**Why Use Classes?**

Python is an object-oriented programming language, which means it manipulates programming constructs called *objects*. You can think of an object as a single data structure that contains data as well as functions; the functions of an object are called its *methods*. For example, any time you call

len("Eric")

Python is checking to see whether the string object you passed it has a length, and if it does, it returns the value associated with that *attribute*. When you call

my\_dict.items()

Python checks to see if my\_dict has an items() method (which all dictionaries have) and executes that method if it finds it.

But what makes "Eric" a string and my\_dict a dictionary? The fact that they're instances of the str and dict classes, respectively. A class is just a way of organizing and producing objects with similar attributes and methods.

**Instructions**

**1.**

Check out the code in the editor to the right. We've defined our own class, Fruit, and created a lemoninstance.When you're ready, click Run to get started creating classes and objects of your own.

|  |
| --- |
| class Fruit(object):  """A class that makes various tasty fruits."""  def \_\_init\_\_(self, name, color, flavor, poisonous):  self.name = name  self.color = color  self.flavor = flavor  self.poisonous = poisonous  def description(self):  print "I'm a %s %s and I taste %s." % (self.color, self.name, self.flavor)  def is\_edible(self):  if not self.poisonous:  print "Yep! I'm edible."  else:  print "Don't eat me! I am super poisonous."  lemon = Fruit("lemon", "yellow", "sour", False)  lemon.description()  lemon.is\_edible() |
| I'm a yellow lemon and I taste sour.  Yep! I'm edible. |

**Class Syntax**

A basic class consists only of the classkeyword, the name of the class, and the class from which the new class **inherits** in parentheses. (We'll get to inheritance soon.) For now, our classes will inherit from the object class, like so:

class NewClass(object): # Class magic here

This gives them the powers and abilities of a Python object. By convention, user-defined Python class names start with a capital letter.

**1.**

Create a class called Animal in the editor. For now, in the body of your class, use the pass keyword. (passdoesn't do anything, but it's useful as a placeholder in areas of your code where Python expects an expression.)

|  |
| --- |
| class Animal(object):  pass |

**Classier Classes**

We'd like our classes to do more than... well, *nothing*, so we'll have to replace our passwith something else.

You may have noticed in our example back in the first exercise that we started our class definition off with an odd-looking function: \_\_init\_\_(). This function is required for classes, and it's used to **initialize** the objects it creates. \_\_init\_\_() always takes at least one argument, self, that refers to the object being created. You can think of \_\_init\_\_() as the function that "boots up" each object the class creates.

**Instructions**

**1.**

Remove the pass statement in your class definition, then go ahead and define an \_\_init\_\_() function for your Animal class. Pass it the argument self for now; we'll explain how this works in greater detail in the next section. Finally, put the passinto the body of the \_\_init\_\_()definition, since it will expect an indented block.

Hint

Your \_\_init\_\_() function should look something like this:

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

|  |
| --- |
| class Animal(object):  def \_\_init\_\_(self):  pass |

# Let's Not Get Too Selfish

Excellent! Let's make one more tweak to our class definition, then go ahead and **instantiate** (create) our first object.

So far, \_\_init\_\_() only takes one parameter: self. This is a Python convention; there's nothing magic about the word self. However, it's overwhelmingly common to use self as the first parameter in \_\_init\_\_(), so you should do this so that other people will understand your code.

The part that is magic is the fact that selfis the first parameter passed to \_\_init\_\_(). Python will use the first parameter that \_\_init\_\_() receives to refer to the object being created; this is why it's often called self, since this parameter gives the object being created its identity.

**1.**

Let's do two things in the editor:

* Pass \_\_init\_\_() a second parameter, name.
* In the body of \_\_init\_\_(), let the function know that name refers to the created object's name by typing self.name = name. (This will become crystal clear in the next section.)

Hint

Your syntax should look like this:

class Animal(object): def \_\_init\_\_(self, name): # Set the name parameter here!

|  |
| --- |
| class Animal(object):  def \_\_init\_\_(self, name):  self.name = name |

**Instantiating Your First Object**

Perfect! Now we're ready to start creating objects.

