Developing with IronPython

Developing with IronPython

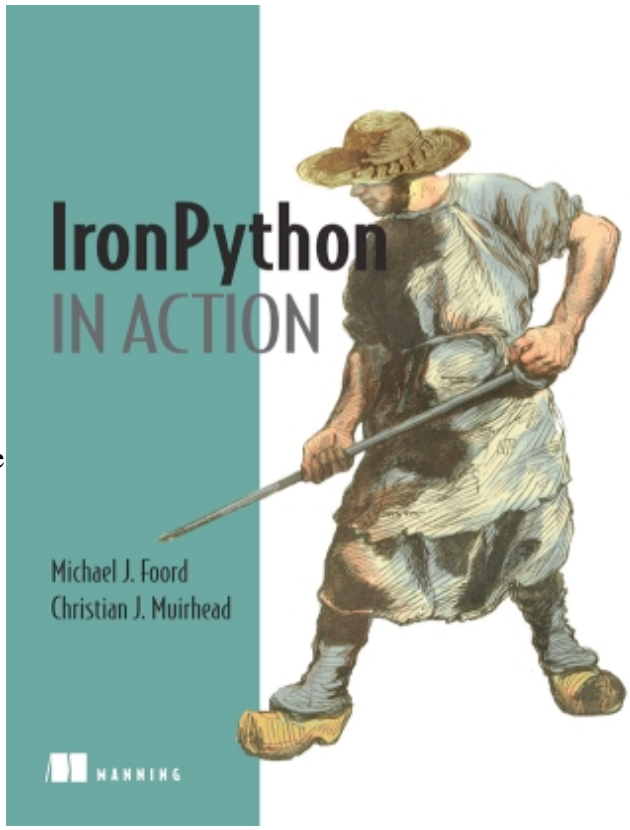
- Reading Results
- Practical 6 : Database access
- IronPython and Threads
- Basic Principles
- Thread Class & ThreadStart Delegate
- Aborting Threads
- Cleaning up in a Finally Block
- Windows Forms and Threads
- The CallTarget0 Delegate
- Invoking onto the Control Thread
- Practical 7: Fetching Tweets in the Background

- www.ironpythoninaction.com
- tartley.com
- Resolver Systems

This tutorial is taken by:

- Michael Foord
- Jonathan Hartley

Both of us work for Resolver Systems in London, creating a spreadsheet development environment with IronPython. There are currently around 40 000 lines of IronPython code in Resolver One, plus around 130 000 lines in the test framework.



Michael is also the author of *IronPython in Action*, the first (English) book on IronPython - published by Manning Publications.

It covers topics like:

- Windows Forms and Windows Presentation Foundation
- Databases, web services and XML
- Silverlight
- Extending and embedding IronPython with C# and VB.NET
- Metaprogramming, testing and design patterns
- System administration (shell-scripting, WMI and Powershell with IronPython)
- ASP.NET

Introduction to IronPython

- IronPython is a Python *compiler*
- Runs on .NET and Mono

- Originally created by Jim Hugunin
- Now being developed by a Microsoft team
- Version 2, built on the Dynamic Language Runtime (Python 2.5)
- Version 2 runs on Silverlight (and Moonlight 2 on Linux)

IronPython was originally created as an experiment to demonstrate how bad the .NET framework was for dynamic languages.

His early version ran the Pystone benchmark about 1.7 times faster than CPython, so it became a serious project. Eventually he was hired by Microsoft to work on the CLR (Common Language Runtime) and they took on IronPython - releasing it with an OSI approved open source license.

After completing IronPython 1 they abstracted out the dynamic type system and dynamic language hosting environment to create the Dynamic Language Runtime. As well as making it easier to implement dynamic languages for .NET, the DLR allows dynamic languages to interoperate through a shared type system. IronPython 2 is built on the DLR. Other (Microsoft) DLR languages include IronRuby (still in alpha) and Managed JScript (an ECMA 3 compliant implementation of Javascript for Silverlight).

