Chapter 21
Classes

Abstract Classes are the central construct for object-oriented programming (OOP). Knowing how to design your own objects will deepen your understanding of Python and help with learning other OOP languages. Object-oriented programming requires a paradigm shift from functional programming, which essentially works by grouping related steps into reusable functions. In functional programming, to make an omelet, you get six eggs and repeat `break_egg` until all the eggs are broken. In contrast, OOP revolves around the objects involved in the problem. Methods and properties belong to object classes; Making an omelet involves `Egg` objects. The `Egg` objects can have a `shell` property and a `break_shell` method and you could make each egg object break its shell. It may sound strange at first, but thinking about objects you have already been using will help. When you create a list, you're creating an instance of a list object. Then you apply list methods to it, such as, `mylist.sort()`). Similarly, you have used geoprocessing objects, such as `Describe` objects which have properties such as `dataType` and `dataPath`. Throughout this book you have used objects, methods, properties, and dot notation. All of these are aspects of OOP. Up to this point, you haven't designed your own objects; In this chapter you'll learn the syntax to do so. The examples in this chapter demonstrate basic OOP syntax and some of the benefits.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- List some benefits of object-oriented programming.
- Explain the terms 'object', 'class', 'property', and 'method'.
- Define a class and create an object instance.
- Create class properties and methods.
- Modify object properties and call object methods.
- Call a class defined in a user-defined module.

21.1 Why Use OOP?

User-defined functions increase code reusability. However, they have their limitations. Functions don't store information like variables do. Every time a function is run, it starts afresh. Functions and variables are often closely related to each other and need to interact with each other. As an example, suppose you are studying park maintenance and utilization using a dataset of hiking trails. Each trail has an identification number. You could store the trail IDs in a list as follows:

```
>>> trailList = [1, 2, 5, 10, 15]
```

As the database is updated, you may need to modify the trail list. To do so, you could append or remove IDs from the list:

```
>>> trailList.append(16)
>>> trailList.remove(5)
```

However, each land trail has additional information associated with it, such as the vegetation classification for the trail. How can you keep track of the vegetation? You might think of using a dictionary of trail IDs and vegetation classifications, as in the following example:

```
>>> trailVegetation = {1: 'barren', 2: 'some bare ground',
    5: 'stunted vegetation', 10: 'barren', 15: 'over-grown'}
```

Then you might need another dictionary for the trail lengths, and other dictionaries for the trail surfaces, the benches, the trail maintenance organizations, the man-made structures near each trail, the grade, and so forth. Functions need to access various trail data by passing in arguments and using the trail IDs to identify the value of each attribute for that trail. For example, a function calculating the costs for maintaining a trail needs to know the trail length and vegetation classification. The function, `calculateCost`, defined in Example 21.1 passes these as arguments. An upcoming example will show a different way to do this using OOP.

Example 21.1

```
def calculateCost(trail_ID, vegetation, length):
    '''Calculate trail maintenance based on
       vegetation and length.'''
    rate = 1000
    if vegetation[trail_ID] == 'barren':
        rate = 800
    elif vegetation[trail_ID] == 'some bare ground':
        rate = 900
    cost = length[trail_ID]*rate
    return cost
```

As the code grows, it becomes difficult to keep track of which functions need which variables and how everything relates. With all of these variables separate but depending on the same set of properties, maintaining the variables correctly can grow cumbersome. Suppose you need to remove the trail with ID number 5. Each variable that references that trail must be modified to reflect the removal. The trail 5 entry for the vegetation dictionary, the length dictionary, the volunteer dictionary, and so forth must all be deleted. This process could be made easier by grouping these trail attributes with the trail. Then when a trail needs to be added, updated or removed, the trail information is all together in one place. Object-oriented programming does this by grouping variables and related functions so the functions can act on the variables and interact with each other more smoothly.