We can access attributes of our objects using *dot notation*. Here's how it works:

class Square(object): def \_\_init\_\_(self): self.sides = 4 my\_shape = Square() print my\_shape.sides

1. First we create a class named Squarewith an attribute sides.
2. Outside the class definition, we create a new instance of Square named my\_shape and access that attribute using my\_shape.sides.

**1.**

Outside the Animal class definition, create a variable named zebra and set it equal to Animal("Jeffrey").

Then print out zebra's name.

Hint

You can create a new Animalobject named "Jeffrey" like this:

zebra = Animal("Jeffrey")

You can print out "Jeffrey"'s name like this:

print zebra.name

|  |
| --- |
| class Animal(object):  def \_\_init\_\_(self, name):  self.name = name  zebra = Animal("Jeffrey")  print zebra.name |
| Jeffrey |

|  |
| --- |
| class Animal(object):  def \_\_init\_\_(self, name):  self.name = name  zebra = Animal("Jeffrey")  print zebra |
| <Animal object at 0x7f0a4f4635d0> |

**More on \_\_init\_\_() and self**

Now that you're starting to understand how classes and objects work, it's worth delving a bit more into \_\_init\_\_() and self. They can be confusing!

As mentioned, you can think of \_\_init\_\_() as the method that "boots up" a class' instance object: the init bit is short for "initialize."

The first argument \_\_init\_\_() gets is used to refer to the instance object, and by convention, that argument is called self. If you add additional arguments—for instance, a name and age for your animal—setting each of those equal to self.name and self.age in the body of \_\_init\_\_() will make it so that when you create an instance object of your Animalclass, you need to give each instance a name and an age, and those will be associated with the particular instance you create.

**Instructions**

**1.**

Check out the examples in the editor. See how \_\_init\_\_() "boots up" each object to expect a name and an age, then uses self.nameand self.age to assign those names and ages to each object? Add a third attribute, is\_hungry to \_\_init\_\_(), and click Run to see the results.

Hint

Your code should look something like this:

def \_\_init\_\_(self, name, age, is\_hungry) self.name = name self.age = age self.is\_hungry = is\_hungry

|  |
| --- |
| # Class definition  class Animal(object):  """Makes cute animals."""  # For initializing our instance objects  def \_\_init\_\_(self, name, age, is\_hungry):  self.name = name  self.age = age  self.is\_hungry = is\_hungry  # Note that self is only used in the \_\_init\_\_()  # function definition; we don't need to pass it  # to our instance objects.  zebra = Animal("Jeffrey", 2, True)  giraffe = Animal("Bruce", 1, False)  panda = Animal("Chad", 7, True)  print zebra.name, zebra.age, zebra.is\_hungry  print giraffe.name, giraffe.age, giraffe.is\_hungry  print panda.name, panda.age, panda.is\_hungry |
| Jeffrey 2 True  Bruce 1 False  Chad 7 True |

# Class Scope

Another important aspect of Python classes is scope. The scope of a variable is the context in which it's visible to the program.

It may surprise you to learn that not all variables are accessible to all parts of a Python program at all times. When dealing with classes, you can have variables that are available everywhere (global variables), variables that are only available to members of a certain class (member variables), and variables that are only available to particular instances of a class (instance variables).

The same goes for functions: some are available everywhere, some are only available to members of a certain class, and still others are only available to particular instance objects.

**Instructions**

**1.**

Check out the code in the editor. Note that each individual animal gets its own name and age (since they're all initialized individually), but they all have access to the member variable is\_alive, since they're all members of the Animal class. Click Run to see the output!

|  |
| --- |
| class Animal(object):  """Makes cute animals."""  is\_alive = True  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age  zebra = Animal("Jeffrey", 2)  giraffe = Animal("Bruce", 1)  panda = Animal("Chad", 7)  print zebra.name, zebra.age, zebra.is\_alive  print giraffe.name, giraffe.age, giraffe.is\_alive  print panda.name, panda.age, panda.is\_alive |
| Jeffrey 2 True  Bruce 1 True  Chad 7 True |

# A Methodical Approach

When a class has its own functions, those functions are called methods. You've already seen one such method: \_\_init\_\_(). But you can also define your own methods!

**Instructions**

**1.**

Add a method, description, to your Animal class. Using two separate print statements, it should print out the name and age of the animal it's called on. Then, create an instance of Animal, hippo (with whatever name and age you like), and call its description method.