Visual Basic .NET 10 and C# 4.0 (.NET 4.0) will be 'partially dynamic' through the DLR - which will become part of the Common Language Runtime. It will include mechanisms for easier interaction with 'dynamic objects' (possible now through the IronPython hosting API - but a bit clumsy).

IronPython is a Python compiler - it compiles Python in-memory to IL (Intermediate Language), the .NET bytecode (which can be compiled to native code by the JIT). It 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.

The .NET Framework

- Microsoft's 'grand' programming framework for Windows
- Powerful application runtime
    - JIT compiler
    - Automatic garbage collection
    - **Huge** libraries (ASP.NET, winforms, WPF, ADO, WCF...)
- Cross-platform with Mono
- 'Standard' programming language is C# ('Java done right')

Advantages of IronPython

- Access to .NET libraries
- Easy to extend with C#

- Political reasons!
- Silverlight!!
- Binary distributions or apps with multiple 'Python engines'
- A better threading model than CPython (no GIL)
- Sandbox code with AppDomains

IronPython has several advantages over CPython for the Python programmer.

These include:

- Interoperation with existing .NET components including the huge .NET libraries - in particular the WIndows Forms and WPF user interface libraries
- Extending IronPython with C# (for performance reasons) is dramatically easier than extending CPython with C (IronPython can use .NET class libraries natively)
- IronPython (based on .NET) may be an easier sell politically (in a corporate IT environment) than CPython
- Script the browser with Silverlight
- Create binary distributions of applications, or applications with multiple Python engines (difficult with CPython)
- Applications can execute code inside AppDomains with restricted privileges (again - hard in CPython)
- .NET has a better threading model than CPython (no Global Interpreter Lock and the ability to abort threads) - so you can create threaded applications that use multiple CPU cores

Disadvantages of IronPython

- No Python stack frames & other incompatibilities
- All strings are Unicode in IronPython
- No C extensions - Ironclad for the future though
- Performance profile is very different
- Dependency on .NET or Mono

There is a cost to using IronPython of course.

A substantial part of the Python standard library (or existing Python code) *does* work. However, the performance profile for IronPython is very different to CPython (function calls and arithmetic are massively faster - using the builtin types is slower). This means that code optimised *for CPython* will typically run slower on IronPython.

IronPython is not *fully* compatible with CPython though - it doesn't use Python stack frames for example. Code that depends on CPython 'implementation details' is unlikely to work.

Another potential source of incompatibility (although it seems to be less of a problem in practice) is that all strings are Unicode in IronPython (like Python 3). This actually makes working with strings much more pleasant!

The single biggest incompatibility problem is that Python C extensions don't work. Several of these (particularly the builtin modules) have been re-implemented by the IronPython team in C#. For others, the community has provided compatibility wrappers around an equivalent .NET library. You can find several of these in FePy: the IronPython Community Edition (maintained by Seo

Sanghyeon).

For the rest, one possibility is Ironclad. This is an open source project by Resolver Systems, and is basically a re-implementation of the Python C API in C#. Although by no means complete, over 900 of the Numpy tests pass (and we have used this to build Numpy integration into Resolver One) and hundreds of Scipy and matplotlib tests also pass.

Connecting Python to .NET

- Two *orthogonal* ways of organising classes in .NET
    - Assemblies
    - Namespaces
- Use clr.AddReference to make classes in an assembly available for import

```
import clr # built-in to IronPython
clr.AddReference('System.Data')
```

- Assemblies are found by searching sys.path (plus some other locations)

.NET Assemblies physically partition an application or library into one or more .Net EXE or DLL files. The 'System.Data' name above corresponds to a 'System.Data.dll' file that will be located on the system path.

One or more source code files can be compiled to create a single assembly.

Assemblies don't really have any Python analogue. (except maybe the kind of sys.path manipulations big projects might do to ensure modules are available for importing.)

