# IronPython
## IN ACTION

Michael J. Foord
Christian Muirhead
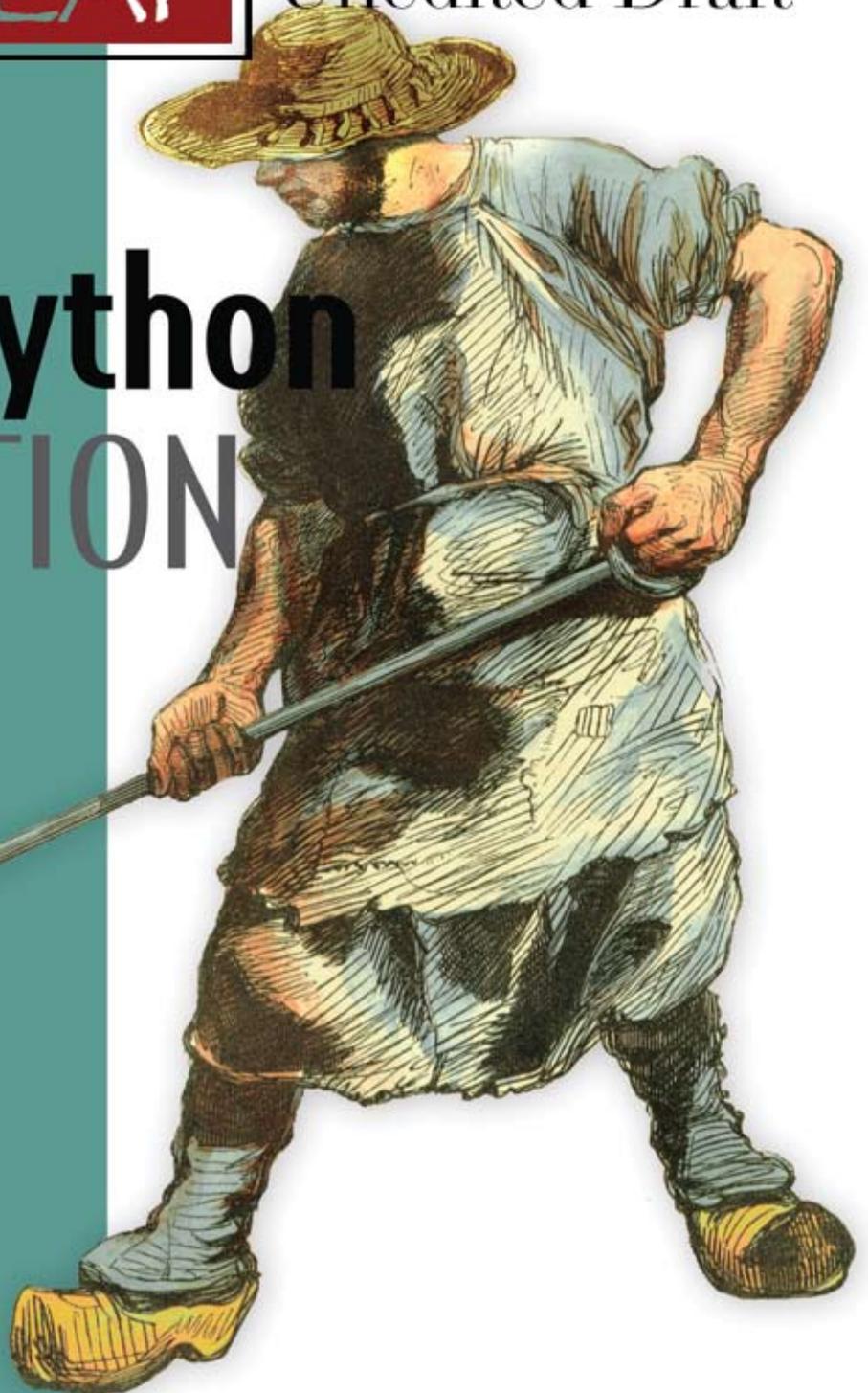
**MANNING**

**MEAP Edition**
**Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

# Contents

# Appendices

# Part 1: Getting Started with IronPython

Like all good books, and possibly a few bad ones, this one starts with an introduction. In this section we'll cover what IronPython is, how it came into being, and why a language like Python is a big deal for .NET. We also get to use the IronPython interactive interpreter which is both a powerful tool and a great way of showing off some of the features of Python. Chapter 2 is a swift tutorial to the Python language. It won't make you a Python master but it will prepare you for the examples used throughout this book and serve as a useful reference well beyond. Chapter 3 briefly introduces .NET and then wades into programming with IronPython, taking Windows Forms as the example. Whilst gaining an understanding of concepts essential to any real work with IronPython you'll be getting your hands dirty with some real code. First though, let's discuss how IronPython fits in with .NET.

# Chapter 1: A New Language for .NET

.NET is a system created by Microsoft for building and running programs. At the heart of it is the Common Language Runtime, usually abbreviated to CLR, which runs .NET applications. The CLR is a powerful system including a just-in-time compiler, built in memory management, and security features. Fortunately, you can write .NET programs that take advantage of many of these features without having to understand them, or even be aware of them. Along with the runtime comes a vast array of libraries and classes, collectively known as the framework classes. Libraries available in the .NET framework include the Windows Forms and WPF[1] graphical user interfaces, libraries for communicating across networks, working with databases, creating web applications and a great deal more.

The traditional languages for writing .NET programs are Visual Basic.NET, C# and C++. IronPython is a .NET compiler for a programming language called Python, making IronPython a first class .NET programming language. If you're a .NET developer you can use Python for tasks from web development, to creating simple administration scripts, and just about everything in between. If you're a Python programmer it means you can use your favorite language to take advantage of the .NET framework.

IronPython isn't cast in the same mold as traditional .NET languages, although there are similarities. It is a dynamically typed language, which means a lot of things are done differently and you can do things that are either impossible or more difficult with alternative languages. Python is also a 'multi-paradigm' language, it supports such esoteric styles of programming as procedural and functional programming, object oriented programming, meta-programming and more.

Microsoft has gone to a great deal of trouble to integrate IronPython with the various tools and frameworks that are part of the .NET family. They have built specific support for IronPython into the following projects:

- Visual Studio, the Integrated Development Environment
- ASP.NET (Active Server Pages), the Web Application framework
- Silverlight, IronPython in your browser for client-side web programming
- XNA[2], the game programming system
- Microsoft Robotics Kit
- Jasper, a declarative database project

By the end of IronPython in Action, I hope you will have learned everything you need to tackle creating full applications with IronPython, integrating IronPython as part of another application, or just using it as another tool in your toolkit. You will also have explored some of these alternative programming techniques and used a variety of different aspects of the

---

[1] Windows Presentation Foundation: Microsoft's 'next generation' user interface framework.
[2] As far as I can tell, XNA is a recursive acronym standing for "XNA's Not Acronymed". I may be perpetuating a longstanding joke here of course.

.NET framework. This will enable you to make the best use of the Python language and the wealth of classes made available by .NET.

IronPython is already being used in commercial systems, both to provide a scripting environment for programs written in other .NET languages, and to create full applications. Resolver[3] is one great example, and is how I got involved with IronPython. You can see a screenshot of Resolver in figure 1.1. At last count, there were over twenty five thousand lines of IronPython code in Resolver, plus around ninety thousand more in the test framework developed alongside it.

