STUDENT MANUAL

Advanced Programming Techniques with Python®

Advanced Programming Techniques with Python®

Advanced Programming Techniques with Python®

Part Number: 094022
Course Edition: 1.1

Acknowledgements

PROJECT TEAM

Author	*Media Designer*	*Content Editor*
Bob Carver Peter Lammers	Brian Sullivan	Michelle Farney

Logical Operations wishes to thank the Logical Operations Instructor Community for their instructional and technical expertise during the creation of this course.

Notices

DISCLAIMER

TRADEMARK NOTICES

Advanced Programming Techniques with Python®

About This Course

Python® continues to be a popular programming language, perhaps owing to its easy learning curve, small code footprint, and versatility for business, web, and scientific uses. Python is useful for developing custom software tools, applications, web services, and cloud applications. In this course, you'll build upon your basic Python skills, learning more advanced topics such as object-oriented programming patterns, development of graphical user interfaces, data management, creating web service-connected apps, performing data science tasks, unit testing, and creating and installing packages and executable applications.

Course Description

Target Student

This course is designed for existing Python programmers who have at least one year of Python experience and who want to expand their programming proficiency in Python 3.

Course Prerequisites

To ensure your success in this course, you should have experience with object-oriented programming and Python 2.x or 3.x. You can obtain this level of skills and knowledge by taking the following Logical Operations courses:

- *Introduction to Programming with Python®*

Course Objectives

In this course, you will expand your Python proficiencies. You will:

- Select an object-oriented programming approach for Python applications.
- Create object-oriented Python applications.
- Create a desktop application.
- Create data-driven applications.
- Create and secure web service-connected applications.
- Program Python for data science.
- Implement unit testing and exception handling.
- Package an application for distribution.

The CHOICE Home Screen

Logon and access information for your CHOICE environment will be provided with your class experience. The CHOICE platform is your entry point to the CHOICE learning experience, of which this course manual is only one part.

On the CHOICE Home screen, you can access the CHOICE Course screens for your specific courses. Visit the CHOICE Course screen both during and after class to make use of the world of support and instructional resources that make up the CHOICE experience.

Each CHOICE Course screen will give you access to the following resources:

- **Classroom**: A link to your training provider's classroom environment.
- **eBook**: An interactive electronic version of the printed book for your course.
- **Files**: Any course files available to download.
- **Checklists**: Step-by-step procedures and general guidelines you can use as a reference during and after class.
- **Spotlights**: Brief animated videos that enhance and extend the classroom learning experience.
- **Assessment**: A course assessment for your self-assessment of the course content.
- Social media resources that enable you to collaborate with others in the learning community using professional communications sites such as LinkedIn or microblogging tools such as Twitter.

Depending on the nature of your course and the components chosen by your learning provider, the CHOICE Course screen may also include access to elements such as:

- LogicalLABS, a virtual technical environment for your course.
- Various partner resources related to the courseware.
- Related certifications or credentials.
- A link to your training provider's website.
- Notices from the CHOICE administrator.
- Newsletters and other communications from your learning provider.
- Mentoring services.

Visit your CHOICE Home screen often to connect, communicate, and extend your learning experience!

How to Use This Book

As You Learn

This book is divided into lessons and topics, covering a subject or a set of related subjects. In most cases, lessons are arranged in order of increasing proficiency.

The results-oriented topics include relevant and supporting information you need to master the content. Each topic has various types of activities designed to enable you to solidify your understanding of the informational material presented in the course. Information is provided for reference and reflection to facilitate understanding and practice.

Data files for various activities as well as other supporting files for the course are available by download from the CHOICE Course screen. In addition to sample data for the course exercises, the course files may contain media components to enhance your learning and additional reference materials for use both during and after the course.

Checklists of procedures and guidelines can be used during class and as after-class references when you're back on the job and need to refresh your understanding.

At the back of the book, you will find a glossary of the definitions of the terms and concepts used throughout the course. You will also find an index to assist in locating information within the instructional components of the book. In many electronic versions of the book, you can click links on key words in the content to move to the associated glossary definition, and on page references in the index to move to that term in the content. To return to the previous location in the document after clicking a link, use the appropriate functionality in your PDF viewing software.

As You Review

Any method of instruction is only as effective as the time and effort you, the student, are willing to invest in it. In addition, some of the information that you learn in class may not be important to you immediately, but it may become important later. For this reason, we encourage you to spend some time reviewing the content of the course after your time in the classroom.

As a Reference

The organization and layout of this book make it an easy-to-use resource for future reference. Taking advantage of the glossary, index, and table of contents, you can use this book as a first source of definitions, background information, and summaries.

Course Icons

Watch throughout the material for the following visual cues.

Icon	*Description*
	A **Note** provides additional information, guidance, or hints about a topic or task.
	A **Caution** note makes you aware of places where you need to be particularly careful with your actions, settings, or decisions so that you can be sure to get the desired results of an activity or task.
	**Spotlight** notes show you where an associated Spotlight is particularly relevant to the content. Access Spotlights from your CHOICE Course screen.
	**Checklists** provide job aids you can use after class as a reference to perform skills back on the job. Access checklists from your CHOICE Course screen.
	**Social** notes remind you to check your CHOICE Course screen for opportunities to interact with the CHOICE community using social media.

1 Selecting an Object-Oriented Programming Approach for Python Applications

Lesson Time: 1 hour

Lesson Introduction

There are many ways you can approach writing an application to perform the processing tasks that you need. The object-oriented programming approach offers many benefits. In this lesson, you will implement fundamentals for implementing an object-oriented approach to writing an application in Python®, and see what benefits can be leveraged from that approach.

Lesson Objectives

In this lesson, you will:

- Describe object-oriented design concepts.
- Describe the benefits of object-oriented programming.

TOPIC A

Implement Object-Oriented Design

To successfully write an application using an object-oriented design approach, you must follow the principles of object-oriented design. In this topic, you will implement object-oriented design for your Python application project.

Note: To learn more about topics of interest related to the course content, view the Spotlight presentations from the **Spotlight** tile on the CHOICE course screen.

Programming Styles

A programming style is a set of guidelines for writing code for an application. It's not the language or the syntax of a language, but a way to approach the organization and formatting of code to make code easier to understand and collaborate on, and to reduce errors. There are four major programming styles supported by Python to some extent:

- ***Functional***. In a functional programming style, coders create and apply functions, essentially treating statements like math equations. Smaller functions are treated as modular and combined to get the desired processing from the code. Functional programming is well suited for parallel processing and is often used in academia and data science.
- ***Imperative***. Imperative programming statements change the state of the application by focusing on "how" data should be processed. Scientists like imperative programming because they can write code that clearly reflects their processes. This type of programming is often used to manipulate data and data structures.
- ***Procedural***. Procedural style is a step-by-step approach to coding. You write code to perform one task, then write additional code to perform additional tasks. This is the style most coders start with because it is the simplest and works well for iterative tasks, and tasks involving sequencing, selection, and modularization.
- ***Object-Oriented***. Object-oriented programming (OOP) organizes data and code into objects that can be manipulated by code. This allows developers to focus on objects they want to work with, and what they want to do with those objects rather than application logic. OOP creates highly reusable code that is suitable for use in large, complex applications, and that can be worked on by a team of developers. OOP code bases also tend to scale well.

Note: Python does not fully implement OOP because it doesn't support some OOP features, but this programming style can still be used to a large degree.

Note: The remainder of the course and the sample applications use the OOP programming style.

Principles of Object-Oriented Design

Object-oriented design has four primary principles which we will discuss in the coming course:

- ***Abstraction***: The goal is to focus on the essential features and capabilities of an object relative to the context in which it is used.
- ***Encapsulation***: This is the process of obscuring the complexity of an object, while providing user interfaces to access the data and capabilities of the object.
- ***Modularity***: The act of breaking down problems and tasks into modules to reduce the complexity of the problem, and the solution.

- ***Hierarchy***: Allows you to rank or order abstractions to create subsystems made up of different components of the system.

Objects

In coding, an object can perform specific actions or have actions performed on it. The formal definition of an ***object*** in an OOP application is a collection of data and associated behaviors. You can think of an OOP program as a collection of objects that interact with each other through their data and behaviors.

One of the key benefits of OOP is the encapsulation of objects. Defining objects hides the complexity inherent in the object, making it easier to be used by exposing the data and behaviors. A good analogy is a car. A car can be thought of as a type of object. Most people don't know how to build a car, and many don't know how to repair a car. Fortunately, you don't need to know that to use a car. By creating an object that has a well-defined and well-understood set of behaviors, such as a steering wheel, gas pedal, and brake pedal, it can be used without understanding the internal complexity of the object. By abstracting the complexity of objects, developers don't need to understand the internal complexity of the object, they can just use the functionality. This allows objects to be highly reusable across an application, and allows members of the developer team to build complex core logic on top of the abstracted objects.

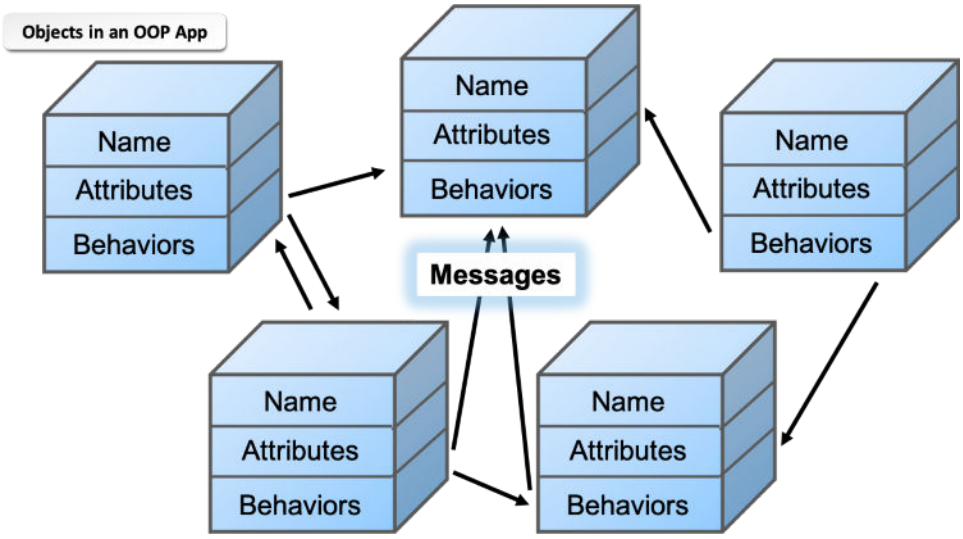


Figure 1-1: Objects in an OOP App.

Classes

When using the OOP style, it's helpful to start thinking about components of your code in terms of objects. That's the goal, to think of the code as objects that can have things done to them, or can do things to other objects. You may find that your application needs many different objects that have the same attributes and behaviors. In that case, you define a ***class*** to describe objects of that type. Classes are the data and functions you use in your applications, grouped together into a single entity.

A class can be thought of as a template for the objects that belong to that class. It describes the data fields and operations that belong to the objects of that class. For example, if you're writing a program that uses cars, you can create a car class that contains information on the car's make, model, price, and how many doors it has. If you're writing an application that tracks people, you might create a person class that contains information on each person's name, age, height, weight, and gender. An object-oriented programming class defines an object using members.

Class Definition

Classes define:

- ***Attributes***—Denote features of an object of the class (such as the weight, engine, and number of doors of a car). A class can contain many attributes, each describing a different feature of the car.
- ***Behaviors***—The functionality objects of a given class process. The tasks they can perform and how they respond to messages. An object's behavior is defined by its methods.
- ***Methods***—Similar to Python functions, perform operations using the attributes of the class. A class method should be contained within the class and only use the attributes from that class.

You define the attributes and methods inside a class definition. The class definition uses the `class` keyword, along with the class name. While not required, it's become somewhat standard in Python for class names to begin with an uppercase letter. Follow the class name with a colon to indicate more code is required to define the class:

```
>>> class Car:
...
>>>
```

After defining the class, you can use it anywhere in your program to denote that object. Each use of the class is called an instance. Each class instance is assigned to a variable in the program:

```
>>> car1 = Car()
>>>
```

The `car1` variable is an instance of the `Car` class.

If you're creating an app to track orders and manage inventory for the products in a retail store, you might create classes to represent:

- **Products**: These are anything you might sell in the store. This item will have data associated with it such as a name, product category, weight, quantity available, and price.
- **The shopping cart**: This represents what a customer may buy through their shopping and may contain many products, and will have its own price created from the products and quantities in the cart.
- **Order**: This represents what a customer has purchased and may include many products, a sum of the price, as well as other information such as a purchase date and shipping option.

Attributes

Attributes are variables that hold data that applies to the class. There are two types of attributes you can define in Python:

- ***Class attributes***—Apply to all instances of the class. For example, the car class may only apply to vehicles with two axles and four wheels.
- ***Instance attributes***—Apply only to a specific instance of the class. For example, cars have different makes and some attributes may only apply to specific makes of cars.

You define class attributes as part of the class definition:

```
 >>> class Car:
...    axles = 2
...    wheels = 4
...
```

Instance attributes only apply to a specific instance of the class. Python allows you to create instance attributes "on the fly" after the class instance has been created:

```
>>> car1 = Car()
>>> car1.make = "Chevy"
>>>
```

The `make` attribute shown here only applies to the `car1` instance of the `Car` class.

Behaviors and Methods

In addition to attributes, classes have behaviors. Behaviors are the actions that an object can take, and methods are the way those actions are performed. Methods determine the functionality of the class, including how its data and attributes are handled, and what actions it can take.

They may be subroutines or functions, and may be declared as public, protected, or private. As with attributes, there are two types of methods that you can define in Python:

- ***Class methods*** which are independent of any instance data, attributes, and methods. They are bound to the class, not any one specific object of the class, and use parameters to pass data into the method. They can also modify the state of the class, which will apply across all objects of the class.
- ***Instance methods*** which access the unique attributes available in a specific instance of a class.

Public Interface

Since one of the benefits of OOP is to hide the complexity of code while making useful data and capabilities available, each object has a public interface. The public interface consists of the attributes and methods that are accessible via code. Code can access the public interface for an object to retrieve attributes or call on the object's methods to execute tasks without knowing the inner workings of the object.

For example, objects that are members of the `Car()` class might allow you to access standard driving functions through the public interface such as acceleration and braking, while the complexity of how the engine, drive train, and brakes work to accelerate and decelerate remains hidden.

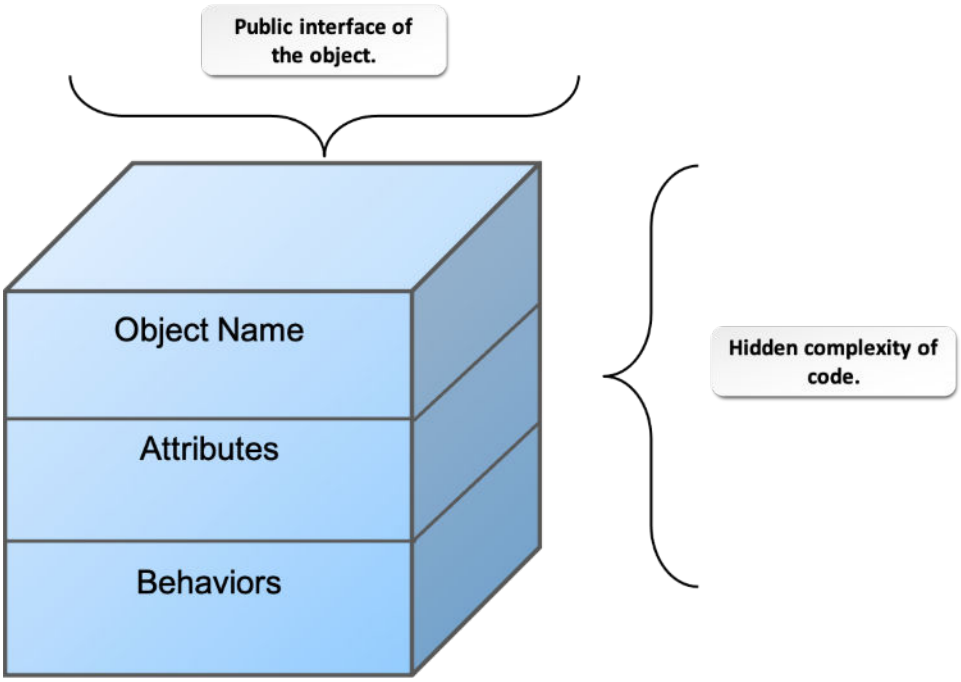


Figure 1-2: Public interface.

Guidelines for Implementing Object-Oriented Design

	Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines to help when implementing object-oriented design for Python programs.

Implement Object-Oriented Design

When implementing object-oriented design for Python programs:

- Use classes to combine data and functions that are related.
- Define the class using the `class` statement.
- Create an instance of the class by using the format `variable = Class(args)`.
- Reference any class properties from the instance using the format `variable.property`.
- Reference any class methods from the instance using the format `variable.method()`.
- Use class attributes for sharing values between instances.
- Use instance attributes for creating attributes unique to each instance.
- Define additional methods as needed for manipulating class properties.

ACTIVITY 1-1
Designing OOP Project Requirements

Scenario
You are a developer at Woodworkers Wheelhouse. Woodworkers Wheelhouse sells hardwood lumber, tools, and other supplies for woodworkers and other hobbyists. They also have a delivery service for lumber materials and want an application that allows local customers to place orders that will be delivered. You will start with a simple inventory solution to track products. For this inventory app, you want to define some of the different OOP objects you will need.

1. What classes would you define for inventory items?

2. What attributes might you need for the wood products?

3. What methods might you need?

TOPIC B

Leverage the Benefits of Object-Oriented Programming

Choosing an object-oriented approach for your application design provides many benefits you and your team can leverage to create not only a functional application, but a code base that is easy to maintain, update, and extend. In this topic, you will leverage the benefits of object-oriented design.

Composition

Composition is one of the basic principles of object-oriented programming design. Composition is the act of building, or composing, a new object from other objects, creating a ***class composite***. A class composite can contain one or more objects from another class component.

You know that objects are used to conceal complexity, and composite objects do the same. If you create a video game where you construct, upgrade, and race cars, a car may be an object built from other objects such as an engine, wheels, axles, and so forth. This can provide you additional programmatic functionality as drivers need access to some attributes and behaviors to drive the car, whereas the mechanic needs access to another set of attributes and behaviors to repair and upgrade the car.

Benefits of Modularity in OOP

Modularity in coding in general has many benefits; it tends to improve design and make unit testing easier because the nature of making code modular requires you to decompose or break down the problems to solve, and the tasks the app must complete. This, in turn, facilitates creation of an efficient and elegant application design because it forces you to lay out the moving parts of the app by defining what data must be accessed, what input must be captured, what computations to make, how results are to be output, and so forth.

In object-oriented programming, this naturally leads to the next step of defining classes and objects to address the needs of each discrete task that must be completed. This modularity pays benefits, as having separate objects and classes allows coding work to be split amongst a large team of developers. OOP itself does particularly well in this regard because encapsulation allows developers to hide object complexity and share only the interface component other developers will need to access the attribute and methods of a class.

This will make application updates, extensions, maintenance, and problem solving easier. If you need to add new reporting features, you know that it's the output class and objects that must be modified. If there's a problem with the app where data is not captured correctly, you know that you will need to look at the user input class of objects.

Superclasses and Inheritance

Another key concept in OOP is inheritance. When creating classes, you should try to keep the class definition as generic as possible so that it can fit as many application environments as possible. However, often you need to build onto a class definition to fit a specific application environment.

Python allows you to do that with ***inheritance***. Inheritance allows you to create a ***subclass*** (also called a derived class, or subtype) from a base class (also called the ***superclass***). A subclass contains (inherits) all of the original attributes and methods of the superclass, plus adds its own attributes and methods specific to the subclass, and is used to extend the functionality of the base class.

To create a subclass, you must reference the base class that it inherits when you define it:

```
>>> class Sedan(Car):
```

When you create the subclass, Python doesn't automatically call the constructor for the base class, so you must reference it in the constructor for the subclass:

```
>>>class Car:
...    def __init__(self, make, model):
...       self.make = make
...       self.model = model
...
>>>class Sedan(Car):
...    def __init__(self, make, model, doors, seats):
...       super(Sedan, self).__init__(make, model)
...       self.doors = doors
...       self.seats = seats
...
>>>
```

The constructor for the `Sedan` subclass uses the `super` method to inherit the attributes and methods of the superclass. This calls the `__init__` method for the `Car` base class, passing the constructor values to the base class.

Polymorphism

The subclass can define its own methods, but it can also define the same methods as the base class. This is called ***polymorphism***. If the same method is defined in both the base class and the subclass, objects created using the subclass use the polymorphed method from the subclass, while objects created using the base class use the methods from the base class.

In polymorphism, the ability for a subclass to define the same methods as the base class, and have those methods be used in place of those defined in the base class, is sometimes called overloading methods. This makes OOP very flexible and allows developers to reuse methods for subclasses rather than defining new ones to perform similar tasks, which simplifies code. It also makes it very easy to make a small change to a method in the subclass to get the desired result rather than writing an entirely new class, making code much easier to reuse.

Define any overload methods that need to supersede the superclass methods. In the following example, the display method is overloaded to display the key elements that make up the sedan subclass of the `Car()` superclass when the display method is used:

```
...    def display(self):
...       print('{0} {1} is a sedan with {2} doors and {3}
seats'.format(self.make, self.model, self.doors, self.seats))
>>>
```

Subclass Object Instantiation

Use the subclass name to instantiate new objects instead of the superclass name. Use the superclass name to instantiate any objects that don't use the subclass, but instead use the superclass. Remember that overloaded methods will behave differently for the subclass and superclass instances:

```
>>> sedan1 = Sedan("Ford", "Focus", 10000.00, 5, 4, 5)
>>> car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> sedan1.doors
4
>>> sedan1.display()
Ford Focus is a sedan with 4 doors and 5 seats
>>>car1.display()
Chevy Volt
>>>
```

Guidelines for Leveraging the Benefits of Object-Oriented Programming

Use the following guidelines for leveraging the benefits of object-oriented programming.

Leverage the Benefits of Object-Oriented Programming

To leverage the benefits of object-oriented programming:

- Consider making your code as modular as possible, as it tends to improve design and make unit testing easier.
- Keep object classes as generic as possible. Use inheritance to create customized objects from parent objects.
- Create classes and subclasses in modules so that you can easily use them in your Python programs.

ACTIVITY 1-2
Justifying OOP Use for a Coding Project

Scenario
You want to use OOP for this Woodworkers Wheelhouse project, but you will need to explain your rationale during an upcoming team meeting. You need to think of the reasons you feel OOP will benefit this project.

---

1. If you had to troubleshoot an issue with the Woodworkers Wheelhouse project, how might OOP make that effort easier?

2. The timeline for the Woodworkers Wheelhouse project does not allow for much extra time when developing. How might OOP save time with coding?

3. You will have a small team to help develop the Woodworkers Wheelhouse project. How might OOP make the development for the project easier?

Summary

In this lesson, you learned how to implement object-oriented design in your Python application, including how to define classes, objects, attributes, and methods. You also leveraged the benefits of object-oriented design including the use of superclasses, subclasses, and polymorphism.

How do you plan to use Python in your environment?

How do you plan to use OOP in your environment?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

2 Creating Object-Oriented Python Applications

Lesson Time: 3 hours

Lesson Introduction

Having decided on an object-oriented approach to build your app, development starts with the creation of classes and objects to represent the programmatic elements and tasks that make up the app. In this lesson, you will create a class, instantiate objects of that class, use built-in methods to compare objects and perform arithmetic operations on objects, and create a class factory to make object creation easier.

Lesson Objectives

In this lesson, you will:

- Create a class for object-oriented Python applications.
- Work with built-in methods.
- Implement the factory design pattern in Python.

TOPIC A

Create a Class

Many steps go into defining a class and separating what attributes will apply at the class level, and which will be controlled at the object level. In this topic, you will create classes, class methods, instantiate objects, define instance methods, and properties, and learn how to clean up objects when they are no longer needed.

Class Creation

As you likely already know by now, Python® is a fairly clean language and creating a class is no different. To create a class, use the `class` keyword followed by the name of the class and a colon:

```
>>>class Car:
```

This is the class definition, and the name of the class must follow Python rules for naming variables. That is, it must start with a letter or underscore and can only contain letters, underscores, and numbers. It's also recommended the class names use CapWords notation, so the first word and any subsequent words start with a capital letter.

The class definition line is followed by the contents of the class, its attributes and methods, indented by four spaces (following the PEP 8 style guide for Python code).

You can save the class as something like `car_class.py`, then load the class with Python `import car_class.py as Car`. You can then instantiate class objects into the Python 3 interpreter in the following way:

```
>>> ferrari_GTO = Car()
>>> toyota_prius = Car()
```

This instantiates two objects from the `class. named ferrari_gto`, and `toyota_prius`.

Note: This is very similar to a function call, but Python knows the difference between defined functions and classes and, therefore, will create two new objects of the class.

Class Methods

Class methods are functions that are independent of any instance data, attributes, and methods. They are bound to the class, not any one specific object of the class.

Define a class method by adding the `@classmethod` decorator line before the method definition. You then define the method much like a function, including the `def` keyword:

```
>>>class Car:
...    @classmethod
...    def calcMPG(cls, miles, gallons):
...       return miles / gallons
...
```

The class method uses parameters to pass data into the method, and the `return` statement to pass a single value back to the main program. The first parameter must be the `cls` keyword. This tells Python to reference the class instead of an instance of the class.

The `calcMPG` class method uses the variables passed to it in the arguments, and returns the output back to the main program.

Class methods have access to the state of the class and can modify the state of class-defined attributes that apply across all objects of the class. In the `Car()` class example, color might be a variable defined at the class which can be set by a class method, allowing users to define the color of

vehicle and have it apply to all objects of the class. That way, they can select the color they want, and all cars they view will be in the selected color.

Instance Methods

Instance methods work on attributes defined for the instance. Instance methods must always use at least one parameter, called `self`. The `self` parameter points to the instance that's using the method. When you use the instance method, you don't include the `self` parameter, Python assumes it by default.

To define an instance method again, you use the `def` keyword, just like when defining functions:

```
>>>class Car:
...    def __init__(self, make, model, price, inventory):
...       self.make = make
...       self.model = model
...       self.price = price
...       self.inventory = inventory
...    def display(self):
...       print({0} {1} - price: {2:.2f}, inventory: {3:d}'.format(self.make,
                                                                   self.model,
                                                                   self.price,

self.inventory))
...
```

The `__init__` instance method is a special method that defines the initial values for the instance attributes. This will be discussed more later. The `display` method is an instance method created to display the instance values instead of having to use the `print` statement all the time when you want to see the attribute values.

Inside the instance method definitions, the `self` keyword is used to reference the attribute values for the instance. The `__init__` method uses that to apply the parameter values to the instance attributes. The `display` method also uses that in the `self` keyword to reference the values of the attributes for the current instance.

Use instance method in order to gain access to the unique attributes available in a specific instance of a class. For example, you might access the `drive_train` method to select the 4-wheel drive option for an instance that supports it.

Object Initialization with Constructors

The `__init__` ***constructor method*** provides a way for an application to define initial values for attributes when you instantiate it. The default format for a constructor method lists the attribute values as parameters, then assigns the parameter values to the attributes:

```
>>> class Car:
...    def __init__(self, make, model, price, inventory):
...       self.make = make
...       self.model = model
...       self.price = price
...       self.inventory = inventory
...
>>>
```

However, having to provide values for each parameter can be somewhat painful, especially if there are any values that are often the same, such as the number of doors or seats on a sedan object. To prevent having to enter common values, you can define default values inside the constructor definition:

```
>>> class Sedan(Car):
...    def __init__(self, make, model, price, inventory, doors = 4, seats = 5):
...       Car.__init__(self, make, model, price, inventory)
...       self.doors = doors
...       self.seats = seats
...
>>>
```

Properties

You can set properties for a class using the `property()` function which is a fine option for smaller apps. However, in Python, classes are dynamic. Any program using the class can add, delete, or modify instance attributes on the fly. That can be somewhat dangerous, as you lose control over how the class you created is being used.

To regain control over the attributes, Python provides ***properties***. A property is a protected attribute that can only be modified by using special instance methods called ***setters***, and retrieved by using special class methods called ***getters***.

Getters and setters are often used in Python to:

- Add validation logic for values being set and retrieved.
- To create protected attributes, making them essentially private to prevent direct access and modification by external users.

To identify a property, start the attribute name with two underscores:

```
... self.__make = make
... self.__model = model
... self.__price = price
... self.__inventory = inventory
```

You must define getters and setters for each property. The getters start with `get_` and the setters start with `set_`, like this:

```
>>>class Car:
...    def __init__(self, make):
...       self.__make = make
...    def get_make(self):
...       return self.__make
...    def set_make(self, make):
...       self.__make = make
...
>>>
```

This example is quite simple, but now that you have methods that intervene when you set or get data values, you can use those methods to perform certain checks or perform other tasks when the data values are being changed. For example, in the setter, you could write code to make sure the data value is in a valid format or within an appropriate range. You might trigger changes in other attributes, too.

The downside to using getters and setters is that now you need to remember and use separate methods for each property and function that you need to use in your program. For programs with lots of properties, that can be a lot of methods to use! To solve that problem, Python uses the ***property method***. With the `property` method, you define the getter, setter, and an optional deleter method for the property:

```
... make = property(get_make, set_make, del_make)
```

You still need to define all of the getter, setter, and deleter methods in your class code, but at least now using them is much easier. The program can reference the property with just the property name:

```
>>> car1.make = "Chevy"
```

Python calls the appropriate getter or setter method to accomplish the function.

Destructor Methods

Another useful method is the ***destructor***. The destructor method runs when you delete an instance of the class. The destructor uses the `__del__` method:

```
...    def __del__(self):
...
```

The code that you define in the destructor only runs when a class instance is removed from memory by Python as the program runs. You can manually delete a class instance at any time using the `del` statement.

Destructor methods are useful if you have cleanup to perform before the instance goes away, such as closing files and committing changes to databases.

Note: Python will attempt to automatically remove unused class instances from memory when it detects they are no longer in use in an application. However, for large applications that use a lot of memory, it can be beneficial to manually delete the instance yourself to help increase performance.

Modules

In a small program, you can keep all of your code in one place. As programs grow and become more complex, it gets harder to locate and edit specific classes in the code base. Modules help solve this issue. Modules are Python files such as `car_class.py`. One file is one module, two files are two modules. You can load modules as you need them for use in your app. For example, if you have two modules in the same folder, you can load a class from one module and have it call the other module. Storing them in modules helps you keep the code separate and organized which will make maintenance and update easier.

For example, if you're accessing data on cars from a database, you might put all classes and functions related to accessing the database into a module named `dbaccess.py`.

Use the `import` statement to instantiate modules or specific classes or functions from modules in your app. You can import from built-in or custom-created modules.

For example, the following code could be used to import data from a database module:

```
import dbaccess
db = dbaccess.Database()
```

This makes all classes and functions in the module available using the `dbaccess.<class or function>` notation.

You can also import a specific class:

```
from dbaccess import Database
db = Database()
```

You can also rename the class in case you are using the existing name somewhere else in your app's namespace.

```
from dbaccess import Database as Cars
db = Cars()
```

You can also import multiple classes from a module:

```
from dbaccess import Database, Access_Queries
```

This loads the `database` class and the `Access_Queries` class.

String Interpolation with f-string

Many Python apps at one point or another generate output for the user to read. If the output is being generated at a command prompt or IDE, you will need to format the string output to include text, variables, and other data you wish to include. Python 3.6 added the formatted string literals approach, or f-string. It's called f-string because, the syntax is an `f` followed by the string to be formatted.

```
>>> f"My car is a {self.make}."
My car is a Chevy.
```

F-string allows you to embed expressions and can perform arithmetic operations in line:

```
>>>chevystock  = 25
>>> fordstock = 10
>>> f"We have {chevystock} Chevrolets and {fordstock} Fords in stock for
{chevystock + fordstock} cars total."
We have 25 Chevrolets and 10 Fords in stock for 35 cars total.
```

Additional Information

For more information on f-string interpolation, see: **https://www.python.org/dev/peps/pep-0498/**.

Class Creation Code Examples

You create a complete class definition by putting together the class definition, attributes, and methods into one code block.

Define a Class

Define a new class by using the `class` keyword, along with the class name and a colon. The class name you choose must be unique within your program. Although it's not required, it's somewhat of a de facto standard in Python to start class names with an uppercase letter:

```
>>> class Car:
...
```

Define Class Attributes

After creating the class, you can define any class attributes. The class attributes apply to all instances of the class. You reference the class attributes directly from the class and not from an instance:

```
...    wheels = 4
...    axles = 2
...
```

Define a Class Method

Class methods are defined to provide methods for use outside of class instances, such as common functions that can be used with the object. Remember to use the `@classmethod` decorator before defining the class method, and use the `cls` keyword as the first parameter:

```
...    @classmethod
...    def calcMPG(cls, miles, gallons):
...        return miles / gallons
...
```

Note: Be careful when defining methods in a class definition. Make sure that the method code block is indented farther than the method definition statement.

Define an Instance Method

Define instance methods that support the class object. Use the `__init__` method to create a class constructor to define any instance variables for the class. Then create additional instance methods to support functions required to interact with the class objects:

```
...   def __init__(self, make, model, price, inventory):
...      self.make = make
...      self.model = model
...      self.price = price
...      self.inventory = inventory
...   def sellCar(self, quantity):
...      self.inventory = self.inventory - quantity
...
>>>
```

Create an Instance

The class definition creates the class template, but doesn't put the class to use. To use a class, you have to instantiate it. When you instantiate a class, you create an instance of the class in your program. Each instance represents one occurrence of the object. Each instance is a separate object in Python. If you create a second instance of the `Car` class, the attributes you define are separate from the attributes you define in the first instance:

```
>>> car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> car1.make
Chevy
>>>car2 = Car("Ford", "Focus", 10000.00, 5)
>>> car2.make
Ford
>>> car1.make
Chevy
>>> Car.wheels
4
>>>Car.calcMPG(120, 3.3)
36.36363636363637
>>>>>>car1.sellCar(3)
>>>car1.inventory
7
>>>
```

When you create one or more properties for a class, you must also create the getters and setters for each property.

Create a Property

Use two underscores in front of each attribute name that you want to become a property:

```
>>>class Car:
...   def __init__(self, make, model, price, inventory):
...      self.__make = make
...      self.__model = model
...      self.__price = price
...      self.__inventory = inventory
...
```

Create a Getter Method

When you create properties, create a getter method for each property. The getter method just returns the value of the property to the calling program:

```
...   def get_make(self):
...      return self.__make
...   def get_model:
...      return self.__model
...   def get_price:
...      return self.__price
...   def get_inventory:
...      return self.__inventory
```

Create a Setter Method

For each property, create a setter method so that programs that use the class can change the value of the property:

```
...   def set_make(self, make):
...      self.__make = make
...   def set_model(self, model):
...      self.__model = model
...   def set_price(self, price):
...      self.__price = price
...   def set_inventory(self, inventory):
...      self.__inventory = inventory
```

Define a Property() Method

After you define the getters and setters, you can combine them in property methods:

```
...   make = property(get_make, set_make)
...   model = property(get_model, set_model)
...   price = property(get_price, set_price)
...   inventory = property(get_inventory, set_inventory)
```

Use the Property in an Instance

To reference the class properties, don't use the underscores in the attribute, but instead use the property names:

```
>>>car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> car1.make
'Chevy'
>>> car1.make = "Ford"
>>> car1.make
'Ford'
>>>
```

Using a Constructor

When you define a constructor, the instance must provide a value for each parameter defined in the constructor. If you don't provide enough parameters, you'll get an error message from Python:

```
>>> class Car:
...   def __init__(self, make, model, price, inventory):
...      self.make = make
...      self.model = model
...      self.price = price
...      self.inventory = inventory
...
>>> car1 = Car("Chevy", "Volt", 15000.00)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'inventory'
>>>
```

Defining Default Attribute Values

If you don't want to have to define values for all of the attributes, define default attribute values in the constructor. Then, any missing parameters are assumed to take their default values. A good practice is to initialize all instance attributes in the constructor, even if they only create placeholders. That will prevent exceptions for situations when an attribute doesn't exist:

```
>>> class Sedan(Car):
...   def __init__(self, make, model, price, inventory, doors = 4, seats = 5):
...      Car.__init__(self, make, model, price, inventory)
...      self.doors = doors
...      self.seats = seats
```

```
...
>>> sedan1 = Sedan("Chevy", "Volt", 15000.00, 10)
>>>sedan1.display()
This sedan has 4 doors and 5 seats
>>>
```

Guidelines for Creating a Class

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines when creating a class in Python.

Create a Class

When creating a class in Python:

- Define a constructor for the class using the `__init__()` method.
- Define properties for the class by creating getters, setters, and deleters for any attributes you want to define.
- Create a constructor when you need to initialize attributes or properties in your classes.
- Define commonly used attribute values in a constructor.
- Create a destructor if you need to perform housekeeping functions on a class before it deletes, such as closing files.
- Use the `del` statement to remove a class object from memory when it's no longer needed in an application.

ACTIVITY 2-1
Creating and Initializing a Class

Before You Begin
Ensure that Python version 3.9 and the PyCharm Community Edition version 2020.2 are installed on your computer.

Scenario
You will create an application that maintains the inventory and specifications of the wood products. As you use object-oriented programming classes in your Python applications, you'll want to save them in separate module files. Using PyCharm, you'll create a new project to build a module for handling the `Wood` and `Plywood` classes. You can then use that module in any Python application that requires those classes. You will create a new module to test creation and use of the new classes.

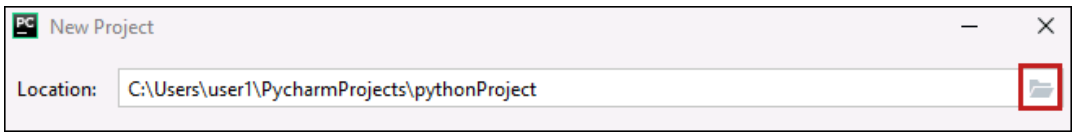
Note: Activities may vary slightly if the software vendor has issued digital updates. Your instructor will notify you of any changes.

---

1. Open PyCharm and create a new project.
   a) From the desktop, double-click the **PyCharm Community Edition 2020.2.3 x64** shortcut.
   b) In the Welcome to PyCharm window, select **New Project**.

Note: If you already have a project open, select **File→Close Project** from the menu bar to open the PyCharm Community Edition window, then select **New Project**.

   c) In the **Location** box, select the **Browse** button.



   d) Navigate to the **C:\094022Data\Creating Object-Oriented Python Applications** folder.
   e) On the toolbar, select the **New Folder** button.
   f) In the **Enter a new folder name** box, type ***WWProject-L2***, and select **OK**.
   g) Select **OK** again to select the new folder.

Note: If you receive a message from Windows Firewall, select **Allow Access**.

   h) In the **New Project** window, select **Existing interpreter**.
   i) In the **Interpreter** box, verify that **C:\Users\<username>\AppData\Local\Programs\Python\Python39\Python.exe** is selected.
   j) Uncheck the **Create a main.py welcome script** check box.
   k) Select **Create** to create the project.

Note: If the **Tip of the Day** dialog box is shown, close it.

   l) In the **Project** pane, on the left side of the PyCharm window, verify that **WWProject-L2** is listed.

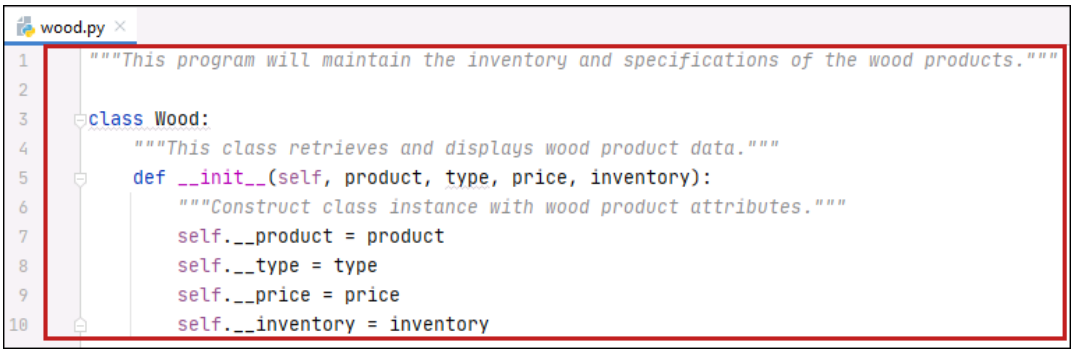
Note: If at any time during this course, you receive a message about Windows Defender impacting performance, select **Don't show again**.

2. Create the class module file.

   a) Right-click **WWProject-L2** and select **New→Python File**.
   b) In the **Name** box, type ***wood*** and press **Enter**.

3. Type the code to create the `Wood` class.

   a) Starting on line 1, type the class definition statement, and define the constructor by adding the following code.



Note: Type your code so that it matches the code image exactly. Indentation and capitalization are very important and can cause issues if they do not match the code images.

b) Starting on line 12, type the setter and getter methods by adding the following code after the constructor.

```
10          self.__inventory = inventory
11
12      # Define setter and getter methods for wood product attributes.
13      def get_product(self):
14          return self.__product
15      def set_product(self, product):
16          self.__product = product
17
18      def get_type(self):
19          return self.__type
20      def set_type(self, type):
21          self.__type = type
22
23      def get_price(self):
24          return self.__price
25      def set_price(self, price):
26          self.__price = price
27
28      def get_inventory(self):
29          return self.__inventory
30      def set_inventory(self, inventory):
31          self.__inventory = inventory
```

Note: Alignment matters in Python. Please make sure your code is indented as shown in the images.

c) Starting on line 33, type the property definitions for each attribute of the `Wood` class by typing the following after the getter and setter methods.

```
31          self.__inventory = inventory
32
33      #Property definitions for each attribute.
34      product = property(get_product, set_product)
35      type = property(get_type, set_type)
36      price = property(get_price, set_price)
37      inventory = property(get_inventory, set_inventory)
```

d) Starting on line 39, type the `display()` and `sellWood()` methods by typing the following after the property definitions.

```
37      inventory = property(get_inventory, set_inventory)
38
39      def display(self):
40          """Display the attributes of a wood product."""
41          print("{0} {1} - price:{2:.2f}, inventory:{3:d}".format(self.product,
42                                                                  self.type,
43                                                                  self.price,
44                                                                  self.inventory))
45
46      def sellWood(self, quantity):
47          """Sell a specified number of a wood product."""
48          self.__inventory = self.__inventory - quantity
49          print(f"\nSold {quantity} pieces of {self.product}\n")
```

Note: PyCharm autosaves the code so there is no need to save it. This also means that if you want to undo any changes, you must do so before closing the project.

4. Type the code to create the `Plywood` subclass in the **wood.py** file.

a) After the `Wood` class code, starting on line 51, type the code to create the `Plywood` subclass.

```
49          print(f"\nSold {quantity} pieces of {self.product}\n")
50
51  class Plywood(Wood):
52      """This class retrieves and displays plywood data."""
53      def __init__(self, product, type, price, inventory, width, length):
54          """Construct class instance with plywood attributes."""
55          super(Plywood, self).__init__(product, type, price, inventory)
56          self.__width = width
57          self.__length = length
58
```

b) After the `Wood` class code, starting on line 59, type the code to define the setter and getter methods.

```
57          self.__length = length
58
59      # Define setter and getter methods for plywood specific attributes.
60      def get_width(self):
61          return self.__width
62      def set_width(self, width):
63          self.__width = width
64
65      def get_length(self):
66          return self.__length
67      def set_length(self, length):
68          self.__length = length
69
```

c) After the `Wood` class code, starting on line 70, type the code to create the property definitions for each attribute and display the attributes.

```
68          self.__length = length
69
70      #Property definitions for each attribute.
71      width = property(get_width, set_width)
72      length = property(get_length, set_length)
73
74      def display(self):
75          """Display the attributes of a plywood product."""
76          print(f"This Plywood is {self.width} inches wide and {self.length} inches long")
77
```

5. Create the **woodtest.py** test file in the **WWProject-L2** project.

a) In the **Project** pane on the left, right-click the **WWProject-L2** project, and select **New→Python File**.

b) In the **Name** box, type ***woodtest*** and press **Enter**.

6. Write code to create an instance of each class.

a) Starting on line 1, type the following to import the `Wood` and `Plywood` class definitions from the **wood.py** file.

```
wood.py  woodtest.py
1   """This program will import the Wood and Plywood class definitions and test them."""
2
3   from wood import *
```

b) Starting on line 5, type the following to create an instance of both the `Wood` and `Plywood` classes.

```
3   from wood import *
4
5   #Create an instance of the wood and plywood classes.
6   wood1 = Wood("Lumber-2x4 inch", "Untreated", 5.50, 10)
7   plywood1 = Plywood("Plywood-1/4 inch", "Sanded", 33.00, 10, 48, 96)
8
```

7. Enter code to perform some basic tests of the `Wood` and `Plywood` classes.

a) Starting on line 9, type the following to display both the `Wood` and `Plywood` instance values using the `display()` method.

```
7   plywood1 = Plywood("Plywood-1/4 inch", "Sanded", 33.00, 10, 48, 96)
8
9   #Display the wood and playwood instance values.
10  wood1.display()
11  plywood1.display()
12
```

b) Starting on line 13, type the following to simulate selling three pieces of wood by using the `sellWood()` method, then display the results.

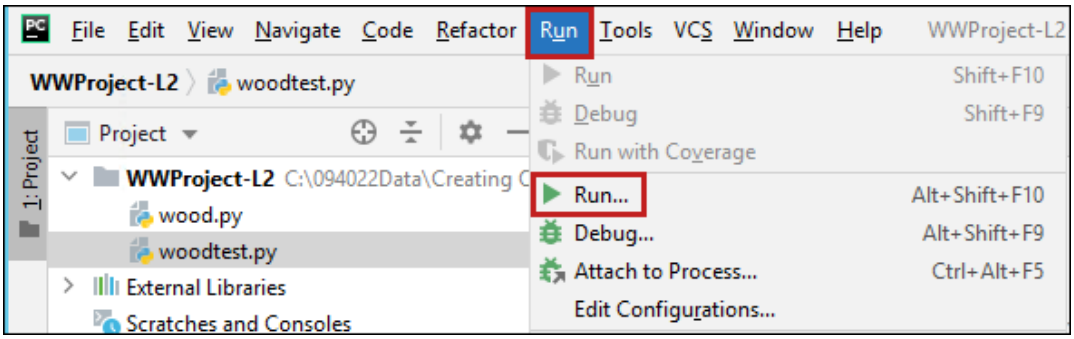
```
11  plywood1.display()
12
13  #Sell three pieces of wood and display the results.
14  wood1.sellWood(3)
15  wood1.display()
16
```

c) Starting on line 17, type the following to simulate changing the plywood information by changing the width to **24**.

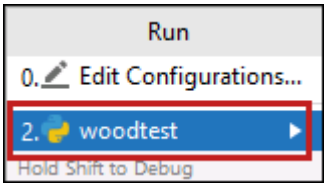
```
15  wood1.display()
16
17  #Change the playwood width to 24 inches and display the results.
18  plywood1.width = 24
19  plywood1.display()
20
```

8. Run the **woodtest.py** program to test it.

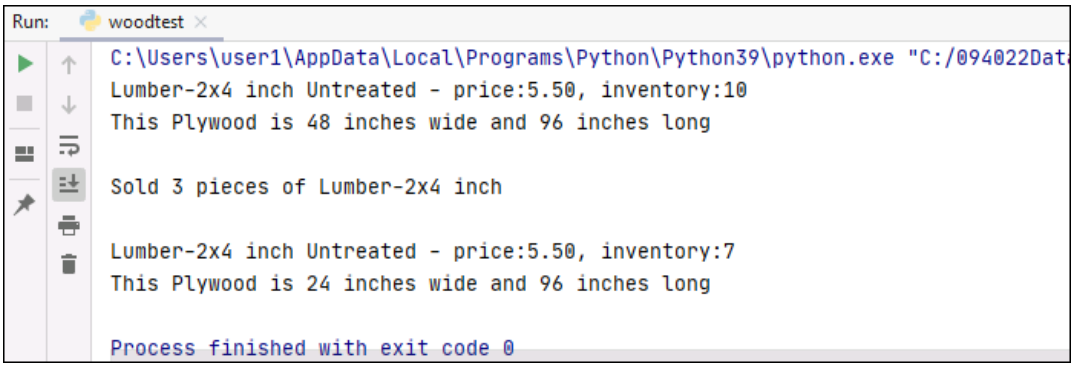
a) Select **Run→Run**.



b) In the pop-up window, select **woodtest**.



c) Examine the output.



The first two lines show the `wood1` and `plywood1` instance values that you defined. The third line shows that three pieces of Lumber-2x4 inch were sold. The fourth line shows the result of the `sellWood(3)` method that you ran, and the fifth line shows the result of changing the `width` property of the plywood to 24.

d) In the **Run** pane, close the **woodtest** tab.

TOPIC B

Use Built-in Methods

Python supplies built-in methods to allow you to perform some common takes with objects. In this topic, you will use comparison methods to compare objects, and numeric methods to perform arithmetic operations on objects.

Comparison Methods

It's common programming practice to use if-then statements in Python code to perform comparisons between two values, such as to compare two variables that hold integers, or two strings. Sometimes, it comes in handy to be able to compare two objects, as well, but that can get somewhat tricky.

Just how do you compare two `Car` class instances? There can be many attributes assigned to a class, so it would be impossible for Python to know just which ones to compare:

```
>>> car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> car2 = Car("Ford", "Focus", 10000.00, 20)
>>> if (car1 > car2):
...     print("The Volt is bigger")
...
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unorderable types: Car() > Car()
>>>
```

Python has no way to determine what the greater-than operation means when working with the `Car` class objects. However, Python does provide a way for us to tell it just what to do. This is called a ***comparison method***:

```
...   def __gt__(self, other):
...       if (self.price > other.price):
...           return True
```

This code uses the price attribute of the `Car` class to determine if one `Car` object is greater than another. Since it is a method, it returns a True Boolean value if the condition is true. Not only can you now use the `__gt__` method for the class, you can also use the actual greater-than symbol (>) to compare `Car` objects.

List of Comparison Methods

Python provides a whole host of comparison ***magic methods*** that you can define in your classes when you need to use them in comparisons.

Comparison Method	*Description*
`__eq__`	Equal
`__ge__`	Greater-than or equal
`__gt__`	Greater-than
`__le__`	Less-than or equal
`__lt__`	Less-than
`__ne__`	Not equal

You don't need to define all of the comparison methods in the class, only the ones you need to use in your applications.

Comparison Method Example

Define comparison methods after the constructor. You don't need to define all of the comparison methods, just the ones that you need to use for your class:

```
>>> class Car:
...   def __init__(self, make, model, price, inventory):
...      self.make = make
...      self.model = model
...      self.price = price
...      self.inventory = inventory
...   def __gt__(self, other):
...      if (self.price > other.price):
...         return True
...   def __lt__(self, other):
...      if (self.price < self.other):
...         return True
...   def __eq__(self, other):
...      if (self.price == self.other):
...         return True
...
>>> car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> car2 = Car("Ford", "Focus", 10000.00, 3)
>>> if (car1 > car2):
...   print("The {0} is more expensive than the {1}".format(car1.model,
car2.model)
...
The Volt is more expensive than the Focus
>>>
>>> car1.price = 10000.00
>>> if (car1 == car2):
...   print("Both cars are the same price")
...
Both cars are the same price
>>>
```

Numeric Methods

Python also includes several numeric operation methods. Numeric methods enable you to define how your classes will behave when basic mathematical operations (such as addition, subtraction, multiplication, and division) are performed on them, treating the classes as though they were simple numbers. Numeric methods also support some advanced mathematical operations, such as raising values to a power, or shifting values left or right.

For our `Car` class, if you want to be able to find out the price difference between two cars, first define the `__sub__` method in your `Car` class:

```
...   def __sub__(self, other):
...      return self.price - other.price
...
```

Then, you can easily subtract one car from another, implying that it's actually the price you want to subtract:

```
difference = car1 - car2
```

List of Numeric Methods

As you can guess, Python provides numeric magic methods for most of the common mathematical operations you'll run into.

Numeric Method	*Description*
`__add__`	Addition
`__and__`	Boolean AND
`__div__`	Division
`__divmod__`	Division modulo
`__floordiv__`	Returns the integer part of the division
`__lshift__`	Shifts the number left
`__mod__`	Modulo
`__mul__`	Multiplication
`__or__`	Boolean OR
`__pow__`	Raising to a power
`__rshift__`	Shifts the number right
`__sub__`	Subtraction
`__xor__`	Boolean XOR

Just as with the comparison methods, you don't need to define all of the numeric methods for your classes, just the ones you think you'll need to use in your applications.

Numeric Method Example

Numeric methods provide a way for you to define how your classes will behave when simple mathematical operations are performed on them.

Just as with comparison magic methods, you only need to define the numeric magic methods that you think you'll need to use for the class objects. Just define them anywhere after the constructor method:

```
>>> class Car:
...    def __init__(self, make, model, price, inventory):
...       self.make = make
...       self.model = model
...       self.price = price
...       self.inventory = inventory
...    def __add__(self, other):
...       return self.price + other.price
...    def __sub__(self, other):
...       return self.price - other.price
...

>>> car1 = Car("Chevy", "Volt", 15000.00, 10)
>>> car2 = Car("Ford", "Focus", 10000.00, 3)
>>> car1 - car2
5000.0
>>>
```

Guidelines for Working with Built-in Methods

Follow these guidelines when working with built-in methods.

Work with Built-in Methods

When working with built-in methods:

- Use comparison methods to compare attributes of one class instance with another.
- Use comparison magic methods to perform comparisons.
- Use numeric methods to define how your classes will behave when basic mathematical operations are performed on them.
- Use numeric magic methods to perform arithmetic operations with objects.

ACTIVITY 2-2

Working with Comparison and Numeric Methods

Before You Begin

You have created the **WWProject-L2** project with **wood.py** and **woodtest.py**. The **WWProject-L2** project is open in PyCharm, and **wood.py** is open for editing.

Scenario

You will now add a constructor to the `Plywood` class that assumes default values for the width and length attributes in the constructor. You will also add a destructor method in the `Plywood` class that displays the final values of the `plywood` instance before it is deleted.

You will add methods to the `Wood` class that enable you to compare prices on wood. You will add a check for greater-than, less-than, and equal to the class definition.