Classes group closely related functions and variables together. They also provide a convenient way to work with a group of objects that have common attributes, such as a set of trails that each have a length, a vegetation classification, and a volunteer organization. A class allows you to design a basic trail and each time you need to create a new trail, you can just specify the values of its attributes. This concept of grouping functions and variables related to a particular type of item is the key principle of OOP. Classes are the container for these related functions and variables.

You design objects by creating an object template, called a *class*. Which acts as a blueprint for the object. The class specifies functions and variables associated with the object. In this way, the class structure provides a means for grouping related functions and variables.

21.2 Defining a Class

To define a Python class, use the `class` keyword followed by the class name and a colon. The contents of the class are indented to indicate that they are a related block of code. Python class names usually use upper camel case capitalization. The format for a Python class is as follows:

```
class ClassName:
    '''Class docstring.
    '''
    code statement(s)
```

Example 21.2 shows a `Trail` class definition. Think of a class definition as a blueprint. It isn't creating an object itself; It simply describes how to make one. You can create multiple instances of objects from the blueprint. The `Trail` class has three properties, `ID`, `length`, and `vegetation` and it defines three functions or *methods*. The term 'method' is used to refer to functions that are defined within a class. The `Trail` class contains the methods, `__init__` (a special method), `calculateCost`, and `reportInfo`. The argument list for all three of these methods starts with a special variable named `self`. Next we will discuss how to create an object from the class blueprint and the special role of the `__init__` method and the `self` variable in creating objects.

Example 21.2: A `Trail` class, created using the `class` keyword and indenting all of the contents.

```
# excerpt from trailExample.py
class Trail:
    '''Pedestrian path.

       Properties:
         ID          A unique identifier
         length:     Length in kilometers
         vegetation: Plant growth on the trail
    '''
    def __init__(self, tid, theLength, theVegetation):
        '''Initialize trail properties.'''
        self.ID = tid
        self.length = theLength
        self.vegetation = theVegetation

    def calculateCost(self):
        '''Calculate maintenance costs based
           on vegetation and length.'''
        rate = 1000
        if self.vegetation == 'barren':
            rate = 800
        elif self.vegetation == 'some bare ground':
            rate = 900
        cost = self.length*rate
        return cost

    def reportInfo(self):
        '''Print trail properties'''
        print 'ID: {0}'.format(self.ID)
        print 'Length: {0}'.format(self.length)
        print 'Vegetation: {0}'.format(self.vegetation)
```

21.3 Object Initialization and Self

Once the `Trail` class is defined, you can create a `Trail` object, a specific instance of a `Trail` with a particular ID, length, and vegetation classification. The syntax for creating an object instance looks similar to calling a function that returns a value. You call the class, using the class name followed by parentheses containing arguments and store the return value in a variable, as in Example 21.3. If you attempt to run Example 21.3 without running Example 21.2 beforehand, you will receive a `NameError`, since `Trail` is not yet defined.

When you call a class, the `__init__` function inside the class is automatically invoked. The `__init__` method for the `Trail` class sets the values of `self.ID`, `self.length`, and `self.vegetation`. The terms `__init__` and `self` need some additional explanation.

- `__init__` is a special method which is automatically called when an object of that class is created. It is used to create an initial state for the object (similar to a constructor in other languages such as C++ and Java). Use one of these inside your class to initialize object properties. Note that the name `__init__` requires two underscores on the front and two on the back; otherwise, it will not be recognized as the initialization method.
- `self`, as the word suggests, refers to the particular instance of the object itself. This means that when you set `self.vegetation` with an assignment statement inside of `__init__`, the vegetation property for the current object is initialized. You should use `self` as the first parameter in the list of any function defined within a class. This allows you to refer to object properties within other methods without passing them as arguments. For example, `calculateCost` uses `self.vegetation` and `self.length` without passing these into this method. Though you could use a name other than `self` as a placeholder for the first position in the parameter list, it is best to conform to convention so that your code is consistent with others.