**Figure 1.1 Resolver: A Full Application written in IronPython (screenshot to be supplied soon…)**
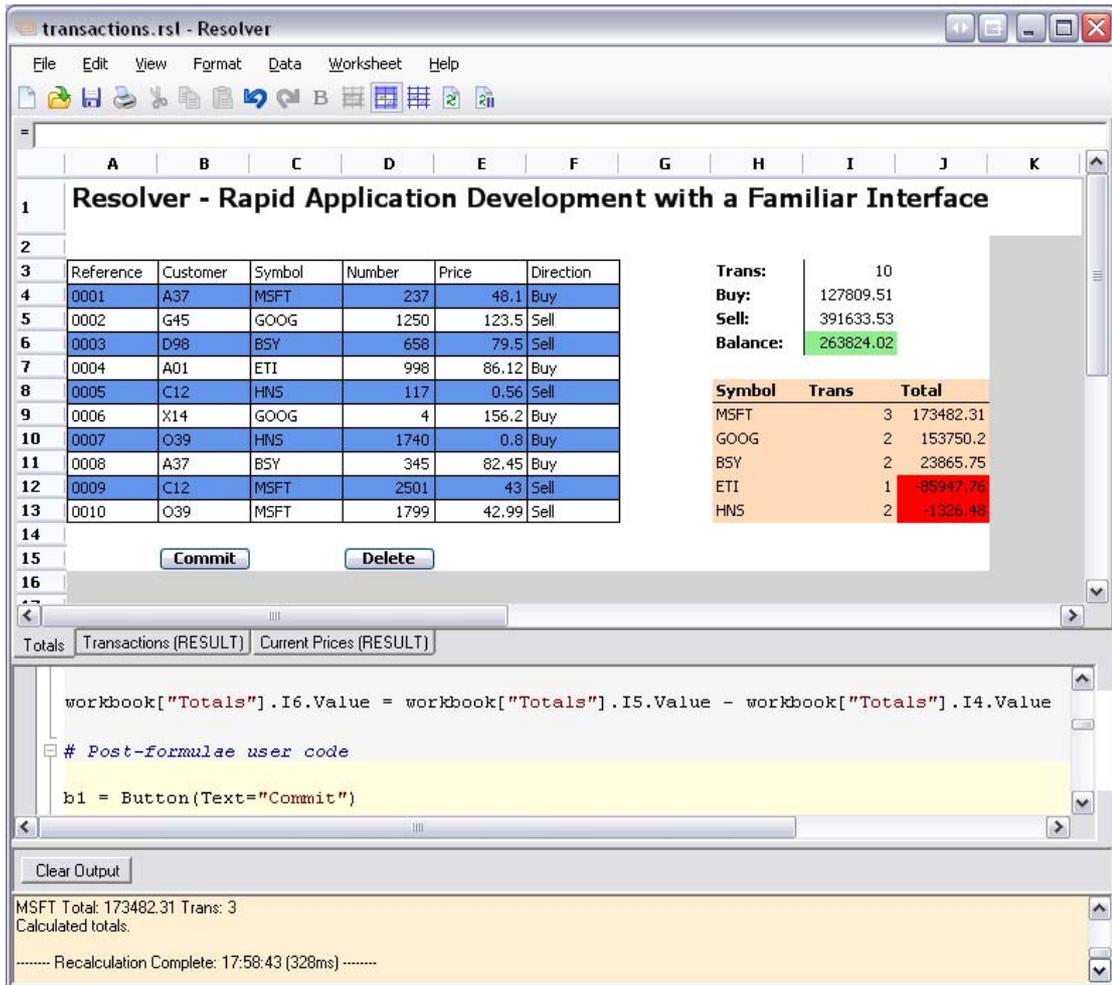


Before we can achieve any of this, an introduction is in order. This chapter introduces IronPython and the Python programming language. It explains why Python is a good fit for the .NET framework and will give you a tantalizing taste of what is possible with IronPython, via the interactive interpreter.

---

[3] See http://www.resolversystems.com

# 1.1 An Introduction to IronPython

Python is a dynamic language that has been around since 1990 and has a thriving user community. Dynamic languages don't require you to declare the type of your objects and they allow you greater freedom to create new objects and modify existing ones at runtime. On top of this, the Python philosophy places great importance on readability, clarity and expressiveness. Figure 1.2 is from a presentation[4] by Guido van Rossum, the creator of Python. It explains why readability is so important in Python.

**Figure 1.2 A slide from a presentation, emphasizing a guiding philosophy of Python**
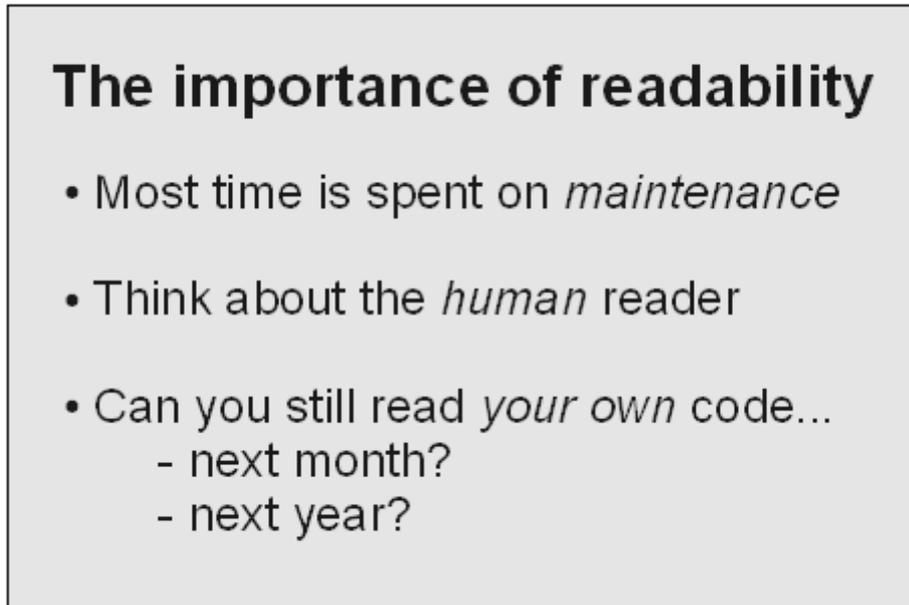


IronPython is an Open Source implementation of Python for .NET. It has been developed by Microsoft as part of making the CLR a better platform for dynamic languages. In the process they have created a fantastic language and programming environment. But what exactly is IronPython?

## 1.1.1 What is IronPython?

IronPython primarily consists of the IronPython engine, along with a few other tools to make it convenient to use. The IronPython engine compiles Python code into assemblies, which run on the CLR. These assemblies can be run from memory, or saved to disk.

> **Assemblies**
> Assemblies are .NET libraries or executables. .NET consists of a *great deal* of these assemblies, in which the framework classes live, in the form of dlls.
> Because of the memory management and security features that .NET provides, code in .NET assemblies is called 'managed code'[5].

---

[4] http://www.python.org/doc/essays/ppt/hp-training/index.htm
[5] .NET does provide ways to access 'unmanaged' code, contained in traditional compiled dlls.

Assemblies contain code compiled from .NET languages into bytecode, known as IL which stands for 'Intermediate Language'. This is run with the JIT (Just in Time compiler) for fast execution.

In straightforward terms, this means that you can run Python programs directly from sourcecode files, *or* distribute them in binary form. You can see how Python code is compiled and run by the IronPython Engine in figure 1.3.

**Figure 1.3 How Python code and the IronPython Engine fits into the .NET world**
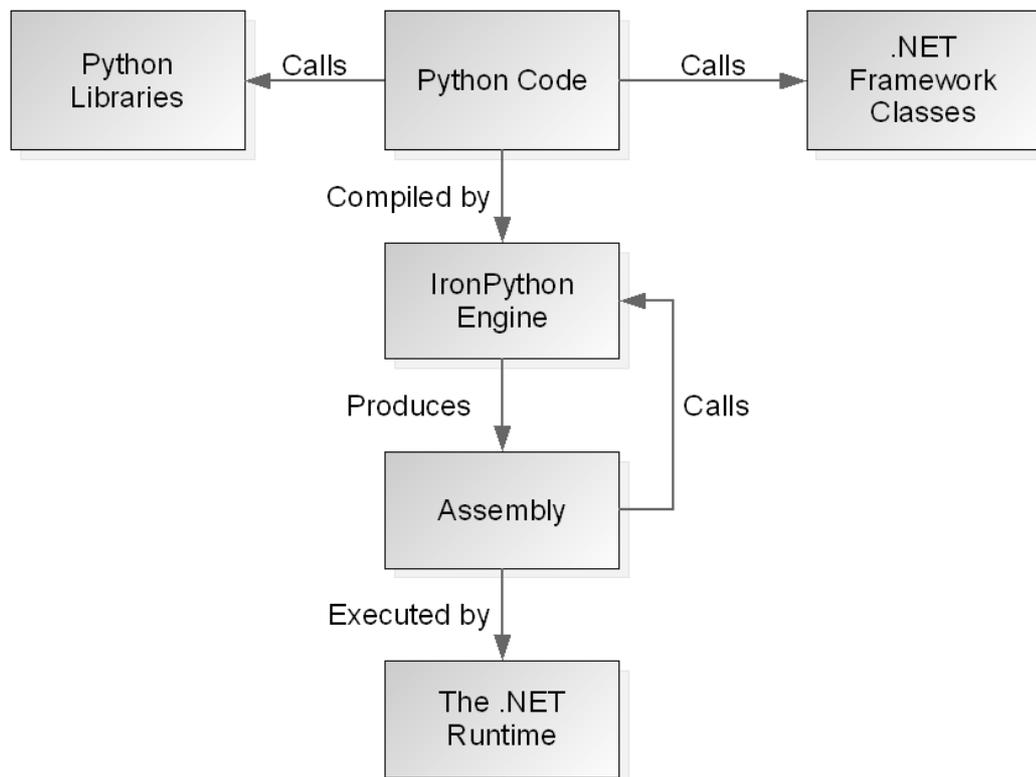