---

1. In **wood.py**, edit the `Plywood` constructor to add default values.
   a) In **wood.py**, scroll down to the `Plywood` class definition area.
   b) On line 53, modify the `__init__` function to set the default value of the `width` parameter to ***48*** and the `length` parameter to ***96***

```
51  class Plywood(Wood):
52      """This class retrieves and displays plywood data."""
53      def __init__(self, product, type, price, inventory, width = 48, length = 96):
54          """Construct class instance with plywood attributes."""
55          super(Plywood, self).__init__(product, type, price, inventory)
56          self.__width = width
57          self.__length = length
```

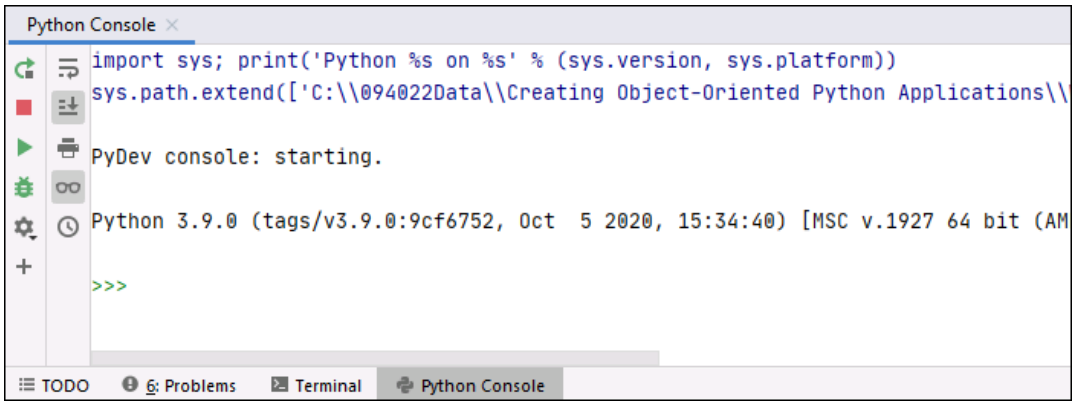
2. Define the destructor.

a) Under the `__init__` constructor section, but before the `get_width()` method, starting on line 59, type the following to add a `__del__` section that uses the `print` method to display the instance values of all the attributes.

```
51  class Plywood(Wood):
52      """This class retrieves and displays plywood data."""
53      def __init__(self, product, type, price, inventory, width = 48, length = 96):
54          """Construct class instance with plywood attributes."""
55          super(Plywood, self).__init__(product, type, price, inventory)
56          self.__width = width
57          self.__length = length
58
59      def __del__(self):
60          """Delete a product."""
61          print(f"The {self.type} {self.product} that is {self.width} "
62                f"by {self.length} inches is being deleted")
63
64      # Define setter and getter methods for plywood specific attributes.
```

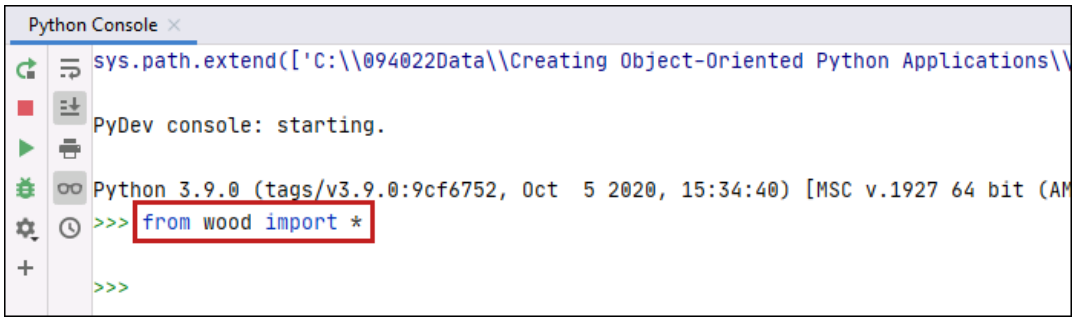
3. Test the new constructor and destructor in the Python Console.

a) Select **Tools→Python or Debug Console**.
The **Python Console** pane appears at the bottom of the PyCharm window.

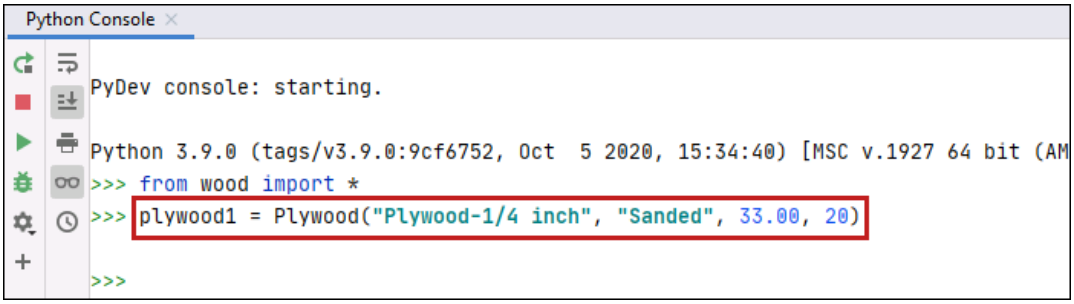


b) If prompted about Windows Firewall, select **Allow access**.

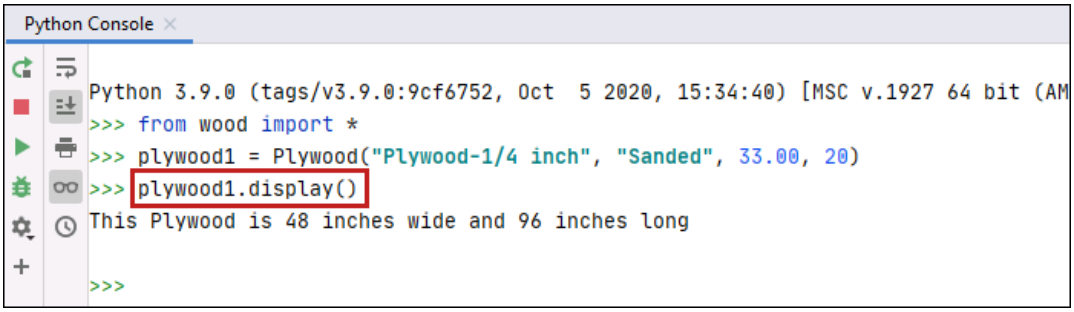
c) In the **Python Console** pane, enter the following to import all of the classes from the **wood.py** module.



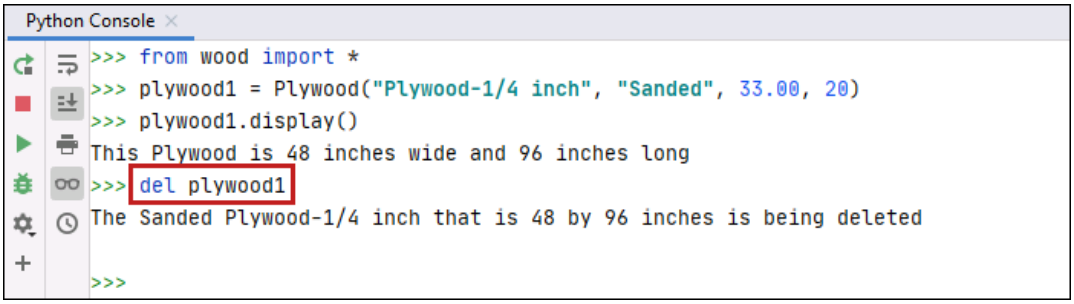
d) Enter the following to create an instance of the `Plywood` class, but don't specify the width or length.



e) Enter the following to display the instance attributes by using the `display()` method.

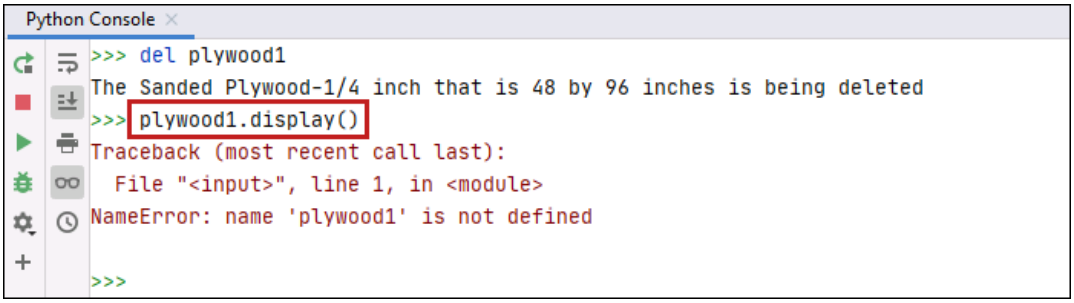


f) Enter the following to use the `del` method to destroy the `plywood1` instance you created.

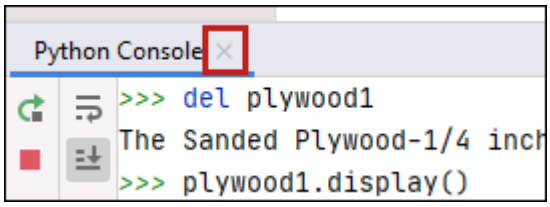


A message is displayed by the destructor that you created.

g) Enter the following to try to display the `plywood1` instance to verify that it no longer exists. You will get an error message that looks like the following.



h) Close the Python Console pane.



4. Edit the `Wood` class to add the new methods.

a) Following the `__init__` method definition, starting on line 12, type the following to add new definitions for the `__eq__`, `__gt__`, and `__lt__` methods as shown.

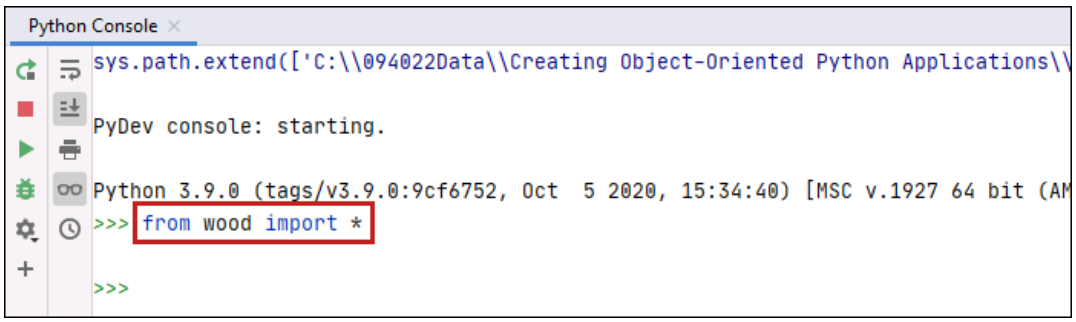
```
7           self.__product = product
8           self.__type = type
9           self.__price = price
10          self.__inventory = inventory
11
12      def __eq__(self, other):
13          """Define the equal method."""
14          if (self.price == other.price):
15              return True
16
17      def __gt__(self, other):
18          """Define the greater than method."""
19          if (self.price > other.price):
20              return True
21
22      def __lt__(self, other):
23          """Define the lesser than method."""
24          if (self.price < other.price):
25              return True
26
27      # Define setter and getter methods for wood product attributes.
```

In each case, a Boolean `True` value is returned if the condition is met.

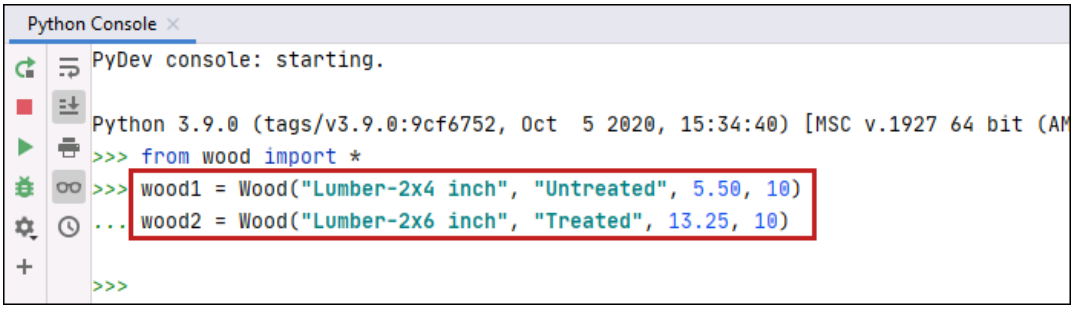
5. Test the new methods using the Python Console.

a) Select **Tools→Python or Debug Console**.

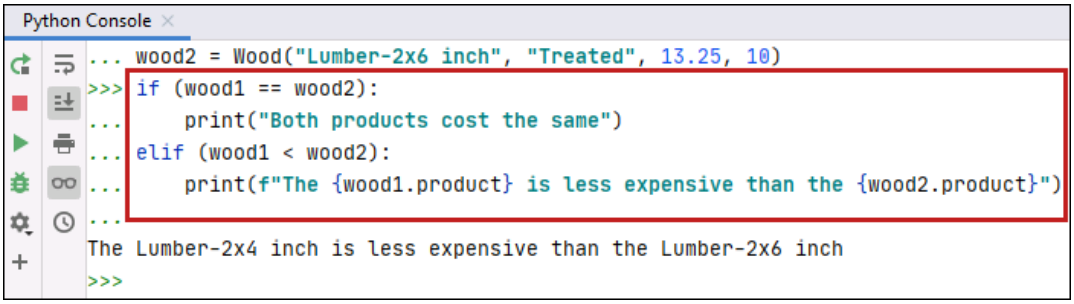
b) Enter the following to import the `Wood` class from the **wood.py** module file.



c) Enter the following to create two instances of the `Wood` class, one more expensive than the other.



d) Enter the following to check if both of the `Wood` instances have the same price or if one is less than the other by using an `if-elif` statement and a `print` statement.



Note: You must press **Enter** twice after the fourth line to return to the `>>>` prompt.

Since the wood prices are not the same, the `if-then` statement fails, and the first `print` statement is skipped. Since `wood1` is less than `wood2`, the `elif` statement is true, and the second `print` statement is executed.

e) Close the Python Console pane.

TOPIC C

Implement the Factory Design Pattern

In object-oriented programming, creating and managing objects can be difficult and time consuming. In this topic, you will create a class factory to make object creation and management easier.

Design Patterns

Design patterns are used to formalize the structures used in software development. Design patterns are not a finished application that completes a task or solves a specific problem, but a repeatable structure designed to solve common problems, speed up code creation, and to ensure members of the development team approach coding tasks in the same way.

In this sense, design patterns can be thought of in a similar way to a document template. An organization might require certain information in order for expenses to be approved. They might need the date the expense was incurred, the approver, the project or cost center the expense is related to, and a receipt. The organization might create a template to ensure this information is always captured in a standard way, so that expense reports can be filled out correctly by anyone and be processed efficiently.

In the same way, selecting a design pattern can help ensure coders working on separate and distinct portions of the code and classes of objects are creating them in the same way, making code easier to read and integrate with the rest of the code. Using a design pattern can also prevent small coding issues from becoming large problems as the code base grows and more code is integrated, and it makes code easier to maintain and troubleshoot over the long term.

Types of Design Patterns

There are many types of design patterns; some (not all) of the most well known are listed in the following table, with a brief description of the general types of problems they attempt to address.

Category of Design Pattern	*Examples*	*Description*
Creational	• Abstract Factory • Builder • Factory Method • Object Pool	Creational design patterns address class instantiation, that is class and object creation.
Structural	• Adaptor • Bridge • Composite • Private Class Data • Proxy	These design patterns govern how objects should be composed to gain the functionality required from the objects.

Category of Design Pattern	*Examples*	*Description*
Behavioral	• Chain of Responsibility • Command • Interpreter • State	This type of design pattern governs how class objects communicate with one another.

Note: The Factory Pattern is used in this course and is the focus of the remainder of this topic.

Additional Information

For additional information on design patterns, see: **https://sourcemaking.com/design_patterns**.

Factory Pattern or Class Factory

Factory patterns, also called a class factory, gather all subclasses that belong to the same superclass together into a single class definition. The factory provides one-stop shopping to instantiate any of the subclasses from a single class method. While this may sound confusing, it actually can help keep your subclasses in order, and help you to easily find just the right subclass for your application.

The key to a class factory is the `factory` method that you define for the superclass object. The factory method uses a single parameter—the name of the subclass that you want to instantiate. Based on the parameter, the factory instantiates the requested subclass object. The code to implement a simple class factory looks like the following:

```
class Car:
   def factory(type):
      class Sedan(Car):
         def display(self):
            print("This car is a sedan")
      class Sports(Car):
         def display(self):
            print("This car is a sports car")
      if (type == "Sedan"):
         return Sedan()
      if (type == "Sports"):
         return Sports()
```

With this approach, you can group all of the subclass definitions together in the factory class for easy access to any subclass.

For example, with a simple class factory, you just need to call the `factory` method of the superclass with the name of the subclass you want:

```
>>> car1 = Car.factory("Sedan")
>>> car1.display()
This car is a sedan
>>>
```

Polymorphic Class Factory

Often, subclasses overload the same class methods, tweaking the methods so that they incorporate the differences related to each subclass. Trying to keep track of all the overloaded methods for each subclass can be yet another object-oriented programming nightmare. It would be nice to be able to

call a single method in your programs to retrieve the information specific to the subclass that you're working with. The polymorphic class factory does just that.

The ***polymorphic class factory*** enables you to call a single method to instantiate any subclass from a superclass. The polymorphic class factory references all of the overload methods in subclasses and uses the correct one based on the subclass that was instantiated. There's no need to worry about using the correct method for the correct subclass. The polymorphic class factory does that for you.

Assume that you have two subclasses that overload the same method:

```
class Sedan(Car):
   def display(self):
      print("This is a sedan")

class Sports(Car):
   def display(self):
      print("This is a sports car")
```

Both the `Sedan` and `Sports` subclasses define the `display()` method, and each one does something different. You can now create a polymorphic class factory method that calls the appropriate subclass `display()` method based on the subclass of the object being used:

```
def displayCar(carType):
   carType.display()
```

Now you can create multiple objects of different subclasses and call the same `displayCar()` method to run the `display()` method for any of the subclasses, and Python will use the appropriate class method for each object:

```
>>> car1 = Sedan()
>>> car2 = Sports()
>>> displayCar(car1)
This is a sedan
>>> displayCar(car2)
This is a sports car
>>>
```

Abstract Class Factories

You may run into situations where the superclass used for an object isn't used to create any instances; all of the object instances in the application are created using subclasses. Because of that, there isn't any reason to create a detailed class definition for the superclass. Each subclass just defines the attributes and methods that it needs:

```
>>> class Car:
...    pass
...
>>> class Sedan(Car):
...    def display(self):
...       print("This is a sedan")
...
>>> class Sports(Car):
...    def displayCar(self):
...       print("This is a sports car")
...
>>>
```

Notice, though, that the two separate subclasses use different methods to display the object information. Because of that, you can confuse which methods belong to which subclass. If you accidentally use the wrong method for a subclass, you'll get an error message:

```
>>> car1.displayCar()
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
AttributeError: 'Sedan' object has no attribute 'displayCar'
>>>
```

To avoid this situation, you can create an ***abstract class factory***. The abstract class factory creates the superclass with abstract templates for any method that you want to use for subclasses. You don't use any code here, other than to provide an error message to display if the method isn't overloaded by the subclass:

```
class Car:
   def displayCar():
      print("Sorry, this method is not implemented")
   def sellCar():
      print("Sorry, this method is not implemented")
```

Each subclass definition must overload all of the methods defined in the abstract class factory. If any methods aren't overloaded and a program attempts to use them, the superclass method takes over and displays an error message.

Guidelines for Using Design Patterns

Use the following guidelines when creating classes to determine when you should use a design pattern or class factory.

Use Design Patterns

When using a design pattern or class factory:

- Create a simple class factory when you have multiple subclasses and want to easily manage them from the superclass.
- When creating a simple class factory, create a factory method in the superclass.
- When creating a simple class factory, create the subclasses inside the factory method.
- Ensure that the factory method creates the appropriate subclass object based on the parameter passed to it.
- When creating a polymorphic class factory, create a separate method for each overloaded method in the subclasses.
- Ensure the polymorphic method runs the appropriate subclass overload method based on the parameter passed to it.
- When creating an abstract class factory, ensure you create method templates for all of the overloaded methods used in subclasses.
- Ensure the superclass method templates produce some type of error message or signal if a subclass doesn't overload the method.

ACTIVITY 2-3
Creating Class Factories

Before You Begin
You have created the **WWProject-L2** project with **wood.py** and **woodtest.py**. The **WWProject-L2** project is open in PyCharm, and **wood.py** is open for editing.

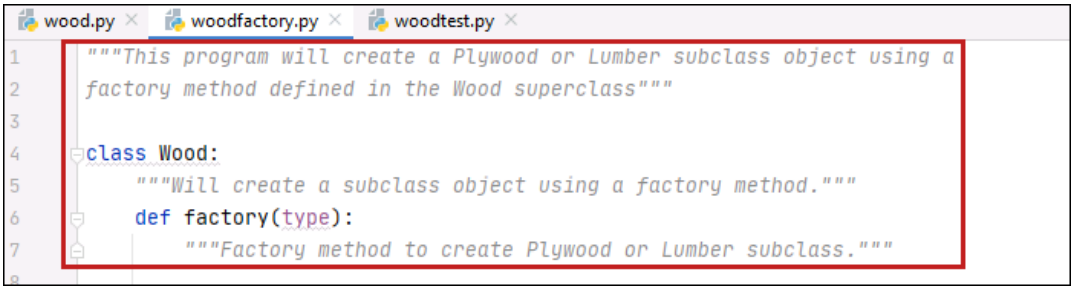
Scenario
Create a simple class factory that can create a `Plywood` or `Lumber` subclass object using a single factory method defined in the `Wood` superclass. Don't worry about adding attributes or extra methods for this exercise; focus on the factory method and how to allow it to create subclass objects.

You will then write code that uses a class factory to generate a `Plywood` and a `Lumber` subclass object, and you will use the `display()` method to test which object is which.

---

1. Create the **woodfactory.py** file in the **WWProject-L2** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L2** project, and select **New→Python File**.
   b) In the **Name** box, type ***woodfactory*** and press **Enter**.

2. Enter the code for the `Wood` class.
   a) At the top of the open **woodfactory.py** file, starting on line 1, type the following to start the `Wood` class definition, and the `def` statement to start the `factory()` method as shown.



b) Under the `factory()` definition, starting on line 9, type the following to create the `Plywood` subclass.

```
4   class Wood:
5       """Will create a subclass object using a factory method."""
6       def factory(type):
7           """Factory method to create Plywood or Lumber subclass."""
8
9           class Plywood(Wood):
10              """Will display plywood text."""
11              def display(self):
12                  print("This is new Plywood")
```

This class contains a single method named `display()` that just displays some text indicating that the method has been run, as shown.

c) After the `Plywood` subclass definition, starting on line 14, type the following to create the `Lumber` subclass.

```
9           class Plywood(Wood):
10              """Will display plywood text."""
11              def display(self):
12                  print("This is new Plywood")
13
14          class Lumber(Wood):
15              """Will display lumber text."""
16              def display(self):
17                  print("This is new Lumber")
```

This class also includes a single method named `display()` that just displays some text indicating that the method has been run.

d) After the `Lumber` subclass definition, starting on line 19, type the following to add an `if-elif` statement that tests the type attribute and run the appropriate subclass.

```
16              def display(self):
17                  print("This is new Lumber")
18
19          #Test the type attribute and run appropriate subclass.
20          if (type == "Plywood"):
21              return Plywood()
22          elif (type == "Lumber"):
23              return Lumber()
```

3. Write code to instantiate and display two different classes of wood.

a) In **woodfactory.py**, skip a line after the end of the `Wood` class definition, then starting on line 25, type the following to create Plywood and Lumber products, using the `Wood` class' `factory()` method.

```
22          elif (type == "Lumber"):
23              return Lumber()
24
25  #Create Plywood and Lumber products.
26  plywood1 = Wood.factory("Plywood")
27  lumber1 = Wood.factory("Lumber")
28
```

b) Skip a line and starting on line 29, type the following to call the `display` method for each wood product.

```
25  #Create Plywood and Lumber products.
26  plywood1 = Wood.factory("Plywood")
27  lumber1 = Wood.factory("Lumber")
28
29  #Call the display method.
30  plywood1.display()
31  lumber1.display()
32
```

4. Run the woodfactory.py program from the WWProject-L2 project.

a) In the **Project** pane, right-click **woodfactory.py** and select **Run 'woodfactory'**.
b) Observe the results shown in the **Run** pane.

```
Run:  woodfactory
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Dat
This is new Plywood
This is new Lumber

Process finished with exit code 0
```

The `plywood1` object displayed the message for a plywood product, and the `lumber1` object displayed the message for a lumber product.

5. Clean up the workspace.

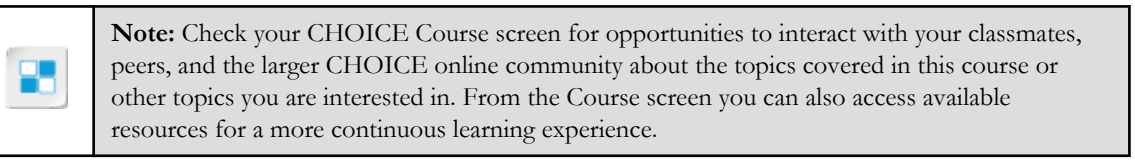
a) In the **Run** pane, close the **woodfactory** tab.
b) Select **File→Close Project**.
The project is closed.

Summary

In this lesson, you created classes and initialized objects of those classes. You then used comparison methods to compare specific attributes of objects and numeric methods to perform arithmetic computation on objects. Finally, you implemented a class factory to make object creation easier.

Which built-in methods do you think you'll use most often in your classes?

Do you think you might use a factory design pattern for your project? Why or why not?



Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

3 Creating a Desktop Application

Lesson Time: 2 hours

Lesson Introduction

Sometimes Python® evokes ideas of writing and running command-line code. However, Python has much more capability. Python is well suited to create robust desktop and web-connected apps that incorporate a graphical user interface (GUI) for data input and user interaction. You can create a GUI for your desktop app to provide users with an elegant interface for your app.

Lesson Objectives

In this lesson, you will:

- Design a graphical user interface for an application.
- Create an interactive app.

TOPIC A

Design a Graphical User Interface (GUI)

Graphical user interfaces don't come prefabricated; you must design and thoughtfully lay out the elements of your GUI to enable users of your app to interact with it efficiently and effectively. In this topic, you will design and lay out your GUI.

Graphical User Interfaces

There are many different ways to interact with software applications, through the command line, through voice, and, perhaps the most ubiquitous, through a ***graphical user interface (GUI)***. A GUI uses visual components (graphics) to allow you to interface with the software. Most modern operating systems provide a GUI as a way to interface with the operating system.

The Microsoft® Windows® operating system has a GUI that provides windows, a taskbar, icons, and menus as part of its GUI. Apple® offers a similar GUI for its operating system. Web applications, both websites and the applications you run on them, provide their own GUIs, often consisting of a menu at the top or along the side, clickable buttons, and other interactive elements that allow you to interface with the software.

Likewise, smart phones and tablets offer a streamlined, touch-based (GUI). Even smart watches and other devices provide a GUI to allow users to interact with the device and its software.

When you create a desktop, web, or mobile app, you must create a GUI to allow users to access, configure, and use the app.

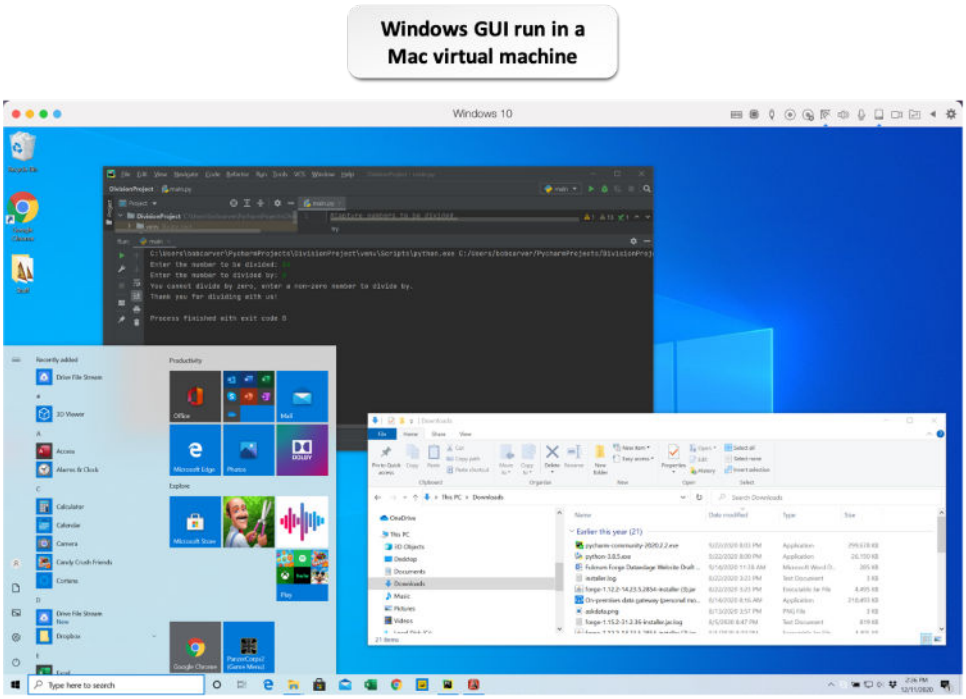


Figure 3-1: Graphical User Interfaces (GUIs).

GUI Libraries

The standard Python library includes the TkInter (TK interface) package which is the standard Python interface to the Tk GUI toolkit. This toolkit simplifies GUI programming in the Python

environment. Standard library packages help you create GUI widgets from your Python scripts and build your graphical programs. The following are some of the more popular Python GUI libraries.

Library	*Description*
TkInter	Uses the TK graphical library and is the default GUI library included as part of the standard Python suite.
PyGTK	Uses the GTK+ graphical library, used mainly in the GNOME desktop environment in Linux®.
PyQt	Uses the Qt graphical library, which is used mainly in the KDE desktop environment in Linux.
wxPython	Uses the wxWidgets graphical library, which is a multiplatform graphical environment.

It's a good idea to consider which library you'll use before you design an application's user interface, since the libraries vary somewhat.

TkInter Library

The `tkinter` library provides a standard class and methods for creating windows and widgets. Just import all of the items in the `tkinter` module, and instantiate the `Tk()` class:

```
from tkinter import *
window = Tk()
window.title("This is a test window")
window.geometry("300x100")
```

This code creates a window, and sets the title and size of the window, which is straightforward. However, before you can add widgets to the window, you must create a frame by defining a subclass of the `tkinter Frame` class:

```
class Application(Frame):
   def __init__(self, master):
      super(Application, self).__init__(master)
      self.create_widgets()

window = Tk()
window.title("Test Application Window")
window.geometry("300x100")
app = Application(window)
app.mainloop
```

This code template defines the class `Application` as a subclass of the `tkinter Frame` class. It then uses the empty `Tk()` instance as the parent class for the `app` instance.

After instantiating the new window, the `mainloop()` method is called. It enters into an endless loop, listening for window events, and passing them to event handler methods defined in the program.

Widgets

When you make the move to GUI programming, you need to learn a new set of terms. For example, the main area in a window is typically called the ***frame***. The frame contains all of the objects the program uses to interact with the user, and it is the central part in a GUI program. The frame is composed of objects called ***widgets*** (short for window gadgets) that display and retrieve information. Most graphical programming languages provide a library of widgets for you to use in your programs. Although not an official standard, a common set of widgets is available in just about every graphical programming environment.

Widget	Description
Button	Triggers an event in the window when selected.
Canvas	Draws shapes such as lines, polygons, etc, in the app.
Check button	Allows the user to select or deselect an individual item.
Entry	Provides an area to enter or display a single line of text.
Frame	A container you can use to organize widgets.
Label	Places static text in the window.
LabelFrame	A container widget that acts as a spacer or container in complex GUI layouts.
List box	Displays multiple values to select one or more items.
Menu	Displays a menu toolbar at the top of the window.
Menubutton	A button that displays menus in your app.
Message	A multiline text field for accepting user input.
PanedWindow	A container widget that can hold any number of panes which can be arranged horizontally or vertically.
Progressbar	Indicates that something is happening in the background.
Radiobutton	Allows the user to select one item from a group.
Scale	A slide widget.
Scrollbar	Controls the viewed items in a list box or frame.
Separator	Places a horizontal or vertical bar in the window.
Spinbox	Allows a user to select a value from a range of numbers.
Text	Provides a large area to enter or display multiple lines of text.
tkMessageBox	Used to display message boxes for user feedback in apps.
Toplevel	Creates a separate window container.

Each widget has its own set of properties that define how it appears in the window and how to handle any data or actions that occur while the user interacts with the widget. Before you start to code your user interface, it's a good idea to plan your layout on paper or in a ***wireframe*** drawing tool.

Window Geometry Managers and Methods

The key to a user-friendly GUI application is the placement of the widgets in the window area. Too many widgets grouped together can make a user interface confusing. The `tkinter` package provides three Geometry Managers which provide methods for positioning widgets in the GUI:

- **.pack() method**: packing widgets into available spaces in the window.
- **.place() method**: using positional values to place elements in the GUI.
- **.grid() method**: using a grid system.

The ***.pack method*** does what it says, it attempts to pack widgets into a window as best it can in the space available. When you choose this method, Python places the widgets in the window for you, starting at the top-left of the window and moving along to the next available space, either to the right or below the previous widget. The packing method works fine for small windows with just a few widgets, but if you have lots of widgets, your GUI can become cluttered and difficult to use.

The ***.place method*** requires that you define the precise location of each widget, using X and Y coordinates within the window. This provides the most accurate control over where the widgets appear, but it can be difficult to work with as you have to know the X and Y coordinates for the elements you're positioning and other elements already positioned.

The compromise between the packing and place methods is the ***.grid method***. The grid method creates a grid system in the window using rows and columns, somewhat like a spreadsheet. You place each widget in the window at a specific `row` and `column` location:

```
self.text1 = Label(self, text="This is some window text")
self.text1.grid(row=0, column=0)
```

You can also define a widget to span multiple rows or columns:

```
self.text1.grid(row=0, columnspan=3)
```

The grid method also defines the `sticky` property. This determines where in the cell the widget is positioned: `N` for top-aligned (north), `S` for bottom-aligned (south), `E` for right (east), and `W` for left (west).

The grid method provides some important benefits, such as:

- It forces the developer to consciously plan out the user interface, perhaps even using a pencil-and-paper approach to plan columns and rows where widgets should appear. For developers coming from visual layout environments like Xcode® or Visual Studio®, this approach seems familiar.
- It tends to adapt the layout well when the window size is changed by the user.

.grid Code Examples

For most simple layouts, the grid method provides overall control over where the widgets appear in the window, without making you determine exact window coordinates for each widget. Here are the rules for using the grid method:

- Define the grid positioning method in the `Application` subclass:

```
class Application(Frame):
   def __init__(self, master):
      super(Application, self).__init__(master)
      super.grid()
      super.create_widgets()
```

- Define the grid location for each widget as you create it:

```
def create_widgets(self):
   self.label1 = Label(self, text="Welcome to my window")
   self.label1.grid(row=0, column=0, sticky=W)
   self.button1 = Button(self, text="Click me!", command=self.display)
   self.button1.grid(row=1, column=0, sticky=W)
   self.output = Entry(self, width=10)
   self.output.grid(row=2, column=0, sticky=W)
```

UI Design Tools

While it is entirely possible to design and lay out your GUI using something like TkInter, the larger and more complex the app, the more difficult GUI design is. Several tools are available to help developer teams create functional and efficient user interfaces that meet design requirements and provide exceptional user experience. Some tools provide the ability to create wireframes to allow you to visualize what the GUI will look like, while other tools provide richer UI design and prototyping facilities, allowing designers and developers to build out the UI and test user experience prior to wiring the UI up to the code from the app. These tools help speed UI design, and help ensure good user experience for app users.

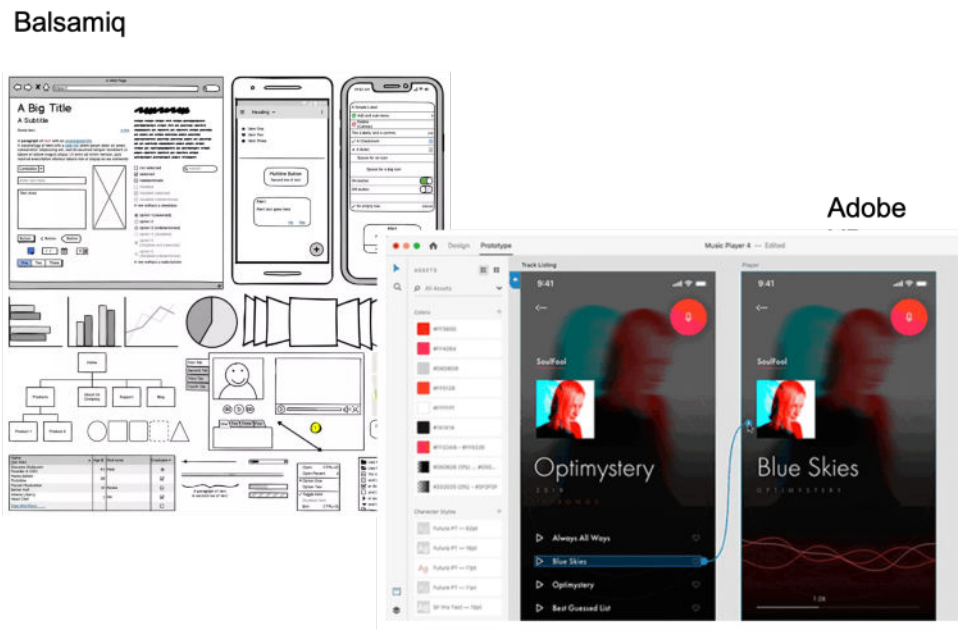


Figure 3-2: UI design tools.

Some of the most popular tools for wireframing are:

- MockFlow: **https://mockflow.com/**
- Balsamiq: **https://balsamiq.com/**
- Adobe® Comp: **https://www.adobe.com/uk/products/comp.html**

Some of the most popular tools for UI prototyping are:

- Axure®: **https://www.axure.com/**
- Sketch: **https://www.sketch.com/**
- InVision Studio: **https://www.invisionapp.com/studio**
- Proto.io: **https://proto.io/**
- Adobe XD: **https://www.adobe.com/uk/products/xd.html**
- Qt Designer: **https://build-system.fman.io/qt-designer-download**

Qt Designer

Qt, pronounced "cute" is a software development framework for creating and deploying connected embedded devices and apps. Qt provides APIs for many different aspects of app development including network connectivity, multimedia, and other function areas. One of the areas where Qt is widely used is in GUI design. One of the most popular Qt tools is Qt Designer.

Qt Designer allows designers and developers to rapidly build GUIs with widgets available in the Qt GUI framework. The designer offers an easy-to-use, drag-and-drop interface that allows you to lay out and organize elements like text fields, drop-down boxes, and buttons.

Qt Designer generates .ui files that are XML-based and store widgets in a tree, and that can be translated to languages like Python. Qt Designer is popular with Python programmers because, like Python, Qt Designer offers rapid prototyping in GUI design.

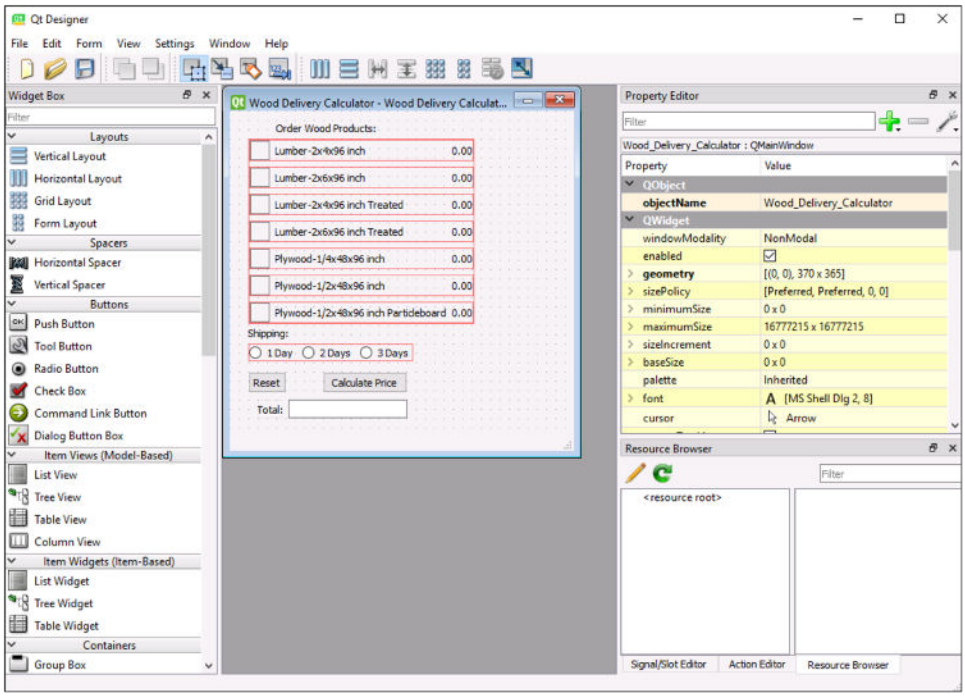


Figure 3-3: Qt Designer.

Guidelines for Designing a GUI

	Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Use the following guidelines to help when planning a graphical user interface (GUI) design.

Designing a GUI

When designing a graphical user interface (GUI):

- Identify the widgets you require:
  - Determine what data the application needs to collect from the user.
  - Determine the most appropriate widget to use for each data value the user must provide. (For example, use a radio button when the user should select a single object from a group of options.)
  - Determine what actions will trigger the processing of the input data, and what widgets will be involved. (For example, the user selects a **Calculate price** button.)
  - Determine what widgets will display the program results.
- Before you start development, make sure you select a GUI library that:
  - Provides a wide selection of widgets of the types you need to use.
  - Provides a look and feel that you and your users would like to use.
  - Provides licensing options that are acceptable for your purpose.
  - Is compatible with all of the operating systems and distribution methods you need to support for your finished application.

ACTIVITY 3-1
Designing a GUI

Scenario
You've been asked to design an application, the **Wood Delivery Calculator**, which helps Woodworkers Wheelhouse customers place orders for wood products to be delivered. To do that, you'll need to create a window with widgets to allow customers to specify the quantity of each wood product they want, and select the shipping method they want (1 Day, 2 Day, or 3 Days). Once customers select their options, they'll be able to view the total cost for the order.

The user interface will include some means to do the following:

- Choose products (enter the quantity to purchase):
  - Lumber-2x4x96 inch
  - Lumber-2x6x96 inch
  - Lumber-2x4x96 inch Treated
  - Lumber-2x6x96 inch Treated
  - Plywood-1/4x48x96 inch
  - Plywood-1/2x48x96 inch
  - Plywood-1/2x48x96 inch Particleboard
- Choose a shipping method (just one):
  - 1 Day
  - 2 Days
  - 3 Days
- Clear all selections (reset).
- Display the total price.

---

1. Consider the following questions as you plan the layout for the **Wood Delivery Calculator**.

2. **How many windows will the application use?**

3. **What type of widget(s) might you provide to enable the user to specify the number of each product to purchase?**

4. **What type of widget(s) might you provide to enable the user to select a shipping method?**

5. **What type of widget(s) might you provide to enable the user to clear (reset) the selections?**

6. **What type of widget(s) might you provide to enable the user to calculate the total?**

7. Create a wireframe to depict the layout.
   a) In your course manual's margins or in a drawing application, create a wireframe sketch of the **Wood Delivery Calculator** user interface.

ACTIVITY 3-2
Controlling the Layout of Widgets

Before You Begin
PyCharm is open.

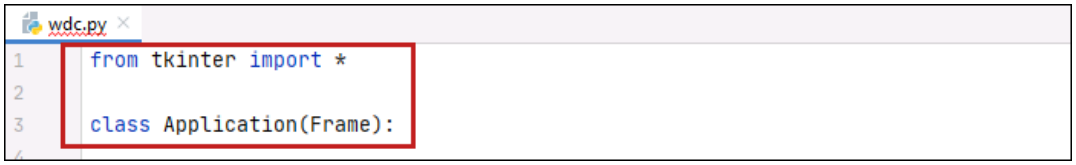
Scenario
Woodworkers Wheelhouse has a delivery service for lumber materials and wants an application that allows local customers to place orders that will be delivered. Customers should be able to select the number of each product they want and how quickly they are delivered. You have settled on a layout for the **Wood Delivery Calculator** window. You will start by creating a new project folder. Then, you will develop the layout using the `tkinter` library.

---

1. Create a new project.
   a) In the Welcome to PyCharm window, select **New Project**.
   b) In the **Location** box, select the **Browse** button.
   c) Navigate to the **C:\094022Data\Creating a Desktop Application** folder.
   d) On the toolbar, select the **New Folder** button.
   e) In the **Enter a new folder name** box, type ***WWProject-L3***, and select **OK**.
   f) Select **OK** again to select the new folder.
   g) In the **New Project** window, verify that **Existing interpreter** is selected.
   h) Select **Create** to create the project.

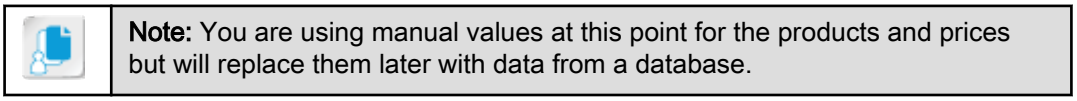
2. Create the **wdc.py** file in the **WWProject-L3** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L3** project, and select **New→Python File**.
   b) In the **Name** box, type ***wdc*** and press **Enter**.

3. Define the `Application` subclass.
   a) At the top of the open **wdc.py** file, starting on line 1, type the following to import the methods from the `tkinter` library, and define the `Application` subclass of the `Frame` class.



b) After the class definition, starting on line 5, type the following to create the constructor.

```
3   class Application(Frame):
4
5       def __init__(self, master):
6           super(Application, self).__init__(master)
7           self.grid()
8           self.products = ["Lumber-2x4x96 inch",
9                            "Lumber-2x6x96 inch",
10                           "Lumber-2x4x96 inch Treated",
11                           "Lumber-2x6x96 inch Treated",
12                           "Plywood-1/4x48x96 inch",
13                           "Plywood-1/2x48x96 inch",
14                           "Plywood-1/2x48x96 inch Particleboard"]
15          self.prices = [5.50,
16                         8.50,
17                         8.75,
18                         13.25,
19                         33.00,
20                         37.25,
21                         19.25]
22          self.shipdays = {"1 Day": 20.00,
23                           "2 Days": 15.00,
24                           "3 Days": 10.00}
25          self.lblproduct = list(self.products)
26          self.lblprice = list(self.prices)
27          self.entry = list(self.products)
28          self.create_widgets()
29
```



Note: You are using manual values at this point for the products and prices but will replace them later with data from a database.

The constructor defines a list object that contains the wood products, prices, shipdays dictionary, and list objects for the `lblproduct`, `lblprice`, and the `entry` variables.

4. Define the application widgets.

a) After the constructor, starting on line 30, type the following to add the `create_widgets()` method and the first label by entering the following code.

```
27          self.entry = list(self.products)
28          self.create_widgets()
29
30      def create_widgets(self):
31          line = 1
32          self.label2 = Label(self, text='Order Wood Products:')
33          self.label2.grid(row=line, column=1, sticky=W)
34
```

b) Starting on line 35, type the following to create a new label to identify the wood products list, and the code to iterate through the products list to create the products entry boxes.

```
30      def create_widgets(self):
31          line = 1
32          self.label2 = Label(self, text='Order Wood Products:')
33          self.label2.grid(row=line, column=1, sticky=W)
34
35          for i in range(len(self.products)):
36              line += 1
37              self.lblproduct[i] = Label(self, text=self.products[i])
38              self.lblprice[i] = Label(self,
39                                       text="{0:.2f}".format(self.prices[i]))
40              self.entry[i] = Entry(self, width=3)
41              self.entry[i].grid(row=line, column=0)
42              self.lblproduct[i].grid(row=line, column=1, sticky=W)
43              self.lblprice[i].grid(row=line, column=2, sticky=W)
44
```

A separate variable is used to keep track of the row numbers dynamically in the loop.

c) Starting on line 45, type the following to define the three radio buttons used to select the delivery days.

```
42              self.lblproduct[i].grid(row=line, column=1, sticky=W)
43              self.lblprice[i].grid(row=line, column=2, sticky=W)
44
45          self.label1 = Label(self, text="Shipping:")
46          line += 1
47          self.label1.grid(row=line, column=0, sticky=W)
48
49          self.ship = StringVar()
50          line += 1
51
52          self.radio1 = Radiobutton(self,
53                                    text="1 Day",
54                                    variable=self.ship,
55                                    value="1 Day")
56          self.radio1.grid(row=line, column=0, sticky=W)
57          self.radio1.select()
58
59          self.radio2 = Radiobutton(self,
60                                    text="2 Days",
61                                    variable=self.ship,
62                                    value="2 Days")
63          self.radio2.grid(row=line, column=1)
64
65          self.radio3 = Radiobutton(self,
66                                    text="3 Days",
67                                    variable=self.ship,
68                                    value="3 Days")
69          self.radio3.grid(row=line, column=2, sticky=W)
70
```

All of the buttons will be on the same line, and will be tied to the same variable. The `Radiobutton select()` method is called to make "1 Day" the default delivery selection.

d) After the delivery code, starting on line 71, type the following to create the two buttons to reset the form and calculate the order price.

```
                                    value="3 Days")
        self.radio3.grid(row=line, column=2, sticky=W)

        line += 1
        self.button1 = Button(self, text="Reset")
        self.button1.grid(row=line, column=0)
        self.button2 = Button(self, text="Calculate Price")
        self.button2.grid(row=line, column=1)

```

e) After the buttons, starting on line 77, type the following to create an empty line in the grid layout by creating a `Label` with no text.

```
        self.button2 = Button(self, text="Calculate Price")
        self.button2.grid(row=line, column=1)

        line += 1
        self.label3 = Label(self, text="")
        self.label3.grid(row=line, column=0)

```

f) Finally, after the empty line, starting on line 81, type the following to set the **Total** label, and an `Entry` box to display the total price.

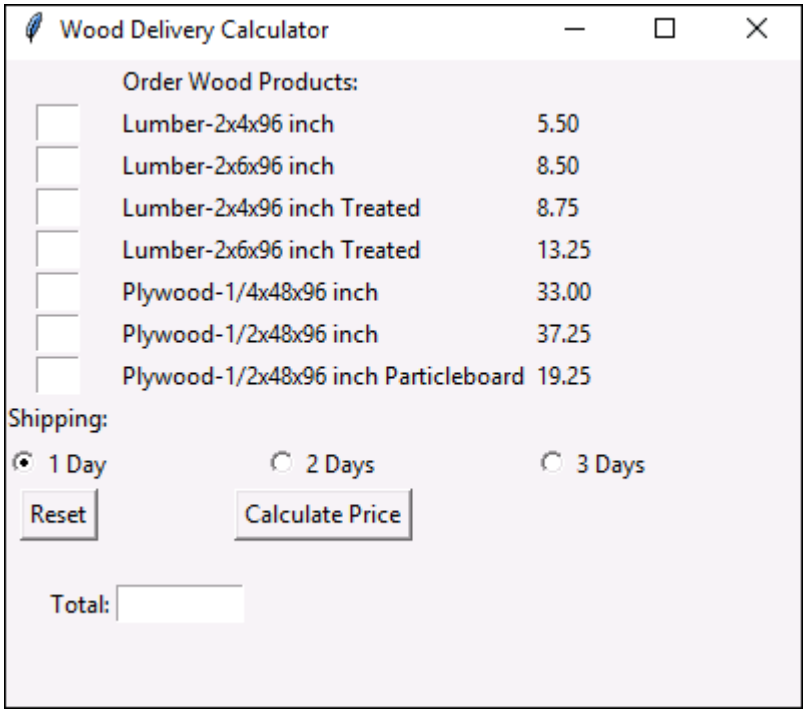
```
        self.label3 = Label(self, text="")
        self.label3.grid(row=line, column=0)

        line += 1
        self.label4 = Label(self, text="Total:")
        self.label4.grid(row=line, column=0, sticky=E)
        self.result = Entry(self, width=10)
        self.result.grid(row=line, column=1, sticky=W)

```

5. Now that you've completed the `Application` subclass, instantiate the `Tk()` class. Starting on line 87, type the following to create the `Application` subclass instance, and start the main loop.

```
        self.result = Entry(self, width=10)
        self.result.grid(row=line, column=1, sticky=W)

window = Tk()
window.title("Wood Delivery Calculator")
window.geometry('400x325')
app = Application(window)
app.mainloop()

```

Note: You may have to experiment with the width and height settings depending on your desktop default fonts.

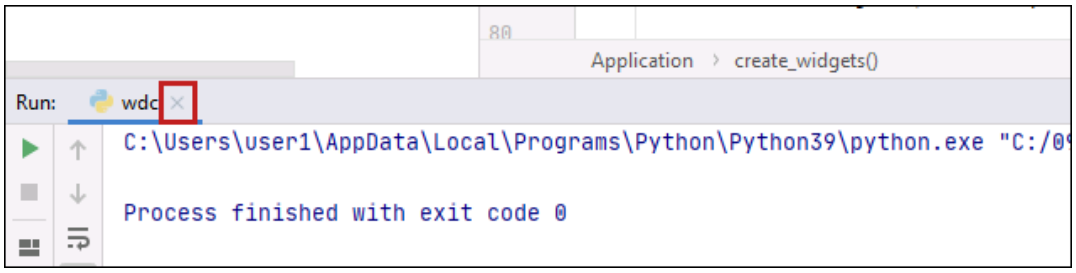
6. Run the **Wood Delivery Calculator** application.
   a) Run **wdc.py**.



   b) Enter a number into the entry box for each wood product to make sure they all work.
   c) Select the shipping radio buttons to make sure you can only make one selection.

   For now, nothing will happen when you select the **Reset** or **Calculate Price** buttons, as you haven't defined any event handlers.

   a) In the top-right corner of the window, select the **X** to close the window and end the program.
   b) In the **Run** pane, close the **wdc** tab.



TOPIC B

Create Interactive Applications

Once your GUI has been designed and laid out, you must wire it up to the code behind the GUI in order for your app to function. In this topic, you will create an interactive app by doing just that.

Events

Programming for a GUI environment is different than programming for a command-line environment. In a command-line, executed app, the order in which commands are input controls the sequence of execution. For example, the program has code that prompts the user for input, then processes the input, and then displays the results on the command line based on the input provided. The user can only respond to the input requests as they are presented by the program.

In contrast, a GUI program displays an entire set of interaction widgets all at once, all in the same window. The user decides which widget to engage with, and, therefore, what gets processed when. Because the program doesn't control which widget is activated at any given time, this process is called event-driven programming. In event-driven programming, Python calls different methods within the program, based on which ***event*** (or action) occurs in the GUI window. There isn't a set flow to the program code; it's just a bunch of methods that individually run in response to an event.

For example, your user can enter data into a text widget, but nothing happens until the user presses a button in the program window to submit the text. The button triggers an event, and your program code must detect that event and then run the code method to read the text in the text widget and process it.

Event Handlers

In event-driven programming, widgets in the GUI must be linked to events, and events must be linked to methods in the code. These methods are called ***event handlers***. You must create separate event handler methods for each event that can be generated from the widgets in the window. The event handlers do the bulk of the work in GUI programs. They retrieve the data from the widgets, process the data, and then display the results in a window using other widgets. This might seem a bit cumbersome at first, but once you get used to coding with event handlers, you'll see just how easy it is to work in a GUI environment.

Event handlers are class methods created in the `Application` subclass that handle events generated by widgets. You must define the event handler name inside the widget using the `command` parameter:

```
self.button1 = Button(self, text="Submit", command=self.display)
```

The actual event handler is a separate method that you must create inside the `Application` subclass:

```
def display(self):
    print("The button was clicked")
```

It's imperative that you match the command name defined in the widget to the method name, including the text case.

Widget Interaction: Respond to Button Click

Follow these guidelines for interacting with various types of widgets.

When you create a `Button` widget, use the `command` parameter to define the event handler method that Python will call when the program user selects the button:

```
self.button1 = Button(self, text="submit", command=self.display)
```

Widget Interaction: Entry Widgets

To retrieve the value entered into an `Entry` widget, use the `get()` method on the `Entry` widget object. You can also display data from your program in an `Entry` widget by using the `insert()` method, or clear out any text already in the `Entry` widget by using the `delete()` method. The `delete()` method uses two parameters—the starting character position (with the first character at position 0), and the end position (you can use the special keyword `END`). The `insert()` method also uses two parameters—the starting location of the inserted text, and the text to insert into the widget:

```
def convert(self):
    varText = self.text1.get()
    varReplaced = varText.upper()
    self.text1.delete(0, END)
    self.text1.insert(END, varReplaced)
```

Widget Interaction: Radiobuttons

`Radiobutton` widgets must use a control variable to pass the radio button state. Python defines four types of control variables:

- `BooleanVar()`—for Boolean `True` and `False` values.
- `DoubleVar()`—for floating point values.
- `IntVar()`—for integer values.
- `StringVar()`—for text values.

