**Object-oriented programming**

**Introduction**

As you have seen from the earliest code examples in this course, it is not compulsory to organise your code into classes when you program in Python. You can use functions by themselves, in what is called a *procedural* programming approach. However, while a procedural style can suffice for writing short, simple programs, an object-oriented programming (OOP) approach becomes more valuable the more your program grows in size and complexity.

The more data and functions comprise your code, the more important it is to arrange them into logical subgroups, making sure that data and functions which are related are grouped together and that data and functions which are not related don’t interfere with each other. Modular code is easier to understand and modify, and lends itself more to reuse – and code reuse is valuable because it reduces development time.

As a worst-case scenario, imagine a program with a hundred functions and a hundred separate global variables all in the same file. This would be a very difficult program to maintain. All the variables could potentially be modified by all the functions even if they shouldn’t be, and in order to pick unique names for all the variables, some of which might have a very similar purpose but be used by different functions, we would probably have to resort to poor naming practices. It would probably be easy to confuse these variables with each other, since it would be difficult to see which functions use which variables.

We could try to make this code more modular even without object orientation. We could group related variables together into aggregate data structures. In the past, some other languages, like C++, introduced a struct type which eventually became indistinguishable from a class, but which initially didn’t have any methods – only attributes. This allowed programmers to construct compound variables out of many individual variables, and was the first step towards object orientation. In Python, we often use dictionaries for ad-hoc grouping of related data.

We could also split up the functions and data into separate *namespaces* instead of having them all defined inside the same *global namespace*. This often coincides with splitting the code physically into multiple files. In Python we do this by splitting code up into *modules*.

The main additional advantage of object orientation, as we saw in the previous chapter, is that it combines data with the functions which act upon that data in a single structure. This makes it easy for us to find related parts of our code, since they are physically defined in close proximity to one another, and also makes it easier for us to write our code in such a way that the data inside each object is accessed as much as possible only through that object’s methods. We will discuss this principle, which we call *encapsulation*, in the next section.

Some people believe that OOP is a more intuitive programming style to learn, because people find it easy to reason about objects and relationships between them. OOP is thus sometimes considered to be a superior approach because it allows new programmers to become proficient more quickly.

**Basic OOP principles**

The most important principle of object orientation is *encapsulation*: the idea that data inside the object should only be accessed through a public *interface* – that is, the object’s methods.

The age function we saw in the previous chapter is a good example of this philosophy. If we want to use the data stored in an object to perform an action or calculate a derived value, we define a method associated with the object which does this. Then whenever we want to perform this action we call the method on the object. We consider it bad practice to retrieve the information from inside the object and write separate code to perform the action outside of the object.

Encapsulation is a good idea for several reasons:

* the functionality is defined *in one place* and not in multiple places.
* it is defined in a logical place – the place where the data is kept.
* data inside our object is not modified unexpectedly by external code in a completely different part of our program.
* when we use a method, we only need to know what result the method will produce – we don’t need to know details about the object’s internals in order to use it. We could switch to using another object which is completely different on the inside, and not have to change any code because both objects have the same interface.

We can say that the object “knows how” to do things with its own data, and it’s a bad idea for us to access its internals and do things with the data ourselves. If an object doesn’t have an interface method which does what we want to do, we should add a new method or update an existing one.

Some languages have features which allow us to enforce encapsulation strictly. In Java or C++, we can define access permissions on object attributes, and make it illegal for them to be accessed from outside the object’s methods. In Java it is also considered good practice to write setters and getters for all attributes, even if the getter simply retrieves the attribute and the setter just assigns it the value of the parameter which you pass in.

In Python, encapsulation is not enforced by the language, but there is a convention that we can use to indicate that a property is intended to be private and is not part of the object’s public interface: we begin its name with an underscore.

It is also customary to set and get simple attribute values directly, and only write setter and getter methods for values which require some kind of calculation. In the last chapter we learned how to use the property decorator to replace a simple attribute with a method without changing the object’s interface.

