IronPython aims to be a fully compatible implementation of the Python language. At the same time, the value of a separate implementation than CPython is to make available the .NET ecosystem of libraries. IronPython does this by exposing .NET concepts as Python entities. Existing Python syntax and new Python libraries (like *clr*) are used to make .NET features available to IronPython code.

Loading .NET assemblies

The smallest unit of distribution of functionality in .NET is an assembly which usually corresponds to a single file with the *.dll* file extension. The assembly is available either in the installation folder of the application, or in the GAC (Global assembly cache). Assemblies can be loaded by using the methods of the *clr* module. The following code will load the System.Xml.dll assembly which is part of the standard .NET implementation, and installed in the GAC:

```
>>> import clr
>>> clr.AddReference("System.Xml")
```

The full list of assemblies loaded by IronPython is available in *clr.References*:

```
>>> "System.Xml" in [assembly.GetName().Name for assembly in clr.References]
True
```

All .NET assemblies have a unique version number which allows using a specific version of a given assembly. The following code will load the version of System.Xml.dll that ships with .NET 2.0 and .NET 3.5:

```
>>> import clr
>>> clr.AddReference("System.Xml, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")
```

You can load assemblies that are neither in the GAC nor in the appbase (typically, the folder of ipy.exe or your host appplication executable) either by using *clr.AddReferenceToFileAndPath* or by setting *sys.path*. See *clr.AddReference-methods* for details.

Note

IronPython only knows about assemblies that have been loaded using one of *clr.AddReference-methods*. It is possible for other assemblies to already be loaded before IronPython is loaded, or for other assemblies to be loaded by other parts of the application by calling System.Reflection.Assembly.Load, but IronPython will not be aware of these.

Assemblies loaded by default

By default, core assemblies (*mscorlib.dll* and *System.dll* on .NET Framework / *System.Private.CoreLib.dll* on .NET Core) are automatically loaded by the DLR. This enables you to start using these assemblies (which IronPython itself is dependent on) without having to call *clr.AddReference-methods*.

Using .NET types

Once an assembly is loaded, the namespaces and types contained in the assembly can be accessed from IronPython code.

Importing .NET namespaces

.NET namespaces and sub-namespaces of loaded assemblies are exposed as Python modules:

```
>>> import System
>>> System #doctest: +ELLIPSIS
<module 'System' (CLS module, ... assemblies loaded)>
```

Contents

```
>>> System.Collections #doctest: +ELLIPSIS
<module 'Collections' (CLS module, ... assemblies loaded)>
```

The types in the namespaces are exposed as Python types, and are accessed as attributes of the namespace. The following code accesses the System.Environment class from mscorlib.dll:

```
>>> import System
>>> System.Environment
<type 'Environment'>
```

Just like with normal Python modules, you can also use all the other forms of *import* as well:

```
>>> from System import Environment
>>> Environment
<type 'Environment'>
```

```
>>> from System import *
>>> Environment
<type 'Environment'>
```

Warning

Using `from <namespace> import *` can cause Python builtins (elements of *__builtins__*) to be hidden by .NET types or sub-namespaces. Specifically, after doing `from System import *`, `Exception` will access the System.Exception .NET type, not Python's Exception type.

The root namespaces are stored as modules in *sys.modules*:

```
>>> import System
>>> import sys
>>> sys.modules["System"] #doctest: +ELLIPSIS
<module 'System' (CLS module, ... assemblies loaded)>
```

When new assemblies are loaded, they can add attributes to existing namespace module objects.

Import precedence relative to Python modules

import gives precedence to .py files. For example, if a file called *System.py* exists in the path, it will get imported instead of the *System* namespace:

```
>>> # create System.py in the current folder
>>> f = open("System.py", "w")
>>> f.write('print "Loading System.py"')
>>> f.close()
>>>
>>> # unload the System namespace if it has been loaded
>>> import sys
>>> if sys.modules.has_key("System"):
...     sys.modules.pop("System") #doctest: +ELLIPSIS
<module 'System' (CLS module, ... assemblies loaded)>
>>>
>>> import System
Loading System.py
>>> System #doctest: +ELLIPSIS
<module 'System' from '...System.py'>
```

Note

Do make sure to delete System.py:

```
>>> import os
>>> os.remove("System.py")
>>> sys.modules.pop("System") #doctest: +ELLIPSIS
<module 'System' from '...System.py'>
>>> import System
>>> System #doctest: +ELLIPSIS
<module 'System' (CLS module, ... assemblies loaded)>
```

Accessing generic types

.NET supports generic types which allow the same code to support multiple type parameters which retaining the advantages of types safety. Collection types (like lists, vectors, etc) are the canonical example where generic types are useful. .NET has a number of generic collection types in the System.Collections.Generic namespace.

IronPython exposes generic types as a special *type* object which supports indexing with *type* object(s) as the index (or indices):

```
>>> from System.Collections.Generic import List, Dictionary
>>> int_list = List[int]()
>>> str_float_dict = Dictionary[str, float]()
```

Note that there might exist a non-generic type as well as one or more generic types with the same name [1]. In this case, the name can be used without any indexing to access the non-generic type, and it can be indexed with different number of types to access the generic type with the corresponding number of type parameters. The code below accesses System.EventHandler and also System.EventHandler<TEventArgs>

```
>>> from System import EventHandler, EventArgs
>>> EventHandler # this is the combo type object
<types 'EventHandler', 'EventHandler[TEventArgs]'>
>>> # Access the non-generic type
>>> dir(EventHandler) #doctest: +ELLIPSIS
['BeginInvoke', 'Clone', 'DynamicInvoke', 'EndInvoke', ...
>>> # Access the generic type with 1 type paramter
>>> dir(EventHandler[EventArgs]) #doctest: +ELLIPSIS
['BeginInvoke', 'Call', 'Clone', 'Combine', ...
```

[1] This refers to the user-friendly name. Under the hoods, the .NET type name includes the number of type parameters:

```
>>> clr.GetClrType(EventHandler[EventArgs]).Name
'EventHandler`1'
```

Accessing nested types

Nested types are exposed as attributes of the outer class:

```
>>> from System.Environment import SpecialFolder
>>> SpecialFolder
<type 'SpecialFolder'>
```

Importing .NET members from a type

.NET types are exposed as Python classes. Like Python classes, you usually cannot import *all* the attributes of .NET types using `from <name> import *`:

```
>>> from System.Guid import *
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
ImportError: no module named Guid
```

You can import specific members, both static and instance:

```
>>> from System.Guid import NewGuid, ToByteArray
>>> g = NewGuid()
>>> ToByteArray(g) #doctest: +ELLIPSIS
Array[Byte](...
```

Note that if you import a static property, you will import the value when the *import* executes, not a named object to be evaluated on every use as you might mistakenly expect:

```
>>> from System.DateTime import Now
>>> Now #doctest: +ELLIPSIS
<System.DateTime object at ...>
>>> # Let's make it even more obvious that "Now" is evaluated only once
>>> a_second_ago = Now
>>> import time
>>> time.sleep(1)
>>> a_second_ago is Now
True
>>> a_second_ago is System.DateTime.Now
False
```

Importing all .NET members from a static type

Some .NET types only have static methods, and are comparable to namespaces. *C#* refers to them as static classes , and requires such classes to have only static methods. IronPython allows you to import all the static methods of such *static classes*. System.Environment is an example of a static class:

```
>>> from System.Environment import *
>>> Exit is System.Environment.Exit
True
```

Nested types are also imported:

```
>>> SpecialFolder is System.Environment.SpecialFolder
True
```

However, properties are not imported:

```
>>> OSVersion
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'OSVersion' is not defined
>>> System.Environment.OSVersion #doctest: +ELLIPSIS
<System.OperatingSystem object at ...>
```