Group radio buttons together by using the same control variable for each radio button in the group. To retrieve the value of the selected radio button from the group, use the `get()` method for the control variable. The `get()` method retrieves the `value` parameter specified for the selected `Radiobutton` object:

```
def create_widgets(self):
    self.var1 = StringVar()
    self.radio1 = Radiobutton(self, text="Option 1", variable=self.var1,
value="1")
    self.radio2 = Radiobutton(self, text="Option 2", variable=self.var1,
value="2")
    self.radio3 = Radiobutton(self, text="Option 3", variable=self.var1,
value="3")
def process(self):
    selection = self.var1.get()
    if (selection == "1"):
        print("Option 1 was selected")
```

Widget Interaction: Checkbuttons

The `Checkbutton` widget also uses a control variable, but it doesn't return a value, it just returns a Boolean `True` value if the check box is selected, or a `False` value if it's not selected. Thus, you use the `BooleanVar()` control variable type for the `Checkbutton` widget. Use the `get()` method for the control variable to determine the state of the check box at the time of an event trigger:

```
def create_widgets(self):
    for i in range(len(self.options)):
        self.checkboxVar[i] = BooleanVar()
        self.checkbox[i] = Checkbutton(self, text=self.options[i],
variable=self.checkboxVar[i])

def process(self):
    for box in range(len(self.options)):
```

```
    if (self.checkboxvar[box].get()):
        print("%s was selected".format(self.options[box]))
```

Widget Interaction: Text

As with the `Entry` widget, you retrieve text from the `Text` widget using the `get()` method, remove text using the `delete()` method, and add text using the `insert()` method. However, there's a bit of a twist to these methods in the `Text` widget. Because the widget works with multiple lines of text, the index value you specify is not a single value. Instead, it's a text value that has two parts: "x.y". The x is the row location (starting at 1), and y is the column location (starting at 0). So, to reference the first character in the `Text` widget, you use the index value of "1.0":

```
def create_widgets(self):
    self.text1 = Text(self, width=20, height=10)

def process(self):
    varText = self.text1.get("1.0", END)
    varReplaced = varText.upper()
    self.text1.delete("1.0", END)
    self.text1.insert(END, varReplaced)
```

Menu Interaction

A staple of GUI programs is the menu bar at the top of the window. The menu bar provides drop-down menus so program users can quickly make selections. You use the `Menu` widget to create menu bars in your `tkinter` programs.

First, define a main `Menu` instance. After that, you can add main menu bar top-level entries by creating subclass instances of the main `Menu` instance. For menu entries that contain submenus, use the `add_cascade()` method to add the entry. For menu entries that point directly to event handlers, use the `add_command()` method.

```
menubar = Menu(self)
filemenu = Menu(menubar)
filemenu.add_command(label="New", command=self.new)
filemenu.add_command(label="Open", command=self.open)
filemenu.add_command(label="Quit", command=self.quit)
menubar.add_cascade(label="File", menu=filemenu)
menubar.add_command(label="Help", command=self.help)
window.config(menu=menubar)
```

.Bind()

You can use `.bind()` to call an event handler whenever a defined event occurs. The event handler can be said to be bound to the event because it's called each time the specified event occurs.

For example, you can bind mouse clicks in your GUI to functions. The following code shows where on the TkInter canvas the corresponding button click took place.

```
#creates tkinter window or root
window base = Tk ()
base . geometry ( '300x150' )

#Function to be called when the mouse scroll button is pressed
def mouse_scroll ( label ):
    print ( 'Mouse scroll button click at x = % d, y = % d' %( label . x ,
label . y ))
```

```
#Function to be called when right mouse is pressed
def  mouse_rt_click ( label ):
    print ( 'Mouse right button click at x = % d, y = % d' %( label . x ,
label . y ))

#Function to be called when the left mouse button is pressed twice
def left_double_click ( label ):
    print ( 'Mouse left button double click at x = % d, y = % d' %( label .
x ,  label . y ))

Function = Frame(base, height = 100, width  = 200)
Function.bind ('<Button-2>', mouse_scroll )
Function.bind ('<Button-3>', mouse_rt_click )
Function.bind ( '<Double 1>', left_double_click)
Function.pack()
mainloop ()
```

Tabs with TkInter

Tabs in windows that are part of a GUI are a popular, modern navigation element that allow users to access different, but related information and configuration options. They allow for segmented access and making it easier to manage complex objects and processes. With a tabbed window or interface, a user can select the specific element desired and see the information and configuration related to that element.

The following code creates a window in TkInter with two tabs:

```
import tkinter as tk  #import TkInter module.
from tkinter import ttk  #import TkInter ttk module which includes the
Notebook widget.

root = tk.Tk()  #Create the root, parent window and instantiates the Tk class
without any parameters at this point.

root.title("Tabbed Window") #Gives the parent widget (window) a title)

tabControl = ttk.Notebook(root) #Creates the tab control, you can optionally
add height, padding, and width.

first_tab = ttk.Frame(tabControl) #Create the first tab.
second_tab = ttk.Frame(tabControl) #Create the second tab.

tabControl.add(first_tab, text ='First Tab')  #Adds the first tab.
tabControl.add(second_tab, text ='Second Tab')  #Adds the second tab.
tabControl.pack(expand = 1, fill ="both")  #Pack the tab control to make tabs
visible.

#Create the label widget for the First Tab as a child of the parent (root).
ttk.Label(first_tab,
    text ="Welcome to \
    the First Tab").grid(column = 0,
                         row = 0,
                         padx = 30,
                         pady = 30)

#Create the label widget for the Second Tab as a child of the parent (root).
ttk.Label(second_tab,
```

```
    text ="Welcome to\
    the Second Tab").grid(column = 0,
                          row = 0,
                          padx = 30,
                          pady = 30)

root.mainloop()  #Run the main app.
```

Additional Information

For additional information on ttk generally, see: **https://docs.python.org/3/library/tkinter.ttk.html**.

For additional information on the Notebook widget, see: **https://docs.python.org/3/library/tkinter.ttk.html#notebook**.

Built-in Methods for Closing Windows

Python provides two built-in methods for managing your TkInter app and the windows of your TkInter app. The following table lists some of these commonly used built-in methods.

Method	*Description*
`root.destroy()`	Causes TkInter to exit the `mainloop()` and destroys all the widgets in the `mainloop()`. The window and its associated widgets will disappear. `destroy()` can also be used to remove individual widgets as well.
`root.quit()`	Generally considered safer for managing the closure of TkInter windows and is often regarded as better. The `quit()` function exits the `mainloop()`, stops the TCP interpreter, but does not destroy any existing widgets or the window.
	 **Note:** The `quit()` function should not be used if you call your app from IDLE as IDLE is a TkInter app; so, if you call the `quit()` function in your app, the TCL interpreter and IDLE will be terminated.

Guidelines for Creating Interactive Applications

Follow these guidelines when creating interactive applications.

Create Interactive Applications

When creating interactive applications:

- Create event handlers to have your app respond to clicks in your GUI.
- Use `insert()`, `get()`, and `delete()` when working with entry and text widgets.
- Use Radiobutton and Checkbutton widgets to make object configuration easier in your GUI.

ACTIVITY 3-3
Creating an Interactive Application

Before You Begin
The **WWProject-L3** project is open in PyCharm.

Scenario
You have already laid out the controls for the **Wood Delivery Calculator**, but at this point, they are not yet functional. You will add an event handler to calculate the cost of the order based on the selections the program user makes from the product and shipping widgets. The order cost will be based on the quantity entered for each product multiplied by the cost of the product, and a flat amount added for the shipping option selected. A list object that holds the product prices for the calculations has already been added to the project.

---

1. Examine the prices list object.
   a) In **wdc.py**, starting on line 15, examine the `self.prices` list object.

```
8         self.products = ["Lumber-2x4x96 inch",
9                          "Lumber-2x6x96 inch",
10                         "Lumber-2x4x96 inch Treated",
11                         "Lumber-2x6x96 inch Treated",
12                         "Plywood-1/4x48x96 inch",
13                         "Plywood-1/2x48x96 inch",
14                         "Plywood-1/2x48x96 inch Particleboard"]
15        self.prices = [5.50,
16                       8.50,
17                       8.75,
18                       13.25,
19                       33.00,
20                       37.25,
21                       19.25]
22        self.shipdays = {"1 Day": 20.00,
```

2. Define the event handler methods in the `Button` widgets.
   a) On lines 72 and 74, type the following to add `command` parameters to the **Reset** and **Calculate Price** button widgets to point to the methods that will handle their button clicks.

```
68                                  value="3 Days")
69        self.radio3.grid(row=line, column=2, sticky=W)
70
71        line += 1
72        self.button1 = Button(self, text="Reset", command=self.reset)
73        self.button1.grid(row=line, column=0)
74        self.button2 = Button(self, text="Calculate Price", command=self.calculate)
75        self.button2.grid(row=line, column=1)
76
```

3. Define the `reset()` method to clear the total field form, select the 1 Day shipping, and clear all of the `entry` widgets.

   a) After the `create_widgets()` method, but before the code to instantiate the window, starting on line 87, type the following to insert a new `reset()` method, as shown.

```
84          self.result = Entry(self, width=10)
85          self.result.grid(row=line, column=1, sticky=W)
86
87      def reset(self):
88          """Reset the Radiobuttons and Entry boxes"""
89          self.radio1.select()
90          for i in range(len(self.products)):
91              self.entry[i].delete(0,END)
92          self.result.delete(0,END)
93
94  window = Tk()
95  window.title("Wood Delivery Calculator")
```

   This method selects the 1 Day shipping, clears each of the product `Entry` widgets, and deletes the existing text in the totals `Entry` widget.

4. Define the `calculate()` method to calculate the total order cost.

a) After the `reset()` method, but before the code to instantiate the window, starting on line 94, type the following to add a new `calculate` method.

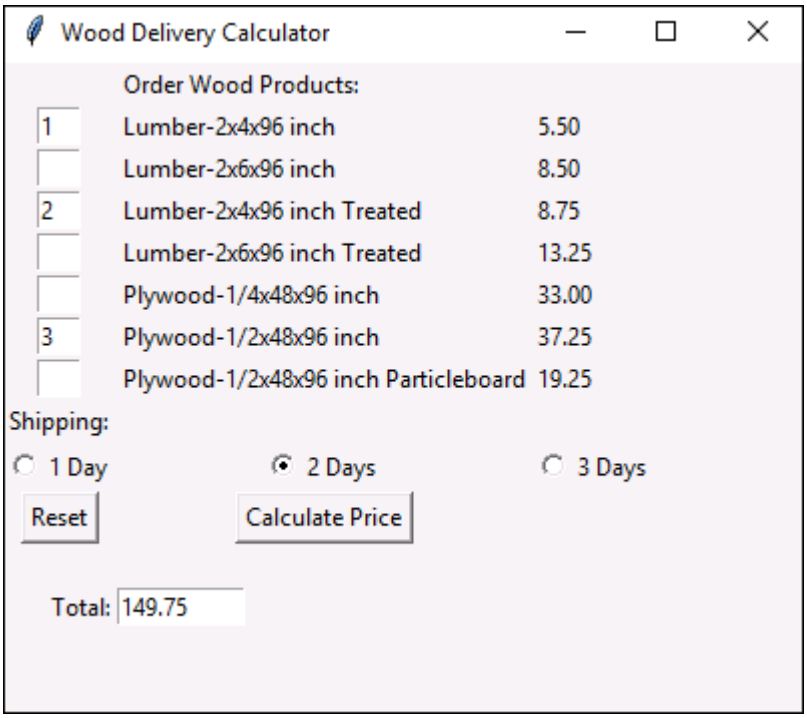
```
92          self.result.delete(0,END)
93
94      def calculate(self):
95          """Event handler for the calculator"""
96          self.total = 0
97          for i in range(len(self.products)):
98              if (self.entry[i].get()):
99                  self.total += (self.prices[i] * int(self.entry[i].get()))
100
101         if (self.ship.get() == "1 Day"):
102             self.total += self.shipdays["1 Day"]
103         elif (self.ship.get() == "2 Days"):
104             self.total += self.shipdays["2 Days"]
105         elif (self.ship.get() == "3 Days"):
106             self.total += self.shipdays["3 Days"]
107
108         strPrice = "{0:.2f}".format(self.total)
109         self.result.delete(0, END)
110         self.result.insert(END, strPrice)
111
112 window = Tk()
```

- Lines 96 through 99 determine which products have a quantity ordered by iterating through all of the `entry` variables in the list, using the `get()` method to determine if they have a value or not. If they do, then the quantity is multiplied by the price and added to `self.total`.
- Lines 101 through 106 determine which shipping method was selected, and adds the shipping cost to the total price of the order, using the `get()` method to check the value returned by the `Radiobutton` group.
- Lines 108 through 110 display the total price, rounding the floating point value to two decimal places, and converting it into a string value to display in the `Entry` widget.

5. Run the updated **Wood Delivery Calculator** application.

a) Run **wdc.py**.

b) Enter a value from **1** to **9** for two or three of the products and select a shipping type. Select **Calculate Price**, and the total cost should appear in the **Total** field.



c) Select the **Reset** button to ensure the form fields are reset.

6. Clean up the workspace.

a) Close the Wood Delivery Calculator window.
b) In the **Run** pane, close the **wdc** tab.

ACTIVITY 3-4

Adding Menu Interaction to an Application

Before You Begin

The **WWProject-L3** project is open in PyCharm.

Scenario

You know some users like to use menu bar commands in applications. You decide to add a menu that has the **Reset** and **Calculate Price** commands available. You will also add the **Quit** command to give users another way to close the application window.

---

1. Define the menu used for the application.
   a) At the end of the `create_widgets()` method and before the `reset()` method, starting on line 87, type the following to add a menu bar that has two top-level entries: **File** and **Quit**.

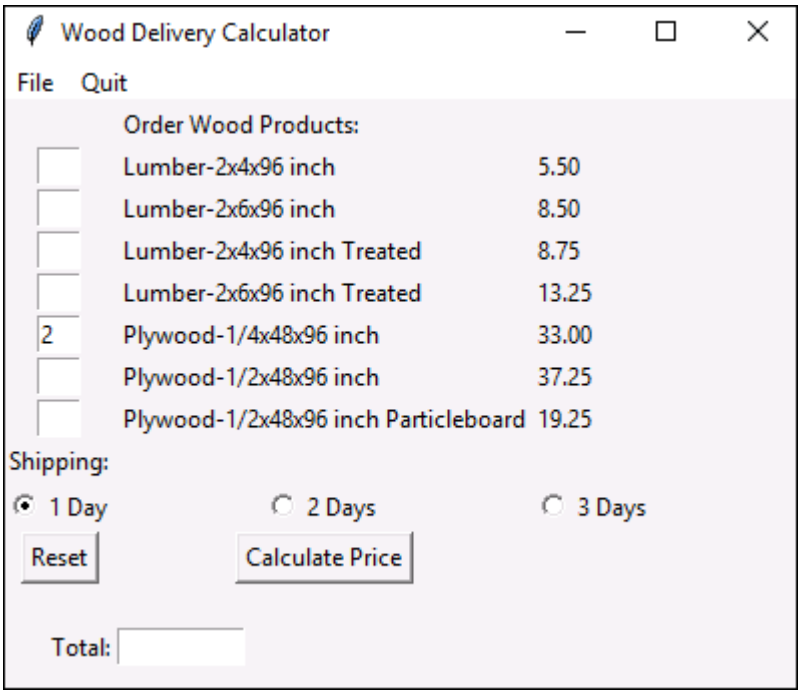
```
82          self.label4 = Label(self, text="Total:")
83          self.label4.grid(row=line, column=0, sticky=E)
84          self.result = Entry(self, width=10)
85          self.result.grid(row=line, column=1, sticky=W)
86
87          menubar = Menu(self)
88          filemenu = Menu(menubar)
89          menubar.add_cascade(label="File", menu=filemenu)
90          menubar.add_command(label="Quit", command=window.quit)
91
92      def reset(self):
93          """Reset the Radiobuttons and Entry boxes"""
```

   b) Starting on line 91, type the following to add sub-entries named **Reset** and **Calculate Price** to the **File** menu.

```
87          menubar = Menu(self)
88          filemenu = Menu(menubar)
89          menubar.add_cascade(label="File", menu=filemenu)
90          menubar.add_command(label="Quit", command=window.quit)
91          filemenu.add_command(label="Reset", command=self.reset)
92          filemenu.add_command(label="Calculate Price", command=self.calculate)
93          window.config(menu=menubar)
94
95      def reset(self):
```

2. Run the **Wood Delivery Calculator** application.
   a) Run **wdc.py**.

b) Specify the number of products and shipping method to match the image below.



c) Select **File→Calculate Price** from the application menu bar.
The price is calculated the same as if you had selected the **Calculate Price** button.

d) Select **File→Reset**.
The form fields are reset the same as if you had selected the **Reset** button.

e) Select **Quit** to close the **Wood Delivery Calculator** window.

f) Select **File→Close Project**.
The project is closed.

Summary

In this lesson, you designed and laid out a GUI for your app working with TkInter and Qt Designer to build out the interface for your app. You then wired up the GUI, connecting methods to events to create interactive widgets to allow your app to function as required.

What steps will you take to design a GUI interface for your application?

What types of event handlers do you anticipate using in your applications?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

4 Creating Data-Driven Applications

Lesson Time: 2 hours, 30 minutes

Lesson Introduction

Many web and desktop apps process data. As a developer, you have to connect to that data, bring it into your app, and process it as defined by the requirements of your app. In this lesson, you will connect to data, then store, update, and delete data in a database.

Lesson Objectives

In this lesson, you will:

- Connect to data in Python.
- Manage data in a database.

TOPIC A

Connect to Data

To process data, your app first has to connect to that data. In this topic, you will connect to data so that your app can work with it.

Apps and Data

While some apps take user input, process the data entered, and return results, many apps access and use external data. Apps may look up data; for example, the car app we discussed earlier may pull data about features of cars and capabilities of engines from a data source. Other apps may both look up and collect data. A web-based, e-commerce app would display products and information about them from a data source, and collect purchase information and payment data from buyers.

If your app must look or store information, it must be connected to a data source of some sort. Apps can get data from many different types of data sources including:

- Files: Including formatted text files (such as comma delimited CSV files), spreadsheets, and XML files.
- Databases: Which offer structure and relationship capabilities for storing, accessing, and processing large amounts of data. Databases might be located locally on the same machine as the app, on a different computer, or in the cloud.

File-Based Data Source

In some cases, it makes sense to save data in a local file. This provides the benefit of keeping data in a form that is easy to copy and share (by sharing the file that contains the data). While Python® can both read and write to files, file-based storage is commonly used for apps that are only looking up small amounts of simple data.

Python has built-in functions for handling all of your file storage needs. Before you can do any reading or writing of data in a file, your program must first open the file. You use the `open()` function to open a file:

```
myfile = open('textfile.txt', 'mode')
```

The first parameter specifies the file to open, and the second parameter specifies the mode to open it in. The mode defines the privileges the program has for interacting with the data file. There are four different modes.

Mode	*Description*
r	Read-only access.
w	Write-only access. The file specified is replaced if it already exists.
r+	Read and write access.
a	Write only, but append new data to the end of an existing file.

The `open()` function returns a special value called a ***file pointer***. After you open the file, you can use the file pointer to reference the opened file in any read or write operations.

To write new data to the file, use the `write()` method:

```
myfile.write("This is a test")
```

Python writes the specified data directly to the file at the location of the file pointer.

Reading data from a file is a little trickier. There are two ways to read data from a flat file. The `read()` method allows you to specify how many characters to read from the file:

```
test = myfile.read(4)
```

This returns the first four characters in the file. Working your way through a file character-by-character can be somewhat challenging. To make life easier, Python also includes the `readline()` function. The `readline()` function reads the file one line at a time:

```
test = myfile.readline()
```

This is an excellent way to process text files.

When the file pointer gets to the end of the file, both the `read()` and `readline()` functions return a `NULL` value, letting you know that there's no more data.

When you're done using the file, you should close the file pointer by using the `close()` function:

```
myfile.close()
```

Relational Databases

The downside to simple file-based storage is that it's hard to search for specific data. You have to read each line one-by-one and check the data to find what you're looking for. If your app needs to search for a lot of data, file-based storage is inefficient. This is where databases come in, they make storage and retrieval of data reliable and fast.

A database stores data in a file, but uses a separate database engine program that interfaces with the data file to do all the hard work of reading, writing, and looking for data for you. All your program needs to do is tell the database what you're looking for, and the database does the rest.

The key to using a database is the ability to submit simple commands to the database engine, and receive the results which are then used by the app for processing or to display answers to queries. Most modern databases use Structured Query Language (SQL) for querying the database. SQL defines simple text commands that clients can submit to a database server to create databases and tables, insert data, and query data values.

Python supports connectivity and interaction with many different databases. The following are some of the more popular ones.

Database	*Description*
Microsoft® SQL Server®	The commercial Microsoft database server that supports medium-to-large database applications.
Oracle®	A commercial database commonly used in Unix® environments.
MySQL™	A popular open source database commonly used in websites.
PostgreSQL®	A popular open source database that incorporates advanced database features.

Cloud Databases

Cloud databases offer the same core capabilities as other databases in terms of how data is stored and retrieved by Python, but offer many advantages in terms of deployment, management, maintenance, scalability and security because the hardware underlying the database is provided and managed by the cloud service provider. The core advantages of cloud databases are:

- **Infrastructure deployment and maintenance**. Cloud service providers allow you to deploy servers and hardware with the click of a button and manage maintenance of hardware.
- **Scalability**. You can scale your solution with the click of a button, adding processors, memory, storage, and scaling out to multiple global regions.

- **Security**. Cloud service providers have expert staff securing their networks, hardware, virtual machines, and hardware. In addition, they have dedicated expert staff monitoring for attacks.
- **Latest technology**. Cloud service providers offer the latest technology and make it easy for customers to keep their products up to date.
- **Reducing capital expenses**. Buying from cloud service providers allows you to buy what you need when you need it, and pay for what you use on an operational basis rather than making capital investments in hardware, network, and security up front.

The following are some of the most popular cloud databases.

Cloud Database	*Description*
Amazon Web Service	• Amazon Relational Database Service (RDS) that runs on Oracle, SQL, or MySQL instances. • Amazon SimpleDB which is a schema-less database intended for small workloads. • Amazon DynamoDB which is a NoSQL database designed for high availability across Amazon cloud zones.
Oracle Database	Provides enterprise-scale, cloud-based database infrastructure with robust governance and security controls.
Microsoft® Azure®	Provides a comprehensive, cloud-based platform for creating and running web-based, data-backed applications on the biggest global cloud infrastructure.
Google Cloud Platform™	Offers a wide range of data services to large and small businesses.
IBM® DB2®	Offers robust data management and analytics for transactional data and data warehouse workloads.
MongoDB® Atlas	A popular NoSQL database that offers robust scaling and automation.

As you can see, some cloud database options simply provide common relational databases like MySQL, which operates on the cloud service provider's infrastructure.

MySQL Databases

MySQL is touted as the world's most popular open source relational database. It is owned by Oracle corporate but the Community Edition is available free under the GNU General Public License. Many providers, including Oracle, offer proprietary versions hosted on their platforms for a fee. Most major cloud service providers including AWS, Microsoft Azure, and Google Cloud offer MySQL databases as part of their service offerings. It is popular and widely used by small businesses and large enterprises for web apps hosted on WordPress, Drupal®, Joomla!®, Facebook, and Twitter, and is a popular choice as a database for Python apps.

With such a large install base and an active open source community, MySQL provides the following benefits:

- It's very secure with built-in data security and support for transactional processing, which makes it a safe choice for e-commerce apps.
- It offers high performance and can be optimized for different web, app, and data access scenarios.
- It is highly scalable, allowing you to easily add capacity to your apps.
- Providers offer robust support and strong reliability through hardened and optimized database and server configurations.

Additional Information

For more information, see: **https://www.mysql.com/**.

MySQL Community Edition download: **https://www.mysql.com/products/community/**.

MySQL Connector

To use a MySQL database as a data source for a Python app, you must install the MySQL Connector for Python, which enables Python apps to access a MySQL database using an API that is compliant with the Python Database API Specification v2.0 (PEP 249). The connector includes support for nearly all MySQL features through the latest version, converting parameter values back and forth between MySQL and Python as necessary, secure network connections, data compression, and other features.

When using the MySQL Connector to connect to a MySQL database from Python, you must provide the following arguments:

- **Host Name**: The name of the host, server, or the IP address of the host or server where MySQL is running. This can also be your local machine (localhost or 127.0.0.0).
- **Database Name**: The name of the database you wish to connect to.
- **Username**: A user name that allows you to access the MySQL database.
- **Password**: The password for the user name used to access the MySQL database.

MySQL Database Connections

When connecting to a MySQL database use the following methods.

Method	*Description*
`mysql.connector.connect()`	Connect to a MySQL database using the required arguments. Once the connection is established you can create a ***cursor*** object to execute database operations.
`cursor.execute()`	Execute SQL queries from Python.
`connection.close()`	Once your database operations are done, use a `cursor.execute` to close the cursor object, then use `connection.close()` to close the database connection.

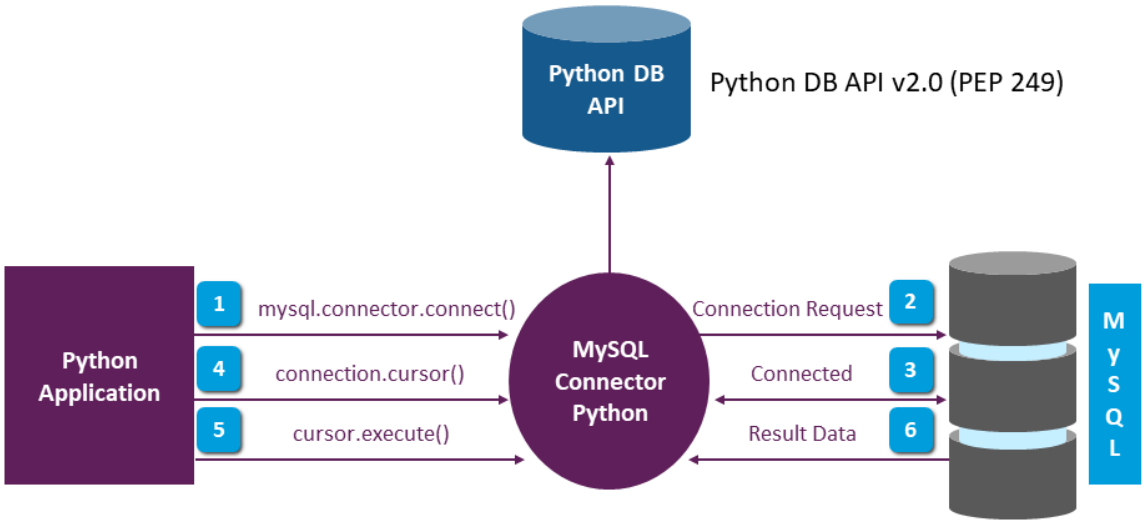


Figure 4-1: MySQL database connections.

To create a connection to the database named "Cars" on the local machine with the database user account "carsdbaccess", you would execute the following code:

```
import mysql.connector
from mysql.connector import Error
connect:
   connection = mysql.connector.connect
(host="localhost",database='cars",user='carsdbaccess",password="QDS342!!$")
   if connection.is_connected():
      db_Info  =  connection.get_server_info ()
            print ( "Connected to MySQL Server version " ,  db_Info )
            cursor  =  connection.cursor ()
            cursor.execute ( "select database();" )
            record  =  cursor.fetchone ()
            print ( "You're connected to database: " ,  record )

except Error as e:
   print("Error while connecting to MySQL", e)
close:
   if (connection.is_connected())
   cursor.close()
   connection.close()
   print("MySQL connection is closed")
```

MySQL Database Creation

You can create a database with `mysql.connector` using the `CREATE DATABASE` statement. To create a new database named "newdb" with the previous id and passwords, you would use the following code:

```
import  mysql.connector

newdb = mysql.connector.connect(host="localhost",user="carsdbaccess","QDS342!!
$")

cursor = newdb.cursor()

mycursor.execute ( "CREATE DATABASE mydatabase" )
```

Guidelines for Connecting to Databases in Python

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when connecting to databases in Python.

Connect to Databases in Python

When connecting to databases in Python:

- Choose a database that meets the data storage, security, access, and connectivity requirements required by your app and organization.
- Work with database administrators to get credentials and access required for your app to the database.
- When creating a database to use with an app:
  - Determine the different data elements required for the application.
  - Divide the data elements into common groups (called data normalization).

- Determine the data type required to store each data element.
- Determine a primary key for each table.
- Create a separate database for each application.
- Create the tables and data fields.

ACTIVITY 4-1
Connecting to a Database in Python

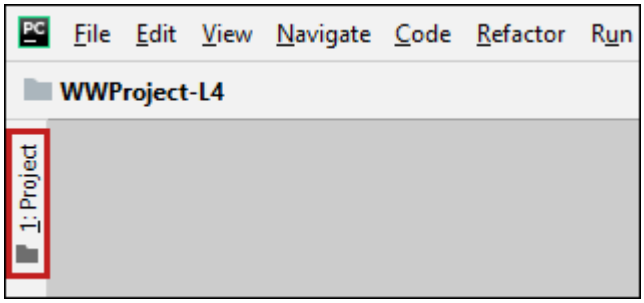
Data Files
All project files in C:\094022Data\Creating Data-Driven Applications\WWProject-L4.

Before You Begin
PyCharm is open.

Scenario
You want to modify the Wood Delivery Calculator to get the product and price data from a database to make it easier to update product information. Before you begin creating the databases for the Wood Delivery Calculator, you want to practice connecting to and reviewing the inventory database. For the purpose of this activity, you will run a premade Python script to create the inventory database. Then, you will connect to it and review a table and the data it contains.

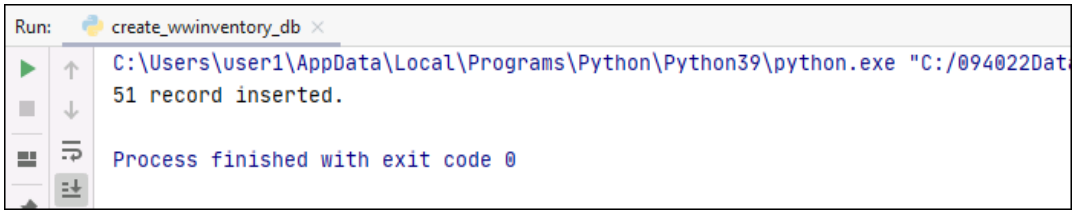
---

1. Open the **WWProject-L4** project.
   a) In the Welcome to PyCharm window, select **Open**.
   b) Navigate to the **C:\094022Data\Creating Data-Driven Applications\WWProject-L4** folder.
   c) With the **WWProject-L4** folder selected, select **OK**.
   d) If the **Project** pane is not visible, select the **1: Project** tab.



   e) In the **Project** pane on the left side of the PyCharm window, verify that **WWProject-L4** is listed.

2. Create the wwinventory database.
   a) In the **Project** pane on the left, double-click **create_wwinventory_db.py**.
   b) Scroll down and briefly review the code.
   c) Run **create_wwinventory_db.py**.
   d) Observe that 51 records were inserted into the new database.



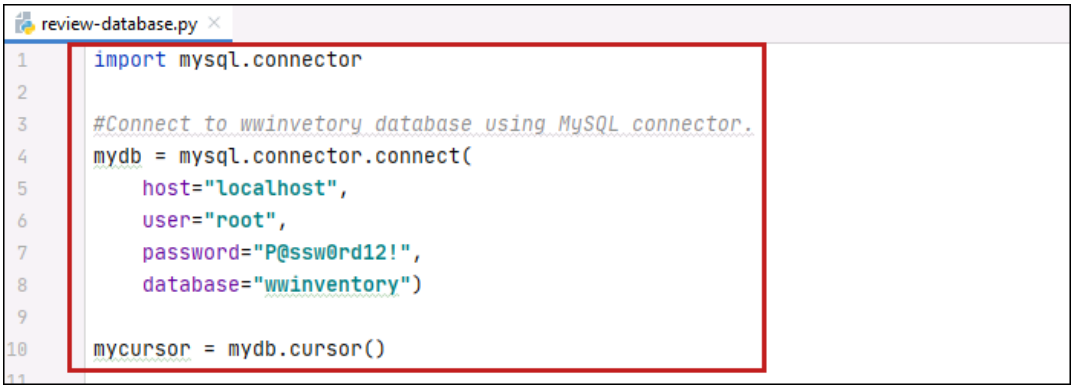
e) In the **Run** pane, close the **create_wwinventory_db** tab.
f) Close the **create_wwinventory_db.py** editor tab.

3. Create the review-database.py file in the WWProject-L4 project.

a) In the **Project** pane on the left, right-click the **WWProject-L4** project, and select **New→Python File**.
b) In the **Name** box, type ***review-database*** and press **Enter**.

4. Review a table in the database.

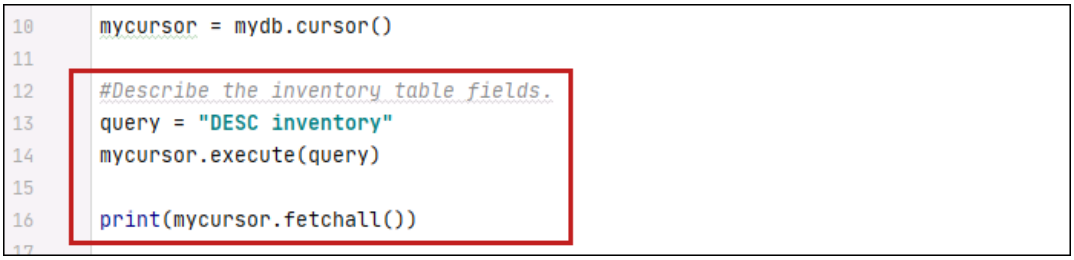
a) In **review-database.py**, starting on line 1, type the following to write the statements that connect to the database.



These statements:

- Import the MySQL Connector library.
- Use the `connect()` method to connect to the database which includes the MySQL server name, user name and password, and the database.
- Use the `cursor()` method to create a cursor object to point to the database connection.

b) After the `cursor` method, starting on line 12, type the following to add code to describe the database.



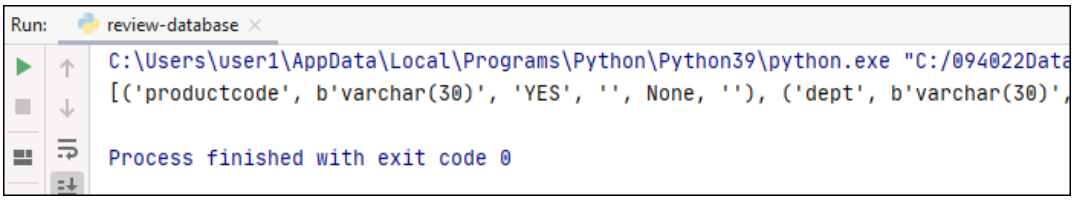
These statements:

- Use the `execute()` method to run the DESC query of the inventory table. This will return the structure of the table.
- Print the results of the `fetchall()` method to show the table information.

5. Run the review-database.py script.

a) Run **review-database.py**.

b) Observe the results that list each field in the table along with some of their attributes.



c) In the **Run** pane, close the **review-database** tab.

6. Print the records of a table in the database.

a) In **review-database.py**, delete the highlighted code that describes the inventory table.

```
10  mycursor = mydb.cursor()
11
12  #Describe the inventory table fields.
13  query = "DESC inventory"
14  mycursor.execute(query)
15
16  print(mycursor.fetchall())
```

You will replace this code in the next substep.

b) After the `cursor` method, starting on line 12, type the following to add code to print the fields from the inventory table.

```
10  mycursor = mydb.cursor()
11
12  #Print the inventory table fields.
13  query = "SELECT * FROM inventory"
14  mycursor.execute(query)
15
16  records = mycursor.fetchall()
17
18  for record in records:
19      print(record)
20
21  mydb.close()
22
```

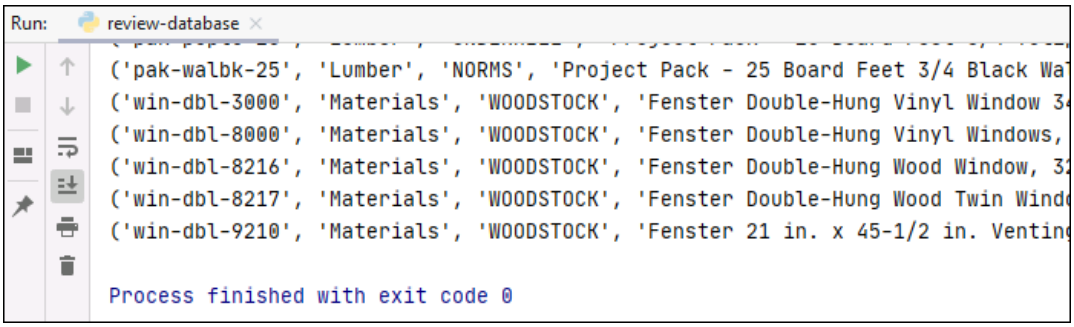
These statements:

- Use the `execute()` method to run the SELECT query of the inventory table. This will return the values of the fields in the table.
- Use the `fetchall()` method to retrieve the information for all of the records.
- Use a loop to print each record on its own line.
- Close the database connection.

7. Run the review-database.py script.

a) Run **review-database.py**.

b) Observe the results that list all records in the table.



c) In the **Run** pane, close the **review-database** tab.

d) Close the **review-database.py** editor tab.

ACTIVITY 4-2
Creating a Database to Support an App

Before You Begin
The **WWProject-L4** project is open in PyCharm.

Scenario
You will write a script that will create the MySQL database file named wwproducts which has one table that holds the products and prices used for the **Wood Delivery Calculator**. You will also create a database named wworders that has a table to store order information from the **Wood Delivery Calculator**.

- The first database will be stored in the default **MySQL** folder in a file named **wwproducts.idb**.
- It includes one table:
  - The **products** table, which contains:
    - The **products** field
    - The **price** field
- The second database will be stored in the default **MySQL** folder in a file named **wworders.idb**.
- It includes one table:
  - The **ordersummary** table, which contains:
    - The **ordernumber** field
    - The **orderdate** field
    - The **shipmethod** field
    - The **totalprice** field

---

1. Create the **create-databases.py** file in the **WWProject-L4** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L4** project, and select **New→Python File**.
   b) In the **Name** box, type ***create-databases*** and press **Enter**.

2. Create a new database.

a) In **create-databases.py**, starting on line 1, type the following to connect to the MySQL server.

```
create-databases.py
1  import mysql.connector
2
3  #Connect to MySQL connector.
4  mydb = mysql.connector.connect(
5      host="localhost",
6      user="root",
7      password="P@ssw0rd12!")
8
9  mycursor = mydb.cursor()
10
```

These statements:

- Import the MySQL Connector library.
- Use the `connect()` method to connect to the database which includes the MySQL server name, user name and password, and the database.
- Use the `cursor()` method to create a cursor object to point to the database connection.

b) Starting on line 11, type the following to create the wwproducts and wworders databases.

```
9  mycursor = mydb.cursor()
10
11 # Create database for products.
12 mycursor.execute("CREATE DATABASE wwproducts")
13
14 print("wwproducts database created.")
15
16 # Create database for orders.
17 mycursor.execute("CREATE DATABASE wworders")
18
19 print("wworders database created.")
20
21 mydb.close()
22
```

These statements:

- Use the `execute()` method to create each database.
- Print that each database was created.
- Close the database connection.

3. Create the **products** table.

a) Starting on line 23, type the following to connect to the wwproducts database.

```
21  mydb.close()
22
23  #Connect to wwproducts database.
24  mydb = mysql.connector.connect(
25      host="localhost",
26      user="root",
27      password="P@ssw0rd12!",
28      database="wwproducts")
29
30  mycursor = mydb.cursor()
31
```

You connected to the MySQL server at the beginning of your code to create the two databases. Now, you will connect to one of those databases in order to create tables inside it.

b) Starting on line 32, type the following to create the products table.

```
30  mycursor = mydb.cursor()
31
32  #Create products table.
33  mycursor.execute("CREATE TABLE products (product VARCHAR(255), "
34                   "price DECIMAL(12,2), PRIMARY KEY(product))")
35  mydb.close()
36
37  print("products table created in the wwproducts database.")
38
```

These statements:

- Use the `execute()` method to create the products table with the product and price fields.
- Close the database connection.
- Print that the table was created.

4. Create the **ordersummary** table.

a) Starting on line 39, type the following to connect to the wworders database.

```
37  print("products table created in the wwproducts database.")
38
39  #Connect to wworders database.
40  mydb = mysql.connector.connect(
41      host="localhost",
42      user="root",
43      password="P@ssw0rd12!",
44      database="wworders")
45
46  mycursor = mydb.cursor()
```

You connected to the wwproducts databases previously and will now connect to the wworders database.

b) Starting on line 48, type the following to create the products table.

```
46  mycursor = mydb.cursor()
47
48  #Create ordersummary table.
49  mycursor.execute("CREATE TABLE ordersummary (ordernumber "
50                   "INT AUTO_INCREMENT PRIMARY KEY, orderdate DATE, "
51                   "shipmethod VARCHAR(255), totalprice DECIMAL(12,2))")
52  mydb.close()
53
54  print("ordersummary table created in the wworders database.")
55
```

These statements:

- Use the `execute()` method to create the ordersummary table with the ordernumber, orderdate, shipmethod, and totalprice fields.
- Close the database connection.
- Print that the table was created.

5. Run the **create-databases.py** script.

a) Run **create-databases.py**.

b) Observe that two databases and two tables were created.

```
Run:  create-databases
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Dat
wwproducts database created.
wworders database created.
products table created in the wwproducts database.
ordersummary table created in the wworders database.

Process finished with exit code 0
```

c) In the **Run** pane, close the **create-databases** tab.

d) Close the **create-databases.py** editor tab.

TOPIC B

Store, Update, and Delete Data in a Database

Once you connect to a database, you have to perform actions on the data in that database including accessing the data, processing it, and storing it back into the database. In this topic, you will store, update, and delete data in a database.

Programmatic Database Workflow

Before you get into the nuts and bolts of working with databases and the data they contain, it's worthwhile to articulate the common workflow developers program so their apps can work effectively with databases. The workflow goes like this:

1. Create database (if one doesn't exist).
2. Connect to the database.
3. Get the cursor.
4. Execute commands and queries to add, retrieve, and update data.
5. Commit changes to the database.
6. Close the connection to the database.

Python Data Structures

While you are likely familiar with the built-in Python data structures, you should be aware of the object-oriented features of these data structures. The following table lists the data structures, provides a brief description, and when they should be used instead of a class, and when they should not be used.

Data Structure	*Description*	*Recommended Usage as a Data Structure*
Empty objects	An object instantiated without a subclass.	While you can set attributes on an empty object, this practice is not recommended. You are most often better off using built-in objects designed for storing data.
Tuples	Store a specified number of objects in order. They are immutable as objects cannot be added, removed, or replaced on the fly.	Use tuples to store data values in a sequence, that is, data that is different from one another. You can also unpack data stored in tuples into variables. One disadvantage is that tuples can be hard to read, which can make code hard to maintain because of the need to know the position index of the data stored in the tuple. Use tuples when the values stored will be used at the same time, that is unpacked, moved to variables, and processed.
Named tuples	Similar to tuples, but assign object-like variable referencing for data stored in the named tuple.	Named tuples can be used wherever you would use a tuple and are immutable. They are a good way to group read-only data together, make reading code easier, and make code more self-documenting by allowing you to access data by name rather than position index.

Data Structure	Description	Recommended Usage as a Data Structure
Dataclasses	Similar to objects with a clean syntax for adding predefined attributes.	Dataclasses are instantiated like other objects, and once done allow you to access and update the attributes stored inside and assign other attributes to the object. Dataclasses require less code to create, provide a useful string representation, and allow you to sort and compare data stored in the dataclass.
Dictionaries	Containers that allow you to map objects to other objects.	Dictionaries look up values based on a key. They are the recommended data structure to find an object based on another object.
Lists	Store multiple items in a single variable. Like tuples lists store values in a sequence.	Lists can be simpler to use and are managed in the same way you manage other objects. You don't need to import them or call methods on them. Lists are best used to store multiple instances of the same type of object, such as strings or numbers, and can be sorted.
Sets	Sets store values, but don't ensure values are unique. A number added to a set 10 times would only show up once.	Where as lists and tuples cannot have the same value more than once, sets can. You might use a set to store the names of song artists to whom you wish to assign songs.

Tables and Data Fields

Relational databases offer good performance in sorting and looking up the data they store because of how the database is structured. Relational databases store data in fields and groups common data fields into tables, and relate tables to each other based on shared data.

Data fields define the individual data elements in an application. Items such as your name, age, address, and phone number are stored in the database as data fields.

Each data field is created using a specific data type. The data type defines how the data is stored in the database. The following are common data types.

Data Type	Values Stored
Integer	Whole numbers
Floating Point	Values with decimal places
Character	Standard text that's usually a set size (such as a zip code)
Variable Character	Standard text that can vary in length (such as a name)
Date	Dates
Date/Time	Dates and times

Database Queries

The SQL `SELECT` statement is used to submit data queries to MySQL. The database engine processes the query and returns a result set. The result set contains all of the data records in the table that match the query criteria. The format of the `SELECT` statement in a string concatenated query is:

```
SELECT fields list FROM table WHERE criteria
```

The `WHERE` clause is what limits what data records are returned:

```
sqlselect = "SELECT product, price FROM prices WHERE price > 1.00"

cursor.execute(sqlselect)
```

You then submit the query to the database using the `execute()` method.

The `execute()` method returns the result set generated by the `SELECT` statement as a list of values. The list matches the table data field order. You can iterate through the result set using a standard `for` statement:

```
import mysql.connector

mydb = mysql.connector.connect(
   host="localhost"
   user="adminuser"
   password="adminpass"
   database="productsdb"
)

cursor = mydb.cursor

sqlselect = "SELECT product,price FROM prices"

for row in cursor.execute(sqlselect):
         prices[row[0]] = row[1]
```

For each iteration, the product data field value is assigned to the `row[0]` list variable, and the price data field value is assigned to the `row[1]` list variable.

SQL Injection Attack Prevention

SQL injection attacks modify SQL direct queries to databases and can do any of the following:

- Retrieve hidden data such as extra records.
- Change application processing logic.
- Perform a UNION attack to retrieve data from different tables.
- Allow attackers to examine the database which may allow them to gain further access to sensitive data.
- Perform a BLIND SQL injection to ask the database true or false questions to exploit the database.

Hackers may be able to steal sensitive data such as credit card numbers, bank account numbers, change or delete data, perform denial-of-service attacks on the database, or create a back-door to allow continued access to the database.

A fundamental and recommended way to prevent SQL injection attacks is to use parameterized queries instead of string concatenation within the query. This process of escaping values in queries is accommodated by the `mysql.connector` which uses the placeholder `%s` to escape values.

Inserting records using string concatenation:

```
cursor = mydb.cursor

sqlinsert = """INSERT INTO Prices (products, price) VALUES ('Doohicky',
'4.99')"""

cursor.execute(sqlinsert)

mydb.commit()
```

Inserting records using a parameterized statement:

```
cursor = mydb.cursor

sqlinsert = "INSERT INTO Prices (products, price) VALUES (%s, #s)"

insertvalues = ("Doohickey", "4.99")

cursor.execute(sqlinsert, insertvalues)

mydb.commit()
```

Additional Information

SQL injection attack and prevention: **https://portswigger.net/web-security/sql-injection**.

Data Insertion

To insert data into a MySQL database, use the `INSERT INTO` statement. The statement is formatted in the following way:

`INSERT INTO table (field1, field2) VALUES (field1_value, field2_value)`

The Python connector for MySQL *does not* automatically commit changes you've made. In order to do so, you must include the `COMMIT` statement which will commit the current transactions, saving changes to the database. Once you've committed your changes, you can also use the `.rowcount` method to display the number of records affected.

To insert the new product "Doohickey" at a price of "4.99" into the Prices table, you would use the following code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host="localhost"
   user="adminuser"
   password="adminpass"
   database="productsdb"
)

cursor = mydb.cursor

sqlinsert = "INSERT INTO Prices (products, price) VALUES (%s, #s)"

insertvalues = ("Doohickey", "4.99")

cursor.execute(sqlinsert, insertvalues)

mydb.commit()

print(cursor.rowcount, "record(s) inserted.
```

Data Update

You can update records in a table using the `UPDATE` statement:

`"UPDATE table SET field = "new value" WHERE value = 'old value'"`

To update the name of "Doohickey" in the Prices database to "Thingamajig", you can use the following code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host="localhost"
   user="adminuser"
   password="adminpass"
   database="productsdb"
)

cursor = mydb.cursor

sqlupdate = "UPDATE Prices SET product = %s WHERE product = %s"
sqlupdatevalue = ("Thingamajig", "Doohickey")

cursor.execute(sqlupdate, sqlupdatevalue)

mydb.commit()

print(cursor.rowcount, "record(s) updated.
```

Data Deletion

To delete records from a MySQL database, use the `DELETE FROM` statement:

```
"DELETE FROM table WHERE field = 'value to match'"
```

Note: The `DELETE FROM` statement deletes the entire record associated with the matched value.

To delete the Thingamajig product from the Prices table, use the following code:

```
import mysql.connector

mydb = mysql.connector.connect(
   host="localhost"
   user="adminuser"
   password="adminpass"
   database="productsdb"
)

cursor = mydb.cursor

sqldelete = "DELETE FROM Prices WHERE product = %s"
sqldeletevalue = ("Thingamajig")

cursor.execute(sqldelete, sqldeletevalue)

mydb.commit()

print(cursor.rowcount, "record(s) deleted.
```

Guidelines for Storing, Updating, and Deleting Data in a Database

Follow these guidelines when storing, updating, and deleting data in a database.

Store, Update, and Delete Data in a Database

When storing, updating, and deleting data in a database:

- Use tuples to store data values in a sequence, that is, data that is different from one another.
- Use named tuples to group read-only data together, make reading code easier, and make code more self-documenting by allowing you to access data by name rather than position index.
- Consider using dataclasses for objects with predefined attributes where you wish to sort and compare the attributes of the dataclass.
- User parameterized commands and queries to help prevent SQL injection attacks.
- Remember to commit changes made by your app to a MySQL database prior to closing the connection in order to save the changes your app has made.

ACTIVITY 4-3
Inserting Data Into a Database

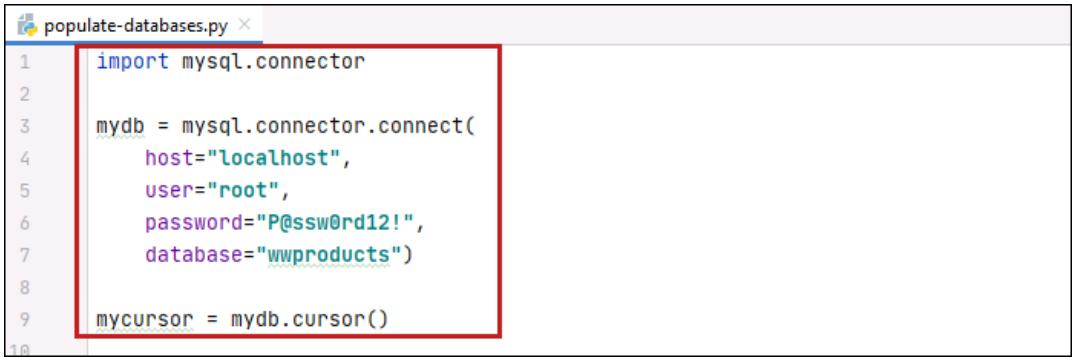
Before You Begin
The **WWProject-L4** project is open in PyCharm. You have created the **wwproducts** and **wworders** database files.

Scenario
Now that you have created the databases and tables, you need to add the actual data. You will add the product information into the products table. The wworders database will contain information from completed orders so data will be added to it as orders are completed through the app.

---

1. Create the **populate-databases.py** file in the **WWProject-L4** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L4** project, and select **New→Python File**.
   b) In the **Name** box, type ***populate-databases*** and press **Enter**.

2. Populate the products table.
   a) In **populate-databases.py**, starting on line 1, type the following to import the MySQL connector and connect to the wwproducts database.



b) Starting on line 11, type the following to populate the products table.

```
mycursor = mydb.cursor()

sql = "INSERT INTO products (product, price) VALUES (%s, %s)"
val = [
    ("Lumber-2x4x96 inch",5.50),
    ("Lumber-2x6x96 inch",8.50),
    ("Lumber-2x4x96 inch Treated",8.75),
    ("Lumber-2x6x96 inch Treated",13.25),
    ("Plywood-1/4x48x96 inch",33.00),
    ("Plywood-1/2x48x96 inch",37.25),
    ("Plywood-1/2x48x96 inch Particleboard",19.25)
]

mycursor.executemany(sql, val)

mydb.commit()
mydb.close()

print(mycursor.rowcount, "records inserted in products table.")
```

These statements:

- Use a variable for the parameterized `INSERT` query to insert the product and price values to the products table.
- Use a variable for the product and price values.
- Use the `executemany()` method to insert the values to the table.
- Use the `commit()` method to commit the current insert transaction to the table.
- Close the database connection.
- Print the number of records that were inserted.

3. Run the **populate-databases.py** script.

a) Run **populate-databases.py**.

b) Observe that seven records were inserted into the products table.

```
Run:  populate-databases
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Dat
7 records inserted in products table.

Process finished with exit code 0
```

c) In the **Run** pane, close the **populate-databases** tab.

d) Close the **populate-databases.py** editor tab.

ACTIVITY 4-4
Updating Data in a Database

Before You Begin
The **WWProject-L4** project is open in PyCharm. You have created the **wwproducts** and **wworders** database files.

Scenario
You will create a new script to update the prices stored in the **wwproducts** database file. You will need to create a new application and UI to display the product information. It needs to read the product data from the **wwproducts** database and allow users to commit any changes back to the **wwproducts** database.

---

1. Create the **update-database.py** file in the **WWProject-L4** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L4** project, and select **New→Python File**.
   b) In the **Name** box, type ***update-database*** and press **Enter**.

2. Create the template code to build a window and import the `tkinter` and `mysql.connector` libraries.
   a) In **update-database.py**, starting on line 1, type the following to create the application frame of the new app.

update-database.py

```
from tkinter import *
import mysql.connector

class Application(Frame):

    def __init__(self, master):
        super(Application, self).__init__(master)
        self.grid()

window = Tk()
window.title("Wood Product Prices")
window.geometry("400x300")
app = Application(window)
app.mainloop()
```

These statements:

- Import the `tkinter` library and the `mysql.connector` library.
- Define the `Application` class and the constructor method.
- Instantiate the `Tk()` object, set the window title and size, instantiate the `Application` class, and run the `mainloop()` method.

3. In the `Application` constructor method, add code to read the prices from the **wwproducts** database, and create the widgets method.

a) In the `Application` constructor method, but before the code that instantiates the window, starting on line 10, type the following to connect to the MySQL database and read the product information.

```
6       def __init__(self, master):
7           super(Application, self).__init__(master)
8           self.grid()
9
10          self.mydb = mysql.connector.connect(
11              host="localhost",
12              user="root",
13              password="P@ssw0rd12!",
14              database="wwproducts")
15
16          self.mycursor = self.mydb.cursor()
17          self.prices = []
18          self.products = []
19          self.mycursor.execute("SELECT product, price FROM products")
20          self.row = self.mycursor.fetchone()
21          while self.row is not None:
22              self.products.append(self.row[0])
23              self.prices.append(self.row[1])
24              self.row = self.mycursor.fetchone()
25          self.create_widgets()
26
```

These statements:

- Connect to the database and create the cursor.
- Create the products and prices lists.
- Submit a `SELECT` statement to retrieve the products and prices from the products table.
- Using a `while` statement to add the product and price values from the table into products and prices lists.
- Call the `create_widgets()` method, which you haven't provided yet.