Hint

Remember to pass self as an argument to description. Otherwise, printing self.name and self.agewon't work, since Python won't know which self(that is, which object) you're talking about!

Your method should look something like this:

def description(self): print self.name print self.age

After that, all you need to do is create a hippo and call its description method with hippo.description()!

|  |
| --- |
| class Animal(object):  """Makes cute animals."""  is\_alive = True  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age  # Add your method here!  def description(self):  print self.name  print self.age  hippo = Animal("Jack","99")  hippo.description() |
| Jack  99 |

**They're Multiplying!**

A class can have any number of **member variables**. These are variables that are available to all members of a class.

hippo = Animal("Jake", 12) cat = Animal("Boots", 3) print hippo.is\_alive hippo.is\_alive = False print hippo.is\_alive print cat.is\_alive

1. In the example above, we create two instances of an Animal.
2. Then we print out True, the default value stored in hippo's is\_alive member variable.
3. Next, we set that to False and print it out to make sure.
4. Finally, we print out True, the value stored in cat's is\_alivemember variable. We only changed the variable in hippo, not in cat.

Let's add another member variable to Animal.

**Instructions**

**1.**

After line 3, add a second member variable called health that contains the string "good".

Then, create two new Animals: sloth and ocelot. (Give them whatever names and ages you like.)

Finally, on three separate lines, print out the health of your hippo, sloth, and ocelot.

Hint

You can add your member variable right under is\_alive, like so:

is\_alive = True health = "good"

You can print out your hippo's health with

print hippo.health

|  |
| --- |
| class Animal(object):  """Makes cute animals."""  is\_alive = True  health = "good"  def \_\_init\_\_(self, name, age):  self.name = name  self.age = age  # Add your method here!  def description(self):  print self.name  print self.age  hippo = Animal("Jack","99")  sloth = Animal("May", 11)  ocelot = Animal("Happy",30)  hippo.description()  print hippo.health  print sloth.health  print ocelot.health |
| Jack  99  good  good  good |

# It's Not All Animals and Fruits

Classes like Animal and Fruit make it easy to understand the concepts of classes and instances, but you probably won't see many zebras or lemons in real-world programs.

However, classes and objects are often used to model real-world objects. The code in the editor is a more realistic demonstration of the kind of classes and objects you might find in commercial software. Here we have a basic ShoppingCart class for creating shopping cart objects for website customers; though basic, it's similar to what you'd see in a real program.

**1.**

Create an instance of ShoppingCart called my\_cart. Initialize it with any values you like, then use the add\_itemmethod to add an item to your cart.

Hint

Since the ShoppingCart class has an \_\_init\_\_() method that takes a customer name, I'd create my cart like so:

my\_cart = ShoppingCart("Eric")

Calling the add\_item()method might then be:

my\_cart.add\_item("Ukelele", 10)

|  |
| --- |
| class ShoppingCart(object):  """Creates shopping cart objects  for users of our fine website."""  items\_in\_cart = {}  def \_\_init\_\_(self, customer\_name):  self.customer\_name = customer\_name  def add\_item(self, product, price):  """Add product to the cart."""  if not product in self.items\_in\_cart:  self.items\_in\_cart[product] = price  print product + " added."  else:  print product + " is already in the cart."  def remove\_item(self, product):  """Remove product from the cart."""  if product in self.items\_in\_cart:  del self.items\_in\_cart[product]  print product + " removed."  else:  print product + " is not in the cart."    my\_cart = ShoppingCart("AMY")  my\_cart.add\_item("APPLE",10) |
| APPLE added. |

# Warning: Here Be Dragons

Inheritance is a tricky concept, so let's go through it step by step.

Inheritance is the process by which one class takes on the attributes and methods of another, and it's used to express an is-a relationship. For example, a Panda is a bear, so a Panda class could inherit from a Bear class. However, a Toyota is not a Tractor, so it shouldn't inherit from the Tractor class (even if they have a lot of attributes and methods in common). Instead, both Toyota and Tractor could ultimately inherit from the same Vehicle class.