When a class is invoked, the caller must provide arguments for the argument list in the `__init__` method. When the `Trail` class is invoked in Example 21.3, the caller provides three arguments. The argument list should correspond to the `__init__` argument list, except for `self` which is passed implicitly into all class methods. Since the `self` variable is passed implicitly, only three, not four, arguments are needed to create a `Trail` object. The three arguments, an identification number, a length, and a vegetation classification, must be specified in this order in the class call. The `__init__` method constructs an instance of the object and initializes its properties. The class returns an object instance, which can be saved using an assignment statement. Example 21.3 saves the returned object in a variable named `myTrail`.

Example 21.3: Creating an object instance. This statement creates a `Trail` instance called `myTrail`.

```
myTrail = Trail(1, 2.3, 'barren')
```

21.4 Using Class Objects

A class blueprint can define object properties and methods. An instance of a `Trail` object has three properties (`ID`, `length`, and `vegetation`), and two methods (`calculateCost` and `reportInfo`) in addition to the `__init__` method. Notice the differences between the class method, `calculateCost`, in Example 21.2 and the stand-alone `calculateCost` function defined in Example 21.1. Example 21.1 uses three parameters (`trail_ID`, `vegetation` and `length`); Whereas, Example 21.2 only uses `self` as a parameter. The class method does not need to use a trail ID number to access its vegetation classification or the length, because the `Trail` object stores this information about itself. It can simply refer to `self.vegetation` and `self.length`. Variables that are assigned a value within the class `__init__` method as `self.variableName` (such as `self.length` and `self.vegetation`) are object *properties* or *attributes*. General Python documentation uses the term ‘attribute’, but ArcGIS Resources documentation uses the term ‘property’. This book generally uses the term ‘property’, for consistency with the `arcpy` documentation. This is the same terminology we used earlier to discuss `arcpy` object properties, such as the `dataType` property of the `arcpy Describe` object. Using properties and methods from user-defined classes has the same format as using other objects. Once a `Trail` object has been created, you can access its properties and methods using dot notation. The following code creates a `Trail` object, prints the `length` and `vegetation` properties and runs the `calculateCost` method:

```
>>> myTrail2 = Trail(2, 5.0, 'some bare ground')
>>> myTrail2.length
5.0
```

```
>>> myTrail2.vegetation
'some bare ground'
>>> myCost = myTrail2.calculateCost()
>>> myCost
4500.0
```

You don't have to pass trail length or vegetation information into the `calculateCost` function because these properties are stored with the trail itself. This centralization of control on variables is an advantage of OOP over functional programming. You can change the values of object properties and the value stored by the object will be modified. The following code changes the length property of `myTrail2` to 2.1 and then `calculateCost` uses that new length to return a much lower cost for this trail:

```
>>> myTrail2.length = 2.1
>>> myTrail2.calculateCost()
1890.0
```

`myTrail2.length` corresponds to `self.length` inside the class definition. The value of `self.length` for this object changed when `myTrail2.length` changed. When referring to an object property from outside of the class definition, use the variable name before the dot (e.g., `myTrail2.length`); When referring to an object property inside the class definition, use `self` before the dot (e.g., `self.length`).

A class blueprint allows you to create multiple object instances. The examples above each create one `Trail` object, but you can make multiple `Trail` objects and place them in a list or dictionary. The sample dataset, 'trails.txt' contains trail ids, lengths, and vegetation classifications, one trail per line (e.g., `'1,2.3,barren'` is the first line). Example 21.4 reads this data, creates a set of `Trail` objects, and stores them within a dictionary.

Example 21.4: Read a dataset and create `Trail` objects.

```
# excerpt from trailExample.py
data = 'C:/gispy/data/ch21/trails.txt'
trailDict = {}
with open(data, 'r') as f:
    # Read each line.
    for line in f:
        # Strip '\n' from the end and split the line.
        line = line.strip()
        lineList = line.split(',')
        tID = int(lineList[0])
        tLength = float(lineList[1])
        tVeg = lineList[2]
        # Create a Trail object.
        theTrail = Trail(tID, tLength, tVeg)
        # Add the Trail object to the dictionary.
        trailDict[traID] = theTrail
```

