Real World IronPython

Dynamic Languages on .NET

Michael Foord

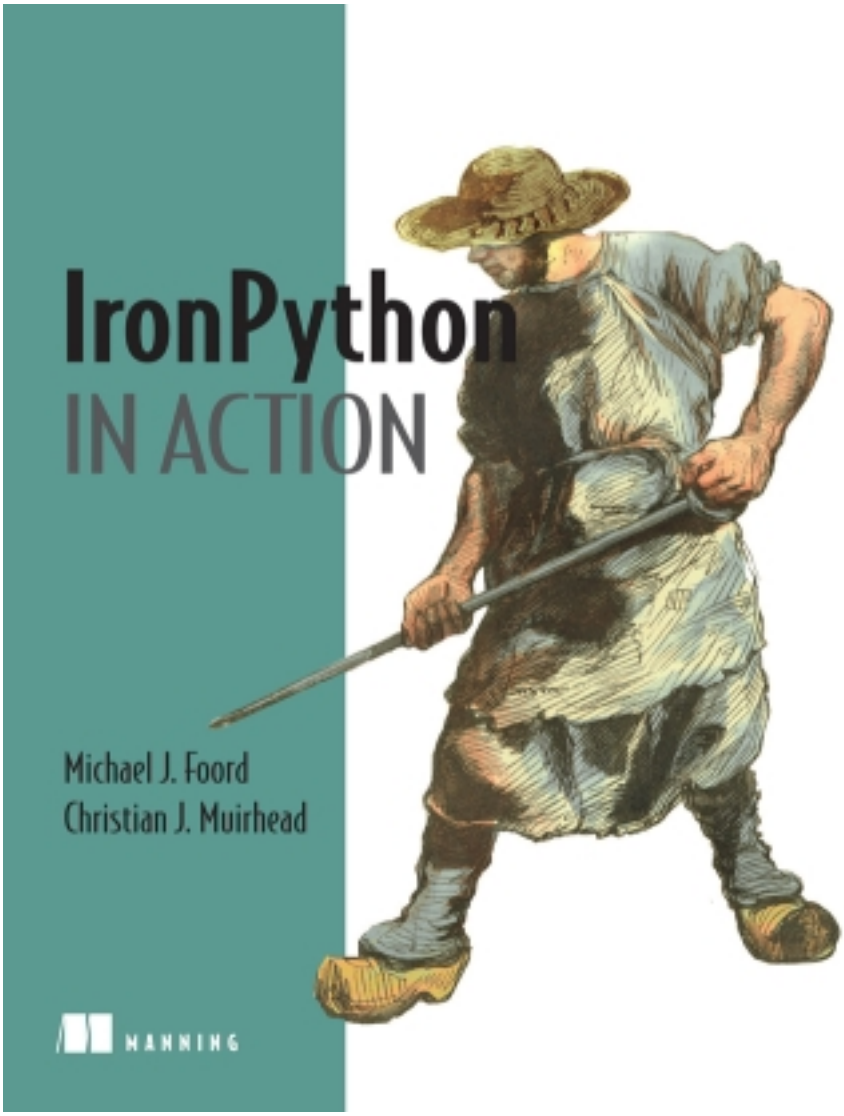
Resolver Systems

michael@voidspace.org.uk

www.ironpythoninaction.com

www.resolversystems.com

Introduction



- Developing with Python since 2002
- Joined Resolver Systems in 2006
- Programming full time with IronPython
- Creating a programmable .NET spreadsheet
- www.resolversystems.com
- wwww.ironpythoninaction.com

Stick 'em up

- Who has experience with IronPython?
- Regular Python?
- Dynamic languages in general?
- Calibration: Who has never used a dynamic language?

IronPython

- IronPython is a Python *compiler*

- Runs on .NET and Mono (in fact included in Mono)
- Originally created by Jim Hugunin
- Now being developed by a Microsoft team
- Version 2.0 is built on the Dynamic Language Runtime (Python 2.5)
- Runs on Silverlight

IronPython is a port of the popular programming language Python to the .NET framework. The project was started by Jim Hugunin when he wanted to write an article 'Why the .NET Framework is a Bad Platform for Dynamic Languages'. Jim had already implemented Python for the JVM (Jython), so he knew that Virtual Machines intended to run static languages *could* support dynamic languages, and he wanted to know *why* .NET had a reputation for being bad.

As it turned out he discovered that the CLR was actually a pretty good VM for dynamic languages and his 'toy' implementation of Python actually ran faster than CPython! He demoed this fledgling version of IronPython to Microsoft, got hired, and the rest is history.

Why is it called IronPython? It Runs On .NET! (A backronym by John Lam, but it's pretty good.)

Microsoft are serious about IronPython and dynamic languages for the .NET framework. Microsoft have built IronPython support into various projects.

IronPython is very well integrated with the .NET framework. Strings in Python are .NET string objects, *and* they have the Python methods that you would expect. The same is true for the other types.