Figure 1.3 shows the state of IronPython version 1. In April 2007 the IronPython team released an early version of IronPython 2. This introduces a radical new development, the DLR: The Dynamic Language Runtime. The DLR is a hosting platform and dynamic type system that have been taken out of IronPython 1 and turned into a system capable of running many different dynamic languages. We will be hearing more about the DLR in a short while.

Because Python is a highly dynamic language, the generated assemblies remain dependent on the IronPython dlls. Despite this, they are still just compiled .NET code and so you can use classes from the .NET framework directly within your code without needing to do any type conversions yourself.

Accessing the .NET framework from IronPython code is extremely easy. As well as being a programming language in its own right IronPython can be used for all the typical tasks you might approach with .NET. This includes web development with ASP (Active Server Pages, the .NET web application framework) or creating smart client applications with Windows Forms or the Windows Presentation Foundation. As an added bonus, IronPython

also runs on the version of the CLR that is shipped with Silverlight 1.1. You can use IronPython for writing client-side applications that run in a web browser, something that Python programmers have wanted for years!

IronPython itself is written in C# and is a full implementation of Python. IronPython 1 is Python version 2.4, whilst IronPython 2 is Python 2.5. If you've used Python before, IronPython is Python with none of the core language features missing or changed. Let me make this clear: IronPython is Python.

Development cycles are typically very fast with Python. With dynamically typed languages tasks can be achieved with less code, which makes IronPython ideal for prototyping applications or scripting system administration tasks which you can't afford to spend a lot of time on. Because of the readability of well written Python code, it scales well to writing large applications. You may well find that your prototypes or scripts can be refactored into full programs much more easily than writing from scratch in an alternative language.

If you're already developing with .NET, you needn't do without your favorite tools. Microsoft has incorporated IronPython support into Visual Studio through the SDK[6]. You can use Visual Studio to create IronPython projects, with full access to the designer and debugger. Figure 1.4 shows Visual Studio being used to create a Windows application with IronPython.

**Figure 1.4 Generated IronPython code in Visual Studio**
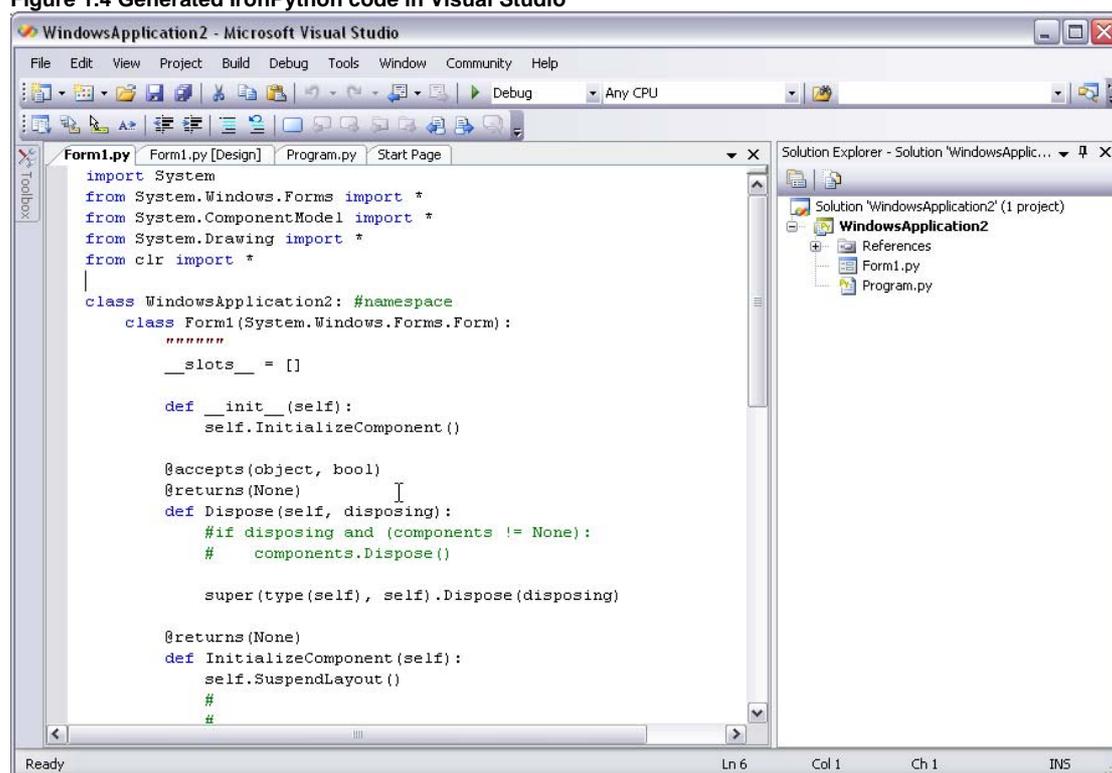


There is an alternative version of .NET called Mono. This provides a C# compiler, runtime and a large proportion of the framework for platforms other than Windows.

---

[6] Software Development Kit. The Visual Studio SDK is a Microsoft extension which includes IronPython support.

IronPython runs fine on Mono. This opens up the possibility of creating fully featured cross-platform programs using IronPython. Windows Forms is available on Mono, so GUI applications written with IronPython can run on any of the many platforms that Mono works on.

IronPython is a particularly interesting project for Microsoft to have undertaken. Not only have they taken a strong existing language and ported it to .NET, but they have chosen to release it with a sensible Open Source license. You have full access to the source code for IronPython, which is a very good example of compiler design, and you can create derivative works and release them under a commercial license. This is at least partly due to the man who initiated IronPython, Jim Hugunin. Let's explore his role in creating IronPython along with a brief history lesson.

## 1.1.2 A Brief History of IronPython

The standard version of Python is called CPython; the C is because it is written in C. CPython is overwhelmingly the most used version of Python and most Python code is written to run on it. CPython isn't Python though. Python is a programming language, and CPython is just one (albeit a very important one[7]) implementation.

**Python Implementations**

The most common Python implementation is called CPython. Other implementations include:

- IronPython for .NET
- Jython for the Java VM
- PyPy, an experimental implementation of Python in Python with a multitude of backends (target platforms)
- CLPython, Python written in Common Lisp
- Stackless, an alternative to CPython which makes minimal use of the C stack and has support for green threads

IronPython is not the first version of Python to target an alternative platform to the usual Python runtime. The most famous alternative is 'Jython', Python for the Java VM. The original version of Jython, or 'JPython' as it was known then, was created by a gentleman called Jim Hugunin.

Over the last few years dynamic languages have been rising in popularity. Their emphasis on concise code and empowering the programmer has attracted a great deal of developer interest. However, back in 2003 the CLR was widely regarded as being a bad platform for hosting dynamic languages[8]. Jim decided to write an article examining in depth *why* .NET was so bad for these languages.

---

[7] Python has no formal specification. It is defined by the language reference documentation and from CPython which is called a reference implementation.

[8] For example, see the InfoWorld article, from 2004, 'Does .Net have a dynamic-language deficiency?'. Ironically this was written by Jon Udell who now works for Microsoft. http://www.infoworld.com/article/04/02/27/09FEmsnetdynamic_1.html

His experience with the Java VM proved that it was certainly *possible* to create language runtimes capable of hosting both static and dynamic languages and he was intrigued as to what Microsoft had gotten so wrong.  Naturally the way he explored this was by attempting a toy implementation of Python.  To his horror he discovered that contrary to popular opinion, Python worked very well on .NET.  In fact his initial attempt ran the basic Python benchmark 'pystone' 1.7 times faster than CPython.

