Functional Programming with Python

by Gerald Britton

Course Overview

Course Overview

Hello everyone, my name is Gerald Britton. I'm a senior solutions designer at TD Bank in Toronto, Canada, and a Pluralsight author. Welcome to my course: Functional Programming with Python. Perhaps you are a seasoned Python programmer, and have heard all the fuss about functional programming. Perhaps you feel a little bit left out since Python is not a pure FP language. Or maybe you're coming to Python from a functional language like Haskell, Scala, or F#, and you're wondering how to work in a language that seems to be lacking the things you take for granted in your favorite FP language. Whatever your background, this course will help you become a functional Python programmer. Some of the major topics that we will cover include an overview of functional programming principles, including higher-order functions, pure functions, recursion, immutability, and matching. A discussion of Python language constructs that are available out of the box to the aspiring FP programmer, and how to handle the difficulties of recursion and immutability in a language that appears to have no special facilities for unlimited recursion or immutable variables. All of these topics will be covered in the context of a business problem and order processing system so that you can see how to use these concepts in the real world. By the end of this course, you'd have learned how to apply functional techniques to make your programs more succinct, less error prone, and easier to reason about. Before beginning the course, you

should be familiar with basic Python programming, including how to write classes, functions, and methods, and how to create and use modules and packages. Also be sure you have a good Python IDE ready to go. I'll be using Visual Studio code across platform IDE for all the demos in this course. I hope you'll join me on this journey to learn how to use Python with a functional style of programming in this course: Functional Programming with Python, at Pluralsight.

Introducing Functional Programming

Introduduction

Hi, my name is Gerald Britton. I'm a senior IT solutions designer and Pluralsight author. In this course, you'll learn about functional programming, or FP, as it is often called. FP is a programming paradigm. An approach to solving programming problems. Though functional programming has been around since the 1950s, it has been overshadowed by other programming styles. In this introductory module, I'll review the main programming paradigms that are used today, and then talk about the main principles behind FP. In succeeding modules, you'll see how to take a typical business system, an order processing system, and transform it to a functional style as each functional concept is introduced.

Programming Paradigms

Wikipedia identifies eight different programming paradigms. While not definitive in an academic sense, it's good enough to get us started. The Imperative style is perhaps the most basic paradigm. In an imperative program, the code tells the computer what to do step by step, like in a Turing machine. At the lowest level, machine code is imperative. It could hardly be anything else. An assembly language, which makes machine code easier to use by introducing mnemonics for off codes and helping with housekeeping is also imperative. Procedural programming allows the developer to group code into functions which can be individually developed, tested, and maintained, a big step up from imperative languages. Some early procedural languages are Fortran, COBOL, Algol, Basic, and of course, C. Note that just because code can be grouped into functions, procedural languages are not considered functional since the code that glues the functions together tends to be imperative. Object oriented programming enhances the ideas

behind procedural languages by grouping code and data into objects. OOP has become the predominant paradigm in use today. You probably already know one or more OOP languages, which include Java, C++, Smalltalk, and of course, Python. Declarative programming focuses on what the result should look like rather than how to achieve the result. A great example of a declarative language is SQL. When you write a query, you leave it up to the database engine to decide how to carry that out. Most databases use a cost based optimizer to produce an efficient execution plan. XML is also used for declarative programming. The Hypertext Markup Language, or HTML, is probably the easiest example. HTML declares what should be on a webpage, and the browser uses that declaration to render the images the user sees. Logic programming uses mathematical logic to represent and execute computer programs. Current examples include Prolog, and Planner. This list is by no means exhausted, though it does reflect the major paradigms in use today. Except for functional programming of course.

A Brief History of Functional Programming

Functional programming is not a new idea. It has been with us more than 60 years. In the late 1950s, John McCarthy developed Lisp while at MIT. Lisp introduced many of the ideas of functional programming. McCarthy's language was not purely functional though. Like other languages including Python, Lisp is multi-paradigm, though later efforts sought to purify the language. Today, common Lisp is still a multi-paradigm language. In the early 60s, Ken Iverson developed APL, a language that makes use of just about every symbol on the keyboard. Some skeptics refer to APL programs as write only, that's a sarcastic dig at some hard to read APL programs. APL went on to influence some key components of functional programming. In 1977, John Backus gave a Turing award lecture entitled: Can Programming Be Liberated from the von Neumann Style? A functional style and its algebra of programs. His lecture and accompanying paper popularized research into functional programming, though it emphasized functional level programming rather than the lambda calculus style which forms the basis of most FP languages today. Note that we won't be digging into lambda calculus or the deep math of functional programming in this course. Also in the 1970s, Robin Milner at the University of Edinburgh developed ML, one of the first general purpose FP languages. ML is still with us today, OCaml is a popular descendent. In the 1980s, Per Martin-Lof developed intuitionistic type theory, which helped bring the rigor of mathematical provability to software construction using types. Also in the 1980s, the Haskell language saw its beginnings. Haskell has influenced many other FP languages, including a relative newcomer: F#. Well I stopped at the 1980s not because the FP

language development stopped, far from it, but because the period from the 50s to the 90s brought forth the concepts used in FP languages today.

Motivation by John Carmack

Maybe you're asking yourself why you should bother using functional programming. Well perhaps John Carmack, founder and technical director of id Software and Armadillo Aerospace will convince you. He advises to "Do it whenever it is convenient and think hard about the decision when it isn't convenient. " Carmack also observes that maintaining and mutating state is a large problem in most computer systems, and that using a functional style makes the state presented to your code explicit, which makes it much easier to reason about. And in a completely pure system, makes thread race conditions impossible. Think about that last bit. Have you ever had a problem with race conditions? Most programmers have, especially those working on multi-threaded applications. Consider the implications of that claim. That in a completely pure system thread race conditions are impossible. Not just unlikely, but impossible.

Principles of Functional Programming

Not everyone agrees on the components of functional programming, but in this course, you'll learn about most of the important building blocks and how to implement them in Python. One important building block is first-class functions. These are functions that accept another function as an argument or return a function. This is an important quality that leads to increased code reusability and allows for code abstractions. Pure functions are functions that have no side effects. Side effects are any actions performed by a piece of code that affects state outside that code. A common example would be the Python print function. This has a side effect of writing something to the console, and so it's not pure. Sometimes side effects are more subtle. For example, perhaps an object is passed to a function which then makes changes to that object. Such changes are called mutations. Pure functions mutate nothing and have no side effects. Immutable variables and objects have the characteristic that once set they cannot be changed. In FP languages such as Haskell or F#, immutability is supported by the compiler. In Python, you'll see other ways to achieve immutability. Lazy evaluation seeks to only do actual computation when the results are needed. This has many benefits including reduced memory requirements, and eliminating unnecessary computation for results that may never be used. Recursion offers an alternative to iteration and a way to reduce or eliminate temporary loop variables. In Python, a for loop needs such a variable to receive a reference to each object being iterated over. By definition,

that variable is mutable. Using recursion instead of iteration eliminates the loop variable. Though Python does not always play nice with recursion due to built-in limits, you'll see how to simulate recursion using a technique called trampolining. Many FP languages support pattern matching syntax to simplify type checking in object element extraction while reducing the need for extra variables. F# for example, offers a match expression to do just that. In fact, the usefulness of matching is leading to its inclusion in other languages as well. C# 7. 0 will include match expressions, for example. Python does not have a match expression, at least not yet, but you'll see at least one way to simulate that behavior using a match class we'll build. Matching in Python has been discussed on the Python ideas mailing list. It could be that as you follow this course, it has already been implemented. At the moment this course was prepared however, it's little more than a gleam in the eyes of the core developers. Most FP languages support the idea that statements can have a return value. Python does not support this construct, and judging by some recent discussions on the Python IDE's mailing list, emphatically not. However, I do feel it's important to state what can and cannot be achieved in Python as we move towards the functional programming paradigm. In case you're getting worried, rest assured that we are not abandoning object-oriented programming or the principles embodied in the acronym SOLID. After all, everything in Python is an object and that's not going to change. FP offers a way to bring some discipline to OOP that can help make programs smaller and more maintainable. We'll still use modified forms of popular design patterns.

