> *The Greek shall inherit the Earth*

Inheritance

Introduction

We are going to look at a class that represents a simple television. This class will have a channel and a power state. We will have methods to turn the TV on and off and change the channel up and down. The following is the class definition.

```
class TV(object) :
    """Representation of a simple television."""

    def __init__(self) :
        self._channel = 1
        self._power = False

    def turn_on(self) :
        """Turns the TV on."""
        self._power = True

    def turn_off(self) :
        """Turns the TV off."""
        self._power = False

    def channel_up(self) :
        """Changes the channel up by 1.
           If the channel goes above 100 then loops back to 1.
        """
        if self._power :
            self._channel += 1
            if self._channel > 100 :
                self._channel = 1

    def channel_down(self) :
        """Changes the channel down by 1.
           If the channel goes below 1 then loops back to 100.
        """
        if self._power :
            self._channel -= 1
            if self._channel < 1 :
                self._channel = 100

    def __str__(self) :
        if self._power :
            return "I'm a TV on channel {0}".format(self._channel)
        else :
            return "I'm a TV that is turned off"
```

Now if we save our file as tv.py we can have a look at the functionality of an object of this class.

```
>>> tv = TV()
>>> str(tv)
"I'm a TV that is turned off"
>>> tv.turn_on()
>>> str(tv)
"I'm a TV on channel 1"
>>> tv.channel_down()
>>> str(tv)
"I'm a TV on channel 100"
>>> for i in range(20) :
    tv.channel_up()


>>> str(tv)
"I'm a TV on channel 20"
```

It is clear that this TV does not have much functionality. As a TV it is also a little annoying to have to keep hitting the change channel to get to the channel of interest. What if we wanted another "Deluxe TV" class that could 'jump' straight to a channel? One way to do this is to copy and modify our code. A better way to do this is to write a class that includes the same functionality as our existing TV class but with a little extra. Object-oriented programming does this easily through the use of **inheritance**. Inheritance is the concept where one class, the **subclass**, 'inherits' properties and methods of another existing class, the **superclass**. This is another example of where we can reuse existing code without having to duplicate it.

Let's now write a new class that represents a simple Deluxe TV where it is possible to 'jump' to a selected channel. Here is the class definition.

```
class DeluxeTV(TV) :
    """Representation of a Deluxe TV where the channel can be set
       without using up and down.
    """

    def set_channel(self, channel) :
        """Sets the TV channel to the indicated 'channel' if the TV is on.
           If 'channel' is invalid an error message is output.
        """
        if self._power :
            if 1 < channel < 100 :
                self._channel = channel
            else :
                print("{0} is not a valid channel".format(channel))
```

First notice in the line `class DeluxeTV(TV)` we have `TV` instead of `object` in the brackets. This says that `DeluxeTV` inherits from TV. In fact every other class that we have written so far has inherited from the `object` class. The `object` class is the type which all other classes should inherit from, either directly or indirectly (as in this case). The next thing to notice is that `DeluxeTV` does not have an `__init__` method. As we are not adding or changing any data structures of our TV class we do not need an `__init__` method as it uses the inherited method from `TV`. Our new class simply has the one method definition that allows us to set the channel.

We can now add this class definition after our `TV` class definition and save a deluxe_tv.py file and have a look at the functionality of our new TV.

```
>>> tv = DeluxeTV()
>>> str(tv)
"I'm a TV that is turned off"
>>> tv.turn_on()
>>> str(tv)
"I'm a TV on channel 1"
>>> tv.channel_up()
>>> str(tv)
"I'm a TV on channel 2"
>>> tv.set_channel(42)
>>> str(tv)
"I'm a TV on channel 42"
>>> tv.set_channel(200)
"200 is not a valid channel"
>>> str(tv)
"I'm a TV on channel 42"
```

The first few examples show that we have not changed the other functionality of the TV. This is because our `DeluxeTV` has all the methods of our `TV`. The last two examples shows our new functionality and that we can now 'jump' to any valid channel.

Overriding Methods

Let's continue with our TV classes and write a new, Super Deluxe TV that extends the functionality of a `DeluxeTV`, and can store favourite channels. We will also make the channel up and down methods move through the favourite channels only. In this case, not only will we need to add new methods to store and remove favourite channels, but we will need to make different `channel_up` and `channel_down` methods.

