Chapter 5

Object-Oriented Programming in Maya

CHAPTER OUTLINE

BY THE END OF THIS CHAPTER, YOU WILL BE ABLE TO:

- Compare and contrast object-oriented and procedural programming.
- Describe what objects are.
- Create your own custom class.
- Define attributes.
- Distinguish data attributes and methods.
- Compare and contrast instance attributes and class attributes.
- Distinguish the @staticmethod and @classmethod decorators.

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00005-3

- Describe and implement inheritance.
- Locate more information about object-oriented programming.
- Compare and contrast the pymel and cmds modules.
- Explain the underlying mechanics of the pymel module.
- Describe the benefits and disadvantages of the pymel module.
- Install the pymel module if needed.
- Locate online documentation for the pymel module.
- Use pymel methods to take advantage of OOP in your daily work.

As you have seen up to this point, Python's basic language features can work in conjunction with Maya commands to build a variety of useful functions and tools. Nevertheless, because this approach to Python programming differs little from MEL, the advantages of Python may not yet be immediately clear.

In Chapter 4, we introduced the concept of modules, and examined the many ways in which they are more powerful than ordinary MEL scripts. Part of the reason modules are so useful is because they allow you to encapsulate functions and variables, called attributes, into namespaces. Although we did not explicitly point out the relationship, modules are objects, just like any other piece of data in Python (including integers, strings, lists, dictionaries, and even functions themselves). In fact, Python is a fully object-oriented language, and most of its built-in types have attributes and functions that you can access much like those in modules.

In this chapter, we take a high-level look at the two programming paradigms available when using Python with Maya: object-oriented programming and procedural programming. After discussing these concepts, we will thoroughly examine the heart of object-oriented programming: objects. To better understand objects, we will create a custom Python class and use it to explore many of the basic components of classes, including data attributes, class attributes, and methods. We then introduce the concept of inheritance by creating a new class derived from our first one. Thereafter, we turn back to Maya to compare and contrast the maya.cmds module with an alternative, object-oriented suite: pymel. We conclude with some basic examples to introduce the pymel module and illustrate its unique features.

OBJECT-ORIENTED VERSUS PROCEDURAL PROGRAMMING

While MEL developers may be new to the concept, many programmers have at least heard of object-oriented programming (OOP). To some, it simply means programming with "classes." But what does that actually

mean, and what are the benefits of these "classes" and "objects"? To answer these questions, we need to compare and contrast OOP with procedural programming, another widely used programming paradigm.

For the sake of this discussion, think of a program as a means of operating on a data set to produce some result. In procedural programming, a program is made up of variables and functions designed to operate on data structures. Most Maya developers are probably familiar with the concepts behind procedural programming from MEL, as in the following hypothetical code snippet.

```
// convert a string to uppercase in MEL
string $yourName = "AnnieCat";
string $newName = `toupper($yourName)`;
print($newName);
// prints ANNIECAT //
```

This example perfectly illustrates the main tenet of procedural programming: *Data and the code that processes the data are kept separate*. The variables `$yourName` and `$newName` are simply collections of data, with no inherent features or functionality available to developers. Processing the data requires functions and procedures external to the strings, in this case, the MEL function **toupper()**.

A clue as to the nature of OOP is contained in the words "object-oriented." OOP refers to a programming paradigm that relies on objects as opposed to procedures. To understand this principle, consider the previous MEL example translated into Python.

```
# convert a string to uppercase in Python
your_name = 'Harper';
new_name = your_name.upper();
print(new_name);
# prints HARPER
```

Here you can see that the uppercasing function **upper()** is associated with the variable `your_name`. While we pointed out in Chapter 2 that variables in Python are objects with a value, type, and identity, we didn't really explore this principle further. *An object can be thought of as an encapsulation of both data and the functionality required to act on them*.

So if a procedural program is defined as a collection of variables and functions that operate on a data set, an object-oriented program can be defined as a collection of classes that contain both data and functionality. The term *object* refers to a unique occurrence of a class, an instance. While everything in Python is represented internally as an object, we will explore how you can define your own types of objects.

Basics of Class Implementation in Python

In addition to the set of built-in objects, Python also allows developers to create their own objects by defining *classes*. The basic mechanism to do so is the **class** keyword, followed by the name of the class being defined. Proper coding convention dictates that a class name should be capital case (e.g., **SpaceMarine**, **DarkEldar**).

1. Execute the following lines to define a class called **NewClass**.

```
class NewClass:
    pass;
```

2. If you use the **dir()** function on this class, you can see that it has some attributes by default, which should be familiar.

```
dir(NewClass);
# Result: ['__doc__', '__module__']#
```

   Although this basic syntax is the minimum requirement for defining a class, it is referred to as an old-style or classic class. Up until Python 2.2, this syntax was the only one available. Python 2.2 introduced an alternate class declaration, which allows you to specify a parent from which the class should derive. It is preferable to derive a class from the built-in object type.

3. Execute the following lines to redefine the **NewClass** class.

```
class NewClass(object):
    pass;
```

4. If you use the **dir()** function now, you can see that the class inherits a number of other attributes.

```
dir(NewClass);
# Result: ['__class__', '__delattr__', '__dict__',
'__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__']#
```

Although it is not required to define a basic class by deriving from the object type, it is strongly recommended. We will talk more about the importance of inheritance later in this chapter.

Instantiation

Once you have defined a class, you can create an instance of that class by calling the class (using function notation) and binding its return value to a variable.