Importing .NET Classes

- Classes in a referenced assembly are now available for import from their namespace

```
from System.Data.Common import DbConnection
```

.NET Namespaces represent the logical partitioning of an application or library. Namespaces behave similarly to Python's packages and modules (although they're read-only).

For example, once a reference is added to the System.Data assembly, the following imports are valid:

```
# Get a reference to the to level namespace 'System'
# 'System' will be represented in IP as an object of type 'module
import System

# Or, get a reference to the contained namespace 'Data'
# Again, 'Data' will be an instance of 'module'
from System import Data

# Or, get a reference to a sub contained namespace 'Common'
# Again, as an instance of 'module'
```

```
from System.Data import Common

# Or, get a reference to a class within the namespace
# 'DbConnection' is a class.
from System.Data.Common import DbConnection
```

Often a namespace will have a one-to-one relationship with an assembly, but it doesn't necessarily have to. A namespace can be split between several assemblies, or a single assembly can implement several namespaces.

An assembly's name doesn't have to be the common prefix to all of the namespaces whose classes it contains, but usually that's how things are organised - it's easier to manage.

Two assemblies can each define classes in the same namespace. Say assembly Fish defines the class Cod in the Species.Fauna namespace, and assembly Mammals defines Dog in the same namespace. You could add references to Fish and Mammals, then import both classes from the namespace, eg:

```
import clr
clr.AddReference("Mammals")
clr.AddReference("Fish")
from Species.Fauna import Cod, Dog
```

Python and .NET Types (1)

- IronPython maps Python's built-in types to the equivalent .NET types

```
>>> import System
>>> int is System.Int32
True
>>> str is System.String
True
```

- Cleverness about what methods are visible: str.upper() vs String.ToUpper()
- None gets treated as null

The standard Python methods on builtins are always available, but the .NET ones don't show up unless the clr module has been imported.

Having the standard Python types actually be the .NET types means that the interface with normal C# code can be very natural, usually without any need for conversions at the interface.

Python and .NET Types (2)

- Built-in containers are custom types which implement standard .NET interfaces
    - list -> IList
    - dict -> IDictionary
- .NET types you might see in the wild:
    - Enumerations: nicer constants/flags for APIs
    - Structs: like classes, but are value types

    - Delegates: statically typed function pointers

Often APIs you're using in the framework expect enumerables, ILists or dictionaries, so you can pass them straight in from IronPython code without needing to do any translation.

Delegates are heavily used in the .NET framework for callbacks, especially for event handlers in the GUI toolkit. In a lot of cases, IronPython can detect what delegate type is expected in a particular place and automaticaly wrap function references, as long as the number of arguments is right.

Subclassing .NET Types

- IronPython classes can inherit from .NET classes
- Override methods on base class
    - Should be signature compatible with the overridden method
- Doesn't create new .NET classes for each Python class
    - One shared between all subclasses of each .NET base class
- .NET code can call into Python code
    - But only from .NET base class references

Arrays and Generics

- Uses similar syntax for creating Arrays

```
from System import Array
a = Array[float]([1.2, 2.2, 4])
```

- And for instantiating generics

```
from System.Collections.Generics import SortedDictionary
d = SortedDictionary[str, list]()
```

The Python containers are generally more useful/flexible than arrays and a lot of generic classes, but sometimes they're necessary for talking to an API. The .NET generic collections can also be useful if you want to use a container with different performance tradeoffs than the Python ones.

.NET classes can have *overloaded* methods - distinguished by types of arguments:

```
ExampleClass.Add(Int32, Int32)
ExampleClass.Add(String, String)
```

Often IP can determine the correct method from the types of arguments, occasionally it needs help:

```
ExampleClass.Add.Overloads[int, int]('hello', None)
```

IronPython subclasses of the same C# base class will be indistinguishable to .NET's static typing.