We can do this by redefining `channel_up` and `channel_down`. Redefining an existing method of a superclass is known as **overriding** the method. This is useful as not all classes want to have the same functionality for each method of its superclass. When we call `tv.channel_up()` and `tv.channel_down()` on a `SuperDeluxeTV`, the new definitions will be executed, and the old definitions will be ignored.

```
class SuperDeluxeTV(DeluxeTV) :
    """Representation of a Super Delux TV where channels can be saved
       to favourites. Channel up and down goes through the favourites.
    """

    def __init__(self) :
        super().__init__()
        self._favourites = []          # All favourite TV channels.

    def store(self) :
        """Stores the current channel as one of the favourites."""
        if self._channel not in self._favourites and self._power :
            self._favourites.append(self._channel)
            self._favourites.sort()
```

```
    def remove(self) :
        """Removes the current channel so that it is no longer a favourite."""
        if self._power :
            if self._channel in self._favourites :
                self._favourites.remove(self._channel)
            else :
                print("{0} is not in favourites".format(self._channel))

    def channel_up(self):
        """Moves to the next higher favourite channel.

           It does not matter if the current channel is a favourite or not.
           Channel will wrap around to 1 if it passes MAX_CHANNEL while
           searching for the next higher favourite channel.
        """
        if self._power :
            if not self._favourites :
                print("No favourites stored")
            else :
                while True :
                    self._channel += 1
                    if self._channel > 100 :
                        self._channel = 1
                    if self._channel in self._favourites :
                        break

    def channel_down(self) :
        """Moves to the previous (lower) favourite channel.

           It does not matter if the current channel is a favourite or not.
           Channel will wrap around to MAX_CHANNEL if it passes 1 while
           searching for the previous favourite channel.
        """
        if self._power :
            if not self._favourites :
                print("No favourites stored")
            else :
                while True :
                    self._channel -= 1
                    if self._channel < 1 :
                        self._channel = 100
                    if self._channel in self._favourites :
                        break

    def __str__(self) :
        if self._power :
            return "I'm a Super Deluxe TV on channel {0}".format(self._channel)
        else :
            return "I'm a Super Deluxe TV that is turned off"
```

In the class above, we also need to override the `__init__` method so that the favourites list will be created. However, unlike the `channel_up` and `channel_down` methods, we still want to keep the existing `__init__` functionality from the `DeluxeTV` and `TV` classes. To do this, we need to explicitly call the `__init__` method of the superclass (`DeluxeTV`): `super().__init__()`. The super function works out what superclass has the method attempting to be called and returns

that class, allowing the method to operate on the superclass type. It is simply required from then to input the inputs required to create that subclass. Once that is done, we can add in the additional code to create the favourites list.

We have also overridden the `__str__` method to give a slightly different text representation.

After saving our code as super_deluxe_tv.py we can have a look at a few examples of our new class.

```
>>> tv = SuperDeluxeTV()
>>> tv.turn_on()
>>> str(tv)
"I'm a Super Deluxe TV on channel 1"
>>> tv.channel_up()
No favourites stored
>>> tv.set_channel(42)
>>> str(tv)
"I'm a Super Deluxe TV on channel 42"
>>> tv.store()
>>> tv.set_channel(87)
>>> tv.store()
>>> tv.channel_up()
>>> str(tv)
"I'm a Super Deluxe TV on channel 42"
>>> tv.channel_up()
>>> str(tv)
"I'm a Super Deluxe TV on channel 87"
>>> tv.set_channel(50)
>>> tv.channel_down()
>>> str(tv)
"I'm a Super Deluxe TV on channel 42"
```

In fact we have already been overriding methods in the classes that we have previously written. The `__init__`, `__str__`, `__eq__`, etc. methods that we have written all override the existing, corresponding, method in the `object` class.

Inheritance Syntax

```
class Class(SuperClass) :
    ...
```

Semantics

A new class, *`Class`*, will be created, **inheriting** from *`SuperClass`*. Instances will be able to use all the methods and class variables defined in the superclass, as well as any methods defined in *`Class`*. Instances will also be able to access methods and class variables of the superclass of *`SuperClass`*, and so on through the superclasses (these are all **indirect superclasses** of *`Class`*, and *`SuperClass`* is a **direct superclass**). The collection of superclass-subclass relationships is known as the **inheritance hierarchy**.

