Guide to Python introspection

How to spy on your Python objects

Level: Introductory

Patrick O'Brien (pobrien@orbtech.com), Python programmer, Orbtech

01 Dec 2002

Introspection reveals useful information about your program's objects. Python, a dynamic, object-oriented programming language, provides tremendous introspection support. This article showcases many of its capabilities, from the most basic forms of help to the more advanced forms of inquisition.

What is introspection?

In everyday life, introspection is the act of self-examination. Introspection refers to the examination of one's own thoughts, feelings, motivations, and actions. The great philosopher Socrates spent much of his life in self-examination, encouraging his fellow Athenians to do the same. He even claimed that, for him, "the unexamined life is not worth living." (See Resources for links to more about Socrates.)

In computer programming, introspection refers to the ability to examine something to determine what it is, what it knows, and what it is capable of doing. Introspection gives programmers a great deal of flexibility and control. Once you've worked with a programming language that supports introspection, you may similarly feel that "the unexamined object is not worth instantiating."

This article introduces the introspection capabilities of the Python programming language. Python's support for introspection runs deep and wide throughout the language. In fact, it would be hard to imagine Python without its introspection features. By the end of this article you should be very comfortable poking inside the hearts and souls of your own Python objects.

We'll begin our exploration of Python introspection in the most general way possible, before diving into more advanced techniques. Some might even argue that the features we begin with don't deserve to be called "introspective." We'll have to agree that whether they fall under the umbrella of introspection or not is open to debate. For the purposes of this article, our only concern is finding the answers to interesting questions.

So let's begin our inquiry, using Python interactively. When we start Python from the command line, we enter the Python shell, where we can enter Python code and get an immediate response from the Python interpreter. (The commands listed in this article will execute properly using Python 2.2.2. You may get different results or errors if using an earlier version. You can download the latest version from the Python Web site [see Resources].)

Listing 1. Starting the Python interpreter in interactive mode

```
$ python
Python 2.2.2 (#1, Oct 28 2002, 17:22:19)
[GCC 3.2 (Mandrake Linux 9.0 3.2-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once you have Python running and are looking at the Python prompt (>>>), you may be wondering what words will be recognized by Python. Most programming languages have reserved words, or keywords, that have special meaning in that language, and Python is no exception. You may also have noticed that Python suggested we type `help` to get more information. Perhaps we can ask Python for some help about keywords.

Python's online help utility

Let's start by typing `help`, as suggested, and see if it gives us any clues about keywords:

Listing 2. Asking Python for help

```
>>> help
Type help() for interactive help, or help(object) for help about object.
```

Since we don't know what object might contain keywords, let's try `help()` without specifying any particular object:

Listing 3. Starting the help utility

```
>>> help()

Welcome to Python 2.2!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

We seem to be getting closer, now. Let's enter `keywords` at the help prompt:

Listing 4. Asking for help with keywords

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

and                 elif                global              or
assert              else                if                  pass
break               except              import              print
class               exec                in                  raise
continue            finally             is                  return
def                 for                 lambda              try
del                 from                not                 while

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

When we typed `help()`, we were greeted with a message and some instructions, followed by the help prompt. At the prompt, we entered `keywords` and were shown a list of Python keywords. Having gotten the answer to our question, we then quit the help utility, saw a brief farewell message, and were returned to the Python prompt.

As you can see from this example, Python's online help utility displays information on a variety of topics, or for a particular object. The help utility is quite useful, and does make use of Python's introspection capabilities. But simply using help doesn't reveal how help gets its information. And since the purpose of this article is to reveal all of Python's introspection secrets, we need to quickly go beyond the help utility.

Before we leave help, let's use it to get a list of available modules. Modules are simply text files containing Python code whose names end in `.py`. If we type `help('modules')` at the Python prompt, or enter `modules` at the help prompt, we'll see a long list of available modules, similar to the partial list shown below. Try it yourself to see what modules are available on your system, and to see why Python is considered to come with "batteries included."