.NET can only tell the difference between classes that are distinct at the CLR level. If you have two

different subclasses of object, and an array that you only intended to contain one type, the instances of the other type can still be stored in it, because as far as the CLR is concerned they are of the same type.

```
class Custom1(object): pass
class Custom2(object): pass

c = Custom1()
a = Array[Custom1]([c])
a[0] = Custom2()  # this will work
```

Making GUIs

- The Windows Forms GUI framework lives in System.Windows.Forms
- That's the assembly and the namespace
- Hello world:

```
clr.AddReference('System.Windows.Forms')
from System.Windows.Forms import Application, Label, Form
form = Form()
label = Label(Parent=form, Text='Hello world!')
Application.Run(form)
```

The constructor for the Label actually takes no parameters; the keyword arguments are IronPython syntactic sugar to make constructing GUIs a little more convenient. It's equivalent to:

```
label = Label()
label.Parent = form
label.Text = 'Hello world!'
```

User Interaction

- Add controls with event handlers to forms

```
button = Button(Parent=form, Text='hey')
def onClick(sender, args):
    print 'quit it'
button.Click += onClick
Application.Run(form)
```

- onClick function is automatically wrapped in EventHandler delegate
- More on this later...

Introducing Stutter

- Yet Another Twitter Client
- Very simple

- Missing some bits which we're going to implement now
- Let's see what it looks like...

Stutter is a simple application, written in IronPython, that talks to the Twitter service (http://twitter.com/). Twitter offers excellent web APIs that allow you to see what your friends are up to, as well as update your own status. Stutter takes advantage of these APIs to retrieve "tweets" and submit new tweets. Tweets are cached to a database to keep a long history and avoid having to bother the Twitter service so much.

Creating a Form

- Create a subclass of Form
- in __init__():
    - call the superclass!
    - Create control instances
    - Add controls and lay them out
- Controls added using self.Controls.Add()

To create a form in .NET you'll need to create a subclass of the .NET Form class. In the constructor you can set up properties of the form and create control instances (eg. TextBox, Button, Panel) which should appear on the form. It's useful to keep references to the important controls as instance variables for later use.

Controls which hold other controls are known as container controls. Contain controls include Form and Panel. By combining such controls you can construct complex layouts. More on this later.

Useful links:

- System.Windows.Forms (v2.0): http://msdn.microsoft.com/en-us/library/system.windows.forms.form(VS.80).aspx
- System.Windows.Forms (v3.5): http://msdn.microsoft.com/en-us/library/system.windows.forms.aspx

Creating a Form: Example

```
class MainForm(Form):

    INITIAL_WIDTH = 600
    INITIAL_HEIGHT = 600

    def __init__(self):
        Form.__init__(self)
        self.Text = 'stutter'
        self.Height = self.INITIAL_HEIGHT
        self.Width = self.INITIAL_WIDTH

        self.postTextBox = TextBox()
        self.postTextBox.Multiline = True
```

```
        self.friendsListBox = ListBox()
        ...
```

This is a sample from our Stutter application (MainForm.py). We create a subclass of Form called MainForm. In the constructor, the Form constructor is called (rather important) and then we set the title ("Text"), Height and Width of the form. Then it's just a matter of creating a few controls and configuring them.

Practical 1

Add a Post button

In the MainForm.__init__, create new .NET Button, and store it as self.postButton. Label the button 'Post'.

In the _layout() method, add this button, not to the form itself, but to the form's upper panel control.

Laying Out the Form (1)

- On Windows you can use Visual Studio.
- but... programmatic layout works everywhere
- Programmatic layout is required for highly dynamic interfaces
- Parameterising layout

On Windows, most people use Visual Studio to do layout. This works ok but isn't an option for Unix or Mac users. (Mono Develop has a GTK# designer and a Windows Forms designer is in the works.)

Programmatic layout works on all platforms but can be tedious. It is necessary when dynamic interfaces need to built at runtime. One advantage of programmatic layouts is that they can be quick to change if parameterised correctly. Such changes can often be time-consuming if using a visual design tool.