**Relationships between objects**

In the next section we will look at different ways that classes can be related to each other. In Python, there are two main types of relationships between classes: *composition* and *inheritance*.

**Composition**

Composition is a way of *aggregating* objects together by making some objects attributes of other objects. We saw in the previous chapter how we can make a datetime.date object an attribute of our Person object, and use it to store a person’s birthdate. We can say that a person *has a* birthdate – if we can express a relationship between two classes using the phrase *has-a*, it is a composition relationship.

Relationships like this can be one-to-one, one-to-many or many-to-many, and they can be unidirectional or bidirectional, depending on the specifics of the the roles which the objects fulfil.

According to some formal definitions the term *composition* implies that the two objects are quite strongly linked – one object can be thought of as *belonging* exclusively to the other object. If the owner object ceases to exist, the owned object will probably cease to exist as well. If the link between two objects is weaker, and neither object has exclusive ownership of the other, it can also be called *aggregation*.

Here are four classes which show several examples of aggregation and composition:

class Student:

def \_\_init\_\_(self, name, student\_number):

self.name = name

self.student\_number = student\_number

self.classes = []

def enrol(self, course\_running):

self.classes.append(course\_running)

course\_running.add\_student(self)

class Department:

def \_\_init\_\_(self, name, department\_code):

self.name = name

self.department\_code = department\_code

self.courses = {}

def add\_course(self, description, course\_code, credits):

self.courses[course\_code] = Course(description, course\_code, credits, self)

return self.courses[course\_code]

class Course:

def \_\_init\_\_(self, description, course\_code, credits, department):

self.description = description

self.course\_code = course\_code

self.credits = credits

self.department = department

self.department.add\_course(self)

self.runnings = []

def add\_running(self, year):

self.runnings.append(CourseRunning(self, year))

return self.runnings[-1]

class CourseRunning:

def \_\_init\_\_(self, course, year):

self.course = course

self.year = year

self.students = []

def add\_student(self, student):

self.students.append(student)

maths\_dept = Department("Mathematics and Applied Mathematics", "MAM")

mam1000w = maths\_dept.add\_course("Mathematics 1000", "MAM1000W", 1)

mam1000w\_2013 = mam1000w.add\_running(2013)

bob = Student("Bob", "Smith")

bob.enrol(mam1000w\_2013)

Why are there two classes which both describe a course? This is an example of the way that translation of real-life concepts into objects in your code may not always be as straightforward as it appears. Would it have made sense to have a single course object which has both description, code and department attributes and a list of students?

There are two distinct concepts, both of which can be called a “course”, that we need to represent: one is the theoretical *idea* of a course, which is offered by a department every year and always has the same name and code, and the other is the course as it is run *in a particular year*, each time with a different group of enrolled students. We have represented these two concepts by two separate classes which are linked to each other. Course is the theoretical description of a course, and CourseRunning is the concrete instance of a course.

We have defined several relationships between these classes:

* A student can be enrolled in several courses (CourseRunning objects), and a course (CourseRunning) can have multiple students enrolled in it in a particular year, so this is a many-to-many relationship. A student knows about all his or her courses, and a course has a record of all enrolled students, so this is a bidirectional relationship. These objects aren’t very strongly coupled – a student can exist independently of a course, and a course can exist independently of a student.
* A department offers multiple courses (Course objects), but in our implementation a course can only have a single department – this is a one-to-many relationship. It is also bidirectional. Furthermore, these objects are more strongly coupled – you can say that a department *owns* a course. The course cannot exist without the department.
* A similar relationship exists between a course and its “runnings”: it is also bidirectional, one-to-many and strongly coupled – it wouldn’t make sense for “MAM1000W run in 2013” to exist on its own in the absence of “MAM1000W”.

What words like “exist” and “owns” actually mean for our code can vary. An object which “owns” another object could be responsible for creating that object when it requires it and destroying it when it is no longer needed – but these words can also be used to describe a logical relationship between concepts which is not necessarily literally implemented in that way in the code.