We can also take .NET assemblies and import classes / objects from them, and use them with *no* wrapping needed.

Dynamic Languages on .NET

- A framework for creating Dynamic Languages
- Including language interoperation
- The DLR will be part of .NET 4 (C# 4.0 / VB.NET 10)
- DLR Languages
    - IronPython
    - IronRuby
    - Managed JScript (closed source, Silverlight only)
    - IronScheme (community project on codeplex)

Dynamic languages are:

"...a broad term to describe a class of **high level programming languages** that execute **at runtime** many common behaviours that other languages might perform during compilation."

Type validation	Method dispatch
Attribute lookup	Inheritance lookup
Type creation	Parsing

All deferred until runtime - a trade-off

Deferring things until runtime that a statically typed language would do at compile time is the tradeoff you make when using a dynamically typed language - this is where both the costs and the benefits come from.

Parsing and byte-code compilation: this is where eval and the ability to easily generate / execute new code at runtime comes from.

Type validation - types are not verified up front. They are verified when they are used - languages like Python and Ruby are strongly typed languages, not weakly typed.

Late bound member lookup - this brings about some interesting capabilities in dynamically typed languages. We can change the implementation at runtime, or even add new implementations - a practise called monkey patching (a term originating in the Python community where the practise is looked down on but it can be particularly useful testing). It also means we can program against the capabilities of objects instead of needing strict interfaces to keep the compiler happy. This is called duck typing. If an object walks like a duck, and quacks like a duck then our code is free to treat it like a duck.

Even function and class creation is done at runtime. Typically in dynamic languages functions and classes are first class objects. They can be created at runtime and we can even trigger effects when they are created. This enables simple use of programming techniques like functional programming and metaprogramming. Object creational patterns like function and class factories are trivially easy in many dynamically typed languages.

OMGWTF?

Adding types at runtime?

Changing the inheritance tree?

Adding or deleting methods from types and from objects?

No type safety?!

Reassign to `self.__class__` ?!?!

That sounds crazy and dangerous!

http://www.flickr.com/photos/cyanocorax/220561744

Life as a developer is hard enough as it is. Clearly, writing bug-free code is a difficult problem to solve in the general case. Why on Earth would we want to let go of our most fundamental safety aids, compilation checks? Shouldn't we be striving for more safety, not less?

Less Safety is Sometimes Good

Yes, it is more dangerous. That doesn't make it bad. The upside is it is more flexible.

What does type safety buy?

Type safety does eliminate particular classes of errors.

For example, the compiler can assure you that when using the return value from your integer addition method, 1+1 always returns an integer.

But this is not sufficient to confirm that an application actually works.

In order to have confidence about this, the very best method known to todays computer science is automated tests. Unit tests and acceptance tests.

Unit tests are always needed, and they are much more detailed about run-time behaviour than static type checking can ever be.

Another show of hands, who uses unit tests? Who uses them thoroughly? Test-driven development?

With tests in place confirming correct values, then checks of correct type are now redundant, and can safely be removed.

- paraphrased from Jay Fields,

Ruby luminary, card shark.

Money Quote

In 5 years, we'll view compilation as the weakest form of unit testing.

Stuart Halloway

In practice, the benefits of type safety turn out, unexpectedly, to be fairly minimal.

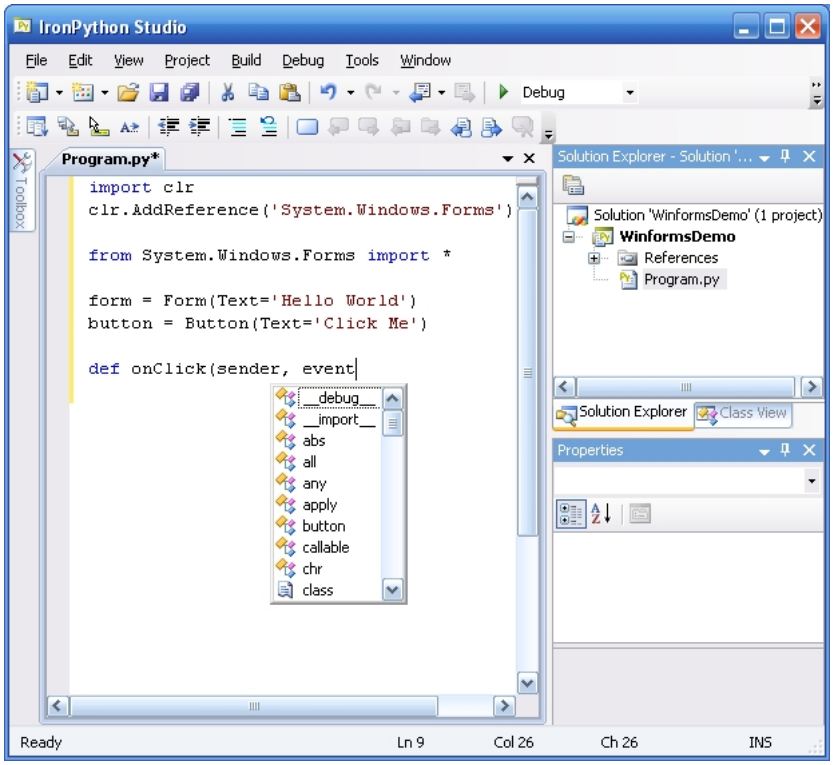
Often overlooked, the costs of maintaining type safety turn out to be extremely high.