5. Execute the following lines to create three **NewClass** instances.

```
instance1 = NewClass();
instance2 = NewClass();
instance3 = NewClass();
```

These creation calls each return a valid, unique **NewClass** instance. Each instance is a separate immutable object. Recall that immutability means only that the value of the underlying data cannot change. As such, each instance is guaranteed to have a consistent identity over the course of its existence as well (and hence could be used as a key in a dictionary). However, using OOP, you can effectively mutate instances by using attributes.

ATTRIBUTES

Like modules, classes have attributes. However, attributes in classes are much more nuanced. An attribute may be a piece of data or a function, and it may belong to the class or to an instance of the class. The basic paradigm for creating an attribute is to define it in the class definition, as in the following example.

```
class NewClass():
    data_attribute = None;
    def function_attribute(*args): pass;
```

At this point, you could pass **NewClass** to the **dir()** function and see its attributes (note that we do not inherit from the object type, simply to have a shorter list of attributes for this example).

```
print(dir(NewClass));
"""
prints:
['__doc__', '__module__', 'data_attribute',
'function_attribute']
"""
```

However, Python also allows you to add attributes to a class on-the-fly, after you have defined the class. You could now execute the following lines to add another data attribute to the class and see the results.

```
NewClass.another_data_attribute = None;
print(dir(NewClass));
"""
prints:
['__doc__', '__module__', 'another_data_attribute',
'data_attribute', 'function_attribute']
"""
```

You can also add attributes to individual instances after creating them, which will not affect other instances of the class. The following example illustrates this process.

```
instance1 = NewClass();
instance1.instance_attribute = None;
instance2 = NewClass();
# only instance1 has instance_attribute
print('Instance 1:', dir(instance1));
print('Instance 2:', dir(instance2));
```

Remember that attributes can be functions. Consequently, you can assign the name of any function to an attribute. A function that is an attribute of a class is referred to as a *method*.

```
def foo(*args): print('foo');
NewClass.another_function_attribute = foo;
instance1.another_function_attribute();
instance2.another_function_attribute();
```

While attributes you add to an instance become available to that instance immediately, adding attributes to the class itself makes the attributes immediately available to all current and further instances of the same class.

Finally, it is worth noting that reexecuting a class definition will not affect existing instances of the class. Because of how Python's data model works, old instances are still associated with an object type somewhere else in memory.

```
class NewClass():
    just_one_attribute = None;
new_instance = NewClass();
print('New:', dir(new_instance));
"""
prints: ('New:', ['__doc__', '__module__',
'just_one_attribute'])
"""
print('Old:', dir(instance1));
"""
prints: ('Old:', ['__doc__', '__module__',
'another_data_attribute', 'another_function_attribute',
'data_attribute', 'function_attribute',
'instance_attribute'])
"""
```

In the following sections, we will explore some of the special distinctions between instance attributes and class attributes.

Data Attributes

While you can add data attributes in a number of ways, a common mechanism for defining them is to do so in the **__init__()** method, which is called when an instance is first created. Because Python automatically searches for this function as soon as an instance is created, attributes that are created in this way are instance attributes, and require that you access them from an instance.

1. Execute the following lines to define a **Human** class with an **__init__()** method, which initializes some basic data attributes based on input arguments.

```
class Human(object):
    def __init__(self, *args, **kwargs):
        self.first_name = kwargs.setdefault('first');
        self.last_name = kwargs.setdefault('last');
        self.height = kwargs.setdefault('height');
        self.weight = kwargs.setdefault('weight');
```

2. Now you can pass arguments to the class when you create an instance, and their values will be bound accordingly. Execute the following code to create three new **Human** instances, passing each one different keyword arguments.

```
me = Human(first='Seth', last='Gibson');
mom = Human(first='Jean', last='Gibson');
dad = Human(first='David', last='Gibson');
```

3. Execute the following lines to confirm that each instance's data attributes have been properly initialized.

```
print('me: ', me.first_name);
print('mom:', mom.first_name);
print('dad:', dad.first_name);
```

While developers coming from other languages may be tempted to refer to **__init__()** as a constructor function, it is in fact quite different from one. Though its parameter list specifies the syntax requirements for instantiation, the instance has already been constructed by the time **__init__()** is called. Arguments passed to the constructor call are passed directly to **__init__()**. Note that the parameters you specify for **__init__()** restrict the constructor to all of its syntactic requirements, including any constraints it may impose on positional arguments. Refer to Chapter 3 for a refresher on the myriad options for passing arguments to functions.

As you can see, the first argument passed to this method is called `self`, while the remaining arguments are those that are passed to the constructor

when the instance was created. The `self` name is used as a convention, and refers to the instance object calling this function. This argument is always passed to methods on objects. Consequently, all ordinary methods will pass the owning instance as the first argument, though we will examine some exceptions later when discussing class attributes.

4. Execute the following lines to print the results of the **dir()** function for the **Human** class and for an instance, `me`.

```
print('Class:   ', dir(Human));
print('Instance:', dir(me));
```

You should see at the end of the output list that an instance allows you to access attributes that do not exist as part of the class definition itself. Remember that you can define attributes in the class definition as well. While data attributes defined both ways may have unique values for any particular instance, those that exist in the class definition can be accessed without an instance of the class, as in the following hypothetical example.

```
import sys;
class NewClass():
    # exists in class definition
    data_attribute1 = 1;
    def __init__(self):
        # added to instance upon instantiation
        self.data_attribute2 = 2;
print(NewClass.data_attribute1);
try:
    print(NewClass.data_attribute2);
except AttributeError:
    print(sys.exc_info()[0]);
instance = NewClass();
print(instance.data_attribute1);
print(instance.data_attribute2);
```

We will discuss the importance of this difference later as we examine class attributes.

As you have seen, you can access attributes by using the dot (**.**) operator. You can both reassign and read an attribute's value by accessing the attribute on the instance.

