Message Passing and Dot Expressions in Python

Methods, which are defined in classes, and instance attributes, which are typically assigned in constructors, are the fundamental elements of object-oriented programming. These two concepts replicate much of the behavior of a dispatch dictionary in a message passing implementation of a data value. Objects take messages using dot notation, but instead of those messages being arbitrary string-valued keys, they are names local to a class. Objects also have named local state values (the instance attributes), but that state can be accessed and manipulated using dot notation, without having to employ `nonlocal` statements in the implementation.

The central idea in message passing was that data values should have behavior by responding to messages that are relevant to the abstract type they represent. Dot notation is a syntactic feature of Python that formalizes the message passing metaphor. The advantage of using a language with a built-in object system is that message passing can interact seamlessly with other language features, such as assignment statements. We do not require different messages to "get" or "set" the value associated with a local attribute name; the language syntax allows us to use the message name directly.

Dot expressions. The code fragment `tom_account.deposit` is called a *dot expression*. A dot expression consists of an expression, a dot, and a name:

```
<expression> . <name>
```

The `<expression>` can be any valid Python expression, but the `<name>` must be a simple name (not an expression that evaluates to a name). A dot expression evaluates to the value of the attribute with the given `<name>`, for the object that is the value of the `<expression>`.

The built-in function `getattr` also returns an attribute for an object by name. It is the function equivalent of dot notation. Using`getattr`, we can look up an attribute using a string, just as we did with a dispatch dictionary.

```
>>> getattr(tom_account, 'balance')
10
```

We can also test whether an object has a named attribute with `hasattr`.

```
>>> hasattr(tom_account, 'deposit')
True
```

The attributes of an object include all of its instance attributes, along with all of the attributes (including methods) defined in its class. Methods are attributes of the class that require special handling.

Method and functions. When a method is invoked on an object, that object is implicitly passed as the first argument to the method. That is, the object that is the value of the `<expression>` to the left of the dot is passed automatically as the first argument to the method named on the right side of the dot expression. As a result, the object is bound to the parameter `self`.

To achieve automatic `self` binding, Python distinguishes between *functions*, which we have been creating since the beginning of the text, and *bound methods*, which couple together a function and the object on which that method will be invoked. A bound method value is already associated with its first argument, the instance on which it was invoked, which will be named `self` when the method is called.

We can see the difference in the interactive interpreter by calling `type` on the returned values of dot expressions. As an attribute of a class, a method is just a function, but as an attribute of an instance, it is a bound method:

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>
```

These two results differ only in the fact that the first is a standard two-argument function with parameters `self` and `amount`. The second is a one-argument method, where the name `self` will be bound to the object named `tom_account` automatically when the method is called, while the parameter `amount` will be bound to the argument passed to the method. Both of these values, whether function values or bound method values, are associated with the same `deposit` function body.

We can call `deposit` in two ways: as a function and as a bound method. In the former case, we must supply an argument for the`self` parameter explicitly. In the latter case, the `self` parameter is bound automatically.

```
>>> Account.deposit(tom_account, 1001)    # The deposit function requires 2 arguments
1011
>>> tom_account.deposit(1000)              # The deposit method takes 1 argument
2011
```

The function `getattr` behaves exactly like dot notation: if its first argument is an object but the name is a method defined in the class, then `getattr` returns a bound method value. On the other hand, if the first argument is a class, then `getattr` returns the attribute value directly, which is a plain function.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#message-passing-and-dot-expressions