IronPython .NET Integration

IronPython demo!

```
>>> from System import Array
>>> int_array = Array[int]((1, 2, 3, 4, 5))
>>> int_array
System.Int32[](1, 2, 3, 4, 5)
>>> dir(int_array)
['Add', 'Address', 'AsReadOnly', 'BinarySearch', 'Clear', 'Clone', 'ConstrainedCopy', 'Contains', 'ConvertAll', 'Copy', 'CopyTo',
'Count', 'CreateInstance', 'Equals', 'Exists', 'Find', 'FindAll',
'FindIndex', 'FindLast', 'FindLastIndex', 'ForEach', 'Get', ...]
```

IronPython Studio

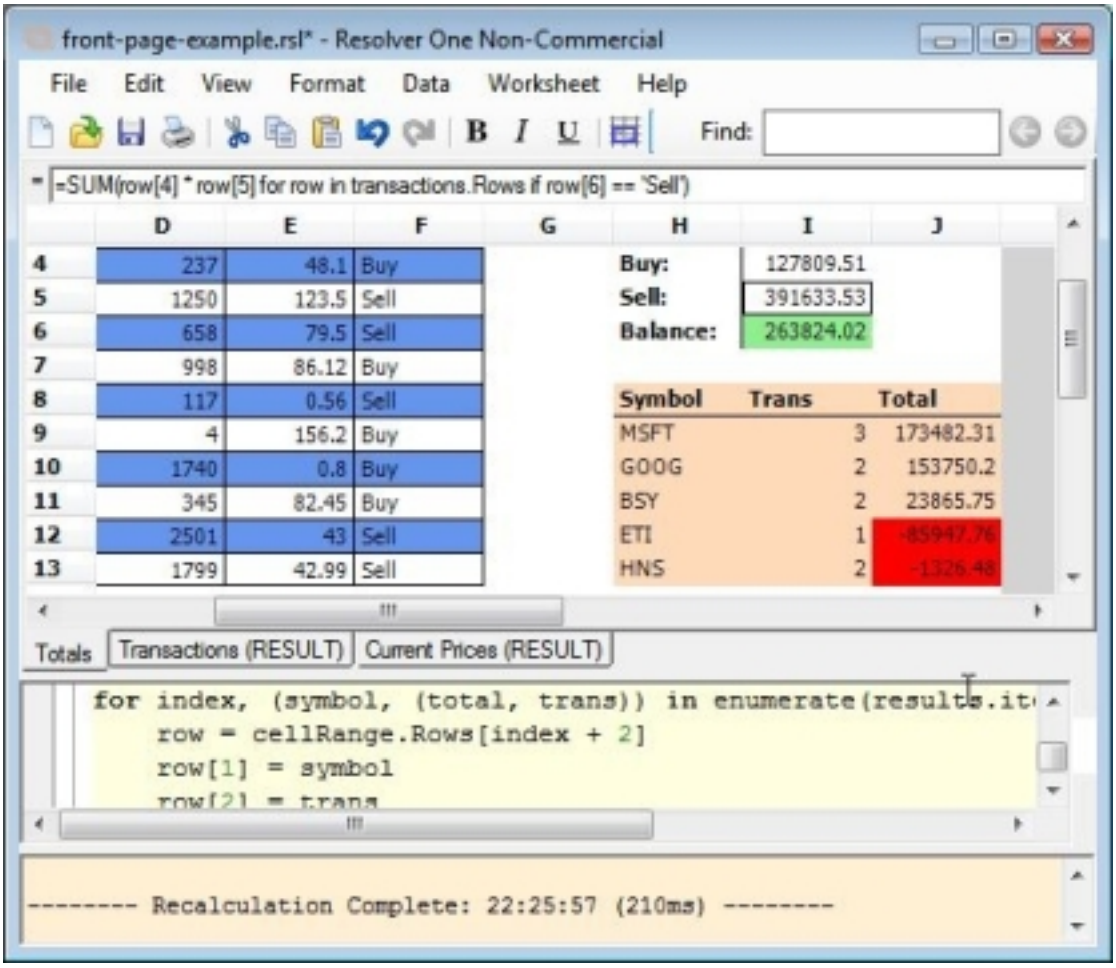


Resolver One

- .NET programmable spreadsheet
- Programming model is central to the application
- Written in and programmed with IronPython
- Use .NET and Python libraries
- Put arbitrary objects in the grid
- Create spreadsheet systems with `RunWorkbook`
- Export spreadsheets as programs
- Import and export Excel spreadsheets

RunWorkbook effectively embeds the calculation engine and lets you override values in a spreadsheet before calculation and fetch the results afterwards. This allows you to treat spreadsheets as data sources or as functions that encapsulate calculations.