Listing 5. Partial listing of available modules

```
>>> help('modules')

Please wait a moment while I gather a list of all available modules...

BaseHTTPServer      cgitb               marshal             sndhdr
Bastion             chunk               math                socket
```

```
CDROM               cmath               md5                 sre
CGIHTTPServer       cmd                 mhlib               sre_compile
Canvas              code                mimetools           sre_constants
        <...>
bisect              macpath             signal              xreadlines
cPickle             macurl2path         site                xxsubtype
cStringIO           mailbox             slgc (package)      zipfile
calendar            mailcap             smtpd
cgi                 markupbase          smtplib

Enter any module name to get more help.  Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".

>>>
```

The sys module

One module that provides insightful information about Python itself is the `sys` module. You make use of a module by importing the module and referencing its contents (such as variables, functions, and classes) using dot (.) notation. The `sys` module contains a variety of variables and functions that reveal interesting details about the current Python interpreter. Let's take a look at some of them. Again, we're going to run Python interactively and enter commands at the Python command prompt. The first thing we'll do is import the `sys` module. Then we'll enter the `sys.executable` variable, which contains the path to the Python interpreter:

Listing 6. Importing the sys module

```
$ python
Python 2.2.2 (#1, Oct 28 2002, 17:22:19)
[GCC 3.2 (Mandrake Linux 9.0 3.2-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.executable
'/usr/local/bin/python'
```

When we enter a line of code that consists of nothing more than the name of an object, Python responds by displaying a representation of the object, which, for simple objects, tends to be the value of the object. In this case, since the displayed value is enclosed in quotes, we get a clue that `sys.executable` is probably a string object. We'll look at other, more precise, ways to determine an object's type later, but simply typing the name of an object at the Python prompt is a quick and easy form of introspection.

Let's look at some other useful attributes of the `sys` module.

The `platform` variable tells us which operating system we are on:

The sys.platform attribute

```
>>> sys.platform
'linux2'
```

The current Python version is available as a string, and as a tuple (a tuple contains a sequence of objects):

Listing 8. The sys.version and sys.version_info attributes

```
>>> sys.version
'2.2.2 (#1, Oct 28 2002, 17:22:19) \n[GCC 3.2 (Mandrake Linux 9.0 3.2-1mdk)]'
>>> sys.version_info
(2, 2, 2, 'final', 0)
```

The `maxint` variable reflects the highest available integer value:

The sys.maxint attribute

```
>>> sys.maxint
```

```
2147483647
```

The `argv` variable is a list containing command line arguments, if any were specified. The first item, argv[0], is the path of the script that was run. When we run Python interactively this value is an empty string:

Listing 10. The sys.argv attribute

```
>>> sys.argv
['']
```

When we run another Python shell, such as PyCrust (see Resources for a link to more information on PyCrust), we see something like this:

Listing 11. The sys.argv attribute using PyCrust

```
>>> sys.argv[0]
'/home/pobrien/Code/PyCrust/PyCrustApp.py'
```

The `path` variable is the module search path, the list of directories in which Python will look for modules during imports. The empty string, `''`, in the first position refers to the current directory:

Listing 12. The sys.path attribute

```
>>> sys.path
['', '/home/pobrien/Code',
'/usr/local/lib/python2.2',
'/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-tk',
'/usr/local/lib/python2.2/lib-dynload',
'/usr/local/lib/python2.2/site-packages']
```

The `modules` variable is a dictionary that maps module names to module objects for all the currently loaded modules. As you can see, Python loads certain modules by default:

Listing 13. The sys.modules attribute

```
>>> sys.modules
{'stat': <module 'stat' from '/usr/local/lib/python2.2/stat.pyc'>,
'__future__': <module '__future__' from '/usr/local/lib/python2.2/__future__.pyc'>,
'copy_reg': <module 'copy_reg' from '/usr/local/lib/python2.2/copy_reg.pyc'>,
'posixpath': <module 'posixpath' from '/usr/local/lib/python2.2/posixpath.pyc'>,
'UserDict': <module 'UserDict' from '/usr/local/lib/python2.2/UserDict.pyc'>,
'signal': <module 'signal' (built-in)>,
'site': <module 'site' from '/usr/local/lib/python2.2/site.pyc'>,
'__builtin__': <module '__builtin__' (built-in)>,
'sys': <module 'sys' (built-in)>,
'posix': <module 'posix' (built-in)>,
'types': <module 'types' from '/usr/local/lib/python2.2/types.pyc'>,
'__main__': <module '__main__' (built-in)>,
'exceptions': <module 'exceptions' (built-in)>,
'os': <module 'os' from '/usr/local/lib/python2.2/os.pyc'>,
'os.path': <module 'posixpath' from '/usr/local/lib/python2.2/posixpath.pyc'>}
```

