
- I've introduced some event-driven programming using the turtle.
- I have tried to push more object-oriented notions earlier, without asking students to synthesize objects or write their own classes. So, for example, in the chapter about the turtle, we create multiple instances of turtles, talk about their attributes and state (color, position, etc), and we favour method-call style to move them around, i.e. `tess.forward(100)`. Similarly, when we use random numbers, we avoid the "hidden singleton generator" in the random module — we prefer to create an instance of a generator, and invoke methods on the instance.
- The ease of constructing lists and the `for` loop seem to be winners in Python, so rather than use the traditional command-line `input` for data, I've favoured using loops and lists right up front, like

this:

```
friends = ["Zoe", "Joe", "Bill"]
for f in friends:
    invitation = "Hi " + f + ".  Please come to my party on Saturday!"
    print(invitation)
```

This also means that I bumped `range` up for early exposure. I envisage that over time we'll see more opportunities to exploit "early lists, early iteration" in its most simple form.

- I dumped `doctest`: it is too quirky for my liking. For example, it fails a test if the spacing between list elements is not precisely the same as the output string, or if Python prints a string with single quotes, but you wrote up the test case with double quotes. Cases like this also confused students (and instructors) quite badly:

```
def addlist(xs):
    """
    >>> xs = [2,3,4]
    >>> addlist(xs)
    9
    """
    return
```

If you can explain the difference in scope rules and lifetimes between the parameter `xs` and the doctest variable `xs` elegantly, please let me know. Yes, I know doctest creates its own scope behind our back, but it is exactly this black magic that we're trying to avoid. From the usual indentation rules, also looks like the doctests are nested inside the function scope, but they are not. Students thought that the parameter had been given its value by the assignment to `xs` in the doctest!

I also think that keeping the test suite separate from the functions under test leads to a cleaner relationship between caller and callee, and gives a better chance of getting argument passing / parameter concepts taught accurately.

There is a good unit testing module in Python, (and PyScripter offers integrated support for it, and automated generation of skeleton test modules), but it looked too advanced for beginners, because it requires multi-module concepts.

So I've favoured my own test scaffolding in Chapter 6 (about 10 lines of code) that the students must insert into whatever file they're working on.

- I've played down command-line input / process / output where possible. Many of our students have never seen a command-line shell, and it is arguably quite intimidating.
- We've gone back to a more "classic / static" approach to writing our own classes and objects. Python (in company with languages like Javascript, Ruby, Perl, PHP, etc.) don't really emphasize notions of "sealed" classes or "private" members, or even "sealed instances".

  So one teaching approach is to allocate each instance as an empty container, and subsequently allow the external clients of the class to poke new members (methods or attributes) into different instances as they wish to. It is a very dynamic approach, but perhaps not one that encourages thinking in abstractions, layers, contracts, decoupling, etc. It might even be the kind of thing that one could write one of those *"x,y,z ... considered harmful"* papers about.

  In our more conservative approach, we put an initializer into every class, we determine at object instantiation time what members we want, and we initialize the instances from within the class. So we've moved closer in philosophy to C# / Java on this one.
- We're moving towards introducing more algorithms earlier into the course. Python is an efficient teaching language — we can make fast progress. But the gains we make there we'd like to invest into deeper problem solving, and more complex algorithms with the basics, rather than cover

"more Python features". Some of these changes have started to find their way in this version, and I'm sure we'll see more in future.

- We're interested in issues around teaching and learning. Some research indicates that "intellectual playfulness" is important. The study referenced in the Odds-and-ends workbook at the end just didn't seem to have anywhere sensible to go in the book, yet I wanted it included. It is quite likely that we'll allow more issues like this to creep into the book, to try to make it more than just about programming in Python.