The `trailDict` dictionary uses trail identification numbers as keys:

```
>>> trailDict.keys()
[1, 2, 10, 5, 15]
```

The dictionary values are `Trail` objects:

```
>>> trailDict.values()
[<__main__.Trail instance at 0x18E8C530>,
 <__main__.Trail instance at 0x18E8C5D0>,
 <__main__.Trail instance at 0x18E8C620>,
 <__main__.Trail instance at 0x18E8C5F8>,
 <__main__.Trail instance at 0x18E8C670>]
```

The `reportInfo` method prints the object properties:

```
>>> for t in trailDict.values():
...     t.reportInfo()
ID: 1
Length: 2.3
Vegetation: barren
ID: 2
Length: 5
Vegetation: some bare ground
ID: 10
Length: 1.6
Vegetation: barren
ID: 5
Length: 4.2
Vegetation: stunted vegetation
ID: 15
Length: 20.0
Vegetation: over-grown
```

An additional `Trail` object can easily be added to the dictionary:

```
anotherTrail = Trail(3, 2.56, 'barren')
trailDict[3] = anotherTrail
```

Or an object can be deleted from the dictionary, using a dictionary key to identify the `Trail` object to be removed:

```
>>> del trailDict[5]
```

Contrast this object-oriented means of storing the trail information with the functional approach. The functional approach creates a list for IDs, dictionaries for lengths and vegetation classifications and then each of these would need to be modified to remove a trail. An OOP `Trail` object can easily be deleted, using the `del` keyword. Deleting the object takes care of removing all of its properties too. Suppose you need to remove all trails longer than 10 km. With the OOP approach, we can also easily delete trails based on one of their properties. The following code deletes long trails:

```
# Delete trails whose length exceeds 10 km.
for t in trailDict.values():
    if t.length > 10:
        del trailDict[t.ID]
```

With the functional approach, the programmer is responsible for keeping track of all attributes associated with a particular ID. Deleting trails based on length would require using more than one loop and deleting each attributes one by one, as in the following implementation:

```
# functionalTrailDelete.py
trailList = [ 1, 2, 5, 10, 15]
trailVegetation = {1: 'barren', 2: 'some bare ground',
5: 'stunted vegetation', 10: 'barren', 15: 'over-grown'}
trailLength = {1: 2.3, 2: 5.0, 15: 4.2, 10: 1.6, 5: 20.0}

# Determine which trails to delete.
longT = []
for k, v in trailLength.items():
    if v > 10:
        longT.append(k)
# Delete the long trails and each corresponding properties.
for i in longT:
    trailList.remove(i)
    del trailLength[i]
    del trailVegetation[i]
```

With OOP, it's also easy to modify objects based on their properties. The following code adds 1.2 km to the `Trail` length if the vegetation is ‘barren’:

```
# Increase barren property lengths by $5000.
for t in trailDict.values():
    if t.vegetation == 'barren':
        t.length = t.length + 1.2
```

Performing these deletions and modifications within the functional approach would again require managing multiple data structures to maintain the accuracy of each property. The OOP approach simplifies these operations. An additional method can be added to the class by simply adding another indented function within the class that uses `self` as the first parameter:

```
def calculateCrowding(self, visits, track):
    '''Calculate number of visitors/100 m
        (count double for narrow trails).'''
    if track == 'single':
        val = 2*visits/(self.length*10)
    else: # default unit is square meters
        val = visits/(self.length*10)
    return round(val,2)
```

Since `self` is passed implicitly, only two arguments are needed to call this method (one less argument than the length of the parameter list):

```
>>> t = Trail(11, 9.5, 'barren')
>>> t.calculateCrowding(20, 'single')
0.42
```

Extensibility is another OOP quality that strengthens reusability. An additional property can be added to a class that's already in use by adding an optional argument to the `__init__` method argument list. For example, a `visits` property can be added to the `Trail` class by modifying the `__init__` method as follows:

```
class Trail:
    def __init__(self, tid, length, vegetation, visits = 0):
        '''Initialize trail properties.'''
        self.ID = tid
        self.length = length
        self.vegetation = vegetation
        self.visits = visits
```