This was unfortunate because a full language implementation is a major undertaking, and Jim now felt honor bound to take his experiment further.

Having made his results public, Jim was invited to present them to Microsoft.  Microsoft was particularly interested in the challenges and difficulties that Jim had encountered as they were keen to make the CLR a better platform for dynamic languages.

The upshot was that Jim joined the CLR team at Microsoft.  A group of programmers were brought together to work on IronPython, and in the process help improve the CLR. Importantly Microsoft agreed to keep IronPython open source, with a straightforward license similar to the BSD[9] license.

Ever since the early releases the Python community has been closely involved in the development of IronPython.  Releases were made often, and Python programmers have been quick[10] to point out bugs in IronPython, or just differences between IronPython and the way CPython behaves.  The IronPython team were (and in fact still are) both fast and scrupulous in fixing bugs and incompatibilities between CPython and IronPython.

After many beta releases, IronPython 1.0 Production was released in September 2006. Meanwhile, other Microsoft teams were working to ensure that IronPython fits into the different members of the .NET family.  This includes a CTP[11] called IronPython for ASP. IronPython for ASP enables IronPython to be used for .NET web development, and introduces a change to the normal ASP model called 'no-compile' pages.

Then in spring 2007 Microsoft surprised just about everyone with two important releases. The first was IronPython 2 alpha.  IronPython 2 is built on top of an important component called the DLR: the Dynamic Language Runtime.

The second surprise announcement, following hot on the heels of the DLR, was the release of Silverlight.  Silverlight is a plugin that runs inside the browser, for animation, video streaming or creating rich-client web applications.  The biggest surprise was that Silverlight includes a cut down version of .NET, called the 'Core-CLR', and the DLR runs on top of it. This means that any of the languages that use the DLR can be used for client side web programming.  The Python community in particular has long wanted a secure version of Python that they could use for client side web programming.  The last place they expected it to come from was Microsoft!

---

[9] The BSD license is a popular (and permissive) open source license which originated in the Berkeley Software Distribution; a Unix like operating system.
[10] At least perhaps partly because of suspicions about Microsoft's intentions for Python…
[11] CTP means 'Community Technology Preview'.

I'm hoping that you're already convinced that IronPython is a great programming language, but as a developer why should you want to use IronPython? There are two types of programmers to whom IronPython is particularly relevant. The first is the large number of Python programmers who now have a new implementation of Python running on a very different platform to the one they are used to. The second type is .NET programmers, who might be interested in the possibilities of a dynamic language or perhaps need to embed a scripting engine into an existing application. We'll take a brief look at IronPython from both these perspectives, starting with Python programmers.

## 1.1.3 IronPython for Python Programmers

As I mentioned before, IronPython is a full implementation of Python 2.4. If you've already programmed with Python there is nothing to stop you experimenting with IronPython straight away.

The important question is; why would a Python programmer be interested in using IronPython? The answer is basically twofold, the platform and the platform. Let me try and make a bit more sense. First of all I mean the underlying platform that IronPython runs on; the CLR. Secondly, along with the runtime comes the whole .NET framework, a huge library of classes a bit like the Python standard library.

There are several reasons why the Common Language Runtime is an interesting platform. The CLR has had an enormous amount of work to make it fast and efficient. Multithreaded programs can take full advantage of multiple processors, something that CPython programs can't do because of a tricky creature called the 'GIL'[12]. Because of the close integration of IronPython with the CLR, extending IronPython through C# code is significantly easier than extending CPython with C. There is no C API to contend with, you can pass objects back and forth across the boundary without hassles and with no reference counting[13] to worry about. On top of all this, .NET has a concept called 'AppDomains'. These allow you to run code with reduced privileges, like preventing it from accessing the file system, which is a feature that has long been missing from CPython.

IronPython uses .NET classes natively and seamlessly, and there are a lot of them. Two of the gems in the collection are Windows Forms and the Windows Presentation Foundation, which are excellent libraries for building attractive and native looking user interfaces. As a Python programmer, you may be surprised by how straightforward the programmers interface to these libraries feels. Whatever programming task you are approaching, it is likely that there is some .NET assembly available to tackle it. This includes third party libraries for sophisticated GUI components, like data grids, where there is nothing comparable available for CPython. Table 1.1 shows a small selection of the libraries available to you in the .NET framework.

---

[12] The 'Global Interpreter Lock', which makes some aspects of programming with Python easier but has this significant drawback.

[13] CPython uses reference counting for garbage collection, which extension programmers have to take account of.

**Table 1.1 Common .NET Assemblies and Namespaces**

| Assembly Name | Purpose |
|---|---|
| | |
| `System` | Contains the base .NET types, exceptions, garbage collection classes and much more. |
| `System.Data` | Classes for working with databases, both high and low level. |
| `System.Drawing` | Provides access to the GDI+ graphics system. |
| `System.Management` | Provides access to Windows management information and events (WMI), useful for system administration tasks. |
| `System.Environment` | Allows you to access and manipulate the current environment, like command line arguments and environment variables. |
| `System.Diagnostics` | Interact with processes. |
| `System.XML` | For processing XML, including SOAP, XSL/T and more. |
| `System.Web` | The ASP.NET web development framework. |
| `System.IO` | Contains classes for working with paths, files and directories. Includes classes to read and write to filesystems or data-streams, synchronously or asynchronously. |
| `Microsoft.Win32` | Classes that wrap Win32 common dialogs and components including the registry. |
| `System.Threading` | Classes needed for multithreaded application development. |
| `System.Text` | Classes for working with strings (like StringBuilder) and the Encoding classes which can convert text to and from bytes. |
| `System.Windows.Forms` | Provides a rich user interface for applications. |
| `System.Windows` | The base namespace for WPF, the new GUI framework that is part of .NET 3.0. |
| `System.ServiceModel` | Contains classes, enumerations, and interfaces to build Windows Communication Foundation (WCF) service and client applications. |

As we go through the book we'll use several of the common .NET assemblies, including some of those new to the .NET 3.0 release. More importantly we'll learn how to understand the MSDN documentation so that you are equipped to use *any* assembly from IronPython. We will also do some client-side web programming with Silverlight, scripting the browser with Python. This is something that has not been possible before IronPython and Silverlight.

Most of the Python standard library works with IronPython; ensuring maximum compatibility is something the Microsoft team has put a lot of work into. Do beware though. Not all of the standard library works; C extensions don't work because IronPython isn't written in C. In some cases, alternative wrappers may be available[14], but parts of the standard library and some common third party extensions don't work yet. If you are willing to swap out components with .NET equivalents, or do some detective work to uncover the problems, it is usually possible to port existing projects.

Where IronPython really shines is with new projects, particularly those that can leverage the power of the .NET platform. In order to take full advantage of IronPython there are a few particular features you will need to know about. These include things that past experience with Python alone won't have prepared you for. Before we turn to actually using IronPython, let's first look at how it fits in the world of the .NET framework.

---

[14] Several of these are provided by IPCE – the IronPython Community Edition. We'll look at this in a later chapter.

## 1.1.4       IronPython for .NET Programmers

IronPython is a completely new language available to .NET programmers.  It opens up new styles of programming and brings the power and flexibility of dynamic programming languages to the .NET platform.

Programming techniques like functional programming and creating classes and functions at runtime are possible with traditional .NET languages like VB and C#.  They're just a lot easier with Python.   Added to which you get straightforward closures, duck typing, metaprogramming and much more thrown in for free.  We'll explore some of the features which make dynamic languages powerful later in the book.

IronPython is a full .NET language.  Every feature of the .NET platform can be used, with the (current) exception of attributes for which you can use stub classes written in C#.  All your existing knowledge of the .NET framework is relevant for use with IronPython.

So why would you want to use IronPython?  Well, let me suggest a few reasons.

Without a compile phase, developing with IronPython can be a lot quicker than with traditional .NET languages, typically requiring less code and resulting in more readable code. This makes it ideal for prototyping and for use as a scripting language.  Classes can be rewritten in C# at a later date (if necessary) with minimal changes to the programmer's interface.