4. Create a form to display the prices values.

a) After the constructor, but before the code that instantiates the window, starting on line 27, type the following to create the headings.

```
25          self.create_widgets()
26
27      def create_widgets(self):
28          self.lblproduct = list(self.products)
29          self.entprice = list(self.prices)
30          line = 1
31          self.label0 = Label(self, text="Wood Product Prices")
32          self.label0.grid(row=0, column=0, columnspan=2, sticky=W)
33
34          self.heading1 = Label(self, text="Product")
35          self.heading1.grid(row=line, column=0, sticky=W)
36          self.heading2 = Label(self, text="Price")
37          self.heading2.grid(row=line, column=1, sticky=W)
38
39      window = Tk()
```

These statements:

- Create the `create_widgets()` method.
- Create headings for the table.

b) After the headings, but before the `window` instantiation code, starting on line 39, type the following to create a row for each product.

```
37          self.heading2.grid(row=line, column=1, sticky=W)
38
39          for i in range(len(self.products)):
40              line += 1
41              self.lblproduct[i] = Label(self, text=self.products[i])
42              self.lblproduct[i].grid(row=line, column=0, sticky=W)
43              self.entprice[i] = Entry(self, width=5)
44              self.entprice[i].grid(row=line, column=1, sticky=W)
45              self.entprice[i].insert(END, self.prices[i])
46
47      window = Tk()
```

These statements:

- Create a row for each product.
- Create two columns, one for the product name, and one for the product price.
- Create the fields to display the prices for each product.

5. Add widgets that will enable the user to submit the update.

a) At the end of the `create_widgets()` method, but before the `window` instantiation code, starting on line 47, type the following to create the Update prices button.

```
45            self.entprice[i].insert(END, self.prices[i])
46
47        self.label4 = Label(self, text="")
48        self.label4.grid(row=line, column=0)
49        self.button1 = Button(self, text="Update prices", command=self.update)
50        line += 1
51        self.button1.grid(row=line, column=0, sticky=W)
52        self.entry4 = Entry(self, width=10)
53        self.entry4.grid(row=line, column=1)
54
55    window = Tk()
```

These statements provide:

- A blank row.
- A button to submit the updated data using the `self.update` method.
- An **Entry** field to display the status.

6. Add the `update()` method to update the prices table with the values from the form field.

a) After the `create_widgets()` method, but before the `window` instantiation line, starting on line 55, type the following to loop and get the updated prices.

```
53        self.entry4.grid(row=line, column=1)
54
55    def update(self):
56        for i in range(len(self.products)):
57            self.prices[i] = self.entprice[i].get()
58
59    window = Tk()
```

These lines:

- Define the `update()` method.
- Use a loop `for` to retrieve the data values from the form and place them in the appropriate prices list.

b) Starting on line 58, type the following to add updating the price for each product in the database to the loop.

```
55      def update(self):
56          for i in range(len(self.products)):
57              self.prices[i] = self.entprice[i].get()
58              sql = "UPDATE products SET price = %s WHERE product = %s"
59              val = (self.prices[i], self.products[i])
60              self.mycursor.execute(sql, val)
61              self.mydb.commit()
62
63          self.mydb.close()
64          self.entry4.delete(0, END)
65          self.entry4.insert(END, "Updated")
66
```

These lines:

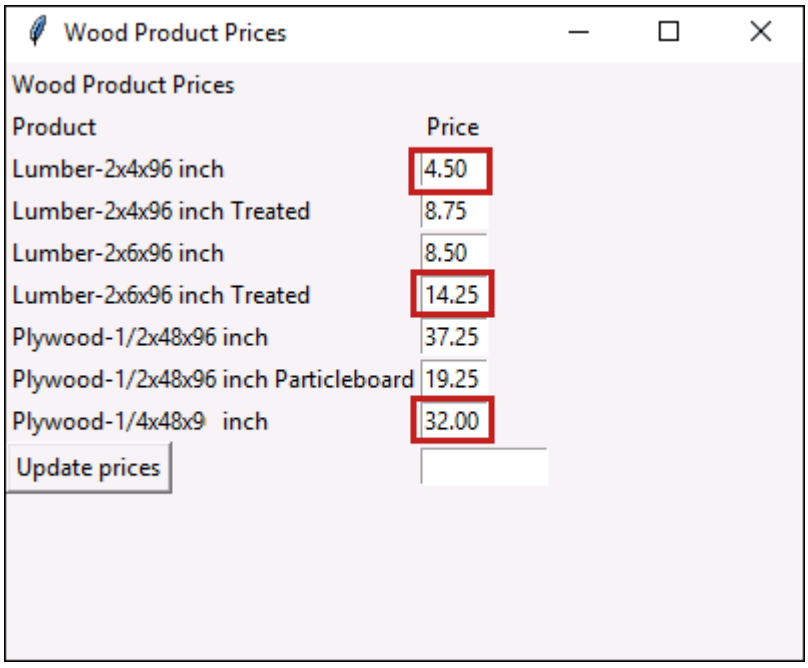
- Use the existing `for` loop to iterate through the products list and submit an `UPDATE` statement for each product, updating the price field in the products table.
- Call the `commit()` method after the `for` loop to commit the updates.
- Close the database connection.
- Display the "Updated" message in the `self.entry4` **Entry** box.

7. Run the update-database.py application.

a) Run the **update-database.py** script.

The window appears with the current database entries.

b) Change the prices as shown, then select the **Update prices** button.



The **Updated** message is displayed.

c) To verify the database was changed, close the Wood Product Prices window, then re-run the program and verify the updated values are there.

d) Close the Wood Product Prices window to return to PyCharm.

e) In the **Run** pane, close the **update-database** tab.

f) Close the **update-database.py** editor tab.

ACTIVITY 4-5
Adding a Database to an Application

Before You Begin
The **WWProject-L4** project is open in PyCharm. You have created the **wwproducts** and **wworders** database files.

Scenario
You will now enable the **Wood Delivery Calculator** to read data from the database and commit order information to the **wworders** database.

---

1. Copy the **wdc.py** file to **wdc-database.py**.
   a) Open the **wdc.py** file in the editor.
   b) Select **File→Save As**.
   c) In the **Copy** dialog box, change the file name to ***wdc-database.py*** and then select **OK**.
   d) Ensure that the **wdc-database.py** file is open, and that it is the active document to edit.

2. Update the constructor to read the product and shipping data from the database.
   a) Above the class declaration, on line 2, type the following to `import` the MySQL connector.

```
wdc.py | wdc-database.py
1  from tkinter import *
2  import mysql.connector
3
4  class Application(Frame):
```

   b) In the class declaration, on lines 9-22, make the following changes to make the `products` and `prices` lists empty lists.

```
6      def __init__(self, master):
7          super(Application, self).__init__(master)
8          self.grid()
9          self.products = []
10         self.prices = []
11         self.shipdays = {"1 Day": 20.00,
12                          "2 Days": 15.00,
13                          "3 Days": 10.00}
14         self.lblproduct = list(self.products)
15         self.lblprice = list(self.prices)
16         self.entry = list(self.products)
17         self.create_widgets()
```

c) In the class declaration, after the `self.shipdays` dictionary object, starting on line 15, type the following to connect to the **wwproducts** database.

```
10          self.prices = []
11          self.shipdays = {"1 Day": 20.00,
12                           "2 Days": 15.00,
13                           "3 Days": 10.00}
14
15          self.mydb = mysql.connector.connect(
16              host="localhost",
17              user="root",
18              password="P@ssw0rd12!",
19              database="wwproducts")
20
21          self.mycursor = self.mydb.cursor()
22
23          self.lblproduct = list(self.products)
```

These statements:

- Connect to the **wwproducts** database.
- Create a cursor object.

d) Following the code you just typed, starting on line 23, type the following to query the database for product and price data.

```
21          self.mycursor = self.mydb.cursor()
22
23          self.mycursor.execute("SELECT product, price FROM products")
24          self.row = self.mycursor.fetchone()
25          while self.row is not None:
26              self.products.append(self.row[0])
27              self.prices.append(self.row[1])
28              self.row = self.mycursor.fetchone()
29          self.mydb.close()
30
31          self.lblproduct = list(self.products)
```

This code:

- Executes a `SELECT` statement to retrieve a products table entry.
- Uses a `while` statement to iterate through the records in the table and append the values to a list.
- Closes the database connection.

3. Update for using data from the database.

a) In the `create_widgets` function, on line 45, convert the value of `self.prices` to a string when creating the label for the price.

```
41      for i in range(len(self.products)):
42          line += 1
43          self.lblproduct[i] = Label(self, text=self.products[i])
44          self.lblprice[i] = Label(self,
45                                   text=str("{0:.2f}".format(self.prices[i])))
46          self.entry[i] = Entry(self, width=3)
47          self.entry[i].grid(row=line, column=0)
48          self.lblproduct[i].grid(row=line, column=1, sticky=W)
49          self.lblprice[i].grid(row=line, column=2, sticky=W)
```

Note: There should be three sets of parentheses in this line of code.

b) In the `calculate` method, starting on line 116, make the following changes to convert the `shipdays` value from a float to a decimal when adding it to the order total.

```
113             self.total += (self.prices[i] * int(self.entry[i].get()))
114
115         if (self.ship.get() == "1 Day"):
116             self.total += Decimal(self.shipdays["1 Day"])
117         elif (self.ship.get() == "2 Days"):
118             self.total += Decimal(self.shipdays["2 Days"])
119         elif (self.ship.get() == "3 Days"):
120             self.total += Decimal(self.shipdays["3 Days"])
121
122         strPrice = "{0:.2f}".format(self.total)
```

Note: Decimal shows as an error because we haven't imported the `decimal` module yet.

The product values in the database are stored as the decimal data type. The `shipdays` values in the dictionary are stored as the float data type. A float and a decimal can not be used in a calculation together without converting one of them.

c) On line 3, type the following `import` statement for the `decimal` module above the class declaration.

wdc-database.py

```
1   from tkinter import *
2   import mysql.connector
3   from decimal import *
4
5   class Application(Frame):
```

In order to use the `decimal` module, it needs to be imported.

4. Add code for saving order information to the **wworders** database.

a) In the `create_widgets` function, after the total row, but before the menubar code, starting on line 94, type the following to create the Submit Order button and a blank label.

```
92          self.result.grid(row=line, column=1, sticky=W)
93
94          line += 1
95          self.lblsubmit = StringVar()
96          self.button3 = Button(self, text="Submit Order", command=self.submit)
97          self.button3.grid(row=line, column=0)
98          self.result2 = Label(self, textvariable=self.lblsubmit)
99          self.result2.grid(row=line, column=1, sticky=W)
100
101         menubar = Menu(self)
```

b) After the `calculate` method, but before the `window` instantiation code, starting on line 134, type the following to write the order data to the **wworders** database.

```
132         self.result.insert(END, strPrice)
133
134     def submit(self):
135         """Insert order information to wworders database"""
136         self.mydb = mysql.connector.connect(
137             host="localhost",
138             user="root",
139             password="P@ssw0rd12!",
140             database="wworders")
141
142         self.mycursor = self.mydb.cursor()
143
144         sql = ("INSERT INTO ordersummary "
145                "(orderdate, shipmethod, totalprice) "
146                "VALUES (%s, %s, %s)")
147         val = (date.today(), self.ship.get(), self.total)
148
149         self.mycursor.execute(sql, val)
150         self.mydb.commit()
151         self.mydb.close()
152         self.lblsubmit.set("Order number "
153                            +str(self.mycursor.lastrowid)+" submitted")
154
155 window = Tk()
```

This code:

- Connects to the **wworders** database.
- Uses a variable for the parameterized `INSERT` query to insert the order date, ship method, and total price values to the `ordersummary` table.
- Uses a variable for the order date, ship method, and total price values.
- Uses the `execute()` method to insert the values to the table.
- Uses the `commit()` method to commit the current insert transaction to the table.
- Closes the database connection.
- Sets text variable for the `lblsubmit` label to display the order number of the order submitted.

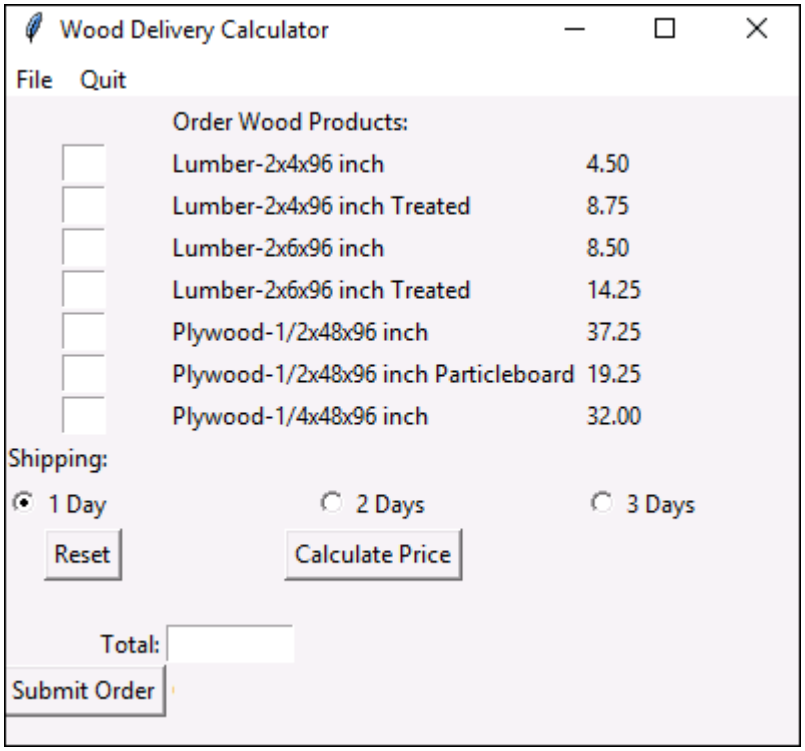
c) On line 4, type the following `import` statement for the `datetime` module above the class declaration.

```
1   from tkinter import *
2   import mysql.connector
3   from decimal import *
4   from datetime import date
5
6   class Application(Frame):
```

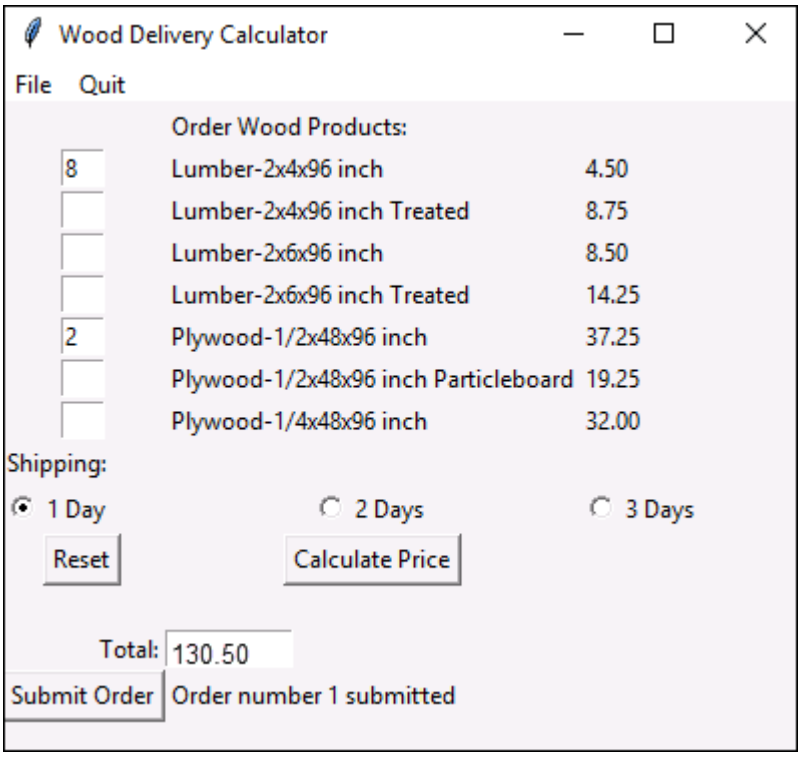
In order to use the `datetime` module, it needs to be imported.

5. Run the **wdc-database.py** application.

a) Run the **wdc-database.py** script.



b) Specify the number of products and shipping method to match the image below, and select **Calculate Price**.



Note: If you select **Submit Order** before you select **Calculate Price**, it will cause an error. You will test and address this issue later in this course.

c) Select **Submit Order**.
The text "Order number 1 submitted" appears to the right of the **Submit Order** button.

d) Select **Reset**.
All selections and total and order number text should be removed.

e) Place a second order by specifying the quantity for different products and a shipping method.
Do not order more than 10 of any one item to avoid astronomical order totals.

f) Select **Calculate Price** and then **Submit Order**.

6. Clean up the workspace.

a) Close the **Wood Delivery Calculator** window.

b) Select **File→Close Project**.
The project is closed.

Summary

In this lesson, you connected to a MySQL database, and created a MySQL database using the MySQL Connector for Python. You then stored, updated, and deleted data using database queries and comments, and learned how to protect your code from SQL injection attacks by using parameterized queries.

How is the data for your application stored?

What kind of actions will you likely perform with data and databases in your app?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

5 Creating and Securing a Web Service-Connected App

Lesson Time: 4 hours

Lesson Introduction

One of the most popular use cases for Python® is to build web-connected apps running as web services for e-commerce and other user-connected apps. In this lesson, you will select network protocols for your app, create a RESTful web service, create a web service client, and secure your web-connected app.

Lesson Objectives

In this lesson, you will:

- Select a network application protocol.
- Create a RESTful web service.
- Create a web service client.
- Secure connected apps in Python.

TOPIC A

Select a Network Application Protocol

Before you can enable your app to communicate over a network, you must select a protocol to use for that communication. In this topic, you will select a network protocol for your web service.

Connectionless vs. Connection-Oriented

The Internet Protocol (IP) standard has become the de facto method of sending data on Local Area Networks (LANs). The IP standard defines how data packets are sent from one device to another across multiple networks. However, by itself, the IP standard doesn't define just how that data is processed at the application level.

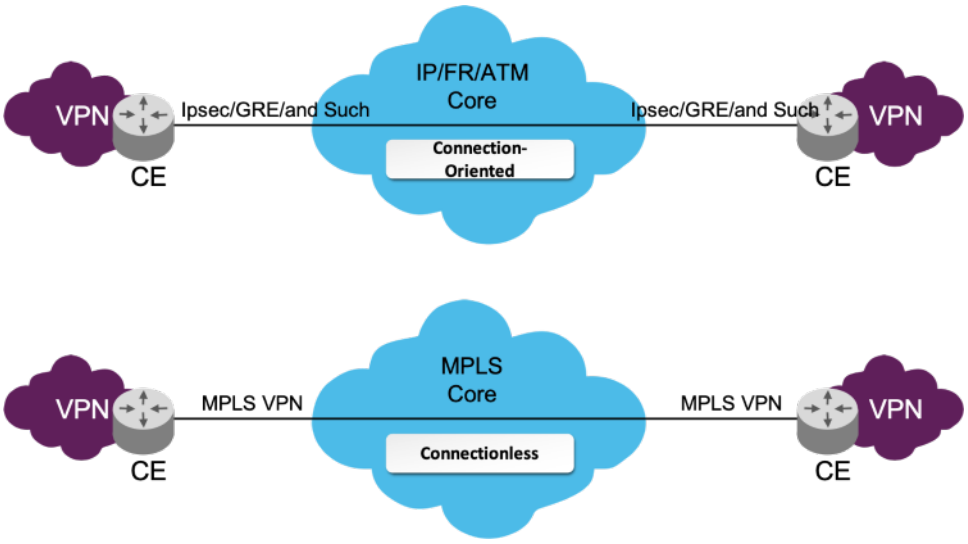


Figure 5-1: Connectionless vs. connection-oriented.

For applications to communicate with each other, a finer control of the data is required. Two popular standards have been built on top of the IP standard that define different methods for moving data between network devices.

Item	*Description*
Transmission Control Protocol (TCP)	TCP defines a detailed protocol for sending data error-free between two network devices, ensuring that all data that is sent on one end is received intact on the other end. The receiving device must acknowledge receipt of each byte of data sent by the sending device, and the sending device can't send more data than the receiving device can handle. These rules ensure that whatever data is sent from one device is properly received by the other device. The downside to this control is reduced speed and throughput of data. There is a lot of overhead involved with validating every byte of data sent. For data-driven applications, that's a small price to ensure the integrity of the data. However, for applications that work in real-time (such as audio and video streams), this extra overhead will cause issues.

Item	Description
User Datagram Protocol (UDP)	To avoid the performance issues of TCP, UDP does not track data the way TCP does. The sending device sends out a packet of data to the receiving device, and has no idea if the packet made it to the destination, or if the destination device was able to process it. While this method can produce lots of dropped packets, it's much faster in transferring data, as there's much less overhead. Most real-time audio and video streaming applications can withstand a few dropped packets in the communication.

TCP is described as providing *connection-oriented* data transfer, whereas UDP provides *connectionless* data transfer.

Client/Server Paradigm

After you decide on a network protocol to use, you must decide on how the network devices will use that protocol to communicate with each other. Both network devices in a connection can't talk at the same time. One must be listening for incoming data at the same time the other device sends it.

That means the two devices must agree on a protocol to establish when to send data and when to listen for data. The most popular protocol method used is the ***client/server paradigm***. With this paradigm, one side of the connection assumes control over the communication channel. The server listens for incoming connections from clients. The client sends a connection request to the server, and the server must accept it before communication can begin.

Once the connection is established, the client sends a command to the server, and the server processes the command, responds to the client requests, provides a user interface for the end user, and so forth. This method is used in most popular IP protocols, such as Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Simple Mail Transfer Protocol (SMTP).

In most server environments, there are multiple server programs that listen for connections on the same network interface. The way the physical server knows which incoming packet goes to which server program is by using ports. A port is a number assigned to each server program listening for incoming packets. The client must know the unique port number for the server it wants to communicate with. Each application server is assigned its own unique port number for the physical server. All standard Internet protocols are assigned well-known port numbers (such as port 80 for HTTP, port 21 for FTP, or port 25 for SMTP). You shouldn't use an existing well-known port number for your application, but usually any port number over 1024 is fair game.

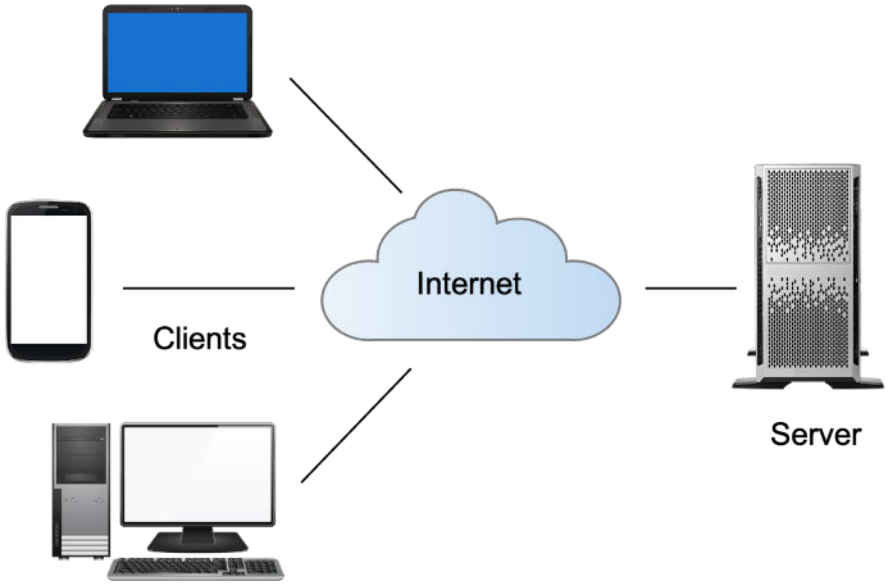


Figure 5-2: Client/server paradigm.

Sockets

To communicate over the network, either local or the Internet, applications require an endpoint. On the server side, a place to listen for requests and send responses. On the client side, a place to submit requests for processing. A socket is one endpoint of a communication link involving apps running over the network.

Applications create sockets to bind themselves to their assigned port number in order to communicate. A socket is bound to the port number so the TCP layer can identify where to send the data received on that port. The endpoint is a combination of the IP address and port number, and every TCP connection can be individually identified by the endpoints used.

Both the server and the client side apps will create sockets in order to communicate with each other. The server side of a web app will have multiple sockets. It will create the initial socket to listen for all requests coming in for the app on the specified IP address and port. Once a client requests to connect, the server will create a new socket on the same port, and establish the communication channel with the client using that socket so that it can continue to listen for incoming connection requests on the original socket.

Depending on the app, multiple sockets may be used to communicate between client and server.

Python Socket Library

The Python `socket` library contains methods for both the client and server programs to interface with the network card and each other by creating, maintaining, and communicating over sockets. To use the `socket` library, you'll need to import it into your Python script:

```
import socket
```

The `socket` library contains lots of different methods to assist you in your network programming. Here are the most common methods that you'll use.

Method	*Description*
`accept()`	Accepts an incoming connection attempt.
`bind(address)`	Binds the socket to an address and port.
`close()`	Closes an established connection.

Method	Description
`connect(address)`	Establishes a connection with a server.
`gethostbyname(host)`	Returns the IP address of the specified host name.
`gethostname()`	Returns the host name of the local system.
`gethostbyaddr(address)`	Returns the host name assigned to an IP address.
`listen(backlog)`	Listens for incoming connections and buffers *backlog* connections, if necessary.
`recv(bufsize)`	Receives up to *bufsize* bytes of data from the connection.
`send(data)`	Sends the specified data through the connection.
`socket(family, type, proto)`	Creates a network socket interface.

Servers

With so many methods available, at first, it may seem somewhat complex to use the `socket` library. However, there's a set process for both the server and client sides of the connection, so coding them is a snap.

For the server, you most often use this sequence of methods:

```
import socket
server = socket.socket(socket.AF_NET, socket.SOCK_STREAM)
host = ''
port= 8000
server.bind(host, port)
server.listen(5)
```

The `socket()` method parameters are somewhat complex, but they're also somewhat standard. The first parameter defines the protocol family (AF_NET requests to use IP), and the second parameter defines the lower-level protocol (SOCK_STREAM is for TCP connections, and SOCK_DGRAM for UDP).

The `bind()` method establishes the program's link to the socket. It requires a tuple value that represents the host address and port number on which to listen. If you bind to an empty host address, the server listens on all IP addresses assigned to the system. If you need to listen on only a specific network interface address, you can specify the specific hostname or IP address for the interface.

After you bind the socket, you use the `listen()` method to start listening for client connections. The program halts at this point, until it receives a connection request from a client. When a connection attempt is received, you pass it to the `accept()` method:

```
client, addr = server.accept()
```

This is somewhat of an odd format. The two variables are required because the `accept()` method returns two data values. The first is a handle that identifies the connection to the client, and the second is the address of the remote client.

After the connection is established, you use the `send()` and `recv()` methods to send data with the client. When the communication is complete, use the `close()` method to close the connection.

Clients

Once you have a server running, you can start working on the client program. The client side of the programming is somewhat simpler:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
host = socket.gethostbyname("localhost")
port = 8000
s.connect((host, port))
```

The client code uses the `gethostbyname()` method to find the connection information for the server, based on the server's hostname. The localhost address points to the same system that the program is running on.

That's all you need for the client! Once the server accepts the connection, you use the `send()` and `recv()` methods to communicate with the server. When the communication is complete, the client also needs to use the `close()` method to close the connection.

Data Transmission

There's a caveat to sending and receiving data across a network connection. Different types of computer hardware systems use different formats for storing data in memory. Because of that, it can be dangerous to send data as text or numeric values from one system to another on the network. To solve that problem, the network interface converts all data to ***network byte order***.

However, because of that, the socket `send()` method can only send bytes. That means the data you place in the `send()` method parameter must be in byte format instead of a text or numeric format. Fortunately, Python provides a function that can convert text to bytes with the `str.encode()` method:

```
s.send(str.encode("This is a test"))
```

On the receiving end of the connection, the `recv()` method returns the received data in raw byte format. To make any sense out of the data, your program will need to convert it to text format. Again, Python comes through with the `bytes.decode()` method:

```
data = s.recv(1024)
string = bytes.decode(data)
```

This is all you need to send and receive simple text and numeric data across the network.

Serialization

Often, you'll have more complex data types (objects, such as lists or dictionaries) that you need to send between devices. To do that, you need to incorporate a process called ***serialization***. With serialization, the complex data type is broken down (serialized) into a text item before being sent on the network. When the data is received on the other end of the connection, the received data is then converted back into the original data type.

While this sounds complicated, as with everything else, there are Python methods to help you out. Currently, the most popular method used to serialize complex data types is the JavaScript Object Notation (JSON) data-interchange format. The `json` Python library implements methods that use JSON to serialize complex data types and convert them back into the original data type. The `dumps()` method serializes the complex data into a text string, and the `loads()` method converts it back into the complex data type.

```
output = str.encode(json.dumps(data))
client.send(output)

data = client.recv(1024)
text = json.loads(bytes.decode(data))
```

Guidelines for Network Programming

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when programming for network communication.

Network Programming

The `json` and `socket` library methods simplify the process of sending complex data between devices across any type of network. When programming for network communication:

- Serialize complex data using the json `dumps()` method.
- Encode the serialized data using the `str.encode()` method.
- Send the encoded data to the remote host using the socket `send()` method.
- On the other end, receive the data using the socket `recv()` method.
- Decode the data using the `bytes.decode()` method.
- Convert the data back to its original data type using the json `loads()` method.

ACTIVITY 5-1
Defining a Network Application Protocol

Scenario
Woodworkers Wheelhouse wants to move the databases to a more centralized location. You are going to turn the **Wood Delivery Calculator** into a client application that connects to an app that is stored on this other location. For now, you assume it will be on a server, but it could later be decided that a web service should be used instead. The server/web service will provide the latest pricing information to each instance of the client application. The server/web service will run on a separate server or be hosted as a web service in the cloud, and instances of the client will run on other computers that connect to the server over a LAN, or the web service over the Internet. Before you start, you'll need to work out some networking details.

---

1. What connection protocol and port will you use for the application?

2. You will modify the Wood Delivery Calculator to be either the client application or the server application. Which one should it be?

3. Which application should initiate the connection?

4. What requests must the server\web service be able respond to?

TOPIC B

Create a RESTful Web Service

There are multiple ways to build a web-connected app. In this topic, you will see the pros and cons of some of those methods and create a RESTful web service for use with your app.

APIs and Web Services

An Application Programming Interface (API) is a set of functions and procedures that can be used to access, interact, and program a software application. APIs are how programmers get different apps to talk to each other, so that an app can share data and utilize the other app's functionality. For example, a company uses Mailchimp® software for their email marketing and Microsoft® Outlook® for Office 365™ to send email. Marketers for the company will connect Mailchimp to Outlook using the Outlook and Mailchimp APIs to allow the two apps to share information.

APIs have been around for a long time, but since the growth of the Internet and cloud-based apps and data, many APIs expose application data and functionality over the Internet. APIs that do are called web APIs.

A web service is any software that makes itself available over the Internet using standard messaging protocols to transfer data through HTTP requests using eXtensible Markup Language (XML) formatted as defined by the Web Services Description Languages (WSDL) definition put forth by the World Wide Web Consortium (W3C). Web services are typically part of Service Oriented Architecture (SOA), which is a design architecture for applications that have different features split up and made available through networked services.

Every web service is an API because a web service, by definition, is an app's data and functionality. Not all APIs, however, are web services as they don't necessarily use standardized protocols for their interaction, and don't necessarily require network functionality. Put more fundamentally, if you're writing an app and you want other machines to be able to interface with it, you'll write an API to allow that to happen. If your app is going to exist on the Internet, and you want users or other apps to be able to connect to it and access its information or capabilities, you'll write a web service to allow that to happen.

Additional Information

For additional information on WSDL, see: **https://www.w3.org/TR/wsdl.html**.

For additional information on XML, see: **https://www.w3schools.com/xml/xml_whatis.asp**.

For additional information on APIs, see: **https://en.wikipedia.org/wiki/API**.

For additional information web services, see: **https://en.wikipedia.org/wiki/Web_service**.

Example Web Service

The following free web service allows you to access an open source collection of dog images: **https://dog.ceo/dog-api/**.

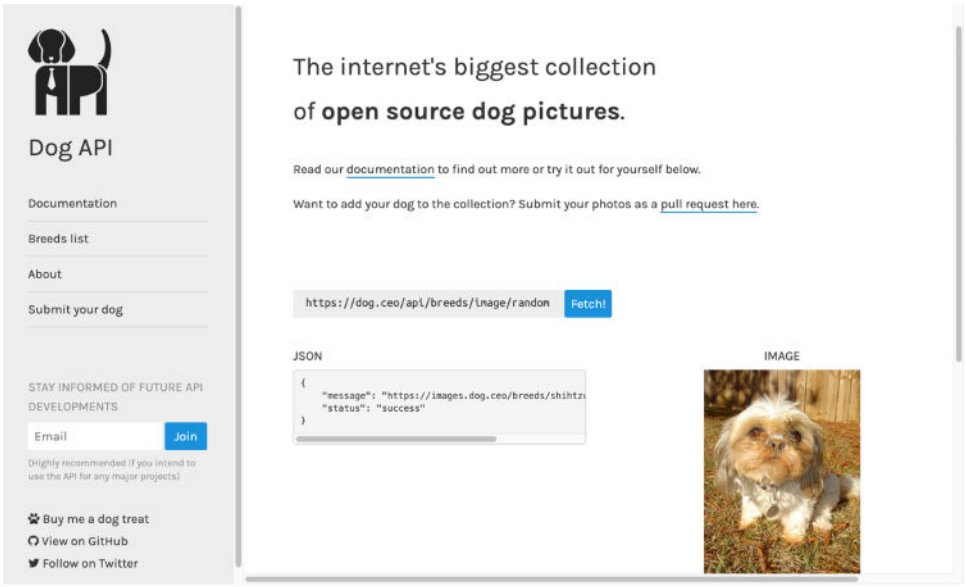


Figure 5-3: Example web service.

Web Service Business Logic

Businesses have rules about how they conduct business. For example, when an item is purchased, purchaser information must be collected as well as payment data. Then, items must be procured from inventory, and then shipped to the requested address. Along the way, a number of other updates and communications must be completed such as updating inventory, potentially ordering replacement inventory, and communicating order status to customers and others.

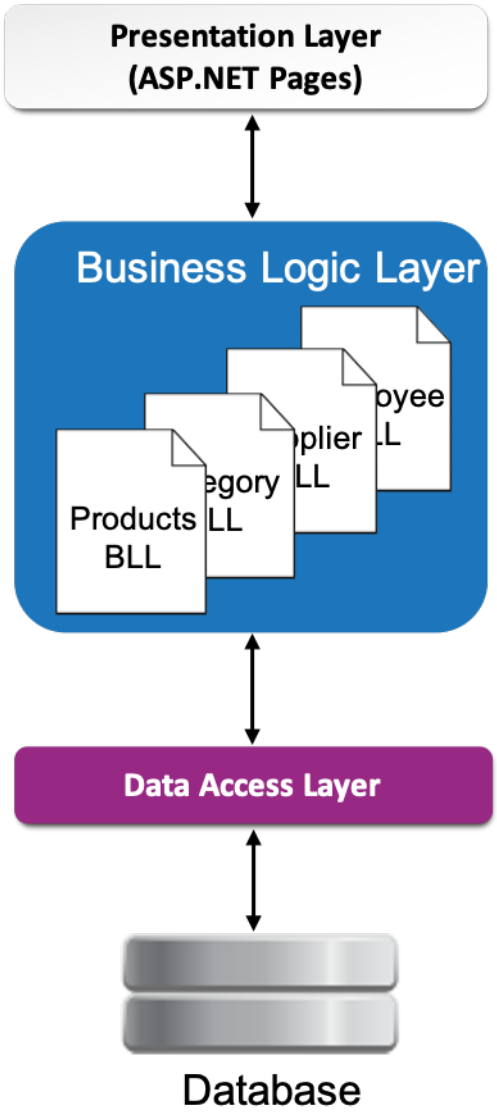


Figure 5-4: Web service business logic.

Any application must have its own logic, application logic that is implemented by the code. The application logic governs how the app functions, what inputs it accepts, how those inputs are processed and coupled with other data, and the outputs it generates. Business apps must translate business logic into application logic so that the app will accurately adhere to business rules so the app can perform tasks and provide services needed by the business. Apps that utilize web services are no different.

There are many theories and approaches to designing business and app logic. One of the most common is to create a workflow which documents each step in the business process, and then adds overlaying app components such as user interface, databases, and communications between users and the app, and between the app and other apps and data sources.

SOAP

Most web services use the Simple Object Access Protocol (SOAP) API, which is a standards-based messaging protocol for connecting to apps and utilizing their exposed functionality, or Representational State Transfer (REST) API. SOAP has been around the longest, has some shortcomings, but remains widely used today.

Microsoft created SOAP in the late 1990s to replace communication protocols such as Distributed Component Object Model (DCOM), and Common Object Request Broker Architecture (COBRA) that relied on binary messaging. These were replaced with a protocol that used XML and would work better for Internet-connected apps that contained built-in error handling. The Internet Engineering Task Force (IETF) standardized the first SOAP API in 1999. The SOAP API is highly modular and extensible allowing developers to use only the components they require.

The primary difficulty with SOAP is that the XML used for sending and receiving requests can be complex, and, depending on the programming language being used, it may need to be created manually. For example, in JavaScript working with SOAP is extremely cumbersome, as even simple tasks require the developer to create the XML structure for each task. This is time consuming, increases code complexity, and makes initial deployment, maintenance, and troubleshooting more difficult.

REST

The Representational State Transfer (REST) API was designed to address the shortcomings of SOAP. REST was created to be a lightweight alternative to SOAP that would be easier for developers to use. With REST, requests are made using a URL. REST uses the GET, POST, PUT, and DELETE verbs from HTTP 1.1. REST also doesn't have to use XML to respond, instead it can respond with data in Command Separated Value (CSV) format, JavaScript Object Notation (JSON), and Really Simple Syndication (RSS) format. This gives developers wide latitude to get the response in a format that is easy for them to process by the programming language they're using. Web services built using REST are called RESTful web services.

SOAP and REST Compared

Both SOAP and REST are widely used in web service creation and programming today. If you're starting a new project, you can choose which protocol to use for messaging. However, if you start working on an existing project, you will likely have to work with the choices already made.

The following is a comparison of advantages between SOAP and REST.

Issue	*SOAP*	*REST*
Standardized	Yes	Yes
Transport	Transport independent	Requires HTTP
Supports distributed enterprise environments	Yes	No, assumes direct point-to-point communication
Modular, pre-built extensibility	Yes	No
Built-in error handling	Yes	No
Automated code creation in some languages	Yes	No
Requires repetitive XML structure creation for message and response in some languages	Yes, SOAP uses XML for all messages	No, REST can use smaller message formats
Learning curve for developers	Larger	Smaller
Design philosophy	Built to replace pre-web technologies	Similar to web technologies

Django

A ***framework*** is a code library that includes reusable code and extensions for common operations, as well as development tools, and a community of developers to get help from. All of this makes it easier to build web apps that are reliable, scalable and easy to maintain. Django and Flask are two of the most popular frameworks for building RESTful web services with Python.

Frameworks often differ in their approach. For example, there are full stack frameworks such as Django that include tools to help developers build large and complex web apps and accomplish common web tasks without needing to use third-party tools and libraries. Django includes libraries to help with things like user authentication, URL routing, and database tasks. It provides a built-in template engine and Object-Relational Mapping (ORM) system to make working with databases easier, and a bootstrapping tool that allows developers to start building their web app without external input.

Flask

On the other end of the framework spectrum are lightweight frameworks, sometimes called microframeworks such as Flask, that don't include everything that Django does, but are extensible via third-party tools and libraries. Flask is simpler and requires minimal framework setup. It's generally easier for developers to get started and get results with Flask because there is less deface code required to wire up the framework. Additionally, Flask is considered by many to be more "Pythonic" than Django because Flask code is more explicit.

Django and Flask Compared

Choosing which web app framework to use depends on your project requirements. The following table is a comparison of features and approaches between Django and Flask.

Feature/Issue	*Django*	*Flask*
Admin Interface	Provides a ready-for-use admin framework which can be customized and makes projects easier to administer. Automatically generates a functional admin module based on the project model.	No built-in admin interface, but developers can add the Flask-Admin extension to add admin interfaces to apps. This extension is inspired by the Django admin interface.
Template Engine (Web templating)	Django comes with a built-in template engine.	Flask leverages the Jinja2, full-featured Python template engine.
Built-in bootstrapping tool	Yes	No
Project approach	Allows developers to divide a project into multiple apps, allows developers across a team to work on individual apps which can be integrated into the app as a whole.	Each project is a single application. Developer can add multiple modules and views to the app.
Object-Relational Mapping support	Includes a robust, built-in ORM system.	Relies on third-party ORMs such as the well-known and widely SQLAlchemy.

Feature/Issue	Django	Flask
Flexibility	Django includes many built-in modules that developers can use without the need for third-party add-ons. However, developers can't modify Django modules, potentially locking them into limitations of those modules.	Flask is extensible, while developers must investigate and select the best third-party extensions, they can choose those that best fit their app requirements.
Common usage scenarios	Larger, complex web apps.	Rapid development of simpler web apps.

Resource Access through Endpoints

Endpoints, also called service endpoints, are URLs that point to a specific resource required by an app. Similar to sockets discussed earlier, endpoints allow web service apps to send and receive data over the network for communication. Endpoints include the URL and service name of the resource. In RESTful web services, these connections are made using REST APIs or HTTP, but other protocols may be used. It's also possible for the same resource to have multiple service endpoints, and to have multiple services with the same endpoint, as requests are received at a front-end gateway or service, and then routed to service endpoints as appropriate after reception.

Guidelines for Creating a RESTful Web Service

Follow these guidelines when creating a RESTful web service.

Create a RESTful Web Service

When creating a RESTful web service:

- Choose a web service API that best suits the programming language being used and other application requirements.
- Choose a web development framework that best suits the size and approach to the project.
- Document the business logic of the process your app will be a part of, then overlay application elements and communications between those elements to create the app logic for your web service.

ACTIVITY 5-2
Creating a RESTful Web Service

Data Files
All project files in C:\094022Data\Creating and Securing a Web Service-Connected Application \WWProject-L5.

Before You Begin
PyCharm is open.

Scenario
Woodworkers Wheelhouse wants to move some of their services and data to the cloud. This includes the databases you use for the Wood Delivery Calculator app. You will need to modify the app to access the data in the cloud. You decide to create a web service that allows you to access the databases. You will use Flask to do this.

---

1. Install Flask and Flask-RESTful.
   a) Open **Command Prompt**.
   b) In the **Command Prompt**, type the following and press **Enter** to install Flask.

```
C:\Users\user1>pip install C:\094022Data\Packages\Flask-1.1.2-py2.py3-none-any.whl
```

Note: For this and the next lesson, you are installing specific versions of packages included as local files on your computer. This is to reduce any compatibility issues that may arise in future versions of these packages. You will see a message about a more recent version of pip being available. You can ignore this message.

   c) Type the following and press **Enter** to install Flask-RESTful.

```
C:\Users\user1>pip install C:\094022Data\Packages\Flask_RESTful-0.3.8-py2.py3-none-any.whl
```

   d) Minimize but do not close the Command Prompt.

2. Open the **WWProject-L5** project.
   a) In the Welcome to PyCharm window, select **Open**.
   b) Navigate to the **C:\094022Data\Creating and Securing a Web Service-Connected Application \WWProject-L5** folder.
   c) With the **WWProject-L5** folder selected, select **OK**.
   d) In the **Project** pane on the left side of the PyCharm window, verify that **WWProject-L5** is listed.

3. Create the **wdc-webservice.py** file in the **WWProject-L5** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L5** project, and select **New→Python File**.
   b) In the **Name** box, type ***wdc-webservice*** and press **Enter**.

4. Create the web service.

a) In **wdc-webservice.py**, starting on line 1, type the following to import the libraries used in the web service.

These statements:

- Import the Flask and Flask-RESTful libraries which will be used to create the web service.
- Import the MySQL Connector library which will be used to connect and read and write to the databases.
- Import the `datetime` library which will be used to obtain the current date.
- Import the `decimal` library which will be used to convert the order total to a decimal data type.

Note: If you receive a pop-up message about PyCharm Professional Edition you can close it.

b) After the `import` methods, starting on line 7, type the following to create the Flask app and the API object.

5. Retrieve product information from the **wwproducts** database.

a) Starting on line 11, type the following to define the `Products` subclass of the `Resource` class.

These statements:

- Define the `Products` subclass of the `Resource` class. Products will be used in the URL to access this class.
- Define the `get` function for this class. This will handle a `GET` method from the client for Products.

b) Starting on line 15, type the following to connect to the **wwproducts** MySQL database.

```
13      def get(self):
14
15          self.mydb = mysql.connector.connect(
16              host="localhost",
17              user="root",
18              password="P@ssw0rd12!",
19              database="wwproducts")
20
21          self.mycursor = self.mydb.cursor()
22
```

c) Starting on line 23, type the following to retrieve the products from the products table.

```
21          self.mycursor = self.mydb.cursor()
22
23          self.products = []
24          self.mycursor.execute("SELECT product, price FROM products")
25          self.row = self.mycursor.fetchone()
26          while self.row is not None:
27              self.products.append(self.row[0])
28              self.row = self.mycursor.fetchone()
29          self.mydb.close()
30          return jsonify({'products':self.products})
31
```

These statements:

- Submit a `SELECT` statement to retrieve the products from the products table.
- Using a `while` statement to add the product values from the table into products lists.
- Close the database connection.
- Serializes the products list to JSON format, wraps it in a Response object, and returns it to the client.

d) Starting on line 32, type the following to create the `Prices` subclass that will process the `GET` method for product prices.

```
30          return jsonify({'products':self.products})
31
32  class Prices(Resource):
33      """Get price data from wwproducts database."""
34      def get(self):
35
36          self.mydb = mysql.connector.connect(
37              host="localhost",
38              user="root",
39              password="P@ssw0rd12!",
40              database="wwproducts")
41
42          self.mycursor = self.mydb.cursor()
43
44          self.prices = []
45          self.mycursor.execute("SELECT product, price FROM products")
46          self.row = self.mycursor.fetchone()
47          while self.row is not None:
48              self.prices.append(float(self.row[1]))
49              self.row = self.mycursor.fetchone()
50          self.mydb.close()
51          return jsonify({'prices':self.prices})
52
```

Note: On line 48, we convert the price value from the database to a float because jsonify cannot encode the decimal type.

These statements are the same as the ones for Products but they retrieve the product price data instead.

6. Insert order information into the **wworders** database.

a) Starting on line 53, type the following to define the `Submit` subclass of the `Resource` class.

```
51          return jsonify({'prices':self.prices})
52
53  class Submit(Resource):
54      """Insert order data to wworders database."""
55      def post(self):
56          postship = request.json.get('ship')
57          posttotal = Decimal(request.json.get('total'))
58
```

These statements:

- Define the `Submit` subclass of the `Resource` class. `Submit` will be used in the URL to access this class.
- Define the `post` function for this class. This will handle a `POST` method from the client for `Submit`.
- Assign the ship value submitted with the `POST` request from the client app to a variable. When the client uses the `POST` method, it will send the values of the ship method and the order total.
- Assign the total value submitted with the `POST` request from the client app to a variable.

b) Starting on line 59, type the following to connect to the **wworders** MySQL database.

```
57          posttotal = Decimal(request.json.get('total'))
58
59          self.mydb = mysql.connector.connect(
60              host="localhost",
61              user="root",
62              password="P@ssw0rd12!",
63              database="wworders")
64
65          self.mycursor = self.mydb.cursor()
66
```

c) Starting on line 67, type the following to insert the order information to the ordersummary table.

```
65          self.mycursor = self.mydb.cursor()
66
67          sql = ("INSERT INTO ordersummary "
68                 "(orderdate, shipmethod, totalprice) "
69                 "VALUES (%s, %s, %s)")
70          val = (date.today(), postship, posttotal)
71
72          self.mycursor.execute(sql, val)
73          self.mydb.commit()
74          self.mydb.close()
75
76          return jsonify(self.mycursor.lastrowid)
77
```

These statements:

- Use a variable for the parameterized `INSERT` query to insert the order date, ship method, and total price values to the ordersummary table.
- Use a variable for the order date, ship method, and total price values.
- Use the `execute()` method to insert the values to the table.
- Use the `commit()` method to commit the current insert transaction to the table.
- Closes the database connection.
- Serializes the lastrowid to JSON format, wraps it in a Response object, and returns it to the client. This is used as the order number and will be displayed in the client window.

d) Starting on line 78, type the following to add defined resources to API and run the web service.

```
76          return jsonify(self.mycursor.lastrowid)
77
78      # add the defined resources along with their corresponding urls.
79      api.add_resource(Products, '/products')
80      api.add_resource(Prices, '/prices')
81      api.add_resource(Submit, '/submit')
82
83      app.run()
84
```

These statements:

- Add each defined resource with its corresponding URL to the API.
- Run the web service app.

7. Run the program and watch the results.

a) Run **wdc-webservice.py**.

Note: At some point, you may be prompted by your local firewall service to allow the application to communicate over the network. If this happens, select the option to allow it.

In the console window, you will see output from the server. There should be no errors. When you run the client app later, there will be more information to see here.

b) Select the **Stop 'wdc-webservice'** button.

c) In the **Run** console pane, close the **wdc-webservice** tab.

TOPIC C

Create a Web Service Client

Once you've created a web service for your app to communicate with clients, you must build the client that connects to it. In this topic, you will build a web service client.

RESTful Web Service Client Requests

When creating RESTful web service clients, you must utilize and accommodate the following key request elements.

Request Element	*Description*
Request verbs	These are commands passed from the client to the web service to perform actions and receive data from the web service. They include the standard HTTP GET, POST, PUT, and DELETE commands.
Request headers	These are additional instructions that are sent with a request and may include authorization information, or the type of response that is required for the request.
Request body	The data sent with the request makes up the request body. This is normally sent with a POST command, when the client is requesting specific data be submitted to the service. For example, this might be user data, or additional information such as a delivery notification associated with an order created when a delivered package is scanned on arrival to its destination.
Response body	This is the main part of the response and is typically the information initially requested being returned. This data may be in the form of an XML document.

Request Methods

RESTful web services support the following methods in requests, which are based on HTTP 1.1 commands.

Command	*Action and Usage in an Online Order App*
POST	Submits data to the web service to be added to the underlying data source. An online order app might use the POST command when a new order is created.
GET	Requests data from the web service. An online order app might use a GET request when a customer requests a status of an order or tracking information for an order in transit.
PUT	Generally used to update an existing item. PUT is often confused with POST as both can be used to create new items. An online order app might use the PUT command when a customer updates an order to change item quantities or alter the shipping speed.
DELETE	Used to delete a resource.

Request Method Examples

The following are examples of using request methods in apps:

GET request example:

The following code sends a GET request to the web service for the product prices. The code:

- Sends a GET request to the URL for the localhost on port 5000. The /prices portion of the URL maps to the Prices subclass in **wdc-webservice.py**.
- Converts the JSON results of the request to text.
- Assigns the text to a list.

```
getprice = requests.get("http://127.0.0.1:5000/prices")
tmpprices = json.loads(getprice.text)
self.prices = tmpprices["prices"]
```

POST request example:

The following code sends a POST request to the web service with the order information. The code:

- Assigns the selected ship method and order total to variables.
- Assigns the data to send with the POST request to a variable that defines a key for each value being sent.
- Sends a POST request to the URL for the localhost on port 5000. The /submit portion of the URL maps to the Submit subclass in **wdc-webservice.py**.

```
ship=self.ship.get()
total=float(self.total)
send={'ship': ship, 'total': total}
posttotal = requests.post("http://127.0.0.1:5000/submit", json = send)
```

Guidelines for Creating a Web Service Client

Follow these guidelines when creating a web service client.

Create a Web Service Client

When creating a web service client:

- When trying to adhere to RESTful web service standards, use the REST API over HTTP/(S).
- Use the appropriate request method for client-to-web service interactions.
- Document which request methods are used for each communication when documenting your app and business logic.

ACTIVITY 5-3
Creating a Web Service Client

Before You Begin
The **WWProject-L5** project is open in PyCharm.

Scenario
Now that you have created the web service, you now need to update the Wood Delivery Calculator app to be a client that connects to the web service. You will use the requests library to enable this.

---

1. Install requests.
   a) Restore the Command Prompt.
   b) Type the following and press **Enter** to install requests.

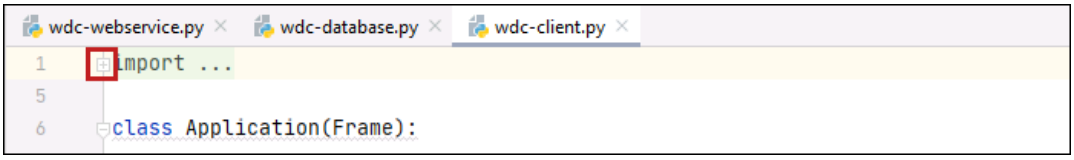
```
C:\Users\user1>pip install C:\094022Data\Packages\requests-2.25.1-py2.py3-none-any.whl
```

   c) Minimize but do not close the Command Prompt.

2. Copy the **wdc-database.py** file to **wdc-client.py**.
   a) Open the **wdc-database.py** file in the editor.
   b) Select **File→Save As**.
   c) In the **Copy** dialog box, change the file name to ***wdc-client.py*** and then select **OK**.
   d) Ensure that the **wdc-client.py** file is open, and that it is the active document to edit.

Note: You are creating a client app to access the web service, but the web service could also be accessed through a browser using HTML, etc.

3. Update the constructor to read the product and shipping data from the database.
   a) If the import lines have been collapsed, then select the **+** sign to expand them.



b) Above the class declaration, starting on line 4, remove the line for the **datetime** import and add the following `import` statements for **requests** and **json** above the class declaration.

```
wdc-webservice.py | wdc-database.py | wdc-client.py
1  from tkinter import *
2  import mysql.connector
3  from decimal import *
4  import requests
5  import json
6
7  class Application(Frame):
```

c) Starting on line 18, remove the code that connects to the MySQL database and selects product and price information from the products table.

```
14         self.shipdays = {"1 Day": 20.00,
15                          "2 Days": 15.00,
16                          "3 Days": 10.00}
17
18         self.mydb = mysql.connector.connect(
19             host="localhost",
20             user="root",
21             password="P@ssw0rd12!",
22             database="wwproducts")
23
24         self.mycursor = self.mydb.cursor()
25
26         self.mycursor.execute("SELECT product, price FROM products")
27         self.row = self.mycursor.fetchone()
28         while self.row is not None:
29             self.products.append(self.row[0])
30             self.prices.append(self.row[1])
31             self.row = self.mycursor.fetchone()
32         self.mydb.close()
33
34         self.lblproduct = list(self.products)
```

d) Below the **self.shipdays** dictionary definition code, starting on line 18, type the following to send a GET request to the web service for the product names.

```
14         self.shipdays = {"1 Day": 20.00,
15                          "2 Days": 15.00,
16                          "3 Days": 10.00}
17
18         getproduct = requests.get("http://127.0.0.1:5000/products")
19         tmpproducts = json.loads(getproduct.text)
20         self.products = tmpproducts["products"]
21
22         self.lblproduct = list(self.products)
```

These statements:

- Send a GET request to the URL for the localhost on port 5000. The /products portion of the URL maps to the Products subclass in **wdc-webservice.py**.
- Convert the JSON results of the request to text.
- Assign the text to a list.

e) Following the code you just typed, starting on line 22, type the following to send a `get` request to the web service for the product prices.

```
18          getproduct = requests.get("http://127.0.0.1:5000/products")
19          tmpproducts = json.loads(getproduct.text)
20          self.products = tmpproducts["products"]
21
22          getprice = requests.get("http://127.0.0.1:5000/prices")
23          tmpprices = json.loads(getprice.text)
24          self.prices = tmpprices["prices"]
25
26          self.lblproduct = list(self.products)
27          self.lblprice = list(self.prices)
```

These statements:

- Send a GET request to the URL for the localhost on port 5000. The /prices portion of the URL maps to the Prices subclass in **wdc-webservice.py**.
- Convert the JSON results of the request to text.
- Assign the text to a list.

f) In the `submit` function, starting on line 130, remove the code that connects to the MySQL database and inserts data to the ordersummary table.