**Exercise 1**

1. Briefly describe a possible collection of classes which can be used to represent a music collection (for example, inside a music player), focusing on how they would be related by composition. You should include classes for songs, artists, albums and playlists. Hint: write down the four class names, draw a line between each pair of classes which you think should have a relationship, and decide what kind of relationship would be the most appropriate.

For simplicity you can assume that any song or album has a single “artist” value (which could represent more than one person), but you should include compilation albums (which contain songs by a selection of different artists). The “artist” of a compilation album can be a special value like “Various Artists”. You can also assume that each song is associated with a single album, but that multiple copies of the same song (which are included in different albums) can exist.

1. Write a simple implementation of this model which clearly shows how the different classes are composed. Write some example code to show how you would use your classes to create an album and add all its songs to a playlist. Hint: if two objects are related to each other bidirectionally, you will have to decide how this link should be formed – one of the objects will have to be created before the other, so you can’t link them to each other in both directions simultaneously!

**Inheritance**

*Inheritance* is a way of arranging objects in a hierarchy from the most general to the most specific. An object which *inherits* from another object is considered to be a *subtype* of that object. As we saw in the previous chapter, all objects in Python inherit from object. We can say that a string, an integer or a Person instance *is an* object instance. When we can describe the relationship between two objects using the phrase *is-a*, that relationship is inheritance.

We also often say that a class is a *subclass* or *child class* of a class from which it inherits, or that the other class is its *superclass* or *parent class*. We can refer to the most generic class at the base of a hierarchy as a *base class*.

Inheritance can help us to represent objects which have some differences and some similarities in the way they work. We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Inheritance is also a way of reusing existing code easily. If we already have a class which does *almost* what we want, we can create a subclass in which we partially override some of its behaviour, or perhaps add some new functionality.

Here is a simple example of inheritance:

class Person:

def \_\_init\_\_(self, name, surname, number):

self.name = name

self.surname = surname

self.number = number

class Student(Person):

UNDERGRADUATE, POSTGRADUATE = range(2)

def \_\_init\_\_(self, student\_type, \*args, \*\*kwargs):

self.student\_type = student\_type

self.classes = []

super(Student, self).\_\_init\_\_(\*args, \*\*kwargs)

def enrol(self, course):

self.classes.append(course)

class StaffMember(Person):

PERMANENT, TEMPORARY = range(2)

def \_\_init\_\_(self, employment\_type, \*args, \*\*kwargs):

self.employment\_type = employment\_type

super(StaffMember, self).\_\_init\_\_(\*args, \*\*kwargs)

class Lecturer(StaffMember):

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.courses\_taught = []

super(Lecturer, self).\_\_init\_\_(\*args, \*\*kwargs)

def assign\_teaching(self, course):

self.courses\_taught.append(course)

jane = Student(Student.POSTGRADUATE, "Jane", "Smith", "SMTJNX045")

jane.enrol(a\_postgrad\_course)

bob = Lecturer(StaffMember.PERMANENT, "Bob", "Jones", "123456789")

bob.assign\_teaching(an\_undergrad\_course)

Our base class is Person, which represents any person associated with a university. We create a subclass to represent students and one to represent staff members, and then a subclass of StaffMember for people who teach courses (as opposed to staff members who have administrative positions.)

We represent both student numbers and staff numbers by a single attribute, number, which we define in the base class, because it makes sense for us to treat them as a unified form of identification for any person. We use different attributes for the kind of student (undergraduate or postgraduate) that someone is and whether a staff member is a permanent or a temporary employee, because these are different sets of options.

We have also added a method to Student for enrolling a student in a course, and a method to Lecturer for assigning a course to be taught by a lecturer.

The \_\_init\_\_ method of the base class initialises all the instance variables that are common to all subclasses. In each subclass we *override* the \_\_init\_\_ method so that we can use it to initialise that class’s attributes – but we want the parent class’s attributes to be initialised as well, so we need to call the parent’s \_\_init\_\_ method from ours. To find the right method, we use the super function – when we pass in the current class and object as parameters, it will return a proxy object with the correct \_\_init\_\_ method, which we can then call.