IronPython may be a new language, but Python isn't.  Python is a mature and stable language.  The syntax and basic constructs have been worked out over years of programming experience.  Python has a clear philosophy of making things as easy as possible for the programmer rather than for the compiler.  Common concepts should be both simple and elegant to express, whilst the programmer should be left as much freedom as possible.

The best way to explain the difference between Python and a static language like C# is to show you.  Table 1.2 demonstrates a simple 'Hello World' program written in both C# and Python.  A 'Hello' class is created, with a 'sayHello' method which prints 'Hello World' to the console.  Differences (and similarities) between the two languages are obvious.

**Table 1.2 Hello World compared in C# and IronPython**

| A Small HelloWorld App in C# | Equivalent in Python |
|---|---|
| ```using System;

class Hello
{
  string msg;

  public Hello()
  {
    this.msg = "Hello World";
  }
``` | ```class Hello(object):
  def __init__(self,
      msg='hello world'):
    self.msg = msg

  def sayHello(self):
      print self.msg

app = Hello ()
app.sayHello()
``` |

```
      public Hello(string msg)
      {
        this.msg = msg;
      }

      public void SayHello()
      {
          Console.WriteLine(
            this.msg);
      }

      public static void Main()
      {
        Hello app = new Hello();
        app.SayHello();
      }
    }
```

You can see from the example above how much extra code is required in the C# for the sake of the compiler. The curly braces, the semi-colons and the type declarations are all 'line noise' that don't actually add to the functionality of the program. They do serve to make the code harder to read.

This example mainly illustrates that Python is syntactically more concise than C#. It is also *semantically* more concise, with language constructs that allow you to express complex ideas with the minimum of code. It is difficult to provide a simple example of this, but we will be looking at some of what Python has to offer in chapter 2.

For those new to dynamic languages, the interactive interpreter will also be a pleasant surprise. Far from being a toy, the interactive interpreter is a fantastic tool for experimenting with classes and objects at the console. You can instantiate classes and explore their properties live, using introspection and the built-in 'dir' and 'help' commands to see what methods and attributes are available to you. As well as experimenting with objects, you can also try out language features to see how they work, not to mention that the interpreter makes a great calculator or alternative shell. We get a chance to play with the interactive interpreter at the end of this chapter.

If you are looking to create a scripting API for an application, embedding IronPython is a great and readymade solution. You can provide your users with a powerful and easy to learn scripting language, which they may already know and that has an abundance of resources for those who want to learn. The IronPython engine and its API have been designed for embedding from the very start, so it requires little work to integrate it into applications.

It's time to take a closer look at Python the language and the different programming techniques it makes possible. I'll even reveal the unusual source of Python's name.

## 1.2  Python on the CLR

The core of the .NET framework is the CLR: the Common Language Runtime.  The CLR runs programs which have been compiled from source code into bytecode.  Any language which can be compiled to IL bytecode can run on .NET.  The predominant .NET languages, VB and C#, are statically typed languages.  Python is from a different class of language, it is dynamically typed.  Let's take a closer look at some of the things that dynamic languages have to offer programmers.

### 1.2.1 Dynamic Languages on .NET and the DLR

For a while, the Common Language Runtime had the reputation of being a bad platform for dynamic languages.  As Jim Hugunin proved with IronPython, this isn't true.  One of the reasons that Microsoft took on the IronPython project was to push the development of the CLR to make it an even better platform for hosting multiple languages, particularly dynamic languages.  The Dynamic Language Runtime, which makes several dynamic languages available for .NET, is the concrete result of this work.

So why all the fuss about dynamic languages?

First of all, dynamic languages are trendy; all the alpha-geeks are using them!  If you have any sense then this won't be enough of an explanation.  Unfortunately, like many programming terms, 'dynamic' is hard to pin down and define precisely.  Typically it applies to languages which are dynamically typed; that don't need variable declarations and where you can change the type of object a variable refers to.

More importantly you can examine and modify objects at runtime.  Concepts like introspection and reflection, while not exclusive to dynamic languages, are *very* important and are simple to use.  Classes, functions and libraries (called modules in Python), are first class objects which can very easily be created in one part of your code and used elsewhere.

In statically typed languages, method calls are normally bound to the corresponding method at compile time.  With Python, the methods are looked up (dynamically) at runtime, so modifying runtime behavior is much simpler.  This is called late binding.

With static typing, you must declare the type of every object.  Every path through your code that an object can follow must respect that type.  If you deviate from this, the compiler will reject the program.  In a situation where you may need to represent any of several different pieces of information you may need to implement a custom class or provide alternative routes through your code which essentially duplicate the same logic.

In dynamic languages, objects still have a type.  Python is a strongly typed language[15], and you can only perform operations that make sense for that type.  For example you can't add strings to numbers.  The difference is that the Python interpreter doesn't check types (or

---

[15] Some dynamic languages are weakly typed and allow you to do some very odd things with objects of different types.

lookup methods) until it actually needs to. Any name can reference an object of any type, and you can change the object that a variable points to. This is dynamic typing. Objects can easily follow the same path, with you only differentiating on type at the point at which it is relevant. One consequence of this is that container types (lists, dictionaries and tuples, plus user defined containers in Python) can automatically be heterogeneous. They can hold objects of any type with no need for the added complexity of generics.

In many cases the actual type doesn't even matter, so long as the object supports the operation being performed, which is called duck typing[16]. Duck typing can remove the need for type checking and formal interfaces. For example to make an object indexable as a dictionary like container, you only need to implement a single method[17].

All this can make programmers who are used to static type checking nervous. In statically typed languages the compiler checks types at compile time, and will refuse to compile programs with type errors. This kind of type checking is impossible with dynamic languages[18] and so type errors can only be detected at runtime. This means that automated testing is more important with dynamic languages, which is convenient as they are usually easier to test. Dynamic language programmers are often proponents of 'strong testing rather than strong typing'[19].

My experience is that type errors form the minority of programming errors and are usually the easiest to catch. A good test framework will greatly improve the quality of your code, whether you are coding in Python or another language, and the benefits of using a dynamic language outweigh the costs.

These factors make dynamic languages compelling for many programmers, but before IronPython they simply weren't available for those using .NET. Microsoft has gone much further than just implementing a single dynamic language though. With IronPython Jim and his team proved that .NET was a good environment for dynamic languages and they created all the infrastructure necessary for one specific language. In the DLR they abstracted out of IronPython a lot of this infrastructure so that other languages could be implemented on top of it.

With the DLR it should be *much* easier to implement new dynamic languages for .NET, and even have those languages interoperate with other dynamic languages because they share a common type system. To prove its worth, there are already four Microsoft created languages that run on the DLR. The first of these is IronPython 2, followed by IronRuby, VBx and Managed JScript. Table 1.3 lists the (current) languages that run on the DLR.

**Table 1.3 Languages that run on the DLR.**

| Language | Notes |
|---|---|

---

[16] If the object walks like a duck, and talks like a duck, let's treat it like a duck...

[17] That method is called '`__getitem__`' and is used for both the 'mapping' and 'sequence' protocols in Python.

[18] Although not all statically typed languages require type declarations. Haskell is a statically typed language which uses type inferencing instead of type declarations. ShedSkin is a Python to C++ compiler which also uses type inferencing and compiles a static subset of the Python language into C++.

[19] A phrase borrowed from Bruce Eckel, a strong enthusiast of dynamic languages in general and Python in particular. See http://www.mindview.net/WebLog/log-0025

| IronPython 2 | The latest version of IronPython, built on top of the DLR. |
|---|---|
| IronRuby | A port of the Ruby language to run on .NET. Implemented by John Lam. |
| VBx | A dynamic version of Visual Basic. |
| Managed JScript | An implementation of ECMA 3.0, otherwise known as Javascript. |
| Vista Smalltalk[20] | A Smalltalk programming environment, which can run in the browser through the Silverlight browser plugin. |
| ToyScript[21] | A simple example language, illustrating how to create languages that use the DLR. |

As an IronPython programmer you needn't even be aware of the DLR, it is just an implementation detail of how IronPython (2) works. It does become relevant when you are embedding IronPython into another application; IronPython 2 has a different hosting API to IronPython 1. Fortunately they are both simple and we will be working with both.

There is another important consequence of the DLR, and this has to do with Silverlight, Microsoft's new browser plugin.