```
128     def submit(self):
129         """Insert order information to wworders database"""
130         self.mydb = mysql.connector.connect(
131             host="localhost",
132             user="root",
133             password="P@ssw0rd12!",
134             database="wworders")
135
136         self.mycursor = self.mydb.cursor()
137
138         sql = ("INSERT INTO ordersummary "
139                "(orderdate, shipmethod, totalprice) "
140                "VALUES (%s, %s, %s)")
141         val = (date.today(), self.ship.get(), self.total)
142
143         self.mycursor.execute(sql, val)
144         self.mydb.commit()
145         self.mydb.close()
146         self.lblsubmit.set("Order number "
147                            +str(self.mycursor.lastrowid)+" submitted")
148
```

g) In the `submit` function, starting on line 130, type the following to send a `post` request to the web service for the order information.

```
128     def submit(self):
129         """Insert order information to wworders database"""
130         ship=self.ship.get()
131         total=float(self.total)
132         send={'ship': ship, 'total': total}
133         posttotal = requests.post("http://127.0.0.1:5000/submit", json = send)
134         self.lblsubmit.set("Order number "
135                            +str(self.mycursor.lastrowid)+" submitted")
136
```

These statements:

- Assign the selected ship method and order total to variables.
- Assign the data to send with the POST request to a variable that defines a key for each value being sent.
- Send a POST request to the URL for the localhost on port 5000. The /submit portion of the URL maps to the Submit subclass in **wdc-webservice.py**.

h) On line 135, update the **self.lblsubmit.set** statement to use the text returned from the web service.

```
128     def submit(self):
129         """Insert order information to wworders database"""
130         ship=self.ship.get()
131         total=float(self.total)
132         send={'ship': ship, 'total': total}
133         posttotal = requests.post("http://127.0.0.1:5000/submit", json = send)
134         self.lblsubmit.set("Order number "
135                            +str(posttotal.text)+" submitted")
136
```

The text returned from the POST request is the order id used in the ordersummary table.

i) In the `calculate` function, on line 115, update the code to use `decimal` module to convert **self.prices** from a float to avoid error when performing a calculation with other decimal numbers.

```
110     def calculate(self):
111         """Event handler for the calculator"""
112         self.total = 0
113         for i in range(len(self.products)):
114             if (self.entry[i].get()):
115                 self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
116
117         if (self.ship.get() == "1 Day"):
```

The text returned from POST request is the order id used in the ordersummary table.

4. Run the **wdc-webservice.py** web service and **wdc-client.py** client app.

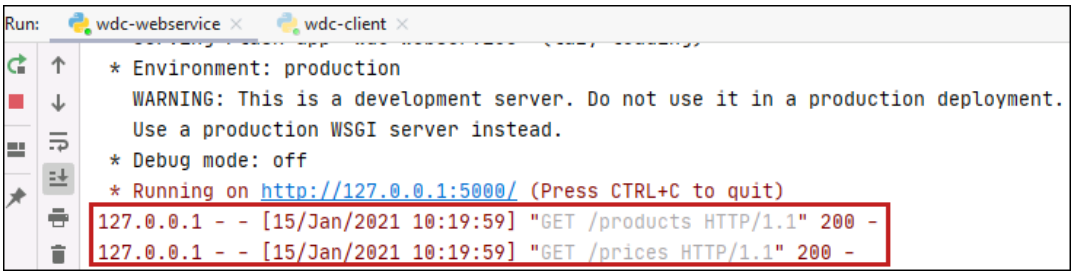
a) Run the **wdc-webservice.py** script.

b) Run the **wdc-client.py** script.

c) The Wood Delivery Calculator window should appear, with the list of products, prices, and shipping options.

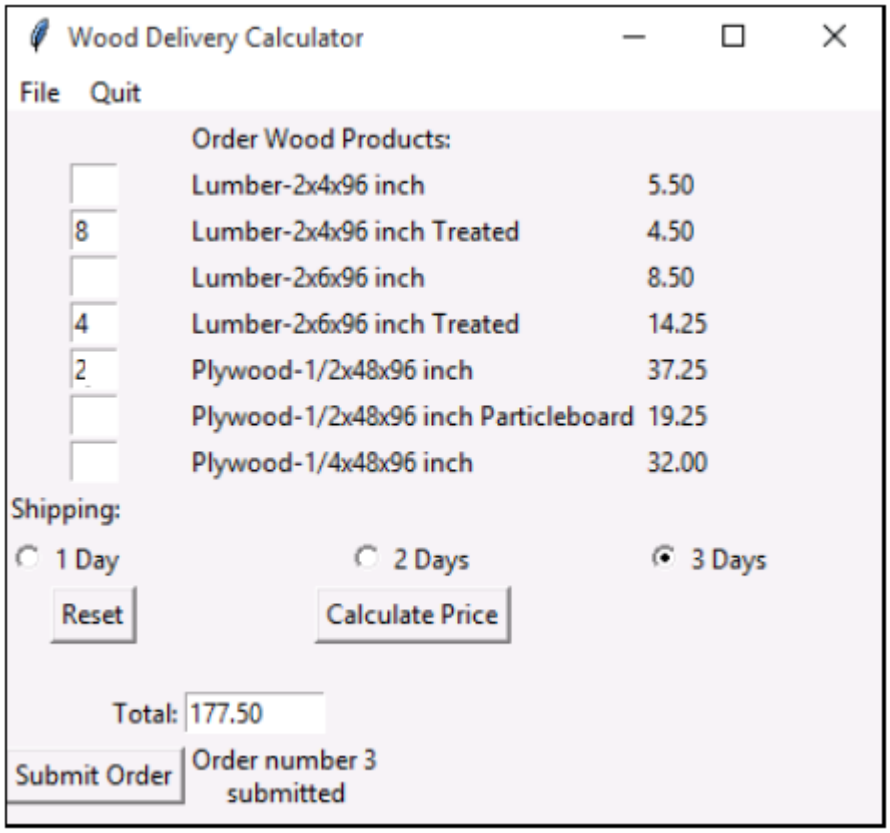
5. Test the client and web service connection.

a) In the console window, select the **wdc-webservice.py** tab and view the server output.



Observe that the web service received two GET requests. One for products and one for prices.

b) In the Wood Delivery Calculator window, specify the number of products and shipping method to match the image below, and select **Calculate Price** and then **Submit Order**.



c) In the console pane, in the **wdc-webservice.py** tab, observe that the web service received one POST request for submit.

6. Close running apps.
   a) Close the Wood Delivery Calculator window.
   b) Select the **Stop 'wdc-webservice'** button.

   c) In the **Run** console pane, close all tabs.

TOPIC D

Secure Connected Applications

Apps connected to the Internet may be subject to numerous types of attacks designed to take over the app or steal information. As developers, you should always strive to write secure code and secure the app and web surfaces it runs through. In this topic, you will secure connected apps.

Web App Security Vulnerabilities

Web apps written with any language are subject to a wide range of cyber attacks that developers and development teams must mitigate, including those listed in the following table.

Cyber Attack	*Description*	*Mitigation Steps*
SQL Injections	Modify SQL direct queries to database to steal sensitive data.	Use parameterized queries that escape text values using the mysql.connector %s placeholder.
Cross Site Scripting (XSS)	Inject arbitrary HTML and JavaScript into your website context. When users visit the site or app, they execute the code which will perform malicious activity.	Parameterize text in your Python app. By default, Flask configures Jinja2 web template engine to escape all values unless instructed otherwise.
Cross Site Request Forgery (CSRF)	Forces execution of unwanted action after a logged in user is compromised, sometimes with the help of a social engineering attack by being sent a link in email or chat.	Use the Flask-WTF library to control upload of files and ItsDangerous library to send cryptographic communications to untrusted environments.
LDAP (Lightweight Directory Access Protocol) Injections	Modify directory queries to grant unauthorized queries and obtain user or directory data and modify content.	Always sanitize incoming data, parameterize queries, and escape text values.
Code and Command Injections	Python code is sent through an app to the Python interpreter and used to execute commands on the operating system to gain access or perform malicious acts.	Always sanitize incoming data, parameterize queries, and escape text values.
XPathi / XPath Injection	When app constructs XPath queries for XML with user-supplied data to determine how XML data is structured, gain access to sensitive data, or elevate privileges.	Always sanitize incoming data, parameterize queries, and escape text values.

As you can see, as a developer, your primary defense for many of these attacks is to sanitize incoming data, parameterize queries, and escape text values.

Note: Be aware that the term "escape text" is sometimes used in place of the term "parameterize queries".

Additional Information

LDAP Injection prevention checklist: **https://cheatsheetseries.owasp.org/cheatsheets/LDAP_Injection_Prevention_Cheat_Sheet.html**

LDAP Injection good and bad example: **https://rules.sonarsource.com/python/RSPEC-2078**

Python Code Injection sample: **https://sethsec.blogspot.com/2016/11/exploiting-python-code-injection-in-web.html**

XPath Injection good and bad example: **https://rules.sonarsource.com/python/RSPEC-2091**

Security with Flask and Python

Flask supports many security libraries and extensions to secure your apps. In the following example, Flask provides basic HTTP authentication for login:

```
#Add basic HTTP authentication to the flask web service.
from flask_httpauth import HTTPBasicAuth

#Define the basic authentication and hard code a user and password that will
allow access to the web service.
#This example uses a plain text password stored in the app for simplicity.
#This is not secure. In a real environment, hash passwords and store them in a
separate database.

auth = HTTPBasicAuth()

user_data = {
    "client": "P@ssw0rd12!"
}

#The following code verifies the password from the client app.

    @auth.verify_password   #This callback function verifies that the username
and password combination are valid.
    def verify(username, password):
        if not (username and password):
            return False    #If the username and password from client don't
match then a false value is returned.
        return user_data.get(username) == password
    @auth.login_required    #This callback function prevents further
processing if the username and password are not valid.
```

Flask Security Libraries and Extensions

The following table lists web app security task requirements and the Flask extension or libraries that support it.

Web App Security Task	*Flask Extension/Library*
Session-based authentication	Flask-Login and Flask-WTF
Role management	Flask-Principal

Web App Security Task	*Flask Extension/Library*
Password hashing	passlib
Basic HTTP authentication	Flask-HTTPAuth or single method decorator
Token-based authentication	Flask-Login, or `include_auth_token`
Token-based account activation (optional)	Flask-Login, Flask-WTF, Flask-Mail
Token-based password recovery / resetting (optional)	ItsDangerous
Two-factor authentication (optional, alpha release)	Passlib and PyQRCode
Email confirmation links	Flask-Mail
Unified sign in (optional)	Flask-Login
User registration (optional)	Flask-WTF
Login tracking (optional)	Flask-Login
JSON/Ajax Support	Supported by Python for Login, registration, change password, confirmation, forgot password, and passwordless login requests.

Additional Information

Flask Security Features: **https://flask-security-too.readthedocs.io/en/stable/**

Flask Security: **https://flask-security-too.readthedocs.io/en/stable/**

Flask Security Libraries

Flask-Login: Login and session support: **https://flask-login.readthedocs.io/en/latest/**

Flask-Principal: Authentication providers and role management: **https://pythonhosted.org/Flask-Principal/**

Flask-WTF: CSRF protection, Forms, file upload, and reCAPTCHA support: **https://flask-wtf.readthedocs.io/en/stable/**

Flask-Mail: Send emails from web apps: **https://pypi.org/project/Flask-Mail/**

itsdangerous: Cryptographically pass data to untrusted environments and to get it back safely: **https://pypi.org/project/itsdangerous/**

Passlib: Python password hashing library: **https://passlib.readthedocs.io/en/stable/**

PyQRCode: Two-factor authentication and QR Code generator: **https://pypi.org/project/PyQRCode/**

Password Hashing with the Passlib

Hashing is the act of encrypting a password text string. The Passlib library for Python provides more than 30 password hashing algorithms and a framework for managing password hashes. It can perform a range of tasks including verification of a password hash in the /etc/shadow folder to encrypting user passwords for user-focused apps.

This example hashes then verifies a password using the PBKDF2-SHA256 algorithm. When the correct password is verified against the hash, the result returns true, when the incorrect password is verified against the hash, the result is false.

```
>>>#Import the hash algorithm
>>>from passlib.hash import pbkdf2_sha256
```

```
>>>#Generate new salt, and hash a password
>>>hash = pbkdf2_sha256.hash("mysecretpass")
>>>hash
'$pbkdf2-sha256$29000$N2YMIWQsBJSDUIKILWLKSLkuQ$1t8iyB2A.WF/
Y4FZv.lfCIhXXN33N23OSgQYThBYRfk'

>>>#verifying the password
>>>pbkdf2_sha256.verify("mysecretpass", hash)
True
>>>pbkdf2_sha256.verify("joshua", hash)
False
```

Additional Information

For additional information on Passlib, see: **https://passlib.readthedocs.io/en/stable/**.

Input Validation for Web Service Forms

If users will send data to your web service app, it's an important security task to validate data that is captured in web forms to make sure the data is correct based on data requirements and to validate security. The Flask-WTF extension adds functions to validate form data entry.

The following represents what might be a typical registration form requiring a user name, email address, and password which can be validated by Flask-WTF. User ID and email string limits are enforced, and passwords submitted are verified to match and terms of service are accepted:

```
from wtforms import Form , BooleanField , StringField , PasswordField ,
validators

class RegistrationForm ( Form ):
    userid = StringField ( 'User ID' , [ validators.Length ( min = 8 , max =
30 )])
    email = StringField ( 'Email Address' , [ validators.Length ( min = 2 ,
max = 35 )])
    pass = PasswordField ( 'New Password' , [
        validators.DataRequired (),
        validators.EqualTo ( 'confirm' , message = 'Passwords must match' )
     ])
     confirm = PasswordField ( 'Repeat Password' )
     accept_tos = BooleanField ( 'I accept the TOS' ,
[ validators.DataRequired ()])
```

The view function would look like this:

```
@app.route ('/register' , methods = [ 'GET' , 'POST' ])
def register ():
    form = RegistrationForm ( request.form )
    if request.method == 'POST' and form.validate ():
    user = User ( form.username.data , form.email.data ,
                  form.password.data)
    db_session.add(user)
    flash('Thanks for registering')
    return redirect( url_for( 'login'))
    return render_template('register.html', form=form)
```

Security Tools

There are a number of security tools (some free and some fee-based) that you can use to improve the security of your code and apps. Some of them are listed in the following table.

Tool	Description
Bandit	Free AST-based static code analyzer that finds common security issues in Python code by processing each file, building an AST and running plug-ins against the AST, then generating a report.
Python Taint (PyT)	Free static analysis tool that identifies command injection, XSS, SQLi, interprocedural, path traversal HTTP attacks in Python web apps.
Tinfoil Security	Fee-based website scanner used to find security vulnerabilities in websites and apps and provide recommended fixes.
Spaghetti Security Scanner	Free web app scanner that detects default files, misconfigurations, and insecure files and supports multiple web app frameworks.
GuardRails	Fee-based static code analysis and vulnerable dependence analysis for Python and other languages for dev teams. Continuously scans code in repositories like GitHub and provides notification, reporting, and escalation of issues.
Safety	Free dependency analyzer that warns you of insecure app dependencies.
Hawkeye	Free vulnerability, security, and risk highlighting tool.
HubbleStack	Free security and compliance auditing framework built by Adobe that provides on-demand, profile-based auditing, real-time security event notifications, alerting, and reporting.
Secure.py	Adds optional security headers and cookies attributes for Python web frameworks including Flask and Django.

Additional Information

Bandit: **https://pypi.org/project/bandit/**

Python Taint (PyT): **https://github.com/python-security/pyt**

Tinfoil Security: **https://www.tinfoilsecurity.com**

Spaghetti Security Scanner: **https://github.com/asifurrouf/Security_Spaghetti**

GuardRails: **https://github.com/apps/guardrails**

Safety: **https://pyup.io/docs/**

Hawkeye: **https://github.com/hawkeyesec/scanner-cli**

HubbleStack: **https://hubblestack.readthedocs.io/en/latest/**

Secure.py: **https://secure.readthedocs.io/en/latest/**

Guidelines for Securing Connected Applications

Follow these guidelines when securing connected applications.

Secure Connected Applications

When securing connected applications:

- Always code with threats in mind, and code using best practices to create secure apps.
- Always sanitize incoming data, parameterize queries, and escape text values.
- Use website and code vulnerability scanners to find potential security issues in websites, apps, and code.
- Use the Python Transport Layer Security/Secure Sockets Layer (TLS/SSL) wrapper library (**ssl.py**) to encrypt messages sent over the communication channel.
- Verify that you are communicating only with those you intend to communicate with. For example, you could maintain a list of IP addresses that are allowed to connect to the server (a

whitelist), and write the server code to accept connections only from IP addresses in that list. The server can call `socket.getpeername()` to determine the client's IP address. Make sure that the list of acceptable addresses is stored in a secure location that can't be modified by unauthorized users or software processes.

- For security monitoring, use the Python `logging` module to log network connections and attempts to connect as they occur.
- Don't permit open-ended commands (such as SQL statements, Python commands, or other scripting/programming code) to be transmitted between the client and server as a means of remote control, as this increases the possibility of illicit instructions being provided to the server. Use a very limited set of commands to provide messaging between the client and server.

ACTIVITY 5-4
Securing Connected Applications

Before You Begin
The **WWProject-L5** project is open in PyCharm.

Scenario
You are concerned that with your data now accessible from a public web service, it is vulnerable to being accessed by unauthorized users. You decide to implement basic HTTP authorization to make the web service more secure.

---

1. Install Flask-HTTPAuth.
   a) Restore the Command Prompt.
   b) Type the following and press **Enter** to install Flask-HTTPAuth.

```
C:\Users\user1>pip install C:\094022Data\Packages\Flask_HTTPAuth-4.2.0-py2.py3-none-any.whl
```

   c) When the install is complete, close the Command Prompt.

2. Add basic HTTP authentication to the Flask web service.
   a) In **wdc-webservice.py**, above the class declaration, on line 6, add the following `import` statement for `flask_httpauth`.

wdc-webservice.py | wdc-database.py | wdc-client.py

```
1  from flask import Flask, jsonify, request
2  from flask_restful import Resource, Api
3  import mysql.connector
4  from datetime import date
5  from decimal import *
6  from flask_httpauth import HTTPBasicAuth
7
8  # create the flask app and API object.
```

b) Below the code to create the Flask app and the API object, starting on line 11, type the following to define the basic authentication and hard code a user and password that will allow access to the web service.

```
8   # create the flask app and API object.
9   app = Flask(__name__)
10  api = Api(app)
11  auth = HTTPBasicAuth()
12
13  user_data = {
14      "client": "P@ssw0rd12!"
15  }
16
17  class Products(Resource):
```

Note: We are using a plain text password and storing it in our web service app. This is a convenience to save time for this course. You should not do this in a real environment. You should use a hash for any passwords and consider storing authentication data in a separate database.

c) In the `Products` subclass, starting on line 20, type the following to verify the password from the client app.

```
17  class Products(Resource):
18      """Get product data from wwproducts database."""
19
20      @auth.verify_password
21      def verify(username, password):
22          if not (username and password):
23              return False
24          return user_data.get(username) == password
25      @auth.login_required
26
27      def get(self):
```

These statements:

- Use `@auth.verify_password` callback function to verify that the username and password combination provided by the client are valid.
- Return a false value if the username and password from the client don't match.
- Use `@auth.login_required` callback function to prevent further processing if the username and password are not valid.

3. Add basic HTTP authentication to the client app.

a) In **wdc-client.py**, below the **self.shipdays** dictionary definition code, on line 18, modify the first GET request by adding the username and password to access the web service.

```
14      self.shipdays = {"1 Day": 20.00,
15                       "2 Days": 15.00,
16                       "3 Days": 10.00}
17
18      getproduct = requests.get("http://127.0.0.1:5000/products",auth=('client', 'P@ssw0rd12'))
19      tmpproducts = json.loads(getproduct.text)
20      self.products = tmpproducts["products"]
```

Note that you are using an incorrect password at this time to test if access will be denied by the web service.

b) Following the statement you just revised, starting on line 20, type the following to check the response from the GET request to see if the unauthorized access code is returned.

```
18      getproduct = requests.get("http://127.0.0.1:5000/products",auth=('client', 'P@ssw0rd12'))
19
20      if getproduct.status_code == 401:
21          print(str(getproduct.status_code)+" "+ getproduct.text)
22          exit()
23
24      tmpproducts = json.loads(getproduct.text)
```

- If the 401 status code is returned, then print the code and the error text, and then exit the app.

4. Test using an incorrect password.

a) Run the **wdc-webservice.py** script.
b) Run the **wdc-client.py** script.
c) In the console window, in the **wdc-client.py** tab, observe text stating there was a 401 Unauthorized Access error.

```
Run:  wdc-webservice  wdc-client
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Data
401 Unauthorized Access

Process finished with exit code 0
```

d) In the console pane, close the **wdc-client.py** tab.
e) In the console pane, in the **wdc-webservice.py** tab, observe that the GET request returned the 401 status code.

```
Run:  wdc-webservice
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Data
 * Serving Flask app "wdc-webservice" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployn
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [15/Jan/2021 10:49:52] "GET /products HTTP/1.1" 401 -
```

f) Leave the **wdc-webservice.py** running. Do not stop it.

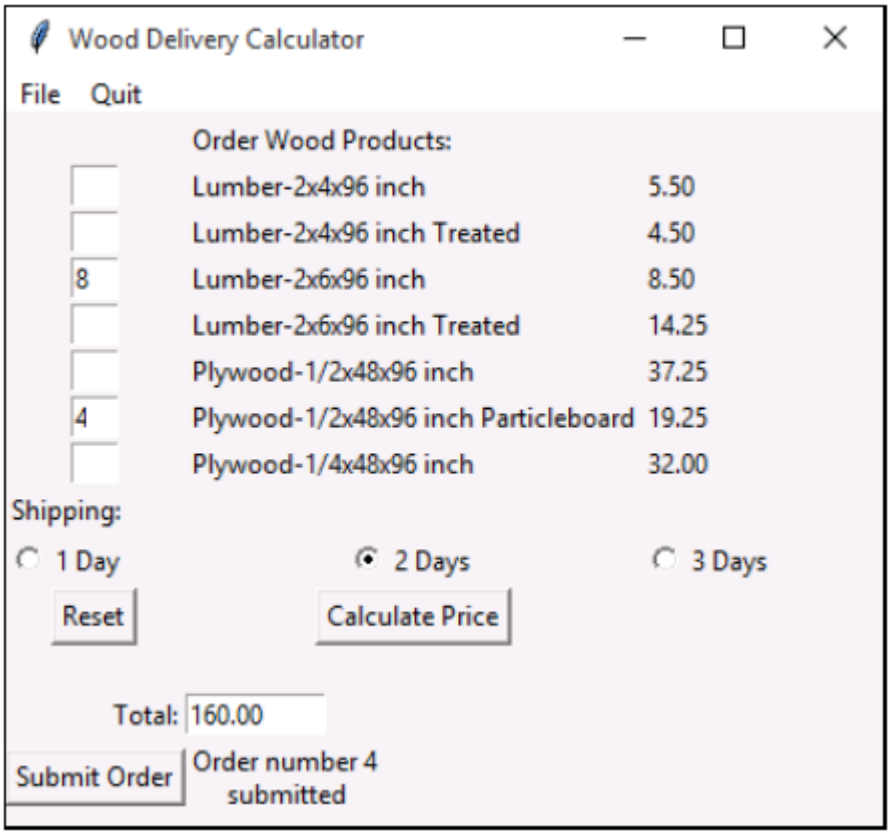
5. Test using the correct username and password.

a) In **wdc-client.py**, below the **self.shipdays** dictionary definition code, on line 18, modify the first GET request by using the correct password to access the web service.

```
16                  "3 Days": 10.00}
17
18      getproduct = requests.get("http://127.0.0.1:5000/products",auth=('client', 'P@ssw0rd12!'))
19
20      if getproduct.status_code == 401:
```

b) Run the **wdc-client.py** script.
The Wood Delivery Calculator window should appear.

c) In the Wood Delivery Calculator window, specify the number of products and shipping method to match the image below, then select **Calculate Price** and then **Submit Order**.



6. Close running apps.

a) Close the Wood Delivery Calculator window.

b) Select the **wdc-webservice** tab, and then select the **Stop 'wdc-webservice'** button.

c) In the **Run** console pane, close all tabs.

d) Select **File→Close Project**.

Summary

In this lesson, you selected a network protocol to use with your client/server web-connected app. You then created a RESTful web service to connect your app to the network. Following that, you created a web service client to connect to your app's web service. Finally, you secured your web-connected app using Flask security libraries and extensions.

Will you use web-connected apps in your environment? Why or why not?

What type of security tools do you think you will use to secure your web-connected apps?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

6 Programming Python for Data Science

Lesson Time: 3 hours

Lesson Introduction

Python® is one of the most popular programming languages for data science and data visualization purposes. In this lesson, you will program Python for data science applications.

Lesson Objectives

In this lesson, you will:

- Clean data with Python.
- Visualize data with Python.
- Perform linear regression with Python.

TOPIC A

Clean Data with Python

To analyze data, it must first be put into a state where it is usable for efficient analysis. In this topic, you will clean data with Python.

Data Science

Data science is the application of domain-based knowledge, mathematical and statistical analysis, as well as programming skills to find patterns, evaluate cause and effect, and make estimations from data. Common data analysis tasks related to data science include:

- Analyzing large amounts of data (also called data mining).
- Combining and correlating large amounts of data including corporate data, local private data, and data from cloud or online sources.
- Evaluating performance of processes including operations, sales, workflows, or any other business-related processes.
- Performing benchmarking to compare business information to baselines or competitor information or industry standards.
- Creating data models by combining and refining data from separate raw data sources to correlate specific data to do focused analysis on a given topic.
- Surfacing trends or insights about processes, or business activities such as cost, sales, or profits.
- Creating reports and dashboards to showcase findings and for further data exploration or analysis.

Data Science and Visualization Tools

Many different types of tools can be used to visualize data. Some tools like Microsoft® Excel® and Google Sheets™ can capture and store data, show it in tabular format, and create visualizations. Other dedicated data visualizations and analysis tools like Tableau® and Microsoft® Power BI® don't allow data input, but connect to a diverse array of data sources and databases, allowing you to analyze data in a variety of ways, and create rich, interactive, and dynamic visual reports to showcase key data points and analytical conclusions.

Some data visualization tools, including those already mentioned are:

- Microsoft Excel: **https://www.microsoft.com/en-us/microsoft-365/excel**
- Google Sheets: **https://www.google.com/sheets/about/**
- Microsoft Power BI: **https://powerbi.microsoft.com/en-us/**
- Tableau: **https://www.tableau.com/**
- Qlik Sense®: **https://www.qlik.com/**
- SAP® Crystal Reports®: **https://www.crystalreports.com/**
- Domo: **https://www.domo.com**
- R Project for Statistical Computing: **https://www.r-project.org/about.html**
- Python or other programming languages.

Project R for Statistical Computing

The ***R Project*** is an open source programming language and software environment for statistical computing and graphics. It is supported by the R Foundation for Statistical Computing. Project R and the R language are widely used by statisticians for data analysis. The R software environment consists of desktop software and a server component named R Serve, which are available for most

operating systems. R supplies an environment that supports a wide array of linear and non-linear analysis and statistical tests, as well as graphing capabilities.

Additional information: **https://www.r-project.org/**.

Python for Data Science

Where does Python fit with all of these data analysis and visualization tools? Python picks up where other tools all leave off by providing easier pathways to cleaning data, automating tasks, and applying machine learning capabilities to data analysis, and to create custom ways to share insights. In 2016, the use of Python overtook R in usage in some data science competitions and continued to grow its footprint in the data science community. As of 2019, 87% of data scientists reported using Python.

Python is not likely the best choice for doing quick, "out-of-the-box" visualizations supported in an app like Tableau, using data from a well-understood, structured data source. However, Tableau may not work well for some types of analysis on specific types of data such as nested data, unstructured NoSQL data sources, or data collected from web scraping. In this case, you may spend too much time getting the right data into a format where it's clean enough to analyze.

R is great for many types of statistical analysis and has robust connectivity and many features. It falls short where you have to clean data before it's stored. For example, when working with data scraped from websites over multiple days, it's advantageous to automate steps in data collection, cleaning, and storage.

Further, advanced data analysis can be aided not only by task automation, but by machine learning which can be used to do everything from fine-tuning data collection, to refining data sets, to only the data that is most important.

Where other tools leave off, Python becomes very useful. It supports both structured and unstructured data and has many libraries you can add to assist with data cleaning, automation, and machine learning.

Data science tools are not mutually exclusive and it's not uncommon for data scientists to use multiple tools on a regular basis based on the best fit for the project and data they are working with. In fact, data scientists may have a preferred "pecking order" of tools they utilize. For example, a data scientist might try to bring data into Tableau first to make use of its robust connectivity, data preparation, and rich visualization tools. If Tableau can't connect to or clean the data effectively, the data scientist may move to R. If R can't accommodate the data cleaning, task automation, or machine learning requirements, they may move on to Python. Some data scientists use Python less for data exploration and more for capture and production, utilizing its capabilities to capture and clean data, then pulling it into a dedicated analysis tool such as Tableau, and then potentially moving back to Python to create custom web-based visualizations.

Python Libraries for Data Science

There are also numerous libraries for creating complex, custom visualizations that may not be supported by other tools. The following are some of the most popular Python libraries for data science.

Library	*Description*
NumPy	For working with data arrays and cleaning data.
Pandas	For data manipulation and analysis.
Matplotlib	An extension of NumPy used to plot data to create visualisations.
Scikit-Learn	A free machine-learning library for Python.
TensorFlow	A deep machine-learning library released by Google.

Library	Description
Seaborn	Uses Matplotlib to plot graphs and visualize random distributions.
PyTorch	A machine learning library developed by Facebook's AI research group used for developing and training neural, network-based deep learning models.
Anaconda	A distribution of Python and R for scientific computing, designed to simplify deployment and package management. Anaconda is the preferred distribution for the SciPy library set.

Additional Information

For more information on NumPy, see: **https://numpy.org/**.

For more information on Pandas, see: **https://pandas.pydata.org**.

For more information on Matplotlib, see: **https://matplotlib.org/**.

For more information on Scikit-Learn, see: **http://scikit-learn.org**.

For more information on TensorFlow, see: **https://www.tensorflow.org/**.

For more information on Seaborn, see: **https://seaborn.pydata.org/**.

For more information on PyTorch, see: **https://pytorch.org/**.

For more information on Anaconda, see: **https://anaconda.org/**.

Appeal of Python for Data Science

Knowing how Python is used in data science naturally leads to the question: why is Python preferred for data science? There are many reasons, starting with perhaps the most obvious: Python is a simple language that is generally easy to learn and not cumbersome. Consider the following comparison of code used for Java and Python to display "Hello, world":

Java:

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, world");
   }
}
```

Python:

```
print("Hello, world")
```

In addition, there are a number of frameworks, such as Flask and Django, that make coding for web, business, and e-commerce easier. In addition, as already discussed, Python is highly scalable, and there are numerous libraries available that are useful for data analysis and visualization.

Python is also well suited and often used for web development, which is advantageous for data scientists hoping to provide web-based access or reporting to their apps or reports. There is also a large community of Python programmers and forums where data scientists can find code samples to leverage, and get questions answered. There is also a large community of data scientists already using Python.

From a capability standpoint, the automation and machine learning capabilities available with the addition of certain libraries, provide data scientists with powerful new ways to collect, analyze, and visualize data.

Data Cleanup

The goal in cleaning data is to refine the data you're working with to include only the essential information needed for analysis, and to format and organize it to make it easy to analyze, find insights, and create reports and visuals. Data from most data sources is not clean to start with. Cleaning data is typically the first step in data analysis. Even well-structured data stored in a relational database usually needs some cleanup, and data taken from non-structured sources such as NoSQL databases, or scraped from websites, may need considerable cleanup.

When performing data cleanup, you may need to do some or all of the following:

- Removing unnecessary data by showing only the subset of data (this also improves performance in data querying).
- Remove duplicate values in data.
- Address missing and corrupt values in data.
- Rename fields so that the field names make more sense in the scope of your analysis.
- Create hierarchies of data such as dates, or by creating other categories of related information.
- Sort and group fields and rows of data.
- Split or merge fields to organize data as required (such as splitting out zip codes that are included in an address field).
- Create new fields based on calculations applied to other fields of data.
- Create new tables of data.
- Merge or union existing data sources.

NumPy Library

NumPy (Numerical Python) is a linear algebra library in Python. It provides `ndarray` class, a multidimensional array object and other derived objects such as masked arrays and matrices, and allows you to perform the following types of operations (and others) on large arrays of data:

- Mathematical
- Logical
- Shape manipulation
- Sort
- Selection
- Input/Output operations
- Discrete Fourier transformations
- Basic linear algebra
- Basic statistical operations
- Random simulation

NumPy can typically perform these operations more efficiently with less code than using Python's built-in sequences. NumPy is vectorized code, with explicit looping, indexing, and other operations executing in the background via an optimized, pre-compiled C code. This makes NumPy execution faster, the code more concise, easier to read, and generally requires fewer lines of code. The code also more closely remembers standard mathematical notation which usually makes it easer to translate mathematical constructs to code. The resulting code also looks more like other Python code. Without vectorization, the code would have numerous `for` loops.

Many data science libraries such as matplotlib and scikit-learn rely on NumPy.

NumPy is part of the SciPy (scientific computing tools for Python) ecosystem of tools.

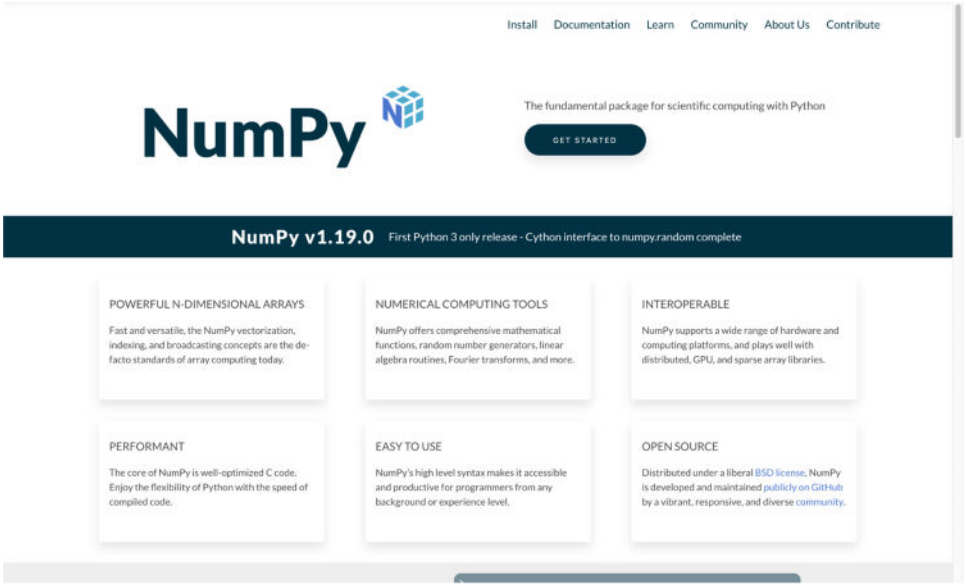


Figure 6-1: NumPy library.

Additional Information

See more about NumPy here: **https://numpy.org/**.

See more about SciPy here: **https://scipy.org/**.

NumPy Arrays

The core functionality of the NumPy package comes form the `ndarray` object. It encapsulates n-dimensional arrays of homogeneous data types, allowing you to perform operations on the arrays and the data contained within. There are some important differences between NumPy arrays and the standard Python sequences:

- NumPy array size is fixed when the array is created (unlike Python lists that can grow).
- Array elements must be the same data type.

The following samples showcase some ways to use NumPy and `ndarray` which is implemented through the array function:

Create an array from a Python list:

```
import numpy as np

py_list = [1,2,3,4,5]

numpy_list = np.array(py_list)

print(numpy_list) #This prints the array values.
```

Note: This creates a one-dimensional array.

Create a two-dimensional (2d) array:

To create a two-dimensional list, you create a list with multiple lists using brackets and create an array from that.

You can then create a two-dimensional array:

```
import numpy as np
```

```
multi_list = [[1,2,3], [4,5,6], [7,8,9]]

twod_array = np.array(multi_list)

print(twod_array) #This displays the result of the generated array.
```

You can also create the array without using a variable:

```
twod_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

You can visualize the 2d array in the following way:

Index	0	1	2
0	1	4	7
1	2	5	8
2	3	6	9

Figure 6-2: NumPy 2d array.

Select items using single `[,]` or double bracket `[] []` notation:

```
print(twod_array[0,1]) # Single-bracket notation. This returns the value "2"

print(twod_array[2][1]) # Double-bracket notation. This returns the value "8"
```

Select elements with slice notation:

Slicing allows you to identify a portion of data to take from an array from one defined index to another defined index using the notation `[start:end]`. You can also define steps in the selection using `[start:end:step]`. If the start index is not defined, it's considered 0 to the notation, `[:1, :2]` is the same as `[0:1, 0:2]`. If the end is not defined it's assumed to be the length of the array at the defined dimension. If a step is not passed, it's assumed to be 1. When slicing data, each column of data is considered a separate element.

To select items using slice notation:

```
print(twod_array[2, :2]) #This displays "[7, 8]". The data in element 2 of the
array, between index 0 and 2 (not included).

print(twod_array[0:3, 1:3]) #This displays "[[2,3] [8,9]]". The data displayed
is from elements 0 through 2, at index positions 1 to 3 (not included).

print(twod_array[0]) #This displays "[1, 2, 3]" all values in the first
element of the array.

print(twod_array[:2]) #This displays "[[1,2,3] [4,5,6]]" all values prior to
element 2.
```

NumPy Logical and Conditional Array Operations

You can also perform logical and conditional operators. For example:

```
import numpy as np

twod_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

```
print(twod_array > 5) #This returns "True" for array values greater than 5
displaying, "[False, False, False, False, False, True, True, True, True, True]"

print(twod_array[twod_array > 5]) #This prints the values over 5 in the array
displaying, "[6, 7, 8, 9]"

print(twod_array[(twod_array > 3) & (twod_array < 9)]) #This uses logical
operator to print values from the array greater than 3 and less than 9
resulting in, "[4, 5, 6, 7, 8]"
```

Additional Information

For the list of NumPy logical operators, see: **https://numpy.org/doc/stable/reference/routines.logic.html**.

Pandas Library

The pandas library is part of the SciPy ecosystem of tools and is built on NumPy. It is designed to facilitate analysis for relational or labeled data. Pandas works with tabular data like those found in SQL database tables and spreadsheets, ordered and unordered time series data, arbitrary data in a matrices with row and column labels, and any form or observational or statistical data sets (labeling is not required). Pandas is a dependency of the `statsmodels` Python module that provides classes and functions for statistical analysis and is extensively used in production of financial applications.

Pandas supports the following two primary data structures.

Data Structure	*Description*
Series	One-dimensional labeled, homogeneously-typed array.
DataFrame	Two-dimensional labeled, size-mutable tabular structure with potentially heterogeneously-typed column. This data frame provides all functionality in R's data frame and more features.

Pandas makes many data analysis activities easier including:

- Handling missing data. Represented at NaN in both floating point and non-floating point data.
- Size mutability. Columns can be inserted and deleted from DataFrames and higher dimensional objects.
- Data alignment (automatic and explicit). Align objects explicitly to a set of labels, or ignore labels and let the data structures align data for you in computations.
- Group by and Split-apply-combine functionality. Useful for aggregating and transforming data.
- Conversion of differently-indexed data in Python and NumPy data sources into DataFrame objects.
- Intelligent, label-based slicing, indexing, and subsetting of large data sets.
- Merging and joining of data sets.
- Hierarchical labeling of axes.
- Robust data access from flat files including CSV, delimited, Excel, database, and HDF5 format files.
- Time series functionality including date range generation, frequency conversion, moving window statistics, and more.

Pandas DataFrames

You can use the following to read, output, and select data in pandas.

Read a CSV file:

Reading is done using the `read_*` for the desired file type. Each column of data is assigned a name based on the imported data such as column headers, and a data type referred to in pandas as a `dtype` when it is read. Each column in a DataFrame is a Series, and when a single column is selected it is returned as a pandas Series.

```
import pandas as pd

orders = pd.read_csv ( "salesdata/orders.csv" ) #Read the orders.csv file in
the path specified.
```

Read an Excel spreadsheet:

```
orders = pd.read_excel('orders.xlsx', sheet_name='sales') #Read the Sales
worksheet of the orders.xlsx Excel file.
```

Output to spreadsheet:

Save output to a file using the `to_*` function.

```
orders.to
```

Display data from the DataFrame:

```
orders #This displays the first and last 10 rows of the dataframe by default.

orders.head(10) #This displays the first 10 rows of the dataframe.
```

View DataTypes:

```
orders.dtypes #This displays the data types attribute for each column of data
in the dataframe.
```

View a technical summary of the DataFrame:

```
orders.info()
```

This will output the range index (number of entries), total columns, column number, column names, non-null count, and dtypes.

View a specific column of data with column name:

View a single column using brackets `[]` containing the name of the column you're interested in.

```
salesprice = orders["salesprice"] #Set the series to a variable name.

salesprice.head() #This displays the first five items in the SalesPrice column.
```

View the shape of a data series (column):

```
orders["salesprice"].shape
```

This will output the number of entries (rows) for this series.

View data in multiple columns:

```
sale_customer_zip = orders[["salesprice", "customername", "zipcode"]]

sale_customer_zip.head() #This displays the first 5 rows of data.
```

Select customers with sales over $100:

```
sales_100 = orders[orders["salesprice"] > 100]

sales_100.head() #This displays the first 5 rows of data.
```

Pandas dtypes:

For a list of pandas dtypes see: **https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#basics-dtypes**.

Pandas Data Cleanup and Manipulation

Pandas is an excellent resource for cleaning up and massaging data ahead of analysis. You can use it to clean, transform, and shape data in the following ways:

Create new columns from existing columns:

In this case, we want to give customers 2% back on each order, so this must be calculated. To do this, we specify the new column and set it equal to a formula that includes calculation data and data from existing columns:

```
import pandas as pd

orders["salesreward"] = orders[salesprice] * .02) #This multiplies the
salesprice column by 2% and stores the value in the salesreward column

orders.head() #View first five columns of data with the newly added column.
```

Calculate summary statistics:

Leverage pandas statistics features to discover valuable information about your data, such as the average sale price using the `mean()` method.

```
orders["salesprice"].mean() #This outputs the arithmetic mean of the series of
data, in this case all sales in the DataFrame.
```

Reshape data:

There are many ways to reshape data to make it easier to work with. This may include sorting pivot columns so that they appear next to each other, and so forth. In this case, you will use the `sort_values` method to sort by customername, and then by zipcode, then customername, and finally orders values in descending order.

```
orders.sort_values(by="customername").head() #This sorts orders by
customername and outputs the first 5 rows.

orders.sort_values(by=['zipcode', 'customer', 'salesprice'],
ascending=False.head() #This sorts the orders by zipcode, customername, then
salesprice in descending order.
```

Manipulate text data:

There are many reasons you may wish to manipulate text data. In this case, the customer name is stored "lastname, firstname" format in a single field and you wish to break the last name into its own field by splitting the field at the comma using *`series`*`.str.split()` method and using the `Series.str.get()` method to extract the required text data.

```
orders["lastname"] = orders["customername"].str.split(",").str.get(0) #This
splits the contents of the customername at the comma and extracts the relevant
data, that precedes the comma
 to the "lastname" field.

orders.rename(columns = {'customername':'firstname'}, inplace = True)
```

Guidelines for Cleaning Data with Python

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when cleaning data with Python.

Clean Data with Python

When cleaning data with Python:

- Remove unnecessary data and leave only the subset of data you need when performing data analysis.
- Rename fields so that the field names make more sense in the scope of your analysis.
- Sort and group fields and rows of data to make analysis easier.
- Split or merge fields to organize data to make analysis easier.

ACTIVITY 6-1
Cleaning Data with Python

Data Files
All project files in C:\094022Data\Programming Python for Data Science\WWProject-L6.

C:\094022Data\Programming Python for Data Science\2020-Sales.csv

Before You Begin
PyCharm is open.

Scenario
You have received sales data for the year 2020 in a CSV file. You would like to use this data to create some visualizations and forecasting in Python. Before you do, you want to review the data and clean and prepare it so it better fits your needs. You want the data to be sorted by date, have first and last name for customers, have sales, cost, and profit numbers, and not be cluttered with data you won't use. First, you will review some of the commands you might use to prepare the data.

---

1. Install NumPy and pandas.
   a) Open Command Prompt.
   b) In the Command Prompt, type the following and press **Enter** to install NumPy.

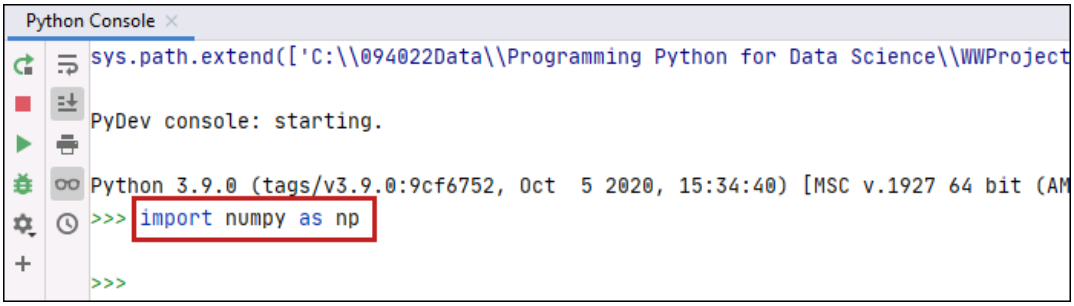
   c) Type the following and press **Enter** to install pandas.

   d) Minimize but do not close the Command Prompt.

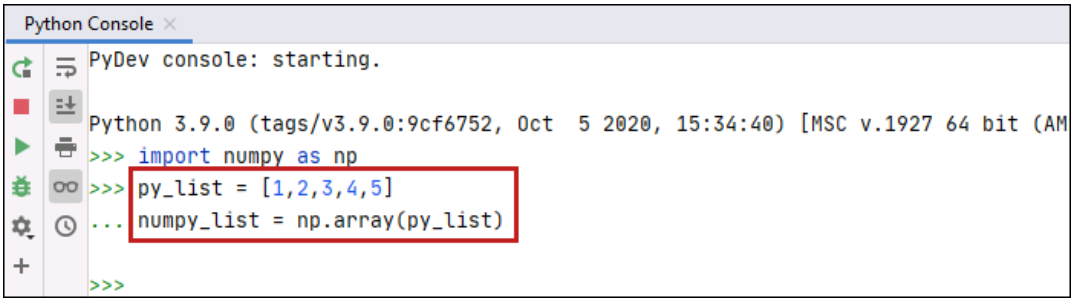
2. Open the **WWProject-L6** project.
   a) In the Welcome to PyCharm window, select **Open**.
   b) Navigate to the **C:\094022Data\Programming Python for Data Science\WWProject-L6** folder.
   c) With the **WWProject-L6** folder selected, select **OK**.
   d) In the **Project** pane on the left side of the PyCharm window, verify that **WWProject-L6** is listed.

3. Create an array with NumPy.
   a) Select **Tools→Python or Debug Console**.

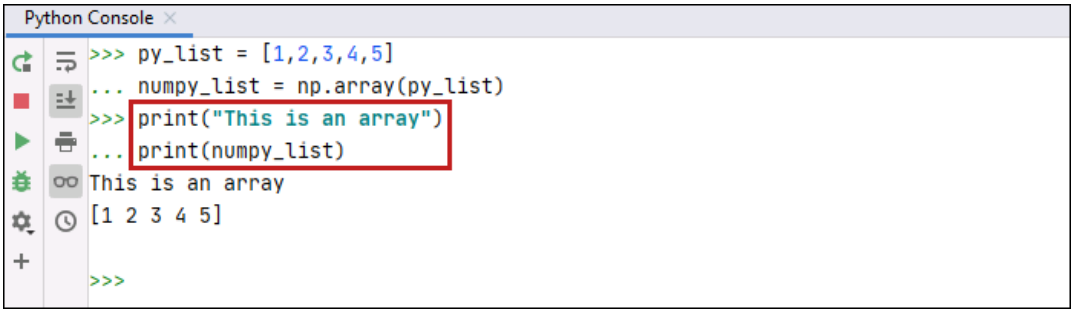
b) Enter the following to import NumPy with the `np` alias.



c) Enter the following to create a variable and then use that variable to create an array.



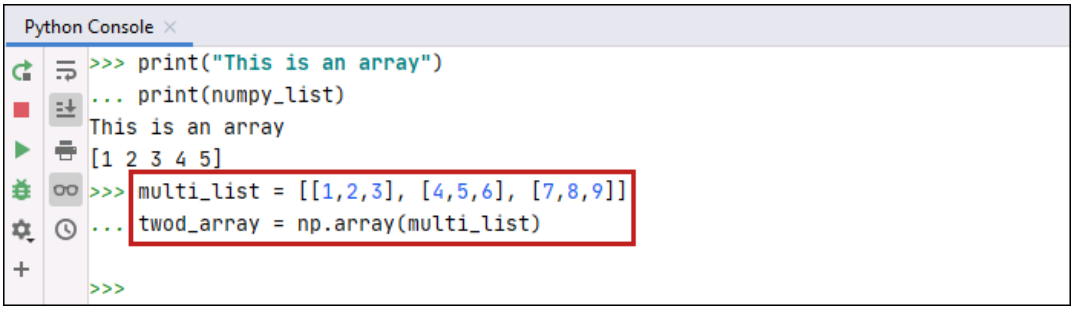
d) Enter the following to print the array.



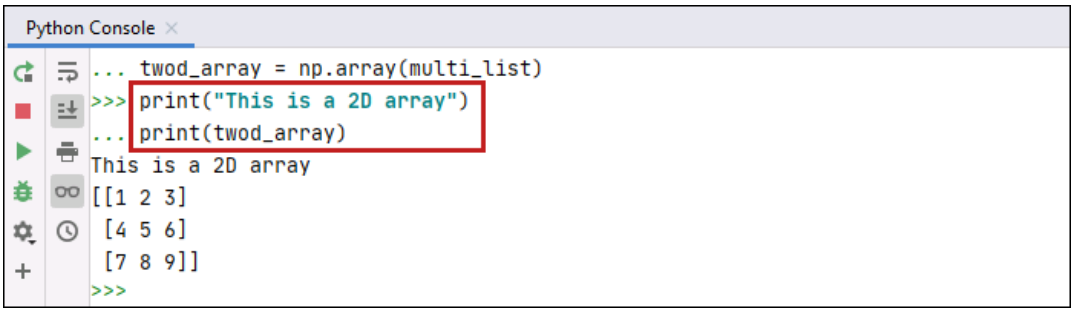
The array is printed below the statement you typed.

4. Create a multidimensional array with NumPy.

a) Enter the following to create a variable and then use that variable to create a multidimensional array.

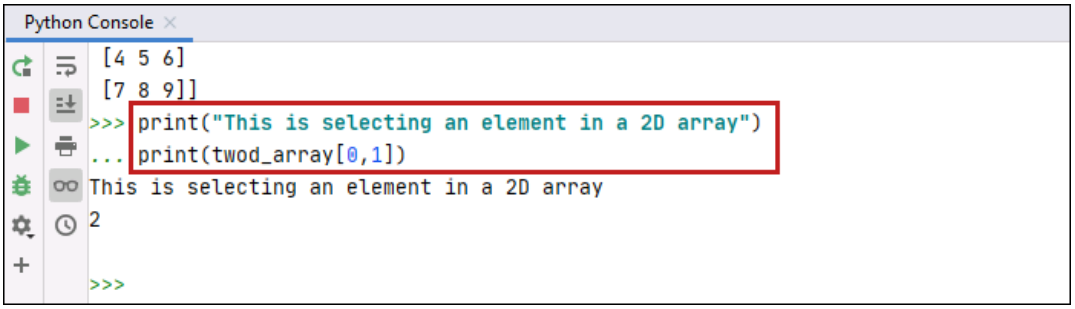


b) Enter the following to print the multidimensional array.



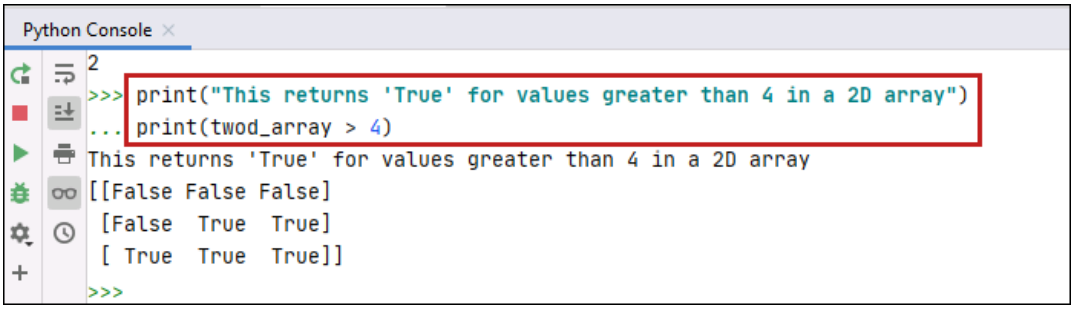
Three rows of three numbers each are printed below the statement you typed.

c) Enter the following to print the selection of a single value in a multidimensional array.



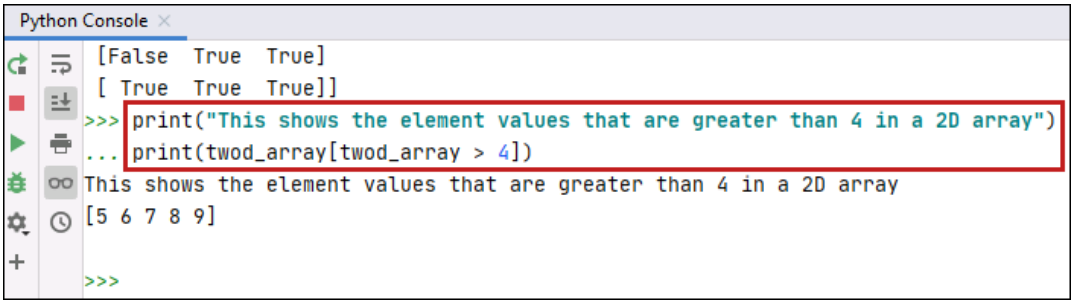
The second value from the first row of the array, which is the number 2, is printed below the statement you typed.

d) Enter the following to print True\False for elements in the multidimensional array that are greater than four.



The four "False" values for the elements equal to or less than four, and the five "True" values for the elements greater than four, are printed below the statement you typed.

e) Enter the following to print the values of the elements in the multidimensional array that are greater than four.



The array elements that are greater than four are printed below the statement you typed.

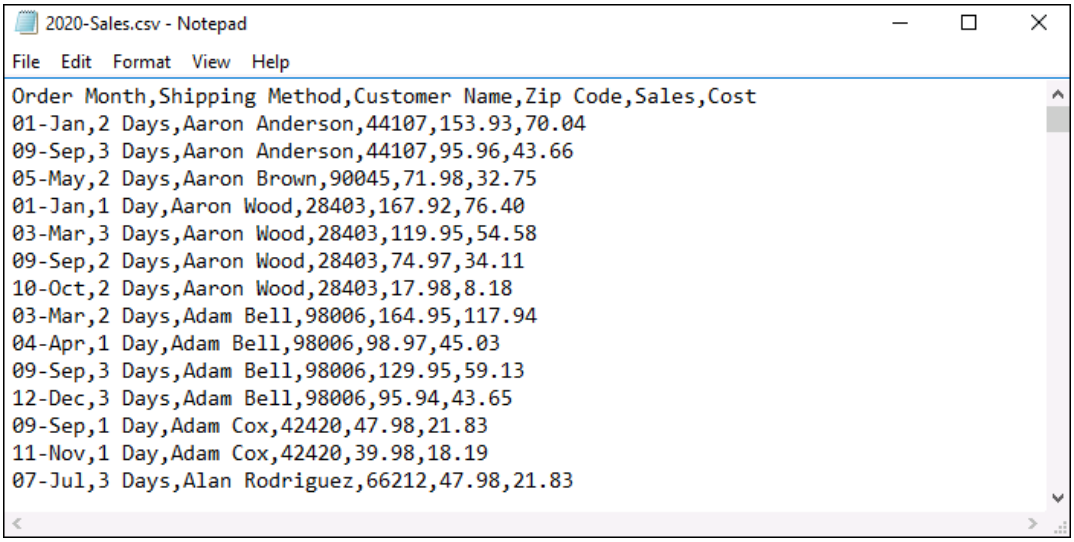
f) Close the **Console** pane.