In each of our overridden \_\_init\_\_ methods we use those of the method’s parameters which are specific to our class inside the method, and then pass the remaining parameters to the parent class’s \_\_init\_\_ method. A common convention is to add the specific parameters for each successive subclass to the *beginning* of the parameter list, and define all the other parameters using \*args and \*\*kwargs – then the subclass doesn’t need to know the details about the parent class’s parameters. Because of this, if we add a new parameter to the superclass’s \_\_init\_\_, we will only need to add it to all the places where we create that class or one of its subclasses – we won’t also have to update all the child class definitions to include the new parameter.

**Exercise 2**

1. A very common use case for inheritance is the creation of a custom exception hierarchy. Because we use the class of an exception to determine whether it should be caught by a particular except block, it is useful for us to define custom classes for exceptions which we want to raise in our code. Using inheritance in our classes is useful because if an except block catches a particular exception class, it will also catch its child classes (because a child class *is* its parent class). That means that we can efficiently write except blocks which handle groups of related exceptions, just by arranging them in a logical hierarchy. Our exception classes should inherit from Python’s built-in exception classes. They often won’t need to contain any additional attributes or methods.

Write a simple program which loops over a list of user data (tuples containing a username, email and age) and adds each user to a directory if the user is at least 16 years old. You do not need to store the age. Write a simple exception hierarchy which defines a different exception for each of these error conditions:

* 1. the username is not unique
  2. the age is not a positive integer
  3. the user is under 16
  4. the email address is not valid (a simple check for a username, the @ symbol and a domain name is sufficient)

Raise these exceptions in your program where appropriate. Whenever an exception occurs, your program should move onto the next set of data in the list. Print a different error message for each different kind of exception.

Think about where else it would be a good idea to use a custom class, and what kind of collection type would be most appropriate for your directory.

You can consider an email address to be valid if it contains one @ symbol and has a non-empty username and domain name – you don’t need to check for valid characters. You can assume that the age is already an integer value.

**More about inheritance**

**Multiple inheritance**

The previous example might seem like a good way to represent students and staff members at first glance, but if we started to extend this system we would soon encounter some complications. At a real university, the divisions between staff and students and administrative and teaching staff are not always clear-cut. A student who tutors a course is also a kind of temporary staff member. A staff member can enrol in a course. A staff member can have *both* an administrative role in the department *and* a teaching position.

In Python it is possible for a class to inherit from multiple other classes. We could, for example, create a class called Tutor, which inherits from both Student and StaffMember. Multiple inheritance isn’t too difficult to understand if a class inherits from multiple classes which have completely different properties, but things get complicated if two parent classes implement the same method or attribute.

If classes B and C inherit from A and class D inherits from B and C, and both B and C have a method do\_something, which do\_something will D inherit? This ambiguity is known as the *diamond problem*, and different languages resolve it in different ways. In our Tutor class we would encounter this problem with the \_\_init\_\_ method.

Fortunately the super function knows how to deal gracefully with multiple inheritance. If we use it inside the Tutor class’s \_\_init\_\_ method, all of the parent classes’ \_\_init\_\_ methods should be called in a sensible order. We would then end up with a class which has all the attributes and methods found in both Student and StaffMember.

**Mix-ins**

If we use multiple inheritance, it is often a good idea for us to design our classes in a way which avoids the kind of ambiguity described above. One way of doing this is to split up optional functionality into *mix-ins*. A Mix-in is a class which is not intended to stand on its own – it exists to add extra functionality to another class through multiple inheritance. For example, let us try to rewrite the example above so that each set of related things that a person can do at a university is written as a mix-in:

class Person:

def \_\_init\_\_(self, name, surname, number):

self.name = name

self.surname = surname

self.number = number

class LearnerMixin:

def \_\_init\_\_(self):

self.classes = []

def enrol(self, course):

self.classes.append(course)

class TeacherMixin:

def \_\_init\_\_(self):

self.courses\_taught = []

def assign\_teaching(self, course):

self.courses\_taught.append(course)

class Tutor(Person, LearnerMixin, TeacherMixin):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(Tutor, self).\_\_init\_\_(\*args, \*\*kwargs)

jane = Tutor("Jane", "Smith", "SMTJNX045")

jane.enrol(a\_postgrad\_course)

jane.assign\_teaching(an\_undergrad\_course)

