Design Patterns with Python

by Gerald Britton

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

Hi everyone. My name is Gerald Britton, and welcome to my course, Design Patterns with Python. In case you're curious, I'm a senior solutions designer at TD Bank in Toronto, Canada. In this course, we're going to learn about the elements of reusable object-oriented software design as presented in the classic Design Patterns book by Gamma, Helm, Johnson, and Vlissides, and how to write those patterns and use them in Python. This material will take an intermediate Python programmer to the next level and lay a foundation for moving beyond simple scripting to complex projects developing full-fledged, large scale applications. Some of the major topics we will cover include a review of object-oriented design principles and the convenient acronym to remember them by, SOLID. Eight design patterns used in many programming languages today. Common programming challenges made easier by using these patterns. Typical business problems and how to solve them using the patterns we look at. And also, several IDEs with excellent tools for developing and debugging Python projects. By the end of this course, you'll have learned how to apply design patterns to break down tricky problems into simple components and create Python programs that are easy to write, easy to read, and easy to maintain. Before beginning this 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. I hope you'll join me on this journey to learn how to use classic design patterns in Python programs with the Design Patterns with Python course at Pluralsight.

Introduction to Design Patterns

Overview

Hello. Welcome to the course, Design Patterns with Python. My name is Gerald Britton, and it will be my pleasure and challenge to guide you through this study, which will start you on your way to creating more stable, maintainable, and understandable Python programs. In this introductory module, we want to answer some basic questions. What are design patterns? How were they discovered? Why do we need them? What sorts of problems do they help us solve? We'll also look at the basic classification scheme for design patterns so that we can better understand the roles and applicability of each pattern we look at. This is also a great time to review some foundational principals of object-oriented programming. This set of principles will be our guide as we apply design patterns to approve our work. We'll look at the five main principles known by their acronym, SOLID. We'll take a quick look at the tools you will need to complete this course and we'll finish this module by taking a look at some Python code to help us understand how interfaces are designed and used in Python.

What are Design Patterns?

To start things off, let's begin by answering this question. What are design patterns? A simple definition is just this: A design pattern is a model solution to a common design problem. It describes the problem and a general approach to solving it. That's it. When you think about it, these patterns, these model solutions, turn up all over the place, not just in software development. Consider this quote from Christopher Alexander, "Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to the problem. " If you guessed that Mr. Alexander is not a software designer, you would be right. He is an architect and design theorist, currently Professor Emeritus of Architecture at the University of California, Berkeley. But he has captured nicely the essence of what we do as programmers. We are confronted with problems to solve. Once we've seen enough problems, we begin to see the patterns between them and we find out that these patterns occur again and again. Patterns abound in other fields as well. Consider these. Clearly architecture has patterns everywhere. This is where Alexander first saw them and began to catalog them. Virtually every municipality has patterns that must be followed for wiring or plumbing a house, a factory, or an office building. These are strict patterns, which are set into the building codes. Imagine for a moment that there were no electrical codes. Every electrician was free to design his own wiring solution for every job. Not only would this be potentially dangerous, well, imagine that my house is wired AC and yours is wired DC. Now supposed I want to borrow your power drill. How do you think that would work out? Anyway, think about the maintenance. Any change or repair done by someone new would take far longer as the new electrician would need to understand what the design was in order to work safely. We see patterns and plenty of legislation in automobile design. And what about mobile phones? What patterns can you think of that we find there? Well this all leads us to answer the why question. Why do we need design patterns? We need design patterns to ensure that our work is consistent, reliable, and understandable. We don't want to reinvent the wheel each time we develop a new program. We want to use a pattern that is consistent with the work that has gone before, and work yet to come. Using design patterns helps us build reliable solutions since we are not solving a problem from scratch, but applying a solution to a problem seen hundreds or thousands of times before. By using design patterns, our work will be recognizable and understandable by those who come after us to change or fix our programs. Always keep in mind that most programs will be maintained and changed long after they are initially developed. Make it easy on those who come after you. In this course, we will look at design patterns that fall into three basic categories. Creational patterns are concerned with, well, creating things. In an object-oriented program, that means creating objects. We'll look at patterns such as factory, builder, and singleton that fall into this category. Structural patterns help us design the relationships between objects. Adapter, facade, and composite are good examples of structural patterns. Behavioral patterns are used to help with object interaction and responsibility. Patterns such as command, observer, and strategy are part of this group. Note that there are other categories of design patterns that are not covered in this course. For example, concurrency patterns, which are concerned with multi-threaded systems, are patterns to be discussed in an advanced course. Software design patterns owe almost everything to this book, Design Patterns Elements of Reusable Object-Oriented Software, first published in 1995 by the now famous Gang of Four, Gamma, Helm, Johnson, and Vlissides. This was the first comprehensive work on software design patterns. Though many more books have been written on this subject, the Gang of Four volume is still considered the authoritative reference.

Object-oriented Programming Fundamentals

In fact, this course would look very different without this work, if the course existed at all. SOLID is an acronym to help us remember five basic principles we should try to follow. The first is the single responsibility principle. This means that a class should have only one responsibility, so either cooking or washing up, but not both. The second is the open/closed principle. A class should be open for extension, usually by inheritance, but closed for modification. Imagine you started to use a class I had designed, only to find out later that I had changed it. You'd be frustrated, and rightly so. Next up is the Liskov substitution principle, named after Barbara Liskov. It tells us that subclasses should be able to stand in for their parents in a program without breaking anything. The interface segregation principle tells that many specific interfaces are better than one do-it-all interface. In Python, we use abstract base classes combined with multiple inheritance to achieve this effect. Dependency inversion says that we should program towards abstractions, not implementations. Implementations can vary, abstractions should not. You'll need some basic tools to complete this course. Naturally, you'll need Python. There are a few flavors of the language implemented with different goals in mind, however, recall the dependency inversion principle, we should not program to one specific implementation. For this course though, we recommend the most recent version of Python in either the 2. x 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, others are commercial. One simple IDE comes with Python, IDLE. It is sufficient for the material in this course, though the feature sets of other IDEs can be compelling. Some popular third-party IDEs are PyCharm, Wing IDE, PyDev for Eclipse, and Visual Studio, though there are many others. The supplementary materials for this module contain more information. Online, you can visit the Python wiki at the link shown to get a good oversite.

Interfaces in Python

Before we leave this module, let's take a look at how Python defines and uses interfaces. Interfaces are the I in SOLID. If you know other popular object-oriented languages, like JAVA, C#, or VB. NET, you already know what interfaces are and how to define and use them. If you know C++, abstract base classes are used for the same purpose. Originally, Python had no out-of-the-box way to define interfaces. You could define a class and use it as a mix-in, but that is actually an implementation. Recall that we want to develop towards abstractions, not implementations. The need for implementations is served in Python by abstract base classes, which were introduced through the Python Enhancement Proposal, or PEP, number 3119. Support for ABCs first appeared in Python versions 2. 6 and 3. 0 in the fall of 2008. Let's see how to define them. To define an abstract base class, we first need to import the module abc. Note that if your version of Python is less than 2. 6, you will need to upgrade to get this module. Once the module is in place, we begin a standard class definition with one addition. We need to tell Python that this class is abstract. To do that, we assign the dunder attribute metaclass to the class definition for ABCMeta, which lives in the abc module. Why do we call it dunder? Because of the double underscores. Double underscores equals dunder. We continue on and define an abstract method using the abstractmethod decorator from the abc module. In a similar fashion, we can just find an abstract property using the matching abstract decorator. There are more things that you can do in an ABC, but this is enough to get us started. Note that in Python 3. 0 and up, abstract base classes are a little different. Instead of setting the metaclass attribute, we need to use the metaclass= parameter in the class definition instead. Now that we have an abstract base class, let's see how to implement it. To start off, we inherit from the ABC. This will enable the abstract base class special processing when we instantiate the class. We'll see one consequence of that logic shortly. We'll add the standard constructor, the dunder init method. Now we need to implement the method and property we declared as abstract. This is really quite straightforward, just define them. Oh, and for the property, don't forget to decorate it with a property decorator or your class will not work as expected. When we instantiate the concrete class, the ABC machinery kicks into play and checks that all our abstract methods and properties have been implemented. This is analogous to the compile time checks found in other object-oriented languages. Now let's see what happens if we forget to implement the abstract methods. Here, we've defined a class called BadClass that is missing the implementations. When we try to instantiate it, we get an exception thrown by ABCMeta that is invoked at instantiation time, and this tells us what we failed to do. So the abstract base class mechanism gives us a way to define interfaces and implement them separately. It also provides important checks on our implementations to ensure that they are complete. This is the D of SOLID. In summary then, what have we learned in this module? First, we looked at what design patterns are in general and how they have come to be recognized. We've also seen why we need them and how they help us build better programs. We've reviewed some of the basic principles of object-oriented programming, especially those captured by the acronym SOLID. We've seen the tools you will need to complete this course. Be sure you get them set up before you continue. Lastly, we looked at how to build interfaces called abstract base classes in Python. One thing to keep in mind though, Python is a dynamic language. Introspection is always available. It is possible to completely bypass or override the ABC machinery. However, doing so breaks the Gentleman's agreement that all good Python programmers endeavor to keep. Now that we've got the basics out of the way, let's dive into our first design pattern.