Tools You Will Need

You'll need some basic tools to complete the exercises. Naturally, you'll need Python. There are a few flavors of the language implemented with different goals in mind For this course though we recommend the most recent version in either the Python 2. x series or the 3. x series. Both are easily downloadable from the Python website for all common platforms. Though not strictly needed, a good integrated development environment, or IDE, can make your life simpler. There are many available. Some are free and others are commercial. One simple IDE comes with Python: IDLE. Some popular third-party IDEs are PyCharm, Wing IDE, PyDev for Eclipse, and Visual Studio, along with many others. Online you can visit the Python wiki to get a good oversight. In this course, we will be using Visual Studio Code with suitable Python plugins. You can find it at the link shown.

Summary

This module introduced the basic ideas that are fundamental to functional programming. You also learned a little bit of the history of FP and how at least one influential software developer, John Carmack, feels about it. We outlined the major concepts of FP that we will concentrate on in this course, and we learned some of the motivations to learning FP. Before we continue, let me leave you with this one idea. While many languages center around performing actions to achieve results, FP is all about computing results. The actions then become secondary. This is a key idea that you will see in action.

First Class Functions

Introduction and Working Example

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. First-class functions are functions that are treated as objects. That means that functions can take other functions as arguments and can also return functions. In Python this is easy since everything is an object. In fact, Python has had first-class function since the very beginning. In this module, you'll begin working with the example to be used throughout this course. I think it's important that you actually work the examples. Don't just watch the pictures go by. All the code is included in the course materials for you to play with. As you begin building and using higher order functions, both in the working example and in your own projects, you'll soon become skilled in the art and see opportunities to simplify your code using this powerful paradigm. The working example we will use consists of the central pieces to an order processing system. At its core, we'll have an order object which will host a list of order items along with a reference to a customer. There will also be various indicators such as expedited, shipped, an enterprise customer indicator, and an item backorder flag, along with other state information, like the order ID and the shipping address. Let's look at the code we're starting with.

Demo 1: Before First-class Functions

Here, in Visual Studio code, you can see the three objects we're starting with. The Order class has the various attributes we mentioned before. There is also a class attribute, a list containing all the orders. Now a production implementation would probably retrieve the orders from persistent storage and store the order list in a collections object or a singleton or at a top-level module, but this will suffice for the example here. Note that there is no constructor at the moment. That means

it is necessary to add all the attributes after instantiation. We'll fix that as we go along. There are three methods to get parts of the order, but only for expedited orders. The OrderItem class is quite simple, just a list of attributes, and the Customer class similarly has just a list of three attributes. While back in the Order class, I want to draw your attention to the repeated code in the three get methods. For each method the code creates an empty list, uses a for loop to iterate over the orders, uses an if statement to check the condition, appends the item to the list if the condition is true, and finally returns the completed list. There are actually 12 lines of duplicated code that have been copied and pasted here. This violates the Don't Repeat Yourself principle, DRY for short. Repeated code can be a maintenance headache since you have to hunt down every repetition for any change. Imagine for a second you were asked to write three new methods for the not-expedited orders. You'd wind up then with 6 methods in total and 24 lines of duplicated code. That's enough to make my head spin. What about you?

First-class Functions in Python

A higher order function is defined as one that takes a function as a parameter, or returns a function as a result or both. Let's review how Python supports higher order functions. Here's a simple example. Function F takes an argument, x, and adds 2 to it. Function g takes another function h as the first argument, and value x as the second. Then invokes function h using argument x. Then takes that result and multiples it by 2. This is an example of function composition. It's pretty simple. To convince yourself that it works, pause the presentation, fire up your Python interpreter, and type in what you see. In case you're wondering why I don't just let you copy and paste it, it's because a very smart professor I once had said, "If you don't write it, you won't remember it. " Here's another way Python supports higher order functions. This one defines a function at x, which then defines an internal function. The internal function takes the value of x, passed to the outer function, and adds a new value y. Then, the new function is returned as the result of the outer function. The next two lines use this to define two new functions, which add 2 and 3 respectively to any value passed in. Pause this presentation again and try this code out in Python. Back already? Did it work? Did you get the values 4 and 6 as results? This example shows that a function can return another function. Also it shows an example of a Closure; an important aspect of functional programming. The sample function closes over the variable x, which is a captured variable, which means that it was bound when the closure was created. Y on the other hand is a free variable. Now Python supports closures out of the box and even gives you a time-saving way to build your own. Check out the partial function in the functools module. The third way Python supports high order functions is called currying in the FP

world, named after Haskell Curry, an American mathematician. Of course the FP language Haskell gets its name from Dr. Curry. The basic idea is to take a function of many arguments and reduce it step by step to a function with only one argument. If you try out the sample code, you should see the first call to print returns a reference to a function. The argument x is curried within that new function created by the called F2. The second call to print calls the function returned by F2 with the value 3. At first glance, closures and curried functions look similar and they certainly are related. I could define a closure of a function which takes two or more parameters for example, but that would not be currying. On the other hand, currying a function results in capturing n-1 of n arguments into a function, which is a type of closure. If you're thinking that the partial function in the functools module can be used for currying, you're quite correct, and that shows that closures and curried functions are both built from partially applied functions. Now, let's see how we can use some of these concepts of higher order functions in the working examples.

Demo 2: Using First-class Functions

For this demo, we'll modify the order class we started with. Let's look at it again. Notice that the three functions do not use the self argument. That means they can be static. Let's start there. So we add the static method decorator, and then we can remove the argument. We'll move onto the next one, do the same thing, add the static method decorator, remove the argument, and one more to go, add the static method decorator again, and remove the argument, we don't need it anymore. There, that's a great start. Now take a step back and think about those three methods. They all iterate over the list of orders, test the expedited indicator, and then add something to a list. When they're done, they return the list. Well, what if we define a new function: test_expedited. It should look a bit like this. It will also be a static method. We'll actually be seeing a lot of those. It takes just one argument and order item to be checked. Then, it just returns a Boolean, indicating if the order has been expedited or not. Now I can pass in my new function as an argument to a higher order function. So instead of testing the expedited attribute directly, it will call the predicate, that is the passed in function. So now I can rewrite the original get_expedited_customer_names like this. I'll declare it as a static method like the others, and give it a meaningful name. This time, however, it will take an argument, but that won't be an instance reference. Instead, it will take a predicate, a function like the test_expedited function I just wrote. The body will look pretty much like the original get_expedited_customer_names method, except for the test where I delegate that logic to the predicate function. As before, we then add the customer name to the list, and at the end of the method, return the completed list. Now I can rewrite the original get_expedited_orders_customer_names method to use the two new

functions. It looks like this. It calls the new filter function, passing in the new predicate. But wait a minute. The get_filtered_orders_customer_names method explicitly references the customer name. What if I created another helper function that returns the customer name when passed in an order object, like this. Now I can use this to create a higher order method that just iterates over the list, tests the predicate, and builds an output list without any special knowledge. It looks like this. I'll just paste it in in one go, you're probably tired of watching me type things anyway. As you can see, this method takes in both a predicate and a function. In the loop, it tests the predicate and then uses the function to get the data to append to the list. Using get_filtered_info, I can simplify the three original methods using both helper methods. Here's the one to return the customer names. From here, I think you can see that it would be easy to modify the other two original methods, and not much harder to add the not expedited versions. You'll need a few more helper methods, but they are quite straightforward. Go ahead and implement them on your own.

Summary

To recap, what have we accomplished so far? We started with an order class with lots of repeated code. As it was, adding new functionality meant even more duplicated code. Then, I reviewed three kinds of higher order functions and how to code them in Python. The first type showed how we can easily compose functions, and the result is a higher order function that combines the processing power of the composed functions. The second example showed that we can capture a variable in an internal function, and return that new function, and functions like this are called Closures. The third example showed how Currying works in Python and how that is related to Closures. Then, I went back to the Order class and refactored it. I removed the duplication using calls to new helper functions, which made the refactored functions higher order through composition. The net result is a cleaner, leaner Order class without special cases. Now the sharp-eyed will have probably noticed that I did not refactor the class completely. Consider that job to be the exercise for this module.

