Transitional Python

A Handbook for Students

Jason M. Pittman

Contents

Preface

This book was born out of the need to help students transition from core programming coursework in another language, chiefly C++. In my opinion, when you're coming from another language, you probably only need to be aware of where and how Python is different. That is, a long form text repeating in-common concepts, structures, and idioms is largely wasted trees.

> *This is an example of how I will express a side or meta-thought that I feel is important but not directly related to the point in the main text.*

Thus, you should think of this book as an organized collection of lecture notes. What I am providing here is not a complete treatment of how to program using Python. Rather, I am providing a synthesis or summary. Thus, I do not spend much time on common idioms you can find amongst languages. Instead, I attempt to draw attention to Python's syntax and semantics which differ from C or C++. However, you undoubtedly will need to fill in some gaps with external sources. My recommendations are as follows.

First, given that you have programming experience, try to leverage the concepts you know to draft what you think the same code may look like in Python. Then, look up the specific function in the Python documentation. As a last resort, use something like stackexchange.com or tutorialspoints.com to review specific examples.

Second, if for some reason you do not have programming experience, try to draft in English what you think the code ought to do. Then, look for examples by searching for the nouns or verbs in your English draft in conjunction with python. You can learn quite a bit about programming in general and Python in specific by looking at code examples.

Speaking of organization, this book has five chapters. You can read them in order or skip around based on chapter headings. Unlike a traditional programming text, the content doesn't really have a priority because we are focused on what is necessary to facilitate a transition. Lastly, one of the things I detest in programming books is the lengthy blocks of text, followed by formatted code example, followed by more blocky text. I don't find such presentation of programming material compelling or conducive to learning. Accordingly, you will find code examples centrally located (just like this paragraph you're reading now) with a meta-discussion for the code in an offset, right justified text block.

A parting note about the meta-explanatory, offset text. Read it. Seriously, read it. Also, when a question appears you should answer it. Even better, read the question aloud and verbalize your answer. Trust me.

Chapter 1

Tools

The place to start any programming course is with tools. In other words, what exactly do we need to get started? With Python, the answer is, very little. In fact, we only need two things regardless of what operating system we are using.

Python interpreter

First, we need the Python language interpreter. I strongly recommend grabbing the latest stable version of Python 3. Pay attention here to what operating system you have as the interpreter needs to match. However, don't worry about missing features or the like. One benefit of using a language like Python is the interpreter is operationally the same across all platforms.

Speaking of which, we should clarify how the interpreter works real quick. The Python interpreter compiles our source to bytecode and then executes the program in the Python virtual runtime environment. If you see a file with the ".pyc" extension, that is a compiled version of the source. With Python 3, these bytecode files are in the ***pycache*** directory.

The salient point here is the bytecode is not an equivalent to a compiled C or C++ binary. In fact, the bytecode is more similar to high-level languages such as Java or C# (.NET). That said, with Python, the compilation to bytecode has the benefit of speeding up subsequent runs of the program.

> *What do you think would happen if you executed the .pyc bytecode file directly? Try it and see!*

Further, the bytecode intermediary also has the benefit of making our Python programs truly cross platform. As long as a computing system has the interpreter installed, our program will execute exactly as intended.

Of course, this means we need the Python interpreter installed on *every* system hosting our program. Further, the interpreter needs to be at least of the same mainline version. This doesn't seem so strange given our experience with various C/C++ compilers but can be different for our understanding of runtime environments.

Text editor

Second, we need a text editor. Any text editor will do. Some people like modern text editors such as Sublime Text, Atom, or Visual Studio Code. Others prefer traditional editors such as vi, or emacs. Some may even opt for a full Integrated Development Environment

Honestly, what you use doesn't matter. My advice is to think about what other needs you may have and try to use a single solution to address as many of those needs as possible. Personally, I don't like software sprawl and prefer applications that serve more than one purpose. I also personally like having the features I need and nothing more or less.

> *This is why I use vim with a customized configuration. The software is lightweight, readily extended to fit precisely the feature set I require, and is useful in a variety of situations beyond programming in Python.*

Chapter 2

Syntax

Our second chapter is all about syntax. We'll start with what must be the single largest shift for most students- scoping. Then, we'll take a look at variables. While we discuss *types* in Chapter 3, there are some general transitions with variables which we ought to discuss. The same is true for expressions. Lastly, we will look at how Python handles list comprehensions and some interesting differences in standard libraries.

Scoping

Without a doubt, scoping or defining scope in Python presents the hardest syntatical transition. Let's look at the foundational example, *Hello, World.* In C or C++, we code this as follows.

```
void main() {
    printf("Hello, World!\n");
}
```

Scope is defined by the curly braces. Comparatively, in Python we can code the same example in two ways. There is a simple *script* way and a formal *programming* manner.