## 1.2.2  Silverlight, a New CLR

At Mix 2007, a conference for Web designers and developers, Microsoft surprised the world by announcing both the DLR *and* Silverlight.

Silverlight is something of a breakthrough for Microsoft. Their usual pattern is to release an early version of a project with an exciting sounding name, and then the final version with an anonymous and boring name[22]. This time round they have broken the mold; Silverlight was originally codenamed 'WPF/E', Windows Presentation Foundation Everywhere. Windows Presentation Foundation is the new user interface library that is part of .NET 3.0, and WPF/E takes it to the web.

What Silverlight *really* is, is a cross platform and cross browser[23] plugin for animation, video streaming and 'rich' web applications. The animation and video streaming features are aimed at designers who would otherwise be using Flash. What is more exciting for developers is that Silverlight 1.1 comes with a version of the CLR called the 'Core CLR'. The Core CLR is a cut-down .NET runtime containing (as the name implies) the core parts of the .NET framework. You can create web applications working with .NET classes that you are already familiar with. It all runs in a 'sandboxed' environment that is safe for running in a browser.

The best thing about Silverlight is that the DLR will run on it. So not only can you program Silverlight applications with C#, but you can also use any of the DLR languages, including IronPython. Figure 1.5 shows one of the Silverlight samples[24] that illustrates the dynamic power of DLR languages running in Silverlight.

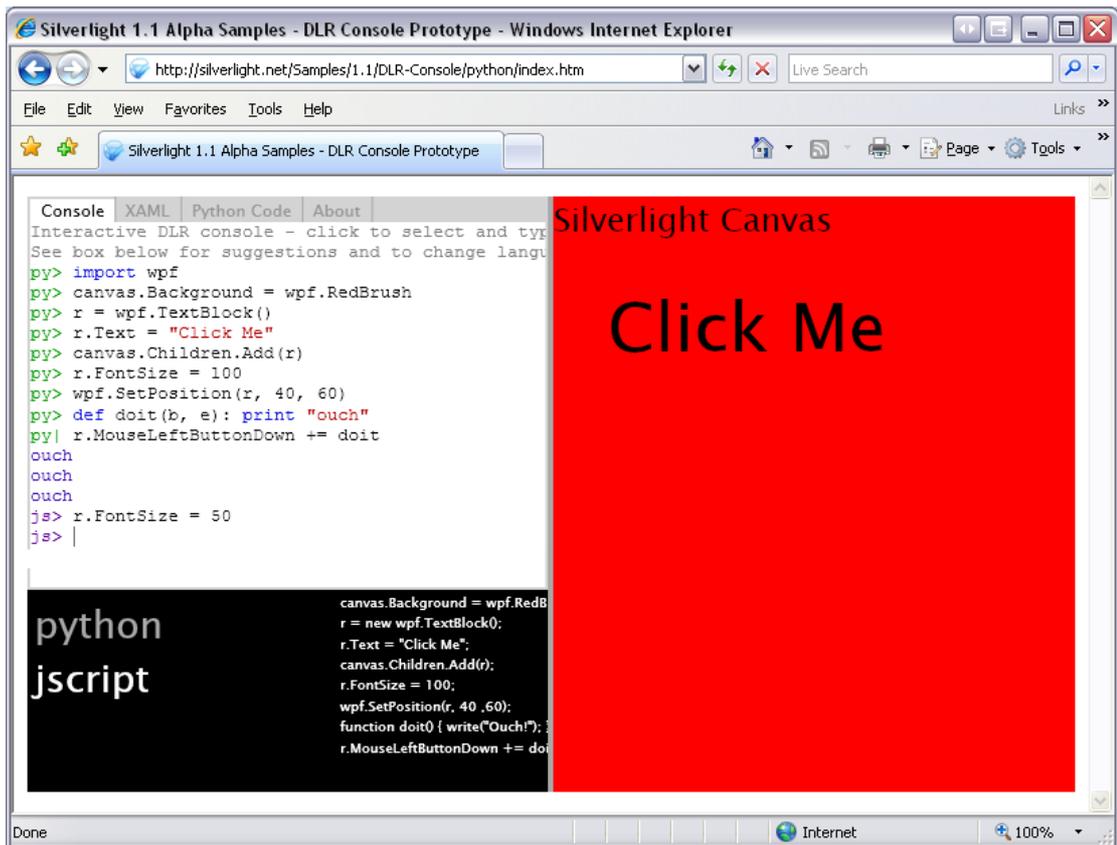**Figure 1.5 The Silverlight DLR Sample with A Python and Jscript Console**

---

[20] See http://vistascript.net
[21] See http://www.iunknown.com/2007/06/getting_started.html
[22] Take Avalon for example, which became WPF.
[23] Although Microsoft's idea of cross platform is Apple and Linux and their idea of cross browser is IE and Firefox. They have said that Opera support is in the works though.
[24] See http://silverlight.net/Samples/1.1/DLR-Console/python/index.htm

This is a console that allows you to execute code with live objects and see the results in the canvas to the right. The console supports both Jscript and IronPython and you can switch between them live!

There is another way of using Silverlight though, that I find particularly interesting. The whole browser DOM[25] is available to Silverlight, so it can interact with and change the HTML of webpages it is hosted in. It can also interact with normal un-managed Javascript. You can call into Silverlight code from Javascript, and vice-versa. It is already possible to write a client side web application with the presentation layer in Javascript, using any of the rich set of libraries available, and the business logic written in IronPython.

It is important to remember that whether it runs in the browser or on the desktop, IronPython code is Python code. In order to understand IronPython, and the place it has in the .NET world, you will need to understand Python. It is time for an overview of the Python language.

## 1.2.3        The Python Programming Language

### Python the Language

Python is an Open Source, high level, cross platform, dynamically typed, interpreted language.

---

[25] Document Object Model – a system that exposes a web page as a tree of objects.

Python is a mature language with a thriving user community. It has traditionally been used most on Linux and Unix type platforms, but with the predominance of Windows on the desktop Python has also drawn in quite a following in the Windows world. It is fully open source, with the source code available from the main Python website and a myriad of mirrors. Python runs on all the major platforms, plus many more including obscure ones like embedded controllers, Windows Mobile and the Symbian S60 platform used in Nokia.

Python was created by Guido van Rossum in 1990, whilst he worked for CWI in the Netherlands. It came out of his experiences of creating a language called ABC. This had many features that made it very easy to use, but also had serious limitations. In creating Python, Guido aimed to create a new language with the good features from ABC, but without the limitations.

Python is now maintained by a core of developers, with contributions from many more individuals. Guido still leads the development and is known as the *Benevolent Dictator for Life*, a term taken from a Monty Python sketch. Oh, and yes, Python **was** named after the Monty Python comedy crew.

Python itself is a combination of the runtime, called CPython, and a large collection of modules written in a combination of C and Python, collectively known as the standard library. The breadth and quality of the standard library has earned Python the reputation of being "Batteries included". The IronPython team has made a great deal of effort to ensure that as much as possible of the standard library still works. This means that IronPython is doubly blessed with a full set of batteries from the .NET framework and almost a full set of spares from the standard library.

One of the reasons for the rise in popularity of Python is the emphasis it places on clean and readable code. The greatest compliment for a Python programmer is not that his code is clever, but that it is elegant. To get a quick overview of the guiding philosophy of Python, type `import this` at a Python console[26]! You can see the result in figure 1.6.