Laying Out the Form (2)

- Panel, GroupBox and others
- Simple positioning done using attributes:
    - Top, Left, Width, Height
- In pixels, relative to parent container
- Use ClientSize to determine the area inside a Control

To arrange controls, container controls such as Panel or GroupBox are used. Container controls can usually hold other container controls. By using a mix of controls the required layout can be acheived.

Most .NET controls have 4 attributes which can be used to do simple positioning and sizing: Top, Left, Width, Height. These values are in pixels and are normally applied relative to the parent container.

The ClientSize attribute of a control indicates the space inside a control. This attribute is a Size object; it itself has Height and Width attributes.

Anchoring and Docking

- What about when the window is resized?
- Ideally, controls should resize with the form
- Two solutions:
    - Docking
    - Anchoring
- You should choose one mechanism: they don't work well together

Practical 2

Layout in Stutter

- Look inside MainForm.py: _layout() method
- Uncomment each of the lines, observing the effect of each.
- See how controls are added to form.
- Containers like the panels have further controls added to them.
- Style notes:
    - hard-coded numbers are avoided
    - positions and sizes of controls are relative to other controls
    - easy to maintain

Docking

- Keeps a control stuck to a particular side of it's parent
- Can stick a control to one side of it's parent, or all sides (Fill).
- Set via Dock property using the DockStyle enumeration:
    - Left, Right, Top, Bottom, Fill
- Docked controls resize to always be completely attached to the given side of the parent

Anchoring

- Keeps a control's edge(s) a fixed distance from the parent
- Set via the Anchor property using the AnchorStyles enumeration
    - Top, Left, Right, Bottom
- Unlike Dock these can be combined to anchor to more than one edge
- An Anchored control will keep a constant distance to the parent edge(s)
- Controls will resize as required to meet the contraints

Useful links:

- Anchor property: http://msdn.microsoft.com/en-us/library/system.windows.forms.control.anchor.aspx
- AnchorStyles: http://msdn.microsoft.com/en-us/library/system.windows.forms.anchorstyles.aspx
- Dock property: http://msdn.microsoft.com/en-us/library/system.windows.forms.control.dock.aspx
- DockStyle: http://msdn.microsoft.com/en-us/library/system.windows.forms.dockstyle.aspx

Dropdown Menus

- Dropdown menu bar is handled by the MenuStrip control
- Each menu is ToolStripMenuItem Add()'ed to MenuStrip.Controls
- Each menu item is a ToolStripMenuItem Add()'ed to the parent's DropDownItems
- MenuStrip instance is then added to Form.Controls
- Also need to set MainMenuStrip on Form

Menu bars are a common feature of most applications. They are easily implemented using .NET.

The top-level object representing the menu bar is MenuStrip.

MenuStrip are loaded with ToolStripMenuItem objects (via Controls.Add). These represent each menu ("File", "Edit" etc).

These ToolStripMenuItem objects are passed further ToolStripMenuItem objects for each menu item (eg. "New", "Open", "Quit") via DropDownItems.Add().

The Form is given the menu bar by adding the MenuStrip instance to Controls and setting the MainMenuStrip attribute.

Practical 3: Add Menu Items

MainForm currently has a dropdown menu but no menu items on it.

- Add 2 menu items:
    - Refresh
    - Quit
- Remember to keep references to the menu items created for later.
- The main menu strip also needs to be added to its parent and docked at the top.

Useful links:

- ToolStripMenuItem class: http://msdn.microsoft.com/en-us/library/system.windows.forms.toolstripmenuitem.aspx

Practical 4: Event Handlers

- Event handlers are hooked up in Stutter.py
- In the constructor we need to add handlers for:
    - the Refresh menu item (function 'onRefresh' already exists)
    - the Quit menu item

Reminder of how event handlers are set ...:

```
ToolStripMenuItem.Click += myHandler

def myHandler(self, source, args):
    pass
```

The 'myHandler' function will be called when the event is fired.