Design Patterns with Python

Overview and Motivating Example

Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we will be learning about one of the most used patterns, the Strategy pattern. The Strategy pattern is classified as a Behavioral pattern, so it is used to control the operation of some object. In this case, the pattern provides a way to take a family of algorithms, encapsulate each one, and make them interchangeable with each other. Since the algorithms form a family, they will normally operate with the same set of inputs and outputs, and this is often accomplished by passing in some common object as input, as we'll see shortly in our example. The algorithms are allowed to vary independently and their implementations can be quite different. Consider the differences between calculating gravitational attraction using Newton's formula as opposed to Einstein's general relativity. Very different, yet bot have the same inputs and outputs, and both should yield the same result, at least on Earth. Sometimes the Strategy pattern is called the Policy pattern, reflecting its use in applying some sort of policy to the behavior of a software system.

Demo 1: Initial Attempt

For a motivating example, imagine that you are working on a system that processes customer orders. When it's time to ship the order, it's important to calculate the correct shipping cost. For our example, we'll consider a shipping cost calculator that does just that. According to the specifications we've received, it must support shipping by three methods, Federal Express, United Parcel Service, and the normal Postal Service. Also, whatever we do must be extendable. That is, new shippers may enter the market that we also want to be able to use and for which we need to calculate shipping costs. Let's look at one solution to do that. For this demo, I'll be using the integrated development environment that ships with most Python distributions, IDLE. If you're running a system that has Python installed, chances are you also have IDLE. The program you see before you is one I put together in an initial attempt to solve the problem of computing shipping costs. After the imports, there are three sections, one for each shipping method. In each section, I instantiate an order object specifying a shipping method, then I instantiate a shipping cost object, which will do the actual number crunching. I compute the cost, and finally, I check that the computed costs meets my expectations. Let's run this program to see if it works. If all three assert statements pass, the program will print a tests passed message to confirm, so here we go. First, we'll start up a Python shell. There it is. Now let's run the program. The tests all passed. Just what we wanted. Now let's take a look at the objects being used. First we have the Order class. It is a simple class that holds a single property, the shipper for the order. Looking at this though, it appears that this might violate the S in SOLID. An order should be about who is ordering what. Adding the shipper seems to go against the single responsibility principle. Next, here is the Shipper class. It is really just an enumeration. Note, starting with Python 3. 4, there is an enum class that can be used for this sort of thing. Here we just assign integers to each shipper name. Finally, this is the ShippingCost class. It uses the default constructor and then defines a shipping_cost method to do the actual work. The method computes the cost according to the shippers stored in the order. If the shipper is not one of those the method is programmed to handle, it raises an exception. For each shipper, this method calls a private helper method to get the actual cost. Now, what if a new shipper is to be added? Well, we'd have to modify this class, adding a new elif condition and a new helper method. If you haven't guessed it already, this violates the open/closed principle, the O in SOLID. In fact, whenever you see a long sequence of if, elif, else statements like this, take a closer look. There may be a better way. Back in the main program, there is another, more subtle violation. When I instantiate a new cost calculator, I am programming to an implementation, that of the ShippingCost object, not an abstraction. This violates the D in SOLID.

Problems Discovered and Introduction to the Strategy Pattern

What kinds of problems did we discover in our work? For one thing, there's a violation of the single responsibility principle. An order should probably not be concerned with how the products will eventually be shipped. There's also a violation of the open/closed principle, since we would need to modify the ShippingCost class to add new shipping methods or shippers. There's also a violation of the dependency inversion principle, since we are programming to a concrete class and method, the ShippingCost class and its shipping_cost method. We also have a long list of if/elif clauses that is a bit fragile and not very attractive. In fact, whenever you see code like this, it's good to ask yourself if this might be done better using the strategy pattern. So, let's fix these problems. This UML diagram shows the structure of the Strategy pattern. Within some context, we have an interface, which is an abstract base class in Python terms that is common to all the supported algorithms. The context uses this interface to call the various algorithms defined by the concrete strategies that implement the base class. Each concrete strategy implementation takes the same inputs and returns the same type of output, but is free to do so as the algorithm being implemented dictates. This diagram shows how the pattern is implemented for our application. The context will be a ShippingCost class, which uses an abstract base class, called AbsStrategy, to execute the various concrete strategies, one each for Federal Express, the Postal Service, and UPS. Setting it up this way also makes it very easy to add new shippers since we would only need to add new concrete strategies, everything else remains the same.

Demo 2: Using the Strategy Pattern

In this demo, we will look at the implementation of the Strategy pattern for our application. Here in IDLE, you can see the context, the ShippingCost class. There's not much to this version compared to the first attempt. This class takes in a strategy object in the constructor and saves that reference for later. Then, in the shipping_cost method, it uses that reference and executes its calculate method on the order supplied. The strategy object should implement the abstract base class, AbsStrategy, which looks like this. It follows the template discussed in the introductory module for this course, defining one abstract method, the calculate method. Now for our concrete strategies, we need one class for each strategy we are implementing. One for Federal Express, one for the Postal Service, and one for UPS. Each of these must implement the abstract method defined in the abstract base class. For this demo, the calculate methods simply return the value we're looking for. Now with this structure in place, let's look at the main program. It starts out by importing the necessary objects, then has a section to test each of the shipping methods as before. This time however, we instantiate a new concrete strategy object and pass that to the ShippingCost class, which knows how to talk to any object that implements the abstract base class called AbsStrategy. Finally, the cost is computed and compared with the desired result. Let's run this. Great. All our tests passed. Just what we need. So what has been achieved by applying the Strategy pattern? First off, the problems in the original version have been fixed. There are no more SOLID violations. Now, since each algorithm is separate, it is easy to test each one in isolation using simple calling code and objects designed to hit the corner cases. It is now also easy to test the outer code with deterministic mock or fake algorithms. For example, in the demos, constant return values were used. And finally, the if/elif/else statement is gone. Recall that this is a sign that there may be an opportunity to apply the Strategy pattern. Now the sharp eye may have noticed that we haven't completely fixed the dependency inversion problem. There is still a direct reference to an order object, for example. Fixing that will involve another pattern, so be patient. Or, skip ahead to the factory pattern if you are an impatient type.

Demo 3: Variations

There are other variations of the Strategy pattern. We'll look at a few. Since functions are first class objects in Python, it is easy to encapsulate strategies and functions. Also, Python has a handy lambda syntax for short, unnamed functions. This provides yet another variation. Let's see that. For the final demo in this module, we will reuse the Order class from before and make a small modification to the ShippingCost class. This time, the ShippingCost class expects a callable function instead of an object implementing AbsStrategy. When the shipping_cost method is called, it in turn calls the strategy function passed in instantiation time. Back in the main module, functions in lambdas will replace the strategy objects used before. Here, the fedex_strategy is defined as a function that simply returns the value 3. The ups_strategy is defined using lambda syntax, though this is not considered the best Python style. Then, for the Postal Service, another lambda is used, this time directly in the instantiation of the ShippingCost object. As before, if everything works as expected, a Tests passed message is printed. I guess I'd better run this to see if I get that message. It worked. So, here is a variation of the Strategy pattern using functions in lambdas instead of class definitions.

Summary

So what have we learned by studying the Strategy pattern? We've seen that it is a simple way to encapsulate algorithms and separate them from the context where they operate. There are various techniques available. You can define a separate class for each algorithm that implements an abstract method, you can define functions to do the work, and you can also use lambda expressions for simple, one line algorithms. A sequence of if/elif/else statements can be a sign that the code may be better implemented using a Strategy pattern. The Strategy pattern is an easy one to learn. It's also an easy one to recognize by others using design patterns with Python. Add this one to your toolkit.

The Observer Pattern

Overview and Motivating Example

Hello again, welcome back to the course, Design Patterns with Python. My name is Gerald Britton and in this module we will be learning about a pattern that is heavily used for event monitoring, the observer pattern, the observer pattern is a behavioral pattern, so it used to control the operation of some object, in this case, the pattern provides a way to define a one-to-many relationship between a set of objects, so that when the state of one object changes, all its dependent objects are notified. This pattern is also known as the dependents pattern or the publish-subscribe pattern, the last name yields a natural example, if you subscribe to a newspaper or magazine, whenever a new edition is published, you'll receive a copy in the mail. If that sounds a little old-fashioned, well, it's the same idea at work, when you subscribe to a Twitter feed or a YouTube channel or an email list or indeed any sort of push service.