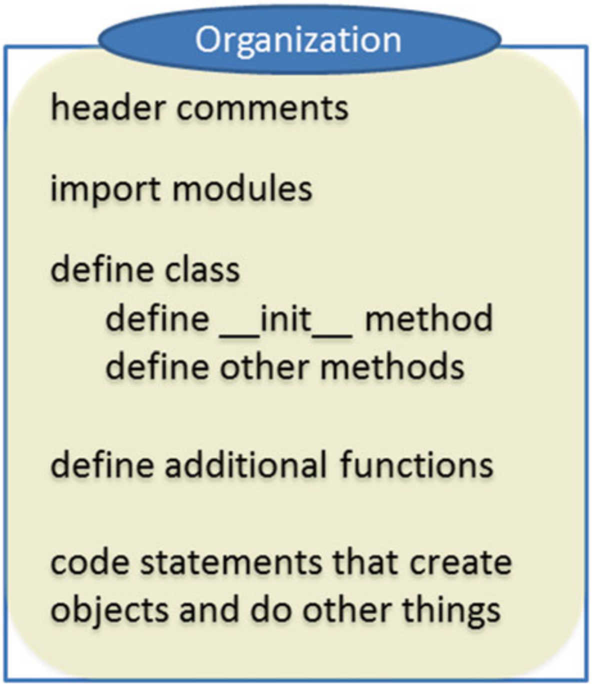
Then the `Trail` objects can be created in one of two ways:

```
>>> t1 = Trail(1, 14.5, 'over-grown')
>>> t2 = Trail(2, 3.4, 'some bare ground', 3.5)
```

When only three arguments are used, the `visits` property value defaults to zero;

```
>>> t1.visits
0
>>> t2.visits
3.5
```

21.5 Where to Define a Class



Just like user-defined function definitions, class definitions must precede any class calls, else a `NameError` will occur. Generally, a class should be placed at the top of a script after the header comments and module imports. Within the class, the `__init__` method should be defined first, followed by any other class methods. Next, stand-alone user-defined functions should be defined. Finally, the main flow of the code should be placed last. This code may contain statements that create objects and perform other tasks. Example 21.5 shows code that contains header comments, followed by a class definition, followed by code that creates objects and uses them.

Example 21.5: Defining a class, creating an object instance, and using the object properties and methods.

```
# highwayInfo.py
# Purpose: Define a highway class.
class Highway:
    '''
    Major road class.
    Properties:
        name          None-numerical road name.
        length        Length based on a network dataset.
        travelTime    Estimated time it takes to travel the
                      full length of this highway.
    '''
    def __init__(self, name, length, tTime):
        '''Initialize a Highway object.'''
        self.name = name
        self.length = length
        self.travelTime = tTime

    def calculateAvgSpeed(self):
        '''Calculate the average speed.'''
        avgSpeed = self.length/float(self.travelTime)
        return avgSpeed

    def getOrientation( self, number ):
        '''Determine highway orientation based on the hwy number.'''
        if number%2 == 1: # if the number is odd...
            orientation = 'NS'
        else:  # else the number is even.
            orientation = 'EW'
        return orientation

    def report(self):
        '''Print highway properties.'''
        print
        print '''{0} Highway
-----------
{1} km long
{2} hours travel time
        '''.format(self.name, self.length, self.travelTime)

if __name__ == '__main__':
    hwy = highway('Lincoln', 4946, 100)
    hwy.report()

favoriteHighway = highway('Blue Ridge Parkway', 755, 19)
favoriteHighway.report()
```

Also, just like user-defined functions, the class definition may either be placed within the script that calls the class or within a separate user-defined module, so that it can be easily re-used. A separate user-module should place test code inside a conditional expression that checks `__name__` against `'__main__'` as discussed in Chapter 16.