5. Execute the following lines to print the **first_name** attribute on the `me` instance, and then reassign it and print the new value.

```
print(me.first_name);
me.first_name = 'S';
print(me.first_name);
```

While you are free to modify data on your instances using this paradigm, another, more useful technique is to use methods.

Methods

A method is a function existing in a class. Ordinarily its first argument is always the instance itself (for which we conventionally use the `self` name). Methods are also a type of attribute. For example, while **len()** is simply a built-in function that you can use with strings, **upper()** is a method that is accessible on string objects themselves. You can add further methods to your class definition by remembering that the first argument will be the instance of the class.

6. Execute the following lines to add a **bmi()** method to the **Human** class, which returns the instance's body mass index using metric computation.

```
def bmi(self):
    return self.weight / float(self.height)**2;
Human.bmi = bmi;
```

Assuming we are working in the metric system, you could now create an instance and call the **bmi()** method on it to retrieve the body mass index if `height` and `weight` are defined.

7. Execute the following lines to create a new **Human** instance and print its BMI. The result should be 22.79.

```
adam_mechtley = Human(
    first='Adam',
    last='Mechtley',
    height=1.85,
    weight=78
);
print('%.2f'%adam_mechtley.bmi());
```

As we indicated earlier in this chapter, methods are one of the primary tools that differentiate OOP from procedural programming. They allow developers to retrieve transformed data or alter data on an instance in sophisticated ways. Apart from the syntactic requirement that the first argument in a method definition be the instance owning the method, you are free to use any other argument syntax as you like, including positional arguments, keyword arguments, and so on.

__str__()

It is also possible to override inherited methods to achieve different functionality. For example, printing the current instance isn't especially helpful.

8. Execute the following line.

```
print(adam_mechtley);
```

The results simply show you the namespace in which the class is defined, the name of the class, and a memory address. Sample output may look something like the following line.

```
<__main__.Human object at 0x130108dd0>
```

In Python, when printing an object or passing it to the **str()** function, the object's **__str__()** method is invoked. You can override this inherited method to display more useful information.

9. Execute the following lines to override the **__str__()** method to print the first and last name of the **Human**.

```
def human_str(self):
    return '%s %s'%(
        self.first_name,
        self.last_name
    );
Human.__str__ = human_str;
```

10. Print one of the instances you previously created and you should see its first and last name.

```
print(mom);
# prints: Jean Gibson
```

As we described in Chapter 4 when discussing modules, Python does not really have the concept of privacy. Just as with modules, Python uses the double-underscore convention to suggest that certain attributes or methods, such as **__str__()**, be treated as internal. Nevertheless, you can reassign internal methods from any context by reassigning another function to the method name.

Note that when you override a method like **__str__()**, all instances will reflect the change. The reason for this behavior concerns the type of the **__str__()** method in the base class.

11. Execute the following line to view the type of the **__str__()** method in the object class.

```
print(type(object.__str__));
# prints: <type 'wrapper_descriptor'>
```

As you can see from the result, the type is given as wrapper_descriptor. The basic idea is that when **__str__()** is invoked, it is not calling a particular implementation on the instance itself, but the implementation defined in the class. For example, reassigning this method on an instance object has no effect.

12. Execute the following lines to try to reassign **__str__()** on the `adam_mechtley` instance. As you can see, there is no effect.

```
def foo(): return 'just some dude';
adam_mechtley.__str__ = foo;
print(adam_mechtley);
# prints: Adam Mechtley
```

On the other hand, overriding ordinary instancemethods will only affect the particular instance that gets the new assignment.

13. Execute the following line to print the type of the **bmi()** method on `adam_mechtley`. You should see the result instancemethod.

```
print(type(adam_mechtley.bmi));
```

14. Execute the following lines to create a new **Human** instance and define a new function that returns a witty string.

```
matt_mechtley = Human(
    first='Matt',
    last='Mechtley',
    height=1.73,
    weight=78
);
msg = 'Studies have shown that BMI is not indicative of
health.';
def wellActually(): return msg;
```

15. Execute the following lines to store the **bmi()** method on `adam_mechtley` in a variable, reassign the **bmi()** name on the `adam_mechtley` instance, and then print the results of **bmi()** for both instance objects.

```
old_bmi = adam_mechtley.bmi;
adam_mechtley.bmi = wellActually;
print(str(matt_mechtley), matt_mechtley.bmi());
print(str(adam_mechtley), adam_mechtley.bmi());
```

You can see from the results that the `matt_mechtley` instance properly returns a BMI value, while the `adam_mechtley` instance returns the value of the **wellActually()** function.

16. Execute the following lines to reassign the **bmi()** instance-method on `matt_mechtley` to that on `adam_mechtley`.

```
adam_mechtley.bmi = matt_mechtley.bmi
print(adam_mechtley.bmi());
```

As you can see, `adam_mechtley` is not in fact printing its own BMI (about 22.8), but is printing the BMI associated with `matt_mechtley` (about 26.06)! The reason is because the method name on the

`adam_mechtley` instance is pointing to the method on the `matt_mechtley` instance, and so the name `self` refers to that instance in the method body.

17. Execute the following line to reassign the original instancemethod to the `adam_mechtley` instance. As you can see from the output, the result is about 22.8, as it should be.

```
adam_mechtley.bmi = old_bmi;
print(adam_mechtley.bmi());
```

You may sometimes want to temporarily override the functionality of methods in your own work. In such cases, you would want to use this pattern to maintain a reference to the original method so you can replace it when you are done. If you fail to replace a method that you are manipulating for some reason (e.g., overriding **__str__()** on someone else's class for debugging purposes), you should always replace the original when you are done, as other code may rely on the original functionality.