5. Review WW-2020-Sales.csv file.

a) Open File Explorer and navigate to **C:\094022Data\Programming Python for Data Science**.

b) Open **2020-Sales.csv** with Notepad.

c) Observe that the file contains sales data for the year 2020.



It has columns for Order Month, Shipping Method, Customer Name, Zip Code, Sales and Cost, and appears to be sorted by customer name.

d) Close Notepad.

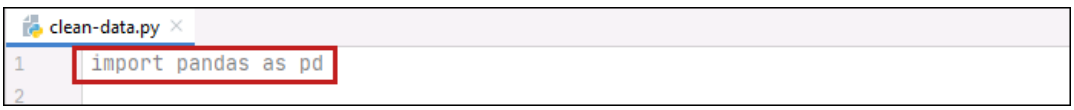
6. Create the clean-data.py file in the WWProject-L6 project.

a) In the **Project** pane on the left, right-click the **WWProject-L6** project, and select **New→Python File**.

b) In the **Name** box, type ***clean-data*** and press **Enter**.

7. Read a .csv file with pandas.

a) In the **clean-data.py** tab, starting on line 1, type the following to import the pandas library.



b) After the `import` method, starting on line 3, type the following to read the .csv file.

```
clean-data.py
1  import pandas as pd
2
3  orders = pd.read_csv("../2020-Sales.csv")
```

This will read the CSV file and store the dataframe in the orders variable.

c) Starting on line 5, type the following to print the data of the dataframe.

```
3  orders = pd.read_csv("../2020-Sales.csv")
4
5  print(orders)
6
```

This will print the first five and last five rows of data.

d) Starting on line 7, type the following to display the technical summary of the dataframe.

```
5  print(orders)
6
7  orders.info()
8
```

This will display information about the dataframe such as number of entries and columns, the column names and datatype, etc.

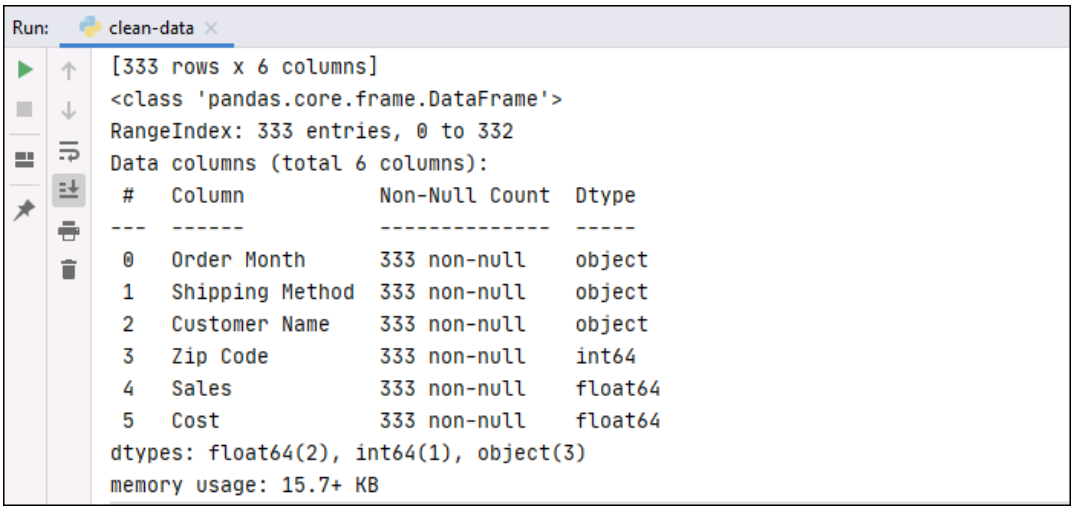
e) Run **clean-data.py**.

f) Scroll up and observe the first five and last five rows of data in the dataframe.

```
Run:  clean-data
C:\Users\user1\AppData\Local\Programs\Python\Python39\python.exe "C:/094022Dat
    Order Month Shipping Method     Customer Name  Zip Code   Sales    Cost
0        01-Jan          2 Days    Aaron Anderson     44107  153.93   70.04
1        09-Sep          3 Days    Aaron Anderson     44107   95.96   43.66
2        05-May          2 Days       Aaron Brown     90045   71.98   32.75
3        01-Jan           1 Day       Aaron Wood     28403  167.92   76.40
4        03-Mar          3 Days       Aaron Wood     28403  119.95   54.58
..          ...             ...              ...       ...     ...     ...
328      08-Aug          2 Days     Virginia Rice     87105  263.89  120.07
329      11-Nov          2 Days     Virginia Rice     87105  124.95   56.85
```

This is the same data you saw when you opened the .csv file in Notepad with the row number added on the left.

g) Scroll down and observe the technical summary of the dataframe.



h) In the **Run** console pane, close the **clean-data** tab.

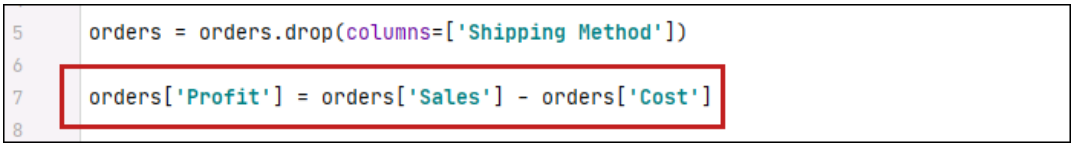
8. Clean up the dataframe.

a) In the **clean-data.py** tab, delete all code except for the `import` statement and the reading of the .csv file.

b) Starting on line 5, type the following to remove a column.

This will remove the **Shipping Method** column.

c) Starting on line 7, type the following to add a column.



This will add the **Profit** column which is the sum of the **Sales** column minus the **Cost** column.

d) Starting on line 9, type the following to split a field.

```
7   orders['Profit'] = orders['Sales'] - orders['Cost']
8
9   orders['Last Name'] = orders['Customer Name'].str.split(' ').str.get(1)
10  orders['Customer Name'] = orders['Customer Name'].str.split(' ').str.get(0)
11  orders.rename(columns = {'Customer Name':'First Name'}, inplace = True)
12
```

These statements:

- Create the **Last Name** column and it contains text after the space (" ") from the **Customer Name** column.
- Change the **Customer Name** column to contain text before the space (" ") from the **Customer Name** column.
- Rename the **Customer Name** column to **First Name**.

e) Starting on line 13, type the following to change the sort of the rows in the dataframe.

```
9   orders['Last Name'] = orders['Customer Name'].str.split(' ').str.get(1)
10  orders['Customer Name'] = orders['Customer Name'].str.split(' ').str.get(0)
11  orders.rename(columns = {'Customer Name':'First Name'}, inplace = True)
12
13  orders = orders.sort_values(by=['Order Month', 'Last Name', 'First Name'])
14
```

This will change the sort of the rows to sort by Order Month, then Last Name, and then First Name.

f) Starting on line 15, type the following to output to a .csv file.

```
13  orders = orders.sort_values(by=['Order Month', 'Last Name', 'First Name'])
14
15  orders.to_csv(r"../2020-Sales-Clean.csv")
16
```

Note: Ensure you type the path and file name as they appear because they will be used in the next topic.

This will output the updated dataframe to a new .csv file named **2020-Sales-Clean.csv**.

g) Starting on line 17, type the following to print the data of the dataframe.

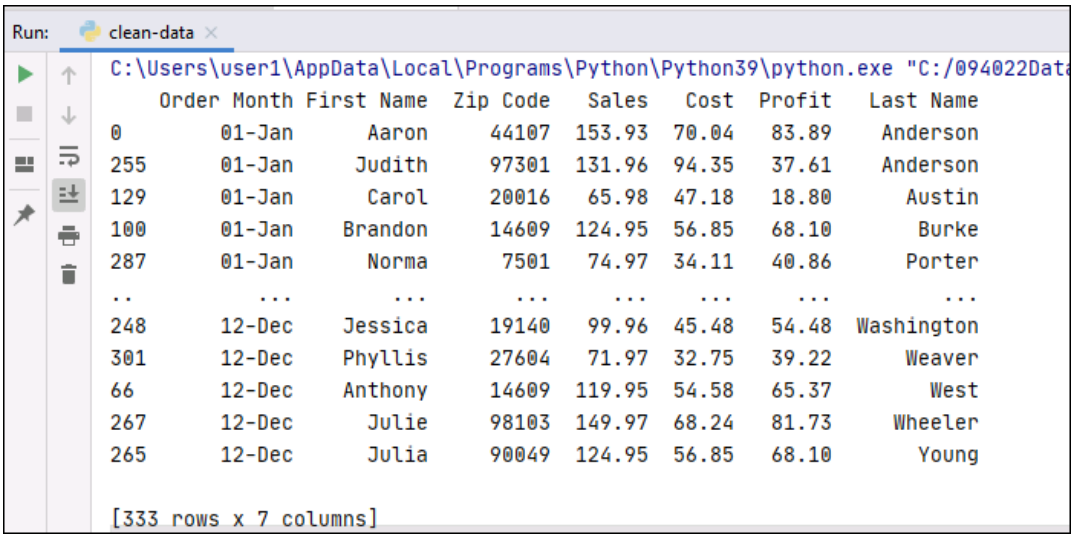
```
15  orders.to_csv(r"../2020-Sales-Clean.csv")
16
17  print(orders)
18
```

This will print the first five and last five rows of data in the dataframe.

9. Run clean-data.py.

a) Run **clean-data.py**.

b) Observe the change to the dataframe.



The **Shipping Method** column is gone, the **Customer Name** column has been replaced by the **First Name** and **Last Name** columns, the **Profit** column has been added, and the dataframe is sorted by **Order Month**, then **Last Name**, and then **First Name**.

c) In the **Run** console pane, close the **clean-data** tab.

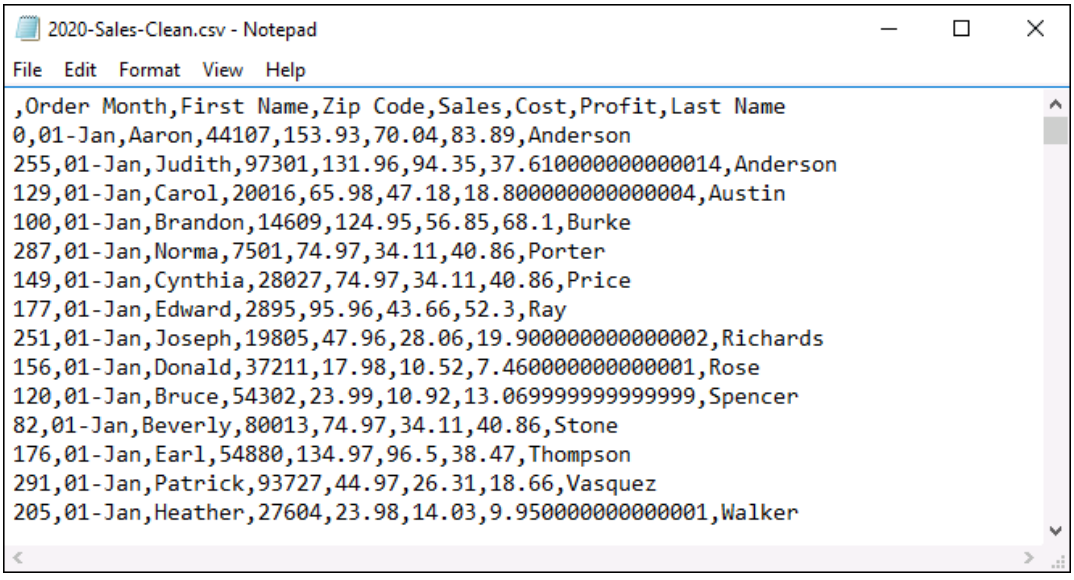
d) Close the **clean-data.py** tab.

10. Review the 2020-Sales-Clean.csv file.

a) In File Explorer, navigate to **C:\094022Data\Programming Python for Data Science**.

b) Open **2020-Sales-Clean.csv** with Notepad.

c) Observe that the file contains the updates.



The changes you made with Python are present in the new .csv file. You might want to limit the number of decimal places on profit, but it is OK for now.

d) Close Notepad.

TOPIC B

Visualize Data with Python

When you find insights from your data science analysis, it's helpful to showcase them graphically. In this topic, you will visualize data with Python.

Data Visualization

The term ***data visualization*** describes the act of using visual materials such as charts, graphs, maps, and other visualizations to analyze data, to find patterns in data, and to report insights gleaned from data. Using visualizations makes it easier to find and show patterns, trends, and correlations in data. In fact, without the use of visualizations, these insights might go undetected. For complex data sets or analysis that must correlate data from multiple sources, data visualization is often essential to gain a meaningful understanding of complex relationships represented in the data.

Data visualization has become a standard practice for modern business intelligence, as the data available for analysis has grown with computing capability and cloud-based storage. Data visualization is used for planning, analysis, and real-time alerting based on pre-defined conditions.

Data visualizations can be static or animated and may update in real time as data is updated. Data visualizations may also include interactive capabilities, which enable users to manipulate data, change the visualization, submit queries against the data, or drill into data for deeper analysis, or even ask natural language questions of the data.

Although data visualization used to be the domain of dedicated ***data scientists***, data analytics has become a necessary part of many jobs. In fact, any job that requires analyzing or reporting on data sets of any size often has a data visualization component. Humans process visual data more quickly, and visualized data allow us to see differences in data more easily.

Today's data visualization tools go beyond the standard charts and graphs used in Microsoft Excel spreadsheets, to displaying data in more sophisticated visualizations, using dedicated data visualization tools and programming languages like Python.

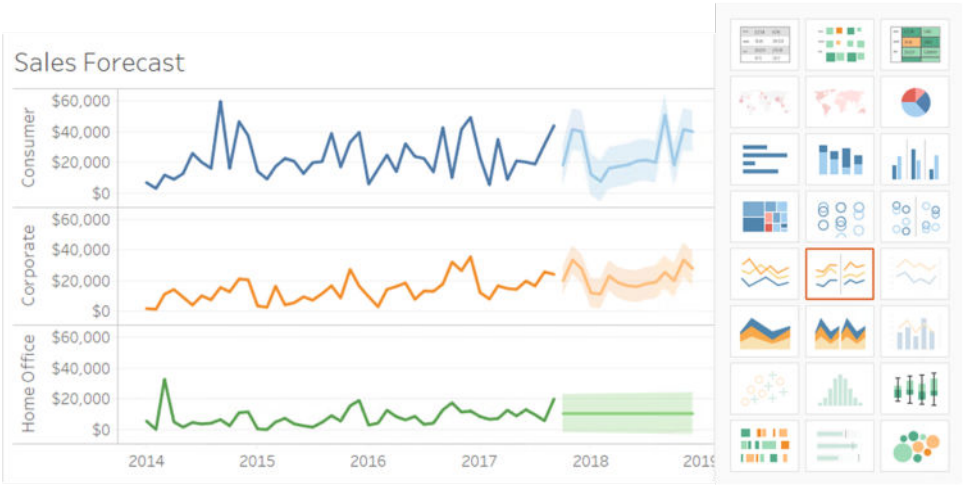


Figure 6-3: Data visualization use cases.

Matplotlib Library

Matplotlib is part of the SciPy ecosystem of tools and is a free, open source, low-level graph plotting library you can use to create visualizations in Python. Most Matplotlib utilities are in the `pyplot` submodule. You must install Matplotlib with PIP to make it available on your system.

Once you do, the following will be available in your Python project:

Plot a line:

To draw a simple line base on two NumPy arrays using the `plot` method:

```
import matplotlib.pyplot as plt
Import numpy as np

xcoords = np.array([0, 5]) #This creates a NumPy array containing [1, 2, 3, 4,
5].
ycoords = np.array([0, 200]) #This creates a NumPy array containing [ 0,
1, ... 199, 200].

plt.plot(xcoords, ycoord) #This plots the xcoords on the X axis by the ycoords
on the Y axis.
plt.show() #This shows the plot.
```

Plot points:

To draw points at position (1, 50) and (4, 150) using the array's shortcut notation parameter `'o'`, which will plot rings:

```
plt.plot(xcoords, ycoords, 'o')
plt.show()
```

Additional Information

Website: **https://matplotlib.org/**

Codebase: **https://github.com/matplotlib/matplotlib**

Plot Line Graphs

Plotting points creates a line graph with Matplotlib, however, there are many more options that you can use to create more interesting visualizations. Again, if we start with two NumPy arrays with non-sequential points and plot them to create a line graph:

```
import matplotlib.pyplot as plt
Import numpy as np

xcoords = np.array([2, 3, 5, 9])
ycoords = np.array([3, 8, 1, 10])

plt.plot(xcoords, ycoord) #This plots the xcoords on the X axis by the ycoords
on the Y axis.
plt.show() #This shows the plot.
```

Add markers to a line graph:

Add markers to a line graph to make key points stand out by adding the marker `argument` to the plot method. You can choose the ring marker used earlier or select from a wide number of available markers.

```
plt.plot(xcoords, ycoords, marker = 'o')
plt.show()
```

Change the style of a plotted line:

By default, Matplotlib plots using a solid line. You can plot a different style of line by adding the `linestyle` argument to the plot method using either full or shortened syntax:

```
plt.plot(xcoords, ycoords, marker = 'o', linestyle = 'dotted') #This uses the
linestyle argument with full syntax to plot a dotted line.
plt.show()

plt.plot(xcoords, ycoords, marker = 'o', linestyle = '--') #This uses the
linestyle argument with shortened syntax to plot a dashed line.
plt.show()
```

Change the color of a plotted line:

Change the color of a plotted line by adding the color argument using a supported color name or the hex value of the color:

```
plt.plot(xcoords, ycoords, marker = 'o', linestyle = '--', color = 'orange')
#This adds the color argument to plot an orange line.
plt.show()

plt.plot(xcoords, ycoords, marker = 'o', linestyle = '--', color = '#7FFFD4')
#This adds the color argument to plot an aquamarine line using the color's hex
code.
plt.show()
```

Note: You can also use the `linewidth` argument to make plotted lines wider or more narrow.

Plotting multiple lines:

To plot multiple lines, create the desired arrays and plot them in sequence:

```
import matplotlib.pyplot as plt
Import numpy as np

x1 = np.array([2, 3, 5, 9]) #First X axis
y1 = np.array([3, 8, 1, 10]) #First Y axis
x2 = np.array([1, 4, 6, 9]) #Second X axis
y2 = np.array([5, 3, 4, 8]) #Second Y axis

plt.plot(x1, y1, x2, y2)
plt.show
```

Matplotlib will automatically color the first plotted line blue, and second orange. If you wish to style lines, you can add arguments as follows:

```
plt.plot(x1, y1, marker = 'o', linestyle = '--', color = '#7FFFD4' x2, y2,
marker = 'o', color = '#B8860B') #This creates a first dashed line in
aquamarine, and a second solid line with markers in dark goldenrod.
plt.show
```

Available Markers, Linestyles, and Colors

See supported linestyles here: **https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/linestyles.html**

See available markers here: **https://matplotlib.org/api/markers_api.html**

See supported colors: **https://matplotlib.org/3.1.0/gallery/color/named_colors.html**

Create Charts

You can create many different types of charts with Matplotlib using the NymPy arrays and the correct Matplotlib method. The following table lists the methods for some of the most popular types of charts.

Visual	*Method*
Scatter plot	scatter()
Bar (vertical)	bar()
Bar (horizontal)	barh()
Histogram	hist()
Pie	pie()

Create a vertical bar chart:

```
import matplotlib.pyplot as plt
Import numpy as np

x = np.array(["Q1", "Q2", "Q3", "Q4"]) #X axis labels for each fiscal quarter
y = np.array([3, 5, 4, 6]) #Y axis labels for sales in the millions

plt.bar(x,y)
plt.show()
```

Create a horizontal bar chart:

```
import matplotlib.pyplot as plt
Import numpy as np

x = np.array(["Q1", "Q2", "Q3", "Q4"]) #X axis labels for each fiscal quarter
y = np.array([3, 5, 4, 6]) #Y axis labels for sales in the millions

plt.barh(x,y)
plt.show()
```

You can change bar colors using the `color` argument, bar width using the `width` argument, and height using the `height` argument.

Additional Information

Matplotlib visualization gallery: **https://matplotlib.org/3.1.0/gallery/index.html**

Python graph gallery: **https://python-graph-gallery.com/all-charts/**

GroupBy Function

The GroupBy function is part of pandas and is used to group large amounts of data and perform processing operations on these groups. The GroupBy function typically splits an object, applies a function, and then combines the results. It returns the grouped object with information about the results.

GroupBy has the following parameters.

Parameter	*Statement Requirements*	*Usage*
`by`	mapping, function, label, or list of labels	Determines the groupings for GroupBy. Typically uses series values or data labels.

Parameter	Statement Requirements	Usage
`axis`	{0 or 'index', 1 or 'columns'}, default 0	Split along rows (0) or columns (1).
`level`	int, level name, or sequence of such, default None	If the axis is a MultiIndex (hierarchical), group by a particular level or levels.
`as_index`	boolean, default True	For aggregated output, return object with group labels as the index.
`sort`	boolean, default True	Sort group keys. You may see better performance by turning this off.
`group_keys`	boolean, default True	When calling apply, add group keys to index to identify pieces.
`squeeze` (depreciated)	boolean, default False	Reduce the dimensionality of the return type if possible, otherwise return a consistent type.
`observed`	boolean, default False	This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.
`dropna`	boolean, default False	If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups

Object-Oriented Interface

The alternative approach to plotting in Matplotlib is to use the object-oriented interface. This interface is stateless; it does not keep track of the "active" plot elements like the stateful interface does. Instead, you create objects that represent the plot and then use these objects to both generate and modify the plot, as needed.

Compared to the stateful approach, the object-oriented approach requires a little more code up front. You need to first construct objects of the `Figure` and `Axes` classes. Objects constructed from methods of these classes will have the necessary information to generate a plot. Consider the following example, which constructs the same plot as the stateful example earlier, but this time using the object-oriented interface:

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(revenue_df['Month'], revenue_df['Revenue'])
```

There are essentially three steps to this process:

1. The `fig` object uses the `plt.figure()` method to construct an instance of the `Figure` class.
2. The `ax` object uses the `fig` object from the previous line to construct an instance of the `Axes` class. The method being used here is `add_axes()`, which takes the dimensions of the axes.
3. The `ax` object calls `plot()` to actually generate the plot.

So, rather than using two objects at once (one for `Figure` and one for `Axes`) to generate or modify the plot, you just need to use `ax`. When it comes time to modify this plot, you simply need to reference `ax`. Any other plots you create in this manner will exist as their own separate objects, so there's no need to worry about which plot is the "active" one. Ultimately, you should prefer to use the object-oriented interface over the stateful interface in most cases, especially when you have multiple, complex plots that will need to be modified.

Note: This course will prefer the object-oriented interface over the stateful interface in its examples, though both are shown in some cases.

Guidelines for Visualizing Data with Python

Follow these guidelines when visualizing data with Python.

Visualize Data with Python

When visualizing data with Python:

- Use arguments to configure colors, styles, and add markers that make your data and insights stand out.
- Avoid the use of the color red in charts, as red is difficult to interpret for people who are color blind.
- Leverage online graph and chart galleries for inspiration about how to chart data.

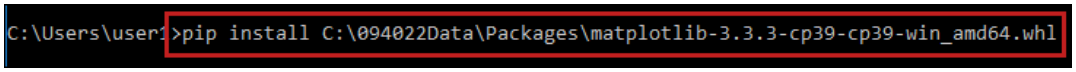
ACTIVITY 6-2
Visualizing Data with Python

Before You Begin
The **WWProject-L6** project is open in PyCharm.

Scenario
Now that the data has been prepared, you are ready to visualize it. You will use the Matplotlib library. You want to start simple, so you want to do sales by month. You are not sure if you want to use a line graph or a bar graph and will try both.

---

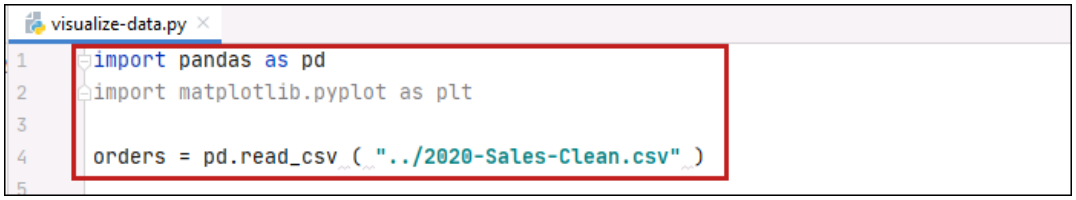
1. Install Matplotlib and pandas.
   a) Restore the Command Prompt.
   b) Type the following and press **Enter** to install Matplotlib.



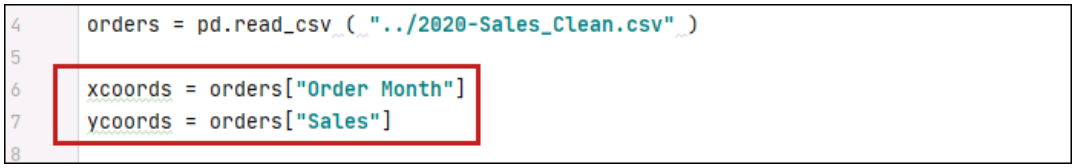
   c) Minimize, but do not close, the Command Prompt.

2. Create the **visualize-data.py** file in the **WWProject-L6** project.
   a) In the **Project** pane on the left, right-click the **WWProject-L6** project, and select **New→Python File**.
   b) In the **Name** box, type ***visualize-data*** and press **Enter**.

3. Plot a line graph for monthly sales data.
   a) In **visualize-data.py**, starting on line 1, type the following to import libraries and read from the clean .csv file.



   b) Starting on line 6, type the following to assign the values of the **Order Month** and **Sales** columns to variables.



c) Starting on line 9, type the following to plot the **Order Month** and **Sales** columns and show the results.

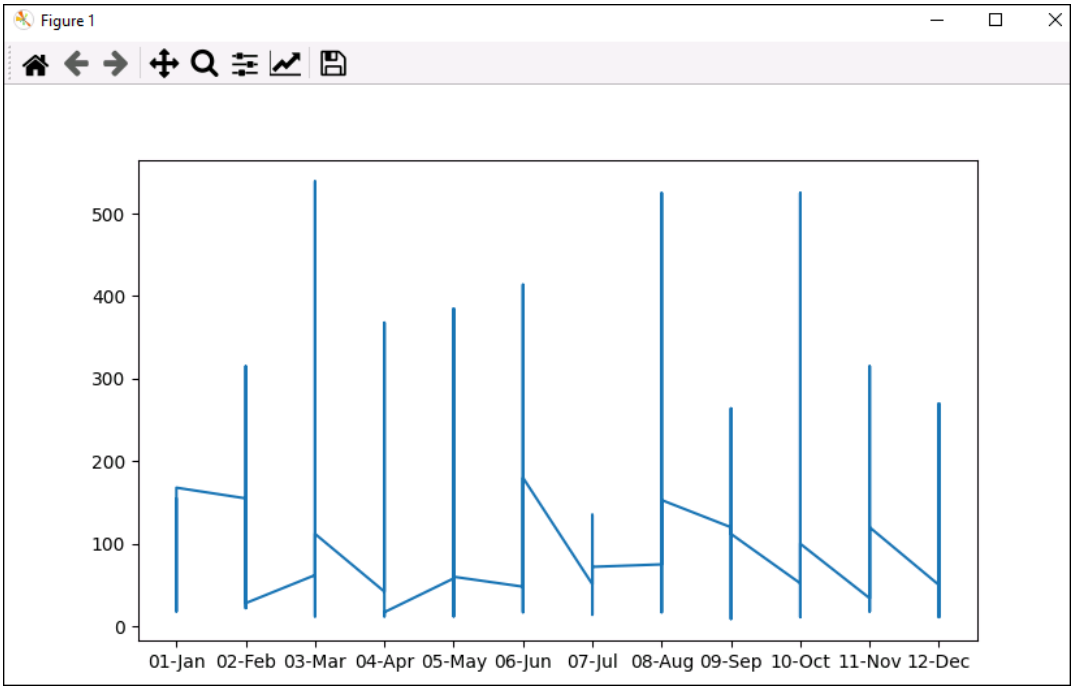
```
7   ycoords = orders["Sales"]
8
9   fig = plt.figure()
10  ax = fig.add_subplot(1,1,1)
11  ax.plot(xcoords,ycoords)
12  plt.show()
```

This is one of several object-oriented approaches to plotting.

- Line 9 creates a `Figure` object.
- Line 10 creates an `Axes` object at the first location in a 1x1 grid of subplots in the figure. You could create multiple subplots, but you are only creating one in this instance.
- Line 11 calls the `plot()` function on the `Axes` object, using the variables defined in lines 6 and 7.
- Line 12 shows the results of the plot.

d) Run **visualize-data.py**.

e) In the Figure 1 window, observe the line graph.



The line graph does not look correct because the detailed sales numbers are all being plotted and not being summarized per month.

> **Note:** If needed, expand the size of the Figure 1 window so that the month labels don't overlap each other.

f) Close the Figure 1 window.

g) In the **Run** pane, close the **visualize-data** tab.

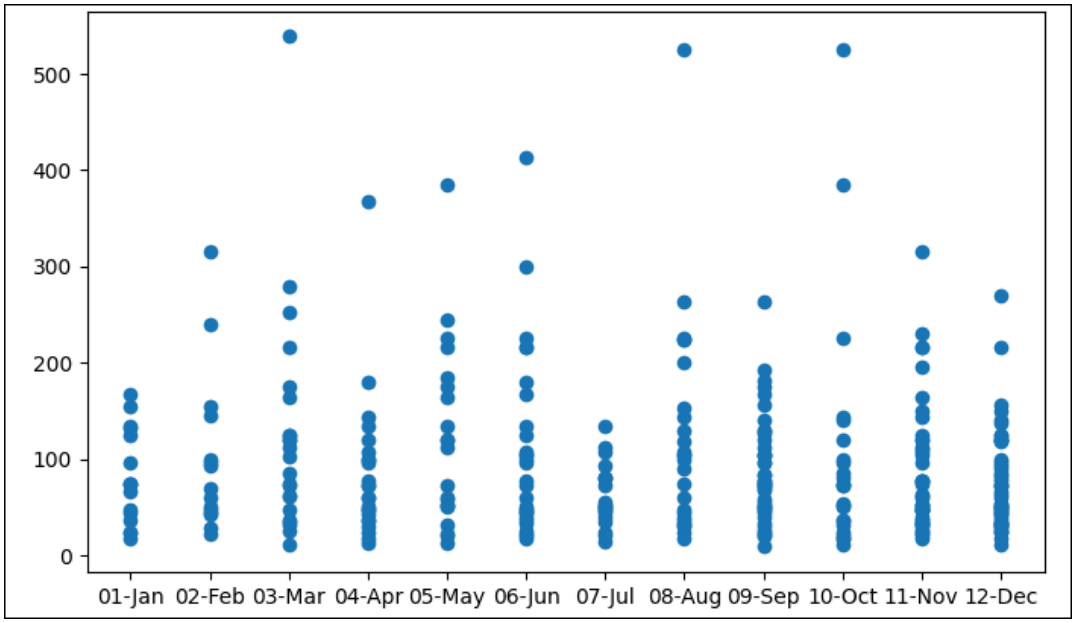
4. Add markers to the line graph.

a) On line 11, modify the `plot` method by adding the marker argument.

```
9   fig = plt.figure()
10  ax = fig.add_subplot(1,1,1)
11  ax.plot(xcoords,ycoords, "o")
12  plt.show()
13
```

b) Run **visualize-data.py**.

c) In the Figure 1 window, observe the marker graph.



This graph makes it much easier to see all of the different sales data points that are being plotted.

d) Close the Figure 1 window.

e) In the **Run** pane, close the **visualize-data** tab.

5. Summarize the data used in graph.

a) After reading the .csv file, starting on line 5, type the following to summarize the numbers in the dataframe by grouping the data by the **Order Month** field.

```
4   orders = pd.read_csv ( "../2020-Sales-Clean.csv" )
5   sumorders = orders.groupby(["Order Month"]).sum()
6
7   xcoords = orders["Order Month"].unique()
```

This will summarize the dataframe so there is one row per month and the numbers are the total for the entire month.

b) On lines 7 and 8, modify the coordinate variables to use summarized data.

```
5   sumorders = orders.groupby(["Order Month"]).sum()
6
7   xcoords = orders["Order Month"].unique()
8   ycoords = sumorders["Sales"]
9
10  fig = plt.figure()
```

These statements:

- Assign only the unique values from the Order Month column to the `xcoords` variable.
- Assign the sales values from `sumorders` to the `ycoords` variable.

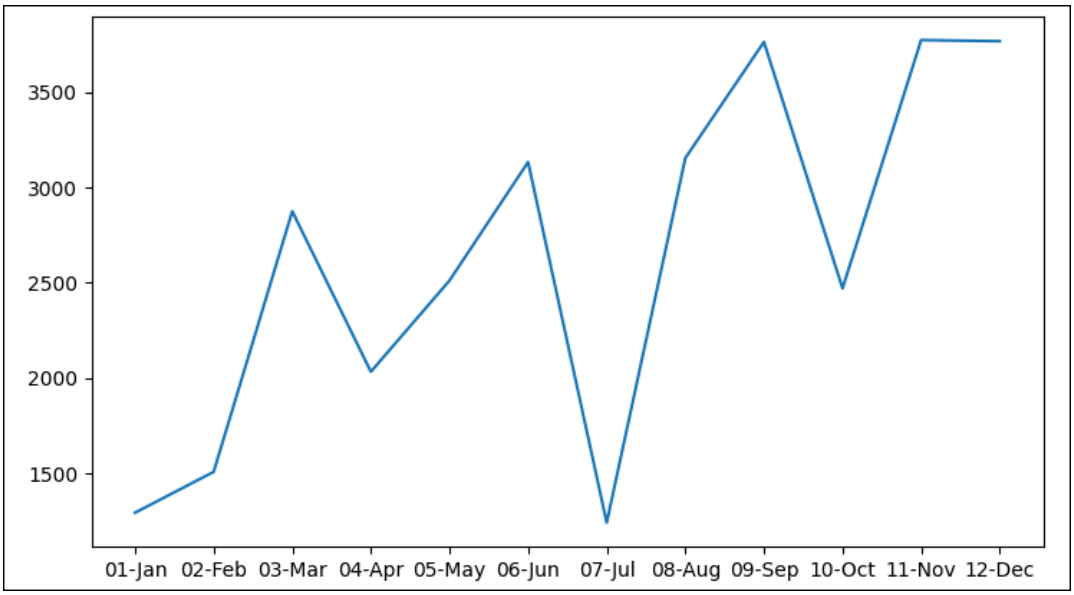
c) On line 12, modify the `plot` method by removing the marker argument.

```
10  fig = plt.figure()
11  ax = fig.add_subplot(1,1,1)
12  ax.plot(xcoords,ycoords)
13  plt.show()
```

Note: You are only removing the ,"o" from the line, not the entire line.

d) Run **visualize-data.py**.

e) In the Figure 1 window, observe the line graph.



With the summarized data, this graph is now legible.

f) Close the Figure 1 window.

g) In the **Run** console pane, close the **visualize-data** tab.

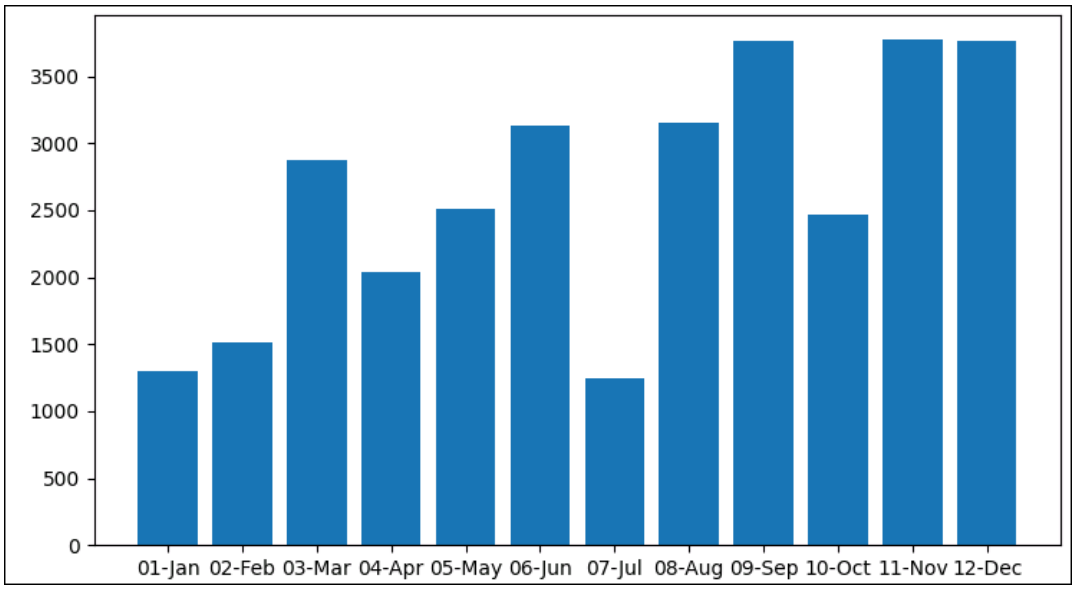
6. Change to a bar graph.

a) Delete lines 10 and 11, then modify the current line 10 (line 12 previously) to match the following.

```
8   ycoords = sumorders["Sales"]
9
10  plt.bar(xcoords, ycoords)
11  plt.show()
```

b) Run **visualize-data.py**.

c) In the Figure 1 window, observe the bar graph.



d) Close the Figure 1 window.

e) In the **Run** console pane, close the **visualize-data** tab.

f) Close the **visualize-data.py** editor tab.

TOPIC C

Perform Linear Regression with Machine Learning

Python can be used to perform direct statistical analysis on data. In this topic, you will perform linear regression with Python using machine learning.

Machine Learning

Machine learning applies artificial intelligence to automatically learn from experience in order to improve accuracy, operations, or other performance metrics without being explicitly programmed. Machine learning has similarities to computational statistics and is particularly well suited to data mining, where machine learning algorithms can be applied to large data sets to find meaningful insights.

Machine learning algorithms are the code behind machine learning that makes it function. These algorithms are designed to consume sample data, sometimes referred to as "training data," and from that create a model based on the sample data, and adjust as they are exposed to new data. In data mining, machine learning algorithms focus on properties and trends identified in the sample data, and then can discover previously unknown properties and trends.

A simple example of machine learning would be to feed a machine learning app 1,000 pictures of cats. Once trained, you can feed raw images into the apps, and it should be able to pick out images containing cats.

Whereas traditional statistical analysis draws insights and exposes trends from the sample of data analyzed, machine learning algorithms employed for statistical analysis seek to find predictable patterns which can be applied, to previously unknown criteria.

Taken together, the ability to analyze large data sets, to learn to find relevant properties and patterns in that data that may provide clarity to existing forecasting and trend models, or to find entirely new influencing properties and opportunities makes machine learning a powerful tool for forecasting and predictive modeling.

Additional Information

Ten examples of how machine learning is used in real life: **https://bigdata-madesimple.com/top-10-real-life-examples-of-machine-learning/**

UCI Machine Learning Repository: **https://archive.ics.uci.edu/ml/index.php**

Kaggle is a no-setup, customizable, Jupyter Notebooks environment. Access free GPUs and a huge repository of community published data & code: **https://www.kaggle.com/**.

Supervised vs. Unsupervised Learning

When working with machine learning algorithms, you have to decide how you will train your algorithm to recognize the data that you're looking for. There are three general ways you can train your algorithm:

- Supervised learning, where you provide a labeled dataset that the algorithm uses to validate the accuracy of the data it extracts.
- Unsupervised learning, where you provide training data without labels, and the algorithm looks for features and patterns based on the algorithm designed to pull out relevant data.
- Semi-supervised learning, where you provide a small subset of labeled data, to prime that algorithm as it evaluates a much larger set of data in conjunction with the subset.

Supervised learning is helpful when dealing with classification problems and regression issues. With classification issues, the algorithm must identify data as belonging to a group or class, whereas regression issues examine continuous data to find or predicate related values. With both issues, you typically have lots of data to work with. If you want to classify customers based on the criteria you have, you can train the algorithm by providing data on your customers. Supervised learning works best when you have lots of data to train the algorithm.

If you don't have data to work with, or don't know the questions you wish to ask, and, therefore, the data you wish to collect, unsupervised learning may be a better choice. These circumstances are not uncommon. You may know the questions you want answered, but might not have a clean dataset with which to train the algorithm. If you're researching something new or breaking into a new market segment, you may not know the questions you wish to ask. With unsupervised learning, the algorithm attempts to find the structure and extract useful information.

Scikit-Learn Library

All the reasons that make Python a great programming language for data science, make it a great tool for forecasting and predictive modelings. Python has many libraries that allow you to apply machine learning to your projects. One of the most popular for predictive data analysis is scikit-learn, which is a free, open source library built on NumPy, SciPy, and matplotlib, that can be used commercially and has a large community.

Scikit-learn provides the following capabilities.

Capability	*Description*	*Application*
Classification	Identify a category to which something belongs.	Image recognition, SPAM email detection, etc.
Regression	Predict continuous-value attributes associated with an object.	Stock prices, drug responses, etc.
Clustering	Grouping of similar objects.	Customer segmentation, grouping of entities in experiments.
Dimensionality reduction	Reducing the number of random variables to consider.	Data visualization, increase efficiency in analysis and processes.
Model selection	Comparison of models, selection, and validation of parameters and models.	Accuracy improvement through tuning models and adjusting parameters.
Preprocessing	Feature extraction and normalization.	Transforming input data for use with machine learning algorithms.

Scikit-Learn Modules

Some of the modules available in scikit-learn are:

- `.cluster` and `.cluster.bicluster` for cluster and bicluster applications.
- `.covariance` for covariance estimation.
- `.decomposition` for working with decomposition.
- `.cross_decomposition` to compare cross decomposition methods.
- `.datasets` for working with dataset analysis.
- `.tree` for working with decision trees.
- `.ensemble` for comparing base models to find a better model.

- `.linear_model` for implementing linear models for analysis and forecasting.
- `.model_selection` for implementing train and testing indices to split data into training and testing sets.

Linear Models Supported

For additional information on linear models supported, see: **https://scikit-learn.org/stable/modules/classes.html**.

Additional Information

For more information about scikit-learn, see: **http://scikit-learn.org**.

For more information on common methods, see: **https://scikit-learn.org/stable/auto_examples/**.

.train_test_split

The `.train_test_split` method splits an array or matrix into random train and test subsets. The method returns a list containing the train-test split of inputs. The method has the following parameters.

Parameter	*Description*	*Usage*
`*arrays`	Sequence of indexable data with same length / shape[0]	Supported inputs are lists, numpy arrays, scipy-sparse matrices, and pandas dataframes.
`test_size`	Float or Int, default=None	Sets the sizes of the test data set, see Additional Information.
`train_size`	Float or Int, default=None	Sets the sizes of the training data set, see Additional Information.
`random_state`	Int, RandomState instance or None, default=None	Controls the shuffling applied to the data before applying the split.
`shuffle`	Boolean, default=True	Shuffle data before splitting. If False, Stratify must be None.
`stratify`	Array-like, default-None	If not None, data is split in a stratified fashion, using this as the class labels.

Additional Information

For additional information, see: **https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html**.

.train_test_split Example

For example, when desiring to perform linear regression on employees to find salary by time of employment, you might use the following code to split the data into training and test data sets. This code does the following:

- `X_train` and `y_train` make the training data set and `X_test` and `y_test` make the test data set.
- `train_test_split` splits the arrays into random train and test subsets.
- `random_state` specifies a particular output instance to use. If you do not specify `random_state`, then the results will be different each time you run your code. By using `random_state=12`, your results will match the images in this course.

```
#split data into training and test data sets
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=12)
```

Regression

Regression is a type of statistical analysis that looks for relationships between variables. Regression is generally performed to find out how a specific phenomenon influences other related variables. Basically, regression seeks to answer the question, does a specific predictor (x) do a good job of predicting another value (y)? For example, you might perform regression to find the impact the number of years of service has on salary of employees to answer the question, is an employee's tenure with the company a strong predictor of a high salary?

When performing regression analysis, you generally consider one phenomenon of interest and have a variety of observations. Observations have two or more features. The assumption is that at least one feature is dependent on others, and so you attempt to establish a relationship between them to gain insights.

.LinearRegression

Linear regression is one of the most widely used approaches to regression. It is one of the simplest regression methods where you attempt to find a relationship between two variables.
The `.LinearRegression` method from the `.linear_module` performs ordinary least squares linear regression.

The `.LinearRegression` method accepts the following parameters.

Parameter	*Description*	*Usage*
`fit_intercept`	Boolean, default=True	Determines if the intercept will be calculated for the model.
`normalize`	Boolean, default=False	Ignored if `fit_intercept` is False. The regressors X will be normalized before regression.
`copy_X`	Boolean, default=True	If True, X will be copied, otherwise it may be overwritten.
`n_jobs`	Int, default=None	Number of jobs to use for the computation.
`positive`	Boolean, default=False	When set to True, forces coefficients to be positive.

It supports the following attributes.

Attribute	*Description*	*Usage*
coef_	Array of shape (n_features,) or (n_targets, n_features)	Estimated coefficients for the linear regression problem.
rank_	Int	Rank of the matrix X. Available when X is dense.
singular_	Array of shape (min(X, y),)	Singular values of X. Available when X is dense.
intercept_	Float or array of shape (min(X, y),)	Independent term in the linear model. Set to 0.0 if `fit_intercept` is False.

Additional Information

For additional information, see: **https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html**.

.LinearRegression Example

In the following example code, the `.LinearRegression` method is used to compare the impact of employee years of service to salary.

The following code:

- Instantiates the `LinearRegression` class.
- Uses the `fit` linear regression method on the training data.

```
#fit linear regression to train data set
regressor = LinearRegression()
regressor.fit(X_train,y_train)
```

The following line predicts the results by defining the `y_pred` array which will contain all the predicted values for the input values in the `X_test` series.

```
#predict the results
y_pred = regressor.predict(X_test)
```

The following code compares the actual output values for `X_test` with the predicted values by creating a dataframe with the actual salary numbers in the `y_test` variable, and the predicted salary numbers in the `y_pred` variable, and then prints them.

```
df = pd.DataFrame({"Actual": y_test, "Predicted": y_pred})
print(df)
```

You can then add code to view predicted versus actual results.

Guidelines for Performing Linear Regression with Machine Learning

Follow these guidelines when performing linear regression with machine learning.

Perform Linear Regression with Machine Learning

When performing linear regression with machine learning:

- If you have a lot of data to train the machine learning algorithm, use a supervised learning approach.
- Use linear regression to find the relationship between two variables.
- When performing linear regression, focus on one phenomenon of interest to get a variety of observations.

ACTIVITY 6-3

Performing Linear Regression with Machine Learning

Data File

C:\094022Data\Programming Python for Data Science\Salary.csv

Before You Begin

The **WWProject-L6** project is open in PyCharm.

Scenario

Woodworkers Wheelhouse employs workers who perform installations for various products that are sold by the store. These installers range from the drivers who deliver wood and lumber, to carpenters who perform jobs such as kitchen remodels. Management wants to analyze the salaries of these employees based on their years of experience. You will use scikit-learn to perform a linear regression using the experience and salary data.

---

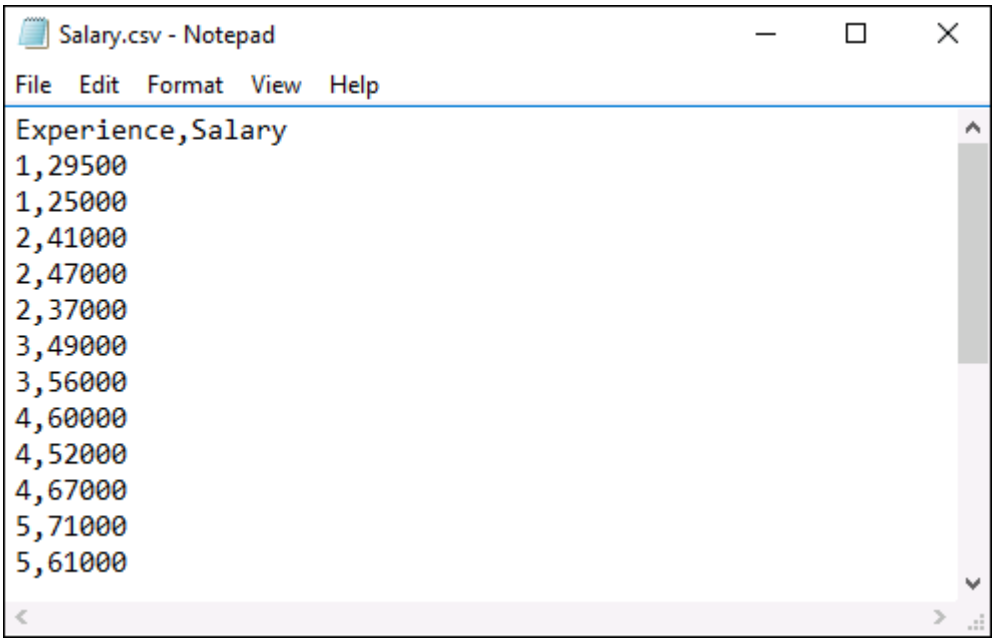
1. Install scikit-learn.
   a) Restore the Command Prompt.
   b) Type the following and press **Enter** to install scikit-learn.

```
C:\Users\user1>pip install C:\094022Data\Packages\scikit_learn-0.24.0-cp39-cp39-win_amd64.whl
```

   c) When the install is complete, close the Command Prompt.

2. Review **WW-2020-Sales.csv** file.
   a) Open File Explorer and navigate to **C:\094022Data\Programming Python for Data Science**.
   b) Open **Salary.csv** with Notepad.

c) Observe that the file contains years of experience and salary data for the installers.



It is a very simple data set that has columns for years of experience and salary.

d) Close Notepad.

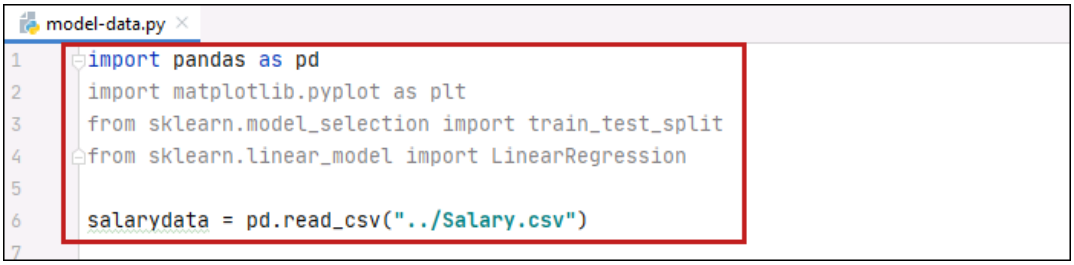
3. Create the **model-data.py** file in the **WWProject-L6** project.

a) In the **Project** pane on the left, right-click the **WWProject-L6** project, and select **New→Python File**.

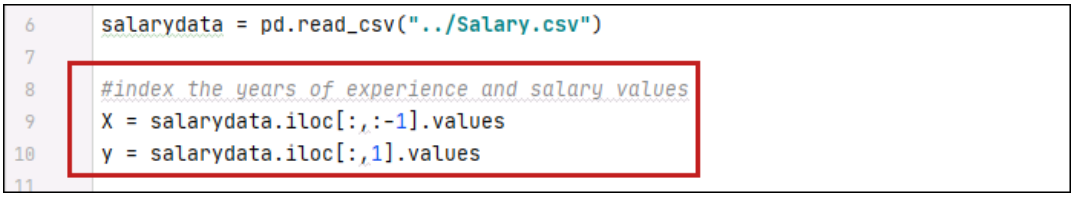
b) In the **Name** box, type ***model-data*** and press **Enter**.

4. Create the predictive model.

a) In **model-data.py**, starting on line 1, type the following to import libraries and read from the **Salary.csv** file.



b) Starting on line 8, type the following to index the years of experience and salary values.



The X variable contains the values from the **Years of Experience** column and the y variable contains the values from the **Salary** column.

c) Starting on line 12, type the following to split the data into training and test data sets.

```
10  y = salarydata.iloc[:,1].values
11
12  #split data into training and test data sets
13  X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=104)
14
```

- `X_train` and `y_train` make the training data set and `X_test` and `y_test` make the test data set.
- `train_test_split` splits the arrays into random train and test subsets.
- `random_state` specifies a particular output instance to use. If you do not specify `random_state`, then the results will be different each time you run your code. By using `random_state=104`, your results will match the images in this course.

d) Starting on line 15, type the following to fit linear regression to the train data set.

```
13  X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=104)
14
15  #fit linear regression to train data set
16  regressor = LinearRegression()
17  regressor.fit(X_train,y_train)
18
```

- Line 16 instantiates the `LinearRegression` class.
- Line 17 uses the `fit` linear regression method on the training data.

e) Starting on line 19, type the following to predict the results.

```
17  regressor.fit(X_train,y_train)
18
19  #predict the results
20  y_pred = regressor.predict(X_test)
21
```

Defines the `y_pred` array which will contain all the predicted values for the input values in the `X_test` series.

f) Starting on line 22, type the following to compare the actual output values for `X_test` with the predicted values.

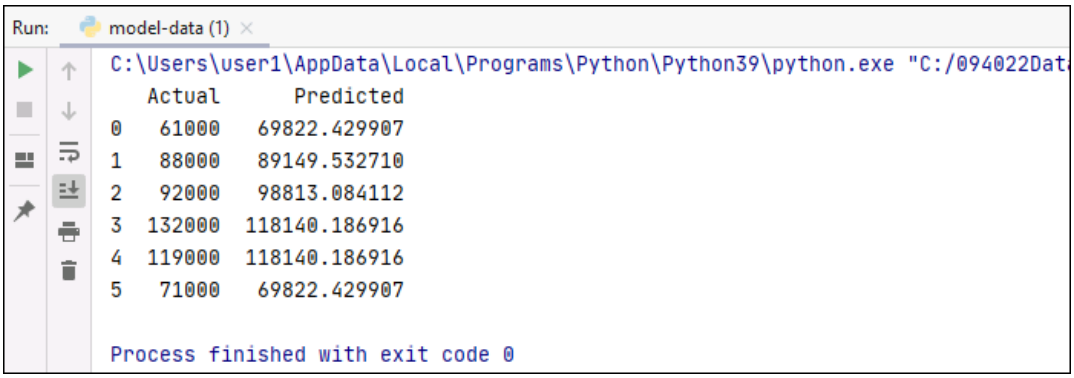
```
20  y_pred = regressor.predict(X_test)
21
22  df = pd.DataFrame({"Actual": y_test, "Predicted": y_pred})
23  print(df)
24
```

Creates a dataframe with the actual salary numbers in the `y_test` variable, and the predicted salary numbers in the `y_pred` variable, and then prints them.

5. Run **model-data.py** and view the results.

a) Run **model-data.py**.

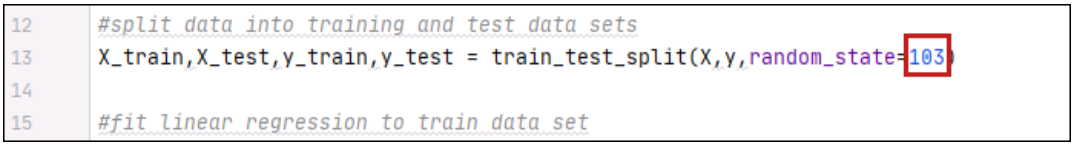
b) In the **Run** pane, observe the actual and predicted results.



c) In the **Run** pane, close the **model-data** tab.

6. Use different random results.

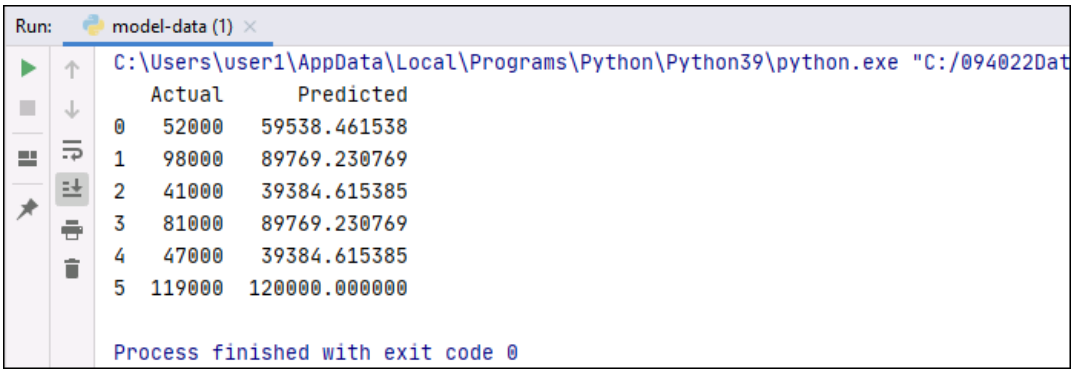
a) On line 13, change the `random_state` to 103.



Changing the `random_state` will give you a slightly different result.

b) Run **model-data.py**.

c) In the **Run** pane, observe the actual and predicted results.



d) In the **Run** pane, close the **model-data** tab.

7. Visualize the linear regression.

a) Starting on line 25, type the following to create a figure with two horizontally stacked subplots.