The keyword module

Let's return to our question about Python keywords. Even though help showed us a list of keywords, it turns out that some of help's information is hardcoded. The list of keywords happens to be hardcoded, which isn't very introspective after all. Let's see if we can get this information directly from one of the modules in Python's standard library. If we type `help('modules keywords')` at the Python prompt we see the following:

Listing 14. Asking for help on modules with keywords

```
>>> help('modules keywords')

Here is a list of matching modules.  Enter any module name to get more help.

keyword - Keywords (from "graminit.c")
```

So it appears as though the `keyword` module might contain keywords. By opening the `keyword.py` file in a text editor we can see that Python does make its list of keywords explicitly available as the `kwlist` attribute of the `keyword` module. We also see in the `keyword` module comments that this module is automatically generated based on the source code of Python itself, guaranteeing that its list of keywords is accurate and complete:

Listing 15. The keyword module's keyword list

```
>>> import keyword
>>> keyword.kwlist
['and', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'yield']
```

The dir() function

While it's relatively easy to find and import a module, it isn't as easy to remember what each module contains. And you don't always want to have to look at the source code to find out. Fortunately, Python provides a way to examine the contents of modules (and other objects) using the built-in `dir()` function.

The `dir()` function is probably the most well-known of all of Python's introspection mechanisms. It returns a sorted list of attribute names for any object passed to it. If no object is specified, `dir()` returns the names in the current scope. Let's apply `dir()` to our `keyword` module and see what it reveals:

Listing 16. The keyword module's attributes

```
>>> dir(keyword)
['__all__', '__builtins__', '__doc__', '__file__', '__name__',
'iskeyword', 'keyword', 'kwdict', 'kwlist', 'main']
```

And how about the `sys` module we looked at earlier?

Listing 17. The sys module's attributes

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'argv', 'builtin_module_names',
'byteorder', 'copyright', 'displayhook', 'exc_info', 'exc_type', 'excepthook',
'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
'getrecursionlimit', 'getrefcount', 'hexversion', 'last_traceback',
'last_type', 'last_value', 'maxint', 'maxunicode', 'modules', 'path',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
```

Without any argument, `dir()` returns names in the current scope. Notice how `keyword` and `sys` appear in the list, since we imported them earlier. Importing a module adds the module's name to the current scope:

Listing 18. Names in the current scope

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'keyword', 'sys']
```

We mentioned that the `dir()` function was a built-in function, which means that we don't have to import a module in order to use the function. Python recognizes built-in functions without our having to do anything. And now we see this name, `__builtins__`, returned by a call to `dir()`. Perhaps there is a connection here. Let's enter the name

`__builtins__` at the Python prompt and see if Python tells us anything interesting about it:

Listing 19. What is __builtins__?

```
>>> __builtins__
<module '__builtin__' (built-in)>
```

So `__builtins__` appears to be a name in the current scope that's bound to the module object named `__builtin__`. (Since modules are not simple objects with single values, Python displays information about the module inside angle brackets instead.) Note that if you look for a `__builtin__.py` file on disk you'll come up empty-handed. This particular module object is created out of thin air by the Python interpreter, because it contains items that are always available to the interpreter. And while there is no physical file to look at, we can still apply our `dir()` function to this object to see all the built-in functions, error objects, and a few miscellaneous attributes that it contains:

Listing 20. The __builtins__ module's attributes

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
'FloatingPointError', 'IOError', 'ImportError', 'IndentationError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'OverflowWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeError', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__', '__name__',
'abs', 'apply', 'bool', 'buffer', 'callable', 'chr', 'classmethod', 'cmp',
'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
'locals', 'long', 'map', 'max', 'min', 'object', 'oct', 'open', 'ord', 'pow',
'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
'setattr', 'slice', 'staticmethod', 'str', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

The `dir()` function works on all object types, including strings, integers, lists, tuples, dictionaries, functions, custom classes, class instances, and class methods. Let's apply `dir()` to a string object and see what Python returns. As you can see, even a simple Python string has a number of attributes:

Listing 21. String attributes

```
>>> dir('this is a string')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__repr__', '__rmul__', '__setattr__', '__str__',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind',
'rindex', 'rjust', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Try the following examples yourself to see what they return. Note that the # character marks the start of a comment. Everything from the start of the comment to the end of the line is ignored by Python:

Listing 22. Using dir() on other objects

```
dir(42)   # Integer (and the meaning of life)
dir([])   # List (an empty list, actually)
dir(())   # Tuple (also empty)
dir({})   # Dictionary (ditto)
dir(dir)  # Function (functions are also objects)
```

To illustrate the dynamic nature of Python's introspection capabilities, let's look at some examples using `dir()` on a custom class and some class instances. We're going to define our own class interactively, create some instances of the

class, add a unique attribute to only one of the instances, and see if Python can keep all of this straight. Here are the results:

Listing 23. Using dir() on custom classes, class instances, and attributes

```
>>> class Person(object):
...     """Person class."""
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def intro(self):
...         """Return an introduction."""
...         return "Hello, my name is %s and I'm %s." % (self.name, self.age)
...
>>> bob = Person("Robert", 35)   # Create a Person instance
>>> joe = Person("Joseph", 17)   # Create another
>>> joe.sport = "football"       # Assign a new attribute to one instance
>>> dir(Person)      # Attributes of the Person class
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'intro']
>>> dir(bob)         # Attributes of bob
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name']
>>> dir(joe)         # Note that joe has an additional attribute
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__', '__new__', '__reduce__', '__repr__',
'__setattr__', '__str__', '__weakref__', 'age', 'intro', 'name', 'sport']
>>> bob.intro()      # Calling bob's intro method
"Hello, my name is Robert and I'm 35."
>>> dir(bob.intro)   # Attributes of the intro method
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__get__',
'__getattribute__', '__hash__', '__init__', '__new__', '__reduce__',
'__repr__', '__setattr__', '__str__', 'im_class', 'im_func', 'im_self']
```

Documentation strings

One attribute you may have noticed in a lot of our `dir()` examples is the `__doc__` attribute. This attribute is a string containing the comments that describe an object. Python calls this a documentation string, or docstring, and here is how it works. If the first statement of a module, class, method, or function definition is a string, then that string gets associated with the object as its `__doc__` attribute. For example, take a look at the docstring for the `__builtins__` object. We'll use Python's `print` statement to make the output easier to read, since docstrings often contain embedded newlines (`\n`):

Listing 24. Module docstring

```
>>> print __builtins__.__doc__   # Module docstring
Built-in functions, exceptions, and other objects.

Noteworthy: None is the `nil' object; Ellipsis represents `...' in slices.
```

Once again, Python even maintains docstrings on classes and methods that are defined interactively in the Python shell. Let's look at the docstrings for our `Person` class and its `intro` method:

Listing 25. Class and method docstrings

```
>>> Person.__doc__          # Class docstring
'Person class.'
>>> Person.intro.__doc__    # Class method docstring
'Return an introduction.'
```

Because docstrings provide such valuable information, many Python development environments have ways of automatically displaying the docstrings for objects. Let's look at one more docstring, for the `dir()` function:

Listing 26. Function docstring

```
>>> print dir.__doc__   # Function docstring
dir([object]) -> list of strings

Return an alphabetized list of names comprising (some of) the attributes
of the given object, and of attributes reachable from it:

No argument:  the names in the current scope.
Module object:  the module attributes.
Type or class object:  its attributes, and recursively the attributes of
    its bases.
Otherwise:  its attributes, its class's attributes, and recursively the
    attributes of its class's base classes.
```

Interrogating Python objects

We've mentioned the word "object" several times, but haven't really defined it. An object in a programming environment is much like an object in the real world. A real object has a certain shape, size, weight, and other characteristics. And a real object is able to respond to its environment, interact with other objects, or perform a task. Computer objects attempt to model the objects that surround us in the real world, including abstract objects like documents and schedules and business processes.

Like real-world objects, several computer objects may share common characteristics while maintaining their own minor variations. Think of the books you see in a bookstore. Each physical copy of a book might have a smudge, or a few torn pages, or a unique identification number. And while each book is a unique object, every book with the same title is merely an instance of an original template, and retains most of the characteristics of the original.

The same is true about object-oriented classes and class instances. For example, every Python string is endowed with the attributes we saw revealed by the `dir()` function. And in a previous example, we defined our own `Person` class, which acted as a template for creating individual Person instances, each having its own name and age values, while sharing the ability to introduce itself. That's object-orientation.

In computer terms, then, objects are things that have an identity and a value, are of a certain type, possess certain characteristics, and behave in a certain way. And objects inherit many of their attributes from one or more parent classes. Other than keywords and special symbols (like operators, such as `+`, `-`, `*`, `**`, `/`, `%`, `<`, `>`, etc.) everything in Python is an object. And Python comes with a rich set of object types: strings, integers, floats, lists, tuples, dictionaries, functions, classes, class instances, modules, files, etc.

When you have an arbitrary object, perhaps one that was passed as an argument to a function, you may want to know a few things about that object. In this section we're going to show you how to get Python objects to answer questions such as:

- What is your name?
- What kind of object are you?
- What do you know?
- What can you do?
- Who are your parents?

Name

Not all objects have names, but for those that do, the name is stored in their `__name__` attribute. Note that the name is derived from the object, not the variable that references the object. The following example highlights that distinction:

Listing 27. What's in a name?

```
$ python
Python 2.2.2 (#1, Oct 28 2002, 17:22:19)
[GCC 3.2 (Mandrake Linux 9.0 3.2-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()               # The dir() function
['__builtins__', '__doc__', '__name__']
>>> directory = dir     # Create a new variable
>>> directory()         # Works just like the original object
['__builtins__', '__doc__', '__name__', 'directory']
>>> dir.__name__        # What's your name?
'dir'
>>> directory.__name__  # My name is the same
```

```
'dir'
>>> __name__             # And now for something completely different
'__main__'
```

Modules have names, and the Python interpreter itself is considered the top-level, or main, module. When you run Python interactively the local `__name__` variable is assigned a value of `'__main__'`. Likewise, when you execute a Python module from the command line, rather than importing it into another module, its `__name__` attribute is assigned a value of `'__main__'`, rather than the actual name of the module. In this way, modules can look at their own `__name__` value to determine for themselves how they are being used, whether as support for another program or as the main application executed from the command line. Thus, the following idiom is quite common in Python modules:

Listing 28. Testing for execution or import

```
if __name__ == '__main__':
    # Do something appropriate here, like calling a
    # main() function defined elsewhere in this module.
    main()
else:
    # Do nothing. This module has been imported by another
    # module that wants to make use of the functions,
    # classes and other useful bits it has defined.
```

Type

The `type()` function helps us determine whether an object is a string or an integer or some other kind of object. It does this by returning a type object, which can be compared to the types defined in the `types` module:

Listing 29. Am I your type?

```
>>> import types
>>> print types.__doc__
Define names for all type symbols known in the standard interpreter.

Types that are part of optional modules (e.g. array) are not listed.

