Object Composition and Inheritance

Prof Matthew Fricke

List the classes we will need to define a course enrollment management program

Relationships between these classes

Recap: Procedural Programming

- Procedural programming uses:
  - Data structures (like integers, strings, lists)
  - Functions (like `addints()`)
- In procedural programming, information must be passed to the function
  - Functions and data structures are not linked

Recap: Object-Oriented Programming (OOP)

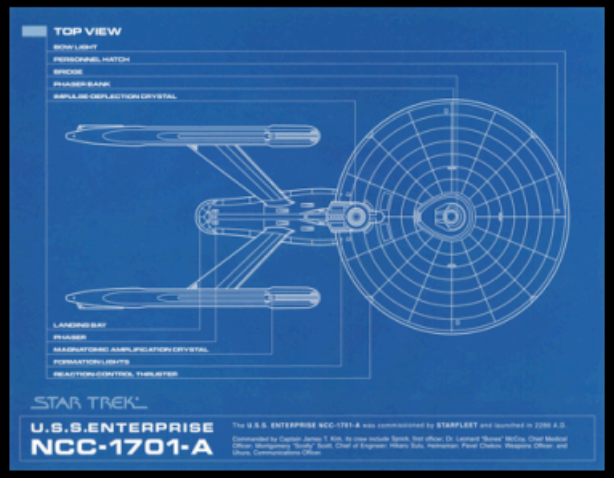
- Object-Oriented programming uses
  - Classes!

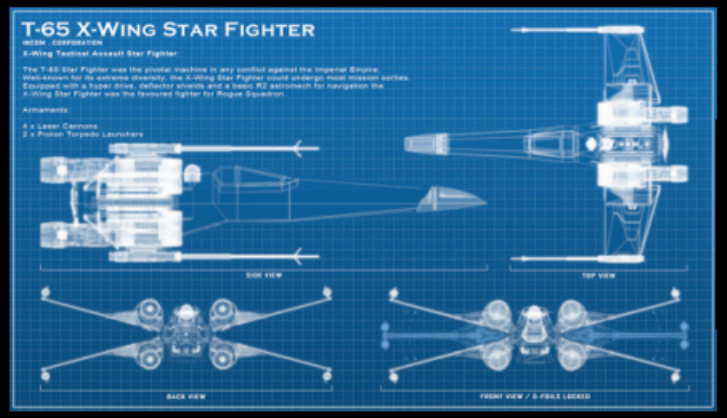
- Classes combine the data and their relevant functions into one entity
  - The data types we use are actually classes!
  - Strings have built-in functions like `lower()`, `join()`, `strip()`, etc.

Classes vs Objects

Classes are like the job description

The object is the person hired to do the job.



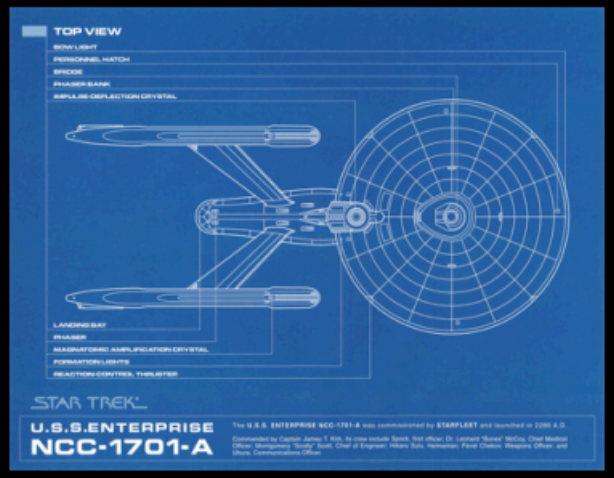


- Imagine when you are writing a class that it is a blueprint.
- Instantiating a class is building the object described by the blueprint.

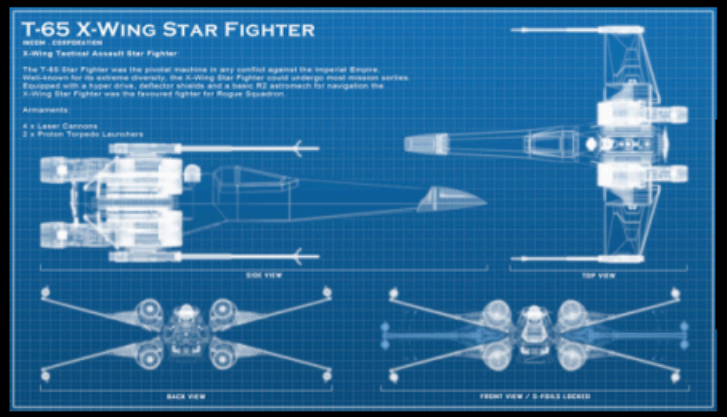
Classes vs Objects

Classes are like the job description

The object is the person hired to do the job.



Instantiations of the class definition



- Imagine when you are writing a class that it is a blueprint.
- **Instantiating** a class is building the object described by the blueprint.

Has a… (composition of objects)