__repr__()

You may have noticed in step 15 that we wrapped the names of the instances in a call to the **str()** function. The reason for doing so is that when a call is made to **print()** with multiple arguments, the **__str__()** method on the object is not actually called.

18. Execute the following line to create two new **Human** instances and try to print their names.

```
sis = Human(first='Ruth', last='Gibson');
baby_sis =Human(first='Emily', last='Gibson');
print(sis, baby_sis);
```

Unfortunately, the results that you see closely resemble the output you saw in step 8, before you overrode the **__str__()** method. Some parts of Python do not invoke the **__str__()** method for output, but instead use the **__repr__()** method. Convention dictates that this method should return a string that would allow the object to be reconstructed if passed to the **eval()** function, for example.

19. Execute the following lines to add a **__repr__()** method to the **Human** class.

```
def human_rep(self):
    return """Human(%s='%s', %s='%s', %s=%s,
    %s=%s)"""%(
        'first', self.first_name,
        'last', self.last_name,
        'height', self.height,
```

```
            'weight', self.weight
        );
    Human.__repr__ = human_rep;
```

20. Execute the following line to try to print the instances you created in step 18 again.

```
print(sis, baby_sis);
```

You should see something like the following output, indicating that the **__repr__()** method has been invoked.

```
(Human(first='Ruth', last='Gibson', height=None,
weight=None), Human(first='Emily', last='Gibson',
height=None, weight=None))
```

21. Because we have followed convention, **__repr__()** returns a statement that could be evaluated. You could leverage this functionality to create a duplicate. Execute the following lines to create a duplicate of `baby_sis`. Note that each of these objects has a unique identity.

```
twin = eval(baby_sis.__repr__());
print(baby_sis, twin);
print(id(baby_sis), id(twin));
```

There are many more built-in methods you can override. Consult Section 3.4 of Python Language Reference for a full listing.

Properties

Properties are special types of methods that can be accessed as though they were ordinary data attributes, and hence do not require function syntax (e.g., `print(instance.property_name)`). When you add a property in a class definition itself, you can add the appropriate decorator to the line prior to the method's definition. A *decorator* is a special, recognized word, prefixed with the @ symbol.

```
# for a read-only property
@property
def property_name(self):
    # return something here
```

Note that Python 2.6 and later (Maya 2010 and later) allows you to also use a decorator to specify that a property can be set by simply passing a value to it like a data attribute (e.g., `instance.property_name = value`).

```
# make the property settable
@property_name.setter
def property_name(self, value):
    # set some values here
```

Another option, compatible with all versions of Maya, is to use the **property()** function in conjunction with new style class syntax, which lets you specify a getter and a setter (as well as some further optional parameters). At minimum a property must have a getter, but other aspects are optional.

```
def getter(self):
    # return something here
def setter(self, value):
    # set some values here
Class.property_name = property(getter, setter);
```

22. Add a `full_name` property to the **Human** class as shown in the following lines.

```
def fn_getter(self):
    return '%s %s'%(self.first_name, self.last_name);
def fn_setter(self, val):
    self.first_name, self.last_name = val.split();
Human.full_name = property(fn_getter, fn_setter);
```

Note the special assignment syntax we use in the **fn_setter()** method to assign multiple variables at the same time, based on items in the sequence returned from the **split()** method.

23. Execute the following lines to create a new **Human** and use its property to get and set the first and last names using the `full_name` property.

```
big_sis = Human(first='Amanda', last='Gordon');
# notice that property is not invoked with parentheses
print(big_sis.full_name);
# prints: Amanda Gordon
# set a new value
big_sis.full_name = 'Amanda Gibson';
print(big_sis.full_name);
# prints: Amanda Gibson
```

Class Attributes

In addition to instance attributes, Python contains support for class attributes. *A class attribute can be accessed by using the class name, and does not require that you create an instance*. Class attributes can be useful for adding data or methods to a class that may be of interest to other functions or classes when you may not have a need to create an instance of the class. You can access them by using the dot operator with the name of the class.

As we pointed out earlier, class data attributes behave exactly like instance attributes on instances: changing the value on one instance has no effect on other instances. However, because they can be accessed without an

instance, you may use them in cases where you have no instance (e.g., when you are creating a new instance).

24. Execute the following lines to add two basic unit-conversion attributes to the **Human** class.

```
Human.kPoundsToKg = 0.4536;
Human.kFeetToMeters = 0.3048;
```

25. Execute the following lines to reconstruct a new **Human** instance using these class attributes to help construct it from American units (feet and pounds, respectively).

```
imperial_height = 6.083;
imperial_weight = 172;
adam_mechtley = Human(
    first='Adam',
    last='Mechtley',
    height=imperial_height*Human.kFeetToMeters,
    weight=imperial_weight*Human.kPoundsToKg
);
print('Height:', adam_mechtley.height);
print('Weight:', adam_mechtley.weight);
```

Because of Python's system of modules, you may find it unnecessary to store such values as part of a class definition, instead preferring to store them as attributes of a module, outside of the class definition. However, when you have several classes defined in a single module, as you often will when working with the Maya API, it can be helpful to further nest such attributes inside of classes, especially if their names might conflict. You will see this practice used frequently starting in Chapter 10 as we introduce plug-in development.