Now Tutor inherits from one “main” class, Person, and two mix-ins which are not related to Person. Each mix-in is responsible for providing a specific piece of optional functionality. Our mix-ins still have \_\_init\_\_ methods, because each one has to initialise a list of courses (we saw in the previous chapter that we can’t do this with a class attribute). Many mix-ins just provide additional methods and don’t initialise anything. This sometimes means that they depend on other properties which already exist in the class which inherits from them.

We could extend this example with more mix-ins which represent the ability to pay fees, the ability to get paid for services, and so on – we could then create a relatively flat hierarchy of classes for different kinds of people which inherit from Person and some number of mix-ins.

**Abstract classes and interfaces**

In some languages it is possible to create a class which can’t be instantiated. That means that we can’t use this class directly to create an object – we can only inherit from the class, and use the subclasses to create objects.

Why would we want to do this? Sometimes we want to specify a set of properties that an object needs to have in order to be suitable for some task – for example, we may have written a function which expects one of its parameters to be an object with certain methods that our function will need to use. We can create a class which serves as a *template* for suitable objects by defining a list of methods that these objects must implement. This class is not intended to be instantiated because all our method definitions are empty – all the *insides* of the methods must be implemented in a subclass.

The abstract class is thus an *interface* definition – some languages also have a type of structure called an interface, which is very similar. We say that a class *implements* an interface if it inherits from the class which specifies that interface.

In Python we can’t prevent anyone from instantiating a class, but we can create something similar to an abstract class by using NotImplementedError inside our method definitions. For example, here are some “abstract” classes which can be used as templates for shapes:

class Shape2D:

def area(self):

raise NotImplementedError()

class Shape3D:

def volume(self):

raise NotImplementedError()

Any two-dimensional shape has an area, and any three-dimensional shape has a volume. The formulae for working out area and volume differ depending on what shape we have, and objects for different shapes may have completely different attributes.

If an object inherits from 2DShape, it will gain that class’s default area method – but the default method raises an error which makes it clear to the user that a custom method must be defined in the child object:

class Square(Shape2D):

def \_\_init\_\_(self, width):

self.width = width

def area(self):

return self.width \*\* 2

**Exercise 3**

1. Write an “abstract” class, Box, and use it to define some methods which any box object should have: add, for adding any number of items to the box, empty, for taking all the items out of the box and returning them as a list, and count, for counting the items which are currently in the box. Write a simple Item class which has a name attribute and a value attribute – you can assume that all the items you will use will be Item objects. Now write two subclasses of Box which use different underlying collections to store items: ListBox should use a list, and DictBox should use a dict.
2. Write a function, repack\_boxes, which takes any number of boxes as parameters, gathers up all the items they contain, and redistributes them as evenly as possible over all the boxes. Order is unimportant. There are multiple ways of doing this. Test your code with a ListBox with 20 items, a ListBox with 9 items and a DictBox with 5 items. You should end up with two boxes with 11 items each, and one box with 12 items.

**Avoiding inheritance**

Inheritance can be a useful technique, but it can also be an unnecessary complication. As we have already discussed, multiple inheritance can cause a lot of ambiguity and confusion, and hierarchies which use multiple inheritance should be designed carefully to minimise this.

A deep hierarchy with many layers of subclasses may be difficult to read and understand. In our first inheritance example, to understand how the Lecturer class works we have to read through *three* different classes instead of one. If our classes are long and split into several different files, it can be hard to figure out which subclass is responsible for a particular piece of behaviour. You should avoid creating hierarchies which are more than one or two classes deep.