Pure Functions

Introduction to Pure Functions

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. In functional programming, we use functions to do specific things, and then combine them together

to do more complex things. This is like the Unix philosophy of programming: write programs that do one thing and do it well, and write those programs to work together. We're going to apply that philosophy at the level of functions. To see how to do that, I'll extend our working example by adding some new functionality. As I do that, I'll want to use pure functions, so let me describe them more fully. Pure functions are simple. They do one thing and do it well. Functions that do lots of work are hard to test. Over time, you might find yourself adding more functionality and then adding more parameters to your functions to handle all that. A better approach is to write smaller functions that do one thing well. Then combine them to increase the overall functionality. Since they are simple, pure functions have a limited number of arguments. In case you're wondering, the ideal number of arguments is 0. You can't always do that though. Still, if your argument list starts climbing beyond two or three, it's time to ask yourself if you should refactor your function into two or even more functions. You'll see an example of that shortly. One tenant of pure functions is this. The set of inputs completely determines the output. Also, a pure function always returns the same output given the same input. Pure functions do not use state, nor do they modify state, and that means they have no side effects. They do not use variables outside their own scope. This makes them easy to test and immune to code changes around them. This also means that functions that print or use the current time for example, are not pure since they modify our use state and may not return the same result given the same inputs. To achieve greater functionality, you don't want to add complexity to your functions. Instead, you combined functions to achieve the desired result.

Simple Functions

As an example of simple functions, think about this. Say you have a list of integers, and now say you want to process that list, but sometimes you only want odd integers, and sometimes even, and sometimes all of them. How could you do that? Well, how about this do it all function? You can call it with various combinations of odd and even, and get back just the odds, just the evens, or all the integers or none of them. To do that, the function has two Boolean arguments. Now, Guido van Rossum, the author of Python and Benevolent Dictator for Life, or BDFL, has had something to say about that approach. His basic idea is this: if a function has Boolean arguments that are almost always called with literal values of true or false, split the function into two. Even though the do it all function is pure, in that it neither uses nor modifies any state, it's still better to split it up like this. Now I have three pure functions, each of which does one thing well. No complex logic in sight. I can get the even integers, I can get the odd integers, and I can get all of

them with different calls. Oh and I've also eliminated the superfluous fourth case that just returns an empty list.

Demo 1: Add New Functionality

Here, back in Visual Studio Code, you can see the order class we finished with in the last module. I've completed it and tidied it up a bit. This time around, we want to add some new functionality. In this system, no new orders are expedited by default. To expedite an order, we need some code that will find an order, give it an order ID, and then set the expedited flag to true. Here's one obvious way to do it. So we have a new staticmethod that just loops through the orders looking for a matching ordered, and if it's found, it sets the expedited flag to true. Well that's pretty simple, but I'm not really happy with it. For one thing, I'm explicitly looping over that list of orders again. I thought I got rid of that sort of thing in the first module. Also, how likely do you think it will be that I'll have to find an order by ID for other purposes besides setting the expedited flag? Pretty likely I think. So let's write a method to return an order object given an order ID. We'll call it get_order_by_id. Here's the first attempt. It just loops through the orders, looks for a matching order ID, and returns that order. Wait a minute though. Remember the method called get_filtered_info from the first module? Here it is again. It loops through the orders using a predicate to test a condition and then uses another function to append to a list, which is returned at the end. Can I use that in get_order_by_id? I sure can. Here's the new version. It has a few things going for it. First, it reuses the code we worked so hard on in the first module. That reduces repeated code and eases maintenance. Second, it uses lambdas to match on the order ID and just get the whole order object. Third, and this is a bigger change but one you'll see again and again. It returns a list of matching orders. Now, no one said that there could be only one order with a given ID, though that's likely in a real system. More importantly, it eliminates a bug in the first version of this method. That version simply didn't handle the order ID not found condition. That means that the method would return a python none if there no was no matching order, and that means the caller would have to have code to handle none. Instead of making life easier, we made it harder. By returning a list, which can be empty of course, that problem disappears. Using the new method, the set_order_expedited method is even simpler. Now it just gets the list of matching orders, which can be empty, and sets the expedited flag for each order that matches the order ID. In fact, we could go one step further and add a second parameter to the method call: a Boolean representing the new state of the expedited flag. How would you go about doing that? What else would you change, add, or delete?

Demo 2: Purify the Functions

Now that we've added the required new functionality, let's make our functions as pure as possible. Recall that a pure function only depends on its inputs and outputs. That means that we cannot reference the order class any longer from within the purified functions, though it can be passed in as a parameter if needed. We need to start with the get_filtered_info method. It actually has two responsibilities. First, filter the list of orders by some predicate, and then build a list of items using the passed in function. We need to break it up so that each function has a single responsibility, and yes, that is the single responsibility principle of object-oriented programming. I told you we'd be using and applying the principles in the acronym SOLID. So to break it up, let's start with a filter function. At this point the function will be different from the built-in filter function since it will return a list. When we get to talk about lazy instantiation, we'll change that. Now to be pure, the filter function must receive the list as an argument. It looks like this. The argument it is any Python iterable. The result is built using the built-in filter function, and the predicate passed in as an argument. The iterable could literally be anything. So could the items in the iterable. Using the new filter function, the get_filtered_info function can also be rewritten. It will use the new filter function we just finished and then apply another function to the results. But wait, that sounds a lot like what the built-in map function does. So let's write a map function we can use here. It looks like this. Even though map returns a list, it may not have to. If all you really want to do is apply the function to every item in the iterable, then check out the recipe for consume in the documentation for the Itertools module. Now, with the map function in place, here's the new get_filtered_info function. It takes three arguments, which makes it a borderline candidate for a currying, but it really is the minimum we can do here. The get_order_by_id function can now also be simplified, and can call the new filter function directly, but let's make that one pure also. That means we have to pass in a reference to the list of orders. It should look like this. There. Now it all fits on one line. Note that although it is pure now in that the output only depends on the input, it is not polymorphic, it operates within a specific domain, that of the list of orders, and the predicate tests the order ID by name. Now, to purify the set_order_expedited method, the new functionality we're adding, we have to pass in the list of orders. Here, this ought to do it. Good. Now you may be wondering if I could have used the order. map function here. Well yes indeed, but I think I'll leave that as an exercise.

Python Lambdas and Performance

So far, we've added some new functionality. Now we have a way to set the expedited flag given an order ID. We also eliminated the need to handle None as a return type by only returning and

processing lists, even if we only expect to modify one item. Then, we added two new and pure functions: filter and map. Using those functions, we refactored the code to purify three other functions. Now at this point, I need to emphasize that it's usually impossible to have a meaningful system that's completely pure. You can't purify everything. After all, somewhere you'll want to get input from a user, or output to a display, or updated database, and those are inherently impure operations because they use or modify state. What you really want to do though is minimize and isolate impure operations. Now, I want to have a sidebar about lambdas in Python. They're different from an FP language like F# or Haskell. So, let's talk about lambda functions. Since Python is dynamic, many things are left to run time, including lambda definitions. This is simple to illustrate. A very handy module in the Python Standard Library is the dis module. You can get it by typing import dis into a Python shell. Now consider a simple function like the one above that uses lambdas. It's a bit like some of the ones we looked at at the order class. Here it uses two lambdas. If you have a Python shell handy, pause the video and try it out on your own. See what happens when you disassemble function F. When you disassemble function F, you should get something like this. First, it loads the code for the first lambda. This was precompiled but not completed. Then it makes a function out of the lambda. The same thing happens for the second lambda. Finally, the original function is invoked and the results are returned. The alternative to using lambdas is to define helper functions to replace the lambdas. Then use those helper functions in the main function. Well let's disassemble that and see how it differs. This time things are simpler. The references to the two helper functions are just loaded since they've been predefined. Then, the original function is called. So should you use lambdas or helper functions? Like so many things, the answer is it depends. Using lambdas puts the helper code right where it is used. That makes it easy to see what's going on. As a bonus, lambdas are anonymous. You don't have to think up names for them or worry about namespace pollution. On the other hand, in performance critical sections, all those calls to make function can add up. However, do not fall into the trap of premature optimization. First, make your code work, make it readable, and make it understandable. Then only if a performance issue crops up and profiling indicates that the lambdas are a problem, refactor them into helper functions.

Summary

To summarize this module, we introduced pure functions. These are functions whose output depends solely on the input, and pure functions must have no side effects. I added new functionality to the order class to set the expedited flag of an order. Along the way, I created new pure functions: map and filter, that use the built-in Python functions of the same name, but return

lists. Then using map and filter and purity principles, the code used to implement the new request was also made pure. Since I'm using lambdas a fair bit, I took a few moments to discuss them from a performance perspective. Now that we have higher order functions that are also pure, it's time to learn about recursion.

