UNIT 4

Classes and methods

Object-oriented features

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming. It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

• Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.

• Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact. For example, the Time class corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part, they provide an alternative syntax for things we have already done, but in many cases, the alternative is more concise and more accurately conveys the structure of the program. For example, in the Time program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one Time object as an argument. This observation is the motivation for methods. We have already seen some methods, such as keys and values, which were invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Methods are just like functions, with two differences:

• Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

• The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

Optional arguments

We have seen built-in functions that take a variable number of arguments. For example, string.find can take two, three, or four arguments. It is possible to write user-defined functions with optional argument lists. For example, we can upgrade our own version of find to do the same thing as string.find.

```
def find(str, ch):
index = 0
```

```
while index < len(str):
if str[index] == ch:
return index
index = index + 1
return -1
```
This is the new and improved version:
```
def find(str, ch, start=0):
index = start
while index < len(str):
if str[index] == ch:
return index
index = index + 1
return -1
```
The third parameter, start, is optional because a default value, 0, is provided. If we invoke find with only two arguments, it uses the default value and starts from the beginning of the string:
```
>>> find("apple", "p")
1
```
If we provide a third argument, it overrides the default:
```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```
As an exercise, add a fourth parameter, end, *that specifies where to stop looking. Warning: This exercise is a bit tricky. The default value of* end *should be* len(str)*, but that doesn't work. The default values are evaluated when the function is defined, not when it is called. When* find *is defined,* str *doesn't exist yet, so you can't find its length.*

The initialization method

The initialization method is a special method that is invoked when an object is created. The name of this method is init (two underscore characters, followed by init, and then two more underscores). An initialization method for the Time class looks like this:
```
class Time:
def __init__(self, hours=0, minutes=0, seconds=0):
self.hours = hours
self.minutes = minutes
self.seconds = seconds
```
There is no conflict between the attribute self.hours and the parameter hours. Dot notation specifies which variable we are referring to.
When we invoke the Time constructor, the arguments we provide are passed along to init:
```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
9:14:30
```
Because the arguments are optional, we can omit them:
```
>>> currentTime = Time()
```

>>> currentTime.printTime()
0:0:0
Or provide only the first:
>>> currentTime = Time (9)
>>> currentTime.printTime()
9:0:0
Or the first two:
>>> currentTime = Time (9, 14)
>>> currentTime.printTime()
9:14:0
Finally, we can make assignments to a subset of the parameters by naming them explicitly:
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
9:0:30

Operator overloading

Some languages make it possible to change the definition of the built-in operators when they are applied to user-defined types. This feature is called operator overloading. It is especially useful when defining new mathematical types. For example, to override the addition operator +, we provide a method named add :
class Point:
# previously defined methods here...
def __add__(self, other):
return Point(self.x + other.x, self.y + other.y)
As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named other to distinguish it from self. To add two Points, we create and return a new Point that contains the sum of the x coordinates and the sum of the y coordinates. Now, when we apply the + operator to Point objects, Python invokes add :
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> p3 = p1 + p2
>>> print p3
(8, 11)
The expression p1 + p2 is equivalent to p1. add (p2), but obviously more elegant.

As an exercise, add a method sub (self, other) *that overloads the subtraction operator, and try it out.*

There are several ways to override the behavior of the multiplication operator: by defining a method named mul , or rmul , or both. If the left operand of * is a Point, Python invokes mul , which assumes that the other operand is also a Point. It computes the dot product of the two points, defined according to the rules of linear algebra:

def __mul__(self, other):
return self.x * other.x + self.y * other.y

If the left operand of * is a primitive type and the right operand is a Point, Python invokes rmul , which performs scalar multiplication:
def __rmul__(self, other):
return Point(other * self.x, other * self.y)
The result is a new Point whose coordinates are a multiple of the original coordinates. If other is a type that cannot be multiplied by a floating-point number, then rmul will yield an error.

This example demonstrates both kinds of multiplication:
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
What happens if we try to evaluate p2 * 2? Since the first operand is a Point, Python invokes mul with 2 as the second argument. Inside mul , the program tries to access the x coordinate of other, which fails because an integer has no attributes:
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

Polymorphism

Most of the methods we have written only work for a specific type. When you create a new object, you write methods that operate on that type. But there are certain operations that you will want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, you can write functions that work on any of those types. For example, the multadd operation (which is common in linear algebra) takes three arguments; it multiplies the first two and then adds the third. We can write it in Python like this:

def multadd (x, y, z):
return x * y + z
This method will work for any values of x and y that can be multiplied and for
any value of z that can be added to the product.
We can invoke it with numeric values:
>>> multadd (3, 2, 1)
7
Or with Points:
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print multadd (2, p1, p2)
(11, 15)
>>> print multadd (p1, p2, 1)
44

In the first case, the Point is multiplied by a scalar and then added to another Point. In the second case, the dot product yields a numeric value, so the third argument also has to be a numeric value. A function like this that can take arguments with different types is called polymorphic. As another example, consider the method frontAndBack, which prints a list twice, forward and backward:

```
def frontAndBack(front):
import copy
back = copy.copy(front)
back.reverse()
print str(front) + str(back)
```

Because the reverse method is a modifier, we make a copy of the list before reversing it. That way, this method doesn't modify the list it gets as an argument. Here's an example that applies frontAndBack to a list:

```
>>> myList = [1, 2, 3, 4]
>>> frontAndBack(myList)
```

Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a Point. To determine whether a function can be applied to a new type, we apply the fundamental rule of polymorphism: If all of the operations inside the function can be applied to the type, the function can be applied to the type. The operations in the method include copy, reverse, and print. copy works on any object, and we have already written a str method for Points, so all we need is a reverse method in the Point class:

```
def reverse(self):
self.x , self.y = self.y, self.x
```

Then we can pass Points to frontAndBack:

```
>>> p = Point(3, 4)
>>> frontAndBack(p)
(3, 4)(4, 3)
```

The best kind of polymorphism is the unintentional kind, where you discover that a function you have already written can be applied to a type for which you never planned.

Inheritance

The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class. The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called "inheritance" because the new class inherits all of the methods of the existing class. Extending this metaphor, the existing class is sometimes called the parent class. The new class may be called the child class or sometimes "subclass." Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize

the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand. On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good. In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

Extending built-ins

We discussed briefly in *Chapter 3* how built-in data types can be extended using inheritance. Now, we'll go into more detail as to when we would want to do that.

When we have a built-in container object that we want to add functionality to, we have two options. We can either create a new object, which holds that container as an attribute (composition), or we can subclass the built-in object and add or adapt methods on it to do what we want (inheritance).

Composition is usually the best alternative if all we want to do is use the container to store some objects using that container's features. That way, it's easy to pass that data structure into other methods and they will know how to interact with it. But we need to use inheritance if we want to change the way the container actually works. For example, if we want to ensure every item in a list is a string with exactly five characters, we need to extend list and override the append() method to raise an exception for invalid input. We'd also have to override __setitem__(self, index, value), a special method on lists that is called whenever we use the x[index] = "value" syntax.

That's right, all that special non-object-oriented looking syntax we've been looking at for accessing lists, dictionary keys, looping over containers, and similar tasks is actually "syntactic sugar" that maps to an object-oriented paradigm underneath. We might ask the Python designers why they did this, when common perception suggests that object-oriented programming is **always** better. That question is easy to answer. In the following hypothetical examples, which is easier to read, as a programmer? Which requires less typing?:

```
c = a + b
c = a.add(b)
l[0] = 5
 l.setitem(0, 5)
d[key] = value
d.setitem(key, value)
for x in alist:
#do something with x
it = alist.iterator()
```

while it.has_next():
x = it.next()
#do something with x

The highlighted sections show what object-oriented code might look like (in practice, these methods actually exist as special double-underscore methods on associated objects). Python programmers agree that the non-object-oriented syntax is easier to read and to write. Non-Python programmers say that syntax like this means Python is not object-oriented. That, however, is hogwash. All of the above Python syntaxes map to object-oriented methods underneath the hood. These methods have special names (with double-underscores before and after) to remind us that there is a better syntax out there. However, we now have the means to override these behaviors. For example, we can make a special integer that always returns 0 when we add two of them together:

class SillyInt(int):
def __add__(self, num):
return 0

This is a very strange thing to do, granted, but it illustrates perfectly the object-oriented principles in action. And now we have an argument when people tell us Python isn't truly object-oriented. It's just object-oriented that has been made easy to work with. Check out the above class in action:

>>> **a = SillyInt(1)**

>>> **b = SillyInt(2)**

>>> **a + b**

0

The awesome thing about the __add__ method is that we can add it to any class we write, and if we use the + operator on instances of that class, it will be called. This is how string, tuple, and list concatenation works.

This is true of all the special methods. If we want to use x in myobj syntax, we can override __contains__. If we want to use myobj[i] = value syntax, we implement __setitem__ and if we want to use something = myobj[i], we implement __getitem__.
There are thirty-three of these special methods on the list class. We can use the dir function to see all of them:

>>> **dir(list)**

['__add__', '__class__', '__contains__', '__delattr__','__delitem__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'

Further, if we want any additional information on how any of these methods works, we can use the help function:

>>> help(list.__add__)

Help on wrapper_descriptor:

__add__(...)

x.__add__(y) <==> x+y

The plus operator on lists concatenates two lists. We don't have room to discuss all of the available special functions in this book, but you are now able to explore all this functionality with dir and help. The official online Python reference (http://docs.python.org/) has plenty of useful information as well. Focus, especially, on the abstract base classes discussed in the collections module.

So to get back to the earlier point about when we would want to use composition versus inheritance: if we need to somehow change any of the methods on the class, including the special methods we definitely need to use inheritance. If we used composition, we could write methods that do the validation or alterations and ask the caller to use those methods, but there is nothing stopping them from accessing the property directly (no private members, remember?). They could insert an item into our list that does not have five characters, and that might confuse other methods in the list.

Often, the need to extend a built-in data type is an indication that we're using the wrong sort of data type. It is not always the case, but if you're suddenly looking to extend a built-in, carefully consider whether or not a different data structure would be more suitable.
As a last example, let's consider what it takes to create a dictionary that remembers the order in which keys were inserted. One way (likely not the best way) to do this is to keep an ordered list of keys that is stored in a specially derived subclass of dict. Then we can override the methods keys, values, __iter__, and items to return everything in order. Of course, we'll also have to override __setitem__ and setdefault to keep our list up to date. There are likely to be a few other methods in the output of dir(dict) that need overriding to keep the list and dictionary consistent (clear and __delitem__ come to mind, to track when items are removed), but we won't worry about them for this example.

So we'll be extending dict and adding a list of ordered keys. Trivial enough, but where do we create the actual list? We could include it in the __init__ method, which would work just fine, but we have no guarantees that any subclass will call that initializer. Remember the __new__ method we discussed in *Chapter 2*? I said it was generally only useful in very special cases. This is one of those special cases. We know __new__ will be called exactly once, and we can create a list on the new instance that will always be available to our class. With that in mind, here is our entire sorted dictionary:

```
from collections import KeysView, ItemsView, ValuesView
class DictSorted(dict):
def __new__(*args, **kwargs):
new_dict = dict.__new__(*args, **kwargs)
new_dict.ordered_keys = []
return new_dict
def __setitem__(self, key, value):
'''self[key] = value syntax'''
```

