Data types as objects

This lab covers

- [Treating types as objects]
- [Using types]
- [Creating user-defined classes]
- [Understanding duck typing]
- [Using special method attributes]
- [Subclassing built-in types]

By now, you've learned the basic Python types as well as how to create your own data types using classes. For many languages, that would be pretty much it as far as data types are concerned. But Python is dynamically typed, meaning that types are determined at runtime, not at compile time. This fact is one of the reasons Python is so easy to use. It also makes it possible, and sometimes necessary, to compute with the types of objects (not just the objects themselves).

Types are objects, too

Fire up a Python session, and try out the following:

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

This example is the first time you've seen the built-in [type] function in Python. It can be applied to any Python object and returns the type of that object. In this example, the function tells you that [5] is an [int] (integer) and that [['hello', 'goodbye']] is a [list]---things that you probably already knew.

Of greater interest is the fact that Python returns objects in response to the calls to type; [<class 'int'>] and [<class 'list'>] are the screen representations of the returned objects. What sort of object is returned by a call of [type(5)]? You have an easy way of finding out. Just use [type] on that result:

```
>>> type_result = type(5)
>>> type(type_result)
<class 'type'>
```

The object returned by [type] is an object whose type happens to be [<class 'type'>]; you can call it a *type object*. A type object is another kind of Python object whose only outstanding feature is the confusion that its name sometime causes. Saying a type object is of type [<class 'type'>] has about the same degree of clarity as the old Abbott and Costello "Who's on First?" comedy routine.

Using types

Now that you know that data types can be represented as Python type objects, what can you do with them? You can compare them, because any two Python objects can be compared:

```
>>> type("Hello") == type("Goodbye")
True
>>> type("Hello") == type(5)
False
```

The types of ["Hello"] and ["Goodbye"] are the same (they're both strings), but the types of ["Hello"] and [5] are different. Among other things, you can use this technique to provide type checking in your function and method definitions.

Types and user-defined classes

The most common reason to be interested in the types of objects, particularly instances of user-defined classes, is to find out whether a particular object is an instance of a class. After determining that an object is of a particular type, the code can treat it appropriately. An example makes things much clearer. To start, define a couple of empty classes so as to set up a simple inheritance hierarchy:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
```

Now create an instance of class [B]:

```
>>> b = B()
```

As expected, applying the [type] function to [b] tells you that [b] is an instance of the class [B] that's defined in your current [__main__] namespace:

```
>>> type(b)
<class '__main__.B'>
```

You can also obtain exactly the same information by accessing the instance's special [__class__] attribute:

```
>>> b.__class__
<class '__main__.B'>
```

You'll be working with that class quite a bit to extract further information, so store it somewhere:

```
>>> b_class = b.__class__
```

Now, to emphasize that everything in Python is an object, prove that the class you obtained from [b] is the class you defined under the name [B]:

```
>>> b_class == B
True
```

In this example, you didn't need to store the class of [b]---you already had it---but I want to make clear that a class is just another Python object and can be stored or passed around like any Python object.

Given the class of [b], you can find the name of that class by using its [__name___] attribute:

```
>>> b_class.__name__
'B'
```

And you can find out what classes a class inherits from by accessing its [__bases__] attribute, which contains a tuple of all of its base classes:

```
>>> b_class.__bases__
(<class '__main__.A'>,)
```

Used together, [__class__], [__bases__], and [__name__] allow a full analysis of the class inheritance structure associated with any instance.

But two built-in functions provide a more user-friendly way of obtaining most of the information you usually need: [isinstance] and [issubclass]. The [isinstance] function is what you should use to determine whether, for example, a class passed into a function or method is of the expected type:

```
>>> class C:
...     pass
...
>>> class D:
...     pass
...
>>> class E(D):
...     pass
...
>>> x = 12
>>> c = C()
>>> d = D()
>>> e = E()
>>> isinstance(x, E)
False
>>> isinstance(c, E)                    1
False
>>> isinstance(e, E)
True
>>> isinstance(e, D)                    2
True
>>> isinstance(d, E)                    3
False
>>> y = 12
>>> isinstance(y, type(5))              4
True
```

The [issubclass] function is only for class types.