Immutable Variables

Introduction to and Motivation for Immutability

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. In this module, I'll show you how to use immutable variables in Python. Immutability, the inability for a variable to change its value, is foundational to most functional programming languages, and if you want to use Python using a functional paradigm, you need to know how to contain mutability. Now out of the box, Python has no way to declare a variable to be immutable. That means that when you want to use immutable variables, you do so by convention, though you'll see a way to make Python classes immutable. Before that though, let's talk about the elephant in the room. How does a program do anything if it can't change anything? Let's look at that. The basic idea is that immutable variables cannot change their values, and we'll do this by following two simple rules. First, local variables do not change. In a Python program that generally means anything in the locals dictionary. Interestingly, the Python documentation for this dictionary stipulates that its contents should not be modified since changes should may not affect the values of the local and free variables used by the interpreter. In other words, local should be considered immutable. The other rule is that global variables can only change their references. Now, if you've been using Python for a while, that may seem redundant since in Python all variables are references to objects, even if they're just simple types like integers, or Booleans, or strings. However, it is this rule that makes it possible for your program to use immutable variables and still do useful work. You'll see that in the examples as we go along. The Python built-in function globals, returns the dictionary representing the current global symbol table. This is always the dictionary of the current module. That implies that application level globals should be in the topmost module. To see how immutable variables can get you into trouble and make debugging harder, consider the following example. Let's say you don't like the fact that pi is an irrational number. Well it's also transcendental and that's important if you're a math nerd. As a first attempt, you might try this. Do you think it will work? Well, go ahead and try that on your own and let me know. Did it work?

Thought not. Even though the names are the same, the argument pi is a reference to the true value of pi, but in the change_pi function, the argument becomes a local variable, so the changing it does not change the value referred to. If you still want to change pi, you might try to do something like this. Now this is a truly nefarious function. It explicitly changes the global value, well actually it changes the references for the global variable pi. Even though we generally will need to change global references, this is not the kind of thing you want to do, and before you congratulate yourself on doing something cool, consider the repercussions. All manner of bugs will arise for anyone using pi after you get through with it, and finding those bugs will not be any fun. You could comment your code like crazy and still your users will hit the problem. By the way, there was a legislative attempt once to change the value of pi. In 1897, bill 246 in Indiana would have done just that. See the link on the slide for the whole story. Now, let's look at a demo that will add some more functionality to the example system. You'll see how to do that and still use immutable variables.

Demo 1: Add Backorder Notification

To enhance our order system, we need to add two new functions. Now each order has a list of order items, and upon request, we need to look through the list of order items. If any backordered items are found, we need to notify the customer on the order, and the second new function we need to add is to mark an order item as backordered. First, let's enhance the OrderItem class by adding a constructor to it. Back in Visual Studio Code that should be easy. Let's do it here. Next, we have to modify the Order class to add a reference to a list of order items. While we're at it, let's add a constructor. We'll need it anyway as you will soon see. Now, let's add a constructor and a simple notify function to the customer class. There. Now we're ready to add the first new function to the order class: notify_backordered. It could look something like this. But that's not functional, it's the traditional imperative style. Now, here's a functional version. Let me just paste this one in. How does it work? Well, starting from the inside, the filter function runs through the order items, returning only those that are backordered. The outer filter function looks at the result of the inner filter. Now since order filter returns a list, and an empty list has a Boolean value of false, the lambda will evaluate to false for that particular order, which will then be excluded from the result list. The Order. map function runs the customer. notify method for any orders returned by the filtering process. The Order. map function also returns a list, which will have one item for each result from the call to the customer. notify method. That list is just discarded. If you've been following this course, and especially the last module, this might be looking strangely familiar. In the last module, we defined three useful functions in the Order class. They are filter, map, and

get_filtered_info. Here they are for reference. Filter just returns a list given a predicate and an iterator, the function tests the predicate on each item in the iterator and returns a list of those items that return true from the predicate. Map returns a list given a function and an iterator. The function is performed on each item in the iterator and the result is returned as a list, get_filtered_info combines these two, that is, it composes them. To return a list to our Order. map acts on the results of Order. filter, but that is just what we're doing here in the nofity_backordered function. The only difference is we don't need to keep the results. So here is a new version of notify_backordered that uses get_filtered_info and just discards the results. Notice that the first lambda uses the built-in any function and a comprehension. Now I could've used Order. filter here, but we're going to be using more comprehensions as we go along, so here's one you can see in action.

Demo 2: Add Mark-backordered Functionality

So we've got half the assignment done. We can notify customers that they have backordered items. Now how about that second request? Mark an order item as backordered. Now the straightforward approach would be to find that order item and set the backordered attribute to true, but that would be mutating that variable and the order item object. We're supposed to do this in an immutable way. While we're at it, let's make all three objects immutable. We already added constructors, what we need now is additional logic to prevent them from being changed. Of course this is Python, and a determined programmer can find a way to change anything. We can't totally prevent that. However, what we can do is create a way to make immutability the default state for an object, and raise an exception if anyone tries to mutate that state, at least that will make it difficult. We'll do this by creating a metaclass to help, shown here. Any class that wants to be immutable can inherit from this. Let me walk you through it. First, notice that I'm using the __slots attribute. This ensures that no new attributes can be created after instantiation. In this case, there's just one entry, the __attrs. I'll use this to keep track of which attributes have already been set. In the init method, I call the super class as per convention. Then initialize attrs to an empty frozenset. You could use a tuple here if you only expected a few attributes. Either way, an immutable type is to be used. Next, I implement the __setattr method, and this is where the magic happens. The idea is to allow a one-time assignment to each attribute, but block any further attempts. I track these assignments using the __attrs we just looked at. Note that I have to allow attrs to be changed or init will fail, but really what I'm changing here is the reference. Now if the name already exists in the attrs set, I raise an AttributeError, since this then would be an attempt to mutate that attribute. Otherwise, I allow the assignment and add the name to the set

of those already assigned. I'm using a set union operation, which returns a new set, so I'm changing the reference, but not the set itself. This bends the rule about local variables, but it is necessary here to make the scheme work. I also implement the __delattr method for much the same reason. We don't want to create new attributes after instantiation, and we don't want to be able to delete them either. Now using the immutable metaclass, I can make the three classes immutable. All I have to do is inherit from the immutable class and add the slots attribute. For example, here is what is needed for the Order class. Import the Immutable class, inherit the class in the Order class definition, add the slots definition, and then we need to delete the default attribute settings since they will interfere with the operation of slots and will be set in the constructor instead. With this in place, let's return to the request: mark an order item as backordered. That means we need a function that takes an order ID and an order item ID. We then have to find that order and the item in question. To mark it backordered however, we will not modify the attribute. We can't do that anymore since we made the OrderItem class immutable. No, what we will do is return a new collection of orders consisting of the existing collection, but replacing the order we are modifying with a new one. Similarly, we'll replace the OrderItem collection for that order with a new collection. The new function could look something like this. Let me just paste it in here. In the first part, we have to match on the given orderid. If it does not match, we just copy the existing order object to the output list. If the order matches though, we have more work to do. Then, we instantiate a new order object, copying all the attributes from the existing order, except for the collection of order items. For the OrderItems, we do something similar. If the OrderItem does not match the one given to the function, just copy it. If it does match, create a new OrderItem object, copying all the attributes from the existing OrderItem except the backordered attribute, which we set to true as requested. This function returns a brand new order collection, and for the order that changed, a brand new collection of OrderItems. Well, what happens to the old collections? They will be garbage collected. In fact, it is the garbage collecting power of modern programming languages that makes this whole thing possible. Now, whenever a calling program asks for the orders collection, it will get one that is completely up to date. This is important when the collection is used asynchronously, perhaps in a multi-threaded application. No thread has to worry now about getting a reference to a collection that is in some intermediate state. Otherwise, you might have one thread thinking an order item is fine, while another thread thinks it is backordered. Now this can no longer happen. Naturally, if your application really is asynchronous, and you maintain one collection of orders, you may need to serialize access to that global variable since its reference will be changing. However, nothing changes in the objects it refers to. They are now immutable. Looking at the problem again, there is some refactoring we can do. Now, it's likely that the pattern of updating a collection by creating

a new one will be useful in general. So, let's abstract its logic into a higher order function, get_updated_tuple is just such a function. Note that I'm placing it outside the Order class entirely since it will likely be useful elsewhere. Also note that it returns a tuple, an immutable collection. It also takes a predicate and a function. It operates in a fashion similar to the mark of backordered function, except that it uses a comprehension to do the iteration. Using this new function, mark_backordered will look a little different. More like this. Notice that it calls get_updated_tuple twice. Once for the Order collection and once for the OrderItem collection. It uses lambdas for the predicate and function arguments, and note that the predicates are actually closures since they capture the arguments passed to the mark_backordered function.