```
    if key not in self.ordered_keys:
    self.ordered_keys.append(key)
    super().__setitem__(key, value)
    def setdefault(self, key, value):
    if key not in self.ordered_keys:
    self.ordered_keys.append(key)
    return super().setdefault(key, value)
    def keys(self):
    return KeysView(self)
    def values(self):
    return ValuesView(self)
def items(self):
    return ItemsView(self)
    def __iter__(self):
    '''for x in self syntax'''
    return self.ordered_keys.__iter__()
```

The __new__ method simply creates a new dictionary and then puts an empty list on that object. We don't override __init__, as the default implementation works (actually, this is only true if we initialize an empty DictSorted object, which is standard behavior. If we want to support other variations of the dict constructor, which accept dictionaries or lists of tuples, we'd need to fix __init__ to also update our ordered_keys). The two methods for setting items are very similar; they both update the list of keys, but only if the item hasn't been added before. We don't want duplicates in the list, but we can't use a set here; it's unordered!

The keys, items, and values methods all return views onto the dictionary. The collections library provides three read-only View objects onto the dictionary; they use the __iter__ method to loop over the keys, and then use __getitem__ (which we didn't need to override) to retrieve the values. So we only need to define our custom __iter__ method to make these three views work. You would think the superclass would do to create these views properly using polymorphism, but if we don't override these three methods, they don't return properly ordered views.

Finally, the __iter__ method is the really special one; it ensures that if we loop over the dictionary's keys (using for...in syntax), it will return the values in the correct order. It simply does this by returning the __iter__ of the ordered_keys list, which returns the same iterator object that would be used if we used for...in on the list instead. Since ordered_keys is a list of all available keys (due to the way we overrode other methods), this is the correct iterator object for the dictionary as well.

Let's look at a few of these methods in action, compared to a normal dictionary:

```
>>> ds = DictSorted()
>>> d = {}
>>> ds['a'] = 1
>>> ds['b'] = 2
>>> ds.setdefault('c', 3)
```

```
3
>>> d['a'] = 1
>>> d['b'] = 2
>>> d.setdefault('c', 3)
3
>>> for k,v in ds.items():
... print(k,v)
...
a 1
b 2
c 3
>>> for k,v in d.items():
... print(k,v)
...
a 1
c 3
b 2
```

Multiple inheritances

Multiple inheritances is a touchy subject. In principle, it's very simple: a subclass that inherits from more than one parent class is able to access functionality from both of them. In practice, this is much less useful than it sounds and many expert programmers recommend against using it. So we'll start with a warning:

As a rule of thumb, if you think you need multiple inheritances, you're probably wrong, but if you know you need it, you're probably right.

The simplest and most useful form of multiple inheritance is called a **mixin**. A mixin is generally a superclass that is not meant to exist on its own, but is meant to be inherited by some other class to provide extra functionality. For example, let's say we wanted to add functionality to our Contact class that allows sending an e-mail to self.email. Sending e-mail is a common task that we might want to use on many other classes. So we can write a simple mixin class to do the e-mailing for us:

```
class MailSender:
def send_mail(self, message):
print("Sending mail to " + self.email)
# Add e-mail logic here
```

For brevity, we won't include the actual e-mail logic here; if you're interested in studying how it's done, see the smtplib module in the Python standard library.

This class doesn't do anything special (in fact, it can barely function as a stand-alone class), but it does allow us to define a new class that is both a Contact and a MailSender, using multiple inheritance:

```
class EmailableContact(Contact, MailSender):
pass
```

The syntax for multiple inheritance looks like a parameter list in the class definition. Instead of including one base class inside the parenthesis, we include two (or more), separated by a comma. We can test this new hybrid to see the mixin at work:

```
>>> e = EmailableContact("John Smith", "jsmith@example.net")
>>> Contact.all_contacts
[<__main__.EmailableContact object at 0xb7205fac>]
>>> e.send_mail("Hello, test e-mail here")
Sending mail to jsmith@example.net
```

The Contact initializer is still adding the new contact to the all_contacts list, and the mixin is able to send mail to self.email so we know everything is working.

That wasn't so hard, and you're probably wondering what the dire warnings about multiple inheritance are. We'll get into the complexities in a minute, but let's consider what options we had, other than using a mixin here:

> We could have used single inheritance and added the send_mail function to the subclass. The disadvantage here is that the e-mail functionality then has to be duplicated for any other classes that need e-mail.
>
> We can create a stand-alone Python function for sending mail, and just call that, with the correct e-mail address supplied as a parameter, when e-mail needs to be sent.

We could **monkey-patch** the Contact class to have a send_mail method after the class has been created. This is done by defining a function that accepts the self argument, and setting it as an attribute on an existing class.

Multiple inheritance works all right when mixing methods from different classes, but it gets very messy when we have to work with calling methods on the superclass. Why? Because there are multiple superclasses. How do we know which one to call? How do we know what order to call them in?

Let's explore these questions by adding a home address to our Friend class. What are some ways we could do this? An address is a collection of strings representing the street, city, country, and other related details of the contact. We could pass each of these strings as parameters into the Friend class's __init__ method. We could also store these strings in a tuple or dictionary and pass them into __init__ as a single argument. This is probably the best course of action if there is no additional functionality that needs to be added to the address.

Another option would be to create a new Address class to hold those strings together, and then pass an instance of this class into the __init__ in our Friend class. The advantage of this solution is that we can add behavior (say, a method to give directions to that address or to print a map) to

the data instead of just storing it statically. This would be utilizing composition, the "has a" relationship we discussed in *Chapter 1*. Composition is a perfectly viable solution to this problem and allows us to reuse Address classes in other entities such as buildings, businesses, or organizations.

However, inheritance is also a viable solution, and that's what we want to explore, so let's add a new class that holds an address. We'll call this new class AddressHolder instead of Address, because inheritance defines an "is a" relationship. It is not correct to say a Friend is an Address, but since a friend can have an Address, we can argue that a Friend is an AddressHolder. Later, we could create other entities (companies, buildings) that also hold addresses. Here's our AddressHolder class:

```
class AddressHolder:
def __init__(self, street, city, state, code):
self.street = street
self.city = city
self.state = state
self.code = code
```

Very simple; we just t take all the data and toss it into instance variables upon initialization.

The diamond problem

But how can we use this in our existing Friend class, which is already inheriting from Contact? Multiple inheritance, of course. The tricky part is that we now have two parent __init__ methods that both need to be initialized. And they need to be initialized with different arguments. How do we do that? Well, we could start with the naïve approach:

```
class Friend(Contact, AddressHolder):
def __init__(self, name, email, phone,
street, city, state, code):
Contact.__init__(self, name, email)
AddressHolder.__init__(
self, street, city, state, code)
self.phone = phone
```

In this example, we directly call the __init__ function on each of the superclasses and explicitly pass the self argument. This example technically works; we can access the different variables directly on the class. But there are a few problems.

First, it is possible for a superclass to go uninitialized if we neglect to explicitly call the initializer. This is not bad in this example, but it could cause bad program crashes in common scenarios. Imagine, for example, trying to insert data into a database that has not been connected to.

Second, and more sinister, is the possibility of a superclass being called multiple times, because of the organization of the class hierarchy. Look at this inheritance diagram:
The __init__ method from the Friend class first calls __init__ on Contact which implicitly initializes the object superclass (remember, all classes derive from object). Friend then calls

__init__ on AddressHolder, which implicitly initializes the object superclass... *again*. The parent class has been set up twice. In this case, that's relatively harmless, but in some situations, it could spell disaster. Imagine trying to connect to a database twice for every request! The base class should only be called once. Once, yes, but when? Do we call Friend then Contact then Object then AddressHolder? Or Friend then Contact then AddressHolder then Object?

Let's look at a second contrived example that illustrates this problem more clearly. Here we have a base class that has a method named call_me. Two subclasses override that method, and then another subclass extends both of these using multiple inheritance. This is called diamond inheritance because of the diamond shape of the class diagram:
Diamonds are what makes multiple inheritance tricky. Technically, all multiple inheritance in Python 3 is diamond inheritance, because all classes inherit from object. The previous diagram, using object.__init__ is also such a diamond.

Converting this diagram to code, this example shows when the methods are called:

```
class BaseClass:
num_base_calls = 0
def call_me(self):
print("Calling method on Base Class")
self.num_base_calls += 1
class LeftSubclass(BaseClass):
num_left_calls = 0
def call_me(self):
BaseClass.call_me(self)
print("Calling method on Left Subclass")
self.num_left_calls += 1
```

This example simply ensures each overridden call_me method directly calls the parent method with the same name. Each time it is called, it lets us know by printing the information to the screen, and updates a static variable on the class to show how many times it has been called. If we instantiate one Subclass object and call the method on it once, we get this output:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
>>> print(s.num_sub_calls, s.num_left_calls, s.num_right_calls,
s.num_base_calls)
1 1 1 2
>>>
```

The base class's call_me method has been called twice. This isn't expected behavior and can lead to some very difficult bugs if that method is doing actual work—like depositing into a bank account twice.