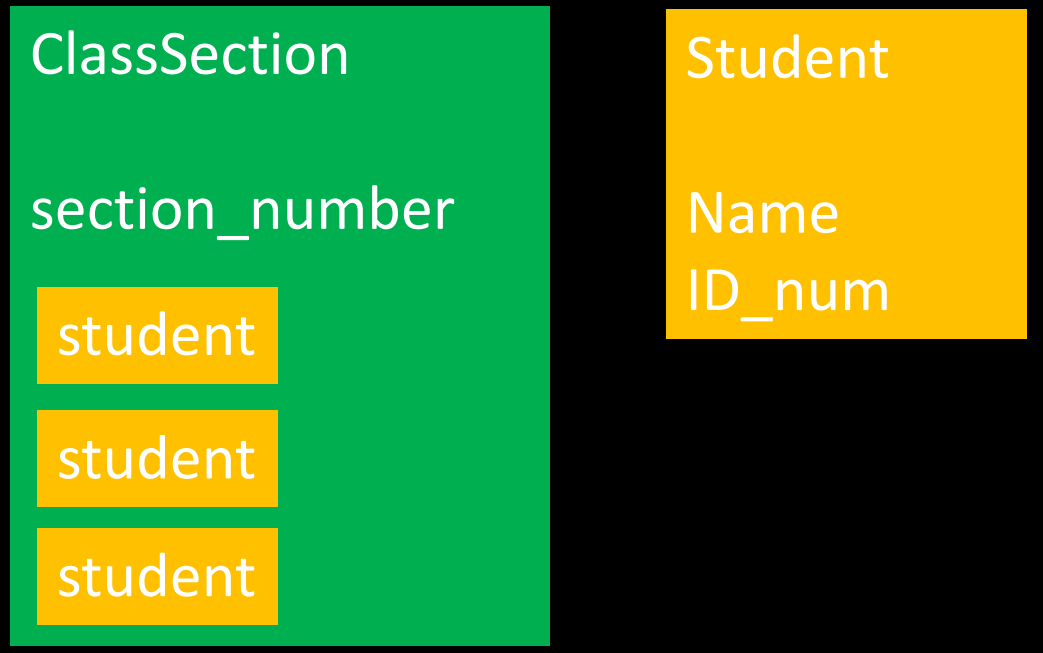
- A course has class sections
- Class sections have students

Student

Name
ID_num

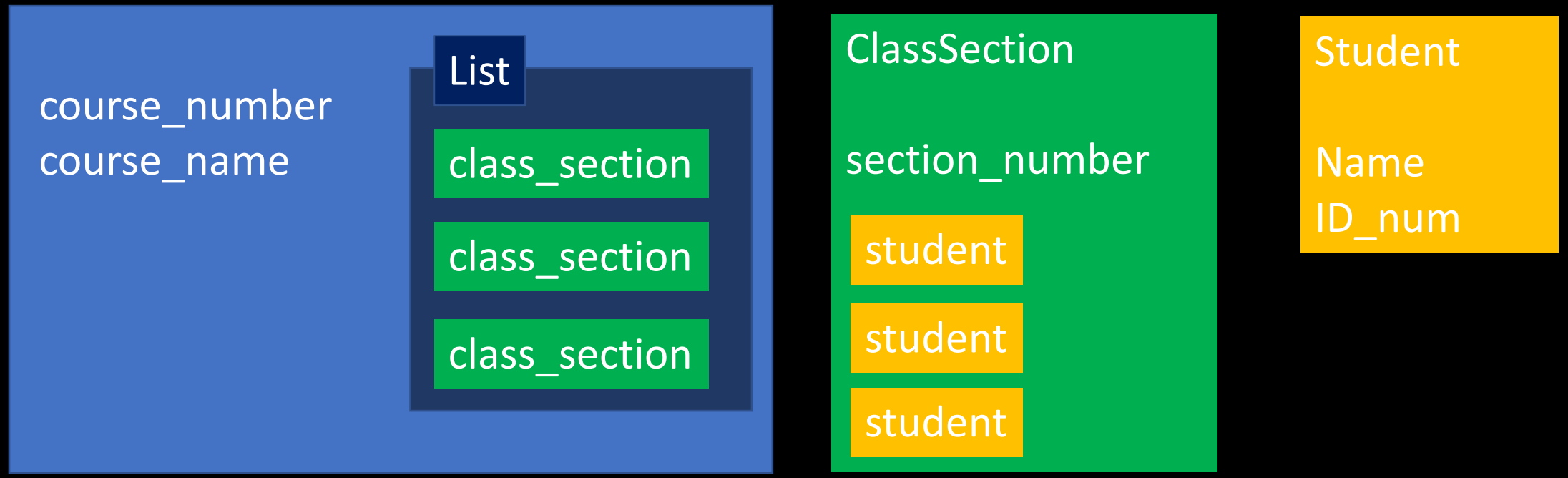
Has a… (composition of objects)

- A course has class sections
- Class sections have students



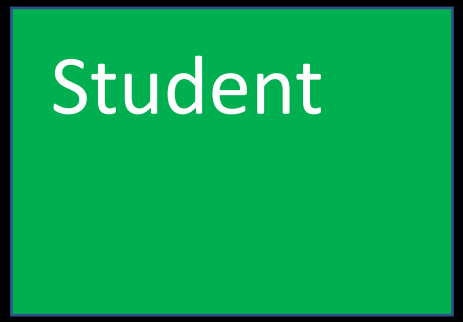
Has a… (composition of objects)

- A course has class sections
- Class sections have students



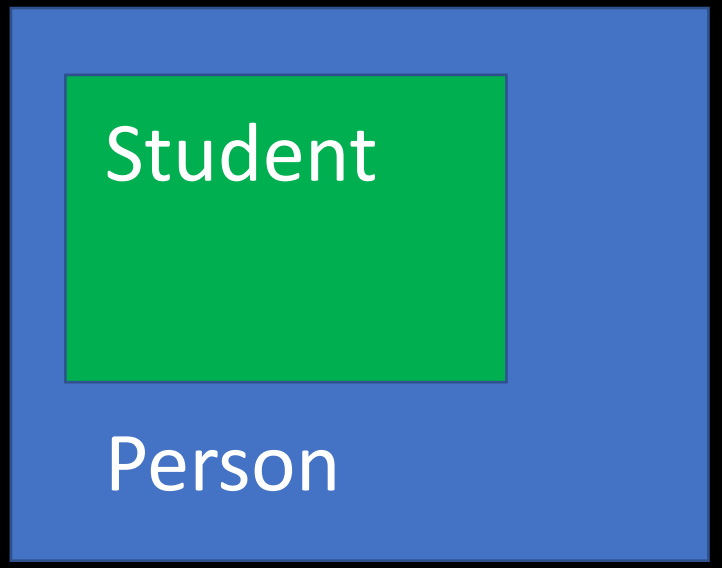
Is a… (inheritance of classes)