Demo 1: Initial Attempt and Introduction to Wing IDE

For a motivating example, consider a dashboard application for a technical support center, the dashboard will need to show Key Performance Indicators or KPIs, which may include the number of open tickets or issues, the number of new ones in the last hour and perhaps the number of tickets closed in the same period. In this example, the dashboard is the observer, the source of the KPIs is the publisher or subject. Keep in mind that other observers may be required later, perhaps one that shows historical averages of the same data or another that shows forecasts for the next hour or day, so the solution must be flexible, let's look at one way to do that. For this demo, we will be using the Wing IDE, a commercial IDE with many advanced features. A brief tour of Wing is probably in order. The main section of the screen shows the file currently being edited, there is good code highlighting for easy identification of identifiers, keywords, strings and so on. Open files are arranged in tabs above the file display and it is easy to move between them with a single mouse click. In the top right corner of a file window is a stick pin to keep a file open in the editor, a red x to close the file and a down arrow for other functions. F1 opens a toolbar at the bottom, there are many tools available, but here we have three of them, one for Debug I/O, where program output can appear, an Exception stool for tracking exceptions and a Python Shell, which is handy for interactive commands. F2 opens a toolbar on the right and it can be moved to the left if you like, here just one tool is showing, the Project tool is a tree view of an open project. With that little introduction to the IDE, let's look at the first attempt to solve the problem of the current KPI display. First, we import the KPI data, which looks like this, it simply constructs an iterable of named tuples with fixed test values, so that we can easily access them. The main program loops through the KPIs finding and displaying the values for open, new and closed tickets, pretty simple, let's run it. In Wing IDE, Shift F5 will run the program you have open. The program ran and it's complete, now if I open the Debug tool, we can see the program output showing that our program correctly reported the KPI values. Well, what's wrong with this then? Plenty. What would I need to do to also send the results via email to my boss? Then suppose I was asked to send the results to a REST API on some web server, after getting that going, what if I'm asked to also include a new KPI, the number of active support techs? This will get real ugly real fast. The observer pattern was discovered as a way to handle just this sort of problem.

Structure of the Observer Pattern

The high level design of the observer pattern is fairly straightforward, first there is some subject, in the first demo, this was the simulated system producing KPIs, then there are one or more observers, that are interested in changes in the subject, the demo has one observer interested in current KPIs. The observer pattern sets up the mechanism for observers to attach to and detach from the subject, in order to obtain update notifications, once a notification has been received, each observer has the opportunity to get state information from the subject, but the pattern also allows for setting state on the subject, so the subject handles the attached observers and notifications, while the observer handles updates received from the subject. Let's look at the UML diagram of the observer pattern. First there is an abstract subject, that contains the required methods attach, detach and notify. Next we have the abstract observer with its one required method, update. The concrete subject implements the required methods, attach, detach, and notify, commonly it also implements a getState method for observers to use to obtain the subject state when it changes, the concrete observer implements the update method. Now when the subject's state changes, it loops through all currently attached observers and calls their update methods, when called, the observer's update methods call the subject's getState method to acquire the changes, in response, the subject returns the state requested. Following the observer pattern, we'll separate the concerns of the subject, object and main program, doing that falls in line with the single responsibility principle, the S in solid, by creating abstract-based classes for the subject and object, we are also following the IM solid, the interface segregation principle and obey the open/closed principle, the O in solid, by programming to ABCs, we have inverted our dependencies, the D in solid and the principle encapsulates what varies is also in force, since each observer is self-contained.

Demo 2: Using the Observer Pattern and One Subtle Bug

For our second demo, we want to implement the classic pattern and code, before we look at the code though, let's review what we want to achieve, we want to implement the classic pattern and that means we want to build abstract base classes for the subject and observer, use those abstract classes to build concrete classes for a KPI displayer, rework the main program to use the new pattern and show the separation of concerns, while we're at it, we'll add a second observer to display forecasted KPIs. In the observer project in Wing IDE, the project menu shows the main components. First, there is a package called observer, opening it, you can see that there are two ABC modules, the first one is the abstract observer-based class with a single abstract method required, the update method, the second is the abstract subject-based class with the three required methods, but wait, these are not abstract, they're actually implementations, in Python, abstract-based classes may indeed have implementations, the function of these methods is simple and standardized, so implementing them here makes sense, so is this still an ABC? Well, yes, from a metaclass perspective, the class has a single attribute defined observer, which will hold the set of references to observer objects. The attached method first checks the type of the observer argument, then adds the new observer to the set, the detach method simply removes the observer from the set, the notify method loops through the observers in the set and invokes the update method in each one, but there's a twist, the notify method takes in an optional argument called value, if supplied, the observer's update methods are called using that value, we'll come back to that later. In the KPIs module, the KPIs class inherits from the abstract subject, it defines properties for the values to be tracked, a set KPIs method is included to set the attributes to new values and when new values are set, the notify method is called. The current KPIs module inherits from the abstract observer and takes a reference to the subject in the constructor, the update method uses this reference to retrieve the needed values from the subject's properties, then displays the results. The forecast KPIs module is much like the current KPIs module, except for the display method, note that in a practical implementation, this observer might need to track the KPIs in order to do some sort of forecasting, perhaps using a linear regression technique or something like that. The main program imports the necessary classes, then starts its work, first it instantiates the KPIs object, which is our subject, next it instantiates the two observer objects, the KPIs are set to various values, then the first observer is detached and once again, new values are set. Now let's run this program and observe, pun intended, the results. Hit Shift F5, now the program has finished, so let's open the Debug I/O tool. Here we can see output from both observers, for each observer, the values are retrieved and printed, the forecast KPIs, the current KPIs and so on, but look, after we detach the current KPIs observer, no more results are displayed by it, since it no longer receives notifications. So what has been achieved so far? The classic gang of four observer design pattern has been implemented in Python, we've separated the concerns of the subject and observer, which greatly simplified the main program and allows the subject and observers to vary independently. Recall that a key principle of object oriented programming is to encapsulate what varies, it is now also easy to add new observers and keep them separate from each other and the subject being observed, there is however one subtle bug, that needs to be addressed before going any further. Like many modern languages, Python runs managed code, among other things, this means that it needs to keep reference counters for objects in use, when the references go to zero, the storage used by an object can be recycled or garbage collected, in our case, the subject keeps a set of observer references and that means we need to detach each one, why? Well, if we don't do the detach, the reference count will stay greater than zero, this can stop garbage collection, a bug like this is often called a dangling reference.

Demo 3: Fixing the Bug with Python Context Managers

In this demo, Python context managers will be used to provide a way to ensure that observers are properly detached. The main program will be changed to use Python with statements, when accessing the subject and its observers, this way, when an observer is done observing, it will detach itself automatically, even if an exception is raised during processing, not only that, the subject will take care of cleaning up any leftover observers when its context exits, as a consequence, there should be no more dangling references, let's see this in action. Back in Wing IDE, the observer project has gained a context managed cousin, the ContextObserver. Let's look at the changes to the abstract-based classes, the abstract observer class has gained two new methods, the enter and exit methods are required to turn a class definition into a context manager, note that the exit method is abstract, so it must be implemented by the concrete observer, the abstract subject class also is now a context manager, here we implemented the exit method and cleared the set of observers, thus removing any dangling references, the current KPIs concrete observer implements the exit method, now Python ensures that the exit method will run when the context ends, so we can use this to take the opportunity to detach the observer from the subject and we do the same thing with the forecast KPIs observer, the main program has also been modified to take advantage of the context managers by using the with statements, let's run this version, Shift F5, the program has exited, let's look at the output. Looking at the output, it is apparent, that after the context managers have exited, the observers are no longer notified, for there is nothing displayed by either one.

Summary

In summary then, what have we learned by studying the observer pattern? We've seen that it is a simple way to define a one-to-many relationship between objects, so that when the one changes, the many are notified. The observer pattern is heavily used in many sorts of applications today, perhaps the heaviest use is in GUI apps, whether on a website, your desktop or your mobile phone, where events like keyboard entry and mouse movement or finger taps are tracked using observers. The classic Model View Controller pattern uses the observer pattern, where the model is equivalent to the subject and view is equivalent to the observer, oh, one more thing, recall the extra logic in the abstract subject-based class I promised to come back to, this enables support for push notifications, where data is sent to an observer's update method, see the assignment in the exercise files for details.

The Command Pattern

Overview and Motivating Example

Hello again, welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module, we will be learning about the Command Pattern, a pattern you'll find in toolkits, command-line programs, GUI menus, in fact, just about anywhere you want a simple way to corral diverse request processing objects into a common structure. As in other modules in this course, we'll be applying the principles of object-oriented programming, known by their acronym, SOLID. If you need a refresher, you can review the introductory module. The Command Pattern is a behavioral pattern, that means, it's used to control the operation of some object. In this case, the pattern provides a way to encapsulate a request as an object. This is in line with the principle, encapsulate what varies, and this encapsulation lets you parametrize various objects with different requests. The Command Pattern also provides a simple way to support queues and logs, perhaps for updating a database or creating an audit trail of requests. Using the Command Pattern, we can bacon support for undoable operations, and for macros, that is, sequences of commands. Sometimes, this pattern is also known as the Action Pattern or the Transaction Pattern.