Static Methods and Class Methods

Python also includes support for some special types of methods known as static methods and class methods. *Static methods and class methods allow you to call a method associated with the class without requiring an instance of the class*. You can add these types of methods to your class definition using decorators on the line before a definition. They take the following basic forms.

```
class ClassName(object):
    @staticmethod
    some_static_method():
        # insert awesome code here
        pass;
    @classmethod
```

```
some_class_method(cls):
    # insert awesome code here
    pass;
```

At this point, you could call either of these functions using the following identical forms.

```
ClassName.some_static_method();
ClassName.some_class_method();
```

Note that the only difference between these two options is that a class method implicitly passes the class itself as the first argument, which allows you shorthand access to any class attributes (by allowing you to access them using `cls.attribute_name` rather than `ClassName.attribute_name` inside the method). It is also worth pointing out that, because these types of methods are called using the name of the class rather than on an instance, they do not pass an instance (`self`) as an argument. Otherwise, you are free to add whatever other arguments to them you like.

26. Execute the following lines to add a static method to the **Human** class that returns the taller person.

```
@staticmethod
def get_taller_person(human1, human2):
    if (human1.height > human2.height):
        return human1;
    else: return human2;
Human.get_taller_person = get_taller_person;
```

27. Execute the following lines to print the taller person.

```
taller = Human.get_taller_person(
    adam_mechtley, matt_mechtley
);
print(taller);
```

Class methods can be used to implement alternate creation methods for a class or in other situations where access to class data is required separate from an instance.

28. Execute the following lines to add a class method that creates a **Human** with some specified default parameters.

```
@classmethod
def create(cls):
    return cls(
        first='Adam',
        last='Mechtley',
        height=6.083*cls.kFeetToMeters,
        weight=172*cls.kPoundsToKg
    );
```

```
Human.create_adam = create;
dopplegaenger = Human.create_adam();
print(dopplegaenger);
```

As with ordinary class attributes, the value of such methods is in those situations when you want to expose some functionality without an instance. While you can exploit a class method to create alternate constructors, they are also handy in cases when you want to access class data attributes.

Human Class

For your benefit, we have printed the entire code for the **Human** class example below, as it would appear if you were to define it all at once.

```
class Human(object):
    kPoundsToKg = 0.4536;
    kFeetToMeters = 0.3048;
    def __init__(self, *args, **kwargs):
        self.first_name = kwargs.setdefault('first');
        self.last_name = kwargs.setdefault('last');
        self.height = kwargs.setdefault('height');
        self.weight = kwargs.setdefault('weight');
    def bmi(self):
        return self.weight / float(self.height)**2;
    @staticmethod
    def get_taller_person(human1, human2):
        if (human1.height > human2.height):
            return human1;
        else: return human2;
    @classmethod
    def create_adam(cls):
        return cls(
            first='Adam',
            last='Mechtley',
            height=6.083*cls.kFeetToMeters,
            weight=172*cls.kPoundsToKg
        );
    # Begin properties
    def fn_getter(self):
        return '%s %s'%(self.first_name, self.last_name)
    def fn_setter(self, val):
        self.first_name, self.last_name = val.split()
    full_name = property(fn_getter, fn_setter)
    # End properties
    # Alternate property defs for Maya 2010+
    """
    @property
```

```
    def full_name(self):
        return '%s %s'%(self.first_name, self.last_name);
    @full_name.setter
    def full_name(self, val):
        self.first_name, self.last_name = val.split();
    """
    def __str__(self):
        return self.full_name;
    def __repr__(self):
        return """Human(%s='%s', %s='%s', %s=%s,
        %s=%s)"""%(
            'first', self.first_name,
            'last', self.last_name,
            'height', self.height,
            'weight', self.weight
        );
```

INHERITANCE

Another critical concept in OOP is inheritance. Although we made some use of it in the previous example, we did not explore it in detail. At the high level, inheritance allows a new class to be derived from a parent class, allowing it to automatically inherit all of its parent's attributes. As we demonstrated briefly in the previous section, this concept also allows you to override functionality as needed in a descendant class.

For example, suppose you have implemented a class like **Human** and decide that you need a similar class, but only with marginal changes. One option might be to just add the additional required functionality to the original class, but the new functionality may be substantially different, or may be something that not all instances require.

With inheritance, rather than having to copy or reimplement features from the original class, you can simply tell Python to create a new class based on an existing class that preserves all the functionality of the original, while allowing new functionality to be introduced.

1. Ensure that you have the **Human** class defined in your current namespace. See the end of the previous section for details.
2. Execute the following code to implement a class called **MyKid** that inherits from the **Human** class presented in the previous section.

```
class MyKid(Human):
    def __init__(self, *args, **kwargs):
        Human.__init__(self, *args, **kwargs);
        self.mom = kwargs.setdefault('mom');
```

```
            self.dad = kwargs.setdefault('dad');
            self.siblings = [];
            if kwargs.setdefault('sibs') is not None:
                self.siblings.extend(
                    kwargs.setdefault('sibs')
                );
        def get_parents(self):
            return '%s, %s'%(
                self.mom.full_name,
                self.dad.full_name
            );
        def set_parents(self, value):
            self.mom, self.dad = value;
        parents = property(get_parents, set_parents);
```

There are two main points related to class derivation that merit discussion. The first point concerns the class declaration statement.

```
class MyKid(Human):
```

Rather than using the built-in object class, **MyKid** uses **Human** as the parent or base class. This syntax is the first step for ensuring that the child/derived class **MyKid** retains the functionality implemented in **Human**.