In Example 21.5 sample script, 'highwayInfo.py', defines a class and then creates two objects instances, `hwy` and `favoriteHighway`. The first object, `hwy`, is created within the conditional construct, but the second object is created outside of the conditional construct to demonstrate the difference. The `hwy` object will only be created when the `__name__` is set to `'__main__'`. The report method, which prints the object properties, is called on both objects. When 'highwayInfo.py' is run as the main script, it creates two objects and prints a report for each:

```
>>> Lincoln Highway
-----------
4946 km long
100 hours travel time

Blue Ridge Parkway Highway
-----------
755 km long
19 hours travel time
```

When the `highwayInfo` module is imported (instead of being run as the main script), `hwy` is not created and the report method is not called for this object. But `favoriteHighway` is still created and the report method is called. The output printed by the import command shows this difference:

```
>>> import highwayInfo
Blue Ridge Parkway Highway
-----------
755 km long
19 hours travel time
```

To improve this script, the code that creates and uses `favoriteHighway` should be moved inside of the conditional construct so that it is not executed when the module is imported.

21.6 Classes Within a Separate User-Defined Module

Recall that to call a function defined in a separate user-defined module, you import the function and then use dot notation, placing the dot between the module name and function name. The same format applies to classes. To use a class within a separate user-defined module, import the module, and then create an object instance by using dot notation between the module name and class name. For example, the following code imports highwayInfo and creates a `Highway` object named `myHwy`:

```
import highwayInfo
myHwy = highwayInfo.Highway('Pacific Highway', 496, 5)
```

The object can then be called by its name, as usual. The module name is not used when referring to object instances. For example, the following code uses the `myHwy` object:

```
>>> myHwy.report()
Pacific Highway Highway
-----------
496 km long
5 hours travel time
```

Example 21.6 puts these steps together, importing the module, creating an object, and using it.

Example 21.6: Calling the Highway class in highwayInfo, from a separate script.

```
# handleHighways.py
# Purpose: Demonstrate using a class defined in a separate
#          user-defined module.
# Usage:   No arguments needed, but highwayInfo.py must be in
#          the same directory.
import highwayInfo
myHwy = highwayInfo.Highway('Pacific Highway', 496, 5)

spdLimit = myHwy.calculateAvgSpeed()
orient = myHwy.getOrientation(5)
print 'Avg travel speed: {0} km/hr'.format(spdLimit)
print 'Orientation: {0}'.format(orient)
```

21.7 Discussion

This chapter introduced basic syntax for creating user-defined objects. It focused on a few pertinent goals:

- Defining, instantiating, and using objects in Python.
- Exploring some advantages of OOP.
- Looking under the hood to expose OOP from the programmer's side and provide some deeper understanding of built-in objects.

For simple projects, functional programming is appropriate and sufficient. But for larger projects OOP can greatly improve reusability and maintainability of code. Associating properties and methods with an object provides a convenient way to manage the object-related data, particularly when multiple instances of an object are involved.

Designing object-oriented programs takes some adjustment when you're accustomed to functional programming. A good way to hone this skill is to practice thinking in terms of objects, properties, and methods. For example, which objects, properties, and methods can you identify to characterize the archeological study data analysis described in Example 21.7?

Example 21.7: Identify objects, properties, and methods to characterize the following scenario:

Field surveys are being planned for archeological study sites. Scripts will be used to estimate the acreage to be surveyed and the associated costs, and to plan 'shovel tests' and store the field notes. Shovel tests are small holes, 30-50 cm in diameter, dug to search for artifacts. The presence or absence and nature of artifacts is recorded for each shovel test along with the stratum, soil color, and texture. Evidence of past changes such as floods, bulldozing, or plowing is also noted. For these sites, the shovel tests are positioned in a grid for extensive coverage. An appropriate sampling interval (e.g., 30m, 20m, 10m, 5m) is determined for each site. A GIS can assign these positions as X and Y coordinates based on the coordinate system of the study site.