In some statically typed languages inheritance is very popular because it allows the programmer to work around some of the restrictions of static typing. If a lecturer and a student are both a kind of person, we can write a function which accepts a parameter of type Person and have it work on both lecturer and student objects because they both inherit from Person. This is known as *polymorphism*.

In Python inheritance is not compulsory for polymorphism, because Python is not statically typed. A function can work on both lecturer and student objects if they both have the appropriate attributes and methods even if these objects *don’t* share a parent class, and are completely unrelated. When you check parameters yourself, you are encouraged not to check an object’s type directly, but instead to check for the presence of the methods and attributes that your function needs to use – that way you are not forcing the parameter objects into an inheritance hierarchy when this is unnecessary.

**Replacing inheritance with composition**

Sometimes we can replace inheritance with composition and achieve a similar result – this approach is sometimes considered preferable. In the mix-in example, we split up the possible behaviours of a person into logical groups. Instead of implementing these sets of behaviours as mix-ins and having our class inherit from them, we can add them as *attributes* to the Person class:

class Learner:

def \_\_init\_\_(self):

self.classes = []

def enrol(self, course):

self.classes.append(course)

class Teacher:

def \_\_init\_\_(self):

self.courses\_taught = []

def assign\_teaching(self, course):

self.courses\_taught.append(course)

class Person:

def \_\_init\_\_(self, name, surname, number, learner=None, teacher=None):

self.name = name

self.surname = surname

self.number = number

self.learner = learner

self.teacher = teacher

jane = Person("Jane", "Smith", "SMTJNX045", Learner(), Teacher())

jane.learner.enrol(a\_postgrad\_course)

jane.teacher.assign\_teaching(an\_undergrad\_course)

Now instead of calling the enrol and assign\_teaching methods on our person object directly, we *delegate* to the object’s learner and teacher attributes.

**Exercise 4**

1. Rewrite the Person class in the last example, implementing additional methods called enrol and assign\_teaching which hide the delegation. These methods should raise an appropriate error message if the delegation cannot be performed because the corresponding attribute has not been set.

**Answers to exercises**

**Answer to exercise 1**

1. The following relationships should exist between the four classes:
   * a one-to-many relationship between albums and songs – this is likely to be bidirectional, since songs and albums are quite closely coupled.
   * a one-to-many relationship between artists and songs. This can be unidirectional or bidirectional. We don’t really need to store links to all of an artist’s songs on an artist object, since a reference to the artist from each song is enough for us to search our songs by artist, but if the music collection is very large it may be a good idea to cache this list.
   * a one-to-many relationship between artists and albums, which can be unidirectional or bidirectional for the same reasons.
   * a one-to-many relationship between playlists and songs – this is likely to be unidirectional, since it’s uncommon to keep track of all the playlists on which a particular song appears.
2. Here is an example program:
3. class Song:
4. def \_\_init\_\_(self, title, artist, album, track\_number):
5. self.title = title
6. self.artist = artist
7. self.album = album
8. self.track\_number = track\_number
9. artist.add\_song(self)
10. class Album:
11. def \_\_init\_\_(self, title, artist, year):
12. self.title = title
13. self.artist = artist
14. self.year = year
15. self.tracks = []
16. artist.add\_album(self)
17. def add\_track(self, title, artist=None):
18. if artist is None:
19. artist = self.artist
20. track\_number = len(self.tracks)
21. song = Song(title, artist, self, track\_number)
22. self.tracks.append(song)
23. class Artist:
24. def \_\_init\_\_(self, name):
25. self.name = name
26. self.albums = []
27. self.songs = []
28. def add\_album(self, album):
29. self.albums.append(album)
30. def add\_song(self, song):
31. self.songs.append(song)
32. class Playlist:
33. def \_\_init\_\_(self, name):
34. self.name = name
35. self.songs = []
36. def add\_song(self, song):
37. self.songs.append(song)
38. band = Artist("Bob's Awesome Band")
39. album = Album("Bob's First Single", band, 2013)
40. album.add\_track("A Ballad about Cheese")
41. album.add\_track("A Ballad about Cheese (dance remix)")
42. album.add\_track("A Third Song to Use Up the Rest of the Space")
43. playlist = Playlist("My Favourite Songs")
44. for song in album.tracks:
45. playlist.add\_song(song)