**1.**

Check out the code in the editor. We've defined a class, Customer, as well as a ReturningCustomer class that inherits from Customer. Note that we don't define the display\_cart method in the body of ReturningCustomer, but it will still have access to that method via inheritance. Click Run to see for yourself!

|  |
| --- |
| class Customer(object):  """Produces objects that represent customers."""  def \_\_init\_\_(self, customer\_id):  self.customer\_id = customer\_id  def display\_cart(self):  print "I'm a string that stands in for the contents of your shopping cart!"  class ReturningCustomer(Customer):  """For customers of the repeat variety."""  def display\_order\_history(self):  print "I'm a string that stands in for your order history!"  monty\_python = ReturningCustomer("ID: 12345")  monty\_python.display\_cart()  monty\_python.display\_order\_history() |
| I'm a string that stands in for the contents of your shopping cart!  I'm a string that stands in for your order history! |

**Inheritance Syntax**

In Python, inheritance works like this:

class DerivedClass(BaseClass): # code goes here

where DerivedClass is the new class you're making and BaseClass is the class from which that new class inherits.

**Instructions**

**1.**

On lines 1-4, we've created a class named Shape.

* Create your own class, Triangle, that inherits from Shape, like this:

class Triangle(Shape): # code goes here

* Inside the Triangle class, write an \_\_init\_\_()function that takes four arguments: self, side1, side2, and side3.
* Inside the \_\_init\_\_()function, set self.side1 = side1, self.side2 = side2, and self.side3 = side3.

|  |
| --- |
| * class Shape(object): * """Makes shapes!""" * def \_\_init\_\_(self, number\_of\_sides): * self.number\_of\_sides = number\_of\_sides * # Add your Triangle class below! * class Triangle(Shape): * def \_\_init\_\_(self, side1, side2, side3): * self.side1 = side1 * self.side2 = side2 * self.side3 = side3 |

# Override!

Sometimes you'll want one class that inherits from another to not only take on the methods and attributes of its parent, but to override one or more of them.

class Employee(object): def \_\_init\_\_(self, name): self.name = name def greet(self, other): print "Hello, %s" % other.name class CEO(Employee): def greet(self, other): print "Get back to work, %s!" % other.name ceo = CEO("Emily") emp = Employee("Steve") emp.greet(ceo) # Hello, Emily ceo.greet(emp) # Get back to work, Steve!

Rather than have a separate greet\_underling method for our CEO, we override (or re-create) the greetmethod on top of the base Employee.greet method. This way, we don't need to know what type of Employee we have before we greet another Employee.

**Instructions**

**1.**

Create a new class, PartTimeEmployee, that inherits from Employee.

Give your derived class a calculate\_wage method that overrides Employee's. It should take self and hours as arguments.

Because PartTimeEmployee.calculate\_wage overrides Employee.calculate\_wage, it still needs to set self.hours = hours.

It should return the part-time employee's number of hoursworked multiplied by 12.00(that is, they get $12.00 per hour instead of $20.00).

|  |
| --- |
| class Employee(object):  """Models real-life employees!"""  def \_\_init\_\_(self, employee\_name):  self.employee\_name = employee\_name  def calculate\_wage(self, hours):  self.hours = hours  return hours \* 20.00  # Add your code below!  class PartTimeEmployee(Employee):  def calculate\_wage(self, hours):  self.hours = hours  return hours \* 12.00 |

**This Looks Like a Job For...**

On the flip side, sometimes you'll be working with a derived class (or *subclass*) and realize that you've overwritten a method or attribute defined in that class' base class (also called a *parent* or *superclass*) that you actually need. Have no fear! You can directly access the attributes or methods of a superclass with Python's built-in super call.

The syntax looks like this:

class Derived(Base): def m(self): return super(Derived, self).m()

Where m() is a method from the base class.

**1.**

First, inside your PartTimeEmployee class:

* Add a new method called full\_time\_wage with the arguments self and hours.
* That method should return the result of a super call to the calculate\_wage method of PartTimeEmployee's parent class. Use the example above for help.

Then, after your class:

* Create an instance of the PartTimeEmployee class called milton. Don't forget to give it a name.
* Finally, print out the result of calling his full\_time\_wage method. You should see his wage printed out at $20.00 per hour! (That is, for 10hours, the result should be 200.00.)