An OOP structure for the domain described in Example 21.7 could be formulated as follows:

Objects
Study sites, shovel tests, positions

Properties
Study site properties: acreage, cost, set of shovel tests, sample interval
Shovel test properties: ID, position, artifacts (present/absent), artifacts description, stratum, soil color, soil texture, past terrain changes, depth
Position properties: shovel test ID, X, Y

Methods
Study site methods: compute acreage, compute cost, calculate sampling interval, determine shovel test grid positions
Shovel test methods: initialize ID and position, set artifacts, set stratum, set soil color, set soil texture, set past terrain changes
Position methods: initialize shovel test ID, X, and Y.

The number of objects, methods, and properties may expand as the project progresses. Fortunately, OOP creates a flexible framework for growing projects.

Some of the OOP syntax can be confusing on first exposure. Here are a few key syntax details worth repeating:

- Refer to object properties with `self` before the dot when inside the class. Refer to object properties with the *property name* before the dot when outside the class.
- Object properties do not need to be passed as arguments into methods other than `__init__`.
- Use `self` as the first parameter when writing a function inside a class. (If a method does not need any input from the user, you still need to list `self` as the only parameter when writing a function inside a class.)
- When calling a class method, pass in one less argument than the length of the parameter list, since `self` is implicitly passed. (If a class method only lists the `self` parameter, don't use any arguments.)

21.8 Key Terms

The `class` keyword	Properties
Object-oriented programming	Methods
Functional programming	The `self` argument
A class versus an object instance	The `__init__` method
Instantiating an object instance	

21.9 Exercises

1. **air_classes** Plan the OOP components for a script that would support daily air traffic analysis of airlines and their flights. The script will use Airline, Flight, and Seat objects. Identify properties and methods for each of these objects based on the following description:

   Each airline name is stored along with a set of numbered flights. The origin and destination airports of each flight are stored and can be used to estimate the length of flight paths. Flight departure and arrival times are stored, so that the flight duration can be derived. Each flight has a set of seats organized in rows and columns. Each seat has a fare rate and a status (occupied or vacant) which can be modified when a new reservation is made. The class (first, business, or economy) can be determined based on the row. Flights can be added and removed. The capacity, the load factor, and the total fare for a flight can be calculated. The total number of flights for each airline for a day can also be derived.

   Airline properties (2):
   Flight properties (5):
   Seat properties (4):

Airline methods (3):
Flight methods (5):
Seat methods (2):

2. **seismic_classes** Plan the OOP components for a script that collects data samples from sensors. The script will use Sensor and DataSample objects. Identify properties and methods for each of these objects based on the following description:

   A network of 14 sensors will be lowered via helicopter into volcanic basins to measure seismic activity and weather conditions. Each sensor station will have an ID, an XYZ location, and daily sample count. Each sample will record a seismic signal as well as the start time of the reading, the air quality, temperature, humidity, and wind speed. The system needs to be able to identify invalid readings and identify seismic signals that exceed a significant threshold. The sensor will report significant seismic signals real-time. It will also routinely purge invalid samples, and then compute daily summary statistics for each reading. Finally, each sensor will need to transmit the summary results to a monitoring center.

   Sensor properties (3):
   DataSample properties (6):

   Sensor methods (4):
   DataSample methods (2):

3. **lineSegments.py** Add code to sample script 'lineSegments.py' to take four arguments representing the x and y values of two points, x1, y1, x2, and y2. For example, an input of 1 2 4 3 would represent two points, (1,2) and (4,3). Create an instance of a LineSegment object for the line segment from the first point to the second point (e.g., (1,2) to (4,3)).

   Add a method `printSegment` to print the values of the endpoints as Endpoint 1 and Endpoint 2 as shown below. Call `printSegment`. Then use the existing class methods to calculate the slope and length of the line segment and print these results. Next use the translateY method to shift the line segment down 3 units. Then call `printSegment` again.