Summary

In this module, you learned about immutability and how to work with it in Python. The idea is that local variables can never change, but global variables can change their references. To see this in action, we enhanced the order management system by adding code to notify customers of backordered items, and by adding the ability to mark items as backordered. We did the latter while preserving immutability, starting with the immutable metaclass. Then, the logic reconstructs the Order collection, copying unchanged items and building new Order objects for those that change. This is a typical approach you will see in FP programs regardless of the language. Then, we change the Order collection to use tuples which are immutable in Python. And along the way, we created a new, pure, higher order function to help us with the work. The latter is a normal part of developing functional programs. The key is to be on the lookout at all times for a functionality that can be abstracted. Abstraction, using pure or higher order functions, lets you concentrate on the details while leaving the heavy lifting to the abstractions.

Lazy Evaluation

Introduction

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. Tell me something. Do you ever procrastinate? Are there jobs you know you have to do, but avoid putting off until the last possible minute? Paying taxes maybe, or cleaning out your bedroom. I know I'm guilty of doing that kind of thing. However, putting things off until absolutely necessary is not always a bad idea, especially if the job could be a big one and there's at least a chance that

you may never have to do it at all. That's the idea behind lazy evaluation. Some operations are costly - perhaps in terms of memory, or CPU time, or disk IO, or network access. If those jobs can be postponed or even avoided altogether, there can be a positive benefit to the whole ecosystem. Let's dig a little deeper to understand the differences between lazy evaluation and its alter ego, strict evaluation.

Strict vs. Lazy Evaluation

Strict evaluation is the traditional programming paradigm. Statements are fully evaluated when reached during program execution. Lazy evaluation delays this and only produces results when they are actually needed by the program. Think of a data structure that contains information extracted from a database server of tens or hundreds of thousands of rows. To fully populate the structure means to connect to the database, which may be remote, and read all the information associated with that data structure. That can be expensive. Using strict evaluation, variables are assigned the results of statement executions immediately. Under lazy evaluation, variables are assigned execution results when needed. Think of strict evaluation as paying up front. In contrast, lazy evaluation is like pay as you go, but strict evaluation can be costly, and lazy evaluation can save resources. However, there are circumstances when strict evaluation can be cheaper overall, especially in highly multi-threaded applications, whereas lazy evaluation might actually cost more overall in the same circumstance. For example, in our order processing system, we have a function to mark certain order items as backordered. Under strict evaluation, this would be the responsibility of one thread. All other threads would block on the orders collection until updates were complete, would be free then to run asynchronously for all read only operations. Under lazy evaluation, any number of threads might try to update the collection at the same time, all other threads would block when one thread was updating the collection. This could increase contention for that resource, and reduce the overall responsiveness of the system. Like so many questions, the best choice is not always clear from the get-go. That is why it is so important to use realistic workloads when testing a multi-threaded application, and not assume that one size fits all. With that background, let's look at features in Python that support lazy evaluation.

Python Tools for Lazy Evaluation

Most languages have some version of a while loop, and while loops can always be used to process results lazily. In Version 1. 4, Python gained the xrange built-in function, which is evaluated lazily, and made it easier to build other lazily evaluated objects. For history buffs, Guido van Rossum

added the code for xrange in October of 1993. That's more than 20 years ago. In Version 1. 5, the __getitem method was added as a way for a class to present an indexible structure to the interpreter. That means you can write a for loop if an object implements getitem, and use the for loop to iterate over the object. The implementation of getitem itself can of course use lazy evaluation. Starting with Python 2. 2, support for iterators and generators was added along with the yield statement. These form the basis of most lazy evaluation in Python today. They make it easy to produce lazy results. In Python 3. 0, the built-in functions map and filter became lazy. They no longer return a list, but rather a reference to an object that supports the iterator protocol. Interestingly, with Python 3. 0, the laziness of xrange was rolled into the range built-in function, which also became lazy. Skipping forward to Version 3. 3, the yield from statement was added, permitting a generator to delegate part of its operation to another generator, allowing for easy refactoring. The language continues to evolve, and the movement towards lazy evaluation also continues. It is often the default operating mode. Now, let's see how to apply some of these ideas in our order processing system.

Demo 1: Compute Order Total Price

Let's enhance the order system even more. Say that you want to see the total value of all the items in an order, also, you want to make it easily available in the order class, perhaps using a Python property. Now if you computed that whenever you instantiated an order, that could be expensive, and if you need to have a grand total available, that would be even more costly since you'd need to iterate through all the order items for all the orders to calculate that total. And if the orders collection is stored in a database, that may involve considerable IO and network transmissions. You'd like to keep all of that to a minimum, plus, any time you change the order collection, this value would have to be recalculated, and worst of all, this may be a value that you will only be asked for now and then. For example, if I modified the Order class to add a total price computation, perhaps like this, then this would be computed every time an order object is instantiated, but that's the opposite of lazy evaluation, called eager evaluation. So, let's put that aside for the moment. Now let's try a lazy approach using a Python property. Actually, I'll start in the OrderItem class and add the property there. No need to make this one lazy, it's just a simple multiplication. Back in the Order class, I'll use that property I just created in the OrderItem class. You might recognize what I'm doing here. It's a classic implementation of the composite design pattern from Object-Oriented Programming. If you want to learn more about OOP, and how to use those design patterns in Python, there are a few good courses on the subject on Pluralsight. com. Back in the Order class here, the summing comprehension is now inside the property

definition. In case you are wondering if you can do this in a functional manner without using a comprehension, it is possible using the reduced function. Here's the alternate approach. Now if you're like me, you may find that the comprehension is easier to read, not to mention shorter to write. Still, reduce has its uses, and along with map and filter, you have a set of higher order functions that can be used to solve many problems in functional programming. There is a problem with the total price method as written, however. The result is computed every time it is asked for. This is arguably worse than eager evaluation, but Python's functools module has a decorator that can help. The LRU cache decorator, new in Version 3. 2 is a memorizing director. It remembers previously computed results, and actually that's just what we want. So, if I decorate the total price method like this, it computes the total price and remembers it. The next time the method is called, the result is returned without computing it afresh. Note that I only ever have to remember one value since order and order item are immutable classes. Now we have true lazy evaluation.

Demo 2: Making Other Functions Lazy

The new total price property is a great example of lazy evaluation. Still, there is more work that we can do. So far, all our work has been about processing collections of orders in order items, and returning new collections whenever we're looking for things, or building a brand new collection if things change. For example, our handy helper function, get_updated_tuple does just what its name implies: returns the new tuple, but what if we don't actually need the whole tuple at once? Well, we can make it lazy. All I really have to do is remove the word tuple, and there it goes. Now instead of returning a tuple, I'm returning a generator, which by definition is lazily evaluated. I really should rename it though. Let's call it get_updated_sequence instead. So I'll just copy this and I'll do a global search replace on it, calling get_updated_seq for sequence, and let's just replace this everywhere. There we go. Now this function returns a sequence in the form of a generator instead of returning a tuple. Now you may be wondering if there's a downside to retuning a generator? Well, there are at least two. First, if the work inside the generator is relatively expensive, and the result may be used by many concurrent tasks, computing it once for all may well be cheaper. Profiling your application can help here. Second, you cannot directly index a generator, though the itertools module contains some functions such as t and islice that can be used to mitigate this limitation, at the cost of extra CPU cycles in memory. But the message should be clear. There is no one size that fits all. There is a general rule though that I try to follow. First off, make my work understandable, then, and only after that, work on the performance if it proves to be a problem. Usually, making code work and making it readable are

the most important things to concentrate on. If you're writing code for automatic stock trading though, fast will win the day, but then you'd probably not be using Python for that application anyway.