Demo 1: Initial Attempt and Introduction to PyCharm

For a Motivating Example, consider a command line order processing system. Three operations must be supported, create an order, update the quantity ordered, and ship the order to the customer. The program must parse the command line arguments, then it must execute the command, whatever it is, finally, it must notify the user of the results and log them for audit purposes. Keep in mind that other operations may be required later. For example, perhaps someone will request a delete order function. Let's take a look a one possible solution. For this demo, I will be using PyCharm from JetBrains, a popular Python IDE with plenty of help for the programmer. It comes in a few editions. I'll be using the free Community Edition. Let's take a brief tour around this IDE. Here is a typical PyCharm session. I have an example Python file in the editor which occupies the main portion of the screen. Open files are arranged in tabs above the file display, and it's easy to move between them with a single mouse click. The left window shows the project structure. It's easy to open and close subfolders, and move between files. This window can also be hidden by clicking the Hide icon. Now, you might have noticed a little orange light bulb near the definition of method F. If I hover over that, a little blue arrow pops up, and if I click the arrow, PyCharm gives suggestions for possible improvements. We won't take that suggestion here. Now, we'd like to run this program. Just above the Edit window, there are two icons, a green arrow, and a green bug. If I hover over the green arrow, PyCharm offers to run the program. Similarly, if I hover over the bug, PyCharm offers to put it in Debug Mode. But before we run it, let's add a little code so it does more than just define a class. First, we'll instantiate a class. Then we'll obtain the results by running method F. And notice how PyCharm does a lot of autocomplete work for us. Finally, we'll print out the results. Now we have a complete program, doesn't do much, but it's complete, so let's run it. When it finishes, the Run window opens up at the bottom, showing the results. Note that there are other windows at the bottom, as well, including a Python Console for interactive use, a Terminal window for an operating system shell, a TODO list, and a Debug window for when you're in Debug Mode. Like the left window, the bottom window can also be hidden. Oh, but before we do, note that our program actually worked, we obtained the results we wanted. Now, let's use PyCharm to try to solve our order command processor problem. The first attempt starts off with a pair of imports, one to give us access to command line arguments, and another to import the class created to do the actual work of executing the commands. The program first checks to see if there are any commands at all, and prints a short usage summary, if none are found. If there is a command, the program instantiates a new CommandExecutor class and gives it the command to run. Let's look at that class. Okay, the CommandExecutor class is a little big, probably too big. Note that the execute_command method uses an if, elif, else statement to dispatch helper methods. You can see that only the update_quantity helper method is implemented so far. It simulates a current quantity called old value, just for testing. Well, let's run this program and see if it works. The first time, it just prints the usage summary, since we've given it no_command to process. However, in PyCharm, we can do that in the Run, Debug configuration. So Run, EditConfigurations, put in a new command. OK that. Let's run it again. We can see now the results of the execution. The update_quantity method did exactly what we wanted. Apparently, we wanted to order pi. Now, back in the CommandExecutor class, notice again the if, elif, else statement. This sort of thing is usually assigned that there may be a better way. In fact, the Strategy pattern also helps with a similar class of problems, though it's not a good fit here, since each helper method has a different signature. This class also has too many concerns. It both parses the arguments and handles invalid ones, and then executes them. Also, adding, changing, or deleting commands would require changing this class, in violation of the open-closed principle. So what kinds of problems did we find in our first attempt? Well, there's a violation of the Single Responsibility Principle. The CommandExecutor class parses commands, and then processes them, these concerns should be separated. The class also violates the Open/Closed Principle, since we would have to change it to add new commands or change or remove existing ones. The main program also violates the Dependency Inversion Principle, since it depends upon the implementation of the ExecuteCommand method in the CommandExecutor class. We also have a long list of if, elif, else clauses that's a bit fragile, and a little hard on the eyes. In fact, whenever you see code like this, it's good to ask yourself if this might be done better using the Command Pattern. Let's see now how that design pattern addresses these issues.

Structure of the Command Pattern

This is the structure of the Command Pattern in an UML diagram. It may look a little complicated at first, so let's dig into it piece by piece. Top left is the Client. This is the part that wants to get something done. For the command line order processor, that would be you or me sitting at the keyboard. For some application with lots of bells and whistles, the application itself is the Client. Next to it, is what we call the Invoker. This component asks the command to perform a request. In the command line order system, this is the main program. Bottom right is the ConcreteCommand. It knows how to perform the action requested, and may enlist the help of a Receiver object, or may go it alone, if the action is simple enough. In the command line order system, there should be a ConcreteCommand for every command in the system. The Receiver could be a separate object or just a method in the Concrete class, depending on the complexity. The ConcreteCommand fully encapsulates the command so that the Client is no longer concerned with the particulars. The Undo method is optional, though it is frequently found in many applications. Think of a Python IDE, for example, where Undo is usually available from the Edit menu. Okay, so what do we get by following the Command Pattern? One important thing is that the commands are encapsulated in separate ConcreteCommand objects. Each one knows how to process its command, and that's all it needs to do. Note though, that the Client doesn't pass in arguments to the Execute method. That information is hidden and allows the Client to invoke any command without knowing any details. It's also easy to add new commands to the system. Just write new ConcreteCommands.

That means we're following the Open/Closed principle. Remember the original CommandExecutor class? We'd have to open it up to add or remove or change any of the commands. Okay, so now we're going to look at how to do this in code. We're going to reimplement the order processing system to produce the same results as before, this time, however, we're going to use the Command Pattern to do it. To put it all together, we'll then rebuild the main program, which is the Invoker in the UML diagram to use the new pattern. In the Command Project in PyCharm, we can see the pattern begin to take shape. Here is the abstract base class for the ConcreteCommand objects. It declares the one required method, execute. Note that we're not adding Undo support at this time. There's also a second abstract base class, the AbsOrderCommand, which we'll use as a mix in to add additional properties to the ConcreteCommands. This will require us to implement name and description values. Notice also in the Project window, that there are separate modules for each command, create_order, update_order, and ship_order. There's also something new, no_command, to which we will return in a moment. As before, I've only implemented the UpdateOrder command. Note that includes implementations of the abstract properties, name and description. You might be wondering how this works without using the property decorator and a method definition or two. Well, the Python abstract base class machinery only checks that something is implemented in the Concrete class. It does not check that the implementation matches the declaration in all aspects. Indeed, in a dynamic language, this could be quite costly at runtime and may be impossible in some cases. But here, this actually serves my purpose, since what I want is constants assigned to those properties. You'll see why in a minute or two. The UpdateQuantity class goes on to define the required Execute method, which looks much the same as before, except that the new value has moved to the dunder init method. This is required in order to expose a parameterless method to the Client. So, let's look at how the main module uses this pattern. To start off, it imports all the ConcreteCommand classes that have been defined. Now, you may be thinking, doesn't this violate the Open/Closed principle? Well, yes, actually, it does, since we'd have to change the main program for changes to the Command Suite. However, this could be easily overcome by putting all the ConcreteCommands in a separate package, and then loading whatever commands are found there at runtime. Now, using the ConcreteCommand classes, the get_commands function builds a dictionary, where the keys are the names of the commands, and the values are the class references. This is what we wanted to use that name property for. There's also a new function, print_usage. That is used when no commands are supplied. However, in this case, it uses the dictionary, created by the get-commands function, and that description property. The parse_commands function uses the setdefault dictionary method. This method looks up the commands supplied, which is in arg zero in the dictionary. If found, it returns a reference to that class. If not found, it returns a reference to the no_command class, and here's what that one looks like. What it does is saves the command entered, whatever it is, and then when the Execute method is run, prints an error message, and exits. This is an example of another design pattern, the Null Pattern. Using this pattern means that the caller does not have to check for null results, or for None in Python. The caller can simply use the object as is, since it implements the required methods and properties. It's hard to emphasize how useful this pattern is. Code can be much easier to read and write, if we can dispense with null object checks. And this is exactly what the Null Pattern gives us. Back in the main program, the parse_command function returns a new instance of command found, or no_command. All the main program has to do then, is execute the command, so let's run it. Our program works, the quantity has been updated to a value of 42, which Douglas Adams calls, the answer.

Demo 2: Using the Command Pattern

So then, what does the Command Pattern give us? It's a great way to encapsulate behavior so that we can separate the command logic from the Client issuing the command. It's a simple pattern to use when building command line programs, as we've seen in the demos we worked through. It's also really helpful when adding additional capabilities such as Validation or Undo. It's easy to imagine that the arguments to a command have to be validated before the command is executed. Exposing a single consistent way to call for validation fits this pattern nicely. You'll find it useful when building menus and applications since each menu item can be considered a command. The assignment in the exercise files shows yet another variant, and includes Multi-level Undo, just for fun.

The Singleton Pattern

Introduction to Singleton

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we'll be learning about the Singleton pattern, a pattern useful when you need to control access to class instances. As in other modules in this course, we'll apply the principles of object-oriented programming known by their acronym, SOLID. If you need a refresher, you can review the introductory module. The Singleton pattern is a Creational pattern, and that means it's used to create objects. In this case, just one object. Singleton can be used to ensure that a class has only one instance. This is handy when you need to control access to a limited resource such as a hardware device, a set of buffer pools, or perhaps connection pools for a web client or database access. Singleton also provides a global point of access for its one instance, and is responsible for creating that instance. But this is not always a good thing, as we'll see later. Singleton can also provide for lazy instantiation, which can be important if the object is costly to instantiate, or may not always be used.