Background: Twitter API

- RESTful web API
- Allows querying of users & tweets in a variety of ways & updates
- Authentication is done using HTTP Basic Auth (401)
- Variety of return types indicated by extension:
    - JSON, RSS, Atom, XML (which we'll use)
- Stutter only needs a small subset of the API:
    - GET: statuses/friends_timeline.xml
    - POST: statuses/update.xml

Twitter offers a well designed web API that applications can use to retrieve and create status updates and private messages. Twitter can report information in a variety of formats, selected by the extension of requested resource.

Twitter's API has been designed using the REST (Representational State Transfer) style. Among other things, this means it's easy to write clients that use the API using just basic web requests.

Useful links:

- Twitter API: http://apiwiki.twitter.com/REST+API+Documentation
- REST: http://en.wikipedia.org/wiki/Representational_State_Transfer

.NET Streams

- .NET streams represent are sequential bytes:
    - files
    - network connections
- Used by the .NET API where I/O is done
- A little over-engineered, but ok
- You get a friendlier API by wrapping a Stream
    - StreamReader
    - StreamWriter
- Need Streams to use the .NET web APIs

Streams appear throughout the .NET API where sequences of bytes need to represented (typically when doing file or network I/O). The Stream base class is generic. There are various subclasses of Stream for different uses (eg. FileStream, MemoryStream, BufferedStream).

Various .NET methods will return Stream instances which your code can use or pass to other APIs. The Stream interface itself isn't easy to use. A friendlier way of dealing with Streams is wrap them with a StreamReader or StreamWriter. These provide convenient methods for manipulating stream data.

Making Web Requests

- Using .NET, web requests are made using the System.Net.WebRequest
- To use:
    - Create a WebRequest
    - Set the Method attribute ('GET' or 'POST')
    - Set Credentials attribute if authenentication is required
    - If POST use the request's stream to add POST data (GetRequestStream())
    - Call GetResponse() on the request to get the response object
    - Call GetResponseStream() on the response to get out response text

Useful links:

- WebRequest: http://msdn.microsoft.com/en-us/library/system.net.webrequest.aspx
- StreamReader: http://msdn.microsoft.com/en-us/library/system.io.streamreader(VS.80).aspx
- StreamWriter: http://msdn.microsoft.com/en-us/library/system.io.streamwriter(VS.80).aspx

Note

There is also a useful alternative for making web requests called WebClient. For making simple requests you can do it on one line:

```
from System.Net import WebClient

page = WebClient.DownloadString(uri)
```

For our use case we need a stream to pass to our XML parser rather than a string, so WebRequest is actually more convenient.

- http://msdn.microsoft.com/en-us/library/system.net.webclient.aspx

Practical 5: Adding Update Support

- Add an update() method to twitter.Client.Client
- Make a POST to http://twitter.com/statuses/update.xml
- One POST argument (status) is required which gives the tweet text
- Don't forget authentication!
- This update method is called from the Post button handler hooked in stutter.py

Useful links:

- Twitter update API: http://apiwiki.twitter.com/REST+API+Documentation#update

XML Parsing

- The Twitter API can return tweet timelines as XML
- Stutter uses the .NET API from System.Xml
- .NET's main parser API is at System.Xml.XmlReader
- XmlReader.Create() takes TextReader
    - the StreamReader we used previously is a TextReader
- Call Read() method to move to next node

Useful links:

- XmlReader: http://msdn.microsoft.com/en-us/library/system.xml.xmlreader(VS.80).aspx

XML Parsing: Example

```
reader = XmlReader(someTextReader)
```

```
while reader.Read():
    if reader.NodeType == XmlNodeType.Element:
        goHandleThisElement(reader)
    elif reader.NodeType == XmlNodeType.Text:
        goHandleThisText(reader)
    ...
```

Cleaner XML Handling

- XML API is functional but not elegant (it's for C# programmers)
- With a dynamic language like IronPython we can do better
- Subclass XmlDocumentReader (xmlreader.py)
- Add methods to handle nodes you care about
    - `onStart_...` for start tags
    - `onEnd_...` for end tags
    - `onText_...` for text between tags

It is common to wrap the very general .NET XML parsing APIs with a intermediate layer to make them more convenient to use. With a dynamic language like IronPython we can create an API that is elegant, requiring minimal programming effort to use.