- A student



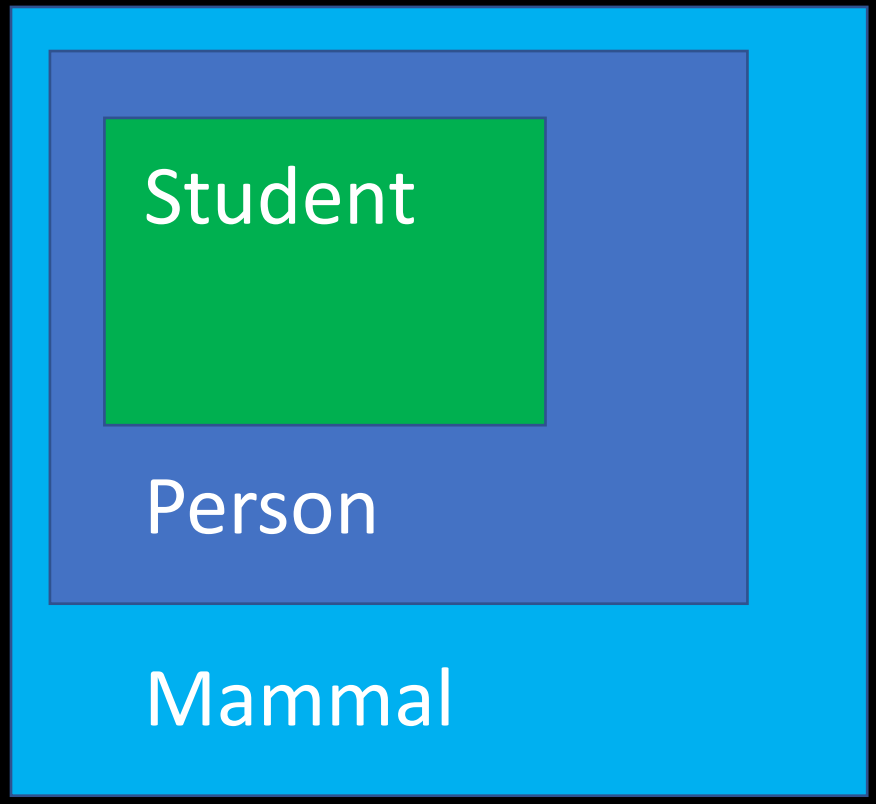
Is a… (inheritance of classes)

- A student is a person



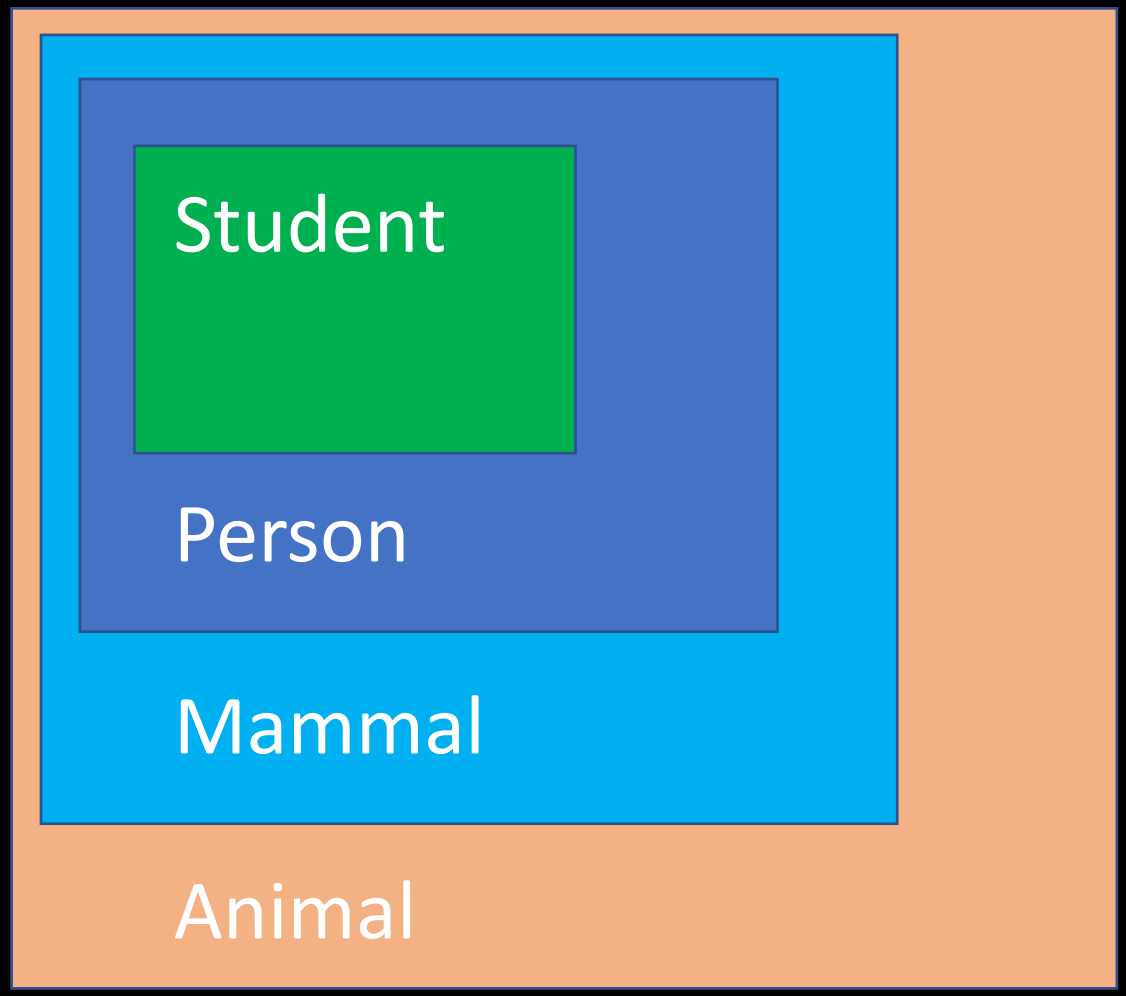
Is a… (inheritance of classes)