The second point to take note of is the class initializer for **MyKid**. This situation is one of the few instances where a class's **__init__**() method needs to be called directly.

```
def __init__(self, *args, **kwargs):
    Human.__init__(self, *args, **kwargs);
    self.mom = kwargs.setdefault('mom');
    self.dad = kwargs.setdefault('dad');
    self.siblings = [];
    if kwargs.setdefault('sibs') is not None:
        self.siblings.extend(
            kwargs.setdefault('sibs')
        );
```

If the base class's initializer is not called, the derived class's functionality may be a bit unpredictable. For example, a derived class will inherit its parent class's methods (as we saw in the previous section when inheriting from **object**). If these methods are not specifically implemented (overridden) in the child class, some of them may require the presence of an instance data attribute that is created in the base class's **__init__**() method, rather than in the base class's definition.

When any method is executed, inherited or otherwise, it is executed in the scope of the derived class. If an inherited method references specific data

attributes and the base class's initializer hasn't also been run in the scope of the derived class, the derived class will not be able to access those data attributes since they do not exist. As a matter of practice, it is always wise to first call the base class's **__init__()** method in the derived class's **__init__()** method, and then override any data attributes or methods as needed.

3. Execute the following line to confirm all of the attributes in the **MyKid** class. As you can see from the output, the class has inherited all of the attributes you defined in the **Human** class earlier, as well as all of the attributes it inherited from **object**.

```
print(dir(MyKid));
"""
prints: ['__class__', '__delattr__', '__dict__',
'__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'bmi', 'create_adam', 'full_name',
'get_parents', 'get_taller_person', 'kFeetToMeters',
'kPoundsToKg', 'parents', 'set_parents']
"""
```

4. Execute the following lines to create two **Human** instances to pass as arguments to a new **MyKid** instance, and then create the new **MyKid** instance.

```
mom = Human(first='Jean', last='Gibson');
dad = Human(first='David', last='Gibson');
me = MyKid(
    first='Seth', last='Gibson',
    mom=mom, dad=dad
);
```

5. Print the **MyKid** instance and note that the base class handles the **__str__()** implementation.

```
print(me);
# prints: Seth Gibson
```

   Although we do not do so here for the sake of economy, you will always want to override **__repr__()** in child classes to ensure they will properly return values that would allow them to be reconstructed as instances of the derived class.

Note also that the **siblings** attribute could store further instances of the **MyKid** class.

6. Execute the following code to use a list comprehension to populate the **siblings** attribute on the `me` instance with further **MyKid** instances.

```
me.siblings.extend(
    [MyKid(
        first=n,
        last='Gibson',
        mom=mom,
        dad=dad
    ) for n in ['Amanda', 'Ruth', 'Emily']]
);
```

7. Execute the following line to print each of the siblings and confirm they were properly created.

```
for s in me.siblings: print(s);
"""
prints:
Amanda Gibson
Ruth Gibson
Emily Gibson
"""
```

Another advantage of inheritance is that you could easily create another derived class with its own separate functionality, which may be completely unimportant for other classes.

8. Execute the following code to create another derived class, **SpaceMarine**.

```
class SpaceMarine(Human):
    def __init__(self):
        Human.__init__(self);
        self.weapon_skill = 4;
        self.ballistic_skill = 4;
        self.strength = 4;
        self.toughness = 4;
        self.wounds = 1;
        self.initiative = 4;
        self.attacks = 1;
        self.leadership = 8;
```

Deriving classes is a powerful yet flexible way to both reuse code and keep codebases organized. Developers interested in learning more on this topic are advised to read both the Python Language Reference, which includes information on both Python's data and execution models, and the FAQs included on Python's design, which give insight into many of the design decisions behind Python.

Armed with a working knowledge of classes in Python, we can now take a look at how you can use some of these paradigms in Maya using an alternate Python implementation to maya.cmds.

PROCEDURAL VERSUS OBJECT-ORIENTED PROGRAMMING IN MAYA

One of the shortcomings of maya.cmds identified by many developers is that maya.cmds does not present a properly object-oriented equivalent to MEL as was originally hoped for. Fortunately, a developer at Luma Pictures named Chad Dombrova, along with other developers from Luma, ImageMovers Digital, and a few other studios, created a set of tools called PyMEL, which seek to address this issue.

Since its initial public release, PyMEL has found its way into many game and film productions, including *Thor*, *True Grit*, *Halo*, *PlanetSide 2*, and *League of Legends*. In this section, we will take a look at how it works, learn basic use techniques and syntax, and compare and contrast the pymel and maya.cmds modules. In the final section of this chapter, we will finish by developing a small tool using the pymel module.

Installing PyMEL

The pymel package ships with versions of Maya from 2011 forward. Earlier versions require a separate install, which can be accomplished a couple of different ways:

- The zip archive can be downloaded from *http://code.google.com/p/pymel/downloads/list*.
- GitHub users can pull source files from *https://github.com/LumaPictures/pymel*.

If you needed to download pymel for your version of Maya, you can install it according to the documentation found at *http://www.luma-pictures.com/tools/pymel/docs/1.0/install.html*.

The most recent documentation can always be found on the Luma Pictures web site, *http://www.luma-pictures.com/tools/pymel/docs/1.0/index.html*.

Introduction to PyMEL

While there are some functions in the pymel package that act the same as their maya.cmds counterparts, its real advantages come from using proper object-oriented paradigms. To get started with the pymel package once it has been installed in your Python search path, it simply needs to be

imported. The core extension module is the primary gateway to working with PyMEL.