**Answer to exercise 2**

1. Here is an example program:
2. # Exceptions
3. class DuplicateUsernameError(Exception):
4. pass
5. class InvalidAgeError(Exception):
6. pass
7. class UnderageError(Exception):
8. pass
9. class InvalidEmailError(Exception):
10. pass
11. # A class for a user's data
12. class User:
13. def \_\_init\_\_(self, username, email):
14. self.username = username
15. self.email = email
16. example\_list = [
17. ("jane", "jane@example.com", 21),
18. ("bob", "bob@example", 19),
19. ("jane", "jane2@example.com", 25),
20. ("steve", "steve@somewhere", 15),
21. ("joe", "joe", 23),
22. ("anna", "anna@example.com", -3),
23. ]
24. directory = {}
25. for username, email, age in example\_list:
26. try:
27. if username in directory:
28. raise DuplicateUsernameError()
29. if age < 0:
30. raise InvalidAgeError()
31. if age < 16:
32. raise UnderageError()
33. email\_parts = email.split('@')
34. if len(email\_parts) != 2 or not email\_parts[0] or not email\_parts[1]:
35. raise InvalidEmailError()
36. except DuplicateUsernameError:
37. print("Username '%s' is in use." % username)
38. except InvalidAgeError:
39. print("Invalid age: %d" % age)
40. except UnderageError:
41. print("User %s is underage." % username)
42. except InvalidEmailError:
43. print("'%s' is not a valid email address." % email)
44. else:
45. directory[username] = User(username, email)

**Answer to exercise 3**

1. Here is an example program:
2. class Box:
3. def add(self, \*items):
4. raise NotImplementedError()
5. def empty(self):
6. raise NotImplementedError()
7. def count(self):
8. raise NotImplementedError()
9. class Item:
10. def \_\_init\_\_(self, name, value):
11. self.name = name
12. self.value = value
13. class ListBox(Box):
14. def \_\_init\_\_(self):
15. self.\_items = []
16. def add(self, \*items):
17. self.\_items.extend(items)
18. def empty(self):
19. items = self.\_items
20. self.\_items = []
21. return items
22. def count(self):
23. return len(self.\_items)
24. class DictBox(Box):
25. def \_\_init\_\_(self):
26. self.\_items = {}
27. def add(self, \*items):
28. self.\_items.update(dict((i.name, i) for i in items))
29. def empty(self):
30. items = list(self.\_items.values())
31. self.\_items = {}
32. return items
33. def count(self):
34. return len(self.\_items)
35. Here is an example program:
36. def repack\_boxes(\*boxes):
37. items = []
38. for box in boxes:
39. items.extend(box.empty())
40. while items:
41. for box in boxes:
42. try:
43. box.add(items.pop())
44. except IndexError:
45. break
46. box1 = ListBox()
47. box1.add(Item(str(i), i) for i in range(20))
48. box2 = ListBox()
49. box2.add(Item(str(i), i) for i in range(9))
50. box1 = DictBox()
51. box1.add(Item(str(i), i) for i in range(5))
52. repack\_boxes(box1, box2, box3)
53. print(box1.count())
54. print(box2.count())
55. print(box3.count())

**Answer to exercise 4**

1. Here is an example program:
2. class Person:
3. def \_\_init\_\_(self, name, surname, number, learner=None, teacher=None):
4. self.name = name
5. self.surname = surname
6. self.number = number
7. self.learner = learner
8. self.teacher = teacher
9. def enrol(self, course):
10. if not hasattr(self, "learner"):
11. raise NotImplementedError()
12. self.learner.enrol(course)
13. def assign\_teaching(self, course):
14. if not hasattr(self, "teacher"):
15. raise NotImplementedError()

self.teacher.assign\_teaching(course)