- A student is a person
- A person is a mammal



Is a… (inheritance of classes)

- A student is a person
- A person is a mammal
- A mammal is an animal

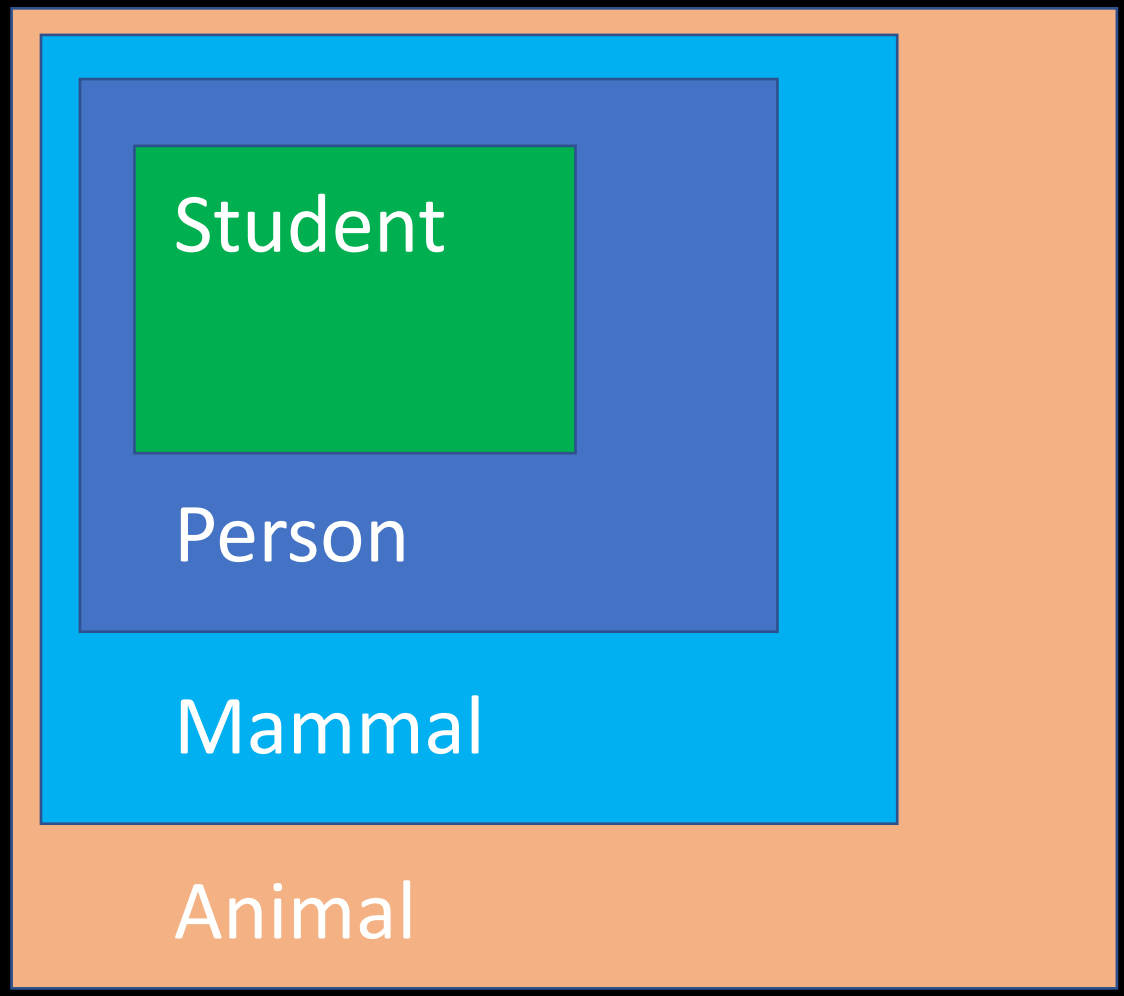


Lot's of names....

- Unfortunately there are lots of terms for the same thing:

- Parent Class
  = Base Class
  = Superclass

- Child Class
  = Derived Class
  = Subclass



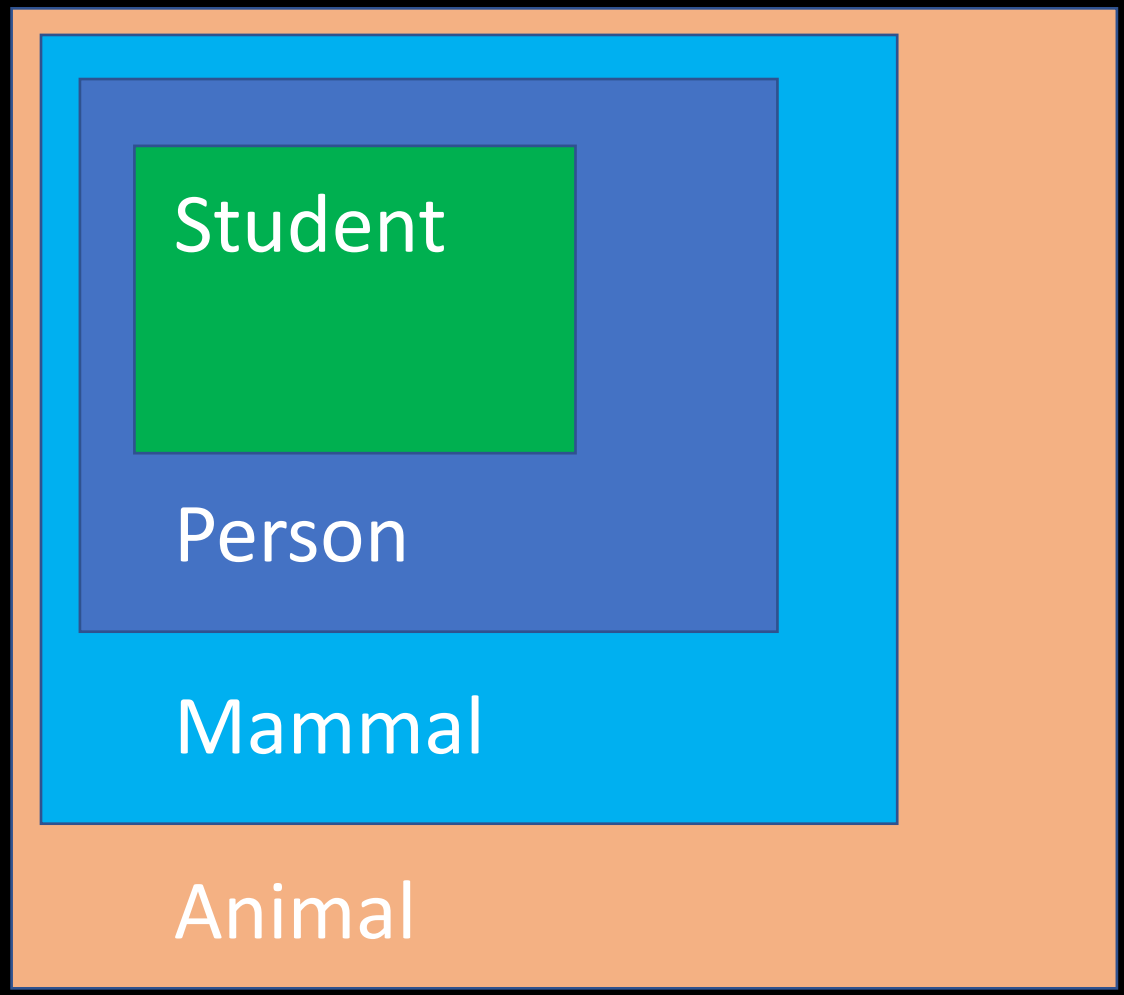
Lot’s of names….

- Unfortunately there are lots of terms for the same thing:

- Parent Class
  = Base Class
  = Superclass

- Child Class
  = Derived Class
  = Subclass

Student is a child/derived/subclass of Person, Mammal, and Animal.



Lot’s of names….

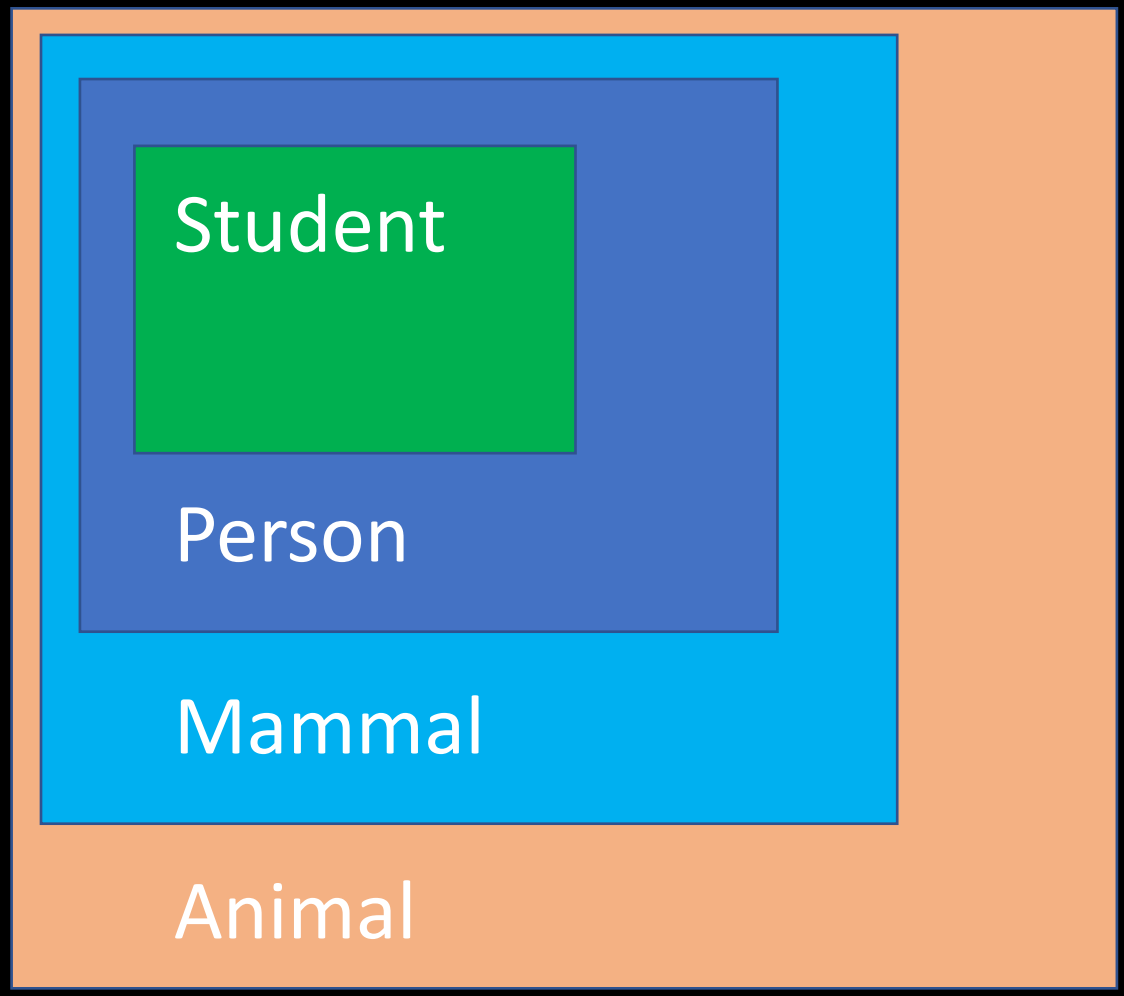
- Unfortunately there are lots of terms for the same thing:

- Parent Class
  = Base Class
  = Superclass

Mammal is a child/derived/subclass of Animal, and a parent/base/superclass of person and student.