> *Think about the difference between stuffing all of your code into main() versus refactoring into functions.*

```
print('Hello, World')
```

The above is a simple script statement using the *print* function. We don't have to call or instantiate anything above or below such a statement. This line by itself in a Python script (e.g., hello.py) will produce the expected result. Alternatively, we can wrap this in a more traditional programming feature.

```
def Hello():
    print('Hello, World')
```

The above is a (user-defined) function or method depending on the surrounding context. More on that in chapter 4. Right now, the critical take away is the difference between language lexical markers for the scope. You see, Python uses plain old whitespace instead of character-based fencing. Thus, every indent serves as a definition of scope.

A word of caution here: pick a whitespace standard for yourself and stick to it. Personally, I prefer four spaces and have my *tab* set accordingly.

Meanwhile, parentheses are largely used the same between C/C++ and Python. There are no semicolons to indicate the end of a statement. We *use* whitespace here as well insofar as a newline indicates... well, a new line or statement. Strictly speaking, you

can use a semicolon but Python doesn't require it as C/C++ do. As well, we can use a semicolon to delimit multiple statements packed onto a single line. In pratice, don't do that however and stick to whitespace over semicolons.

There is a colon in Python which is a lexical indication for the beginning of a new definition block (with definitions being functions, methods, classes, and so forth).

To summarize, consider the following C++ snippet:

```
int main()
{
    int a = 0;
    ++a;
    {
        int a = 1;
        a = 42;
    }
}
```

How would you port this over to Python? What lexical symbols do we keep? Get rid of entirely? Substitute?

> *My advice: start with the low hanging fruit such as semicolons and braces. While you're coding the same snippet in Python, try to imagine how the Python interpreter digests the syntax.*

Variables and Pointers

On the surface, it may seem like variables look, feel, and work identically between C/C++ and Python. Take for instance, a simple value assignment in C/C++.

```
int value = 10;
```

> *Can you code the equivalent Python statement? How about adding the variable to itself?*

The corresponding Python statement is similar enough. So too is how we use the variable. However, underneath the hood there is a significant difference between languages. In fact, second to the scoping discussion, I think this difference is important to work through when transitioning to Python. Therefore, let's start to wrap our minds around C/C++ referencing values through *variables* and Python referencing values through *names*. While it is technically correct to refer to both abstractly as variables, ignoring Python's *names* obfuscates an important point.

Think about how everything in Unix is essentially a file. Now, imagine everything in Python is an object. More precisely, everything in Python is held by an object. Even a simple variable such as **value = 10** will exist within an object when our program is instantiated.

> *Before you read on, take a guess as to why Python's variable form is important!*

There is one overarching reason for this being important: our integration with Python *names* is decoupled from memory addresses. In turn, this has two immediate benefits.

```
int list_of_ints[4] = {1, 2, 3, 4, 5};
```

This is a classic array example from C/C++. Will this compile? Furthermore, let's say that the list of assigned elements were **{1, 2, 3, 4}**. I want to operate on the array itself, say by adding a new element. Can I do that?

```
list_of_ints = [1, 2, 3]
```

Here, because we have a *name* within an object, we are not limited to what we initially declared. Not only can we rewrite the *type* such as **list_of_ints = 'This is my string'** but I can operate on the list itself using powerful built-in methods. Using the idea of adding an element, we get:

```
list_of_ints.append(4)
```

> *What would be the output if we print the value of our appended list?*

The other immediate benefit is we do not have to use pointers and we can give next to no mind to stack and heap management problems. Actually, when I think about it for a few minutes there are other important benefits we ought to discuss.

Expressions

Comprehensions

Standard Library

Chapter 3

Typing

You may have noticed something when we were examining variables in the previous chapter. Specifically, the *typing* of variables. While do I think the concept of static versus dynamic types is easily grasped, there actually is a bit of technical depth here that warrants further discussion.

Static and Dynamic

Coming from the C/C++ languages, we are used to *static* variable typing. Meaning, when we declare a variable with a type such as *int*, *char*, and so forth we can only assign a corresponding value. Yes, we have idiomatic ways of working around this paradigm but stuff like type casting is an exception to the rule.

Consider something like:

```
float c = 10.5;
```

We can compare the C/C++ static typing like in the above example to the following.

```
c = 10.5
```

This is an example of *dynamic* typing and how Python handles variables. In this case, Python will treat the value as a floating-point number even though we didn't explicitly define the type.

The major difference isn't the presence of the **float** type keyword. Rather, the keyword is necessary because statically typed languages have type checks at compile time. In contrast, a dynamically typed language classifies values at runtime. Thus, the following code ought to throw a compilation error.

```
int x = 5;
char c = 'y';

x + y;
```

Let's clear up a little confusion here. First, just because Python is dynamically typed doesn't mean you can create and use variables all will nilly. There are type-value restrictions.

> *What do you think would happen if you executed the .pyc bytecode file directly? Try it and see!*

Type Checking

Chapter 4

Object-Orientation