Demo 1: Logger Example and Introduction to Visual Studio Code

For a motivating example, consider a logging subsystem for some application. Events, errors, warnings, and other messages must be logged to a file, perhaps for auditing or debugging. While there can be only one instance controlling the log, one place to talk to the file system that is, and that means we need to control access. This is a classic use case for the Singleton pattern. Let's look at one possible implementation. For this demo, I'll be using Visual Studio Code, a relative newcomer to the crowded IDE space. Code is a free, lightweight IDE from Microsoft, and runs on Windows, Linux, and OS X. This is a typical Code session. I have a simple Python program in the editor, that just prints the version. Open files are arranged in tabs, which you can see above the Edit window, and navigation between them is easy. On the left you can see five icons which open different window panes. Topmost is EXPLORER, where you can see your open files, and all files in the current directory. In VS Code, projects and directories are equivalent. Under EXPLORER is an icon for Search/Replace, which functions pretty much as you would expect. The third icon shows GIT access, which we won't be using in this demo. Under GIT is an icon for debugging, which we'll use to run our programs. Note the four different sub-views for VARIABLES, variables to WATCH, the CALL STACK, and BREAKPOINTS. The last icon is for access to EXTENSIONS. There is a large and growing extension library. Python itself is an extension, as are other programming languages. So let's run this program. If I click the debug icon, the view in the left window changes. Above it you can see a green arrow, which I can use to run my program. Let's do that. Note that the debugger stops just before starting execution. This is actually a configurable property found in the file, launch. json, which looks like this. In fact, all configuration in this IDE, is done using JSON files. Here we can see the attribute stopOnEntry is set to true, which is why the debugging session stopped where it did. This can be changed to false, if you prefer. Back in my sample program, I'll click the green arrow, the program runs, but where is my output? We can see that in the Debug Console. You can see that I'm running Python 3. 5. 2 in this session. Now, perhaps you don't like the dark screen. No problem. We can change that. If we go File, Preferences, Color Theme, we can change it to the Light theme. And there we have a theme with a white background instead. But as for me, I prefer the dark background and that's what we'll be using. Okay, now that we're a little familiar with the environment, let's look at the classic Singleton pattern. In the classic pattern, Singleton objects are not instantiated like other objects. The class definition provides a static instance method to use to access the one instance. This method does a little reflection and looks to see if the variable, instance, is already defined in the class dictionary. If not, it instantiates a new one and saves that reference in the instance variable, which also causes it to be added to the class dictionary for the next time around. To prove that the Singleton does what it's supposed to do, we call the instance method twice for two different variables and test their equality. In Python, the is operator checks that the two objects are actually the same. Scrolling down, you can see that I've also set the s1 variable ans, to the value 42, then asserted that it's the same in s2. If Singleton works as advertised, then all the asserts should pass. Let's run the program to see if they do. Down in the DEBUG CONSOLE, you can see the Assertions passed message, so it works. Here is the Singleton pattern implemented for the logger class we want. It's the same as a Singleton class we just looked at, except for the addition of the methods open_log, write_log, and close_log, which do the actual work of writing to the log. I've also included logic to test this. First, get the one instance, then open the log, write a record to it with a timestamp, and close the log. Finally, I have a few lines to read the log and print it out to see if it actually got written. Running this, you can see that it works as expected. Now that we have a working Singleton logger, let's pause and take a look at some of the downsides. What's wrong with Singletons anyway? Well for one thing, Singletons violate the single responsibility principle. They do two things. They look after their own instantiation, then hold and process that state. The logger class for example, instantiates itself using the instance method and also keeps the state of the logger file object and performs open, write, and close operations on it. They also have non-standard class access. To get an instance, you have to know that the class is a Singleton, and use the instance method. Well that's fine if you write the code and maintain it forever, but just how likely is that? They're also harder to test. Since Singletons are tightly coupled with the objects that use them, it is hard to swap them out with fakes or mocks for unit testing. Singletons carry global state, like that log file reference in the classic logger class. In a sense, that's no better that globals generally, which have well-known testing and maintenance problems. Also, because the class carries state, it is harder to sub-class or use for other purposes. There are a number of good write-ups of Singleton problems which are worth looking at. Here are just two of them. Well because of all of these things, Singletons are sometimes called an antipattern.

Demo 2: Building a Singleton Base Class

For the second demo, let's fix the single responsibility problem. We'll do that by building a base class for all Singletons, and then inherit from that class for each one we need. We'll also fix the non-standard instance access. Note that this does not fix all the problems with Singletons. The other problems, global state, testing issues, and maintenance issues, still remain. For the second implementation, we are going to use a base class that does the work of managing the instances, and inherit from it for each new Singleton we need. The Singleton base class keeps track of all Singleton instances in a dictionary where the keys are class references and the values are instance references. This way we can reuse this base class for as many Singletons as we need. Also we do have to find the dunder init method, but rather the dunder new method. This method is invoked each time a given class is instantiated, but before init. The class works in a similar fashion to the first attempt. It checks if the class is already present in the dictionary, if not, it instantiates the class for the first and only time, then adds that instance to the dictionary. Finally, it returns the instance from the dictionary. Using this new base class, we create a new Logger class that inherits from the new Singleton base class. The init method opens the log, so we don't need a separate open_log method anymore. The other methods are as before, and I've added some extra logic to manage the reference to the file object. To test this version, I've got a simple main program that imports the new class. There are two variables assigned to the logger instantiation, note that the second one, logger2, uses the name ignored. If the Singleton pattern works correctly, this name will indeed be ignored, since the file will already be open. Now let's run this program. The DEBUG CONSOLE, if I expand it, shows that two records were written to the one file as desired. So the Singleton pattern works as a base class as well. Now we've separated the concerns between the Singleton class and the Logger class. Though it might get a little messy if we need multiple inheritance, is there another way? There is.

Demo 3: Singleton as a Meta Class

In the first demo, we used the classic Singleton pattern, but noted that is has some problems, one of them being a single responsibility violation. For the second demo, we used a base class and fixed that violation, separating the concerns of the Singleton class and the Logger class. For the third demo, we'll build a metaclass. Now other modules in the course have used metaclasses as a way to build abstract base classes, but metaclasses can do more than that. Conceptually, a metaclass is a class's class. That means that an instance of a metaclass is a class itself, and as a class controls instantiation of an instance, so a metaclass can control the building of a class. We can use that here. In this version, Singleton is a metaclass. Note that it inherits from type, not object. Here, we have a call method that takes the place of the new method in the previous example. It maintains a dictionary of classes and instances as before, and in the modified Logger class, we now specify Singleton as its metaclass instead of inheriting from Singleton. The metaclass then is completely distinct and has but one responsibility, manage the instances of the classes it is responsible for. Our separation of concerns is complete. Switching back to the main program, if I run this version, we get the expected results. So this version works too.

Demo 4: The MonoState Mini-pattern

In case you thought we were done, there's yet another way called the MonoState pattern. Basically, MonoState gives in to the idea that there will be some global state, which the Singleton pattern has to do anyway. MonoState does it in a different way though, taking advantage of Python's dynamic nature. MonoState maintains a dictionary containing the single state for all instances. In its dunder new method, the dict object, where an instance's state is stored, is redirected to the single state dictionary, maintained at the class level, thus no matter how many new instances you create of a MonoState derived class, they all share the same state, and I've modified the Logger class to inherit from MonoState. So now, I'll switch back to the main program to run this variant. As you can see, it works just as well as the other ones.

Summary

Adding it all up, you can see the Singleton is a useful pattern for specific applications. It provides controlled access to a single instance by disallowing the creation of new ones, and this is also done lazily, the first time an instance is required. Though it is itself a global object, it reduces the overall size of the global namespace. It is subclassible for extended purposes, and you can have a variable number of instances, easily done using the base class or metaclass variants we looked at. It is more flexible than a static class since you can have multiple instances. Recall, in Python a static class is one with no instances. All attributes are class level attributes instead. The MonoState variant can be used when you do want multiple instances, but you also want that they share the same state, which can be a good thing depending upon the application. And though we didn't demo it, you can use a Python module in place of a Singleton. Remember though, use Singleton sparingly. It is considered by many to be an antipattern.

The Builder Pattern

Introduction to the Builder Pattern

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we'll be learning about the Builder pattern, a pattern that can help with building complex objects. As in other modules in this course, we'll apply the principles of object-oriented programming, known by their acronym, SOLID. If you need a refresher, you can review the introductory module. The Builder pattern is a Creational pattern. That means it's used to create objects. Builder helps by separating the construction of a complex object, say an object representing a custom computer from its representation, what the specifications of the finished computer actually look like. Builder does this by encapsulating the construction of the object. Thereby obeying the principle, encapsulate what varies, which is a corollary of the single responsibility principle, the S in SOLID. It also allows for a multi-step construction process. In fact, this is where Builder really shines. Building a custom computer requires several steps, even when you're just looking at off-the-shelf components. One key benefit of the Builder pattern is that implementations can vary. Today I might want to build a powerful desktop workstation, tomorrow I might need a budget box. Builder can easily accommodate both without changing the client interface. The client sees only the abstraction.