If *`Class`* defines a method with the same name as one in a superclass (direct or indirect), then that method has been **overridden**. When the method *`instance.method(arg1, ...)`* is called, the method defined in *`Class`* is used. However, the overridden method can still be accessed directly by calling *`super().method(arg1, ...)`*

Method Resolution Order

When we override a method, we are creating a new method with the same name. This raises an issue — how does Python determine which method to call? Python has a set of rules that determine which class the method will come from, known as the **Method Resolution Order**, or **MRO**. As an example, consider the four classes below.

```
class A(object) :
    def __init__(self, x) :
        self._x = x

    def f(self) :
        return self._x

    def g(self) :
        return 2 * self._x

    def fg(self) :
        return self.f() - self.g()

class B(A) :
    def g(self) :
        return self._x ** 2

class C(B) :
    def __init__(self, x, y) :
        super().__init__(x)
        self._y = y

    def fg(self) :
        return super().fg() * self._y

class D(A) :
    def f(self) :
        return -2 * self.g()
```

After saving this code as mro.py, we can look at the functionality of our classes. To start with, let's just look at what an object of class `A` does.

```
>>> a = A(3)
>>> a._x
3
>>> a.f()
3
>>> a.g()
6
>>> a.fg()
-3
```

This class is straightforward. The constructor `A(3)` calls the `__init__` method, which creates an attribute `_x` with the value `3`. Method `f` returns the value of `_x`, method `g` returns twice `_x` and method `fg` returns the difference between the results of these two methods.

Now, let's have a look at using an object of the `B` class.

```
>>> b = B(3)
>>> b._x
3
>>> b.f()
3
>>> b.g()
9
>>> b.fg()
-6
```

Let's consider what happens when we construct the object `B(3)`. The Python interpreter will create a new object and attempt to call the `__init__` method. However, the `B` class does not have a definition of the `__init__` method. Python then looks to the superclass for an `__init__` method. The superclass is `A`, which has an `__init__` method. This `__init__` method is called, which creates an attribute `_x` with the value `3`.

What happens when we call `b.f()`? The `B` class does not have a definition of the `f` method. Therefore the interpreter looks in the superclass `A` for an `f` method, which it finds. This method is used, which returns the value of `b.x` (which is 3).

Next the `g` method is called. The `B` class has a `g` method so the interpreter uses that `g` method, returning the square of `x`, in this case 9.

When the `fg` method is called, B again does not have this method so the interpreter looks back to the superclass and uses the `fg` method from `A`, which returns `self.f() - self.g()`. Because the `fg` method was called with a `B` object, the interpreter will look for `f` and `g` methods in the `B` class. As before, `B` does not have a definition of `f`, so the interpreter uses the definition from `A`, which returns 3. The process is repeated for the `g` method call, but as the `B` class has a definition of `g`, that method is called, which returns 9. So the return value of `b.fg()` is 3 - 9, which is -6.

Now, let's look at using an object of the `C` class.

```
>>> c = C(3, 5)
>>> c._x
3
>>> c._y
5
>>> c.f()
3
>>> c.g()
9
>>> c.fg()
-30
```

The `C` class inherits from the `B` class, which in turn inherits from `A`. When `C(3, 5)` is constructed, Python will look for an `__init__` method in `C`, which it finds and uses. This calls `super().__init__(3)` method. The super() function will first look in `B` for an `__init__` method. As `B` does not have an `__init__` method, super will look at `B`'s super class, which is `A` for an `__init__` method. As `A` does have a `__init__` method, super evaluates to `A`. Therefore, the `super().__init__(3)` line in `C`'s `__init__` method results in calling `A.__init__(3)`. `A`'s `__init__` will, therefore, set the value of the attribute `x` to 3. The next line in `C.__init__` will now run, setting the value of the attribute `y` to 5.

When `c.f()` is called, `C` does not have an `f` method, so the interpreter looks in the superclass, `B`. `B` also does not have an `f` method so the interpreter looks in `B`'s superclass, which is `A`. The `f` method of A is finally used, which returns the value of the attribute `_x`.

The process is the same again for the `g` method call. As `C` does not have a `g` method, the interpreter looks in the `B` class. `B` does have a `g` method, so that is used, and returns `3 ** 2`.

Next is the `fg` method call on `C`. This method first calls `super().fg()` which will look for an `fg` method first in the `B` class. As `B` doesn't have an `fg` method, the super function will look in B's superclass, which is `A`, and, therefore, `A.fg()` is called. This method follows the process just described to call both the `f` method from `A` and the `g` method from B and subtract the results together, performing the operation `3 - 9`. The result of the `super().fg()` call is -6, which is multiplied by the attribute `y` (5) to give -30 as the final result.

