

Interactive Interpreter

Boo has an interactive interpreter that is useful for testing out snippets of boo code.

To use it, run the booish.exe command line tool. You can also edit, compile, and debug if you install the [Boo AddIn For SharpDevelop](#).

Default Built-in Commands

describe(obj)

Type describe(obj), where obj is the name of some object, and the interpreter will list the methods and other members of that object.

load(assemblypath)

load(assemblypath) is useful for adding a reference to a dll for later code.

To load an assembly by name instead of file path, use the extended [import](#) syntax, i.e. "import System.Windows.Forms from System.Windows.Forms".

interpreter

You can reference the interpreter object itself as "interpreter". For example, you can turn default [Duck Typing](#) off by typing "interpreter.Ducky = false"

globals()

Prints out all the objects that the interpreter has stored in your session.

last value

The symbol "_" (underscore) refers to the last value returned by the interpreter. For example:

```
>>> 3 + 4
7
>>> _ * 2
14
```

dir(obj)

Similar to describe, dir spits out an array of all the type members contained by an object.

Other features

Support for multi-line code blocks

Just type the line beginning the code block (like "def mymethod():" or "class MyClass:", then return and you'll be able to indent the following lines just like in a regular boo script. Hit return twice to end the indented block.

line history

Hit the up arrow key to see previous lines you have typed in. Edit the line and hit Enter to resubmit it.

code completion

The sharpdevelop version of booish hooks into SharpDevelop's code completion engine.

Duck typing

Implicit [Duck Typing](#) is on by default. Useful for quicker coding, i.e., this code will run fine in the interpreter: (no need to declare "p as string")

```
>>>def doit(p):
...     print p.ToUpper()
...
>>>doit("a string")
A STRING
```

Advanced - implementing your own interpreter

See the source code under src/booish/ for how the interpreter is run.

You can override the Lookup, Declare, GetValue and SetValue methods in AbstractInterpreter to do things like intercepting name resolution or adding outside scopes to the interpreter. This can be useful for example when embedding the interpreter in a C# or boo app.

Here is a sample of using an outside object for the name resolution, by Rodrigo B. de Oliveira:

```
import System
import Boo.Lang.Interpreter from
Boo.Lang.Interpreter

class
ObjectInterpreter(AbstractInterpreter):
```

```
    _context as object

    [getter(Value)]
    _value as object

    def constructor(context):
        _context = context
        self.RememberLastValue = true

        override def Lookup(name as string):
            property =
                _context.GetType().GetProperty(name)
            return property.PropertyType if
                property is not null

            override def GetValue(name as
                string):
                return
                _context.GetType().GetProperty(name).GetValu
                e(
                    _context, null)

            override def SetLastValue(value):
                _value = value

            override def SetValue(name as
                string, value):
                raise
                InvalidOperationException()
```

```
        override def Declare(name as string,  
type as Type):  
            raise  
InvalidOperationException()  
  
class Person:  
    [property(FirstName)]  
    _fname as string = ""  
  
p = Person(FirstName: "Homer")
```

```
i = ObjectInterpreter(p)
i.Eval('"Hello, ${FirstName.ToUpper()}! "')
print i.Value
```