The Computer Builder: Too Many Parameters!

For a motivating example, let's implement the custom computer builder. Now a custom computer can have many components, and this sort of thing can lead to some bad design choices, as we'll see. For this demo, I'll be using the free Visual Studio Community edition, a lean cousin of the full Microsoft Visual Studio. This IDE is a Windows-only product, so if you do your development on Linux or OS X you'll probably want to check out some of the other IDEs mentioned in the introduction and used in other modules in this course. Well here we have the main program, in our first attempt at writing a custom computer builder. It imports one class, Computer, which looks like this. Note that the constructor uses lots of parameters, one for each component. This class also includes a display method to show the results of building the custom computer. So back in the main program, we can see that it instantiates the computer object, specifying all the required parameters. Long parameter lists like this can be a sign that there may be a better way to approach this sort of problem. Now let's run this program to see if it works. Opening up the Immediate Window, we can see the results of execution. The program works. Now, I'm still bothered by that long parameter list, and you may also encounter constructors like this in your own work. They can be hard to understand and maintain since it's not immediately obvious which parameters are mandatory and which are optional. Sometimes constructors seem to grow, like this one, as a project adds new features. The Builder pattern is good at handling just this sort of issue, but before we get there let's see another way to tackle this problem.

The Computer Builder: Exposing the Attributes

For the second attempt let's get rid of those constructor parameters. In fact, I won't need a custom constructor at all, so all we have in the Computer class is the display method. Instead of constructor parameters, I'm setting the attributes in the main program. This is actually one common technique to solving the parameter proliferation problem, though not a very good one. Running this version, we'll see that it produces the same results as before, so this one also works, and we've solved one problem, too many constructor parameters. But now we've got a new one, directly setting attributes in the client program. Not only is this error prone and maintenance unfriendly, it also goes directly against the encapsulate what varies principle. Let's try again.

The Computer Builder: Encapsulation

For the third attempt, I've encapsulated the computer components in a separate class called, MyComputer. It defines two methods, get_computer and build_computer. The second one instantiates a new computer object and encapsulates the setting of its attributes. In the main program then, all I have to do is instantiate my new class, build the computer, get the finished product, and display the results. Let's run this one. As we can see, it still works. So far we've fixed two problems. We've fixed the problem of too many parameters. Now there are none, that long parameter list is gone, and we've fixed the problem of setting the attributes in the client program. Now we've encapsulated those in the MyComputer class, but there's another problem, ordering. A computer cannot be assembled in any old way. There are definite steps to be followed in sequence. For example, you can't install the video card before you install the mainboard, and if you install the hard drive first it may obstruct other work you need to do. On to attempt number four.

The Computer Builder: Ordering

This time let's try to get the ordering correct. To help with that I've created a new class, MyComputerBuilder, which enhances the previous MyComputer class by adding the steps to build a computer in the correct order. The build_computer method does this by calling methods for each of the steps. Note that there's a separate step for the mainboard assembly, which is then installed into the case. The main program looks pretty much the same as before, so let's run it.

The results are the same as last time, so this method works too. Imagine though that I wanted the option of building another, cheaper computer. Well I suppose I could do something like this. I could find mycomputer_builder, make a copy of that, I could paste it back in to the same area, and I could give it a new name. We'll just call it, well let's rename it to cheapone, because that's what we're going to try to do. I'll open up the CheapOne class, rename it, and then start making modifications. So I could simply make cheaper components. For example, maybe I could say that I'll use the onboard graphics instead of a separate graphics card, and maybe I'll use a smaller hard drive, instead of 2 TB, well let's say for the cheap one, 500 GB is enough, and so on, up the way. When I get to the top though, I look at this and I say, wait a minute, I've got these two methods, a get_computer method and a build_computer method, they look suspiciously like the same ones in MyComputerBuilder, and in fact they are exactly the same. Copied code like this is often a real maintenance headache. If that code changes, you'll have to find all those copies and change them all. There must be a better way.

Applying the Builder Pattern

Though we've solved the problems of too many parameters, encapsulating the attributes, and defining the order, we're left with a duplicate code problem. The Builder pattern to the rescue. This is the structure of the Builder pattern in a UML diagram. We can use this to solve the custom computer problem. Top-right is the AbsBuilder. It defines at least one BuildPart method that must be implemented. Below it is a ConcreteBuilder. There can be many of these. They implement the AbsBuilder methods and GetResult method. Top-left is a Director class. It knows how to assemble the product and uses the ConcreteBuilder methods to do the work. It simply calls each builder part method in the correct sequence to get that work done. Finally, the Director calls the GetResult method, which returns the finished product. For the final demo in this module, let's implement the Builder pattern and use it to solve our problem of building a custom computer object. Let's start off by looking at the AbsBuilder metaclass here. I've gone ahead and implemented the get_computer and new_computer methods in the abstract base class since these would otherwise be repeated in each concrete builder in this example. For more complex objects, you should consider making these abstract as well, and implementing them in the concrete builder instead. Each step of building the computer is represented here by an abstract method. The concrete builders must implement each of these. Looking at the revised MyComputerBuilder class, it now inherits from AbsBuilder, and implements the required methods. That's it. The duplicate code is gone, but where? It's in the new Director class. The client, the main program in our example, will pass in the builder it wants to use for the computer to be built. The

Director then uses the methods it knows are in the AbsBuilder metaclass, and calls them in the right order to get the computer built. The Director class also implements a get_computer method to retrieve the finished computer object from the builder. There is actually another concrete builder here, the BudgetBoxBuilder. It looks just like the MyComputerBuilder class except for the components. We've encapsulated what varies and nothing else. The main program builds two computers using the two different builders and shows the results. Let's run it. This time we get two computers built, each with its own set of custom components. The Builder pattern has done its job.

Summary

To recap then, what are the problems we've hit and fixed on the way to the Builder pattern? First off, we had an issue with too many parameters. This can happen as a project matures and new features are added without refactoring the code. We fixed that one by exposing the attributes to the client, but this was actually worse than the original problem and would only make maintenance harder. To fix that we created a new class to encapsulate the attributes and their values, but then that just exposed the next problem. There was no order to the assembly. We enforced the assembly order by separating the attributes into methods, and that looked great until we wanted a different product, then we needed to copy a bunch of code. Finally, we applied the Builder pattern to separate the assembly operation from the product's components. So then, what does the Builder pattern do for us? It separates the how from the what. That is, the assembly of a product, or some object, is separated from the components that go into it. This approach encapsulates what varies, the components, while permitting different representations. In the custom computer example, we saw it was easy to create new concrete builders for different product configurations. The client, the main program in the example, creates a Director object. The Director uses a concrete builder and builds the product in the order required. The builder adds parts to the product, and when the Director is finished building the product, the client receives the finished product. The Builder pattern is a great one to have at hand when you hit problems similar to the ones solved in this module. Be sure to have it in your toolkit.

The Factory Pattern

Introduction to the Factory Pattern

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we'll be learning about the Factory pattern, a pattern that is useful when we want to create different objects, but use one API to rule them all. As in other modules in this course, we'll apply the principles of object-oriented programming known by their acronym, SOLID. If you need a refresher, you can review the introductory module. The Factory pattern is a Creational pattern. In fact, the name itself implies just that. The Factory pattern defines an interface for creating an object, but it lets subclasses define which object to create. It does this through a factory method that lets a class defer instantiation to those subclasses. The Factory pattern is also known as the Virtual Constructor pattern.

Demo 1: Building Car Objects

For a motivating example, consider a way to create an object representing some model of a car. Now, we want to be able to support several car models in our program, but we don't know which one we'll need until runtime. Let's try a brute force approach to solving this problem. Here's one approach to solving the car as an object problem. Here I support three car models, a Chevy Volt, a Ford Focus, and a Jeep Sahara. Oh yes, there's a null car too. We'll come back to that one in a moment. For example, the ChevyVolt class looks like this. It implements a start and stop method, both of which just print messages for testing. The other two real cars look very much the same. Back in main, after the imports, I have a getcar function that takes the name of the desired car as a parameter. Now, the name may come from a command line argument, input from a desktop program or a website, or perhaps be retrieved from a database. Take note of the long if, elif, else statement. When you see these in code or are tempted to write something like it, it's often a sign that you might want to consider another approach. The Factory pattern offers a solution for just this issue. Now depending on the desired car model, the program instantiates the matching class, and in case there is no match, I default to the NullCar class. This is an example of the Null pattern which basically says, return a dummy instance that still implements all the required methods. This then means that I don't have to test for null objects at runtime. The NullCar class looks like this. It accepts the car name as input to the constructor, but then just prints an error message when I try to start the car. So back in the main program to test things, I have a little loop that runs through the supported car models and one unsupported one. Let's run this to see if it works. In the output window you can see that the program works. Thinking about it though, if I wanted to add a different car model, I'd have to open up this code, modifying the imports, and the getcar function. That would break the open/closed principle. Also, I'm directly instantiating the car classes, which breaks the dependency inversion principle, since I'm depending on the implementation of those classes. Let's try again and let's try to get rid of the if, elif, else statement while we're at it.