Finally, let's have a look at using an object of the `D` class.

```
>>> d = D(3)
>>> d._x
3
>>> d.f()
-12
>>> d.g()
6
>>> d.fg()
-18
```

The `D` class is similar to the `B` class. The `f` method call on `D` uses the `f` method defined in `D`. This method calls `self.g()`. `D` does not have a `g` method, so the interpreter calls the `g` method from the superclass (`A`) class performing the operation -2 * 2 * 3. The `g` method call on `D` works exactly the same way.

When the `fg` method of `D` is called the interpreter uses the `fg` method of `A`, as `D` does not have an `fg` method defined. This calls the `f` method using the `f` method of `D`, which in turn uses the `g` method of `A`, and then calls the `g` method of `A` again. The results are then subtracted performing the operation -12 - 6.

In summary, the MRO can be easily determined by simply following the chain of inheritance of the classes involved, remembering to always start at the class the method call is performed on.

Aside: Multiple Inheritance

Python supports the ability for classes to inherit from more than one superclass, known as **multiple inheritance**. As an example, consider these four classes:

```
class A(object) :
    def __init__(self, x) :
        self._x = x

    def f(self) :
        return self._x

class B(A) :
    def f(self) :
        return self._x ** 2
```

```
class C(A) :
    def __init__(self, x, y) :
        super().__init__(x)
        self._y = y

    def g(self) :
        return self.f() * self._y

class D(B, C) :
    def info(self) :
        print("x = {0}, y = {1}".format(self_.x, self._y))
        print("f() -> {0}".format(self.f()))
        print("g() -> {0}".format(self.g()))
```

The class `D` inherits from both `B` and `C`, which each inherit from `A`. Here is an interaction with the `D` class:

```
>>> d = D(3, 5)
>>> d.info()
x = 3, y = 5
f() -> 9
g() -> 45
```

When the `f` method is called, Python will use the `f` method defined in `B`. When the `g` method is called, Python uses the `g` method defined in `C`. This method calls `self.f()`, which will again be the method defined in `B`. We see that without needing to modify anything in the `D` class, we have manipulated the `g` method defined in the `C` class to use the `f` method of `B` instead of `A`. This technique can be useful in certain situations.

However, we must be *very* careful when using multiple inheritance, as it makes the MRO much harder to interpret. If `B` and `C` were to both define the same method, which one is used? If `D` overrides this method and needs to access the superclass method, should super evaluate to `B` or `C`? Is it possible for unwanted side-effects to occur? Does the order of inheritance matter (`class D(B, C):` or `class D(C, B):`)?

Using multiple inheritance is **often** inadvisable where other techniques can be used to achieve the same desired outcome. Some programming languages, such as Java and Ruby, do not support multiple inheritance, as it can be unnecessarily complex.

Writing Exceptions

We previously discussed various types of exceptions, and how to use them. Python has a range of built-in exceptions for various generic purposes. For example, `NameError` is raised when a variable name is not found, and `ValueError` when an input is of an inappropriate value.

Often it is useful to create different types of exception for specific purposes. In the following example, we will write a class to represent a savings bank account, which will have methods for withdrawing and depositing funds, and accumulating interest. When the user tries to withdraw or deposit a negative amount, we will raise a `ValueError`. When the user tries to withdraw too much money, then we would like to raise a different type of error that represents a failed withdrawal.

Python comes with a class to represent exceptions, called `Exception`. By creating a class which inherits from `Exception`, we can create our own custom exceptions. Here is an exception for a failed withdrawal:

```
class WithdrawalError(Exception) :
    """An exception to be raised when a withdrawal fails."""
    def __init__(self, message, balance):
        """
        Parameters
            message (str): The reason for the error
            balance (float): The balance of the account
        """
        super().__init__(message)
        self._balance = balance

    def get_balance(self) :
        return self._balance
```

The exception stores a message detailing why the transaction failed, and the balance of the savings account. Using this, we can write a class to represent the savings account:

```
class SavingsAccount(object) :
    """A simple savings account."""
    def __init__(self, owner, initial_deposit, monthly_interest) :
        self._owner = owner
        self._balance = initial_deposit
        self._interest = monthly_interest

    def get_owner(self) :
        return self._owner

    def get_balance(self) :
        return self._balance

    def add_interest(self) :
        """Add the monthly interest to the balance."""
        if self._balance > 0 :
            interest = self._balance * self._interest / 100.0
            self._balance += interest

    def deposit(self, amount) :
        """Add a positive amount to the balance."""
        if amount < 0 :
            raise ValueError("Can't deposit a negative amount")
        self._balance += amount

    def withdraw(self, amount) :
        """Subtract a positive amount from the balance."""
        if amount < 0 :
            raise ValueError("Can't withdraw a negative amount")
        new_balance = self._balance - amount
        if new_balance < 0 :
            raise WithdrawalError("Not enough funds in account", self._balance)
        else :
            self._balance = new_balance
```

```
    def __repr__(self) :
        repr_string = "SavingsAccount({0}, {1}, {2})"
        return repr_string.format(self._owner, self._balance, self._interest)
```