>>> dir(types)
['BufferType', 'BuiltinFunctionType', 'BuiltinMethodType', 'ClassType',
'CodeType', 'ComplexType', 'DictProxyType', 'DictType', 'DictionaryType',
'EllipsisType', 'FileType', 'FloatType', 'FrameType', 'FunctionType',
'GeneratorType', 'InstanceType', 'IntType', 'LambdaType', 'ListType',
'LongType', 'MethodType', 'ModuleType', 'NoneType', 'ObjectType', 'SliceType',
'StringType', 'StringTypes', 'TracebackType', 'TupleType', 'TypeType',
'UnboundMethodType', 'UnicodeType', 'XRangeType', '__builtins__', '__doc__',
'__file__', '__name__']
>>> s = 'a sample string'
>>> type(s)
<type 'str'>
>>> if type(s) is types.StringType: print "s is a string"
...
s is a string
>>> type(42)
<type 'int'>
>>> type([])
<type 'list'>
>>> type({})
<type 'dict'>
>>> type(dir)
<type 'builtin_function_or_method'>
```

Identity

We said earlier that every object has an identity, a type, and a value. What's important to note is that more than one variable may refer to the exact same object, and, likewise, variables may refer to objects that look alike (having the same type and value), but have separate and distinct identities. This notion of object identity is particularly important when making changes to objects, such as appending an item to a list, as in the example below where the `blist` and `clist` variables both reference the same list object. As you can see in the example, the `id()` function returns the unique identifier for any given object:

Listing 30. The Bourne ...

```
>>> print id.__doc__
id(object) -> integer

Return the identity of an object.  This is guaranteed to be unique among
simultaneously existing objects.  (Hint: it's the object's memory address.)
>>> alist = [1, 2, 3]
>>> blist = [1, 2, 3]
>>> clist = blist
>>> clist
[1, 2, 3]
>>> blist
[1, 2, 3]
>>> alist
[1, 2, 3]
>>> id(alist)
145381412
>>> id(blist)
140406428
>>> id(clist)
140406428
>>> alist is blist    # Returns 1 if True, 0 if False
0
>>> blist is clist    # Ditto
1
>>> clist.append(4)   # Add an item to the end of the list
>>> clist
[1, 2, 3, 4]
>>> blist             # Same, because they both point to the same object
[1, 2, 3, 4]
>>> alist             # This one only looked the same initially
[1, 2, 3]
```

Attributes

We've seen that objects have attributes, and that the `dir()` function will return a list of these attributes. Sometimes, however, we simply want to test for the existence of one or more attributes. And if an object has the attribute in question, we often want to retrieve that attribute. These tasks are handled by the `hasattr()` and `getattr()` functions, as illustrated in this example:

Listing 31. Have an attribute; get an attribute

```
>>> print hasattr.__doc__
hasattr(object, name) -> Boolean

Return whether the object has an attribute with the given name.
(This is done by calling getattr(object, name) and catching exceptions.)
>>> print getattr.__doc__
getattr(object, name[, default]) -> value

Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.
When a default argument is given, it is returned when the attribute doesn't
exist; without it, an exception is raised in that case.
>>> hasattr(id, '__doc__')
1
>>> print getattr(id, '__doc__')
id(object) -> integer

Return the identity of an object.  This is guaranteed to be unique among
simultaneously existing objects.  (Hint: it's the object's memory address.)
```

Callables

Objects that represent potential behavior (functions and methods) can be invoked, or called. We can test an object's callability with the `callable()` function:

Listing 32. Can you do something for me?

```
>>> print callable.__doc__
callable(object) -> Boolean

Return whether the object is callable (i.e., some kind of function).
Note that classes are callable, as are instances with a __call__() method.
>>> callable('a string')
0
>>> callable(dir)
1
```

Instances

While the `type()` function gave us the type of an object, we can also test an object to determine if it is an instance of a particular type, or custom class, using the `isinstance()` function:

Listing 33. Are you one of those?

```
>>> print isinstance.__doc__
isinstance(object, class-or-type-or-tuple) -> Boolean

Return whether an object is an instance of a class or of a subclass thereof.
With a type as second argument, return whether that is the object's type.
The form using a tuple, isinstance(x, (A, B, ...)), is a shortcut for
isinstance(x, A) or isinstance(x, B) or ... (etc.).
>>> isinstance(42, str)
0
>>> isinstance('a string', int)
0
>>> isinstance(42, int)
1
>>> isinstance('a string', str)
1
```

Subclasses

We mentioned earlier that instances of a custom class inherit their attributes from the class. At the class level, a class may be defined in terms of another class, and will likewise inherit attributes in a hierarchical fashion. Python even supports multiple inheritance, meaning an individual class can be defined in terms of, and inherit from, more than one parent class. The `issubclass()` function allows us to find out if one class inherits from another:

Listing 34. Are you my mother?