```
>>> issubclass(C, D)
False
>>> issubclass(E, D)
True
>>> issubclass(D, D)                    5
True
>>> issubclass(e.__class__, D)
True
```

For class instances, check against the class ***1***. [e] is an instance of class [D] because [E] inherits from [D] ***2***. But [d] isn't an instance of class [E] ***3***. For other types, you can use an example ***4***. A class is considered to be a subclass of itself ***5***.

Quick Check: Types

Suppose that you want to make sure that object [x] is a list before you try appending to it. What code would you use? What would be the difference between using [type()] and [isinstance()]? Would this be the look before you leap (LBYL) or easier to ask forgiveness than permission (EAFP) of programming? What other options might you have besides checking the type explicitly?

Duck typing

Using [type], [isinstance], and [issubclass] makes it fairly easy to make code correctly determine an object's or class's inheritance hierarchy. Although this process is easy, Python also has a feature that makes using objects even easier: duck typing. *Duck typing* (as in "If it walks like a duck and quacks like a duck, it probably *is* a duck") refers to Python's way of determining whether an object is the required type for an operation, focusing on an object's interface rather than its type. If an operation needs an iterator, for example, the object used doesn't need to be a subclass of any particular iterator or of any iterator at all. All that matters is that the object used as an iterator is able to yield a series of objects in the expected way.

By contrast, in a language like Java, stricter rules of inheritance are enforced. In short, duck typing means that in Python, you don't need to (and probably shouldn't) worry about type-checking function or method arguments and the like. Instead, you should rely on readable and documented code combined with thorough testing to make sure that an object "quacks like a duck" as needed.

Duck typing can increase the flexibility of well-written code and, combined with the more advanced object-oriented features, gives you the ability to create classes and objects to cover almost any situation.

What is a special method attribute?

A *special method attribute* is an attribute of a Python class with a special meaning to Python. It's defined as a method but isn't intended to be used directly as such. Special methods aren't usually directly invoked; instead, they're called automatically by Python in response to a demand made on an object of that class.

Perhaps the simplest example is the [__str__] special method attribute. If it's defined in a class, any time an instance of that class is used where Python requires a user-readable string representation of that instance, the [__str__] method attribute is invoked, and the value it returns is used as the required string. To see this attribute in action, define a class representing red, green, and blue (RGB) colors as a triplet of numbers, one each for red, green, and blue intensities. As well as defining the standard [__init__] method to initialize instances of the class, define a [__str__] method to return strings representing instances in a reasonably human-friendly format. Your definition should look something like this.

Listing 17.1. File color_module.py

```
class Color:
    def __init__(self, red, green, blue):
        self._red = red
        self._green = green
        self._blue = blue
    def __str__(self):
        return "Color: R={0:d}, G={1:d}, B={2:d}".format (self._red,
                                            self._green, self._blue)
```

If you put this definition into a file called color_module.py, you can load it and use it in the normal manner:

```
>>> from color_module import Color
>>> c = Color(15, 35, 3)
```

You can see the presence of the [__str__] special method attribute if you use [print] to print out [c]:

```
>>> print(c)
Color: R=15, G=35, B=3
```

Even though your [__str__] special method attribute hasn't been explicitly invoked by any of your code, it has nonetheless been used by Python, which knows that the [__str__] attribute (if present) defines a method to convert objects into user-readable strings. This characteristic is the defining one of special method attributes; it allows you to define functionality that hooks into Python in special ways. Among other things, special method attributes can be used to define classes whose objects behave in a fashion that's syntactically and semantically equivalent to lists or dictionaries. You could, for example, use this ability to define objects that are used in exactly the same manner as Python lists but that use balanced trees rather than arrays to store data. To a programmer, such objects would appear to be lists, but with faster inserts, slower iterations, and certain other performance differences that presumably would be advantageous in the problem at hand.

The rest of this lab covers longer examples using special method attributes. The chapter doesn't discuss all of Python's available special method attributes, but it does expose you to the concept in enough detail that you can easily use the other special attribute methods, all of which are defined in the standard library documentation for built-in types.

Making an object behave like a list

This sample problem involves a large text file containing records of people; each record consists of a single line containing the person's name, age, and place of residence, with a double semicolon ([::]) between the fields. A few lines from such a file might look like this:

```
.
.
.
John Smith::37::Springfield, Massachusetts
Ellen Nelle::25::Springfield, Connecticut
Dale McGladdery::29::Springfield, Hawaii
.
.
.
```

Suppose that you need to collect information about the distribution of ages of people in the file. There are many ways the lines in this file could be processed. Here's one way:

```
fileobject = open(filename, 'r')
lines = fileobject.readlines()
fileobject.close()
for line in lines:
    . . . do whatever . . .
```

That technique would work in theory, but it reads the entire file into memory at once. If the file were too large to be held in memory (and these files potentially are that large), the program wouldn't work.

Another way to attack the problem is this:

```
fileobject = open(filename, 'r')
for line in fileobject:
    . . . do whatever . . .
fileobject.close()
```

This code would get around the problem of having too little memory by reading in only one line at a time. It would work fine, but suppose that you wanted to make opening the file even simpler and that you wanted to get only the first two fields (name and age) of the lines in the file. You'd need something that could, at least for the purposes of a [for] loop, treat a text file as a list of lines but without reading the entire text file in at once.

The __getitem__ special method attribute

A solution is to use the [__getitem__] special method attribute, which you can define in any user-defined class, to enable instances of that class to respond to list access syntax and semantics. If [AClass] is a Python class that defines [__getitem__], and [obj] is an instance of that class, things like [x = obj[n]] and [for x in obj:] are meaningful; [obj] may be used in much the same way as a list.

Here's the resulting code (explanations follow):

```
class LineReader:
    def __init__(self, filename):
        self.fileobject = open(filename, 'r')
    def __getitem__(self, index):
        line = self.fileobject.readline()
        if line == "":
                  self.fileobject.close()
            raise IndexError

        else:
            return line.split("::")[:2]

for name, age in LineReader("filename"):
    . . . do whatever . . .
```

At first glance, this example may look worse than the previous solution because there's more code, and it's difficult to understand. But most of that code is in a class, which can be put into its own module, such as the [myutils] module. Then the program becomes