- Child Class
  = Derived Class
  = Subclass

Student is a child/derived/subclass of Person, Mammal, and Animal.



Pet example

- Here is a simple class that defines a Pet object.

In pet.py

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet's name is " + str(self.name)
```

The __str__ built-in function defines what happens when I print an instance of Pet. Here I'm overriding it to print the name.

Pet example

- Here is a simple class that defines a Pet object.

In pet.py

```
class Pet:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def __str__(self):
        return "This pet's name is " + str(self.name)
```

```
>>> from pet import Pet
>>> mypet = Pet('Ben', '1')
>>> print mypet
This pet's name is Ben
>>> mypet.get_name()
'Ben'
>>> mypet.get_age()
1
```

Inheritance

- Now, let's say I want to create a Dog class which inherits from Pet. The basic format of a derived class is as follows:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

In the case of `BaseClass` being defined elsewhere, you can use `module_name.BaseClassName`.

Inheritance

- Here is an example definition of a Dog class which inherits from Pet.

```
class Dog(Pet):
    pass
```

- The pass statement is only included here for syntax reasons. This class definition for Dog essentially makes Dog an alias for Pet.

Inheritance

- Here is an example definition of a Dog class which inherits from Pet.

```
class Dog(Pet):
    pass*
```

- The pass statement is only included here for syntax reasons. This class definition for Dog essentially makes Dog an alias for Pet.

*pass – this is a special keyword in Python. It is a placeholder that does nothing but prevents syntax errors for things that expect there to be a statement.

Pass (an aside)

- For example:

```
if x > 1:
    # I have no idea what to write here yet…
```

```
IndentationError: expected an indented block
```

Pass (an aside)

- For example:

```
if x > 1:
    # I have no idea what to write here yet…
    pass
```

No syntax error – so I can fill it out later

inheritance

- We've inherited all the functionality of our Pet class, now let's make the Dog class more interesting.

```
>>> from dog import Dog
>>> mydog = Dog('Ben', 1)
>>> print mydog
This pet's name is Ben
>>> mydog.get_name()
'Ben'
>>> mydog.get_age()
1
```

```
class Dog(Pet):
    pass
```

inheritance

- For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

inheritance

- For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

Overriding initialization function

Python resolves attribute and method references by first searching the derived class and then searching the base class.

inheritance

- For my Dog class, I want all of the functionality of the Pet class with one extra attribute: breed. I also want some extra methods for accessing this attribute.

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

```
self.name = name
self.age = age
```

We can call base class methods directly using `BaseClassName.method(self, arguments)`. Note that we do this here to extend the functionality of Pet's initialization method.

inheritance

```
>>> from dog import Dog
>>> mydog = Dog('Ben', 1, 'Maltese')
>>> print mydog
This pet's name is Ben
>>> mydog.get_age()
1
>>> mydog.get_breed()
'Maltese'
```

```
class Dog(Pet):
    def __init__(self, name, age, breed):
        Pet.__init__(self, name, age)
        self.breed = breed
    def get_breed(self):
        return self.breed
```

inheritance

- Python has two notable built-in functions:
- `isinstance(`*`object`*`, `*`classinfo`*`)` returns true if *object* is an instance of *classinfo* (or some class derived from classinfo).
- `issubclass(`*`class`*`, `*`classinfo`*`)` returns true if *class* is a subclass of *classinfo*.

```
>>> from pet import Pet
>>> from dog import Dog
>>> mydog = Dog('Ben', 1, 'Maltese')
>>> isinstance(mydog, Dog)
True
>>> isinstance(mydog, Pet)
True
>>> issubclass(Dog, Pet)
True
>>> issubclass(Pet, Dog)
False
```

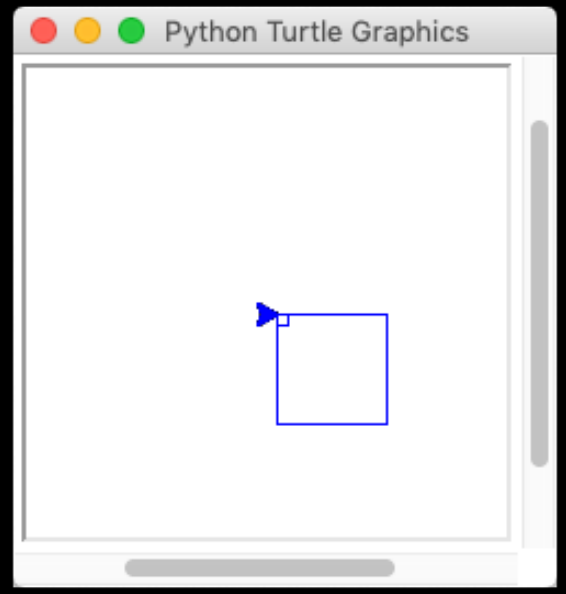
Shapes Example – Defining a “Square” class

In shapes.py

```
import turtle
import math

class Square():

    # Size

    def __init__(self, s ):
        self.size = s
        self.turtle = turtle.Turtle()
        self.colour = "blue"

    def getArea(self):
        return self.size**2

    def draw( self ):
        self.turtle.color( self.colour )
        for i in range(4):
            self.turtle.forward( self.size )
            self.turtle.right( 90 )

    def setColour( self, col ):
        self.colour = col
```

In python3 interpreter