In Action



Can you maintain large projects in dynamic languages?

I'd like to answer this question by showing you what Resolver Systems has done with IronPython.

The three founders of Resolver all worked in the London financial services industry. In that business it is very common for people who aren't programmers to need to build business applications. They don't want to have to go the IT department - they need to be able to create applications for very short term opportunities.

Currently they're all using Excel. Excel is a great application, but beyond a certain level of complexity, the traditional spreadsheet metaphor - of cells in a grid with macros off to one side - breaks down.

So the idea for Resolver One was born - a program where the data and formulae in the grid are turned into code (in an interpreted language) and code that the user writes is executed as part of the spreadsheet.

So how did we end up using IronPython and in fact writing the whole application in IronPython?

Late 2005 two developers started work on Resolver. They chose .NET, and Windows Forms for the user interface, as the development platform, a logical choice for a desktop application. And if you're writing a .NET business application, you write it in C# right? That's what the developers assumed.

But having an interpreted language embedded into Resolver is a central part to the way Resolver One works, so they started evaluating scripting language engines available for .NET. At this time IronPython was at version 0.7 I think. What particularly impressed them about IronPython was the quality of the .NET integration and they decided to see how far they could get writing the whole application in IronPython.

That was almost two years ago. Resolver One is now written (almost) entirely in IronPython, there's over 40000 lines of IronPython production code, *plus* over 120000 lines in the test framework. Resolver One is free for non-commercial use and can be downloaded from the Resolver Systems website.

Why Use IronPython?

- Application development (desktop, web or Silverlight)
- Dynamically explore assemblies and classes
- System administration / scripting
- Prototyping / rapid development
- User scripting of .NET Applications or extensible architectures
- Business rules stored as text or using a custom DSL

Intellipad

The Intellipad tool, part of the Oslo framework, is extensible with (and partly written in) IronPython.

```
@Metadata.CommandExecuted('{Microsoft.Intellipad}BufferView', '{Microsoft.Intellipad}OpenExplorerAtBuffer', 'Ctrl+B')
def OpenExplorerAtBufferExecute(target, sender, args):
    file = sender.Buffer.Uri.AbsolutePath
    exists = File.Exists(file)
    if exists:
        Process.Start(Path.GetDirectoryName(file))
```

Almost all commands that are available in Intellipad have been written in Python using the object model exposed by the application. The Python files are scattered inside the Settings directory.

Commands.py contains most of the commands for Intellipad.

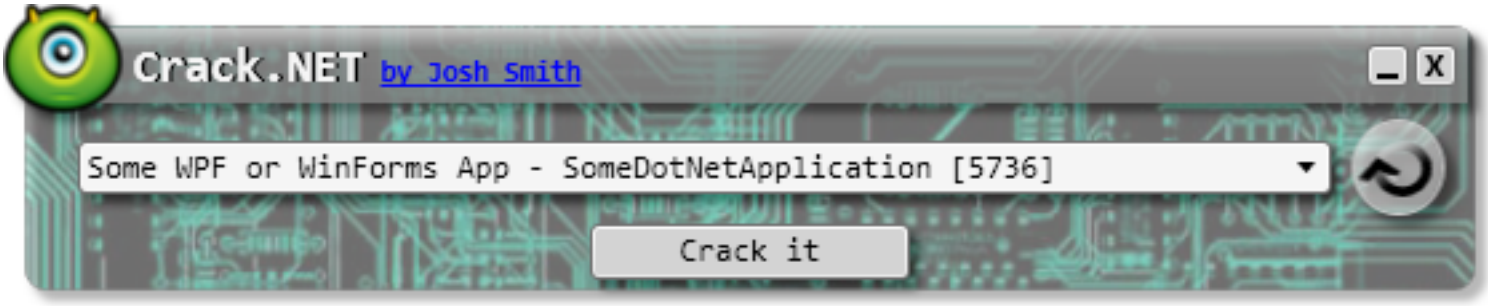
Configuration specific commands are placed in their respective directories (Emacs or VI or VisualStudio).