Summary

In this module, you learned about lazy evaluation and how to work with it in Python. The idea is to compute results when needed, just in time to use them. This saves potentially expensive operations, especially when loading from persistent storage like a database. This can be really important when the result may be rarely, if ever, called for. The opposite of lazy evaluation is called eager evaluation, and this is the default mode for most programming tasks. You learned that you need to select the mode of operation after careful thought. But sometimes you need to profile an application to see which is the best fit, and since Python 3. 0, many built-in iterables are now lazy, including map, filter, and range, and many of the functions in the functools and itertools modules. You also saw that lazy evaluation fits well with immutable properties with a little help from the LRU cache function in functools. Get comfortable with writing and reading programs using lazy evaluation, you'll be seeing it again.

Recursion

Introduction and Foudations of Recursion

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. Next to immutability, I supposed recursion is a technique that many programmers shy away from. In some cases with good reason. Even though most modern programming languages, including Python, support Recursion, not all do it in a way that makes recursive programming both easy and performant. Most functional languages handle recursion fairly well though. Especially a type called tail-call recursion. We'll start off this module looking at the motivation for using a recursive style of programming, and also some of the issues that can arise when either the programmer or the compiler is not fully up to the task of using a recursive paradigm. Recursion has a solid mathematical foundation that dates back centuries. Many mathematical sequences are defined as recurrences, and computing a recurrence is done using recursion. Fast forwarding to 1960, the Dutch computer scientist E. W. Dijkstra wrote a paper on how to do recursion on a computer using a stack. You can read it at the link shown here. Dijkstra famously championed recursion over

iteration. He's most remembered though for an article published as Go To Statement Considered Harmful. Check that one out when you have a chance. Now the stack-based approach to recursion is both the enabler and the sore spot of much recursive programming today, as you'll see in a moment. If you remember your computer science classes, this should look familiar. That's the definition of the Fibonacci sequence, named after the Italian mathematician from the 12th Century who first discovered it. If you're unfamiliar with a notation or just need a refresher, let's read it from top to bottom. Capital F represents a Fibonacci number, with the subscript n, it represents the nth such number. The braces contain the definition of this infinite series. If n = 0, then F0, the 0th Fibonacci number equals 0. If n = 1, then Fibonacci number F sub 1 = 1. If n is greater than 1, then f of n equals the sum of the preceding two Fibonacci numbers. It's the presence of the Fibonacci references on the right-hand side that make this a recurrence. This is solved by first computing the preceding two Fibonacci numbers and adding them up. Of course this may mean additional computations along the way. Think about F sub 10, the 10th Fibonacci number. To solve that, you first need to solve F sub 9 and F sub 8, but to solve them, you need to solve F8 and F7, and F7 and F6, and so on. If you do it by hand, it gets pretty tedious. Now here's another recurrence: f sub n is the sum of the numbers from 0 to n, written as a recurrence. Let's look at that one in Python and see what we discover.

Demo: Summing, Recursively

Now, here's the sum function, written in Python and following the recurrence definition. To test it, I'll just print out the sum of the numbers from 1 to 1000. That should be easy, so let's run it. Oops, that's a problem. Maximum recursion depth exceeded. What's that? The problem is that every time s is called recursively, it needs another frame on the stack. Eventually, that can cause the stack to overflow. To avoid the problem, Python has a built-in recursion limit. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. You can see the current value using sys. getrecursionlimit, here I've got it, so let's just uncomment this code and run this bit. The current and default value, as you can see, is 1000. Now, you can set it higher using the corresponding setrecursionlimit function, but that might cause the very problem Python wants to avoid. There must be a better way.

Tail Recursion

Tail-call recursion, or just tail recursion is a technique used to avoid the problem of stack exhaustion in recursive programs. A tail call is a call to some function g as the final action of

function f. If that leads to another called f, then the function is considered to be tail recursive. Note that functions f and g may actually be one and the same. Now, many functional languages detect this situation and effectively turn the tail-call into iteration instead of recursion, and this optimization causes the program to use only one stack frame per function, thus avoiding stack exhaustion. Now, naturally, it's a bit more complicated than that under the covers, but that's the basic idea. So let's go back to Visual Studio Code and modify the sum function in Python to see what it looks like as a tail recursive function.

Demo: Tail Recursive Sum Function

Back in VS Code, I have the original sum function on the left, and the tail recursive version on the right. Starting from the signature, you can see the first difference. The tail recursive version has an extra argument: acc, short for accumulator. It starts with a value of 0. As the function progresses, when n reaches 0, the value of the accumulator is returned. Contrast that to the original left-hand side where when n reaches 0, then the value of n is returned. In the new function, when n is greater than 0, the function calls itself with a new value for n and the accumulator. Since there is nothing after the last call, and since s does nothing with the value returned, this is a tail-call. Contrast that with the original version on the left. There after the recursive call, the result of that call is added to the current value of n. Well let's run them both to prove they give the same result. If I run the left, the original version, I should get 55, and I do. If I run the tail recursive version, I hope I also get the same result. Indeed, I do. So, this looks promising. Now let's using the tail recursive version, try to do our sum up to 1000, and see if this works. Oh dear, it still fails with the maximum recursion depth exceeded. Why is that? The simple answer is that Python does not do tail-call optimization, and so continues to use up stack frames. In fact, Guido van Rossum, the author of Python is on record as refusing to even consider the idea of tail-call optimization. You see, even though the technique has obvious benefits, it does have a dark downside. In the event of an error, there is no stack trace to examine, and speaking of stack traces, just look at the one generated for this error. If I scroll through the mini map, you can see that it goes all the way back to the very first original call. That's one thing tail-call optimization cannot do.

Trampolining

So now you've seen a simple example of a tail recursive function, and since Python does not optimize tail-calls, we need a workaround, and that workaround is called trampolining. In the Olympics, the trampoline competitions are both exciting and frightening, but the basic idea of

bouncing back up and down is what inspires this software technique. The idea is, when you reach a tail- call, since there's no more to do in the function, simply jump back up to the start of the function with the new arguments. No, I'm not advocating a return to the GO TO statement, we're going to leverage the power of generators instead, and use a simple idea first put forth by Alex Beal, currently a software engineer at Twitter, in a blog post from 2012. It's brilliant in its simplicity. Let's take a look at his tramp function, and how to use it with a tail-call version of the Fibonacci function.

Demo: Fibonacci, Trampoline Style!

Tramp takes a generator as input, along with any other arguments to be passed to the generator, and because it takes a function as an input argument, tramp is of course a higher order function. First, tramp calls a generator to set it up. Recall that in Python the first call to a generator returns the generator, but does not run it. Then, the function runs a little loop. As long as the generator is a generator, it gets the next result. Now the sharp-eyed will have noticed that g is a mutable variable. Sometimes you do need a little mutability to do what you want. When the loop ends, it means that next(g) did not return a generator. Then tramp just returns the current value of g, whatever that is. Now let's use this with the Fibonacci function. Here's the tail-call version of the Fibonacci function. It should look a little familiar by now, except that it has two extra arguments: curr and next. If I run it, I should get the first 10 Fibonacci numbers. So let's do that, and yep, I sure do, there they are. Next, I'm going to change this function in a subtle way. I'll replace the return statements with yield statements. In fact, I've already set up a version of that with those things in place. Now, I can use Alex's tramp function to run it. Here, I've also just modified the print statement to use tramp, and if I run this, I will get the same results as before. Now, acid test time. What if I ask for a large number? So let's comment this, and let's try print(tramp(f, and let's try a hard number, 100. Should that work? If you remember, doing this in a pure recursive fashion in Python was actually fairly slow, but here we've got a result very quickly indeed. Well let's be brave, let's go for 1000, that should blow the recursion limit if nothing else does. It does not blow the recursion limit. In fact, it gets us the correct answer, which is a very large number indeed. Are you ready for another order of magnitude? Let's try 10000. What should happen here? Well in this case, the number is so big, it barely fits on the screen, and then it returns it so quickly. Clearly this approach performs better than plain old Python recursion, and permits unlimited recursion since it's not really recursion at all. That's the whole point behind trampolining.