We can then write functions `withdraw` and `deposit` that the user can interact with:

```
def withdraw(account, amount) :
    try :
        account.withdraw(amount)
    except ValueError as e :
        print("Invalid:", e)
    except WithdrawalError as e :
        print("Cannot withdraw:", e)
        print("Your account balance is ${0}".format(e.get_balance()))
    else :
        print("Withdrawal successful.")


def deposit(account, amount) :
    try :
        account.deposit(amount)
    except ValueError as e :
        print("Invalid:", e)
    else :
        print("Deposit successful.")
```

These functions both use a `try`-`except`-`else` construct. In these constructs, the `else` body is executed if there were no exceptions raised in the `try` body.

The code above is available in the file banking.py. Below is an example interaction with this savings account:

```
>>> savings = SavingsAccount("John Cleese", 100, 0.3)
>>> savings
SavingsAccount(John Cleese, 100, 0.3)
>>> withdraw(savings, 60)
Withdrawal successful.
>>> savings.get_balance()
40
>>> savings.add_interest()
>>> savings.get_balance()
40.12
>>> deposit(savings, -20)
Invalid: Can't deposit a negative amount
>>> withdraw(savings, 45)
Cannot withdraw: Not enough funds in account
Your account balance is $40.12
```

Inheritance as Abstraction

Following from the previous example, we write a class that represents a credit account.

```
class CreditAccount(object) :
    """A simple credit account."""
    def __init__(self, owner, initial_deposit, monthly_interest, limit) :
        self._owner = owner
        self._balance = initial_deposit
        self._interest = monthly_interest
        self._limit = limit

    def get_owner(self) :
        return self._owner

    def get_balance(self) :
        return self._balance

    def add_interest(self) :
        """Subtract the monthly interest from the balance."""
        if self._balance < 0 :
            interest = self._balance * self._interest / 100.0
            self._balance += interest

    def deposit(self, amount) :
        """Add a positive amount to the balance."""
        if amount < 0 :
            raise ValueError("Can't deposit a negative amount")
        self._balance += amount

    def withdraw(self, amount) :
        """Subtract a positive amount from the balance."""
        if amount < 0 :
            raise ValueError("Can't withdraw a negative amount")
        new_balance = self._balance - amount
        if new_balance < -self._limit :
            raise WithdrawalError("Credit limit reached", self._balance)
        else :
            self._balance = new_balance

    def __repr__(self) :
        repr_string = "CreditAccount({0}, {1}, {2}, {3})"
        return repr_string.format(self._owner,
                                  self._balance,
                                  self._interest,
                                  self._limit)
```

Notice that the class definitions of `SavingsAccount` and `CreditAccount` are very similar; in fact, some of the methods are identical. This is to be expected as the credit and savings accounts are both types of bank account. We would like to abstract out the duplicated methods, and create an "account" class which represents the common functionality of the two types of account. Once this is done, the `SavingsAccount` and `CreditAccount` classes can inherit from `Account`.

Between the two classes, we see that the `get_owner`, `get_balance` and `deposit` methods are identical, so these methods can be moved to the `Account` class. The `__init__` method also