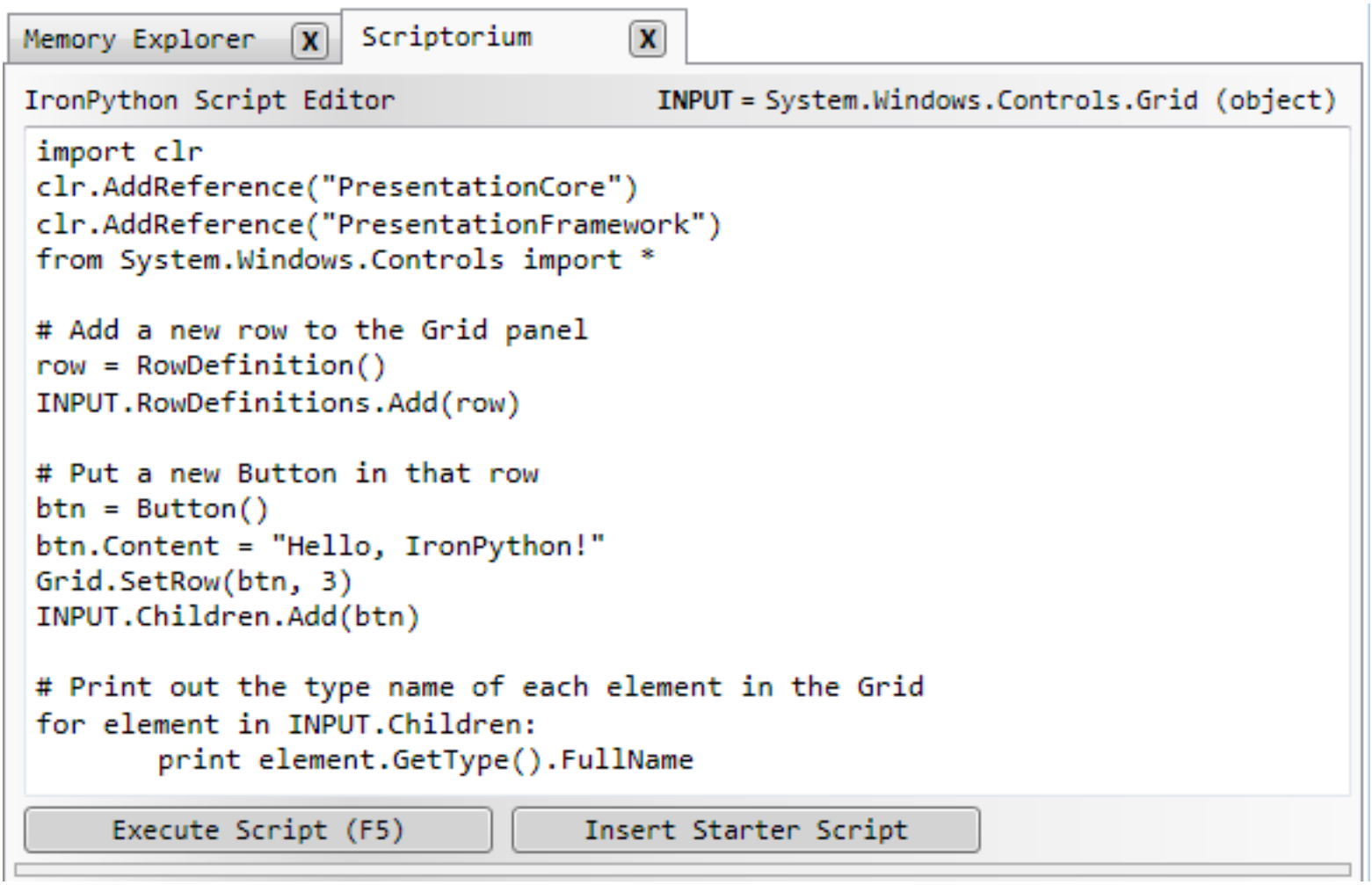
A command definition consists of three parts.

- A "Executed" function definition, that acts as the command handler and provides the logic for the command
- An optional "CanExecute" function definition, that determines when the command is enabled
- A command wireup, that is done by calling the "Common.Command" function. This is the where the Executed and CanExecute are wired together along with the Command Name and default key binding

Crack.NET

Debugger and application explorer: interact with winforms / WPF applications.