For Stutter we've to chosen wrap XmlReader to create a SAX/event-driven style parser. This is implemented in xmlreader.py as the XmlDocumentReader class. To use, subclass XmlDocumentReader and then add methods to handle XML nodes as they are encountered in the stream. The method names should include the event type (eg. onStart...) and the element name. For example to handle the start of the status tag you would just add a method called onStart_status. The rest is handled by XmlDocumentReader. Nodes that don't have a corresponding handler method are ignored.

We'll look at how to use this class to parse Twitter statuses soon.

Twitter Statuses

```
<?xml version="1.0" encoding="UTF-8"?>
<statuses type="array">
  <status>
    <created_at>Thu Aug 14 12:00:49 +0000 2008</created_at>
    <id>887300130</id>
    <text>How long do Britons talk, text, surf and watch?</text>
    ...
    <user>
      <id>621583</id>
      <name>BBC Technology</name>
      <screen_name>bbctech</screen_name>
      <location>London, UK</location>
      ...
    </user>
```

```
  </status>
...
</statuses>
```

This is an example of what a Twitter status timeline looks like in XML format. I've added whitespace to make it more readable. In practice, all unnecessary whitespace is stripped to reduce network overhead.

See also: http://apiwiki.twitter.com/REST+API+Documentation#friendstimeline

Parsing Twitter Statuses

- Subclass that handles Twitter statuses:
    - twitter.StatusReader.StatusReader
- To use:
    - Create an instance
    - call read() with a StreamReader
    - get back a list of dicts with another dict for the user

Parsed Statuses

```
[ {
  'id': '71283719'
  'text': 'Feeding the cat',
  ...
  'user': {
     'id': '132801',
     'location', 'Nether Wallop, UK',
     ...
   }
}, ... ]
```

Sample of the return value of StatusReader.read()

Databases: ADO.NET

- Connect to lots of different DBMSs
    - SQL Server
    - Oracle
    - SQLite
    - PostgreSQL
    - ...
- Two parts
    - Database-specific Data Providers
    - Generic DataSets

DataSet Layer

- Disconnected in-memory data storage and manipulation
- Huge!
- Lots of facilities for slicing and dicing data
- ... which aren't really needed in Python

Data Provider Layer

- Database specific
    - Some come with framework
    - Others provided by DBMS groups
- Classes for getting at data
    - Connection
    - Command (+ Parameter)
    - Reader
    - Transaction

More information about ADO.NET: http://msdn.microsoft.com/en-us/data/aa937722.aspx

Getting Connected

- Make sure your data provider is on sys.path

```
clr.AddReference('System.Data.SQLite')
from System.Data.SQLite import SQLiteConnection
connection = SQLiteConnection(
    'data source=d:\play\tutorial\db.sql3')
connection.Open()
```

- Connection class name and connection string parameters depend on provider

Issuing Commands

- With the Command class!

```
command = connection.CreateCommand()
command.CommandText = 'select count(*) from tweets'
command.ExecuteScalar()
```

- ExecuteScalar returns first value from first row of results
- ExecuteNonQuery for INSERT, UPDATE and DELETE
- ExecuteReader for reading results

Avoid SQL Injection!