```
>>> import shapes
>>> my_square = shapes.Square(50)
>>> my_square.getArea()
2500
>>> my_square.draw()
>>>
```



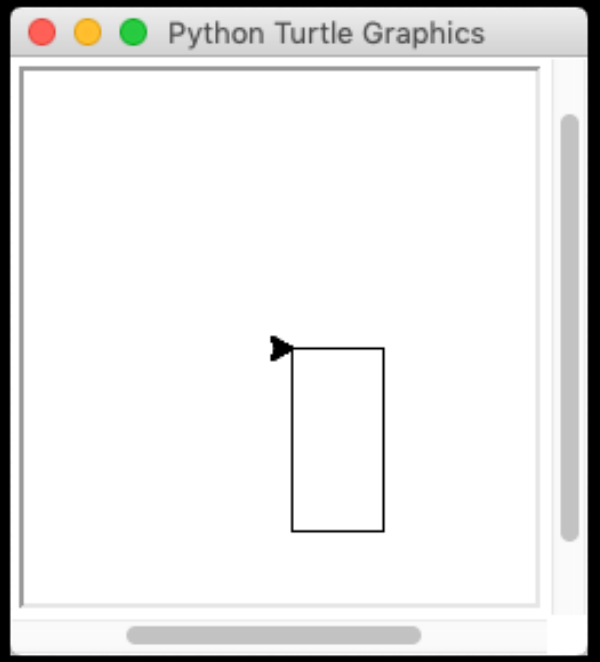
Shapes Example – Defining a “Rectangle” class

In shapes.py

```
class Rectangle():

    def __init__(self, height, length):
        self.length = length
        self.height = height
        self.turtle = turtle.Turtle()
        self.colour = "blue"
    def area(self):
        return self.length*self.height

    def draw( self ):
        for i in range(2):
            self.turtle.forward( self.height )
            self.turtle.right( 90 )
            self.turtle.forward( self.length )
            self.turtle.right( 90 )

    def setColour( self, col ):
        self.colour = col
```

In python3 interpreter

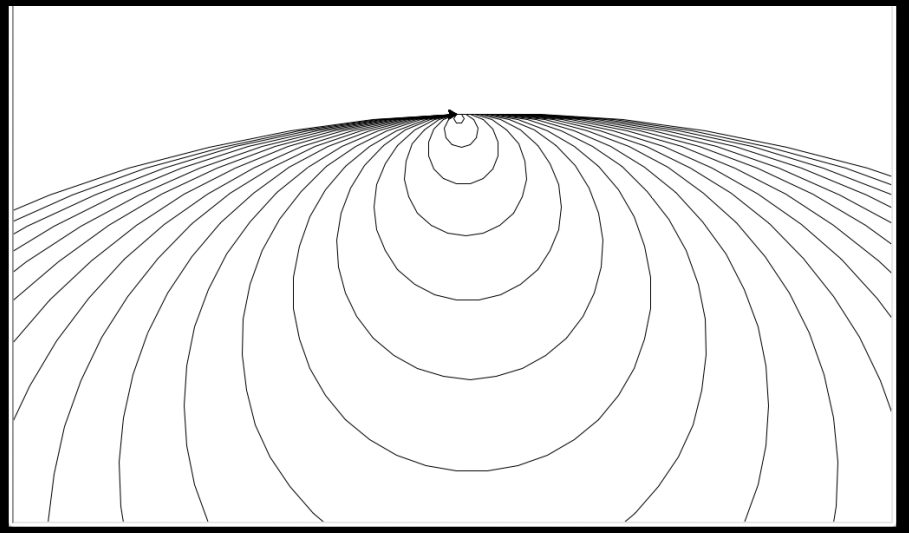
```
>>> import shapes
>>> my_rectangle = shapes.Rectangle(40,80)
>>> my_rectangle.area()
3200
>>> my_rectangle.draw()
>>>
```



Shapes Example – Defining a “RegularPolygon” class

In shapes.py

```
class RegularPolygon():

    def __init__(self, num_sides, size):
        self.size = size
        self.turtle = turtle.Turtle()
        self.colour = "blue"

    def area(self):
        return self.num_sides*(self.size ** 2)/ (4 * math.tan(math.pi / self.num_sides) )

    def draw( self ):
        for i in range( self.num_sides ):
            self.turtle.forward( self.size )
            self.turtle.right( 360 / self.num_sides )

    def setColour( self, col ):
        self.colour = col
```



Practical Reasons to use Inheritance

- Notice that in every Shape had a `colour` variable and a `setColor()` member function.

- Laziness is a virtue: we do not want to write the same code over and over.

- We can use inheritance.

Practical Reasons to use Inheritance

- Notice that in every Shape had a `colour` variable and a `setColor()` member function.
- Since all the shapes we made had colour, that might be something all shapes in general have.

- So let's define class called Shape.

New Parent Class:

```
class Shape:

    def __init__(self):
        self.__colour = colour = black

    def set_color(self, colour):
        self.__colour = colour
```

Rectangle Inherits from Shape:

```
class Rectangle(Shape):

    def __init__(self, length, height):
        super().__init__()
        self. height = height
        self. length = length
        self.turtle = Turtle.Turtle()
```

We use the super keyword to access methods in our parent class. Here we call the parent's constructor to make sure colour = "black" is executed.

And we get all the variables and methods that we defined in Shape for free.

Rectangle Inherits from Shape:

```
>> my_rect = Rectangle()
>> my_rect.setColour(“blue”)
```

```
class Rectangle(Shape):

    def __init__(self, length, height):
        super().__init__()
        self. height = height
        self. length = length
        self.turtle = Turtle.Turtle()
```

And we get all the variables and methods that we defined in Shape for free.

New Parent Class:

```
class Shape:

    def __init__(self):
        self.__colour = colour = “black”
        self.turtle = Turtle.Turtle()

    def set_color(self, colour):
        self.__colour = colour
```

Rectangle Inherits from Shape:

```
class Rectangle(Shape):

    def __init__(self, length, height):
        super().__init__()
        self. height = height
        self. length = length
```

In general we want to move as much as we can into parent classes to make the child classes simpler and more focused.

Shapes Example

- How would we use our regular polygon class to simplify our shapes classes through inheritance.

What is the relationship? Do Square and Rectangle derive from Regular Polygons? Or the other way around or does one of them not derive from either? Why?

How would we implement triangle now?

Logistics

Exam 2 (15% of Final Grade is the Monday after Spring Break)

You have two weeks before Exam 2. The best way to study is to review all the posted lecture slides and type up the code that follows. Run that code, try modifying the code to include other methods, e.g. try implementing getPerimeter() in the RegularPolygon class.

Make sure you can trace through the code (e.g. list the functions that get called and in what class they are defined if I call the setColour() function on the House object)

If you understand the code in the shapes module you will be in good shape for the exam.

Shapes Module (from class)