```
23  print(df)
24
25  #Visualize the training and test results
26  fig, (ax1, ax2) = plt.subplots(1, 2)
27  ax1.scatter(X_train,y_train,color="red")
28  ax1.plot(X_train,regressor.predict(X_train))
29  ax1.set_title("Salary vs Experience - Train Data")
30  ax1.set_xlabel("Experience in years")
31  ax1.set_ylabel("Salary")
32
```

- Line 26 creates a figure with two horizontally stacked subplots.
- Line 27 creates a scatter plot in the first subplot using the training data.
- Line 28 creates a line plot in the first subplot using the predicted training data.
- Lines 29-31 creates a title and axis labels for the first subplot.

b) Starting on line 33, type the following to create the second subplot in the figure.

```
31  ax1.set_ylabel("Salary")
32
33  ax2.scatter(X_test,y_test,color="red")
34  ax2.plot(X_train,regressor.predict(X_train))
35  ax2.set_title("Salary vs Experience - Test Data")
36  ax2.set_xlabel("Experience in years")
37  plt.show()
38
```

- Line 33 creates a scatter plot in the second subplot using the test data.
- Line 34 creates a line plot in the first subplot using the predicted training data.
- Lines 35 and 36 create a title and axis labels for the first subplot.
- Line 37 shows the figure.

8. Run **model-data.py** and view the visualization.

a) Run **model-data.py**.

b) In the Figure 1 window, observe the two subplots.

The subplot on the left shows the data used to train the algorithm, and the subplot on the right shows the predicted data.

	Note: You may need to resize the Figure 1 window in order to view both subplots correctly.

c) Close the Figure 1 window.

d) In the **Run** pane, close the **model-data** tab.

9. Close running apps.

a) In the **Run** pane, close all tabs.

b) Select **File→Close Project**.

Summary

In this lesson, you cleaned data with Python so that it could be analyzed efficiently and effectively using NumPy and pandas. Next, you visualized insights from data analysis using matplotlib. Finally, you used machine learning from scikit-learn to compare predicted results of linear regression to actual results.

How might you use Python for data science analysis in your organization?

How might you use machine learning in Python in your organization?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

7 Implementing Unit Testing and Exception Handling

Lesson Time: 2 hours

Lesson Introduction

Before your applications will be ready for people to use, you must ensure that they run error free and can handle any user, system, or data issues you can anticipate. In this lesson, you will handle exceptions and write and execute unit tests.

Lesson Objectives

In this lesson, you will:

- Add exception handling to a program.
- Write a unit test for Python app.
- Execute a unit test.

TOPIC A

Handle Exceptions

In order for your application code to execute, it must be able to handle any problems that arise while it is running. In this topic, you will handle exceptions through your code.

Exceptions in Programming and Python

Code you write for an app is really a set of instructions. You're telling the computer how to store and process inputs, perform computations on those inputs and any other data your app uses, and how to generate output. But your code can only do what you've told it to do; if something unanticipated comes up, how will your code respond? In programming, these types of events are known as exceptions. When an exception is raised, program execution is stopped and any lines left to run after the exception are not executed. As a programmer, it's your job to write code that can gracefully handle exceptions and continue to function using facilities offered by the `BaseException` superclass and its subclasses.

In object-oriented programming, exceptions are objects. There are multiple exception classes available that you can leverage, and you can define your own exception classes if you like. All exception classes inherit from the built-in `BaseException` superclass. You've likely seen Python's built-in exception handling already anytime you typed your code wrong and gotten a syntax error.

Exception Example

For example, if you write a simple app to perform division by taking two numbers input by a user, the first the numerator, the second the denominator, and print the quotient, then you have to account for the fact that numbers cannot be divided by zero. If a user enters a zero as the denominator, Python will throw a `ZeroDivisionError`, and the program will end.

```
#Capture numbers to be divided.

num = input( 'Enter the number to be divided: ')
denom = input( 'Enter the number to divided by: ')

# Perform division operation

quo =  float(num) /  float(denom)

# Print the output

print( 'The quotient of {0} divided by {1} is: {2}'.format(num , denom , quo))
```

If a user enters a zero as the `denom` variable, the built-in Python exception, `ZeroDivisionError` is raised.

To ensure your apps function properly, you must detect problems by raising exceptions, and handle the problems in a graceful way in your code.

Python Built-in Exceptions

Python has the following built-in exceptions.

Exception	*Description/When Raised*
`AssertionError`	Raised when an ***assertion*** statement, to test if a condition is true, fails.

Exception	Description/When Raised
`AttributeError`	Raised when assigning or referencing an attribute fails.
`EOFError`	Raised when the `input()` function comes to an end-of-file condition without being able to read any data.
`GeneratorExit`	Raised when a `generator` or `coroutine` (iteration routine) is closed.
`ImportError`	Raised when `import` statement cannot successfully load a module.
`IndexError`	Raised when a sequence subscript is out of range (if an index is not integer, a `TypeError` is raised).
`KeyError`	Raised when a key (dictionary) cannot be mapped to existing keys.
`KeyboardInterrupt`	Raised when the user interrupts code execution by hitting CTRL-C or DEL.
`MemoryError`	Raised when the operation runs out of memory.
`NameError`	Raised when the name (either global or local) is not found.
`NotImplementedError`	Derived from the `RuntimeError`, raised when a called derived class needed to override a method is yet to be implemented.
`OSError`	Raised when the operating system returns an OS error such as disk not found.
`OverflowError`	Raised when the result of math operations is too large to be represented.
`RecursionError`	Derived from `RuntimeError`, this is raised when the interpreter detects maximum recursion depth is exceeded.
`ReferenceError`	Raised when a weak reference proxy, created by the `weakref.proxy()` function, accesses an attribute of the referent after it has been garbage collected.
`RuntimeError`	Raised when errors occur that don't fall into other categories.
`StopIteration`	Raised to signal there are no further items produced by the iterator `(next()` and `iterator_next()`).
`StopAsyncIteration`	Raised by `_anext_()` to stop iteration.
`SyntaxError`	Raised when the parser encounters an error in code syntax.
`IndentationError`	Raised when incorrect indentation is encountered in the code.
`SystemError`	Raised when the interpreter encounters an internal error.
`SystemExit`	Raised by `sys.exit()` function when the code cannot be cleanly exited.
`TypeError`	Raised when an operation is applied to the wrong type of object such as adding string data.
`UnboundLocalError`	Raised when a reference is made to local variable in a function or method that has no value.
`UnicodeError`	Raised when a Unicode error is encountered.
`UnicodeEncodeError`	Raised when a Unicode error occurs during encoding.
`UnicodeDecodeError`	Raised when a Unicode error occurs during decoding.

Exception	Description/When Raised
`UnicodeTranslateError`	Raised when a Unicode error occurs during translation.
`ValueError`	Raised when an operation or function received an argument of the correct data type, but inappropriate value.
`ZeroDivisionError`	Raised when the second argument, the denominator, is zero.

Additional Information

For additional information on Python built-in exceptions, see: **https://docs.python.org/3/library/exceptions.html**.

Exception Raising

In the previous example, one of Python's built-in exceptions is triggers. However, as a developer you may have to account for problems and issues that won't be caught by Python's built-in exceptions. The first step is to raise an exception which is done using the keyword `raise`. For example, if you want to restrict a numeric value input to those above zero, there are no Python built-in exceptions that will catch that, so you would use the following code to verify the data entered and raise an exception if the data does not meet requirements:

```
num = int(input('Enter a number greater than zero: '))

if num <=  0:
   raise  Exception( "Numbers must be larger than zero.")

else:
   print(num)
```

Note: The code `int(input())` defines the input to be an integer. Entering a non-integer will cause Python to raise a built-in exception.

Try...Except Blocks

As you can see when exceptions are raised, the error message is presented to the user, and the code exists. In the course of using any app, people will make mistakes. They will input the wrong data, click a button before all information is entered, in short, mistakes will be made. As a programmer, it's your job to anticipate those mistakes and write your code to gracefully handle exceptions that are raised. Having said that, you should understand that not all errors can be handled. Syntax errors, type errors, and value errors are examples of exceptions you can likely handle gracefully in your code. Exceptions like out of memory errors, system errors, and OS errors cannot, in most cases, be gracefully handled by your code.

In Python, the way you handle exceptions is by using the `try`, `except`, `else`, and `finally` blocks of code.

The `try` block lets you test code for errors.

- It executes the statements between the `try` and `except` keywords.
- If no exception is encountered, the `except` clause is skipped, and the execution of the `try` statement completes.
- If an exception is encountered during execution of the `try` clause, the rest of the `try` clause is skipped. If the exception encountered matches one named in one of the `except` statements, that `except` clause is executed, and execution continues after the `try` statement.

- If the exception is not matched to a named `except` statement, it is passed to other `try` statements, and if no handler is found, the exception is unhandled and program execution stops with the relevant error.

The `except` block is where you write code to handle errors you discover.

- You can have more than one `except` clause to specify handlers for different built-in and programmer-defined exceptions.
- You can use tuples to define multiple exceptions in a single statement, if you desire.
- Only one handler will be executed.
- Include the `as` expression in `except` statements to store the `except` object that is raised in a variable to use in the `except` block.
- Only handle exceptions from the corresponding `try` block.

You can optionally include an `else` clause which must follow the `except` clauses. It is executed if no exceptions are found. It's important to understand the order of operations, the `else` block is only executed if nothing in the `except` block is executed, and if anything in the `except` block is executed, the `else` block is not executed. This is where you may opt to put your original intended processing operations.

The `finally` block executes after all other blocks. This block is optional and is used for code that you always want to run regardless of what happens in the `try`, `except`, and `else` blocks.

Try...Except: Example

For example, if we

```
#Capture numbers to be divided.

num =  input(  'Enter the number to be divided: ')
denom = input(  'Enter the number to divided by: ')

# Perform exception handling

try:
   quo =  float(num) /   float(denom)  #Statements in the try clause must be
indented under try.

except ZeroDivisionError:
   print( "You cannot divide by zero, enter a non-zero number to divide by.")
#You can add other exceptions after this one if you choose.

else:
   # Perform division operation

   quo =   float(num) /   float(denom)

   # Print the output

   print( 'The quotient of {0} divided by {1} is: {2}'.format(num  , denom  ,
        quo))

finally:
   print( 'Thank you for dividing with us!')
```

Note: Note that the statements under each block must be indented. If you fail to indent a statement under one block, Python may raise a syntax error when moving to the next block.

Exception Hierarchy

The following figure shows some of the major types of exceptions spread across important base class types. Depending on the complexity of your app, understanding where exceptions live in the hierarchy can help you know how to catch and handle exceptions

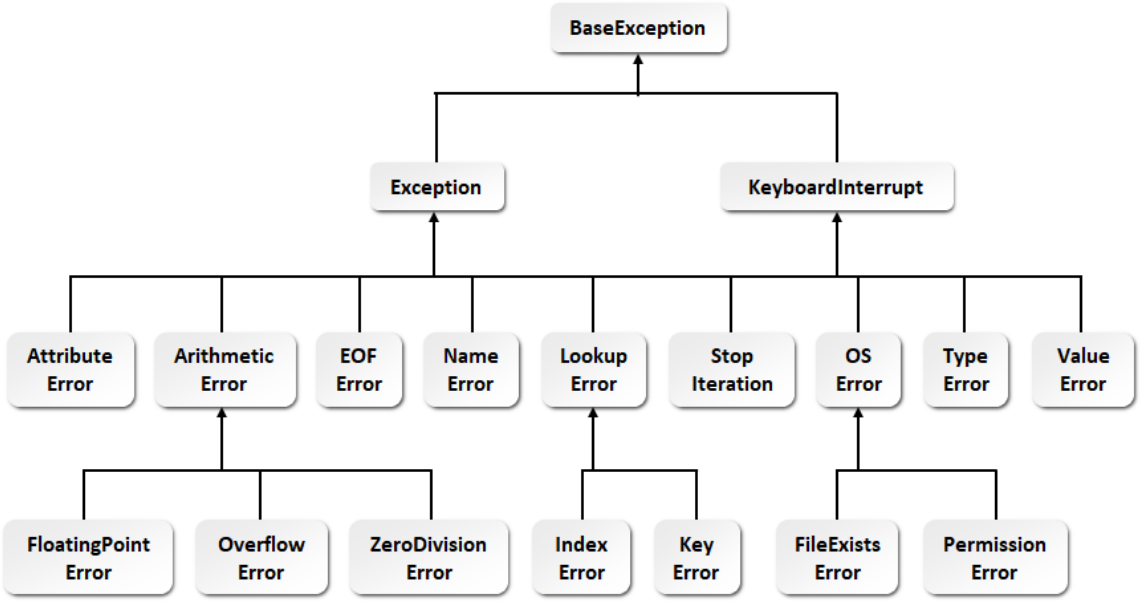


Figure 7-1: Exception hierarchy.

Python Exception Library

Exception hierarchy: **https://docs.python.org/3/library/exceptions.html**.

Guidelines for Adding Exception Handling to a Program

Note: All Guidelines for this lesson are available as checklists from the **Checklist** tile on the CHOICE Course screen.

Follow these guidelines when adding exception handling to a program.

Add Exception Handling to a Program

When adding exception handling to a program:

- Use exception handling routines to catch exceptions and gracefully handle them within your app.
- Use try, except, else, and finally to test inputs and data for required conditions.
- Familiarize yourself with the Python exception hierarchy when trying to determine why exceptions are being raised in your apps and how to handle them.

ACTIVITY 7-1
Testing for Exception Handling Failures

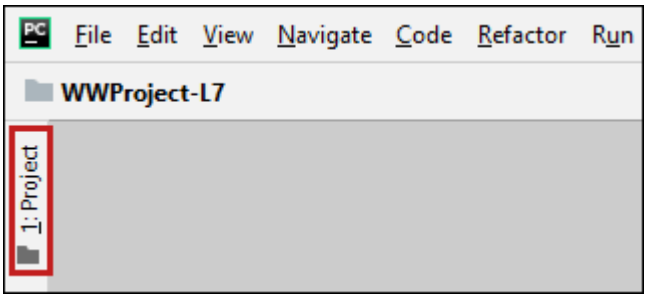
Data Files
All project files in C:\094022Data\Implementing Unit Testing and Exception Handling \WWProject-L7.

Before You Begin
PyCharm is open.

Scenario
Before you release your **Wood Delivery Calculator** app, you want to test it to see if any errors or odd behavior occurs. You will perform actions out of order and try to enter invalid values where possible.

---

1. Open the **WWProject-L7** project.
   a) In the Welcome to PyCharm window, select **Open**.
   b) Navigate to the **C:\094022Data\Implementing Unit Testing and Exception Handling\WWProject-L7** folder.
   c) With the **WWProject-L7** folder selected, select **OK**.
   d) If the **Project** pane is not visible, select the **1: Project** tab.

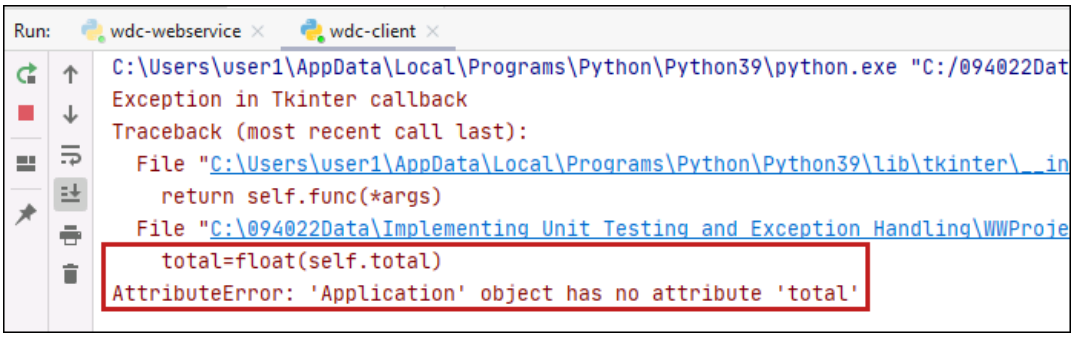


   e) In the **Project** pane on the left side of the PyCharm window, verify that **WWProject-L7** is listed.

2. Run the **Wood Delivery Calculator** web service and client.
   a) In the **Project** pane on the left, double-click **wdc-webservice.py**.
   b) In the **Project** pane on the left, double-click **wdc-client.py**.
   c) Run the **wdc-webservice.py** script.
   d) Run the **wdc-client.py** script.
   The Wood Delivery Calculator window should appear.

3. Test submitting an order before calculating the total.
   a) Enter a quantity for any product but do NOT select **Calculate Price**.
   b) Select **Submit Order**.

c) In the **Run** console pane, in the **wdc-client** tab, observe the **AttributeError** information.



The code is trying to convert the value of `self.total` to a float and assign that to the total variable. The error states that self does not have the `total` attribute.

d) Close the Wood Delivery Calculator window.

	Note: Leave the **wdc-webservice.py** script running and do not close the **Run** pane.

e) In the **wdc-client.py** tab, scroll to the **submit** function, and on line 136, review the code.

```
133     def submit(self):
134         """Insert order information to wworders database"""
135         ship=self.ship.get()
136         total=float(self.total)
137         send={'ship': ship, 'total': total}
138         posttotal = requests.post("http://127.0.0.1:5000/submit", json = send)
139         self.lblsubmit.set("Order number "
140                            +str(posttotal.text)+" submitted")
141
```

The highlighted line is where the error occurred.

f) Scroll to the **calculate** function, and on line 117, review the code.

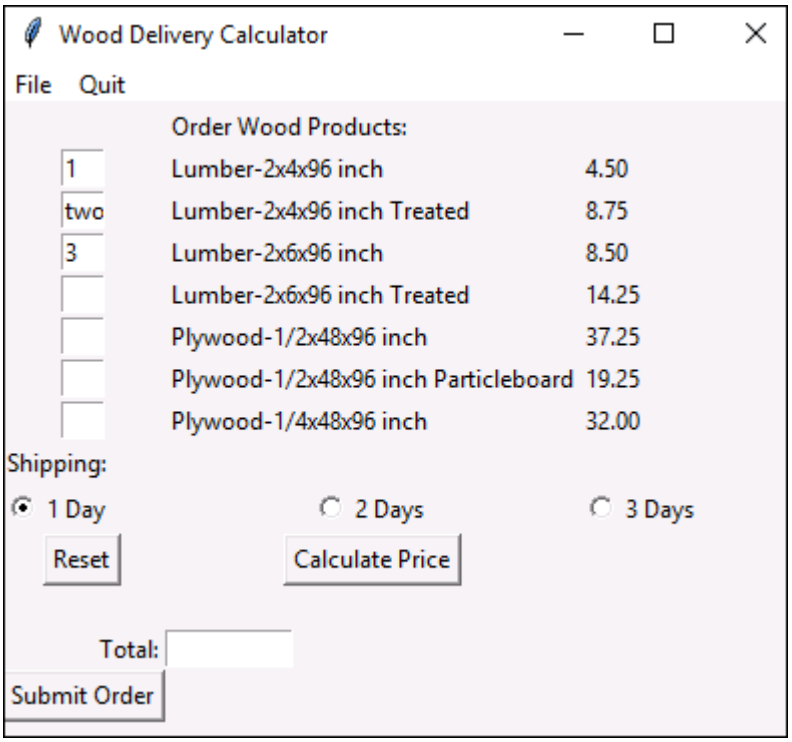
```
115     def calculate(self):
116         """Event handler for the calculator"""
117         self.total = 0
118         for i in range(len(self.products)):
119             if (self.entry[i].get()):
120                 self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
121
```

The highlighted line is where the `total` attribute is defined for `self`. This is why you receive an error if you select **Submit** before selecting **Calculate**. You will need to decide how best to address this. You could define `self.total` initially in the client app and this would avoid the error, but it would allow for the blank order to be submitted. Ideally, the user would be prompted to specify an order that had at least one product selected and had a calculated total.

4. Test entering a non-numeric value for an order quantity of a product.

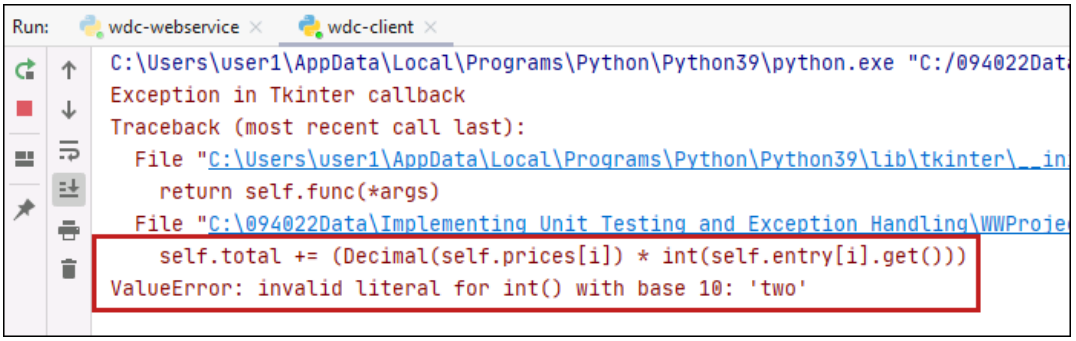
a) Run the **wdc-client.py** script.
The Wood Delivery Calculator window should appear.

b) Specify the quantity of products to match the image below.



c) Select **Calculate Price**.

d) In the **Run** console pane, in the **wdc-client** tab, observe the **ValueError** information.



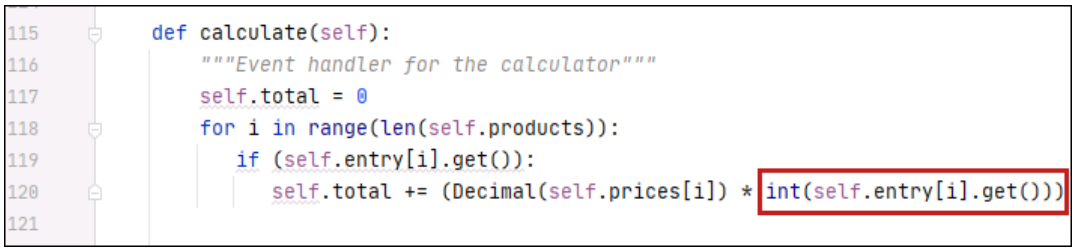
The code is trying to convert the value of `self.entry[i]` to an integer. The error states that the value of 'two' can't be converted to an integer because it is not a numeric value.

e) Close the Wood Delivery Calculator window.

f) Select the **wdc-webservice** tab, and then select the **Stop 'wdc-webservice'** button.

g) In the **Run** console pane, close all tabs.

h) In the **wdc-client.py** tab, in the **calculate** function, on line 120, review the code.



```
115     def calculate(self):
116         """Event handler for the calculator"""
117         self.total = 0
118         for i in range(len(self.products)):
119             if (self.entry[i].get()):
120                 self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
121
```

The highlighted line is where the error occurred. Trying to convert a non-numeric value to an integer or perform a calculation with it will generate an error. You will need to decide how best to address this. You could either check the value of the entry when it is made or when the calculation is performed.

ACTIVITY 7-2
Adding Exception Handling to a Program

Before You Begin
The **WWProject-L7** project is open in PyCharm.

Scenario
Now that you have identified two issues with the **Wood Delivery Calculator** app, you will implement some exception handling to address them. You will address the non-numeric quantity values by checking them when the calculation is performed and display a note to the user that the value is invalid and needs to be a whole number. For submitting a blank order, you will define `self.total` in the initial constructor, and then check if the order total is zero when submitting the order. You will also verify if any products have been ordered.

---

1. Handle non-numeric product quantities.

a) In the **wdc-client.py** tab, in the `calculate` function, remove the code on line 120 that calculates the `self.total`.

```
115     def calculate(self):
116         """Event handler for the calculator"""
117         self.total = 0
118         for i in range(len(self.products)):
119             if (self.entry[i].get()):
120                 self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
121
```

b) On line 120, type the following to evaluate the value for each quantity entry and display a message if it is not an integer.

```
118         for i in range(len(self.products)):
119             if (self.entry[i].get()):
120                 try:
121                     int(self.entry[i].get())
122                 except:
123                     self.lblsubmit.set("'" + self.entry[i].get() + "'" +
124                                        " is not valid entry. "
125                                        "Must be a whole number.")
126                     return
127                 else:
128                     self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
129
```

With these statements:

- The `try` clause tries to convert the value of each entry to an integer.
- The `except` clause uses the submit label to display a message stating which value is invalid and that a whole number is required. It then returns back from the `calculate` function and allows the user to make the necessary changes.
- The `else` clause proceeds with the `calculate` function flow and calculates the price for that product.

2. Handle selecting Submit before selecting Calculate.

a) Under the `Application` class, starting on line 11, type the following to define `self.total` and set the initial value to zero.

```
7   class Application(Frame):
8
9       def __init__(self, master):
10          super(Application, self).__init__(master)
11          self.total = 0
12          self.grid()
```

b) In the **submit** function, on line 146, replace the code that sends the `POST` request to the web service with the following code.

```
142     def submit(self):
143         """Insert order information to wworders database"""
144         ship=self.ship.get()
145         total=float(self.total)
146         if total <= 0:
147             raise Exception("Select Calculate Price button before submitting order.")
148         else:
149             send={'ship': ship, 'total': total}
150             posttotal = requests.post("http://127.0.0.1:5000/submit", json = send)
151             self.lblsubmit.set("Order number "
152                                +str(posttotal.text)+" submitted")
153
```

With these statements:

- The `if` statement checks the total variable to see if it equals zero.
- If the `total` variable equals or is less than zero, then an exception is raised stating that the user needs to select the Calculate button before submitting an order.
- The `else` clause proceeds with the preexisting code to send the `POST` request to the web service.

3. Handle no product quantities being ordered.

a) In the `calculate` function, starting on line 119, type the following to define `self.qty` and set it to a value of zero.

```
116     def calculate(self):
117         """Event handler for the calculator"""
118         self.total = 0
119         self.qty = 0
120         for i in range(len(self.products)):
```

b) In the `calculate` function, starting on line 131, type the following to track the quantity of products ordered.

```
121             if (self.entry[i].get()):
122                 try:
123                     int(self.entry[i].get())
124                 except:
125                     self.lblsubmit.set("'" + self.entry[i].get() + "'" +
126                                        " is not valid entry. "
127                                        "Must be a whole number.")
128                     return
129                 else:
130                     self.total += (Decimal(self.prices[i]) * int(self.entry[i].get()))
131                     self.qty += int(self.entry[i].get())
132
```

c) In the `submit` function, starting on line 150, type the following to add an `elif` clause to the `if` statement that you added in the previous step.

```
148         if total <= 0:
149             raise Exception("Select Calculate Price button before submitting order.")
150         elif self.qty <= 0:
151             raise Exception("Must specify products to order.")
152         else:
153             send={'ship': ship, 'total': total}
```

With these statements:

- The `elif` clause checks the total quantity of products ordered to see if it is equal to or less than zero.
- If the `total` variable equals zero, an exception is raised stating that the user needs to order some products before submitting the order.

4. Run the **Wood Delivery Calculator** web service and client.

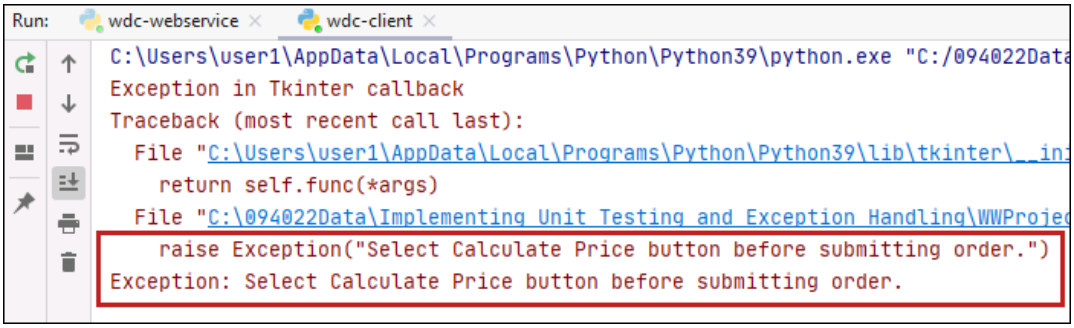
a) Run the **wdc-webservice.py** script.

b) Run the **wdc-client.py** script.
The Wood Delivery Calculator window should appear.

5. Test your changes.

a) Select **Submit Order**.

b) In the **Run** console pane, in the **wdc-client** tab, observe the `Exception` information.



The code did what you specified, but you would prefer to see a message in the app window instead of the error message.

c) Select **Calculate Price**.

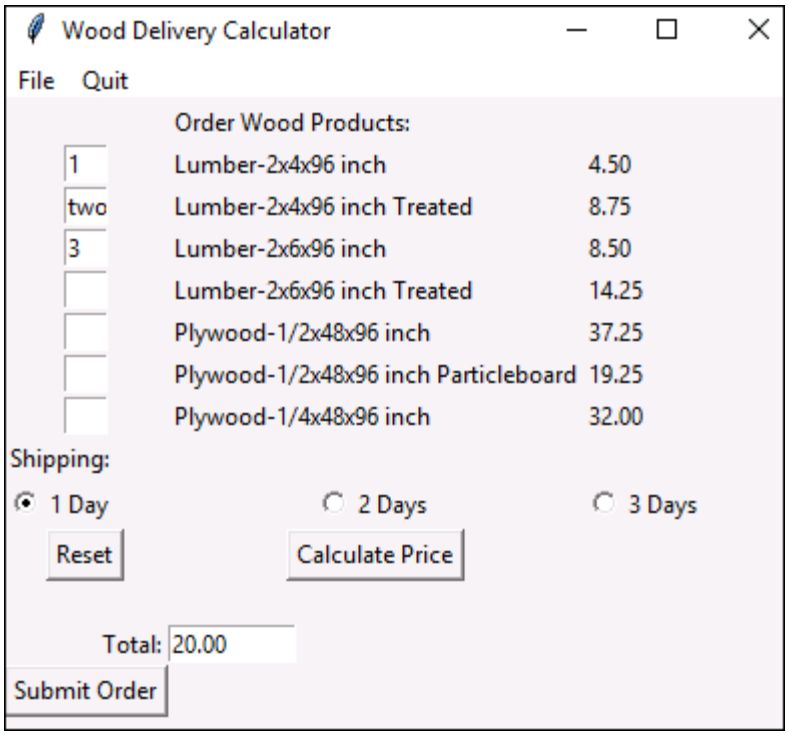
d) Select **Submit Order**.

e) In the **Run** console pane, in the **wdc-client** tab, observe the `AttributeError` information.

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:\Users\user1\AppData\Local\Programs\Python\Python39\lib\tkinter\__ini
    return self.func(*args)
  File "C:\094022Data\Implementing Unit Testing and Exception Handling\WWProjec
    raise Exception("Must specify products to order.")
Exception: Must specify products to order.
```

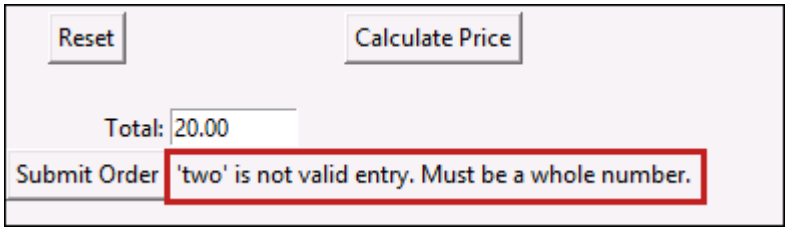
The code did what you specified, but you would prefer to see a message in the app window instead of the error message.

f) Specify the quantity of products to match the image below.



g) Select **Calculate Price**.

h) In the Wood Delivery Calculator window, observe the message to the user is displayed in the **Submit Order** label.



i) Close the Wood Delivery Calculator window.

	**Note:** Leave the **wdc-webservice.py** script running and do not close the **Run** pane.

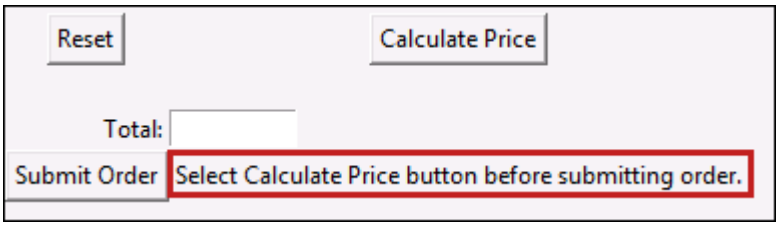
6. Change error exceptions to display the user message in the Wood Delivery Calculator window.
   a) In the **wdc-client.py** tab, in the `submit` function, on line 149, change the "Select Calculate Price button" raise exception statement to display the message in the `lblsubmit` label.

```
148         if total <= 0:
149             self.lblsubmit.set("Select Calculate Price button before submitting order.")
150         elif self.qty <= 0:
151             raise Exception("Must specify products to order.")
```

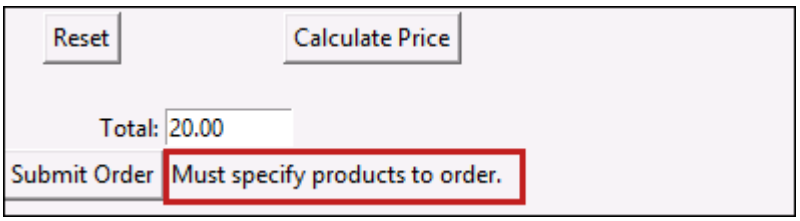
   b) In the `submit` function, on line 151, change the "Must specify products" raise exception statement to display the message in the `lblsubmit` label.

```
148         if total <= 0:
149             self.lblsubmit.set("Select Calculate Price button before submitting order.")
150         elif self.qty <= 0:
151             self.lblsubmit.set("Must specify products to order.")
152         else:
```

   c) Run the **wdc-client.py** script.
   The Wood Delivery Calculator window should appear.
   d) Select **Submit Order**.
   e) In the Wood Delivery Calculator window, observe the message to the user is displayed in the **Submit Order** label.



   f) Select **Calculate Price**.
   g) Select **Submit Order**.
   h) In the Wood Delivery Calculator window, observe the message to the user is displayed in the **Submit Order** label.



7. Clean up.
   a) Close the Wood Delivery Calculator window.

b) Select the **wdc-webservice** tab, and then select the **Stop 'wdc-webservice'** button.

c) In the **Run** console pane, close all tabs.
d) Close the **wdc-client.py** and **wdc-webservice.py** tabs.

TOPIC B

Write a Unit Test

To verify that your application can do what it's supposed to without error, you must test it. In this topic, you will write unit tests.

Unit Testing

Unit testing, also called component testing, tests the functionality of the smallest piece of code that can be logically isolated in a system. In most programming languages, this is a function, a subroutine, a method, or a property. It is the step in the process of software verification testing. The goal of unit testing is to verify that code does what it's designed to do. Unit testing is usually handled by the developer who wrote the code as part of the initial quality assurance testing, prior to code being checked in as complete and ready for integration with the larger code base. As such, bugs found in code during unit testing are typically fixed on the spot and not tracked as part of a larger quality assurance, bug tracking and fixing operation that most software goes through.

Software usually goes through several stages of testing:

- **Unit testing**: Described above, is the first step in software verification testing.
- **Integration testing**: Separate components are tested as a group to see if they work together as designed. At this stage, developers are looking for issues with components interfacing with each other and passing data to each other. This is the second step in software verification testing.
- **System testing**: Developers test the full application, with all integrated components, to verify the application is performing to fulfill all major requirements defined for the software, and that its performance is optimized. This is the last step in software verification testing.
- **Acceptance testing**: Sometimes called functional testing, this is the final stage of the quality assurance (QA) test cycle. It evaluates if the software meets requirements and that it is ready for release to users and customers. This testing may be carried out with a select group of testers using a non-complete, but functional, beta version of the software. By releasing a beta version, development teams dramatically increase usage of the software and that will help find any bugs still present in the software.

In later stages of software verification and acceptance testing, software test teams may use bug tracking software and code to write tests, log bugs, fixes, and updated releases of software. As stated earlier, unit testing is typically done at the time code is completed, by the developer who wrote it, using a unit test framework to assist in unit testing.

Testing Terminology

The following terms are used with regard to unit testing:

- **Test case**: A single test of a discrete piece of code.
- **Test suite**: A group of test cases that belong together to test some functionality.
- **Test fixture**: The environment (starting state setup and end state teardown) of the state of the system/app before and after test execution.
- **Integration test**: Any test that includes components outside the originating component.
- **Interaction test**: Any test on how objects work together.
- **State test**: A test of the results produced by an operation of code.
- **Fake**: A stand-in object that is used in place of a real object. For example, if your app must talk to a database to get data, the database access object may not be written, so you may use a fake object such as a spreadsheet access object.
- **Stub**: A fake object that provides a specific dependency required by the test.

- **Mock**: A stand-in used to check the results of a test.

Importance of Testing

Before continuing, it's worthwhile to discuss the importance of testing in general, and unit testing in specific. Unit tests help developers find and fix problems immediately using the resource: the developer that wrote the code who is most familiar with it, without using extra time and resources to log the problem, then have a new developer learn the code to resolve it. Strong unit testing as part of software verification has the following benefits:

- **Software development becomes agile.** It is easier to add functionality code in a modular way with strong unit testing practices in place.
- **Code quality is improved.** This is a simple side effect of checking one's work.
- **Reduces the impact of bugs.** Software bugs found during unit testing are contained to the discrete piece of code impacted, and are addressed before that code is integrated with the larger code base where the problem may be more difficult to isolate.
- **Makes updating code easier.** It is much easier to update code with confidence when code is put through unit testing because each unit of code has been verified before it's integrated with the larger code base.
- **Provides documentation.** Unit test data is typically stored with each module, essentially expanding the documentation for individual software components. This makes it easier for other developers to learn how those components work.
- **Simplifies later software verification.** Thorough unit testing finds and fixes bugs early, reducing the need to address them in later stages of software verification testing.
- **Improves software design.** Developers who are familiar with, and use, unit testing tend to learn to think about designing their code to be maintained and to integrate well with other code.
- **Reduces costs.** Any time bugs are addressed quickly, and the quality of code is improved, the costs for the entire development project come down.

Unit Testing Frameworks

A unit test framework is a set of software tools to help developers write and run unit tests. They generally include facilities for writing and executing tests and reporting the results of those tests, which may be as simple as pass/fail or may be more complex. Unit test frameworks all allow developers to run tests manually, and most provide facilities to automatically run tests.

You have many unit test frameworks to choose from. Python includes the `unittest` unit test framework inspired by the JUnit unit test framework. It supports:

- Test automation.
- Sharing setup and teardown code for tests.
- Aggregation of tests into collections for building test suites.
- A reporting framework that is independent of tests.

`Unittest` is the default test running for Python.

Other Unit Testing Frameworks

Other unit testing frameworks include:

- Robot Framework for automation: **https://robotframework.org**
- Doctest for interactive testing: **https://docs.python.org/3/library/doctest.html**
- Nose2: **https://docs.nose2.io/en/latest/**

PyCharm Pytest

The PyCharm IDE includes the `pytest` unit testing framework which has the following features:

- A dedicated test runner.
- Code completion of test subject and `pytest` fixtures.
- Code navigation support.
- Detailed reports.
- Multiprocessing test execution.

Other unit test frameworks are available and offer different features and capabilities. You should evaluate unit test frameworks and select one that is best for your project.

Automated vs. Manual Testing

Manual tests are just that, tests written and executed manually. With automated testing, testers or developers write test scripts for the automation framework to execute. Automated testing has many advantages:

- Faster than manual testing.
- Tests are more reliable and consistent as they reduce human error.
- Improves accuracy of testing.
- Can be managed and run by fewer staff.
- Run test suites faster.
- Provides for more comprehensive testing.
- Saves time and cost in the testing process.
- Improves time to market.

In most development environments, testing is automated as much as possible, with manual testing left for ad hoc test cases.

Example Unit Test

The `TestCase` class for the `factorial()` function will consist of separate assertion tests for each rule that must be matched. The first method must define a test to check for a valid factorial value:

```
def test_factorial_valid(self):
   self.assertEqual(factorial(5), 120)
```

The second test method in the `TestCase` must test that the `factorial()` function returns a value of 1 for the factorial of 0:

```
def test_factorial_zero(self):
   self.assertEqual(factorial(0), 1)
```

The third method tests that the `factorial()` function returns a `False` value if you try to find the factorial of a negative number:

```
def test_factorial_negative(self):
   self.assertFalse(factorial(-1))
```

Finally, the fourth method tests that the `factorial()` function returns a `False` value if you submit a text value as the parameter:

```
def test_factorial_text(self):
   self.assertFalse(factorial("text"))
```

With these four tests, you're ready to code the `TestCase` for the `factorial()` function.

Test-Driven Development

With short development cycles and multi-member programming teams, trying to coordinate perfect code can be difficult. One small change made to a minor function can have hidden side effects in other places in the application, often written by a different team of programmers.

Test-driven development (TDD) is a programming paradigm that attempts to minimize errors and bugs in applications. TDD provides a framework that all developers can use to write better, more precise code in their applications.

With TDD, before you write any code, you write a series of specifications (or rules) for each function used in the application. The specifications define how each function works, what data it requires, how it handles the data, and what data it produces. Consider all aspects of quality as you write your unit tests. For example, you might include security tests to test for potential vulnerabilities you have identified. After you write the specifications, identify a series of tests that can verify whether the code matches each specification.

Once you have designed the tests, write each function to pass the tests. A function isn't released until it can pass all of the tests, ensuring that it meets all of the design specifications. By placing the code testing at the beginning of the development cycle instead of the end, you ensure all code matches the design specifications.

Benefits of Test-Driven Development

Creating a series of tests for each function within an application up front may seem somewhat unorthodox, but there are benefits to that process. First, by forcing developers to decide on the specification rules first, you're forcing them to think outside of the box. Trying to determine just how a function could break goes a long way in identifying problems before customers do. This ensures that each function in an application works as expected, no matter what strange thing an unwitting customer may throw at it.

Also, by writing the specification rules up front, you ensure that all code changes made during the development cycle don't break the function, or any other functions that rely on the function being changed.

A side benefit to TDD is that it forces developers to take a modular approach to writing applications. Instead of cramming all of the features of an application into one monolithic block of code, TDD forces developers to split functions into separate modules. Each module provides a separate function of the application and is defined by a separate set of specification rules. Modular code helps the development process by not only making the application easier to manage, but also can aid in code sharing between applications. Modules that are common between applications can be shared.

Finally, in large programming shops, TDD helps ensure all of the developers are working off of the same design goals. Specification rules are written to ensure proper interoperability between functions in the application. As long as developers write to satisfy the specification rules, the chances that changes to one function have a negative impact on another function are minimized. Separate development teams can work on their set of functions knowing just what to expect from other functions contained in the application.

Development teams should evaluate TDD to see if it's right for their project.

Guidelines for Writing a Unit Test

Follow these guidelines when writing a unit test.

Write a Unit Test

When writing a unit test:

- Determine the rules that define the function that's being tested.
- Create a `TestCase` class that contains the separate tests.
- Define separate methods for each rule.
- Determine whether a rule must match a returned value, or a `True` or `False` `Boolean` value.
- Make sure each unit test is fully independent, can run alone, or in a suite of tests regardless of when it is called in the suite.

- Try to keep unit test execution fast. Remember, if a test uses a complex data structure, that data structure must be loaded each time the test executes.
- Automate tests where you can to save time.
- Test all code before storing it in your code repository.
- Use long and descriptive names for testing functions so other testers and developers can understand the nature of the tests.

ACTIVITY 7-3
Writing a Unit Test

Before You Begin
The **WWProject-L7** project is open in PyCharm.

Scenario
You want to test a new script you've created for the **Wood Delivery Calculator**. The PyCharm editor can automatically generate a `unittest` stub for you to build on. The **getWDCData.py** file contains two methods that retrieve the products and prices inside the web service. You will create a test case file to check these methods.

---

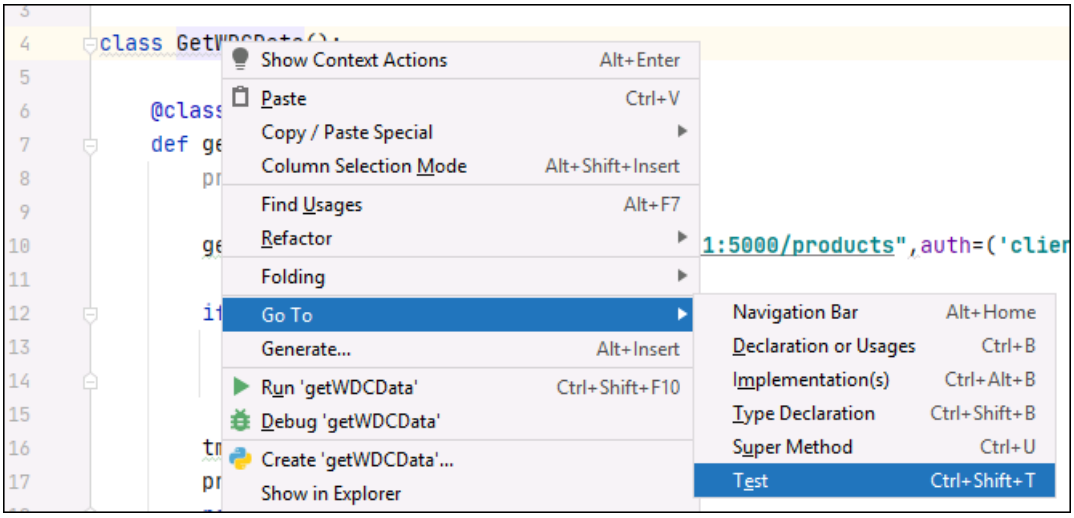
1. Examine the **getWDCData.py** script.
   a) In the **Project** pane, open the **getWDCData.py** file.
   b) Examine the methods in the `GetWDCData` class.

```python
import ...

class GetWDCData():

    @classmethod
    def get_products(self):
        products = list()

        getprod = requests.get("http://127.0.0.1:5000/products",auth=('client', 'P@ssw0rd12!'))

        if getprod.status_code == 401:
            print(str(getprod.status_code)+" "+ getprod.text)
            exit()

        tmpproducts = json.loads(getprod.text)
        products = tmpproducts["products"]
        return products

    @classmethod
    def get_prices(self):
        prices = list()

        getprice = requests.get("http://127.0.0.1:5000/prices")
        tmpprices = json.loads(getprice.text)
        prices = tmpprices["prices"]
        return prices
```

   The `get_products()` and `get_prices()` methods make requests to the web service to read data from the database file and return products and prices data.

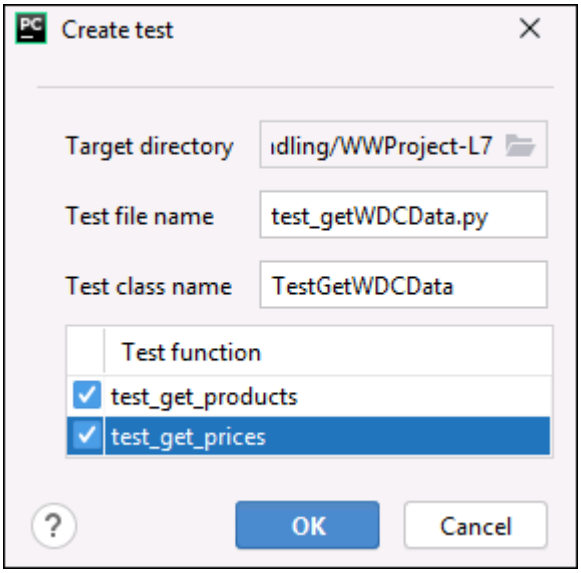
2. Use PyCharm to generate the test suite for both of the methods contained in the `GetWDCData` class.

a) In the `class` line, select and then right-click the `GetWDCData` keyword and select **Go To→Test**.



b) In the **Choose Test for GetWDCData** dialog box, press **Enter** to display the **Create test** dialog box.

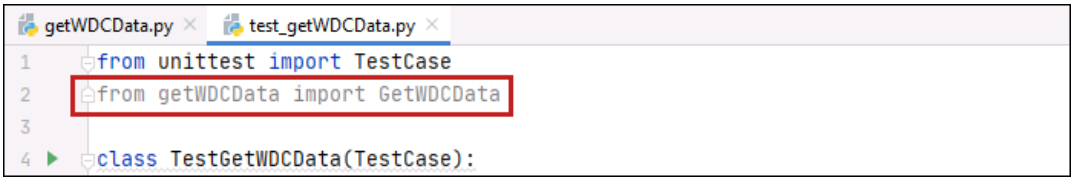
c) In the **Create test** dialog box, select the check boxes to generate the `test_get_products` and `test_get_prices` methods, and select **OK**.



The **test_getWDCData.py** file is created.

3. Modify the default test suite to add tests for the `get_products()` and `get_prices()` methods.

a) In the **test_getWDCData.py** tab, on line 2, type the following to import the `GetWDCData` class from the `getWDCData` library file.



b) For the `test_get_products()`, on line 6, type the following to revise the `self.fail()` statement to a `self.assertEqual` statement.

```
4  class TestGetWDCData(TestCase):
5      def test_get_products(self):
6          self.assertEqual(len(GetWDCData.get_products()), 7)
7
```

This checks if the length of the products list returned matches the number of products you have in the products table. In this case, there should be seven products.

c) For the `test_get_prices()`, on line 9, type the following to revise the `self.fail()` statement to a `self.assertEqual` statement.

```
8      def test_get_prices(self):
9          self.assertEqual(len(GetWDCData.get_prices()),4)
10
```

There should be four prices.

TOPIC C

Execute a Unit Test

Once you've written unit tests, you must execute them to test your code. In this topic, you will execute unit tests.

Test Runners

A test runner is a programmatic component that manages test execution and reporting output to the user. Depending on the unit test frameworks you use, the test runner may be built into an IDE, it may have a graphical interface, or a text-based interface. Different test runners also have different ways they report outcomes, from text messages to test codes.

The default test runner in Python is the `unittest` framework. The PyCharm IDE includes the `pytest` unit testing framework. If you use PyCharm and wish to use `pytest` as the test runner, you must designate `pytest` as the default test running by opening **File→Settings→Tools→Python Integrated Tools** and selecting **pytest** in the **Default test runner** dialog box.

Test Structure

It's recommended to keep your testing files separate from your core app files. This allows you to run tests and make changes to various versions, while keeping your original code pristine, and closely controlling all changes made to the original code. Pytest supports the following two test folder structure layouts:

Using a separate folder structure:

src/

app.py

component1/

Method1.py

Method2.py

tests/

test_app.py

component1

test_method1.py

test method2.py

Using an embedded tests/folder inline with each component in your app:

src/

app.py

tests/

test_app.py

component1/

method1.py

method2.py

tests/

test_method1.py

test_method2.py

Additional Information

Python Unit Testing - Structuring Your Project: **https://www.patricksoftwareblog.com/python-unit-testing-structuring-your-project/**

Typical directory structure for running tests using `unittest`: **https://gist.github.com/tasdikrahman/2bdb3fb31136a3768fac**

Test Execution in PyCharm

You can run the test or debug configurations you've created in PyCharm from the main menu, and from context menus for a test class, directory, or package, and for a test class or method you are currently working on in the editor. If you haven't created a permanent run/debug configuration for the test you're executing, a temporary configuration is created which you can save and reuse later.

PyCharm runs tests in the background, and several tests can be executed at the same time depending on your system resources. Each running test gets its own test results tab. You can run a single test or run all tests in a directory. If you do, tests are run one by one. If you wish to run tests in parallel, you must enable test multiprocessing in pytest.

Unit Test Suites

Single unit tests are great for testing individual functions, but the typical Python application consists of several functions. It can be somewhat tedious creating separate unit tests for all of the functions contained in the application. The `unittest` library also provides a test suite feature that allows you to group multiple unit tests into a single test group. That allows you to run the test group as a single test and view the results.

The PyCharm editor provides an easy way to create test suites from classes instead of individual functions. Just right-click on the class name in the class definition and generate the test. The PyCharm wizard automatically detects the methods defined in the class, and offers you the option to generate tests for each method.

Guidelines for Executing a Unit Test

Follow these guidelines when developing a test suite for an entire application.

Test as You Develop

Develop your testing code as you develop the application functionality that it will test:

- Group all of the methods into a single class object.
- Right-click the class name in the class definition.
- From the menu, select the **Go To→Test** entries.
- Create a new test file.
- Select the methods that you need to test from the class methods.
- Customize the test methods to meet the rules defined for your requirements.

Follow Testing Best Practices

To follow best practices for testing in Python:

- Direct each testing unit toward testing one piece of functionality to prove it works correctly.
- Design each testing unit so it can run alone, as well as within the test suite, in any sequence. This may mean initializing each test with a fresh dataset, and performing cleanup afterwards. (You can write `setUp()` and `tearDown()` methods to accomplish this.)
- Strive to write tests that run quickly, so they won't slow down development.

- When you find a bug, write a test that exposes the bug, and leave it in the suite after you resolve the bug. Tests accumulated over time will help you and other developers identify potential problem areas and will help to ensure that past problems don't reappear.
- Give your testing functions long, descriptive names. These are displayed when a test fails, so descriptive names will be more useful. They will not be called explicitly, so you don't have to be concerned about them cluttering your code.
- Run individual testing units frequently as you develop corresponding units of functionality.
- Run the full suite before and after each programming session.
- Run the full suite before pushing code to a shared repository.