Recap: Recursion, Tail Recusion, and Trampolining

So far, I've reviewed what recursion is, along with its mathematical roots. Next, we took a look at how Python handles recursion today, along with the problem of stack pressure, and how Python attempts to mitigate that problem. Tail recursion, as supported in many FP languages, optimizes recursive calls placed at the end of functions, effectively turning a recursion into iteration, thus, avoiding stack pressure. However, Python does not do that kind of optimization and likely never will since it comes at the cost of lost traceback information when errors occur. I then introduced the concept of trampolining, and how this can be used to simulate tail-call optimization when the compiler doesn't support it. The last thing we looked at is the tramp function by Alex Beal, which uses generators to do trampolining in Python. There are other implementations of trampolining in Python, which you can easily find with a little searching. One thing to know about trampolining though, it relies on garbage collection to get rid of unused objects. In the tramp function, every time it gets the next generator, the previous one is ready to be garbage collected. The other thing to know: generally, pure iteration is faster. Actually, once you have a function in a tail-call form, changing it to pure iteration is usually quite easy. However, faster is not always better. What counts more is clear, concise code that you can reason about. Recursion can give you that more easily than iteration. Now, let's add some new functionality to the order processing system, and learn how to process lists and tuples recursively.

Demo: Processing Lists and Tuples Recursively and Enhancing the Order Processing System

Let's say you need to get a count of expedited orders that have backordered items. Now I bet your customers would want to know that, especially if they're paying a premium for order expedition. I'm sure that you could figure out how to do this with a list comprehension, but let's use tail-call recursion and the tramp function instead. If you think about a list or a tuple, or any type of sequence really, it consists of the first element followed by everything else. We call the first element the head of the list, and everything else is called the tail. Thanks to Python's slicing syntax, these are easy to get. The first element is just List sub 0. That's the head. The rest is the slice from element 1 to the end. That's the tail. So the new function must process the head of the order list if the list is not empty. Then see if there are any backordered items in it. Then process the rest of the list with a tail-call. Let's see what that looks like. The code I'm pasting in here is the function we've been talking about. It counts expedited orders that have backordered items. The structure is similar to the tail recursive version of the Fibonacci function we looked at before. This function takes a reference to a list or a tuple of orders. The definition also includes an accumulator. For the base case, the function checks to see if the list is empty. If so, it just yields

the current value of the accumulator. For the recursive case, it first gets the head of the list, or sub 0. Using that, it sets the variable, add, to 1 if there are any backordered items, and if the order is expedited. Otherwise, the variable is set to 0. Then for the tail-call, it yields a call to itself passing in the tail of the list using a Python slice and a new value for the accumulator, which is the current value plus that add variable that we just set. Since this function is tail recursive and uses yields, it's set up to use the tramp function. I've put a sample of the usage in the doc string. Looking more closely, you can see that this function really only does one thing, it's not higher order. To make it higher order, you might want to think about abstracting the tests for expedited orders and backordered items into function calls. Then you could use lambdas to pass in that logic. Also, there's another opportunity for abstraction here, where the function uses the any function. For one thing, this could be changed to a tail recursive function to process the order items the same way the orders are processed. Check the exercises for this module for more information on that. Finally, it might be interesting to see how this could be written just using comprehensions. It looks like this. Just two lines in the function body. This illustrates that even though the recursive approach is thought of as the purest, there may be other simpler ways to do the same thing.

Summary

In this module, we took a look at building recursive functions in Python. Along the way, you saw Python's limitations and approaches to overcoming them using tail recursion and trampolining. However, as the demo showed, this may result in longer code when a comprehension could do the trick just fine. Plus, keep in mind that although trampolining allows for near infinite recursion, it comes with the cost of extra object creation and destruction. The other less obvious cost is the loss of traceback information in the event of an exception. So, should you use recursion or not? As always, the answer is it depends. If the logic naturally lends itself to recursive expression, and the gains in readability outweigh the overhead and information loss, go recursive. If the logic is easy to do in a simple comprehension as in the last demo, then do it that way. Whatever you do, always favor a highly readable code that reflects the problem definition. If you're worried about performance, keep in mind that you should never optimize prematurely, especially if you wind up with less comprehensible code. And one last thing, on a pedantic note, if you have to prove program correctness, that is often easier done with recursive functions since the function structure translates nicely to a proof by induction. However, that is usually an academic consideration that you may not find on the job everyday.

Simplifying Condition Testing with Matching

Introduction and Matching Examples

Welcome back to the course: Functional Programming with Python. My name is Gerald Britton. In most nontrivial programs, there are places where you need to take action based on input parameters, derive data or real time conditions. Traditionally, this sort of thing is done with special semantics in the language of choice, even in assembler. Python can do this today with a chain of if, elif, else statements, for example, and there's also an object-oriented pattern for this called the command pattern. Modern functional languages have developed a powerful alternate approach. It's called matching. In this module, we'll first take a look at how it works in some languages today, then get to work on implementing a similar functionality in Python, without altering its syntax, of course. Here are three examples of matching in three different languages. You may be familiar with the Structured Query Language, or SQL. I'll not get into the argument over whether to pronounce it "sequel" or S-Q-L. The language has long had a fairly decent match capability, the case expression. Reading from the top it says, return something based on the value of column I. If I equals 42, return the string, The Answer. If I equals 41, return Close but not quite. Otherwise, return the string I can't match that! In SQL, anything after the then keyword is an expression. It can even be another case expression. The second example is from a more modern language, F#. That's a cross platform. NET functional language for Microsoft. F# actually uses the keyword match to do matching. Fancy that. Reading from the top, this snippet says define a function m that takes a parameter i. The function should perform a match on the value of i, and as in the SQL example, if i = 42, return the string, The Answer. If i = 41, return Close but not quite. Otherwise, return the string I can't match that! In F#, anything after the right arrow is a statement, including a body of code, a function call, or even a new match statement. The last example is Haskell, a rich, powerful FP language. The first line declares a new function called match. The subsequent lines have the same semantics as a SQL in F# examples, though the syntax differs a bit. Now, let's look at how you might do something similar in Python, just using the standard syntax.

Demo - Matching in Standard Python

Here's a simple match function implemented in Python in two different ways. The first function uses the standard if, elif, else structure, and the second one uses a conditional expression. The

first function is about as verbose as SQL. Now you may be used to it so you don't even see the repetition. Each section repeats the parameter name i, and a return keyword. Of course, you can put an arbitrary code suite where I just have the return statement. The second function is less verbose, but the condition being tested follows the return value. This may make it hard to spot the condition itself. Also, you cannot put arbitrary code in the return expression, though of course you could call a function there. Now, imagine you had a tuple to match. In Python you might start like this. The first test only looks at the first item in the tuple. The code suite that follows might look at the second item. The second test matches on both items, and this is handy syntax. The third test matches on just the second item. Do you see what is missing here though? Maybe you spotted it right off. There is no else clause. This function would return none if all the tests fail, but should it? There are other kinds of tests you might want, such as testing for class instances, testing using a function that returns a Boolean, testing a regular expression, or testing membership in a collection. It would be nice to be able to handle all these kinds of tests through a special syntax in Python, but since there is no special syntax, let's see what we can do if we build a match class.

Demo - What You Want from a Match Class

Let's think about what we want in a match class. We want a class with a simple API that somehow mirrors the syntax found in FP languages. It needs to be visually distinctive so that it is obvious that this is a matching operation. It should handle multiple possible matches through some kind of or syntax, and it would be great if the class supports a matches everything operation. Plus, we want it to catch missing match cases to help us debug our programs. Here, in VS Code, I've written up an example of how I want to be able to use the match class. I want to put it in a context manager to make the use visually distinctive. The constructor should take an expression which is the value to be matched. I want to have a succinct API call to match on any given pattern, and if matched, I want simple syntax to invoke an action. Here I'm suggesting overloading the right shift operator for this purpose, though this surely could be done in other ways. I want to be able to use some kind of or syntax to allow for more than one match leading to the same result. Patterns should be flexible. This one is a function call, coded as a lambda. Another pattern I want to use is a type. Here I want to match on any integer. I also want a default catch anything pattern. I'll use the ellipses for that. Note that using the ellipses requires Python 3. 0 and up. I also want a way to access any matching results and a way to catch missing match patterns. Now, before looking at the match class I've written, let's see if it will work with these tests. I start off with the value 42, and if I run this, I'll get the result, The Answer. If I change it to 41 and run it

again, I'll get that the matched value is actually 41. Change it to the value 40, an even integer, that should activate the lambda, the function down there, and indeed it does. It tells me that 40 is even. Trying another pattern, any other integer, should match the fourth pattern, so let's see if it does, and indeed it tells me that 45 is an integer. Now, what if I make it a floating point number. Let's just try the first few digits of pi. I get the result that pi was not matched. I get the not matched message from the catch all pattern. If I comment the catch all pattern, let's see what happens. I want to be notified that there's a missing match case here, and indeed I get a ValueError, showing that nothing matched. This would be an indication during debugging that I had left out some important match cases. Here's an example with tuple matching. Let me just uncomment that. I used the ellipses here as a wildcard to ignore the items I'm not interested in. Let's run it and see if it works, and indeed it tells me that it matched my tuple, and similarly, if I simply change the middle 2 in the match pattern to say, 4, it shouldn't match that, and indeed it tells me there is no match. So the matching hits the catchall case and reports the error.