**Figure 1.6 The Zen of Python, as enshrined in the Python Standard Library**

---

[26] The console will need the Python standard library on its path.

```
IronPython 1.0 (1.0.61005.1977) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Python is a multi-purpose programming language used for all sorts of tasks. It's possible you've already used applications or tools written in Python without even being aware of it. If you've used YouTube, Yahoo Maps or Google then you've been using tools built, or supported behind the scenes, with Python.

**Python Quotes**

> *"Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers"*, Cuong Do, Software Architect at YouTube.com.

> *"We chose to use python because we wanted a well-supported scripting language that could extend our core code. Indeed, we wrote much more code in python than we were expecting, including all in-game screens and the main interface",* -- Soren Johnson, lead designer, Civilization IV.

Python gets used for web development with web frameworks like Zope, Django and Turbogears. The BitTorrent application and Spambayes (an outlook plugin to combat spam) are both written in Python. Linux package manager Yum, M-net the distributed file store, MailMan the GNU mailing list manager and Trac a popular project management and bug tracking tool are all written in Python. It is used by companies which include Google, NASA, Sony Imageworks, Seagate and Industrial Light and Magic. It is also used a great deal by the scientific community, particularly (for no particular reason that I could deduce) in bioinformatics and genomics[27].

Because of the clarity of the syntax, Python is very easy to learn. Beginners can start with simple procedural style programming and move into object oriented programming as

---

[27] I've recently been informed that this is because of the simplicity and power of Python's string handling, ideal for slicing and splicing gene sequences.

they understand the concepts. It is the ability of Python to support multiple programming paradigms that we will look at next.

## *1.2.4     Multiple Programming Paradigms*

Python is fundamentally an object oriented programming language. Everything you deal with is an object. That doesn't mean that you are limited to the object oriented programming style of programming in Python.

**Object Oriented Programming**

Object Oriented Programming is a style of programming that originated in the 1960s, first appearing in a language called Simula.

The fundamentals of 'OOP' are that the basic components you work with in a program are 'objects'. Objects may have data associated with them, and also have 'methods' (basically functions) for working with that data. Methods may, or may not, take arguments and return values.

Objects have a type, their 'class', which defines their behavior. The class acts as blueprint for new objects. Individual objects are said to be 'instances' of their class.

Classes may also 'inherit' properties from parent classes. As an example, 'Snake' might be a class, and a 'Python' could be a sub-class of 'Snake'. 'Squasher', an individual, is an instance of  'Python'.[28]

**Procedural programming** is one alternative to object oriented programming, and is its predecessor. Python doesn't force you to create classes if you don't want to. With the rich set of built-in datatypes, and those available in the standard library, procedural programming works fine and is often appropriate for simple tasks. As many people have past experience with procedural programming it is common for beginners to start here. Few can avoid the allure of object oriented programming for long though.

**Functional programming** is an important programming concept.  Pure functional programming allows functions to have no side effects and can be hard to understand, but the basic concepts are straightforward though. In functional programming languages, Python included, functions are first class objects. This means that you can pass functions around and they can be used far from where they are created.

Because class and function creation is done at runtime rather than compile time, it is easy to have functions that create new functions or classes; so called class and function factories. These make heavy use of 'closures'. Local variables used in the same scope a function is defined in can be used from *inside* the function. They are said to have been captured by the function. This is known as lexical scoping[29].

You can have parts of your code returning functions that depend on the local values where they were created. These functions can be applied to data supplied in another part of your code. Closures can also be used to populate *some* arguments of a function, but not all of

---

[28] For a good introduction to Object Oriented Programming with Python, see
http://www.voidspace.org.uk/python/OOP.shtml
[29] See http://en.wikipedia.org/wiki/Lexical_scoping

them. A function can be wrapped inside another function with the populated arguments stored as local variables. The remaining arguments can be passed in at the point you call the wrapper function. This technique is called currying.

**Metaprogramming** is a style of programming that has been gaining popularity recently, through languages like Python and Ruby. It is normally considered an 'advanced' topic, but at times can be very useful. Metaprogramming techniques allow you to customize what happens at class creation time or when attributes of objects are accessed, not just individual attributes (through properties) but you can customize all attribute access.

By now I'm sure you're keen to actually use IronPython, and see what it has to offer for yourself. In the next section we look at the interactive interpreter and you get the chance to try some Python code that uses .NET classes. Some basic .NET terms are explained as we go. It is fairly straightforward but I won't be giving a blow-by-blow account of all the examples, explanations will come soon.

# 1.3 Live Objects on the Console: The Interactive Interpreter

Traditional Python (boy does that make me feel old) is an interpreted language. The source code, in a high level byte-code form, is evaluated and executed at runtime. Features like dynamic object lookup and creating and modifying objects at runtime fit very well with this model. The CLR also runs byte-code, but this is optimized to work with a powerful Just-in-Time compiler and so .NET languages tend to be called compiled languages.

Something else that fits well with dynamically evaluated code is an interactive interpreter, which allows you to enter and execute individual lines (and blocks) of code. By now you should have a good overview of what IronPython is, but in order to really get a feel for it you need to use it. The interactive interpreter gives you a chance to try some Python code. We'll be using some .NET classes directly from Python code. This is both an example of IronPython in Action and a demonstration of the capabilities of the interactive interpreter.

## 1.3.1 Using the Interactive Interpreter

Once you've downloaded the IronPython binary distribution[30] and unpacked it you will see two executables, 'ipy.exe' and 'ipyw.exe'. These are the equivalents of the Python executables 'python.exe' and 'pythonw.exe'. Both of these can be used to launch Python scripts. 'ipy.exe' will launch them with a console window, and 'ipyw.exe' launches programs without one.

```
ipy.exe path_to\python_script.py
```

If you run 'ipy.exe' on its own, it starts an interactive interpreter session. The IronPython interpreter supports tab completion and coloring of the output, both of which are useful. The command line options to enable these are:

---

[30] From the IronPython website on CodePlex: http://www.codeplex.com/IronPython

```
ipy -D -X:TabCompletion -X:ColorfulConsole
```

You should see something like figure 1.7.

**Figure 1.7 The IronPython Interactive Interpreter**

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS>ipy -D -X:TabCompletion -X:ColorfulConsole
IronPython 1.1 (1.1) on .NET 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
>>> _
```

IronPython is dependent on the .NET framework 2.0. The interpreter, and any applications created with IronPython, will only run on computers with .NET 2.0 (or Mono of course) installed. In recent days Microsoft has been pushing .NET 2.0 out to Windows machines via Windows update. A high proportion of Windows machines will already have .NET 2.0 installed. Windows machine that aren't .NET equipped will require at least the redistributable: 'dotnetfx.exe'[31].

The interactive interpreter allows you to execute Python statements and see the results. This is known as a 'read-eval-print' loop[32]. It can be extremely useful for exploring objects, trying out language features or even performing quick one-off operations.

If you enter an expression, the resulting value will be printed to the console. The result of the last expression, whether value or object, is available in the interpreter by using the name '_'. If you can't find a calculator, then you can always turn to the Python interpreter instead.

### Interpreter Code Examples

Code examples that are intended to be typed into an interactive interpreter session start each line that you enter with '>>>' or '...'. This reflects the actual appearance of the session and is a common convention when presenting Python examples.

```
>>> 1.2 * (64 /2.4) + 36 +2 ** 5
100.0
>>> _
100.0
>>> x = _
>>> print x
100.0
```

More importantly, blocks of code can be entered into the interpreter, using indentation in the normal Python manner.

---

[31] From the memorable URL: http://www.microsoft.com/downloads/details.aspx?FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en
[32] From the first appearance of an interactive interpreter with the Lisp language.

```
>>> def checkNumberType(someNumber):
…    if type(someNumber) == int:
…        print 'Yup, that was an integer'
….    else:
…        print 'Nope, not an integer: %s' % type(someNumber)
>>> checkNumber(2.3)
Nope, not an integer: <type 'float'>
>>> type(checkNumber)
<type 'function'>
```

We're not going to get very far in this book without some understanding of .NET terminology. Before demonstrating some more practical uses of the interpreter we'll look at some basic .NET concepts.

## 1.3.2 The .NET Framework: Assemblies, Namespaces and References

Assemblies are the substance of a .NET application, compiled code; in the form of dlls or 'exe' files. Assemblies contain the types that are used throughout an application.

The types in an assembly are contained in namespaces. If there are no explicitly defined namespaces they are contained in the default namespace. Assemblies and namespaces can be spread over multiple physical files and an assembly can contain multiple namespaces. Assemblies and namespaces are roughly the same as 'packages' and 'modules' in Python, but are not directly equivalent.

The important thing to know is that for a program to use a .NET assembly it must have a *reference* to the assembly. This must be explicitly added. In order to add references to .NET assemblies we use the 'clr' module. In Python the term 'module' has a different meaning to the rare, but occasionally used, .NET module. In order to use 'clr' we have to import it, and then we can call 'AddReference' with the name of the assembly we want to use.