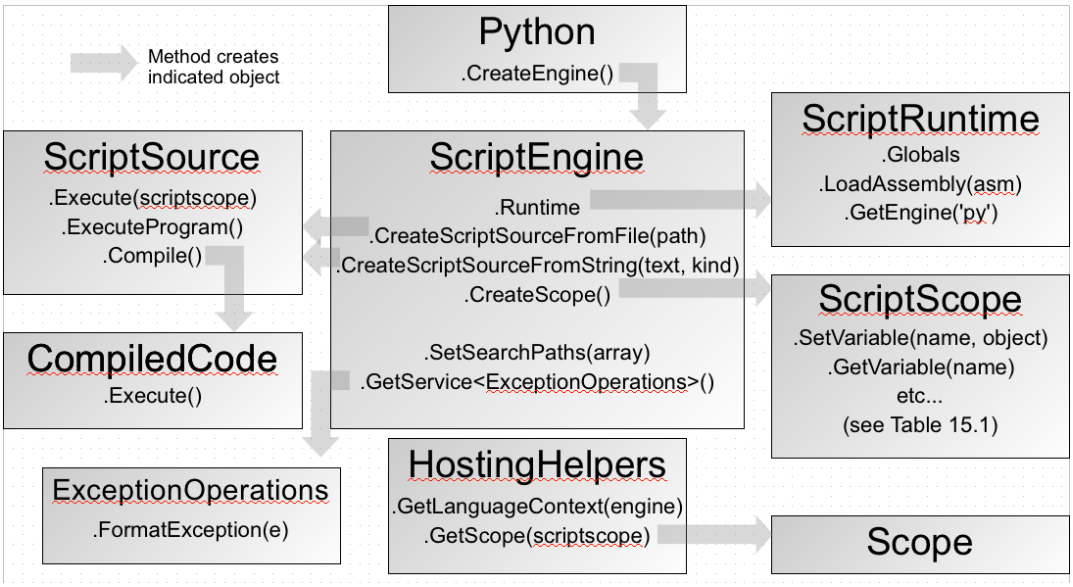




Crack.NET is a runtime debugging and scripting tool that gives you access to the internals of a WPF or Windows Forms application running on your computer. Crack.NET allows you to “walk” the managed heap of another .NET application, and inspect all values on all objects/types.

- http://joshsmithonwpf.wordpress.com/cracknet/
- http://www.codeplex.com/cracknetproject

Embedding IronPython



You have 15 seconds to memorize this...

This shows the major DLR Hosting API components. The most important ones are:

- ScriptEngine
- ScriptRuntime
- ScriptScope
- ScriptSource
- CompiledCode

The ScriptEngine

The Python convenience class from IronPython.Hosting allows us to create a ready configured Python `ScriptEngine`:

```
>>> import clr
>>> clr.AddReference('IronPython')
>>> from IronPython.Hosting import Python
>>> engine = Python.CreateEngine()
>>> engine
<Microsoft.Scripting.Hosting.ScriptEngine object at 0x... [Microsoft.Scripting.Hosting.ScriptEngine]>
```

If we were hosting IronRuby we would use the Ruby class and we would get a ScriptEngine specialised for the Ruby language.

Executing Code

The ScriptSource represents source code and the ScriptScope is a namespace. We use them both to execute Python code - executing the code (the ScriptSource) in a namespace (a ScriptScope):

```
SourceCodeKind st = SourceCodeKind.Statements;
string source = "print 'Hello World'";
script = eng.CreateScriptSourceFromString(source, st);
scope = eng.CreateScope();
script.Execute(scope);
```

The namespace holds the variables that the code creates in the process of executing it.

The ScriptSource has Compile method which returns a CompiledCode object. This also has an Execute method that takes a ScriptScope.

This really is an overview - there are lots of other ways of working with these classes. For example we could execute an expression instead of statements and directly return the result of evaluating the expression. See chapter 15 of IronPython in Action for a more in depth look.

Setting and Fetching Variables

To IronPython the ScriptScope is a module - a namespace that maps names to objects.

```
int value = 3;
scope.SetVariable("name", value);

script.Execute(scope);

int result = scope.GetVariable<int>("name");
```