Demo - Match Class Walkthrough

The match class starts off in a simple way. First, I import some library tools. The functools module has a handy function, singledispatch, that we'll see in a moment. I also import some types. I'll need the sequence type and the function type. The constructor then saves the expression to be matched and sets some other local variables. There are two properties set up for accessing the match expression and any result returned. Since the match class must be usable as a context manager, I need the enter and exit methods. The exit method has some extra logic. First of all, if it was reached because of an exception, the method returns false to allow the exception to bubble up the stack. Second, if the match indicator has not been set, it raises its own exception to report that the match cases are incomplete. Following the exit method are a pair of operator methods to overload the built-in ones. The or method is overridden so that we can use the bar between match instances. We used it in this example here on this line, line six, as you see. Then the right shift method is overridden to enable the use of that operator for result processing. We use this in all the previous examples. Next, I implement the call method, which is where the real work begins. If you haven't used this method before, it makes a class instance callable as a method. This one works like this. If nothing has been matched so far, it calls the CMP, short for compare, function to compare the pattern supplied to the match expression saved in the constructor. Now, the comp function follows. Notice that it is not part of the class, nor does it need to be. It is generic and usable on its own. The comp function uses the singledispatch decorator I imported at the start. As its name implies, it looks at a single argument, the first argument, and dispatches a version of

the function to handle the type of the first argument. The first such is generic. If the pattern is equal to the match expression or is an ellipses, this is a match. The second handles functions. If the pattern is a function, then that function is called and its result is returned. You could use functions that take additional arguments by using the partial function in the functools module. The third method handles type comparisons. It checks that the match expression is an instance of the pattern type. The next method is for sequences, including tuples and lists. It recursively invokes comp for every pair of patterns and match items in the sequence. Note that if your sequence was an iterator, this would exhaust the iterator. If that's not your intention, check out the tee, that's T-E-E method in the itertools module. The last override handles match objects. This allows us to use other match objects in patterns. Now, I would not be so bold as to claim that the match class was complete, or that all edge cases have been accounted for. However, it is enough for us to add new functionality to our order processing system. So, let's do that.

Demo - Using the Match Class

To use the match class in our project, we'll add validations to the order class constructor. Exceptions should be raised if anything is missing or invalid. An order ID must be numeric, not alpha-numeric, or a floating point number, or anything else. The expedited and shipped indicators must be Booleans, not just ordinary values. Even though everything has a Boolean value in Python. The customer argument must be a customer object, and the list of order items must only contain order item objects. How can the match class help out here? Well, let's start with the traditional approach and then see how matching can clean up and simplify our code. Here you see a classic example of the traditional approach. Each condition is tested with an if statement, and each error raises a ValueError. Not much else to say really. You've probably seen or written something like it many times. I find it just a little hard on the eyes though. Probably some encapsulation is in order here. Now, here's how we'll set up validation using matching. The main match code matches on two tuples, although I could've used one big one. Matching on tuples allows for testing multiple patterns in one line and taking action based on the result. Here, I successfully test for invalid order IDs, shipping addresses, and customer references. All the patterns are also tuples except for the last one in the group. You'll notice I've set up a number of helper functions, and this is a typical FP approach. I have an error function to encapsulate the logic of raising the exception, then I have a series of functions to test types and return the negative of what the test returns. This allows me to do negative matching when I need it. The Match suite tests the types of the items. If they don't match, an error is reported, and for the shipping address, I also ensure that it is not empty. I finish up the suite with a catchall match,

which basically tells me that all is well. The second group tests the rest of the parameters in a similar fashion. If all tests pass, the constructor continues as before. Now, you may complain that this version is actually longer than the original, which is correct. That is mostly because of the helper functions, which are often a very good idea on their own. In fact, I'd recommend them for the first traditional approach that we looked at as a way to factor out common repeated code. After all, we want to obey the drive principle, don't repeat yourself. I do think that this version is easier to read though, once you understand the structure of a matching operation. Now, perhaps a future version of Python will support matching syntax directly. There has been talk about it on the Python IDE's mailing list. That will likely simplify the usage quite a bit if and when it arrives.

Summary

In this module, we took a look at matching and how it works in FP languages. Next, I showed you some typical examples of how to use if statements in Python for matching. Then, you saw an example of a Python Match class to implement the idea of matching without changing Python's syntax. It uses some operator overloads, and the singledispatch decorator to help get the job done. Then I used the Match class to add validations to the Order class in the example we've been working on. Even though there were more resulting lines of code, the use of helper functions and the Match class produced code that is easy on the eyes. Argument validation is just one way you can use matching of course. It can actually replace most or all if statements if you like. Well, that leaves the question of whether you should use the Match class or something like it in your projects. If you like FP Match syntax, you may see much to like in this approach. Contrarily, you may want to wait and see if the movement add matching to native Python syntax gathers steam. Whatever you choose, the techniques presented here may help you in many areas of projects you are working on now or will in the future. Remember the goal is not to use this or that technique per say, rather it is to write clear, correct, maintainable code. If you achieve that, the paradigm you choose is decidedly secondary.

Summary

Summing Up

This course on Functional Programming with Python has introduced some of the major concepts and patterns used in FP languages today, and shown how they can be also used in Python

without waiting for the language to directly support the more advanced ideas. Now let's review those ideas and patterns. One of the most fundamental ideas in functional programming is that of higher order functions. These are functions that can take functions as arguments and return functions as results. Fortunately, Python already supports this idea. We looked at three common uses of higher order functions. Composition, where a new function is defined that puts two or more functions together so that they can be used with one call to the new function. Closures, where some of the arguments to a function are captured, that is closed over, in a new order function. And Currying, where a function's signature is reduced argument by argument until only one argument is left, and this uses the ideas behind closures. Using these building blocks allows us to refactor our code into more manageable, interchangeable pieces. Pure functions are functions that have no side effects. Their output depends solely on their input and no external state is referenced or modified. This means that you can count on them. Give them the same inputs, they will always return the same output. Pure functions also tend to be simple. They do one thing and do it well. Complexity is obtained through a combination, using composition, for example. Immutability is sometimes the elephant in the room for procedural or object-oriented programmers looking at functional programming. This goes along with the creation of pure functions. The immutable metaclass combined with slots makes it possible to make classes enforce immutability. This means that a class is now a collection of methods along with initial state since once set, the state cannot be changed. We also change from using lists to hold collections, to using tuples, which are immutable objects in Python. In the chapter on lazy evaluation, we changed our functions that return tuples to return iterators or generators instead. This can result in using less memory and computing resources, though you have to wisely choose when and where to be lazy, especially with large shared collections. Recursion offers a way to directly translate mathematical recurrences to code. However, this can lead to pressure on the stack. You saw how to make your functions tail recursive, and then you saw how to use the tramp function to make any tail recursive function support infinite recursion. The last pattern we looked at was Matching. Matching is a powerful technique for organizing complex conditional execution. I demonstrated a Match class that can do this in Python without adding any new syntax. I hope you enjoyed this Pluralsight course on Functional Programming with Python. I also hope that it will inspire you to think functionally and apply some or all of these techniques in your projects. I'm Gerald Britton, and I hope to see you again soon on Pluralsight.

Course author

Gerald Britton

Gerald Britton is a Pluralsight author and expert on Python programming practices and Microsoft SQL Server development and administration. A multiple-year of the Microsoft MVP award, Gerald has...

Course info

Level	Advanced
Rating	★★★★★ (29)
My rating	★★★★★
Duration	1h 49m
Released	2 Aug 2017

Share course