   Example input: 1 2 4 3

   Example output:

```
>>> Endpoint 1:( 1.0, 2.0 )
Endpoint 2: ( 4.0, 3.0 )
Slope: 0.3
Length: 3.2
Endpoint 1:( 1.0, -1.0 )
Endpoint 2: ( 4.0, 0.0 )
```

4. **treeAnalysis.py** Add to the code found in sample script, 'treeAnalysis.py', to build a Tree class for manipulating forestry data like the data in RDUforest.txt:

   BLOCK PLOT SPECIES DBH 5 91 SG 14
   5 91 LP 23
   5 91 LP 17
   5 91 LP 18
   ...

   Each row represents a tree found in the specified blocks and plots near RDU airport. SG (Sour Gum), LG (Loblolly Pine), and so forth are abbreviations for the tree species. The diameter breast height (DBH), the trunk diameter at 4.5 ft from the ground, is a measure of the tree's maturity. The Tree class should have four properties, `block`, `plot`, `species`, and `dbh`, initialized in an `__init__` method. The class should also have three additional methods. A method named `report` is given already. Add a `calculateDIB` method that will calculate the diameter inside bark (DIB), assuming the DIB is DBH times 0.917. Then add a `calculateHeight` method to calculate height based on species type using the equations given in the script comments. Once the class is built, create four tree objects corresponding to the first four rows of 'RDUforest.txt'. Add code as instructed where you see `###` comments. The output should look like this:

```
>>> Tree 1 Species: SG
Tree 1 DIB: 12.838

Report Tree 1
-------------
Block: 5
Plot: 91
Species: SG
DBH: 14
DIB: 12.838
Height: 100.9

Tree 2 DBH: 23
Tree 2 Height: 87.175
Tree 3 block: 5
Tree 3 plot: 91

Report Tree 4
-------------
Block: 5
Plot: 91
Species: LP
DBH: 18
DIB: 16.506
Height: 87.05
```

5. **campusCrimes.py** Add code to sample script 'campusCrimes.py' to create an `Incident` class to handle crime incidents reported on a university campus. An Incident object should have `ID`, `time`, `type`, `location`, `narrative`, and `status` properties. The class should have `__init__`, `brief`, `isMorning`, and `resolve` methods, as described in the `###` comments. The code should then call these methods as described in the script `###` comments. The script takes one argument, the full path file name of a crime report.

   Example input: C:/gispy/data/ch21/crimes.csv

   Example output:
   ```
   1251: Breaking & Entering, Pending
   Report that someone had pried steel door from hinges. Nothing
   was found to be missing.

   1251: Breaking & Entering, Resolved
   Report that someone had pried steel door from hinges. Nothing
   was found to be missing.

   Did incident 1313 occur in the morning? False
   1313: Larceny, Bummer
   Non-student reported book bag had been stolen.

   1320: Fire Alarm, Resolved
   Units responded to alarm caused by cooking.
   ```

6. **birds.py** Write a script that creates a `Bird` class. A `Bird` object should have `name`, `habitat`, and `weight` properties. The class should include `__init__`, `talk`, and `diet` methods. The `__init__` method should initialize the properties. The `talk` method should print `'Squawk!'`, if the `name` property has the value of `'Toucan'` and should print `'Tweet'` otherwise. The `diet` method should return (not print) `'fruit'` if the habitat is `'South America'`, `'bugs'` if the habitat is `'North America'`, and `'fish'` otherwise. Write test code that only runs when you run 'birds.py' as the main script. The test code should create a `Bird` instance, using a variable, named `bigBird`; Set its name to `'Condor'`, its habitat to `'Sesame Street'`, and its weight to `2000`. In the test code, call the `talk` method three times. Then call the `diet` method on `bigBird` and print the return results. When 'birds.py' is run as the main script, it will print:

   ```
   >>> Tweet
   Tweet
   Tweet
   Condors eat fish.
   ```

   Then restart your IDE and place sample script 'famousBirds.py' in the same directory as your script. The 'famousBirds.py' script will import your script.

Run it to check the printout when your script is not the main script. The output should appear as follows:

```
>>> Squawk!
Squawk!
Toucans eat fruit.
Sam gained an ounce. He weighs 1.9625 lbs
Tweet
Parrots eat fish.
```