Hint

You super call should look something like this:

def full\_time\_wage(self, hours): return super(PartTimeEmployee, self).method(args)

Where method is the method you want (calculate\_wage) and args are the arguments that method takes.

|  |
| --- |
| class Employee(object):  """Models real-life employees!"""  def \_\_init\_\_(self, employee\_name):  self.employee\_name = employee\_name  def calculate\_wage(self, hours):  self.hours = hours  return hours \* 20.00  # Add your code below!  class PartTimeEmployee(Employee):  def calculate\_wage(self, hours):  self.hours = hours  return hours \* 12.00    def full\_time\_wage(self, hours):  return super(PartTimeEmployee, self).calculate\_wage(hours)  milton = PartTimeEmployee('Milton')  print milton.full\_time\_wage(10) |
| 200.0 |

# Class Basics

First things first: let's create a class to work with.

**Instructions**

**1.**

Create a class, Triangle. Its \_\_init\_\_() method should take self, angle1, angle2, and angle3 as arguments. Make sure to set these appropriately in the body of the \_\_init\_\_() method (see the Hint for more).

Hint

Make sure your Triangleinherits from object. Remember, class syntax looks like this:

class ClassName(object): def \_\_init\_\_(args): # Set self.args = args

|  |
| --- |
| class Triangle(object):  def \_\_init\_\_(self, angle1, angle2, angle3):  self.angle1 = angle1  self.angle2 = angle2  self.angle3 = angle3 |

**Class It Up**

Great! Now let's add a member variable and a method to our class.

**1.**

Inside the Triangle class:

* Create a variable named number\_of\_sides and set it equal to 3.
* Create a method named check\_angles. The sum of a triangle's three angles should return True if the sum of self.angle1, self.angle2, and self.angle3 is equal 180, and False otherwise.

Hint

The check\_angles method should look something like this:

def check\_angles(self): if (self.angle1 + self.angle2 + self.angle3 == 180): return True else: return False

|  |
| --- |
| class Triangle(object):  number\_of\_sides = 3  def \_\_init\_\_(self, angle1, angle2, angle3):  self.angle1 = angle1  self.angle2 = angle2  self.angle3 = angle3  def check\_angles(self):  sum = self.angle1 + self.angle2 + self.angle3  if sum == 180:  return True  else:  return False |

# Instantiate an Object

Let's go ahead and create an instance of our Triangle class.

**1.**

Create a variable named my\_triangle and set it equal to a new instance of your Triangle class. Pass it three angles that sum to 180 (e.g. 90, 30, 60).

Print out my\_triangle.number\_of\_sides

Print out my\_triangle.check\_angles()

Hint

Remember, we can instantiate an object like so:

instance = Class(args)

Where args are the arguments \_\_init\_\_() takes, not including self.

|  |
| --- |
| class Triangle(object):  number\_of\_sides = 3  def \_\_init\_\_(self, angle1, angle2, angle3):  self.angle1 = angle1  self.angle2 = angle2  self.angle3 = angle3  def check\_angles(self):  sum = self.angle1 + self.angle2 + self.angle3  if sum == 180:  return True  else:  return False  my\_triangle = Triangle(90, 30, 60)  print my\_triangle.number\_of\_sides  print my\_triangle.check\_angles() |
| 3  True |

# Inheritance

Finally, let's create an Equilateralclass that inherits from our Triangleclass. (An equilateral triangle is a triangle whose angles are all 60˚, which also means that its three sides are equal in length.)

**Instructions**

**1.**

Create a class named Equilateral that inherits from Triangle.

Inside Equilateral, create a member variable named angle and set it equal to 60.

Create an \_\_init\_\_() function with only the parameter self, and set self.angle1, self.angle2, and self.angle3equal to self.angle (since an equilateral triangle's angles will always be 60˚).

Hint

Remember, inheritance looks like this:

class DerivedClass(BaseClass): # Your code here

where DerivedClass is the new class you're making, and BaseClass is the class it inherits from.

|  |
| --- |
| class Triangle(object):  number\_of\_sides = 3  def \_\_init\_\_(self, angle1, angle2, angle3):  self.angle1 = angle1  self.angle2 = angle2  self.angle3 = angle3    def check\_angles(self):  if (self.angle1 + self.angle2 + self.angle3) == 180:  return True  else:  return False    class Equilateral(Triangle):  angle = 60  def \_\_init\_\_(self):  self.angle1 = self.angle  self.angle2 = self.angle  self.angle3 = self.angle |