- Use parameters!
- Faster queries
- No annoying quoting of strings or date values

```
command.CommandText = '''
       select * from tweets
       where friend_id = :id'''
command.Parameters.Add(SQLiteParameter('id', 23))
```

- Name of Parameter class also depends on provider

Reading Results

- Command.ExecuteReader returns a DataReader
- Forward-only access to result set
- Includes meta-data about column names and types

```
reader = command.ExecuteReader()
while reader.Read():
    tweetCreatedAt = reader['created']
    < do something with tweet >
reader.Close()
```

- Close frees the result set

Practical 6 : Database access

Stutterdb.py contains functions saveTweet() and getTweets() to insert and select tweets from the database.

- Implement saveFriend() and getFriends(), which will be similar to the above.
- In saveFriend(), the parameters to set on the command are those enumerated in friendAttrs. Get the value of each attr using friend.get(key)
- In getFriends(), no parameters need setting. The list returned is obtained by calling reader.Read() until it returns False, appending the value of reader['screen_name'] each time.

IronPython and Threads

Basic Principles

- Basic support is the `Thread` class from `System.Threading`
- You can also use the Python threading API
- An unhandled exception in a .NET thread is fatal - not in a Python thread
- You can abort .NET threads
- No Global Interpreter Lock - scale across multiple cores

Thread Class & ThreadStart Delegate

```
>>> from System.Threading import Thread, ThreadStart
>>> def f():
...     Thread.Sleep(5000)
...     print 'Finished'
...
>>> t = Thread(ThreadStart(f))
>>> t.Start()
>>> Finished
```

If you are using IronPython 2 you will see a warning if you use `thread.Sleep`. You can use `Thread.CurrentThread.Join` to achieve the same effect.

Aborting Threads

```
>>> from System.Threading import Thread, ThreadStart
>>> def f():
...     Thread.Sleep(5000)
...     print 'Finished'
...
>>> t = Thread(ThreadStart(f))
>>> t.Start()
```

```
>>> t.Abort()
```

Calling `Thread.Abort` raises a `ThreadAbortException` inside the thread.

Cleaning up in a Finally Block

```
>>> from System.Threading import Thread, ThreadStart
>>> def f():
...     try:
...         print 'Started'
...     finally:
...         Thread.Sleep(5000)
...         print 'Finished'
...
>>> t = Thread(ThreadStart(f))
>>> t.Start()
Started
>>> t.Abort()
Finished
```

The `ThreadAbortException` won't be raised whilst we are inside a `finally` block. Calling `Thread.Abort` doesn't kill the thread until the finally block has completed. `Thread.CurrentThread.ResetAbort` can be used to cancel the thread abort.

Windows Forms and Threads

- Windows Forms (and WPF) must be used inside a Single Threaded Apartment (STA thread)
- Interacting with controls must be done from that thread
- We can communicate from another thread by invoking onto the message loop
- Every control has an `Invoke` method that takes a delegate

The CallTarget0 Delegate

```
import clr
try:
    # IronPython 1.x
    from IronPython.Runtime.Calls import CallTarget0
except ImportError:
    # IronPython 2.x
    clr.AddReference('IronPython')
    from IronPython.Compiler import CallTarget0
```

`CallTarget0` (and `CallTarget1` etc) changed location in between IronPython 1 and 2 (in fact it changed twice in IronPython 2 - the one shown here is its final resting place.)

Invoking onto the Control Thread

```
def SetText():
    textbox.Text = 'Something'

delegate = CallTarget0(SetText)
form.Invoke(SetText)

text = form.Invoke(CallTarget0(lambda: textbox.Text))
```

For callables that take arguments you can use `CallTarget1` and friends. I prefer to wrap the call with a lambda and use `CallTarget0`.

In IronPython 2 `CallTarget0` doesn't work with .NET methods that have a void return signature. To invoke a call to something like `form.Close` you will also need to wrap the call in a lambda.

We can use this to make the database / network calls for Stutter happen on a background thread.

Practical 7: Fetching Tweets in the Background

- Fetching new tweets can take a long time
- We want to do this on a background thread
- And then invoke a GUI refresh onto the control thread
- Complete `DoBackgroundWithInvoke` in threadhelper.py
- This is called from `Stutter.onRefresh`