1. Open a new Maya scene and execute the following line to import the pymel.core module. It may take a moment to initialize.

```
import pymel.core as pm;
```

2. Querying the scene is done in a similar manner to maya.cmds. Execute the following line to get all of the transforms in the scene.

```
scene_nodes = pm.ls(type='transform');
```

3. If you inspect this list, however, you see the first major difference between maya.cmds and pymel. Execute the following line to print all of the items in the `scene_nodes` list.

```
print(scene_nodes)
"""
prints:
[nt.Transform(u'front'), nt.Transform(u'persp'), nt.
Transform(u'side'), nt.Transform(u'top')]
"""
```

4. Compare this result with the standard maya.cmds approach by executing the following lines.

```
import maya.cmds as cmds;
print(cmds.ls(type='transform'));
"""
prints:
[u'front', u'persp', u'side', u'top']
"""
```

PyNodes

While the `ls` command in the cmds module returns a list of Unicode strings, the PyMEL invocation returns a list of a custom class type, which encapsulates data related to a Maya **transform** node.

5. Execute the following line to print the type of the first item in the `scene_nodes` list.

```
print(type(scene_nodes[0]));
# prints <class 'pymel.core.nodetypes.Transform'>
```

This aspect is one of PyMEL's key features: the **PyNode** class. All commands in the pymel.core module that mimic maya.cmds syntax return **PyNode** objects instead of strings. The reason for this approach is that commands accessible in the pymel.core module, even those that are

syntactically equivalent to maya.cmds calls, are not simply wrapped. Most of these calls are derived upon importation or are wrapped manually. This technique allows for greater functionality; API integration (or as the developers call it, hybridization); increased ease of access; occasional refactoring of large commands into smaller, logical chunks as needed (such as the `file` command); and even function calls unique to PyMEL. We will demonstrate a few of these features at the end of the chapter in our PyMEL tool.

PyNode objects in and of themselves are an interesting development and could easily have a whole section devoted to them. In simplest terms, a **PyNode** object is a Python class wrapper for an underlying API object of the given scene object it represents. Each type of node in the scene is represented in the Maya API, and so the **PyNode** class is able to simplify interfaces to many API methods. This approach provides developers with some interesting functionality.

- Code can be written more pythonically, as **PyNode** attributes are accessed through proper attribute references, as opposed to external queries.
- **PyNode** objects incorporate API-wrapped methods that can sometimes be a bit faster than their maya.cmds counterparts.
- Because **PyNode** objects have unique identities, tracking them in code becomes much simpler and more reliable than when working with object names. If the name of an object represented by a **PyNode** changes in the scene, the **PyNode** object's identity remains intact.
- API-based operations tend to be faster than string-processing operations, which in some cases can speed up node and attribute comparisons.

PyMEL Features

Let's take a look at a few more examples of interacting with PyMEL. The following lines illustrate how it handles the creation and manipulation of objects. As you can see, the **PyNode** class allows you to modify attribute values on nodes in the scene using methods rather than `getAttr` and `setAttr` commands.

```
import pymel.core as pm;
# create a sphere and wrap it in a PyNode
my_sphere = pm.polySphere()[0];
# set an attribute
my_sphere.translateX.set(10);
my_sphere.scaleY.set(6);
# get an attribute
translate_x = my_sphere.translateX.get();
scale_y = my_sphere.scaleY.get();
```

```
# get a transform's shape
sphere_shape = my_sphere.getShape();
# connect attributes
my_cube = pm.polyCube()[0];
my_cube.translateX.connect(my_sphere.rotateY);
```

As you can see, PyMEL natively embraces many of the OOP paradigms we have covered in this chapter to perform common operations. You can also work with files quite easily. The PyMEL implementation breaks the `file` command into a few different functions instead of using a single command with numerous flags.

```
pm.newFile(force=True);
pm.saveAs('newSceneName.ma', force=True);
pm.openFile('oldSceneName.ma', force=True);
```

Although we will not talk about building a GUI with Maya commands until Chapter 7, it is worth noting that PyMEL can help simplify GUI creation, as it implements GUI commands as Python contexts using the **with** keyword.

```
with pm.window() as w:
    with pm.columnLayout():
        with pm.horizontalLayout() as h:
            pm.button()
            pm.button()
            pm.button()
            h.redistribute(1,5,2)
        with pm.horizontalLayout() as h2:
            pm.button()
            pm.button()
            pm.button()
            h2.redistribute(1,3,7)
"""horizontalLayout is a custom pymel ui widget derived from
frameLayout"""
```

These examples are merely the tip of the iceberg. To get a better idea of how PyMEL really works, we recommend porting a script from either MEL or maya.cmds to PyMEL as an example of not only how different the two are syntactically, but also as an example of how PyMEL offers different design possibilities with its object-oriented approach to Maya.

Advantages and Disadvantages

PyMEL definitely offers some advantages over maya.cmds, which may make it a worthwhile investment for projects of any size. Some advantages include:

- Experienced Python programmers may have a much easier time learning PyMEL due to its object-oriented nature.
- PyMEL syntax tends to be a bit cleaner and creates neater code.

- Speed is greater in some cases due to API hybridization.
- The pymel package is open source, meaning studios can pull their own branch and add their own features and fixes.

That said, PyMEL does present a few disadvantages compared to maya.cmds:

- PyMEL's object-oriented nature can present a learning curve to MEL programmers. Switching from MEL to Python and learning a new programming paradigm at the same time can be daunting.
- PyMEL is not very widely adopted yet. A small community does mean that sometimes getting questions answered is difficult. Nevertheless, the development team is always eager to answer questions online.
- In some cases, speed can be degraded. Processing large numbers of components, for instance, can be much slower using PyMEL depending on the specific operation.
- Autodesk's long-term roadmap for PyMEL is unclear.
- Because the pymel package is open source, it is possible (though rare) to get into a situation where a custom branch is quite divergent from the main one.