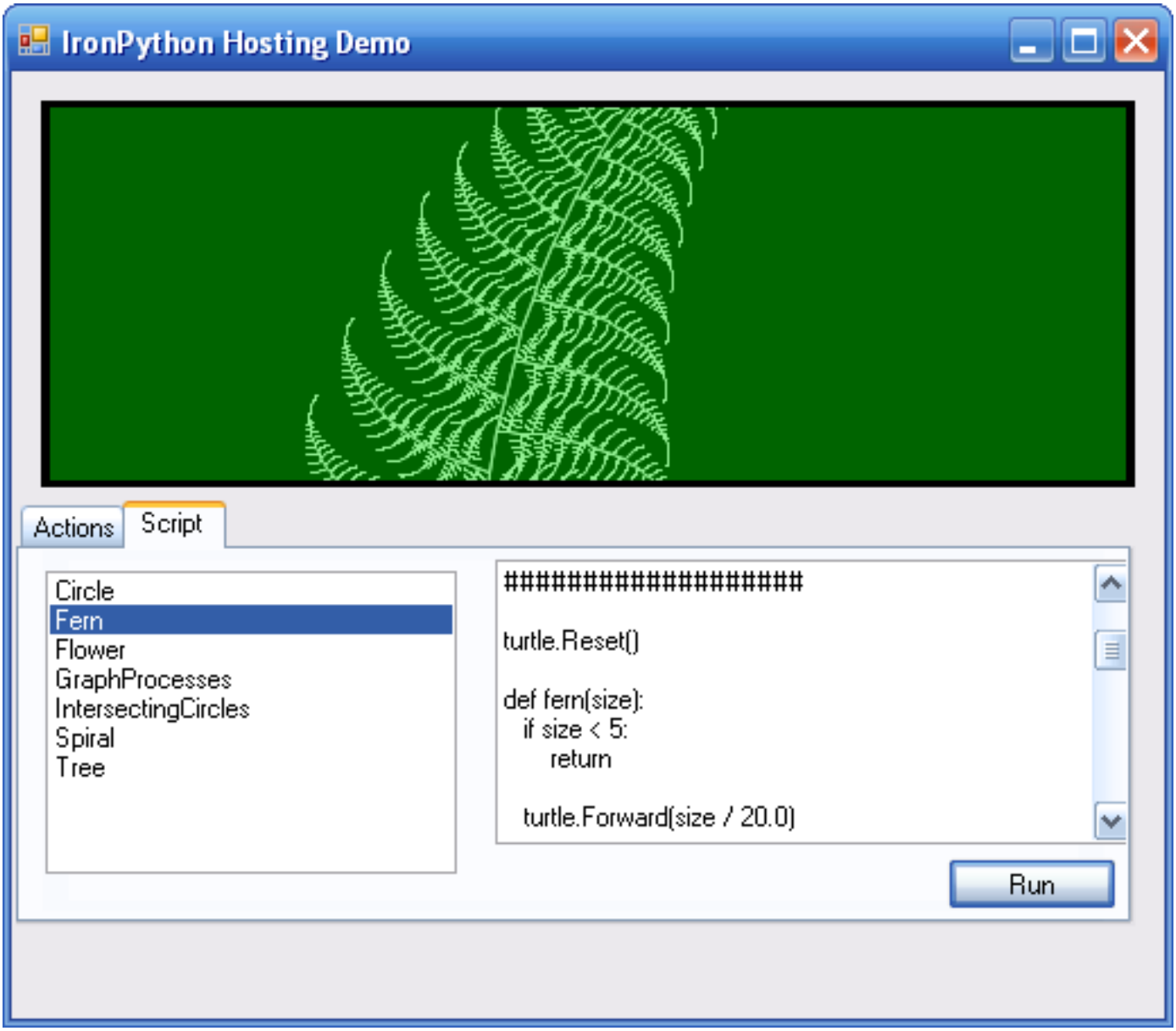
Reflector is a handy tool for finding your way around the available APIs. There are other useful methods on the SciptScope like TryGetVariable, GetVariableNames, ContainsVariable, etc.

So as well as code creating variables we can pre-poulate the namespace with variables that the code has access to. This is one way a hosting application can expose an API / object model to code it executes.

The generic versions of these APIs are great if we are fetching a standard .NET object (but watch out for runtime exceptions if the object we are fetching is of the wrong type and can't be cast to the type you've specified) - but what if we want a dynamic object like an instance of a Python class?

This is the 'type problem'; how can we use and interact with dynamic objects from statically typed .NET languages?

The ScriptedTurtle



A simple example of adding IronPython scripting to an application.

Embedding IronPython into this app is only a handful of lines of code.

Available for download (along with several other good articles on the subject) from: http://www.voidspace.org.uk/ironpython/embedding.shtml

Error Handling

The SyntaxErrorException is a general DLR exception. There are also Python specific ones.

```
catch (SyntaxErrorException e)
{
    ExceptionOperations eo;
    eo = engine.GetService<ExceptionOperations>();
    string error = eo.FormatException(e);

    string caption;
```

```
    string msg = "Syntax error in \"{0}\"";
    caption = String.Format(msg, Path.GetFileName(path));
    MessageBox.Show(error, caption,
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Error);
}
```

SyntaxErrorException is a generic DLR error. There are Python specific errors as well defined in IronPython.Runtime.Exceptions. We can also have a 'catch all' clause that catches 'Exception'.

Errors raised in Python code can be caught as Python exceptions *or* standard .NET exceptions where there is an equivalent.

Dynamic Operations

Solve the type problem by avoiding it until the last minute! (With a little help from `ObjectOperations`.)

```
ObjectOperations ops = engine.Operations;

object SomeClass = scope.GetVariable("SomeClass");
object instance = ops.Call(SomeClass);
object method = ops.GetMember(instance, "method");

int result = (int)ops.Call(method, 99);
```

We can pull objects in as 'object' and then use ObjectOperations to perform dynamic operations on them. This calls back into the DLR to perform the operation. Here we pull out a pure Python class, instantiates it, and call a method on it (casting the result to an integer).

ObjectOperations has many more methods, for example for numeric operations or equality operations, that honour the language semantics of the DLR objects involved.

This is something that gets a lot easier in .NET 4.