Demo 2: The Simple Factory Pattern

The first pattern we'll look at is called the Simple Factory pattern. In UML it looks like this. Starting from the right-hand side, there is an AbsAuto base class. It stipulates that two methods must be implemented, the Start and Stop methods. At the bottom you can see three concrete classes. These are the car models we want to support. At the left, we now have an AutoFactory class. This class has the job of creating and returning an instance of the desired Auto class. Let's look at it in code. Here we are back in Visual Studio Code. We're looking at the SIMPLEFACTORY folder. Note that I have created a little Python package to hold the automobile classes. Using packages like this can help you organize your work. You know it's a package since the subdirectory defines a dunder init module, and here's the one we're using. It just imports the classes we want into the namespace. Now, looking at the abstract base class for the automobile classes, here are the two abstract methods that must be implemented by the concrete classes, Start and Stop. For example, the ChevyVolt class, it implements these two methods and prints the same messages we used in the first demo. The other automobile classes are similar. Note that the NullCar class defines a constructor, whereas the other ones don't, to which I pass the name of the car requested. The Factory pattern allows for different implementations of the concrete classes, and this is a simple example. Now the main program imports a new AutoFactory class, which looks like this. There is a fair bit going on here, so let's dig into it. Note first of all the imports. I import some objects from Python's introspection module, then import the autos package. Importing the autos package will execute those import statements in the dunder init module, adding them to the autofactory namespace. The class itself keeps the automobile names and classes in a dictionary, where the key is the name of the car model and the value is a reference to the class definition for that car. The init method then calls load_autos, which builds the dictionary. Load_autos uses the introspection function getmembers to find the classes imported into the package in the init module, but only concrete classes. Then it looks for those classes that are subclasses of the abstract auto base class and adds those and only those to the dictionary. The create_instance method looks for the carname in the autos dictionary. If found, it returns a new instance of that class. If not found, it returns an instance of the NullCar class, passing in the car name. In the main module then, there's not so much to do. It simply imports and instantiates the AutoFactory class, then loops through the test examples, calling create_instance for each one and testing the Start/Stop methods. Well let's run this one. So we go to the DEBUG window, we click the Start button, and the program is ready to run so let's run it, and in the DEBUG CONSOLE, if I expand it, you can see that this version, using the Simple Factory pattern, works too. So, what are the problems we've solved so far? Well for one thing we eliminated the open/closed violation. Now it's simple to add new automobiles, just create the new classes and drop them into the autos package, then add the new imports to the init module. We also eliminated dependency on the implementation of the automobile classes. The main program now only needs to know that those classes implement the abstract methods in the abstract base class. We've also separated the concerns of the main program and the auto factory loader. One thing to note though, we're limited to one auto factory. What if we need more factories for flexibility? In that case, the classic Factory pattern is what we need.

Demo 3: The Full Factory Pattern

The full Factory pattern is the one described in the seminal work on design patterns for software. The so called Gang of Four, because there are four co-authors, describes a structure like this. At the top-left there is an AbsProduct base class. This is what we want the factory to produce. For our working example, that's a car. Bottom-left is the Concrete Product that will be produced. Just like the Simple Factory pattern we used before, there will be one product type, or car model in the example. Top-right is an AbsFactory base class. It declares one method to be implemented the create_product method. Bottom-right is the Concrete Factory. The implementation of the create_product method returns the finished product. Let's see how this can be implemented. In the Factory folder I've set up two packages this time. An autos package, which is somewhat the same as before, and a new package called factories. I've modified the abstract auto base class slightly, adding a name property, which is used by those factories. In the new factories package I have an AbsFactory base class, which defines just one method, create_auto. Each factory must implement create_auto like the ChevyFactory here. Note that it also sets that new name property after instantiation, but also note that this is not a part of a Factory pattern. I show it here as an example of one implementation. There can be many others. In this case the name property is subsequently used by the ChevyVolt concrete class to format the messages that get printed. Well, the main program looks quite simple now, though there is a new wrinkle, that loader module with its load_factory function, so let's look at that for a minute. This module uses dynamic imports. Given a factory name, it tries to import it from the factory's package. If it fails, it imports the null_factory instead. Then it looks for a class to return. As with the Simple Factory example, it returns the first class that is not abstract and is also a subclass of AbsFactory. To avoid confusion I only allow one factory class per module. The load_factory function then eliminates the need to import all the factories in the dunder init module, which is empty as you can see. To add new factories you only need to add their modules to the factories package. Now you may be wondering if I could have used the same technique in the Simple Factory example. Well yes, I could have. It is good to look at alternatives though. So let's go back to the main program and run it. Well we start up the debugger, and then we will start up the program, and looking at the output, we can see that it works fine. The Full Factory pattern then adds an abstract factory base class. And this means that many factories and many factory types can be implemented, and those implementations can vary. A complex factory might use other patterns to help, the Builder pattern, also covered in this course, is a natural example. Finally, though not part of the pattern per se, the factory loader encapsulates the logic of finding and instantiating the factories. It's really an implementation detail, but you might find it useful in your own work.

Summary

Adding it all up then, what do we get from the Factory pattern? Well, it encapsulates object instantiation. We no longer have to instantiate classes directly, instead we call a factory method to do it for us. This supports the dependency inversion principle. Client programs are no longer dependent on the implementations of the classes they use, in fact, they don't even need to know the names of those classes, instead clients depend upon an abstraction. They know that all objects returned from the factory must adhere to that abstraction, supported in Python by an abstract base class. We looked at two variations of the pattern. The Simple version, which supports just one factory, and is often all you will really need, and the full Classic Factory pattern, which also abstracts the factory and allows for multiple factories. With this one, you first get an instance of the factory, then ask it to produce the object you need. It is the more flexible of the two. There's actually a third variation, the Abstract Factory pattern, which we'll look at in a subsequent module.

The Abstract Factory Pattern

Introduction to the Abstract Factory Pattern

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we'll be learning about the Abstract Factory pattern, a pattern that is useful for creating families of related or dependent objects without specifying their concrete classes. As in other modules in this course, we'll apply the principles of object-oriented programming known by their acronym, SOLID. If you need a refresher, you can review the introductory module. The Abstract Factory pattern is a Creational pattern. It's a close cousin of the Factory pattern also covered in this course. Whereas Factory creates one product class, Abstract Factory can produce a family of classes and it enforces dependencies between the concrete classes. As with the Factory pattern, Abstract Factory defers object creation to concrete subclasses. The Abstract Factory pattern is also sometimes called the Kit pattern.

Demo 1: Car Manufacturers

For a motivating example, consider a collection of car factories. Each factory makes cars for just one manufacturer, but each factory can also make different models, say an economy model, a sport model, and perhaps a luxury model, and we need to support multiple car manufacturers, each one having those same three model types, so how could we solve that? For this module we'll be using the Atom IDE. This is a new IDE from the folks at GitHub and the Atom community. You can find downloads and information at atom. io. When you start up Atom for the first time, it looks pretty empty as you can see, except for those usage hints that flash on the screen. You know I kind of like the way that works. There's no pop-up window to dismiss, as you find frequently in other IDEs. This is just less muss and fuss. Atom is built as a bare bones framework that is easily extendable with independently written packages. All language support, debugging, code highlighting, and completion are implemented through packages, and this applies to Python support as well. Let me open the settings for a moment. That's found in the File menu, Settings. There are too many settings to discuss them all individually. Looking at the packages though, you can see that Atom reports that there are 82 packages installed. Now I only installed four myself, those are the first four you see here. All the rest are Core Packages, all the 78, including language support, like Python. The next thing we really ought to do is open a folder, so let's do that. I'm going to get a brand new window which I can simply do here, File, New Window. Now we have a new empty workspace with one untitled file in it, I can close that. Now to open a folder I simply go to File, Open Folder, find the one I want, and open it up. Now you can see that the left pane contains a folder tree. Let me open up the example Python file I have in here, and you can see the code-color highlighting. It's also easy to manipulate this space, for example, with the mouse if I hold the Ctrl and use the mouse wheel, I can make the text smaller or larger to improve the visibility. Hitting F5, thanks to one of those packages I installed, will run this program. The execution takes place in a separate window as you see, and you can see that the sample program works just fine. Here is one approach to solving the car as an object problem. In this example I support two manufacturers, GM and Ford, and three cars for each one to represent the various models. I put the factories in a Python package here, and put the cars for each manufacturer right in the factory modules, like the Ford module here. Each car type has a class with two methods, Start and Stop, that just print messages. The main program then imports all these cars and then randomly chooses which manufacturer and car type to produce, using the randint function from Python's standard library. In a real implementation these would be provided to the client program, perhaps through some configuration settings or user input. The main work of the program then is to instantiate and test the desired car model. That happens in this long if, elif, else statement, and these are even nested to accommodate the two manufacturers. Let's run this. If I hit F5, that handy package I loaded and installed will do the work for us. You can see that it produces the desired output, and if I run it again, thanks to the random function, I'll get a different car. To be honest though, I really hate this program. All those imports at the top, and that long if, elif, else structure, it's just asking for trouble. It's easy to see that this breaks the open/closed principle too. After all, what would I have to do to add Honda to the mix? Why, open up the main program, import some more cars, modify this long if, elif, else statement? Just a mess. The Abstract Factory pattern will help us resolve these issues.