Having added a reference to the assembly we are then free to import names from it into our IronPython program and use them.

```
>>> import clr
>>> clr.AddReference('System.Drawing')
>>> from System.Drawing import Color, Point
>>> Point(10, 30)
<System.Drawing.Point object at 0x0...2B [{X=10,Y=30}]>
>>> from System.Drawing import Color
>>> Color.Red
<System.Drawing.Color object at 0x0...2C [Color [Red]]>
```
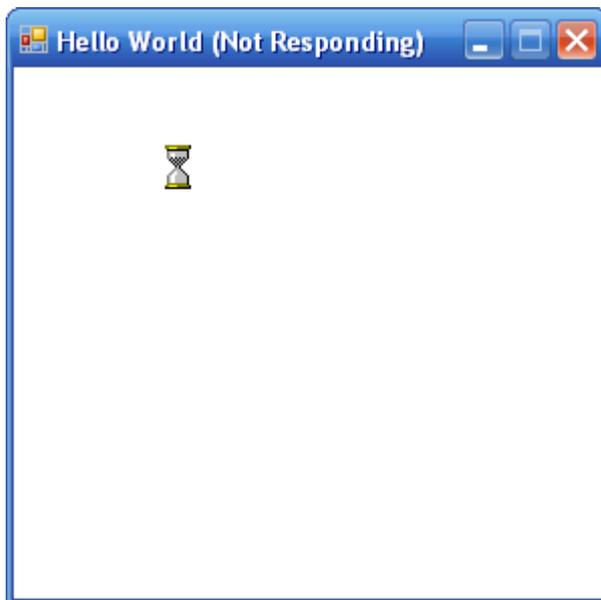
There are a few exceptions. The IronPython engine already has a reference to the assemblies it uses, like 'System' and 'IronPython'. There is no need to add explicit references to these.

The 'clr' module includes more goodies. It has different ways of adding references to assemblies, including specifying precisely which version you require. We'll take a more detailed look at some of these later in the book. For now let's see how we can use live objects from the interactive interpreter.

## 1.3.3 Live Objects and the Interactive Interpreter

The interactive interpreter is shown off at its best when it is used with live classes. To illustrate this here is some example code using Windows Forms. It uses the 'System.Windows.Forms' and 'System.Drawing' assemblies.

**Figure 1.8 A Hello World Form, shown before the event loop is started.**



```
>>> import clr
>>> clr.AddReference('System.Windows.Forms')
>>> clr.AddReference('System.Drawing')
>>> from System.Windows.Forms import Application, Button,
Form     | #1
>>> from System.Drawing import Point
>>> x = 0
>>> y = 0
>>> form = Form()    | #2
>>> form.Text = "Hello World"
```
(annotation) <#1 Import the names we need>
(annotation) <#2 Instantiate a form>

So now you should have created a form, with the title "Hello World". Because we haven't yet started the application loop there is no guarantee that it will be visible, even if it is it will be unresponsive, as you can see in figure 1.8.

**Figure 1.9 Active 'Hello World' Form, with a button.**

```
>>> button = Button(Text="Button Text")    | #3
>>> form.Controls.Add(button)              | #4
>>> def click(sender, event):              | #5
...     global x, y
...     button.Location = Point(x, y)
...     x += 5
...     y += 5
...
>>> button.Click += click                  | #6
>>> Application.Run(form)                   | #7
```

(annotation) <#3 Instantiate a button>
(annotation) <#4 Add it to the form>
(annotation) <#5 Define a click handler function >
(annotation) <#6 Add our handler function>
(annotation) <#7 Start the application loop>

When the application event loop is started and the form is shown. The form will look like figure 1.9. Every time you click the button, it will move diagonally across the form.

We'll cover all the details of what is going on here later, the important thing is that the whole process was done live.

If you run this demonstration you'll notice that when you execute the 'Application.Run(form)' command, the console loses control. Control doesn't return to the console until you exit the form. This is because starting the application loop takes over the thread it happens on.

**The IronPython winforms Sample**

There are various pieces of sample code available for IronPython. One of these is a useful piece of code called 'winforms'.

If you run the IronPython console from the 'tutorial' directory and 'import winforms' then the console is set up on another thread and commands you enter are

invoked back onto the GUI thread. You can create live GUI objects, even though the application loop has started, and see the results immediately.

As well as being able to work with live objects, Python has powerful introspective capabilities. There are a couple of convenience commands for using introspection that are very effective in the interpreter.

## 1.3.4 Object Introspection with 'dir' and 'help'

It is possible when programming to occasionally forget what properties and methods are available on an object. The interpreter is a great place to try things out and there are two commands that are particularly useful.

'dir(object)' will give you a list of all the attributes available on an object. It returns a list of strings, so you can filter it or do anything else you might do with a list. The snippet below looks for all the interfaces in the 'System.Collections' assembly by building a list and then printing all the members whose name begins with 'I'.

```
>>> import System.Collections
>>> interfaces = [entry for entry in dir(System.Collections)
... if entry.startswith('I')]
>>> for entry in interfaces:
...     print entry
...
ICollection
IComparer
IDictionary
IDictionaryEnumerator
IEnumerable
IEnumerator
IEqualityComparer
IHashCodeProvider
IList
>>>
```

The next command is 'help'. 'help' is a built-in function for providing information on objects and methods. It can tell us the arguments that methods take, and for .NET methods it can tell us what types of objects the arguments need to be. Sometimes the result contains other useful information.

As an example, let's use 'help' to have a look at the 'Push' method of 'System.Collections.Stack':

```
>>> from System.Collections import Stack
>>> help(Stack.Push)
Help on method-descriptor Push
 |  Push(...)
 |      Push(self, object obj)
 |
 |          Inserts an object at the top of the
```

```
|           System.Collections.Stack.
|
|          obj: The System.Object to push onto the
|           System.Collections.Stack. The value can be null.
|
```

'help' will also display 'docstrings' defined on Python objects. 'docstrings' are strings that appear inline with modules, functions, classes and methods that explain the purpose of the object. Many members of the Python standard library have useful docstrings defined. This is what we get if use 'help' on the 'makedirs' function from the 'os' module:

```
>>> import os
>>> help(os.makedirs)
Help on function makedirs in module os
|   makedirs(name, mode)
|        makedirs(path [, mode=0777])
|
|           Super-mkdir; create a leaf directory and all
intermediate ones.
|           Works like mkdir, except that any intermediate
path segment (not
|           just the rightmost) will be created if it does
not exist.  This is
|           recursive.
|
```

### Help, the 'site' module, and the Python Standard Library

If you have given the interpreter access to the Python standard library (through the IRONPYTHONPATH environment variable), it breaks 'help'. This is because 'ipy.exe' will import the 'site' module which overwrites the IronPython 'help'.

The solution is to place an empty file, called 'site.py', in the same directory as 'ipy.exe'. This prevents IronPython from importing the one from the Python standard library.

You can customize interactive interpreter sessions by adding commands to your 'site.py' file, which will be imported every time the interpreter starts.

The interpreter is an extremely useful environment for exploring .NET assemblies. You can import objects and construct them live, to explore how they work. It is also useful for one off jobs, like transforming data. You can pull the data in, push it back out again, and walk away[33].

We've now completed our introductory tour of IronPython and the interactive interpreter. It's time for a summary, before moving on to learning more about Python the programming language.

---

[33] This is a quote from a Python expert called Tim Golden who does a lot of work with databases and Python, some of it just using the interactive interpreter.

## 1.4 Summary

We've seen that Python is a flexible and powerful language suitable to a range of tasks. Not only that, but IronPython is a faithful implementation of Python for the .NET platform, which itself is no slouch when it comes to power and features. Whether you are interested in writing full applications, tackling scripting tasks, or embedding a scripting language into another application, IronPython has something to offer you.

Through the course of this book we will be demonstrating these different practical uses of IronPython. There are also sections that providing reference matter for the hard work of turning ideas into reality when it comes to real code.

Important for experimentation is the interactive interpreter. The interpreter is a great tool for trying things out, reminding you of syntax or language features and for examining the properties of objects. The `dir` and `help` commands illustrates the ease of introspection with Python, and I expect that as we work through more complex examples they will be helpful companions on the journey.

In the last section we got our feet wet with Python, but before we can do anything useful with it we'll need to learn a bit more of the language. The next chapter is a fast paced tutorial that will lay the foundations for the examples we build in the rest of the book.