```
>>> print issubclass.__doc__
issubclass(C, B) -> Boolean

Return whether class C is a subclass (i.e., a derived class) of class B.
>>> class SuperHero(Person):   # SuperHero inherits from Person...
...     def intro(self):       # but with a new SuperHero intro
...         """Return an introduction."""
...         return "Hello, I'm SuperHero %s and I'm %s." % (self.name, self.age)
...
>>> issubclass(SuperHero, Person)
1
>>> issubclass(Person, SuperHero)
0
>>>
```

Interrogation time

Let's wrap things up by putting together several of the introspection techniques we've covered in the last section. To do so, we're going to define our own function, `interrogate()`, which prints a variety of information about any object passed to it. Here is the code, followed by several examples of its use:

Listing 35. Nobody expects it

```
>>> def interrogate(item):
...     """Print useful information about item."""
...     if hasattr(item, '__name__'):
...         print "NAME:    ", item.__name__
...     if hasattr(item, '__class__'):
...         print "CLASS:   ", item.__class__.__name__
...     print "ID:      ", id(item)
...     print "TYPE:    ", type(item)
...     print "VALUE:   ", repr(item)
...     print "CALLABLE:",
...     if callable(item):
...         print "Yes"
...     else:
...         print "No"
...     if hasattr(item, '__doc__'):
```

```
...         doc = getattr(item, '__doc__')
...     doc = doc.strip()   # Remove leading/trailing whitespace.
...     firstline = doc.split('\n')[0]
...     print "DOC:     ", firstline
...
>>> interrogate('a string')     # String object
CLASS:    str
ID:       141462040
TYPE:     <type 'str'>
VALUE:    'a string'
CALLABLE: No
DOC:      str(object) -> string
>>> interrogate(42)             # Integer object
CLASS:    int
ID:       135447416
TYPE:     <type 'int'>
VALUE:    42
CALLABLE: No
DOC:      int(x[, base]) -> integer
>>> interrogate(interrogate)    # User-defined function object
NAME:     interrogate
CLASS:    function
ID:       141444892
TYPE:     <type 'function'>
VALUE:    <function interrogate at 0x86e471c>
CALLABLE: Yes
DOC:      Print useful information about item.
```

As you can see in the last example, our `interrogate()` function even works on itself. You can't get much more introspective than that.

Conclusion

Who knew that introspection could be so simple, and so rewarding? And yet, I must end here with a caution: do not mistake the results of introspection for wisdom. The experienced Python programmer knows that there is always more they do not know, and are therefore not wise at all. The act of programming produces more questions than answers. The only thing good about Python, as we have seen here today, is that it does answer one's questions. As for me, do not feel a need to compensate me for helping you understand these things that Python has to offer. Programming in Python is its own reward. All I ask from my fellow Pythonians is free meals at the public expense.

Resources

- The Python Web site is the starting point for all things Pythonic, including the official Python documentation.
- The Python newsgroup, comp.lang.python, is a great source of questions and answers.
- The Orbtech Web site contains a list of additional Python resources.
- PyCrust, the particularly introspective Python shell, is available on SourceForge.
- Wikipedia gives Socrates in a nutshell. You can also read about the trial of Socrates there.
- Read "The Camel and the Snake, or 'Cheat the Prophet': Open Source Development with Perl, Python, and DB2" for an overview of using Python and Perl with IBM DB2.

- Find more resources for Linux developers in the *developerWorks* Linux zone.

About the author

Patrick O'Brien is a Python programmer, consultant, and trainer. He is the author of PyCrust and a developer on the PythonCard project. He most recently lead the PyPerSyst team that ported Prevayler to Python, and continues to lead that project into interesting new territory. Learn more about Patrick and his work at the Orbtech Web site, or contact him at pobrien@orbtech.com.

Share this....

Digg this story del.icio.us Slashdot it!