The thing to keep in mind with multiple inheritance is that we only want to call the "next" method in the class hierarchy, not the "parent" method. In fact, that next method may not be on a parent or ancestor of the current class. The super keyword comes to our rescue once again. Indeed, super was originally developed to make complicated forms of multiple inheritance possible. Here is the same code written using super:

```
class BaseClass:
num_base_calls = 0
def call_me(self):
print("Calling method on Base Class")
self.num_base_calls += 1
class LeftSubclass(BaseClass):
num_left_calls = 0
def call_me(self):
super().call_me()
print("Calling method on Left Subclass")
self.num_left_calls += 1
class RightSubclass(BaseClass):
num_right_calls = 0
def call_me(self):
super().call_me()
print("Calling method on Right Subclass")
self.num_right_calls += 1
class Subclass(LeftSubclass, RightSubclass):
num_sub_calls = 0
def call_me(self):
super().call_me()
print("Calling method on Subclass")
self.num_sub_calls += 1
```

The change is pretty minor; we simply replaced the naïve direct calls with calls to super(). This is simple enough, but look at the difference when we execute it:

```
>>> s = Subclass()
>>> s.call_me()
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
>>> print(s.num_sub_calls, s.num_left_calls, s.num_right_calls,
```

s.num_base_calls)

1 1 1 1

Looks good, our base method is only being called once. But what is super() actually doing here? Since the print statements are executed after the super calls, the printed output is in the order each method is actually executed. Let's look at the output from back to front to see who is calling what.

First call_me of Subclass calls super().call_me(), which happens to refer to LeftSubclass.call_me(). LeftSubclass.call_me() then calls super().call_me(), but in this case, super() is referring to RightSubclass.call_me(). Pay particular attention to this; the super call is **not** calling the method on the superclass of LeftSubclass (which is BaseClass), it is calling RightSubclass, even though it is not a parent of LeftSubclass! This is the **next** method, not the parent method. RightSubclass then calls BaseClass and the super calls have ensured each method in the class hierarchy is executed once.

Different sets of arguments

Can you see how this is going to make things complicated when we return to our Friend multiple inheritance example? In the __init__ method for Friend, we were originally calling __init__ for both parent classes, *with different sets of arguments*:

Contact.__init__(self, name, email)
AddressHolder.__init__(self, street, city, state, code)

How can we convert this to using super? We don't necessarily know which class super is going to try to initialize first. Even if we did, we need a way to pass the "extra" arguments so that subsequent calls to super, on other subclasses, have the right arguments.

Specifically, if the first call to super passes the name and email arguments to Contact.__init__, and Contact.__init__ then calls super, it needs to be able to pass the address related arguments to the "next" method, which is AddressHolder.__init__.

This is a problem whenever we want to call superclass methods with the same name, but different sets of arguments. Most often, the only time you would want to call a superclass with a completely different set of arguments is in __init__, as we're doing here. Even with regular methods, though, we may want to add optional parameters that only make sense to one subclass or a set of subclasses.
Sadly, the only way to solve this problem is to plan for it from the beginning. We have to design our base class parameter lists so that they accept keyword arguments for any argument that is not required by every subclass implementation. We also have to ensure the method accepts arguments it doesn't expect and pass those on in its super call, in case they are necessary to later methods in the inheritance order.

Python's function parameter syntax provides all the tools we need to do this, but it makes the overall code cumbersome. Have a look at the proper version of the Friend multiple inheritance code:

```
class Contact:
all_contacts = []
```

```
def __init__(self, name='', email='', **kwargs):
super().__init__(**kwargs)
self.name = name
self.email = email
self.all_contacts.append(self)
class AddressHolder:
def __init__(self, street='', city='', state='', code='',
**kwargs):
super().__init__(**kwargs)
self.street = street
self.city = city
self.state = state
self.code = code
class Friend(Contact, AddressHolder):
def __init__(self, phone='', **kwargs):
super().__init__(**kwargs)
self.phone = phone
```

We've changed all arguments to keyword arguments by giving them an empty string as a default value. We've also ensured that a **kwargs parameter is included to capture any additional parameters that our particular method doesn't know what to do with. It passes these parameters up to the next class with the super call.