contains similar behaviour, so it can easily be moved to `Account`. The other three methods (`add_interest`, `withdraw` and `__repr__`) look similar, but not in a way that can be easily abstracted, so we will leave them as they are. The file `banking2.py` contains the code below.

```
class Account(object) :
    """An abstraction of different bank accounts."""
    def __init__(self, owner, initial_deposit, monthly_interest) :
        self._owner = owner
        self._balance = initial_deposit
        self._interest = monthly_interest

    def get_owner(self) :
        return self._owner

    def get_balance(self) :
        return self._balance

    def deposit(self, amount) :
        """Add a positive amount to the balance."""
        if amount < 0 :
            raise ValueError("Can't deposit a negative amount")
        self._balance += amount


class SavingsAccount(Account) :
    """A simple savings account."""
    def add_interest(self) :
        """Add the monthly interest to the balance."""
        if self._balance > 0 :
            interest = self._balance * self._interest / 100.0
            self._balance += interest

    def withdraw(self, amount) :
        """Subtract a positive amount from the balance."""
        if amount < 0 :
            raise ValueError("Can't withdraw a negative amount")

        new_balance = self._balance - amount
        if new_balance < 0 :
            raise WithdrawalError("Not enough funds in account", self._balance)
        else :
            self._balance = new_balance

    def __repr__(self) :
        repr_string = "SavingsAccount({0}, {1}, {2})"
        return repr_string.format(self._owner, self._balance, self._interest)
```

```
class CreditAccount(Account) :
    """A simple credit account."""
    def __init__(self, owner, initial_deposit, monthly_interest, limit) :
        super().__init__(owner, initial_deposit, monthly_interest)
        self._limit = limit

    def add_interest(self) :
        """Subtract the monthly interest from the balance."""
        if self._balance < 0 :
            interest = self._balance * self._interest / 100.0
            self._balance += interest

    def withdraw(self, amount) :
        """Subtract a positive amount from the balance."""
        if amount < 0 :
            raise ValueError("Can't withdraw a negative amount")

        new_balance = self._balance - amount
        if new_balance < -self._limit :
            raise WithdrawalError("Credit limit reached", self._balance)
        else :
            self._balance = new_balance

    def __repr__(self) :
        repr_string = "CreditAccount({0}, {1}, {2}, {3})"
        return repr_string.format(self._owner,
                                  self._balance,
                                  self._interest,
                                  self._limit)
```

In the television example above, we used inheritance to add more functionality to an existing type of television. Here, we have used inheritance to abstract functionality from two similar classes. A result of this is that the `Account` class is not a fully functional bank account; it does not have the ability to withdraw funds or accumulate interest. Because of this, we call the `Account` an **abstract class**.

Abstract classes should not be instantiated directly (that is, we should not create instances of the abstract class itself), instead we create subclasses that fill in the missing functionality. We have already seen an example of an abstract class, `Exception`. We never raise an `Exception` itself, but we use various subclasses of `Exception` to represent different things. An advantage of using the concept of an abstract `Account` class is that we can write functions that work with any type of account. For example, the `withdraw` and `deposit` functions we wrote above can also work with `CreditAccount` instances:

```
>>> credit = CreditAccount("Michael Palin", 0, 1.5, 200)
>>> credit
CreditAccount(Michael Palin, 0, 1.5, 200)
>>> withdraw(credit, 150)
Withdrawal successful.
>>> credit.add_interest()
>>> credit.get_balance()
-152.25
>>> deposit(credit, 60)
Deposit successful.
```

```
>>> credit.get_balance()
-92.25
>>> withdraw(credit, 110)
Cannot withdraw: Credit limit reached
Your account balance is $-92.25
```

Designing an Inheritance Hierarchy

As another example we now sketch out some ideas for designing a simple inheritance hierarchy for people at a university.

If we look around campus we will see lots of people and we can look at them more closely, searching for similarities. Some of these people sit in lectures and do exams. These people have a lot in common and so we might define a class to describe their attributes — let's say an `UndergraduateStudent` class. What do members of this class have in common? They have names, addresses, student numbers, student records, courses they are enrolled in. If we look around we will find some other kinds of students that have slightly different attributes — let's put them in a `PostgraduateStudent` class. These students will share some attributes with the other students but have their own attributes — for example, thesis topic and supervisor. We might then decide that both kinds of students have a lot of overlap and decide that they both inherit from `Student`.

There are other people walking around that get paid — some of these teach classes and do research — let's put them in an `Academic` class. Members of this class will have attributes like staff number, pay scale, roles, teaching assignments and research interests. Others might belong to a `TechnicalStaff` class and others might belong to an `Administration` class. Together we might group these together and form the `Staff` superclass and then together with students create the `Person` superclass.

Now that we have sketched out the inheritance hierarchy we then want to determine where specific attributes should live. Let's start with name and address. Everyone has these attributes and so belong to the base `Person` class. On the other hand, the attribute (method) for enrolling in a course is specific to an undergraduate student and so should go in that class.

What we have done is a bottom-up design of the class hierarchy — we started at the bottom and worked our way up moving from subclass to superclass. We could also do top-down design — by starting with the base class (`Person`) and looking for differences — for example staff get paid and students do not. This gives the subclasses of `Person`.