Hello. My name is Austin Bingham, and welcome to Python: Beyond the Basics. This is a course for people who already know the essentials of the Python programing language and are ready to dig deeper, to take the steps from novice to journeyman. In this course, Robert Smallshire and I will cover topics to help you prepare to produce useful, high-quality, Python programs in professional commercial settings. Python is a large language, and even after this course there will be plenty left for you to learn, but we will teach you the tools, techniques, and idioms you need to be a productive member of any Python development tea
Before we really start, it's important that you are comfortable with the prerequisites for this course. We will assume that you know quite a bit already and will spend very little time covering basic topics. If you find that you need to brush up on Python basics before you start this course, you can always watch the Python Fundamentals course first. Python: Beyond the Basics was specifically designed as a follow on course to Python Fundamentals, and Python Fundamentals will get you up to speed on all of the topics you need for this course. First and foremost you will need access to a working Python 3 system for this course. Any version of Python 3 will suffice, and we have tried to avoid any dependencies on Python 3 minor versions. With that said, more recent Python 3 versions have lots of

exciting new features and standard library functionality, so if you have a choice you should probably get the most recent staple version. At a minimum, you need to be able to run a Python 3 REPL. You can of course use an IDE if you wish, but we won't require anything beyond what comes with the standard python distribution. You will of course need to know how to define functions, and you need to be comfortable with concepts like keyword arguments, default argument values, and returning values from functions. Likewise, for this course you need to know how to work with basic, single file module in python. We'll be covering packages in this course, but we won't spend any time covering basic module topics like creating modules or importing them. In this course we'll make extensive use of Python's basic built-in types, so you need to make sure that you are fluent in their syntax and application. In particular, you need to make sure that you know the following types well: int, float, str, list, dict, and set. Many of our examples in this course use these types liberally and without undue explanation, so review these before proceeding if necessary. Like the basic types we just mentioned, this course assumes that you are familiar with the basic Python object model. Python: Beyond the Basics goes into greater depth on some advanced object model topics, so make sure you understand concepts like single inheritance, instance attributes, and other

topics covered in python fundamentals. In python exceptions are fundamental to how programs are built. We'll assume that you're familiar with the basic concept of exceptions, as well as the specifics of how to work with them in Python. This includes raising exceptions, catching them, finally blocks, and defining your own exceptions. In this course you will learn how to define iterable objects and iterators, so we expect you to already know how to use them. This includes syntax like the for loop, as well as how to use the next and iter functions to manually iterate over sequences. Like functions, which we mentioned earlier, classes are a basic part of python, and we'll expect you to be very comfortable with them in this course. You will need to know how to define classes and give them methods, as well as create and work with instances of them. In python, as with many languages, you can treat the data in files in one of two basic ways, text and binary. In this course we'll work with both kinds of files, so you need to make sure that you understand the distinction and the ramifications of the two different modes. And of course you need to know how to work with files in general including opening, closing, reading from, and writing to them. Before you start this course, make sure that you are familiar with unit testing, debugging, and basic deployment of Python programs. Some of these topics will be used directly in the course. Perhaps more importantly, you may want to

apply these skills to the code you write as a part of this course. Some of the topics in this course can be complex and a bit tricky, so knowing how to test and debug your code as you learn might be very useful.

Finally, we need to make a quick note regarding terminology. In Python many language features are implemented or controlled using special methods on objects. These special methods are generally named with two leading and two following underscores. This has the benefit of making them visually distinct, fairly easy to remember, and unlikely to collide with other names. This scheme has the disadvantage, however, of making these names difficult to pronounce, a problem we face when making courses like this. To resolve this issue, we have chosen to use the term "dunder" when referring to these special methods. Dunder is a portmanteau of the term double underscore, and we'll use it to refer to any method with leading and trailing double underscores. So, for example, when we talk about the method __len__, which as you'll recall is invoked by the len function, we'll say "dunder-len." These kinds of methods play a big role in this course, so we'll be using this convention frequently. Python is becoming more popular every day, and it's being applied in all sorts of domains and applications. One of Python's strengths is that it's approachable and easy to learn so that almost anyone can learn to write a basic Python program. As the title says though, this course will

take you beyond that, beyond the basics. We want to teach you some of the deeper aspect of Python to give you the skills you need to write great Python programs.

Organizing Larger Programs

Packages

Hello. My name is Austin Bingham, and welcome to the first module of Python: Beyond the Basics. In this module we'll be covering more of Python's techniques for organizing programs. Specifically, we'll be looking at Python's concept of packages and how these can be used to add structure to your program as it grows beyond simple modules. As you'll recall, Python's basic tool for organizing code is the module. A module typically corresponds to a single source file and you load modules into programs by using the import keyword. When you import a module, it is represented by an object of type module, and you can interact with it like any other object. A package in Python is just a special type of module. The defining characteristic of a package is that it can contain other modules including other packages, so packages are a way to define hierarchies of modules in Python. This allows you to group modules with similar functionality together in ways that communicate their cohesiveness. Many parts of Python's standard library are implemented as packages. To see an example, open your REPL and import urllib and urllib.request. Now, if you check

the types of both of these modules, you'll see that they are both of type module. urllib.request is nested inside urllib. In this case, urllib is a package and request is a normal module. If you look closely at each of these objects, you'll notice an important difference. The urllib package has a dunder-path member that urllib.request does not have. This attribute is a list of file system paths indicating where urllib searches to find nested modules. This hints at the nature of the distinction between packages and modules. Packages are generally represented by directories in the file system while modules are represented by single files. Note that in Python prior to version 3.3 dunder-path was just a single string, not a list. In this course, we're focusing on Python 3.3, but for most purposes the difference is not important. Before we get into the details of packages, it's important to understand how Python locates modules. Generally speaking, when you ask Python to import a module python looks on your file system for the corresponding python source file and loads that code. But how does python know where to look?

Imports from sys.path