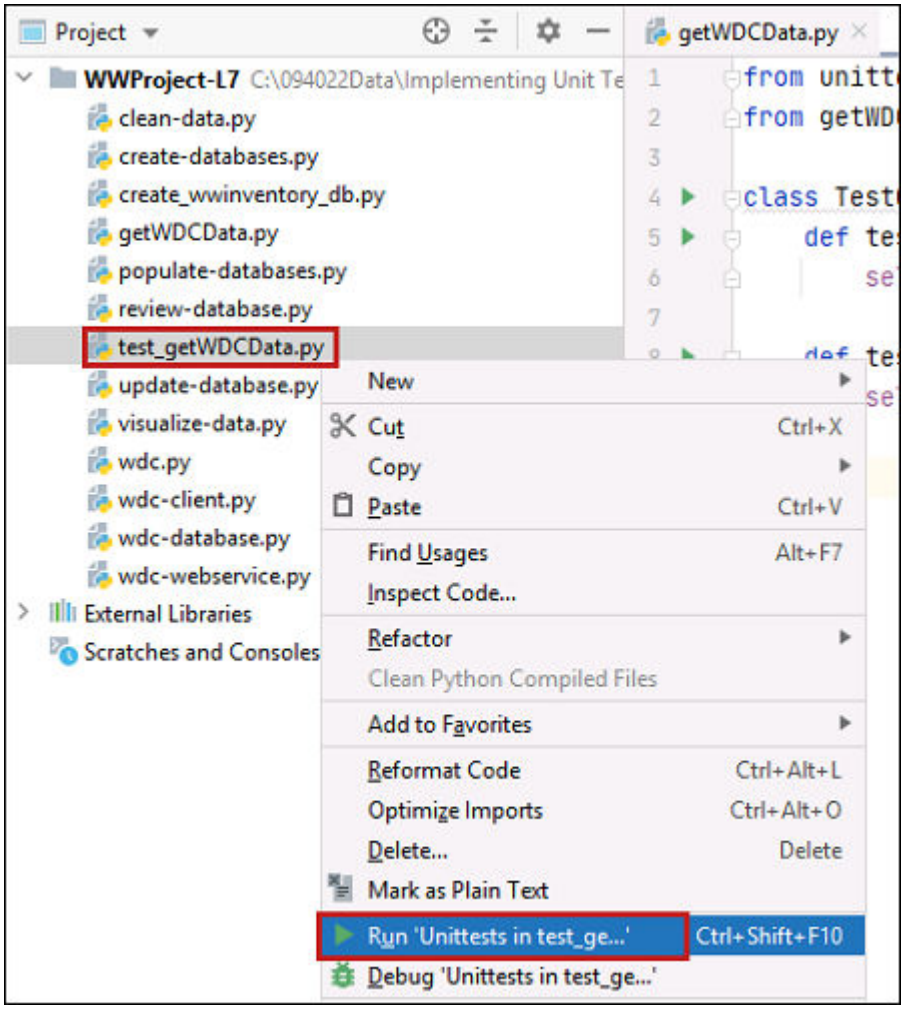
ACTIVITY 7-4
Executing a Unit Test

Before You Begin
The **WWProject-L7** project is open in PyCharm.

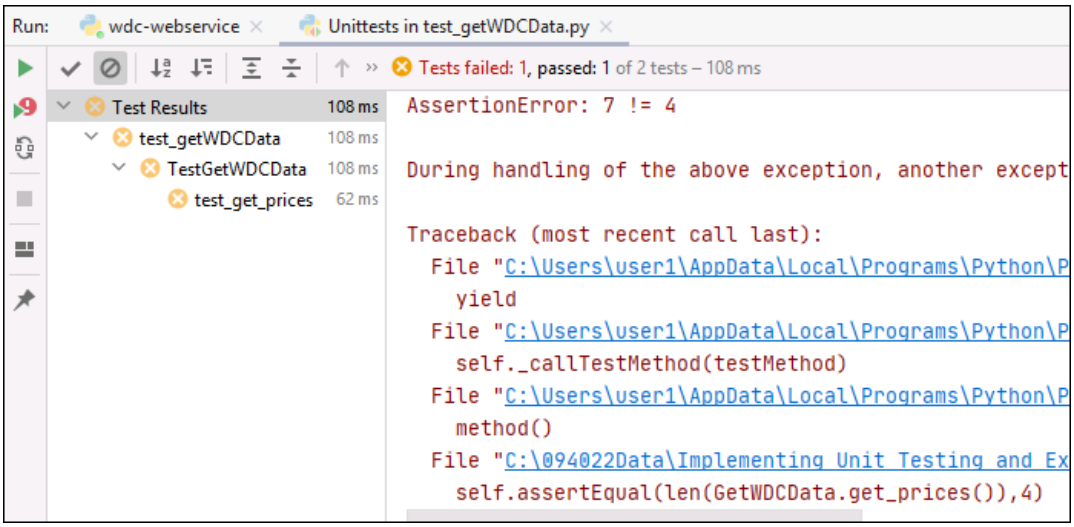
Scenario
Now that you have the test code prepared, you can run the tests for the `GetWDCData.py` class methods and observe the results. If any test fails, you will have to modify the `test_getWDCData.py` function code.

---

1. Perform the unit tests in the **test_getWDCData.py** file.
   a) Run **wdc-webservice.py**.
   b) In the **Project** pane, right-click **test_getWDCData.py**, and select **Run 'Unittests in test_getWDCData.py'**.



c) Observe the unit test output in the console window. Both tests should have completed successfully.



The `unittest` feature in PyCharm runs the unit tests in a special output window in the console area.

d) In the left pane, examine the status of each of the unit tests you defined.

The unit tests show that one test failed. The `test_get_prices` test determined if the `get_prices()` function returns the expected number of product prices. The expected number of prices was four, but seven were returned. Looking back, you discover that seven is correct and you need to update your test code to accommodate this.

e) Close the Unittests in **test_getWDCData.py** tab, in the **Run** console pane.

2. Modify the `get_prices()` function to add a check for text input values.

a) In **test_getWDCData.py**, in the `get_prices()` function, on line 9, review the code.

```
8   def test_get_prices(self):
9       self.assertEqual(len(GetWDCData.get_prices()),4)
10
```

The "4" signifies the number of records expected in the test.

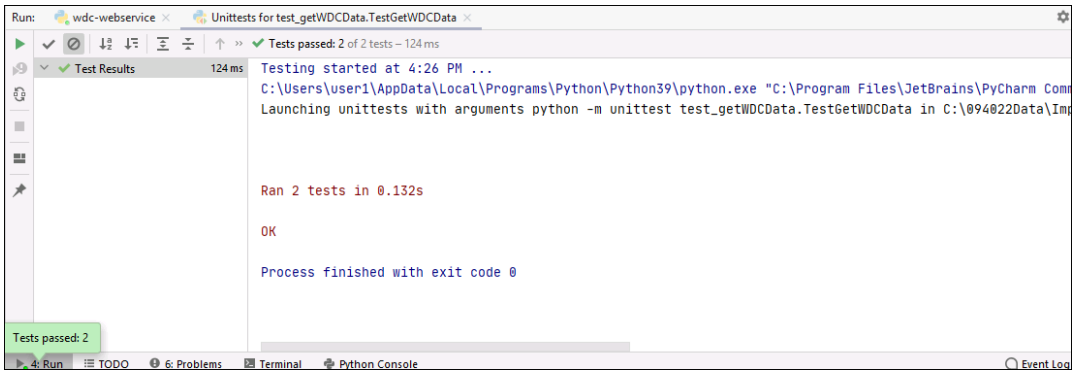
b) On line 9, modify the code to test for **7** price records.

```
8   def test_get_prices(self):
9       self.assertEqual(len(GetWDCData.get_prices()),7)
10
```

3. Re-run the **test_getWDCData.py** test.

a) Run **Unittests for test_getWDCData.py**.

b) Observe the unit test output in the console window. Both tests should have completed successfully.



4. Clean up the workspace.

a) Select the **wdc-webservice** tab, and then select the **Stop 'wdc-webservice'** button.

b) In the **Run** console pane, close all tabs.
c) Close the **test_getWDCData.py** tab.
d) Select **File→Close Project**.

Summary

In this lesson, you wrote code to handle exceptions to allow your app to continue to execute even if something unexpected comes up. You then wrote and executed unit tests to test your code and verify that your app is meeting the necessary requirements.

What approach will your team take to testing application software they develop?

What test framework will your team use for testing software?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

8 Packaging an Application for Distribution

Lesson Time: 1 hour

Lesson Introduction

Once you've created an application, you may want to think about sharing it with the world. The default Python® installation includes utilities to help you bundle all of the files required for your application into one place for easy distribution.

Lesson Objectives

In this lesson, you will:

- Create and install a package.
- Generate alternative distribution files from a Python project.

TOPIC A

Create and Install a Package

Python packages rely on a standard structure, so installation programs can easily install them in any Python installation environment.

Packages

Once you have completed a Python application, you need to distribute it. Even if you don't want to share your Python application with others, you may still need to install your application on multiple computers. Either way, you'll want to bundle all of the files in your application into a Python package.

A Python package is a standard way of bundling all of the program, data, and test files required for an application into a single manageable folder. That folder can then be bundled for distribution using any type of distribution means, such as ZIP files, Linux® package management tools, or the Windows® installer tool.

Before you package your Python application, you must select a name for the package. Try to make the Python package name unique, and consider adding a version identifier to it because you can't install two packages with the same name in your Python environment.

Additional Information

For more on creating Python packages, see: **https://packaging.python.org/overview/**.

Package Structure

When creating a Python distribution package, you must follow a strict structure format. The package itself is a ***package folder*** that contains your application files, including your Python code files, any data files required for your application, as well as any unit test files that you may have built.

The key to the package folder is the **__init__.py** file. Its presence in a folder indicates that the folder is a Python library and contains code files that Python should scan when searching for library modules. The **__init__.py** file itself doesn't need to contain any data; it just needs to exist.

The main configuration file for a Python package is the **setup.py** file. It tells the Python installation program important information about the package. This is where you configure your application package name, any version number information, who wrote the application, how to contact the author, and what files need to be installed along with the program files.

Dependencies

Most Python applications require some external libraries and/or data files to operate. For example, the **Wood Delivery Calculator** web service application requires the `requests`, `json`, `tkinter`, `mysql.connector`, `flask`, `flask_restful`, `datetime`, `decimal`, and `flask_httpauth` libraries to run. These particular libraries are included in the standard Python distribution, but you need to let the Python installer know if the package includes other libraries required for your application to run properly.

You also need to let the Python installer know about any data files that the application requires.

PyPI

A commonly used site for package distribution is ***Python Package Index (PyPI)*** which is a central clearinghouse for shared Python applications and code. When you distribute a package through PyPI, other users can download and install your package to their own computer.

For example, in PyCharm, you can directly install a package from PyPI using the **Settings** dialog box's **Project Interpreter**. When you select the **Install** button (shown as a **+** on the right side of the **Project Interpreter** dialog box), a searchable list of available packages is shown. You simply select a package and select the **Install Package** button to download and add the package to your Python installation. Other Python programming tools and operating systems provide other convenient ways to install packages directly from PyPI.

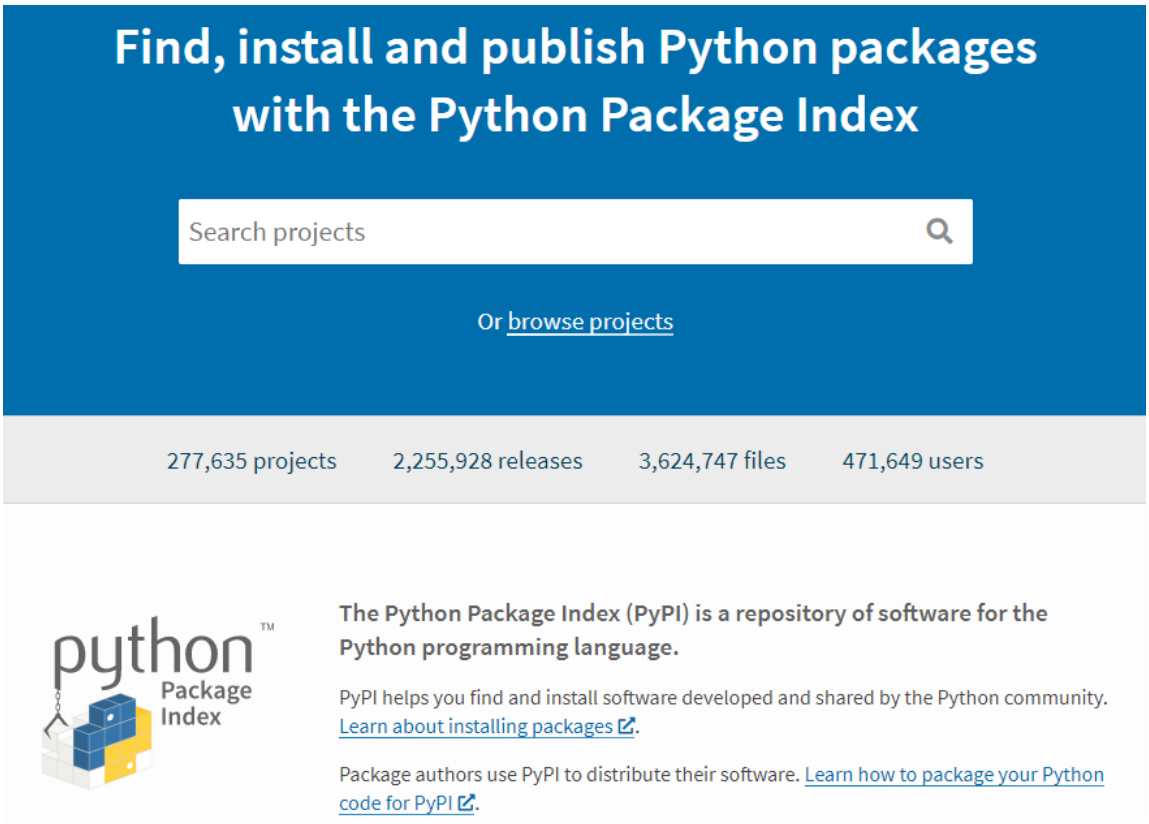


Figure 8-1: Selecting a package to download from PyPI.

	**Note:** Even if you don't plan on distributing your application world-wide, it's always a good idea to check out the package names in PyPI so that your package won't conflict with any other popular Python package that may be installed on your system.

You can download and install packages directly within PyCharm by using the Project Interpreter. But there may be times when you prefer to use the command line. To install packages, you can use the `pip install` command.

You can use `pip install` to install packages from PyPI or another repository, as well as from packages in a local package archive (a TAR or ZIP file) that you have already downloaded and have in a local file.

	**Note:** When you work with Python from the command line, to ensure that Python commands such as `pip install` are found, make sure that you have included the important directories within your PATH environment variable, such as **C:\Python39, C:\Python39\DLLs**, **C:\Python39\Lib\lib-tk**, and **C:\Python39\Scripts**. (These example paths assume Python version 3.9. If you are using a different version of Python, adjust these paths accordingly.)

PyPI Website

See the PyPI website at: **http://pypi.org**.

Other Ways to Distribute Packages

You may want to distribute your packages through means other than PyPI. You can set up your own repository as an alternative to PyPI, or you can deliver files in a format that is ready to install directly. The Python `distutil` utility provides for packaging and installing applications using several different formats. You can choose to distribute your application as source code for everyone to see, or you can choose to distribute your application as compiled Python code to hide your code. You can also choose to distribute your application using a Windows® installer format, a Linux® package management format, or even a simple TAR or ZIP file. PyCharm's **Tools→Run setup.py Task** menu provides a way to run one of these packaging options without having to enter command-line commands directly.

ACTIVITY 8-1
Creating and Installing a Package

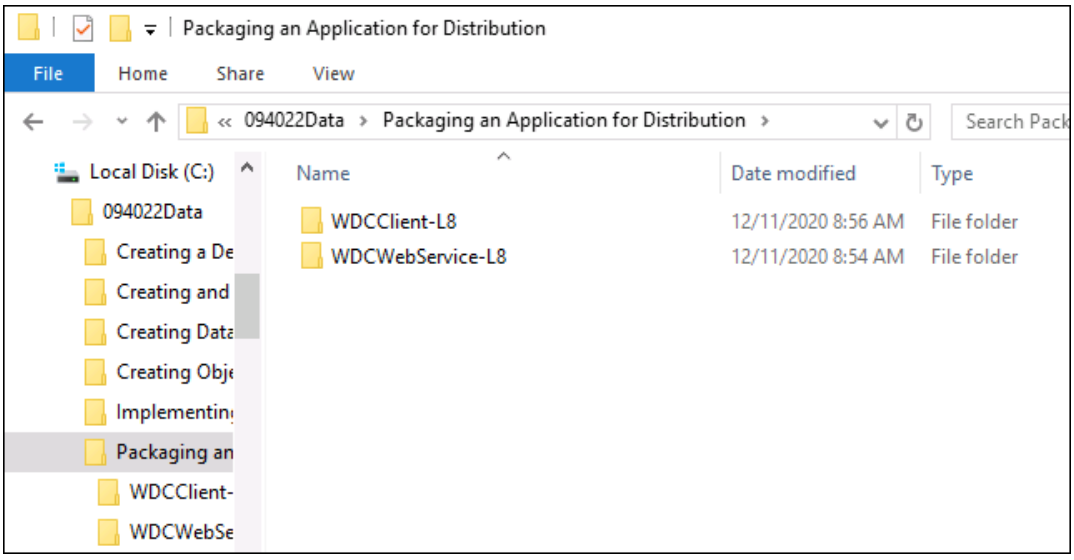
Data Files
All project files in C:\094022Data\Packaging an Application for Distribution\WDCWebService-L8.

Before You Begin
PyCharm is open.

Scenario
You are ready to package the **Wood Delivery Calculator** web service so it can be installed on web servers. Before you can begin, you'll need to use PyCharm to create the basic structure for your application. You have already created two new projects and divided the scripts for web service and client between them. You are only including files you feel are pertinent to each.

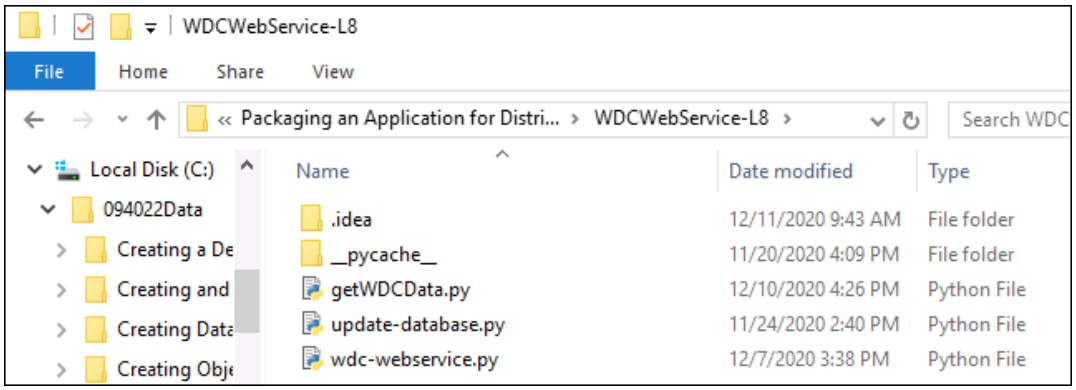
---

1. View project files.
   a) Open File Explorer and navigate to **C:\094022Data\Packaging an Application for Distribution**.
   b) Observe the two project folders. One for the web service and one for the client.



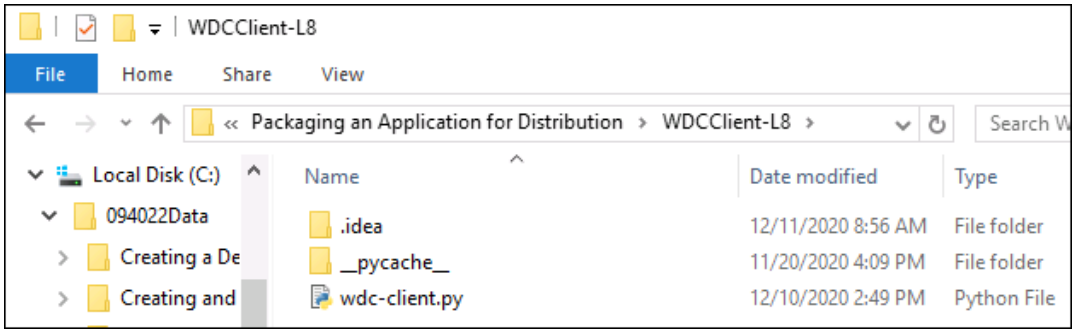
   c) Open the **WDCWebService-L8** folder.

d) Observe that the web service project only contains three Python scripts.



e) Open the **WDCClient-L8** folder.

f) Observe that the client project only contains one Python script.



g) Close File Explorer.

2. Open the WDCWebService-L8 project.

a) In the Welcome to PyCharm window, select **Open**.

b) Navigate to the **C:\094022Data\Packaging an Application for Distribution\WDCWebService-L8** folder.

c) With the **WDCWebService-L8** folder selected, select **OK**.

d) In the **Project** pane on the left side of the PyCharm window, verify that **WDCWebService-L8** is listed.

3. Create a Python package folder in the project.

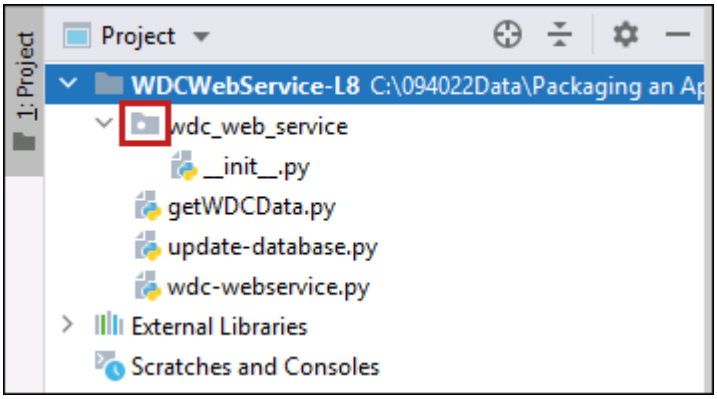
a) In the **Project** pane, right-click the **WDCWebService-L8** project, and select **New→Python Package**.

b) In the **New Package** text box, type ***wdc_web_service*** and press **Enter**.

The new **wdc_web_service** package folder is created in the project folder. It contains an empty initialization file, **__init.py__**.

c) Observe the package folder.

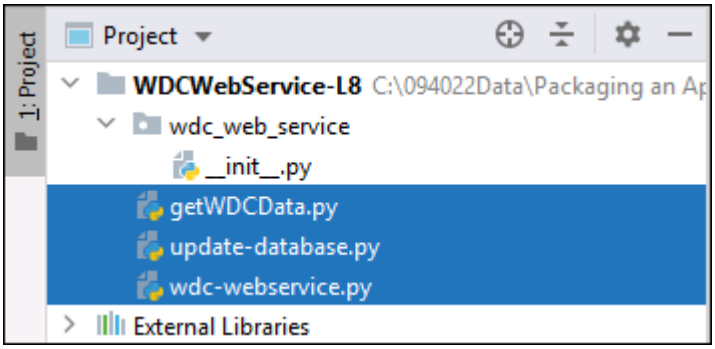
It has a special folder icon that shows that it is a *packaging* folder.



4. Move essential files into the package folder.

a) In the **Project** pane, select **getWDCData.py**.

b) Press and hold **Shift**, then select **wdc-webservice.py** to include all three files as shown.

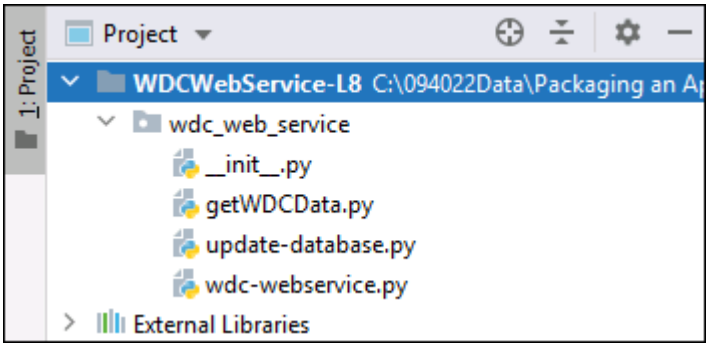


c) Drag the selected files into the **wdc_web_service** package folder.

The **wdc_web_service** package folder will highlight with a red box when you can drop the files.

d) In the **Move** dialog box, select **Refactor**.

The files are now indented beneath the **wdc_web_service** package folder, so they will all be included in the package.



All three files also open in the editor.

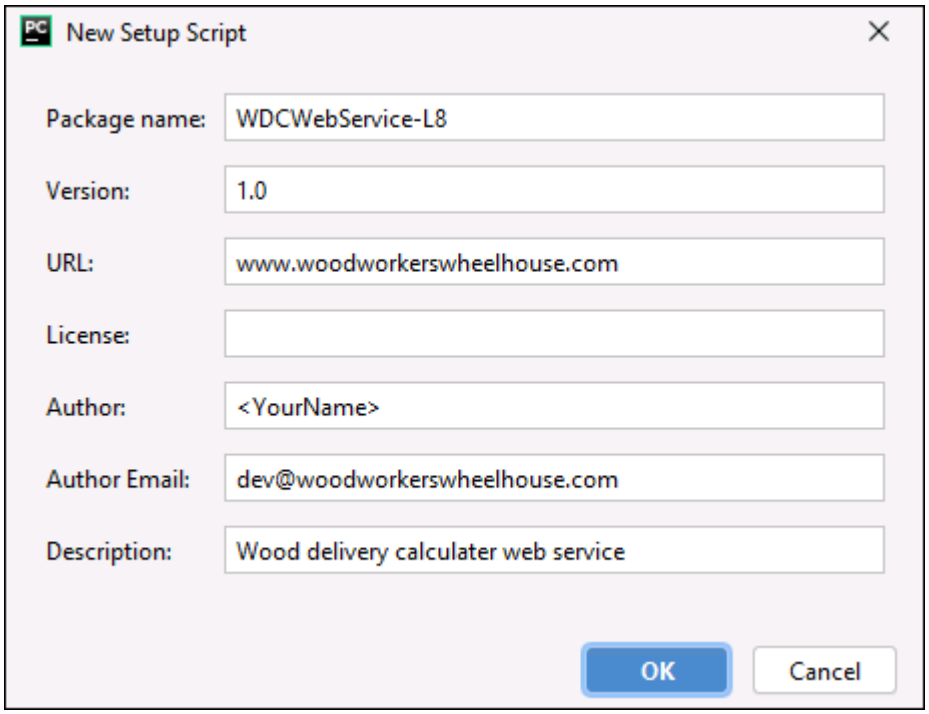
5. Create a **setup.py** file for your project.

a) In the **Project** pane, verify that the **WDCWebService-L8** project is selected.

Note: Be careful to select the **WDCWebService-L8** project folder, not the **wdc_web_service** package folder.

b) Select **Tools→Create setup.py**.

c) Fill in the **New Setup Script** form with the package information.



d) Select **OK** to build the **setup.py** file.

Setup.py is created and shown in the editor.

6. Make note of the various package dependencies.

a) In the editor, select each of the **.py** tabs, and note the various packages that are imported.

- **getWDCData.py** imports _____.
- **update-database.py** imports _____.
- **wdc-webservice.py** imports _____.

The following packages are imported: `requests`, `json`, `tkinter`, `mysql.connector`, `flask`, `flask_restful`, `datetime`, `decimal`, and `flask_httpauth`.

7. Examine the list of package dependencies in **setup.py**.

a) Select the **setup.py** tab.

b) Examine **setup.py**.

The setup is initialized using the values you provided. If you need to add package dependencies or special packages, you could modify the packages value to include more package names after **'wdc_web_service'**.

c) Right-click the **setup.py** tab and select **Close All**.

TOPIC B

Generate Alternative Distribution Files

If you've created an application that you want to distribute to users who do not have Python installed on their computer, you can provide the application along with its own instance of Python, essentially delivering a complete Windows executable version of the application.

Packaging Options

When you create a package using either the **Run setup.py Task** menu from PyCharm or `setuptools` from the command line, you'll need to specify the type of packaging operation you want to perform. Typical options include the following.

Item	*Description*
bdist	Create a built (binary) distribution.
bdist_dumb	Create a "dumb" built distribution.
bdist_msi	Create an executable installer for Microsoft® Windows®.
bdist_rpm	Create an RPM distribution.
bdist_wininst	Create an executable installer for Microsoft Windows.
build	Build everything needed to install.
build_py	Build pure Python modules (copy to build directory).
build_ext	Build C/C++ extensions (compile/link to build directory).
build_clib	Build C/C++ libraries used by Python extensions.
build_scripts	"Build" scripts (copy and fixup #! line).
clean	Clean up temporary files from the `build` command.
install	Install everything from build directory.
install_lib	Install all Python modules (extensions and pure Python).
install_headers	Install C/C++ header files.
install_scripts	Install scripts (Python or otherwise).
install_data	Install data files.
register	Register the distribution with the Python package index.
sdist	Create a source distribution (tarball, ZIP file, etc.).
upload	Upload binary package to PyPI.

When using PyCharm's **Run setup.py Task** menu to generate a package, you are prompted to select an option. Each option may include its own assortment of settings, which are presented in PyCharm through a dialog box.

Windows Executable Generation

PyInstaller is a Python package that bundles a Python application and all of its dependencies (including Python itself) into a complete deliverable. A user can run the packaged application without having to install a separate Python interpreter and related support modules. PyInstaller

works with Python applications created in Python 2.7 and Python 3.3 and later. It supports both windowed (for example, those that use wxPython) and console apps.

PyInstaller works with Windows, Mac OS X®, and Linux, but it does not generate packages across platforms. For example, to create a Windows executable package, you must run PyInstaller on a Windows computer. PyInstaller is available through PyPI, and other sources such as GitHub® and the PyInstaller website (**www.pyinstaller.org**).

Access the Checklist tile on your CHOICE Course screen for reference information and job aids on How to Generate a Windows Executable.

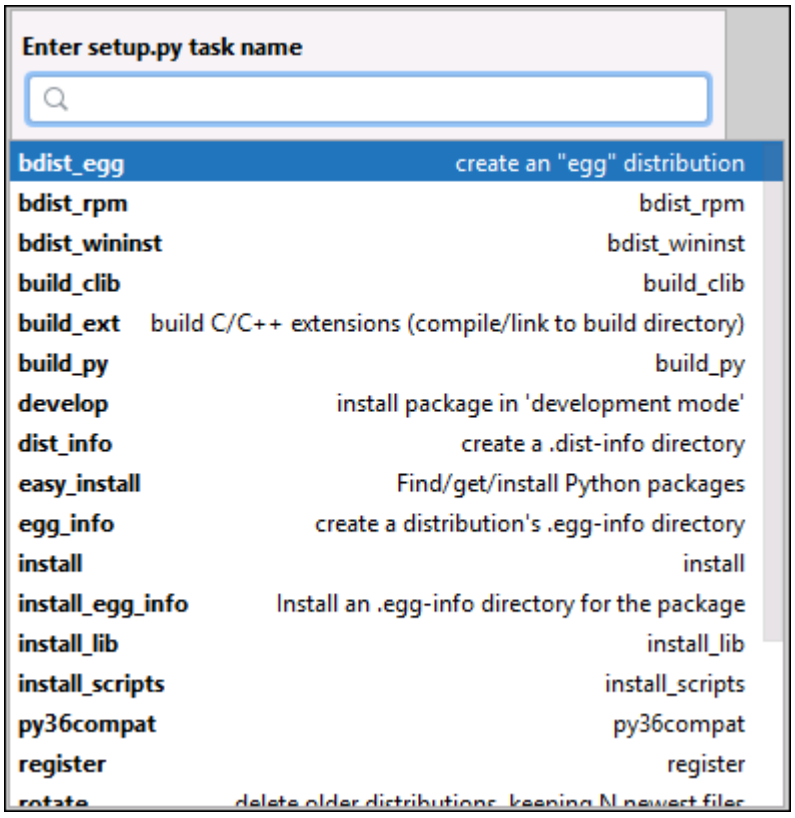
ACTIVITY 8-2
Generating Alternative Distribution Files

Before You Begin
The **WDCWebService-L8** project is open in PyCharm.

Scenario
You have prepared the **Wood Delivery Calculator** web service application for the packaging process. All that is left now is to generate and release the distribution files.

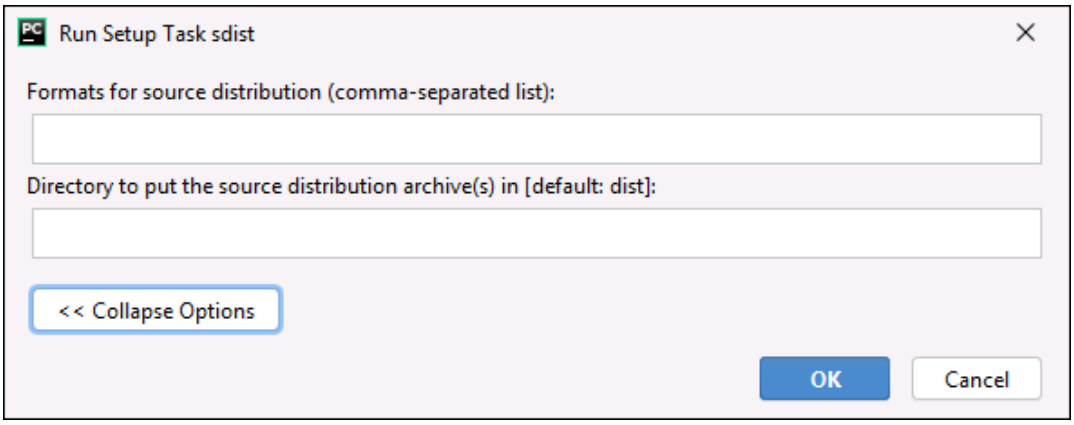
---

1. Run the packaging task.
   a) In the **Project** pane, select the **WDCWebService-L8** project.
   b) Select **Tools→Run setup.py Task**.
   c) Scroll through the various types of packages you can build.



   d) Type **sdist** to select the `sdist` distribution method, and press **Enter**.
   e) In the **Run Setup Task sdist** dialog box, select **Expand Options**.

f) Examine the options for this distribution type.



You can change any setting from the default value by providing your own value in the form.

g) In the **Formats for source distribution (comma-separated list)** box, type ***zip***

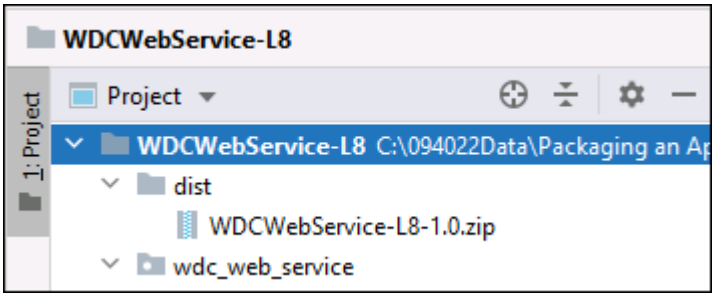
	Note: The default format is tar.gz.

h) Select **OK**.

The distribution is created, as you can see by the **dist** folder that has been added to the **Project** pane.

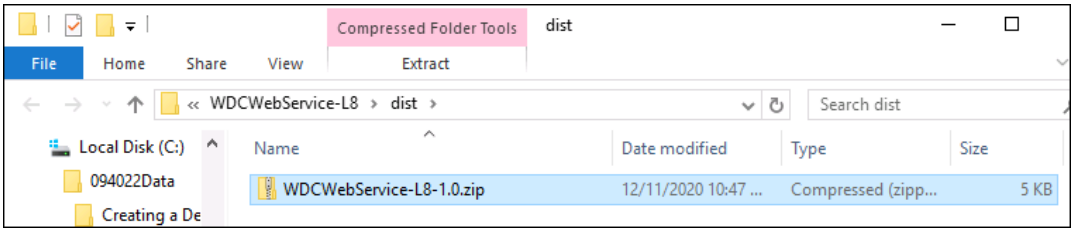
2. View the distributable package.

a) In the **Project** pane, expand the **dist** folder.



The folder contains a ZIP file that in turn contains the application package.

b) In the **Project** pane, right-click **WDCWebService-L8-1.0.zip**, and select **Show in Explorer**.



3. Examine the ZIP file.

a) In **File Explorer**, double-click **WDCWebService-L8-1.0.zip** to open it as a compressed folder.

The ZIP file contains one folder, **WDCWebService-L8-1.0**.

b) Open **WDCWebService-L8-1.0**.

The package includes a **PKG-INFO** file, **setup.cfg** file, **setup.py** script, and the **wdc_web_service** and **WDCWebService_L8.egg-info** folders.

c) Open **wdc_web_service**.

The folder includes all of the Python files you included in the package folder.

d) Close **File Explorer**.

e) Select **File→Close Project**.

ACTIVITY 8-3
Generating a Windows Executable

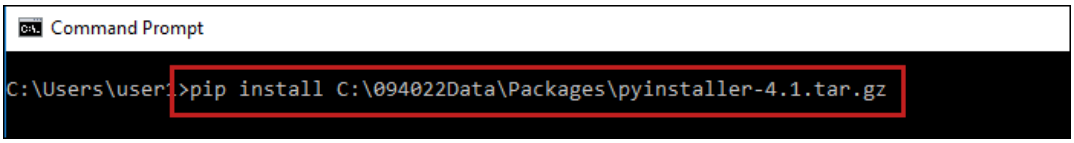
Data Files
All project files in C:\094022Data\Packaging an Application for Distribution\WDCClient-L8.

Before You Begin
PyCharm is open.

Scenario
Computers that will run the **Wood Delivery Calculator** client will not necessarily already have Python installed, so you will generate a version of the application that will include its own Python runtime environment.

---

1. Install PyInstaller from a command console.
   a) Open Command Prompt.
   b) In the Command Prompt, type the following and press **Enter** to install PyInstaller.



2. Examine other PyInstaller options.
   a) Type ***pyinstaller -h*** and press **Enter**.
      A list of options available for the `pyinstaller` command are shown.
   b) Scroll the console window to view the various options provided by the `pyinstaller` command.
   c) Close the Command Prompt.

3. Open the **WDCClient-L8** project.
   a) In the Welcome to PyCharm window, select **Open**.
   b) Navigate to the **C:\094022Data\Packaging an Application for Distribution\WDCClient-L8** folder.
   c) With the **WDCClient-L8** folder selected, select **OK**.

4. Use PyInstaller to create a Windows executable.
   a) Select **View→Tool Windows→Terminal**.
   b) Type `pyinstaller wdc-client.py -w` and press **Enter**.
      Without the "-w" parameter, a console window will open, from which the **Wood Delivery Calculator** is launched. The console output from your application would appear in the console window. Here we are using the option to create an executable that does not display the console window.
   c) PyInstaller runs in the project directory and creates a **build** and **dist** folder.

5. Launch the **Wood Delivery Calculator** web service.
   a) Select **File→Close Project**.

To test the WDC client, you don't need to keep its project open, since you can run it directly from its EXE file. But you will need to launch the web service.

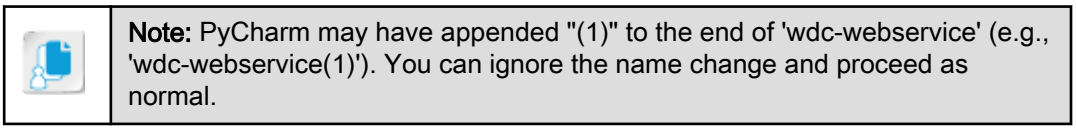
b) In the Welcome to PyCharm window, select **Open**.

c) Navigate to the **C:\094022Data\Packaging an Application for Distribution\WDCWebService-L8** folder.

d) With the **WDCWebService-L8** folder selected, select **OK**.

e) In the **Project** pane, expand **WDCWebService-L8→wdc_web_service**.

f) Right-click **wdc-webservice.py**, and select **Run 'wdc-webservice'**.



Note: PyCharm may have appended "(1)" to the end of 'wdc-webservice' (e.g., 'wdc-webservice(1)'). You can ignore the name change and proceed as normal.

g) Minimize the PyCharm window.

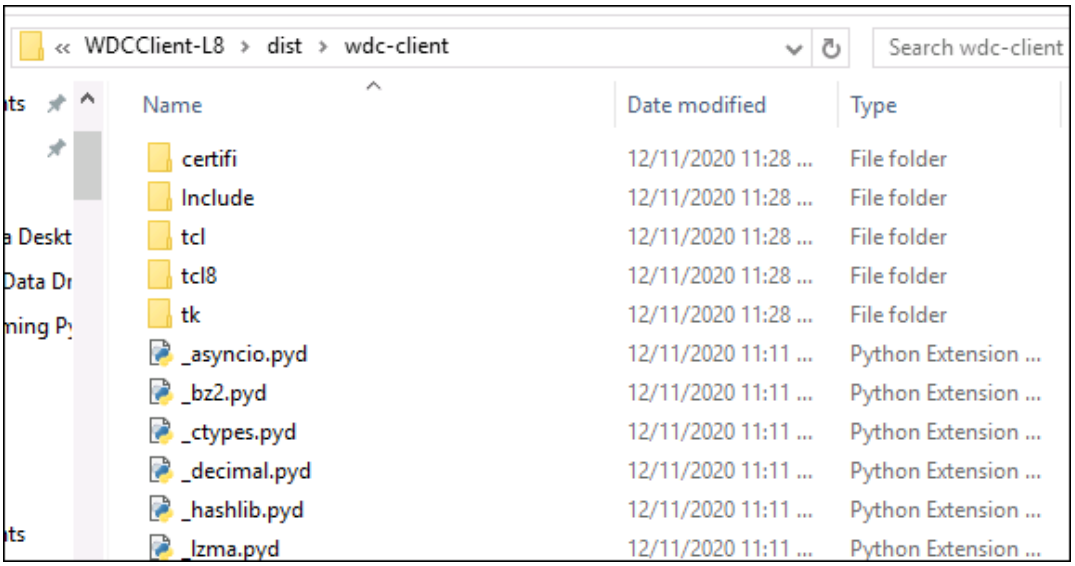
The web service will continue running with the window minimized.

6. Run the Wood Delivery Calculator client executable.

a) In File Explorer, navigate to **C:\094022Data\Packaging an Application for Distribution\WDCClient-L8**.

The **build** and **dist** folders created by PyInstaller are visible here.

b) Open the **dist** folder, and open the **wdc-client** folder.

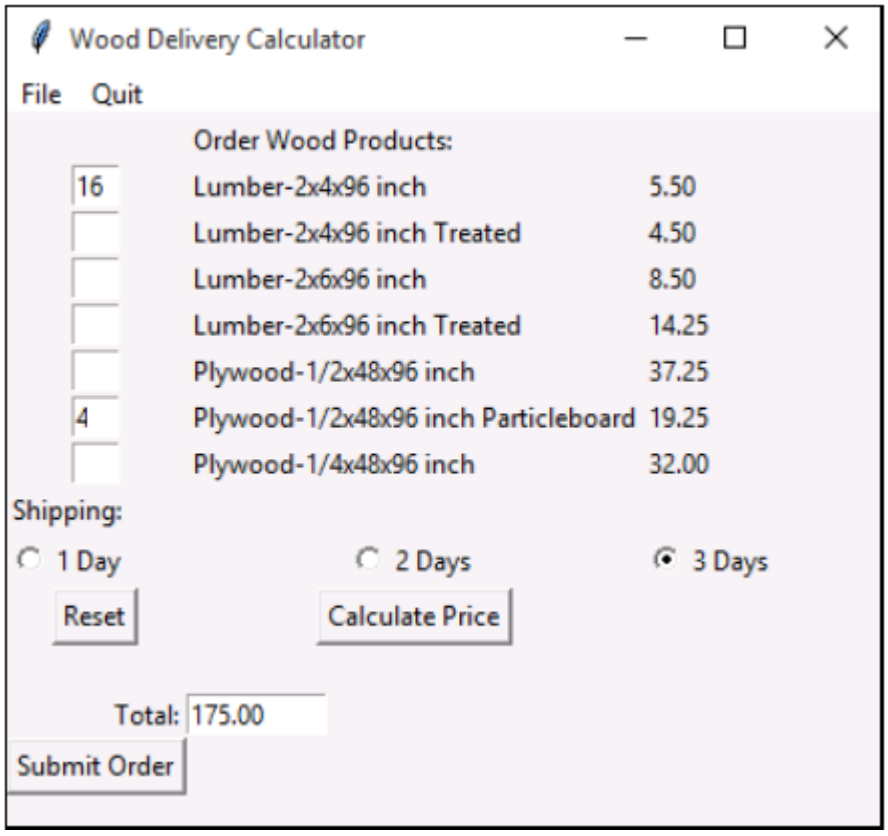


This directory contains the files you would distribute to users. These include the **wdc-client.exe** executable file, along with runtime files needed to provide the Python environment.

c) Double-click **wdc-client.exe**.

The Wood Delivery Calculator window opens.

d) In the Wood Delivery Calculator window, specify the number of products and shipping method to match the image below, and select **Calculate Price**.



e) Select **Submit Order**.

f) Close the Wood Delivery Calculator window to exit the application.

7. Clean up the desktop.

a) Close the File Explorer window.

b) In PyCharm, in the **wdc-webservice Run** tab, select the **Stop 'wdc-webservice'** button.

c) Select **File→Close project**.

d) Close PyCharm.

Summary

In this lesson, you created a package structure for a Python application project, and you generated distribution files.

How do you plan to distribute your Python applications to other users?

What sorts of distribution options will you use to distribute your packages?

Note: Check your CHOICE Course screen for opportunities to interact with your classmates, peers, and the larger CHOICE online community about the topics covered in this course or other topics you are interested in. From the Course screen you can also access available resources for a more continuous learning experience.

Course Follow-Up

Congratulations! You have completed the *Advanced Programming Techniques with Python®* course. You have successfully created, tested, packaged, and delivered Python applications that include graphical user interfaces, networking, database, and data science features.

What's Next?

To continue your Python learning journey, consider the following course: *Using Data Science Tools in Python ®*. In this course, you will use various Python tools to load, analyze, manipulate, and visualize business data. This course will teach you the skills you need to successfully use key libraries to extract useful insights from data, and as a result, provide great value to the business.

You are encouraged to explore Python further by actively participating in any of the social media forums set up by your instructor or training administrator through the **Social Media** tile on the CHOICE Course screen. You may also wish to explore the official Python documentation website at **https://docs.python.org/3/** for more information about the language.

Solutions

ACTIVITY 1-1: Designing OOP Project Requirements

1. **What classes would you define for inventory items?**

   A: Answers will vary, but may include: wood for all products, and then subclasses for different types of wood products that can have specific attributes (lumber, plywood, etc.), each product type has its own class with no subclasses, a class for each type of wood with subclasses for each product type, etc.

2. **What attributes might you need for the wood products?**

   A: Answers will vary, but may include: name, price, cost, width, height, length, depth, material, etc.

3. **What methods might you need?**

   A: Answers will vary, but may include: one for each `get` attribute, one for each `set` attribute, perhaps one to display results of other methods, add a product, delete a product, etc.

ACTIVITY 1-2: Justifying OOP Use for a Coding Project

1. **If you had to troubleshoot an issue with the Woodworkers Wheelhouse project, how might OOP make that effort easier?**

   A: Answers will vary, but may include: You don't have to look for comments or sift through all of your code to tell you where to look. You can see by the module, class, and/or function which part of your code is having an issue. Modularity also helps facilitate unit testing.

2. **The timeline for the Woodworkers Wheelhouse project does not allow for much extra time when developing. How might OOP save time with coding?**

   A: Answers will vary, but may include: By reusing code, either by calling a class or function from multiple places in your code, or through inheritance where you create subclasses that inherit attributes from the superclass.

3. You will have a small team to help develop the Woodworkers Wheelhouse project. How might OOP make the development for the project easier?

   A: Answers will vary, but may include: Developers can work on different objects simultaneously while minimizing the chance that one developer might duplicate someone else's functionality.

ACTIVITY 3-1: Designing a GUI

2. How many windows will the application use?

   A: While applications may have multiple windows, this application is simple enough that it can be implemented in one window.

3. What type of widget(s) might you provide to enable the user to specify the number of each product to purchase?

   A: Since they can type in a number of products to purchase, a group of entry boxes with labels for the product name and price would be an appropriate choice for this.

4. What type of widget(s) might you provide to enable the user to select a shipping method?

   A: Since only one shipping method should be selected from the three options, a group of radio buttons is probably the most appropriate choice for this. Another option would be a list box configured to allow only one item to be selected.

5. What type of widget(s) might you provide to enable the user to clear (reset) the selections?

   A: A button would be appropriate for this. When selected, a **Reset** button could trigger an action that resets the application widgets back to their original selection states.

6. What type of widget(s) might you provide to enable the user to calculate the total?

   A: A label or text box could be used to display the resulting price. There are different ways to trigger the calculation. For example, you might calculate a new result automatically each time the user changes the quantity of a product or shipping method. Or you might provide a separate **Calculate price** button that displays the updated total when the user selects the button.

ACTIVITY 5-1: Defining a Network Application Protocol

1. What connection protocol and port will you use for the application?

   A: For a server, since the **Wood Delivery Calculator** needs to accurately send product, price, and order data across the network, TCP is an appropriate choice. Regarding the port, many numbers are available and appropriate. For a web service, HTTP or HTTPS would most likely be used.

2. **You will modify the Wood Delivery Calculator to be either the client application or the server application. Which one should it be?**

   A: Since customers will run the **Wood Delivery Calculator** window, you'll use that as the client device. You'll create a separate server or web service program to serve the product and price values and write data to the **wworders** database. For the server, it will listen on a TCP port and the client (the **Wood Delivery Calculator** window) will request a connection to the server program. A web service would wait for Internet connection from the client.

3. **Which application should initiate the connection?**

   A: The server or web service will remain running, listening for connections. Clients will come and go. A client initiates a connection and requests data from the server/web service. The server/web service will send the requested data to the client.

4. **What requests must the server\web service be able respond to?**

   A: In addition to **opening to a new connection**, the server\web service must be able to provide **product**, **prices**, accept submitted order data, or **terminate the connection (exit)**. The client will send these commands to the server/web service. The server/web service will return data in response to received client commands. If the server/web service receives a `product` command, it will send the product list. If the server/web service receives a `prices` command, it will send the prices list. Finally, if the server receives an `exit` command, it will terminate the connection with the client.

Glossary

abstract class factory
Creates the superclass with abstract templates for any method that you want to use for subclasses.

abstraction
A tenant of object-oriented programming where the goal is to focus on the essential features and capabilities of an object relative to the context in which it is used.

assertion
A statement in a unit test function that determines if a particular rule passes or fails.

attribute
Features of an object (such as the weight, engine, and number of doors of a car). A class can contain many attributes, each describing a different feature of the object.

behavior
The functionality that objects of a given class have and how they respond to messages. An object's behavior is defined by its methods.

class
Data and functions, grouped together into a single entity, that can be instantiated to create an object.

class attributes
Attributes defined in the base class that apply to all subclasses of the base class.

class composite
A class composite can contain one or more objects for another class component.

class methods
Methods defined in the base class and available in all instances of the class.

client/server paradigm
A model for delegating processing capabilities between multiple systems, including a server, which provides access to shared resources (such as stored files or data in a database), and one or more clients, which communicate with the server to gain access to those resources.

comparison method
A function of a class, which can be used to determine if two objects of that class are the same (equal) or different (greater than, less than, etc.). Python provides a variety of standard comparison methods that you can customize in your own classes to enable objects of that class to be easily compared.

composition
A basic principle of object-oriented design. You can create a new object from other objects, creating a class composite.

constructor method
A special method, defined within a class using the __init__ keyword, that runs when an object of that class is instantiated.

cursor
A pointer to a particular storage location within a database, which you can use to navigate through a database and manipulate individual objects in the database.

data science
The application of domain-based knowledge, mathematical and statistical analysis, as well as programming skills to find patterns, evaluate cause and effect, and make estimations from data.

data scientist
Someone who analyzes data for actionable insights.

data visualization
The act of using visual materials such as charts, graphs, maps, and other visualizations to analyze data, to find patterns in data, and to report insights gleaned from data.

design patterns
A formalized approach to coding used in software development.

destructor
A special method, defined within a class using the __del__ keyword, that runs when an object of that class is destroyed.

encapsulation
This is the process of obscuring the complexity of an object, while providing user interfaces to access the data and capabilities of the object.

event
A condition or combination of conditions that trigger a method to run.

event handler
A method that runs in response to a particular user action, such as selecting a button or pressing a key on the keyboard.

factory pattern
Also called a class factory, gathers all subclasses that belong to the same superclass together into a single class definition.

file pointer
An object that tracks the location of data being accessed within a file.

frame
An object that serves as the main content area for a window, which contains other user interface objects that appear within the window.

framework
A code library that includes reusable code and extensions for common operations, as well as development tools, and a community of developers to get help from.

functional programming
In a functional programming style, coders create and apply functions, essentially treating statements like math equations.

getter
A special method used to provide protected read access to data values within a class.

grid method
A scheme for laying out user interface components within a window, which uses a system of columns and rows to position components.

GUI
(graphical user interface) Uses visual components (graphics) to allow you to interface with the software. Most modern operating systems provide a GUI as a way to interface with the operating system.

hierarchy
Allows you to rank or order abstractions to create subsystems made up of different components of the system.

imperative programming
Imperative programming statements change the state of the application, by focusing on "how" data should be processed. Scientists like imperative

programming because they can write code that clearly reflects their processes.

inheritance
An aspect of object-oriented programming that enables developers to define a subclass that inherits attributes and methods of a superclass.

instance attributes
Attributes that apply only to an instance of a class.

instance methods
Methods defined in, and usable by, a specific instance of a class.

linear regression
An approach to modelling the relationship between two variables by fitting a linear equation to observe data.

machine learning
Applies artificial intelligence to automatically learn from experience in order to improve accuracy, operations, or other performance metrics without being explicitly programmed.

magic method
Built-in methods that you can use when defining a class to add certain types of functionality commonly required by classes, including constructors, destructors, comparison methods, and so forth.

method
Similar to Python functions, perform an operation using the attributes of the class. A class method should be contained within the class and only use the attributes from that class.

modularity
The act of breaking down problems and tasks into modules to reduce the complexity of the problem and the solution.

network byte order
A data format optimized for sending data across a network connection.

object
A collection of data and associated attributes and behaviors in an object-oriented program.

object-oriented design
A software design and planning process for addressing software requirements with a system of interacting objects.

OOP
(object-oriented programming) Organizes data and code into objects that can be manipulated by code. This allows developers to focus on objects they want to work with, and what they want to do with those objects rather than application logic.

package folder
A special container within an application project that contains only the files that you intend to be delivered to the user's computer when the package is installed.

packing method
A scheme for laying out user interface components within a window, which attempts to pack widgets into a window as efficiently as possible in the available space.

place method
A scheme for laying out user interface components within a window, which requires you to provide a specific X,Y location for each widget.

polymorphic class factory
The polymorphic class factory references all of the overload methods in subclasses, and uses the correct one based on the subclass that was instantiated.

polymorphism
Where a subclass can define its own methods, but it can also define the same methods as the base class.

procedural programming
Procedural style is a step-by-step approach to coding. You write code to perform one task, then write additional code to perform additional tasks. This is the style most coders start with because it is the simplest and works well for iterative tasks.

property
A protected attribute of a class that can be set only by using special instance methods called

setters, and whose values can be read only by using special class methods called getters.

property method
An interface to access instance attributes.

PyInstaller
A Python package that bundles a Python application and all of its dependencies, including Python, into a complete deliverable.

PyPI
(Python Package Index) A central clearinghouse for shared Python applications and code.

R Project
An open source programming language and software environment for statistical computing and graphics.

regression
A type of statistical analysis that looks for relationships between variables.

serialization
A means of encoding complex data types (objects) for transmission or storage as a simple text value.

setter
A special method used to provide protected write access to data values within a class.

subclass
A class based on another class, which inherits attributes and methods of the superclass, and can add its attributes and methods specific to the subclass.

superclass
The base class from which a subclass is derived.

TDD
(test-driven development) A software development methodology that involves designing and implementing tests before writing the code that will be tested. Because tests are created first, they can be used to drive development. Once a unit of code is thought to be complete, it can be tested and modified repeatedly until it passes testing. Once a unit has passed, its unit tests can remain in place to ensure that if changes are made to code later on, they do not cause the code to fail.

widget
Short for window gadgets that display and retrieve information in a GUI. Most graphical programming languages provide a library of widgets for you to use in your programs.

wireframe
A type of drawing or diagram that illustrates the logical design of a user interface, without necessarily focusing on the visual style of the user interface.

Index

W

094022S rev 1.1
ISBN-13 978-1-4246-4078-2
ISBN-10 1-4246-4078-4