```
1  import turtle # for simple drawings
2  import math # for tangent function
3
4  # Base class
5  # Provides colour and a drawing turtle
6  class Shape:
7      def __init__(self):
8          self.turtle = turtle.Turtle()
9          self.setColour("blue")
10         self.mirror = False
11         self.flipped = False
12
13     def setColour( self, col ):
14         self.colour = col
15         self.turtle.color( self.colour )
16
17     def setMirror( self, is_mirror ):
18         self.mirror = is_mirror
19
20     def setFlipped( self, is_flipped ):
21         self.flipped = is_flipped
22
```

We moved the functions and variables that were common to our square, rectangle, and regular polygon classes to the shape base class.

We added some more functions like fill() and setFlipped()

Shapes Module (code from class)

- Notice the RegularPolygon inherits from Shape.

- Now we can simplify classes like RegularPolygon by removing the code for turtles and colours that is now in Shape.

```
22
23  # Regular polygon inherits from Shape
24  class RegularPolygon(Shape):
25
26      def __init__(self, num_sides, side_len):
27          super().__init__()
28          self.side_length = side_len
29          self.num_sides = num_sides
30
31      def getArea(self):
32          return self.num_sides*(self.side_length ** 2)/ (4 * math.tan(m
33
34      def draw( self ):
35          if self.mirror:
36              for i in range( self.num_sides ):
37                  self.turtle.backward( self.side_length )
38                  if self.flipped:
39                      self.turtle.left( 360 / self.num_sides )
40                  else:
41                      self.turtle.right( 360 / self.num_sides )
42          else:
43              for i in range( self.num_sides ):
44                  self.turtle.forward( self.side_length )
45                  if self.flipped:
46                      self.turtle.left( 360 / self.num_sides )
47                  else:
48                      self.turtle.right( 360 / self.num_sides )
49
50      def fill(self):
51          self.turtle.begin_fill()
52          self.draw()
53          self.turtle.end_fill()
54
55      def setSideLength(self, len):
56          self.side_length = len
57
```

Shapes Module (code from class)

- Notice the RegularPolygon inherits from Shape.

- Now we can simplify classes like RegularPolygon by removing the code for turtles and colours that is now in Shape.

```
57
58  # Rectangle inherits from Shape
59  class Rectangle(Shape):
60
61      def __init__(self, height, length):
62          Super().__init__()
63          self.length = length
64          self.height = height
65
66      def getArea(self):
67          return self.length*self.height
68
69      def draw( self ):
70          if mirror:
71              for i in range(2):
72                  self.turtle.backwards( self.height )
73                  self.turtle.right( 90 )
74
75                  self.turtle.backwards( self.width )
76                  self.turtle.right( 90 )
77          else:
78              for i in range(2):
79                  self.turtle.forward( self.height )
80                  self.turtle.right( 90 )
81
82                  self.turtle.forward( self.width )
83                  self.turtle.right( 90 )
84
```

Shapes Module (code from class)

- We can define shape classes. Since they inherit from the RegularPolygon class they get lots of useful functions for free.
- E. g. We don't have to write the draw function for each one.
- Notice we use super().__init__ to make the RegularPolygon base class instantiate itself so we can use its data.

```
85  # The following shapes inherit from RegularPolygon
86  class Square(RegularPolygon):
87
88      def __init__(self, side_len):
89          super().__init__(4, side_len)
90
91  class EquilateralTriangle(RegularPolygon):
92
93      def __init__(self, side_len):
94          super().__init__(3, side_len)
95
96  class Pentagon(RegularPolygon):
97
98      def __init__(self, side_len):
99          super().__init__(5, side_len)
100
101 class Hexagon(RegularPolygon):
102
103     def __init__(self, side_len):
104         super().__init__(6, side_len)
105
106 class Hectanonagon(RegularPolygon):
107     def __init__(self, side_len):
108         super().__init__(100, side_len)
109
```

Shapes Module (code from class)

- We can use composition of objects (recall the student, section, course example) to build more complex shapes.
- The House, CircOrbits, and Envelope classes contain Triangle, Square, and RegularPolygon objects.
- And they inherit from the Shape object.

```
110 class Envelope(Shape):
111     def __init__(self, size ):
112         self.flap = EquilateralTriangle( size )
113         self.base = Square( size )
114 
115     def draw(self):
116         self.base.draw()
117         self.flap.draw()
118 
119     def setColour( self, col  ):
120         self.base.setColour( col )
121         self.flap.setColour( col )
122 
123 class CircOrbits():
124     def __init__(self, size_increment, number ):
125         self.circles = []
126         for i in range(number):
127             self.circles.append(Hectanonagon(size_increment*i))
128 
129     def draw(self):
130         for i in self.circles:
131             i.turtle.speed( 10 ) # Make the turtle go faster
132             i.draw()
133 
134 class House(Shape):
135     def __init__(self, size ):
136         self.walls = Square( size )
137         self.roof = EquilateralTriangle( size )
138         self.roof.setFlipped( True )
139         self.walls.setColour("red")
140         self.roof.setColour("grey")
141 
142     def draw(self):
143         self.walls.fill()
144         self.roof.fill()
145 
146     def setColour( self, col  ):
147         self.walls.setColour( col )
148         self.roof.setColour( col )
```

Using the shapes module

If the shapes code from the previous slides is saved in the shapes.py file. Then we can use that code with the following, for example:

```
>>> import shapes
>>> e = shapes.Envelope( 100 )
>>> e.draw()
>>> h = shapes.House( 100 )
>>> h.draw()
>>>
```