```
import myutils
for name, age in myutils.LineReader("filename"):
    . . . do whatever . . .
```

The [LineReader] class handles all the details of opening the file, reading in lines one at a time, and closing the file. At the cost of somewhat more initial development time, it provides a tool that makes working with one-record-per-line large text files easier and less error-prone. Note that Python already has several powerful ways to read files, but this example has the advantage that it's fairly easy to understand. When you get the idea, you can apply the same principle in many situations.

How it works

[LineReader] is a class, and the [__init__] method opens the named file for reading and stores the opened [fileobject] for later access. To understand the use of the [__getitem__] method, you need to know the following three points:

- [Any object that defines [__getitem__] as an instance method can return elements as though it were a list: all accesses of the form [object[i]] are transformed by Python into a method invocation of the form [object.__getitem__(i)], which is handled as a normal method invocation. It's ultimately executed as [__getitem__(object, i)], using the version of [__getitem__] defined in the class. The first argument of each call of [__getitem__] is the object from which data is being extracted, and the second argument is the index of that data.]

- [Because [for] loops access each piece of data in a list, one at a time, a loop of the form [for arg in sequence:] works by calling [__getitem__] over and over again, with sequentially increasing indexes. The [for] loop first sets [arg] to [sequence.__getitem__(0)], then to [sequence.__getitem__(1)], and so on.]
- [A [for] loop catches [IndexError] exceptions and handles them by exiting the loop. This process is how [for] loops are terminated when used with normal lists or sequences.]

The [LineReader] class is intended for use only with and inside a [for] loop, and the [for] loop always generates calls with a uniformly increasing index: [__getitem__(self, 0)], [__getitem__(self, 1)], [__getitem__(self, 2)], and so on. The previous code takes advantage of this knowledge and returns lines one after the other, ignoring the [index] argument.

With this knowledge, understanding how a [LineReader] object emulates a sequence in a [for] loop is easy. Each iteration of the loop causes the special Python attribute method [__getitem__] to be invoked on the object; as a result, the object reads in the next line from its stored [fileobject] and examines that line. If the line is nonempty, it's returned. An empty line means that the end of the file has been reached; the object closes the [fileobject] and raises the [IndexError] exception. [IndexError] is caught by the enclosing [for] loop, which then terminates.

Remember that this example is here for illustrative purposes only. Usually, iterating over the lines of a file by using the [for line in fileobject:] type of loop is sufficient, but this example does show how easy it is in Python to create objects that behave like lists or other types.

Quick Check: __getitem__

The example use of [__getitem__] is very limited and won't work correctly in many situations. What are some cases in which the implementation above will fail or work incorrectly?

Implementing full list functionality

In the previous example, an object of the [LineReader] class behaves like a list object only to the extent that it correctly responds to sequential accesses of the lines in the file it's reading from. You may wonder how this functionality can be expanded to make [LineReader] (or other) objects behave more like a list.

First, the [__getitem__] method should handle its index argument in some way. Because the whole point of the [LineReader] class is to avoid reading a large file into memory, it wouldn't make sense to have the entire file in memory and return the appropriate line. Probably the smartest thing to do would be to check that each index in a [__getitem__] call is one greater than the index from the previous [__getitem__] call (or is 0, for the first call of [__getitem__] on a [LineReader] instance) and to raise an error if this isn't the case. This practice would ensure that [LineReader] instances are used only in [for] loops as was intended.

More generally, Python provides several special method attributes relating to list behavior. [__setitem__] provides a way of defining what should be done when an object is used in the syntactic context of a list assignment, [obj[n] = val]. Some other special method attributes provide less-obvious list functionality, such as the [__add__] attribute, which enables objects to respond to the [+] operator and hence to perform their version of list concatenation. Several other special methods also need to be defined before a class fully emulates a list, but you can achieve complete list emulation by defining the appropriate Python special method attributes. The next section gives an example that goes farther toward implementing a full list emulation class.

Giving an object full list capability

[__getitem__] is one of many Python special function attributes that may be defined in a class to permit instances of that class to display special behavior. To see how special method attributes can be carried farther, effectively integrating new abilities into Python in a seamless manner, look at another, more comprehensive example.

When lists are used, it's common for any particular list to contain elements of only one type, such as a list of strings or a list of numbers. Some languages, such as C++, have the ability to enforce this restriction. In large programs, the ability to declare a list as containing a certain type of element can help you track down errors. An attempt to add an

element of the wrong type to a typed list results in an error message, potentially identifying a problem at an earlier stage of program development than would otherwise be the case.

Python doesn't have typed lists built in, and most Python coders don't miss them. But if you're concerned about enforcing the homogeneity of a list, special method attributes make it easy to create a class that behaves like a typed list. Here's the beginning of such a class (which makes extensive use of the Python built-in [type] and [isinstance] functions to check the type of objects):

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)                          1
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
                if not isinstance(element, self.type):
                    raise TypeError("Attempted to add an element of "
                                    "incorrect type to a typed list.")
        self.elements = initial_list[:]
```

The [example_element] argument defines the type that this list can contain by providing an example of the type of element ***1***.

The [TypedList] class, as defined here, gives you the ability to make a call of the form

```
x = TypedList ('Hello', ["List", "of", "strings"])
```

The first argument, ['Hello'], isn't incorporated into the resulting data structure at all. It's used as an example of the type of element the list must contain (strings, in this case). The second argument is an optional list that can be used to give an initial list of values. The [__init__] function for the [TypedList] class checks that any list elements, passed in when a [TypedList] instance is created, are of the same type as the example value given. If there are any type mismatches, an exception is raised.

This version of the [TypedList] class can't be used as a list, because it doesn't respond to the standard methods for setting or accessing list elements. To fix this problem, you need to define the [__setitem__] and [__getitem__] special method attributes. The [__setitem__] method is called automatically by Python any time a statement of the form [TypedListInstance[i] = value] is executed, and the [__getitem__] method is called any time the expression [TypedListInstance[i]] is evaluated to return the value in the *i*th slot of [TypedListInstance]. Here's the next version of the [TypedList] class. Because you'll be type-checking a lot of new elements, this function is abstracted out into the new private method [__check]:

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]
    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
```

```
    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
        return self.elements[i]
```

Now instances of the [TypedList] class look more like lists. The following code is valid, for example:

```
>>> x = TypedList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
```

The accesses of elements of [x] in the [print] statement are handled by [__getitem__], which passes them down to the list instance stored in the [TypedList] object. The assignments to [x[2]] and [x[3]] are handled by [__setitem__], which checks that the element being assigned into the list is of the appropriate type and then performs the assignment on the list contained in [self.elements]. The last line uses [__getitem__] to unpack the first five items in [x] and then pack them into the variables [a], [b], [c], [d], and e, respectively. The calls to [__getitem__] and [__setitem__] are made automatically by Python.

Completion of the [TypedList] class, so that [TypedList] objects behave in all respects like list objects, requires more code. The special method attributes [__setitem__] and [__getitem__] should be defined so that [TypedList] instances can handle slice notation as well as single item access. [__add__] should be defined so that list addition (concatenation) can be performed, and [__mul__] should be defined so that list multiplication can be performed. [__len__] should be defined so that calls of [len(TypedListInstance)] are evaluated correctly. [__delitem__] should be defined so that the [TypedList] class can handle [del] statements correctly. Also, an [append] method should be defined so that elements can be appended to [TypedList] instances by means of the standard list-style [append], as well as [insert] and [extend] methods.

Try this: Implementing list special methods

Try implementing the [__len__] and [__delitem__] special methods, as well as an [append] method.

Subclassing from built-in types

The previous example makes for a good exercise in understanding how to implement a listlike class from scratch, but it's also a lot of work. In practice, if you were planning to implement your own listlike structure along the lines demonstrated here, you might instead consider subclassing the list type or the [UserList] type.

Subclassing list

Instead of creating a class for a typed list from scratch, as you did in the previous examples, you can subclass the list type and override all the methods that need to be aware of the allowed type. One big advantage of this approach is that your class has default versions of all list operations because it's a list already. The main thing to keep in mind is that every type in Python is a class, and if you need a variation on the behavior of a built-in type, you may want to consider subclassing that type:

```
class TypedListList(list):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
```

```
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")

    def __setitem__(self, i, element):
        self.__check(element)
        super().__setitem__(i, element)

>>> x = TypedListList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

Note that all that you need to do in this case is implement a method to check the type of items being added and then tweak [__setitem__] to make that check before calling [list]'s regular [__setitem__] method. Other methods, such as [sort] and [del], work without any further coding. Overloading a built-in type can save a fair amount of time if you need only a few variations in its behavior, because the bulk of the class can be used unchanged.

Subclassing UserList

If you need a variation on a list (as in the previous examples), there's a third alternative: You can subclass the [UserList] class, a list wrapper class found in the [collections] module. [UserList] was created for earlier versions of Python when subclassing the list type wasn't possible, but it's still useful, particularly for the current situation, because the underlying list is available as the [data] attribute:

```
from collections import UserList
class TypedUserList(UserList):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
```

```
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.data[i] = element
    def __getitem__(self, i):
        return self.data[i]

>>> x = TypedUserList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

This example is much the same as subclassing [list], except that in the implementation of the class, the list of items is available internally as the [data] member. In some situations, having direct access to the underlying data structure can be useful. Also, in addition to [UserList], there are [UserDict] and [UserString] wrapper classes.

When to use special method attributes

As a rule, it's a good idea to be somewhat cautious with the use of special method attributes. Other programmers who need to work with your code may wonder why one sequence-type object responds correctly to standard indexing notation, whereas another doesn't.

My general guidelines are to use special method attributes in either of two situations:

- [If I have a frequently used class in my own code that behaves in some respects like a Python built-in type, I'll define such special method attributes as useful. This situation occurs most often with objects that behave like sequences in one way or another.]
- [If I have a class that behaves identically or almost identically to a built-in class, I may choose to define all of the appropriate special function attributes or subclass the built-in Python type and distribute the class. An example of the latter solution might be lists implemented as balanced trees so that access is slower but insertion is faster than with standard lists.]

These rules aren't hard-and-fast rules. It's often a good idea to define the [__str__] special method attribute for a class, for example, so that you can say [print(instance)] in debugging code and get an informative, nice-looking representation of your object printed to the screen.

Quick Check: Special method attributes and subclassing existing types

Suppose that you want a dictionary-like type that allows only strings as keys (maybe to make it work like a [shelf] object, as described in [chapter 13]). What options would you have for creating such a class? What would be the advantages and disadvantages of each option?

Summary

- [Python has the tools to do type checking as needed in your code, but by taking advantage of duck typing, you can write more flexible code that doesn't need to be as concerned with type checking.]
- [Special method attributes and subclassing built-in classes can be used to add listlike behavior to user-created classes.]
- [Python's use of duck typing, special method attributes, and subclassing makes it possible to construct and combine classes in a variety of ways.]