Functions as Delegates

IronPython will do casting for us when we pull objects out of the scope. So we can pull Python functions out as .NET delegates if we specify the argument and return types.

```
//get a delegate to the python function
Func<int, bool> IsOdd;
IsOdd = scope.GetVariable<Func<int, bool>>("IsOdd");

//invoke the delegate
bool b = IsOdd(1);
```

Python definition of IsOdd would look something like:

```
IsOdd = lambda x: bool(x % 2)
```

This code pulls out a function that takes an integer as the argument and returns a bool. We use the Func delegate to get a callable delegate on the .NET side, and the IronPython engine casts the function to this type for us. The last type in the generic specification `Func<int, bool>` is the return type.

This example is from the DLR Hosting Blog.

This technique is particularly useful for writing business rules that you want to be able to change at runtime.

C# 4.0 - Dynamic

Dynamic operations with the `dynamic` keyword: not only method calls, but also field and property accesses, indexer and operator calls and even delegate invocations can be dispatched dynamically (at runtime):

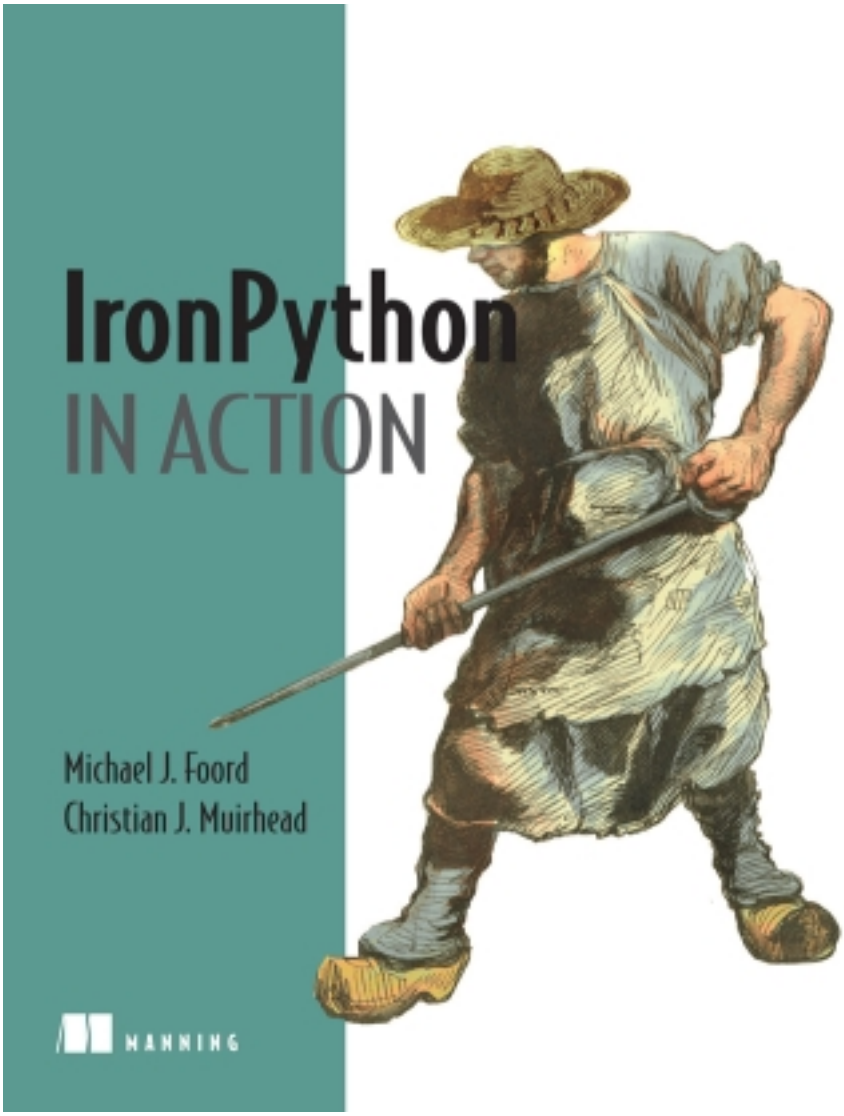
```
dynamic d = GetDynamicObject(…);
d.M(7); // calling methods
d.f = d.P; // getting and settings fields and properties
d[“one”] = d[“two”]; // getting and setting through indexers
int i = d + 3; // calling operators
string s = d(5,7); // invoking as a delegate
```

Example from: http://code.msdn.microsoft.com/csharpfuture/

When compiled this generates IL that calls into the DLR. The DLR either uses reflection to do the lookups, or if the object is a DLR object then it used the correct semantics for the language they are implemented in. (The cost of course is that you can get runtime errors if you call methods that don't exist or attempt to perform invalid operations.) The advantage of this is that it makes it much easier to use DLR objects from .NET languages.

For example see: http://keithhill.spaces.live.com/Blog/cns!5A8D2641E0963A97!6676.entry?wa=wsignin1.0

IronPython in Action



- **www.resolversystems.com**
- **wwww.ironpythoninaction.com**
- **www.voidspace.org.uk/blog**
- **www.voidspace.org.uk/ironpython/**
- **www.ironpython.info**

Questions?