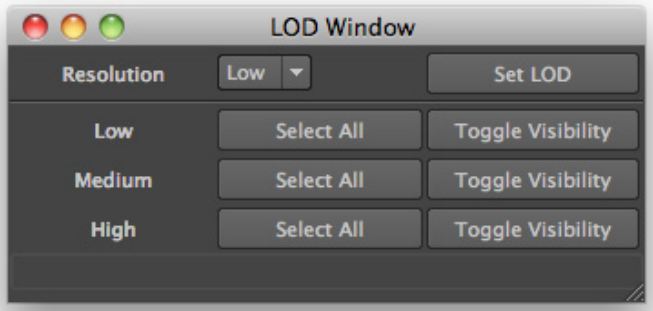
Overall, we encourage developers to give pymel at least a cursory evaluation. Because PyMEL comes preinstalled with versions of Maya from 2011 and later, you may have to do little more than import pymel.core if you are up-to-date. In the final section of this chapter, we will observe PyMEL in action by developing a small tool to manage scene objects based on attributes.

A PyMEL Example

The following example demonstrates how you might implement a tool that manages the level of detail tagging for a game using PyMEL. The basic premise is that objects can be selected and have an attribute applied to them that determines the level of detail. Once this tag has been applied, objects can be selected and shown or hidden.

1. Download the lodwindow.py module from the companion web site and save it somewhere in your Python search path. The lodwindow.py module contains a class, **LODWindow**, which has a **create()** method for drawing the window.
2. Execute the following lines to import the **LODWindow** class, create an instance of it, and then draw it using the **create()** method.

```
from lodwindow import LODWindow;
win = LODWindow();
win.create();
```



■ **FIGURE 5.1** The LOD window.

You should now see a window appear like that shown in Figure 5.1. The window has a dropdown menu for selecting a level of detail to apply to the currently selected objects using the Set LOD button. There is also a group of buttons that allows you to select all objects with a given level of detail, and show/hide all objects with a given level of detail.

3. Execute the following lines to create a grid of cylinders with different numbers of subdivisions.

```
import pymel.core as pm;
for res in range(4)[1:]:
    for i in range(3):
        cyl = pm.polyCylinder(sa=res*6, n='barrel1');
        cyl[0].tx.set(i*cyl[1].radius.get()*2);
        cyl[0].tz.set((res-1)*cyl[1].radius.get()*2);
```

4. Select all of the low-resolution cylinders in the back row, select the Low option from the LOD window dropdown menu, and press the Set LOD button. Repeat the same steps for the corresponding medium- and high-resolution cylinders.
5. Press the Select All button in the Medium row of the LOD window and notice that it selects all of the cylinders in the middle row (those with 12 subdivisions).
6. Press the Toggle Visibility button for the High row in the LOD window. Notice that all of the high-resolution cylinders disappear. Play around with the other buttons to your liking.

For the most part, the GUI's implementation is pretty straightforward. Because we have not yet covered all of the nuances of how GUIs work in Maya, however, we will only highlight a few of the key PyMEL features

at work in the module. After you have read Chapters 7 and 8, you may want to revisit this example to better appreciate its advantages.

The first item of note is the use of the **horizontalLayout()** function.

```
with pm.horizontalLayout():
    pm.text(label='Resolution')
    with pm.optionMenu() as self.res_menu:
        pm.menuItem(l='Low');
        pm.menuItem(l='Med');
        pm.menuItem(l='Hi');
    # ...
```

As we noted earlier, **horizontalLayout()** returns a custom PyMEL GUI widget derived from a form layout, which allows for advanced positioning of GUI controls. In this case, we have to do very little to get controls to display nicely.

Second, this example makes use of **CallBack()**, which is a function wrapper that provides a simple way of passing a callback and arguments to a GUI control.

```
set_res_btn = pm.button(
    label='Set LOD',
    command=pm.Callback(self.on_set_res_btn)
);
```

Third, this example makes use of the **setText()** method to manage the `status_line` text field, as opposed to using the `textField` command in edit mode.

```
if selected:
    self.tag_nodes(selected, res);
    self.status_line.setText(
        'Set selection to resolution %s'%res
    );
else:
    self.status_line.setText('No selection processed.');
```

Finally, this example makes use of node types for building object lists rather than passing in a string corresponding to the name of the type. Using this approach, developers can minimize errors due to typos, as an IDE can offer automatic code completion for these types.

```
poly_meshes = [
    i for i in pm.ls(
        type=pm.nt.Transform
    ) if type(i.getShape())==pm.nt.Mesh
];
```

While these PyMEL examples are relatively simple, we hope they have whet your appetite for some of the more complex possibilities available with PyMEL. We want to reiterate that if you are using Maya 2011 or later, because PyMEL is built in, evaluating it for your workflow is as simple as importing a module. Experiment, have fun, and consult the companion web site for information on places to communicate with other developers about this powerful tool.

CONCLUDING REMARKS

In this chapter, we summarized one of the final main advantages of the Python language itself by introducing object-oriented programming. You have seen that creating classes with a range of different types of attributes is simple, powerful, and incredibly flexible. You have also seen how the concept of inheritance can help you design an organized hierarchy of classes. Finally, you have seen how PyMEL offers a fully object-oriented alternative to Maya's built-in cmds module, enabling you to harness the power of OOP in your daily work. We hope you have been intrigued and give it a shot!

In all the chapters that follow, we make heavy use of classes to define GUIs, design Maya tools, and create API plug-ins, so it is essential that you feel comfortable with the basics presented in this chapter. Nevertheless, we will return to some high-level OOP concepts in Chapter 9 when we introduce the Maya API, so working through the examples in the intervening chapters will help give you more practice. Hopefully, because you have been taking advantage of some OOP features by using Python already, many of these concepts are already clear. You now have all the prerequisites to start building all kinds of great Maya tools, so it is time we start looking into some!