Demo 2: The Abstract Factory Pattern

The Abstract Factory base pattern, not to be confused with an Abstract Factory base class, which is a Python construct used in both the Factory and Abstract Factory pattern implementations, looks like this in UML. Starting from the top, there's an AbstractFactory interface, which is implemented as an abstract base class in Python, an interface in Java and C# and so on. In this example it stipulates that two methods must be implemented for two different products. In the example we looked at, there are actually three products, one for the economy, one for the sport, and one for the luxury car models. The diagram shows two factories, one for each product type or car manufacturer in the example. Each factory makes different products, and these products implement an abstract product base class. Finally, we have two concrete product classes for each factory, one set for the first factory, which makes the A1 and B1 products, and one for the second, which makes the A2 and B2 products. Naturally, this can be extended to add more product types and more factories. Here we are back in Atom looking at the AbstractFactory folder. This folder holds the main program and two packages, a factories package and an autos package. The factories package holds an abstract base class, which looks like this. Three abstract static methods are defined, one for each required car type. By the way, in case you're wondering, I made these methods static since the class doesn't hold state. That's not required by the pattern, just a simplification I made for this example. There are then two concrete factories, one each for Ford and GM. The Ford factory looks like this. It imports the abstract factory base class and the car models it needs. The class then implements the required methods, instantiating the desired car models for each one. The GM Factory is quite similar. To organize the car models I have an autos package. It holds the abstract auto module and two sub packages, one for the GM models and one for Ford. The abstract auto base class looks like this, it just defines two required methods. For the concrete cars, well let's open Ford and look at the Fiesta. So here's the concrete class for the Ford Fiesta. It imports the abstract base class, then implements the required methods, which just print messages. The other concrete car classes are quite similar. The main program then, which I built to test the factories, imports the two factories and tests them in a little loop. Running this program, which I can do by hitting F5, you can see that the right cars are chosen for Ford and GM, and for the economy, sport, and luxury models.

Summary

This brief road trip with the Abstract Factory pattern shows that it shares some concepts with the Factory pattern. For one thing, it encapsulates object instantiation, which supports dependency inversion so that clients can write to an abstraction and not an implementation. So then which one should you use, Factory or Abstract Factory? Well, remember that Factory is great when you don't know which concrete classes you'll need, maybe not until run time, and Abstract Factory is superior when you have families of similar objects. Like so many other questions of this type, the only definitive answer is, it depends.

The Null Pattern

Introduction to the Null Pattern

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we will be learning about the Null pattern. We've encountered this pattern a few times before, already in this course. In spite of its name, there really are a few things we need to talk about. Some call this a mini pattern, but that's because it is usually easy to implement with a few lines of code, not because it's unimportant. As is other modules in this course, we will apply the principles of object-oriented programming known by their acronym, SOLID. If you need a refresher, you can review the introductory module.

Demo

Note that we're using Visual Studio Code for this demo. If you like what you see, you can download it at code. visualstudio. com. Now to demonstrate the problem we are talking about, consider this program, which uses the Factory pattern. In the main program we import and use an object factory to get an instance of the desired object. The program then tests to see that it got a valid object back, and if so, executes the do_something method. If not, it just prints an error message. If you've been developing programs for a while in any language, you've probably seen code similar to this, try to get some object, see if you got it, then use it. The object factory itself looks like this, it only has one static method, create_object, which creates an object of the requested type or returns None if it can't find a matching class. My class looks like this. It implements an abstract base class called AbsClass, by defining the one required method. Here's the abstract base class, and you can see that abstract method. Now let's go back to the main program and let's try to run it to see if it actually does, well, do something. So I go to debug and start up the debugger, and it's ready so I'll start the program, and down in the bottom in the DEBUG CONSOLE you can see the message, Doing something, so it actually did work. It gave us what we needed. Now if I change the class name it's looking for, we'll just call it myotherclass, and try again, well this time it ought to trigger the else clause, and it does. I get the message, Not doing anything. Well, I don't know about you, but I get tired of writing none tests all over the place. I'll even confess that I've forgotten to do that when I needed to. The Null pattern will allow us to stop writing those tests. Let's see how easy that is to put together. So if I go to the object factory, I'll start off by importing an additional class, the nullclass, and instead of returning None, I'll return an instance of this nullclass, and what does that nullclass look like? Well, here it is right here. It implements the abstract base class as required, but the do_something method just does something default, in this case I just print a different kind of message. So if I save everything, and now I go back to the main program, it means I don't actually need all of this logic. I'm just going to need this and nothing else, so let's just take that here, and let's comment these few lines. Save that program. Let's go back to the original class, myclass, and run this one. So I go to debug, start out the debugger, run the program, and I get the message I was looking for, Doing something. Okay, let's test the other one, if I say myotherclass. Should this work? I sure hope so. Start it up again. Indeed it does work. All of that other logic was commented and yet I get the message back that I was expecting. Why? Because it's coming from the nullclass. A perfect example of how to use the Null pattern.

Summary

This short module looked at the Null pattern. It provides a default object that implements all required methods and properties. Note that this object need do nothing at all except mimic a real object, providing minimal implementations of required methods in properties. This helps clients by removing the need to test for None. Clients then can just use the object returned. And note, this is not limited to classes. This can be quite useful for functions, iterators, and generators as well. Other patterns like the Strategy pattern, also covered in this course, can use the Null pattern. So the next time you see a test for None, after a function or method call, consider implementing the Null pattern instead, so that you can remove that test for None.

Course Summary

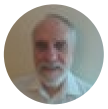
Design Patterns Covered

Hello again. Welcome back to the course, Design Patterns with Python. My name is Gerald Britton, and in this module we're going to take a look back at what we've covered, and a look forward to other design patterns we should talk about. The design patterns discussed in the classic 1995 volume of the same name by Gamma, Helm, Johnson and Vlissides, number 23. A quick trip to Wikipedia shows descriptions of 53 design patterns as of this writing, including 16 for concurrent processing, which was little used in the everyday business computing when the Gang of Four did their work, but it's become more and more important as CPU clock speeds have plateaued and multi-core processors are used everywhere, even in your mobile phone, something surely unforeseen in 1995. In this course, we've looked at eight design patterns that are in frequent use today. First, we considered the Strategy pattern, a great way to take diverse algorithms having the same input and output parameters, encapsulating them, and then exposing the standard interface to client programs. Strategy encapsulates what varies, thus obeying the single responsibility as well as the open/closed principles. Second, the Observer pattern was a subject. It defines a one-to-many dependency between objects so that when one object changes state, all its dependent objects are notified and updated automatically. This separates the concerns of the observed object, which is called the subject, and the observer itself. We noted that this pattern also comprises two thirds of the Model View Controller pattern, popular for programming applications on the web, on the desktop, and on your mobile phone. Third up was the Command pattern. In some ways this one is a generalization of the Strategy pattern, since the request handled by Command need not have the same signature at all. This pattern is also handy for supporting undo requests, the kind of thing an IDE does routinely. Next, we looked at the Singleton pattern. A pattern used to ensure that there can be only one instance of an object. This pattern is often found in logging systems or in programs controlling a piece of hardware, a graphical display for instance. We saw a number of ways to implement that in Python including the related Monostate pattern, but noted that Singleton is often considered an antipattern. It is effectively an object-oriented way to handle global state, which should mostly be avoided whenever possible. The fifth pattern we looked at was the Builder pattern. Builder separates construction of a complex object from its representation so that the same construction process can create many different representations. We followed the Builder pattern with the Factory pattern. Factory defines an interface for creating an object, but lets subclasses decide which class to instantiate. We also noted that those classes could use the Builder pattern to assist if the objects aren't complex enough. Close on the heels of the Factory pattern we met the Abstract Factory pattern, which defines an interface for creating families of related or dependent objects. In some ways, Abstract Factory is a generalization of Factory. The last pattern we looked at, the Null pattern, is sometimes called the Mini pattern, more because of its size than its importance. Null ensures that when a function or method is to return an object of some type, it always does just that. An empty object, or a None object in Python, is never returned, freeing client programs for testing return results, an often missed yet necessary part of much programming today regardless of the language used.

Summary of Summaries

Well I suppose this is a summary of summaries. There are many design patterns for programming, all of which can be exploited in Python programs. These patterns solve real world programming problems. In fact, they were all discovered in just that context. The right design pattern can make your work easier to write, easier to read, and easier to maintain, and note, the last two are more important than the first one. Design patterns are independent of the application area you are working in today or may work in tomorrow. Master them and use them.

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	Intermediate
Rating	★★★★★ (103)
My rating	★★★★★
Duration	1h 57m
Released	13 Oct 2016

Share course