Type-system unification (*type* and *System.Type*)

.NET represents types using System.Type. However, when you access a .NET type in Python code, you get a Python *type* object [2]:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
```

```
>>> isinstance(type(ba), type)
True
```

This allows a unified (Pythonic) view of both Python and .NET types. For example, *isinstance* works with .NET types as well:

```
>>> from System.Collections import BitArray
>>> isinstance(ba, BitArray)
True
```

If need to get the *System.Type* instance for the .NET type, you need to use *clr.GetClrType*. Conversely, you can use *clr.GetPythonType* to get a *type* object corresponding to a *System.Type* object.

The unification also extends to other type system entities like methods. .NET methods are exposed as instances of *method*:

```
>>> type(BitArray.Xor)
<type 'method_descriptor'>
>>> type(ba.Xor)
<type 'builtin_function_or_method'>
```

[2] Note that the Python type corresponding to a .NET type is a sub-type of *type*:

```
>>> isinstance(type(ba), type)
True
>>> type(ba) is type
False
```

This is an implementation detail.

Similarity with builtin types

.NET types behave like builtin types (like *list*), and are immutable. i.e. you cannot add or delete descriptors from .NET types:

```
>>> del list.append
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: cannot delete attribute 'append' of builtin type 'list'
>>>
>>> import System
>>> del System.DateTime.ToByteArray
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'DateTime'
```

Instantiating .NET types

.NET types are exposed as Python classes, and you can do many of the same operations on .NET types as with Python classes. In either cases, you create an instance by calling the type:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5) # Creates a bit array of size 5
```

IronPython also supports inline initializing of the attributes of the instance. Consider the following two lines:

```
>>> ba = BitArray(5)
>>> ba.Length = 10
```

The above two lines are equivalent to this single line:

```
>>> ba = BitArray(5, Length = 10)
```

You can also call the *__new__* method to create an instance:

```
>> ba = BitArray.__new__(BitArray, 5)
```

Invoking .NET methods

.NET methods are exposed as Python methods. Invoking .NET methods works just like invoking Python methods.

Invoking .NET instance methods

Invoking .NET instance methods works just like invoking methods on a Python object using the attribute notation:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
>>> ba.Set(0, True) # call the Set method
>>> ba[0]
True
```

IronPython also supports named arguments:

```
>>> ba.Set(index = 1, value = True)
>>> ba[1]
True
```

IronPython also supports dict arguments:

```
>>> args = [2, True] # list of arguments
>>> ba.Set(*args)
>>> ba[2]
True
```

IronPython also supports keyword arguments:

```
>>> args = { "index" : 3, "value" : True }
>>> ba.Set(**args)
>>> ba[3]
True
```

Argument conversions

When the argument type does not exactly match the parameter type expected by the .NET method, IronPython tries to convert the argument. IronPython uses conventional .NET conversion rules like conversion operators , as well as IronPython-specific rules. This snippet shows how arguments are converted when calling the Set(System.Int32, System.Boolean) method:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
>>> ba.Set(0, "hello") # converts the second argument to True.
>>> ba[0]
True
>>> ba.Set(1, None) # converts the second argument to False.
```

```
>>> ba[1]
False
```

See *appendix-type-conversion-rules* for the detailed conversion rules. Note that some Python types are implemented as .NET types and no conversion is required in such cases. See *builtin-type-mapping* for the mapping.

Some of the conversions supported are:

Python argument type	.NET method parameter type
int	System.Int8, System.Int16
float	System.Float
tuple with only elements of type T	System.Collections.Generic.IEnumerable<T>
function, method	System.Delegate and any of its sub-classes

Method overloads

.NET supports overloading methods by both number of arguments and type of arguments. When IronPython code calls an overloaded method, IronPython tries to select one of the overloads *at runtime* based on the number and type of arguments passed to the method, and also names of any keyword arguments. In most cases, the expected overload gets selected. Selecting an overload is easy when the argument types are an exact match with one of the overload signatures:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5) # calls __new__(System.Int32)
>>> ba = BitArray(5, True) # calls __new__(System.Int32, System.Boolean)
>>> ba = BitArray(ba) # calls __new__(System.Collections.BitArray)
```

The argument types do not have be an exact match with the method signature. IronPython will try to convert the arguments if an *unamibguous* conversion exists to one of the overload signatures. The following code calls __new__(System.Int32) even though there are two constructors which take one argument, and neither of them accept a *float* as an argument:

```
>>> ba = BitArray(5.0)
```

However, note that IronPython will raise a TypeError if there are conversions to more than one of the overloads:

```
>>> BitArray((1, 2, 3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Multiple targets could match: BitArray(Array[Byte]), BitArray(Array[bool]), BitArray(Array[int])
```

If you want to control the exact overload that gets called, you can use the *Overloads* method on *method* objects:

```
>>> int_bool_new = BitArray.__new__.Overloads[int, type(True)]
>>> ba = int_bool_new(BitArray, 5, True) # calls __new__(System.Int32, System.Boolean)
>>> ba = int_bool_new(BitArray, 5, "hello") # converts "hello" to a System.Boolan
>>> ba = int_bool_new(BitArray, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() takes exactly 2 arguments (1 given)
```

TODO - Example of indexing Overloads with an Array, byref, etc using Type.MakeByrefType

---

Using unbound class instance methods

It is sometimes desirable to invoke an instance method using the unbound class instance method and passing an explicit *self* object as the first argument. For example, .NET allows a class to declare an instance method with the same name as a method in a base type, but without overriding the base method. See System.Reflection.MethodAttributes.NewSlot for more information. In such cases, using the unbound class instance method syntax allows you chose precisely which slot you wish to call:

```
>>> import System
>>> System.ICloneable.Clone("hello") # same as : "hello".Clone()
```

```
'hello'
```

The unbound class instance method syntax results in a virtual call, and calls the most derived implementation of the virtual method slot:

```
>>> s = "hello"
>>> System.Object.GetHashCode(s) == System.String.GetHashCode(s)
True
>>> from System.Runtime.CompilerServices import RuntimeHelpers
>>> RuntimeHelpers.GetHashCode(s) == System.String.GetHashCode(s)
False
```

Calling explicitly-implemented interface methods

.NET allows a method with a different name to override a base method implementation or interface method slot. This is useful if a type implements two interfaces with methods with the same name. This is known as explicity implemented interface methods. For example, *Microsoft.Win32.RegistryKey* implements *System.IDisposable.Dispose* explicitly:

```
>>> from Microsoft.Win32 import RegistryKey
>>> clr.GetClrType(RegistryKey).GetMethod("Flush") #doctest: +ELLIPSIS
<System.Reflection.RuntimeMethodInfo object at ... [Void Flush()]>
>>> clr.GetClrType(RegistryKey).GetMethod("Dispose")
>>>
```

In such cases, IronPython tries to expose the method using its simple name - if there is no ambiguity:

```
>>> from Microsoft.Win32 import Registry
>>> rkey = Registry.CurrentUser.OpenSubKey("Software")
>>> rkey.Dispose()
```

However, it is possible that the type has another method with the same name. In that case, the explicitly implemented method is not accessible as an attribute. However, it can still be called by using the unbound class instance method syntax:

```
>>> rkey = Registry.CurrentUser.OpenSubKey("Software")
>>> System.IDisposable.Dispose(rkey)
```

Invoking static .NET methods

Invoking static .NET methods is similar to invoking Python static methods:

```
>>> System.GC.Collect()
```

Like Python static methods, the .NET static method can be accessed as an attribute of sub-types as well:

```
>>> System.Object.ReferenceEquals is System.GC.ReferenceEquals
True
```

TODO What happens if the sub-type has a static method with the same name but a different signature? Are both overloads available or not?

Invoking generic methods

Generic methods are exposed as attributes which can be indexed with *type* objects. The following code calls System.Activator.CreateInstance<T>

```
>>> from System import Activator, Guid
>>> guid = Activator.CreateInstance[Guid]()
```

Type parameter inference while invoking generic methods

In many cases, the type parameter can be inferred based on the arguments passed to the method call. Consider the following use of a generic method [3]:

```
>>> from System.Collections.Generic import IEnumerable, List
>>> list = List[int]([1, 2, 3])
>>> import clr
>>> clr.AddReference("System.Core")
>>> from System.Linq import Enumerable
>>> Enumerable.Any[int](list, lambda x : x < 2)
True
```

With generic type parameter inference, the last statement can also be written as:

```
>>> Enumerable.Any(list, lambda x : x < 2)
True
```

See *appendix* for the detailed rules.

> [3] System.Core.dll is part of .NET 3.0 and higher.

ref and *out* parameters

The Python language passes all arguments by-value. There is no syntax to indicate that an argument should be passed by-reference like there is in .NET languages like C# and VB.NET via the ref and out keywords. IronPython supports two ways of passing ref or out arguments to a method, an implicit way and an explicit way.

In the implicit way, an argument is passed normally to the method call, and its (potentially) updated value is returned from the method call along with the normal return value (if any). This composes well with the Python feature of multiple return values. *System.Collections.Generic.Dictionary* has a method bool TryGetValue(K key, out value). It can be called from IronPython with just one argument, and the call returns a *tuple* where the first element is a boolean and the second element is the value (or the default value of 0.0 if the first element is *False*):

```
>>> d = { "a":100.1, "b":200.2, "c":300.3 }
>>> from System.Collections.Generic import Dictionary
>>> d = Dictionary[str, float](d)
>>> d.TryGetValue("b")
(True, 200.2)
>>> d.TryGetValue("z")
(False, 0.0)
```

In the explicit way, you can pass an instance of *clr.Reference[T]* for the ref or out argument, and its *Value* field will get set by the call. The explicit way is useful if there are multiple overloads with ref parameters:

```
>>> import clr
>>> r = clr.Reference[float]()
>>> d.TryGetValue("b", r)
True
>>> r.Value
200.2
```

Extension methods

Extension methods are currently not natively supported by IronPython. Hence, they cannot be invoked like instance methods. Instead, they have to be invoked like static methods.

Accessing .NET indexers

.NET indexers are exposed as *__getitem__* and *__setitem__*. Thus, the Python indexing syntax can be used to index .NET collections (and any type with an indexer):

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
>>> ba[0]
False
>>> ba[0] = True
>>> ba[0]
True
```

The indexer can be called using the unbound class instance method syntax using *__getitem__* and *__setitem__*. This is useful if the indexer is virtual and is implemented as an explicitly-implemented interface method:

```
>>> BitArray.__getitem__(ba, 0)
True
```

Non-default .NET indexers

Note that a default indexer is just a property (typically called *Item*) with one argument. It is considered as an indexer if the declaraing type uses DefaultMemberAttribute to declare the property as the default member.

See *property-with-parameters* for information on non-default indexers.

Accessing .NET properties

.NET properties are exposed similar to Python attributes. Under the hood, .NET properties are implemented as a pair of methods to get and set the property, and IronPython calls the appropriate method depending on whether you are reading or writing to the properity:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
>>> ba.Length # calls "BitArray.get_Length()"
5
>>> ba.Length = 10 # calls "BitArray.set_Length()"
```

To call the get or set method using the unbound class instance method syntax, IronPython exposes methods called *GetValue* and *SetValue* on the property descriptor. The code above is equivalent to the following:

```
>>> ba = BitArray(5)
>>> BitArray.Length.GetValue(ba)
5
>>> BitArray.Length.SetValue(ba, 10)
```

Properties with parameters

COM and VB.NET support properties with paramters. They are also known as non-default indexers. C# does not support declaring or using properties with parameters.

IronPython does support properties with parameters. For example, the default indexer above can also be accessed using the non-default format as such:

```
>>> ba.Item[0]
False
```

Accessing .NET events

.NET events are exposed as objects with __iadd__ and __isub__ methods which allows using += and -= to subscribe and unsubscribe from the event. The following code shows how to subscribe a Python function to an event using +=, and unsubscribe using -=

```
>>> from System.IO import FileSystemWatcher
>>> watcher = FileSystemWatcher(".")
>>> def callback(sender, event_args):
...     print event_args.ChangeType, event_args.Name
>>> watcher.Created += callback
>>> watcher.EnableRaisingEvents = True
```

```
>>> import time
>>> f = open("test.txt", "w+"); time.sleep(1)
Created test.txt
>>> watcher.Created -= callback
>>>
>>> # cleanup
>>> import os
>>> f.close(); os.remove("test.txt")
```

You can also subscribe using a bound method:

```
>>> watcher = FileSystemWatcher(".")
>>> class MyClass(object):
...     def callback(self, sender, event_args):
...         print event_args.ChangeType, event_args.Name
>>> o = MyClass()
>>> watcher.Created += o.callback
>>> watcher.EnableRaisingEvents = True
>>> f = open("test.txt", "w+"); time.sleep(1)
Created test.txt
>>> watcher.Created -= o.callback
>>>
>>> # cleanup
>>> f.close(); os.remove("test.txt")
```

You can also explicitly create a delegate instance to subscribe to the event. Otherwise, IronPython automatically does it for you. [4]:

```
>>> watcher = FileSystemWatcher(".")
>>> def callback(sender, event_args):
...     print event_args.ChangeType, event_args.Name
>>> from System.IO import FileSystemEventHandler
>>> delegate = FileSystemEventHandler(callback)
>>> watcher.Created += delegate
>>> watcher.EnableRaisingEvents = True
>>> import time
>>> f = open("test.txt", "w+"); time.sleep(1)
Created test.txt
>>> watcher.Created -= delegate
>>>
>>> # cleanup
>>> f.close(); os.remove("test.txt")
```

[4] The only advantage to creating an explicit delegate is that it is uses less memory. You should consider it if you subscribe to lots of events, and notice excessive System.WeakReference objects.

Special .NET types

.NET arrays

IronPython supports indexing of *System.Array* with a *type* object to access one-dimensional strongly-typed arrays:

```
>>> System.Array[int]
<type 'Array[int]'>
```

IronPython also adds a *__new__* method that accepts a IList<T> to initialize the array. This allows using a Python *list* literal to initialize a .NET array:

```
>>> a = System.Array[int]([1, 2, 3])
```

Further, IronPython exposes *__getitem__* and *__setitem__* allowing the array objects to be indexed using the Python indexing syntax:

```
>>> a[2]
3
```

Note that the indexing syntax yields Python semantics. If you index with a negative value, it results in indexing from the end of the array, whereas .NET indexing (demonstrated by calling *GetValue* below) raises a *System.IndexOutOfRangeException* exception:

```
>>> a.GetValue(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: Index was outside the bounds of the array.
>>> a[-1]
3
```

Similarly, slicing is also supported:

```
>>> a[1:3]
Array[int]((2, 3))
```

Multi-dimensional arrays

TODO

.NET Exceptions

raise can raise both Python exceptions as well as .NET exceptions:

```
>>> raise ZeroDivisionError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
>>> import System
>>> raise System.DivideByZeroException()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: Attempted to divide by zero.
```

The *except* keyword can catch both Python exceptions as well as .NET exceptions:

```
>>> try:
...     import System
...     raise System.DivideByZeroException()
... except System.DivideByZeroException:
...     print "This line will get printed..."
...
This line will get printed...
>>>
```

The underlying .NET exception object

IronPython implements the Python exception mechanism on top of the .NET exception mechanism. This allows Python exception thrown from Python code to be caught by non-Python code, and vice versa. However, Python exception objects need to behave like Python user objects, not builtin types. For example, Python code can set arbitrary attributes on Python exception objects, but not on .NET exception objects:

```
>>> e = ZeroDivisionError()
>>> e.foo = 1 # this works
>>> e = System.DivideByZeroException()
>>> e.foo = 1
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'DivideByZeroException' object has no attribute 'foo'
```

To support these two different views, IronPython creates a pair of objects, a Python exception object and a .NET exception object, where the Python type and the .NET exception type have a unique one-to-one mapping as defined in the table below. Both objects know about each other. The .NET exception object is the one that actually gets thrown by the IronPython runtime when Python code executes a *raise* statement. When Python code uses the *except* keyword to catch the Python exception, the Python exception object is used. However, if the exception is caught by C# (for example) code that called the Python code, then the C# code naturally catches the .NET exception object.

The .NET exception object corresponding to a Python exception object can be accessed by using the *clsException* attribute (if the module has excecuted *import clr*):

```
>>> import clr
>>> try:
...     1/0
... except ZeroDivisionError as e:
...     pass
>>> type(e)
<type 'exceptions.ZeroDivisionError'>
>>> type(e.clsException)
<type 'DivideByZeroException'>
```

IronPython is also able to access the Python exception object corresponding to a .NET exception object [5], thought this is not exposed to the user [6].

[5] The Python exception object corresponding to a .NET exception object is accessible (to the IronPython runtime) via the *System.Exception.Data* property. Note that this is an implementation detail and subject to change:

```
>>> e.clsException.Data["PythonExceptionInfo"] #doctest: +ELLIPSIS
<IronPython.Runtime.Exceptions.PythonExceptions+ExceptionDataWrapper object at ...>
```

[6] ... except via the DLR Hosting API *ScriptEngine.GetService<ExceptionOperations>().GetExceptionMessage*

Python exception	.NET exception	
Exception	System.Exception	
SystemExit		IP.O.SystemExit
StopIteration	System.InvalidOperationException subtype	
StandardError	System.SystemException	
KeyboardInterrupt		IP.O.KeyboardInterruptException
ImportError		IP.O.PythonImportError
EnvironmentError		IP.O.PythonEnvironmentError
IOError	System.IO.IOException	
OSError	S.R.InteropServices.ExternalException	
WindowsError	System.ComponentModel.Win32Exception	
EOFError	System.IO.EndOfStreamException	
RuntimeError	IP.O.RuntimeException	
NotImplementedError	System.NotImplementedException	
NameError		IP.O.NameException
UnboundLocalError		IP.O.UnboundLocalException
AttributeError	System.MissingMemberException	
SyntaxError		IP.O.SyntaxErrorException (System.Data has something close)

Python exception	.NET exception	
IndentationError		IP.O.IndentationErrorException
TabError		IP.O.TabErrorException
TypeError		Microsoft.Scripting.ArgumentTypeException
AssertionError		IP.O.AssertionException
LookupError		IP.O.LookupException
IndexError	System.IndexOutOfRangeException	
KeyError	S.C.G.KeyNotFoundException	
ArithmeticError	System.ArithmeticException	
OverflowError	System.OverflowException	
ZeroDivisionError	System.DivideByZeroException	
FloatingPointError		IP.O.PythonFloatingPointError
ValueError	ArgumentException	
UnicodeError		IP.O.UnicodeException
UnicodeEncodeError	System.Text.EncoderFallbackException	
UnicodeDecodeError	System.Text.DecoderFallbackException	
UnicodeTranslateError		IP.O.UnicodeTranslateException
ReferenceError		IP.O.ReferenceException
SystemError		IP.O.PythonSystemError
MemoryError	System.OutOfMemoryException	
Warning	System.ComponentModel.WarningException	
UserWarning		IP.O.PythonUserWarning
DeprecationWarning		IP.O.PythonDeprecationWarning
PendingDeprecationWarning		IP.O.PythonPendingDeprecationWarning
SyntaxWarning		IP.O.PythonSyntaxWarning
OverflowWarning		IP.O.PythonOverflowWarning
RuntimeWarning		IP.O.PythonRuntimeWarning
FutureWarning		IP.O.PythonFutureWarning

Revisiting the *rescue* keyword

Given that *raise* results in the creation of both a Python exception object and a .NET exception object, and given that *rescue* can catch both Python exceptions and .NET exceptions, a question arises of which of the exception objects will be used by the *rescue* keyword. The answer is that it is the type used in the *rescue* clause. i.e. if the *rescue* clause uses the Python exception, then the Python exception object will be used. If the *rescue* clause uses the .NET exception, then the .NET exception object will be used.

The following example shows how `1/0` results in the creation of two objects, and how they are linked to each other. The exception is first caught as a .NET exception. The .NET exception is raised again, but is then caught as a Python exception:

```
>>> import System
>>> try:
...     try:
...         1/0
...     except System.DivideByZeroException as e1:
...         raise e1
... except ZeroDivisionError as e2:
...     pass
>>> type(e1)
<type 'DivideByZeroException'>
```

```
>>> type(e2)
<type 'exceptions.ZeroDivisionError'>
>>> e2.clsException is e1
True
```

User-defined exceptions

Python user-defined exceptions get mapped to *System.Exception*. If non-Python code catches a Python user-defined exception, it will be an instance of *System.Exception*, and will not be able to access the exception details:

```
>>> # since "Exception" might be System.Exception after "from System import *"
>>> if "Exception" in globals(): del Exception
>>> class MyException(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
>>> try:
...     raise MyException("some message")
... except System.Exception as e:
...     pass
>>> clr.GetClrType(type(e)).FullName
'System.Exception'
>>> e.Message
'Python Exception: MyException'
```

In this case, the non-Python code can use the *ScriptEngine.GetService<ExceptionOperations>().GetExceptionMessage* DLR Hosting API to get the exception message.

Enumerations

.NET enumeration types are sub-types of *System.Enum*. The enumeration values of an enumeration type are exposed as class attributes:

```
print System.AttributeTargets.All # access the value "All"
```

IronPython also supports using the bit-wise operators with the enumeration values:

```
>>> import System
>>> System.AttributeTargets.Class | System.AttributeTargets.Method
<enum System.AttributeTargets: Class, Method>
```

Value types

Python expects all mutable values to be represented as a reference type. .NET, on the other hand, introduces the concept of value types which are mostly copied instead of referenced. In particular .NET methods and properties returning a value type will always return a copy.

This can be confusing from a Python programmer's perspective since a subsequent update to a field of such a value type will occur on the local copy, not within whatever enclosing object originally provided the value type.

While most .NET value types are designed to be immutable, and the .NET design guidelines recommend value tyeps be immutable, this is not enforced by .NET, and so there do exist some .NET valuetype that are mutable. **TODO** - Example.

For example, take the following C# definitions:

```
struct Point {
    # Poorly defined struct - structs should be immutable
    public int x;
    public int y;
}

class Line {
    public Point start;
```

```
    public Point end;

    public Point Start { get { return start; } }
    public Point End { get { return end; } }
}
```

If *line* is an instance of the reference type Line, then a Python programmer may well expect "*line.Start.x = 1*" to set the x coordinate of the start of that line. In fact the property *Start* returned a copy of the *Point* value type and it's to that copy the update is made:

```
print line.Start.x    # prints '0'
line.Start.x = 1
print line.Start.x    # still prints '0'
```

This behavior is subtle and confusing enough that C# produces a compile-time error if similar code is written (an attempt to modify a field of a value type just returned from a property invocation).

Even worse, when an attempt is made to modify the value type directly via the start field exposed by Line (i.e. "`line.start.x = 1`"), IronPython will still update a local copy of the *Point* structure. That's because Python is structured so that "foo.bar" will always produce a useable value: in the case above "line.start" needs to return a full value type which in turn implies a copy.

C#, on the other hand, interprets the entirety of the "`line.start.x = 1`" statement and actually yields a value type reference for the "line.start" part which in turn can be used to set the "x" field in place.

This highlights a difference in semantics between the two languages. In Python "line.start.x = 1" and "foo = line.start; foo.x = 1" are semantically equivalent. In C# that is not necessarily so.

So in summary: a Python programmer making updates to a value type embedded in an object will silently have those updates lost where the same syntax would yield the expected semantics in C#. An update to a value type returned from a .NET property will also appear to succeed will updating a local copy and will not cause an error as it does in the C# world. These two issues could easily become the source of subtle, hard to trace bugs within a large application.

In an effort to prevent the unintended update of local value type copies and at the same time preserve as pythonic and consistent a view of the world as possible, direct updates to value type fields are not allowed by IronPython, and raise a ValueError:

```
>>> line.start.x = 1 #doctest: +SKIP
Traceback (most recent call last):
  File , line 0, in input##7
ValueError Attempt to update field x on value type Point; value type fields can not be directly modified
```

This renders value types "mostly" immutable; updates are still possible via instance methods on the value type itself.

Proxy types

IronPython cannot directly use System.MarshalByRefObject instances. IronPython uses reflection at runtime to determine how to access an object. However, *System.MarshalByRefObject* instances do not support reflection.

You *can* use *unbound-class-instance-method* syntax to call methods on such proxy objects.

Delegates

Python functions and bound instance methods can be converted to delegates:

```
>>> from System import EventHandler, EventArgs
>>> def foo(sender, event_args):
...     print event_args
>>> d = EventHandler(foo)
>>> d(None, EventArgs()) #doctest: +ELLIPSIS
<System.EventArgs object at ... [System.EventArgs]>
```

Variance

IronPython also allows the signature of the Python function or method to be different (though compatible) with the delegate signature. For example, the Python function can use keyword arguments:

```
>>> def foo(*args):
...     print args
>>> d = EventHandler(foo)
```

```
>>> d(None, EventArgs()) #doctest: +ELLIPSIS
(None, <System.EventArgs object at ... [System.EventArgs]>)
```

If the return type of the delegate is void, IronPython also allows the Python function to return any type of return value, and just ignores the return value:

```
>>> def foo(*args):
...     return 100 # this return value will get ignored
>>> d = EventHandler(foo)
>>> d(None, EventArgs())
```

If the return value is different, IronPython will try to convert it:

```
>>> def foo(str1, str2):
...     return 100.1 # this return value will get converted to an int
>>> d = System.Comparison[str](foo)
>>> d("hello", "there")
100
```

TODO - Delegates with out/ref parameters

Subclassing .NET types

Sub-classing of .NET types and interfaces is supported using *class*. .NET types and interfaces can be used as one of the sub-types in the *class* construct:

```
>>> class MyClass(System.Attribute, System.ICloneable, System.IComparable):
...     pass
```

.NET does not support multiple inheritance while Python does. IronPython allows using multiple Python classes as subtypes, and also multiple .NET interfaces, but there can only be one .NET class (other than *System.Object*) in the set of subtypes:

```
>>> class MyPythonClass1(object): pass
>>> class MyPythonClass2(object): pass
>>> class MyMixedClass(MyPythonClass1, MyPythonClass2, System.Attribute):
...     pass
```

Instances of the class *do* actually inherit from the specified .NET base type. This is important because this means that statically-typed .NET code can access the object using the .NET type. The following snippet uses Reflection to show that the object can be cast to the .NET sub-class:

```
>>> class MyClass(System.ICloneable):
...     pass
>>> o = MyClass()
>>> import clr
>>> clr.GetClrType(System.ICloneable).IsAssignableFrom(o.GetType())
True
```

Note that the Python class does not really inherit from the .NET sub-class. See *type-mapping*.

Overriding methods

Base type methods can be overriden by defining a Python method with the same name:

```
>>> class MyClass(System.ICloneable):
...     def Clone(self):
...         return MyClass()
>>> o = MyClass()
```

```
>>> o.Clone() #doctest: +ELLIPSIS
<MyClass object at ...>
```

IronPython does require you to provide implementations of interface methods in the class declaration. The method lookup is done dynamically when the method is accessed. Here we see that *AttributeError* is raised if the method is not defined:

```
>>> class MyClass(System.ICloneable): pass
>>> o = MyClass()
>>> o.Clone()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute 'Clone'
```

Methods with multiple overloads

Python does not support method overloading. A class can have only one method with a given name. As a result, you cannot override specific method overloads of a .NET sub-type. Instead, you need to use define the function accepting an arbitrary argument list (see *_tut-arbitraryargs*), and then determine the method overload that was invoked by inspecting the types of the arguments:

```
>>> import clr
>>> import System
>>> StringComparer = System.Collections.Generic.IEqualityComparer[str]
>>>
>>> class MyComparer(StringComparer):
...     def GetHashCode(self, *args):
...         if len(args) == 0:
...             # Object.GetHashCode() called
...             return 100
...
...         if len(args) == 1 and type(args[0]) == str:
...             # StringComparer.GetHashCode() called
...             return 200
...
...         assert("Should never get here")
...
>>> comparer = MyComparer()
>>> getHashCode1 = clr.GetClrType(System.Object).GetMethod("GetHashCode")
>>> args = System.Array[object](["another string"])
>>> getHashCode2 = clr.GetClrType(StringComparer).GetMethod("GetHashCode")
>>>
>>> # Use Reflection to simulate a call to the different overloads
>>> # from another .NET language
>>> getHashCode1.Invoke(comparer, None)
100
>>> getHashCode2.Invoke(comparer, args)
200
```

Note
Determining the exact overload that was invoked may not be possible, for example, if *None* is passed in as an argument.

Methods with *ref* or *out* parameters

Python does not have syntax for specifying whether a method paramter is passed by-reference since arguments are always passed by-value. When overriding a .NET method with ref or out parameters, the ref or out paramter is received as a *clr.Reference[T]* instance. The incoming argument value is accessed by reading the *Value* property, and the resulting value is specified by setting the *Value* property:

```
>>> import clr
>>> import System
```

```
>>> StrFloatDictionary = System.Collections.Generic.IDictionary[str, float]
>>>
>>> class MyDictionary(StrFloatDictionary):
...     def TryGetValue(self, key, value):
...         if key == "yes":
...             value.Value = 100.1 # set the *out* parameter
...             return True
...         else:
...             value.Value = 0.0  # set the *out* parameter
...             return False
...     # Other methods of IDictionary not overriden for brevity
...
>>> d = MyDictionary()
>>> # Use Reflection to simulate a call from another .NET language
>>> tryGetValue = clr.GetClrType(StrFloatDictionary).GetMethod("TryGetValue")
>>> args = System.Array[object](["yes", 0.0])
>>> tryGetValue.Invoke(d, args)
True
>>> args[1]
100.1
```

Generic methods

When you override a generic method, the type parameters get passed in as arguments. Consider the following generic method declaration:

```
// csc /t:library /out:convert.dll convert.cs
public interface IMyConvertible {
    T1 Convert<T1, T2>(T2 arg);
}
```

The following code overrides the generic method *Convert*:

```
>>> import clr
>>> clr.AddReference("convert.dll")
>>> import System
>>> import IMyConvertible
>>>
>>> class MyConvertible(IMyConvertible):
...     def Convert(self, t2, T1, T2):
...         return T1(t2)
>>>
>>> o = MyConvertible()
>>> # Use Reflection to simulate a call from another .NET language
>>> type_params = System.Array[System.Type]([str, float])
>>> convert = clr.GetClrType(IMyConvertible).GetMethod("Convert")
>>> convert_of_str_float = convert.MakeGenericMethod(type_params)
>>> args = System.Array[object]([100.1])
>>> convert_of_str_float.Invoke(o, args)
'100.1'
```

Note

Generic method receive information about the method signature being invoked, whereas normal method overloads do not. The reason is that .NET does not allow normal method overloads to differ by the return type, and it is usually possible to determine the argument types based on the argument values. However, with generic methods, one of the type parameters may only be used as the return type. In that case, there is no way to determine the type paramter.

Calling from Python

When you call a method from Python, and the method overrides a .NET method from a base type, the call is performed as a regular Python call. The arguments do not undergo conversion, and neither are they modified in any way like being wrapped with *clr.Reference*. Thus, the call may need to be written differently than if the method was overriden by another language. For example, trying to call TryGetValue on the MyDictionary type from the *overriding-ref-args* section as shown below results in a TypeError, whereas a similar call works with *System.Collections.Generic.Dictionary[str, float]*:

```
>>> result, value = d.TryGetValue("yes")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: TryGetValue() takes exactly 3 arguments (2 given)
```

Overriding properties

.NET properties are backed by a pair of .NET methods for reading and writing the property. The C# compiler automatically names them as *get_<PropertyName>* and *set_<PropertyName>*. However, .NET itself does not require any specific naming pattern for these methods, and the names are stored in the the metadata associated with the property definition. The names can be accessed using the *GetGetMethod* and *GetSetMethods* of the *System.Reflection.PropertyInfo* class:

```
>>> import clr
>>> import System
>>> StringCollection = System.Collections.Generic.ICollection[str]
>>> prop_info = clr.GetClrType(StringCollection).GetProperty("Count")
>>> prop_info.GetGetMethod().Name
'get_Count'
>>> prop_info.GetSetMethod() # None because this is a read-only property
>>>
```

Overriding a virtual property requires defining a Python method with the same names as the underlying getter or setter .NET method:

```
>>>
>>> class MyCollection(StringCollection):
...     def get_Count(self):
...         return 100
...     # Other methods of ICollection not overriden for brevity
>>>
>>> c = MyCollection()
>>> # Use Reflection to simulate a call from another .NET language
>>> prop_info.GetGetMethod().Invoke(c, None)
100
```

Overiding events

Events have underlying methods which can be obtained using EventInfo.GetAddMethod and EventInfo.GetRemoveMethod

```
>>> from System.ComponentModel import IComponent
>>> import clr
>>> event_info = clr.GetClrType(IComponent).GetEvent("Disposed")
>>> event_info.GetAddMethod().Name
'add_Disposed'
>>> event_info.GetRemoveMethod().Name
'remove_Disposed'
```

To override events, you need to define methods with the name of the underlying methods:

```
>>> class MyComponent(IComponent):
...     def __init__(self):
...         self.dispose_handlers = []
...     def Dispose(self):
...         for handler in self.dispose_handlers:
...             handler(self, EventArgs())
...
...     def add_Disposed(self, value):
```

```
...         self.dispose_handlers.append(value)
...     def remove_Disposed(self, value):
...         self.dispose_handlers.remove(value)
...     # Other methods of IComponent not implemented for brevity
>>>
>>> c = MyComponent()
>>> def callback(sender, event_args):
...     print event_args
>>> args = System.Array[object]((System.EventHandler(callback),))
>>> # Use Reflection to simulate a call from another .NET language
>>> event_info.GetAddMethod().Invoke(c, args)
>>>
>>> c.Dispose() #doctest: +ELLIPSIS
<System.EventArgs object at ... [System.EventArgs]>
```

Calling base constructor

.NET constructors can be overloaded. To call a specific base type constructor overload, you need to define a *__new__* method (not *__init__*) and call *__new__* on the .NET base type. The following example shows how a sub-type of *System.Exception* choses the base constructor overload to call based on the arguments it receives:

```
>>> import System
>>> class MyException(System.Exception):
...     def __new__(cls, *args):
...         # This could be implemented as:
...         #     return System.Exception.__new__(cls, *args)
...         # but is more verbose just to make a point
...         if len(args) == 0:
...             e = System.Exception.__new__(cls)
...         elif len(args) == 1:
...             message = args[0]
...             e = System.Exception.__new__(cls, message)
...         elif len(args) == 2:
...             message, inner_exception = args
...             if hasattr(inner_exception, "clsException"):
...                 inner_exception = inner_exception.clsException
...             e = System.Exception.__new__(cls, message, inner_exception)
...         return e
>>> e = MyException("some message", IOError())
```

Accessing protected members of base types

Normally, IronPython does not allow access to protected members (unless you are using *private-binding*). For example, accessing MemberwiseClone causes a TypeError since it is a protected method:

```
>>> import clr
>>> import System
>>> o = System.Object()
>>> o.MemberwiseClone()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot access protected member MemberwiseClone without a python subclass of object
```

IronPython *does* allow Python sub-types to access protected members of .NET base types. However, Python does not enforce any accessibility rules. Also, methods can be added and removed dynamically from a class. Hence, IronPython does not attempt to guard access to *protected* members of .NET sub-types. Instead, it always makes the protected members available just like public members:

```
>>> class MyClass(System.Object):
...     pass
>>> o = MyClass()
```

```
>>> o.MemberwiseClone() #doctest: +ELLIPSIS
<MyClass object at ...>
```

Declaring .NET types

Relationship of classes in Python code and normal .NET types

A class definition in Python does not map directly to a unique .NET type. This is because the semantics of classes is different between Python and .NET. For example, in Python it is possible to change the base types just by assigning to the *__bases__* attribute on the type object. However, the same is not possible with .NET types. Hence, IronPython implements Python classes without mapping them directly to .NET types. IronPython *does* use *some* .NET type for the objects, but its members do not match the Python attributes at all. Instead, the Python class is stored in a .NET field called *.class*, and Python instance attributes are stored in a dictionary that is stored in a .NET field called *.dict* [7]

```
>>> import clr
>>> class MyClass(object):
...     pass
>>> o = MyClass()
>>> o.GetType().FullName #doctest: +ELLIPSIS
'IronPython.NewTypes.System.Object_...'
>>> [field.Name for field in o.GetType().GetFields()]
['.class', '.dict', '.slots_and_weakref']
>>> o.GetType().GetField(".class").GetValue(o) == MyClass
True
>>> class MyClass2(MyClass):
...     pass
>>> o2 = MyClass2()
>>> o.GetType() == o2.GetType()
True
```

Also see *Type-system unification (type and System.Type)*

[7] These field names are implementation details, and could change.

__clrtype__

It is sometimes required to have control over the .NET type generated for the Python class. This is because some .NET APIs expect the user to define a .NET type with certain attributes and members. For example, to define a pinvoke method, the user is required to define a .NET type with a .NET method marked with DllImportAttribute , and where the signature of the .NET method exactly describes the target platform method.

Starting with IronPython 2.6, IronPython supports a low-level hook which allows customization of the .NET type corresponding to a Python class. If the metaclass of a Python class has an attribute called *__clrtype__*, the attribute is called to generate a .NET type. This allows the user to control the the details of the generated .NET type. However, this is a low-level hook, and the user is expected to build on top of it.

The ClrType sample available in the IronPython website shows how to build on top of the __clrtype__ hook.

Accessing Python code from other .NET code

Statically-typed languages like C# and VB.Net can be compiled into an assembly that can then be used by other .NET code. However, IronPython code is executed dynamically using *ipy.exe*. If you want to run Python code from other .NET code, there are a number of ways of doing it.

Using the DLR Hosting APIs

The DLR Hosting APIs allow a .NET application to embed DLR languages like IronPython and IronRuby, load and execute Python and Ruby code, and access objects created by the Python or Ruby code.

Compiling Python code into an assembly

The pyc sample can be used to compile IronPython code into an assembly. The sample builds on top of *clr-CompileModules*. The assembly can then be loaded and executed using *Python-ImportModule*. However, note that the MSIL in the assembly is not CLS-compliant and cannot be directly accessed from other .NET languages.

dynamic

Starting with .NET 4.0, C# and VB.Net support access to IronPython objects using the *dynamic* keyword. This enables cleaner access to IronPython objects. Note that you need to use the *hosting-apis* to load IronPython code and get the root object out of it.

Integration of Python and .NET features

- Type system integration.
  - See "Type-system unification (type and System.Type)"
  - Also see *extensions-to-python-types* and *extensions-to-dotnet-types*
- List comprehension works with any .NET type that implements IList
- *with* works with with any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T>
- pickle and ISerializable
- __doc__ on .NET types and members:
  - __doc__ uses XML comments if available. XML comment files are installed if **TODO**. As a result, *help* can be used:

```
>>> help(System.Collections.BitArray.Set) #doctest: +NORMALIZE_WHITESPACE
Help on method_descriptor:
Set(...)
    Set(self, int index, bool value)
                Sets the bit at a specific
    position in the System.Collections.BitArray to
    the specified value.
<BLANKLINE>
    index:
                The zero-based index of the
    bit to set.
<BLANKLINE>
    value:
                The Boolean value to assign
    to the bit.
```

  - If XML comment files are not available, IronPython generates documentation by reflecting on the type or member:

```
>>> help(System.Collections.Generic.List.Enumerator.Current) #doctest: +NORMALIZE_WHITESPACE
Help on getset descriptor System.Collections.Generic in mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089.Enumerator.Current:
<BLANKLINE>
Current
    Get: T Current(self)
```

Extensions to Python types

import clr exposes extra functionality on some Python types to make .NET features accessible:

- *method* objects of any builtin or .NET types:
  - instance method
    - Overloads(t1 [, t2...])
- *type* objects
  - instance method
    - __getitem__(t1 [, t2...]) - creates a generic instantiation

Extensions to .NET types

IronPython also adds extensions to .NET types to make them more Pythonic. The following instance methods are exposed on .NET objects (and .NET classes where explicitly mentioned):

- Types with op_Implicit
  - **TODO**
- Types with op_Explicit
  - **TODO**
- Types inheriting from a .NET class or interface

.NET base-type	Synthesized Python method(s)
System.Object	all methods of *object* eg. __class__, __str__, __hash__, __setattr__
System.IDisposable	__enter__, __exit__
System.Collections.IEnumerator	next
System.Collections.ICollection System.Collections.Generic.ICollection<T>	__len__
System.Collections.IEnumerable System.Collections.Generic.IEnumerable<T> System.Collections.IEnumerator System.Collections.Generic.IEnumerator<T>	__iter__
System.IFormattable	__format__
System.Collections.IDictionary System.Collections.Generic.IDictionary<TKey, TValue> System.Collections.Generic.ICollection<T> System.Collections.Generic.IList<T> System.Collections.IEnumerable System.Collections.Generic.IEnumerable<T> System.Collections.IEnumerator System.Collections.Generic.IEnumerator<T>	__contains__
System.Array	◦ Class methods: ▪ Indexing of the type object with a type object to access a specific array type ▪ __new__(l) where l is IList<T> (or supports __getitem__?) ◦ __getitem__, __setitem__, __slice__
System.Delegate	◦ Class method : __new__(type, function_or_bound_method) ◦ __call__
System.Enum	__or__ **TODO** ?

- Types with a .NET operator method name

.NET operator method	Synthesized Python method
op_Addition, Add	__add__
Compare	__cmp__
get_<Name> [8]	__getitem__
set_<Name> [9]	__setitem__

[8] where the type also has a property <Name>, and a DefaultMemberAttribute for <Name>

[9] where the type also has a property <Name>, and a DefaultMemberAttribute for <Name>

Equality and hashing

TODO - This is currently just copied from IronRuby, and is known to be incorrect

Object equality and hashing are fundamental properties of objects. The Python API for comparing and hashing objects is __eq__ (and __ne__) and __hash__ respectively. The CLR APIs are System.Object.Equals and System.Object.GetHashCode respectively. IronPython does an automatic mapping between the two concepts so that Python objects can be compared and hashed from non-Python .NET code, and __eq__ and __hash__ are available in Python code for non-Python objects as well.

When Python code calls __eq__ and __hash__

- If the object is a Python object, the default implementations of __eq__ and __hash__ get called. The default implementations call System.Object.ReferenceEquals and System.Runtime.CompileServices.RuntimeHelpers.GetHashCode respectively.
- If the object is a CLR object, System.Object.Equals and System.Object.GetHashCode respectively get called on the .NET object.
- If the object is a Python subclass object inheriting from a CLR class, the CLR's class's implementation of System.Object.Equals and System.Object.GetHashCode will get called if the Python subclass does not define __eq__ and __hash__. If the Python subclass defines __eq__ and __hash__, those will be called instead.

When static MSIL code calls System.Object.Equals and System.Object.GetHashCode

- If the object is a Python objects, the Python object will direct the call to __eq__ and __hash__. If the Python object has implementations for these methods, they will be called. Otherwise, the default implementation mentioned above gets called.
- If the object is a Python subclass object inheriting from a CLR class, the CLR's class's implementation of System.Object.Equals and System.Object.GetHashCode will get called if the Python subclass does not define __eq__ and __hash__. If the Python subclass defines __eq__ and __hash__, those will be called instead.

Hashing of mutable objects

The CLR expects that System.Object.GetHashCode always returns the same value for a given object. If this invariant is not maintained, using the object as a key in a System.Collections.Generic.Dictionary<K,V> will misbehave. Python allows __hash__ to return different results, and relies on the user to deal with the scenario of using the object as a key in a Hash. The mapping above between the Python and CLR concepts of equality and hashing means that CLR code that deals with Python objects has to be aware of the issue. If static MSIL code uses a Python object as a the key in a Dictionary<K,V>, unexpected behavior might happen.

To reduce the chances of this happenning when using common Python types, IronPython does not map __hash__ to GetHashCode for Array and Hash. For other Python classes, the user can provide separate implementations for __eq__ and Equals, and __hash__ and GetHashCode if the Python class is mutable but also needs to be usable as a key in a Dictionary<K,V>.

System.Object.ToString, __repr__ and __str__

ToString on Python objects

Calling ToString on Python objects calls the default System.Object.ToString implementation, even if the Python type defines *__str__*:

```
>>> class MyClass(object):
...     def __str__(self):
...         return "__str__ result"
>>> o = MyClass()
>>> # Use Reflection to simulate a call from another .NET language
>>> o.GetType().GetMethod("ToString").Invoke(o, None) #doctest: +ELLIPSIS
'IronPython.NewTypes.System.Object_...'
```

__repr__ / __str__ on .NET objects

All Python user types have *__repr__* and *__str__*:

```
>>> class MyClass(object):
...     pass
>>> o = MyClass()
>>> o.__repr__() #doctest: +ELLIPSIS
'<MyClass object at ...>'
>>> o.__str__() #doctest: +ELLIPSIS
'IronPython.NewTypes.System.Object_...'
>>> str(o) #doctest: +ELLIPSIS
'<MyClass object at ...>'
```

For .NET types which do not override ToString, IronPython provides *__repr__* and *__str__* methods which behave similar to those of Python user types [10]:

```
>>> from System.Collections import BitArray
>>> ba = BitArray(5)
>>> ba.ToString() # BitArray inherts System.Object.ToString()
'System.Collections.BitArray'
```

```
>>> ba.__repr__() #doctest: +ELLIPSIS
'<System.Collections.BitArray object at ... [System.Collections.BitArray]>'
>>> ba.__str__() #doctest: +ELLIPSIS
'<System.Collections.BitArray object at ... [System.Collections.BitArray]>'
```

For .NET types which *do* override ToString, IronPython includes the result of ToString in *__repr__*, and maps ToString directly to *__str__*:

```
>>> e = System.Exception()
>>> e.ToString()
"System.Exception: Exception of type 'System.Exception' was thrown."
>>> e.__repr__() #doctest: +ELLIPSIS
"<System.Exception object at ... [System.Exception: Exception of type 'System.Exception' was thrown.]>"
>>> e.__str__() #doctest:
"System.Exception: Exception of type 'System.Exception' was thrown."
```

For Python types that override ToString, *__str__* is mapped to the ToString override:

```
>>> class MyClass(object):
...     def ToString(self):
...         return "ToString implemented in Python"
>>> o = MyClass()
>>> o.__repr__() #doctest: +ELLIPSIS
'<MyClass object at ...>'
>>> o.__str__()
'ToString implemented in Python'
>>> str(o) #doctest: +ELLIPSIS
'<MyClass object at ...>'
```

[10] There is some inconsistency in handling of __str__ that is tracked by http://ironpython.codeplex.com/WorkItem/View.aspx?WorkItemId=24973

OleAutomation and COM interop

IronPython supports accessing OleAutomation objects (COM objects which support dispinterfaces).

IronPython does not support the *win32ole* library, but Python code using *win32ole* can run on IronPython with just a few modifications.

Creating a COM object

Different languages have different ways to create a COM object. VBScript and VBA have a method called CreateObject to create an OleAut object. JScript has a method called **TODO**. There are multiple ways of doing the same in IronPython.

1. The first approach is to use System.Type.GetTypeFromProgID and System.Activator.CreateInstance . This method works with any registered COM object:

```
>>> import System
>>> t = System.Type.GetTypeFromProgID("Excel.Application")
>>> excel = System.Activator.CreateInstance(t)
>>> wb = excel.Workbooks.Add()
>>> excel.Quit()
```

2. The second approach is to use *clr.AddReferenceToTypeLibrary* to load the type library (if it is available) of the COM object. The advantage is that you can use the type library to access other named values like constants:

```
>>> import System
>>> excelTypeLibGuid = System.Guid("00020813-0000-0000-C000-000000000046")
>>> import clr
>>> clr.AddReferenceToTypeLibrary(excelTypeLibGuid)
>>> from Excel import Application
>>> excel = Application()
```

```
>>> wb = excel.Workbooks.Add()
>>> excel.Quit()
```

3. Finally, you can also use the interop assembly. This can be generated using the tlbimp.exe tool. The only advantage of this approach was that this was the approach recommeded for IronPython 1. If you have code using this approach that you developed for IronPython 1, it will continue to work:

```
>>> import clr
>>> clr.AddReference("Microsoft.Office.Interop.Excel")
>>> from Microsoft.Office.Interop.Excel import ApplicationClass
>>> excel = ApplicationClass()
>>> wb = excel.Workbooks.Add()
>>> excel.Quit()
```

Using COM objects

One you have access to a COM object, it can be used like any other objects. Properties, methods, default indexers and events all work as expected.

Properties

There is one important detail worth pointing out. IronPython tries to use the type library of the OleAut object if it can be found, in order to do name resolution while accessing methods or properties. The reason for this is that the IDispatch interface does not make much of a distinction between properties and method calls. This is because of Visual Basic 6 semantics where "excel.Quit" and "excel.Quit()" have the exact same semantics. However, IronPython has a strong distinction between properties and methods, and methods are first class objects. For IronPython to know whether "excel.Quit" should invoke the method Quit, or just return a callable object, it needs to inspect the typelib. If a typelib is not available, IronPython assumes that it is a method. So if a OleAut object has a property called "prop" but it has no typelib, you would need to write "p = obj.prop()" in IronPython to read the property value.

Methods with *out* parameters

Calling a method with "out" (or in-out) parameters requires explicitly passing in an instance of "clr.Reference", if you want to get the updated value from the method call. Note that COM methods with out parameters are not considered Automation-friendly [11]. JScript does not support out parameters at all. If you do run into a COM component which has out parameters, having to use "clr.Reference" is a reasonable workaround:

```
>>> import clr
>>> from System import Type, Activator
>>> command_type = Type.GetTypeFromProgID("ADODB.Command")
>>> command = Activator.CreateInstance(command_type)
>>> records_affected = clr.Reference[int]()
>>> command.Execute(records_affected, None, None) #doctest: +SKIP
>>> records_affected.Value
0
```

Another workaround is to leverage the inteorp assembly by using the unbound class instance method syntax of "outParamAsReturnValue = InteropAssemblyNamespace.IComInterface(comObject)".

> [11] Note that the Office APIs in particular do have "VARIANT*" parameters, but these methods do not update the value of the VARIANT. The only reason they were defined with "VARIANT*" parameters was for performance since passing a pointer to a VARIANT is faster than pushing all the 4 DWORDs of the VARIANT onto the stack. So you can just treat such parameters as "in" parameters.

Accessing the type library

The type library has names of constants. You can use *clr.AddReferenceToTypeLibrary* to load the type library.

Non-automation COM objects

IronPython does not fully support COM objects which do not support dispinterfaces since they appear likey proxy objects [12]. You can use the unbound class method syntax to access them.

> [12] This was supported in IronPython 1, but the support was dropped in version 2.

Miscellaneous

Security model

When running Python code using ipy.exe, IronPython behaves like Python and does not do any sand-boxing. All scripts execute with the permissions of the user. As a result, running Python code downloaded from the Internet for example could be potentially be dangerous.

However, ipy.exe is just one manifiestation of IronPython. IronPython can also be used in other scenarios like embedded in an application. All the IronPython assemblies are security-transparent. As a result, IronPython code can be run in a sand-box and the host can control the security priviledges to be granted to the Python code. This is one of the benefits of IronPython building on top of .NET.

Execution model and call frames

IronPython code can be executed by any of the following techniques:

1. Interpretation
2. Compiling on the fly using DynamicMethod
3. Compiling on the fly using DynamicMethod
4. Ahead-of-time compilation to an assembly on disk using ipyc
5. A combination of the above - ie. a method might initially be interpreted, and can later be compiled once it has been called a number of times.

As a result, call frames of IronPython code are not like frames of statically typed langauges like C# and VB.Net. .NET code using APIs like those listed below need to think about how it will deal with IronPython code:

- StackTrace.__new__
- GetExecutingAssembly
- Exception.ToString

Accessing non-public members

It is sometimes useful to access private members of an object. For example, while writing unit tests for .NET code in IronPython or when using the interactive command line to observe the innner workings of some object. ipy.exe supports this via the -X:PrivateBinding` command-line option. It can also be enabled in hosting scenarios via the **TODO** property ; this requires IronPython to be executing with FullTrust.

Mapping between Python builtin types and .NET types

IronPython is an implementation of the Python language on top of .NET. As such, IronPython uses various .NET types to implement Python types. Usually, you do not have to think about this. However, you may sometimes have to know about it.

Python type	.NET type
object	System.Object
int	System.Int32
long	System.Numeric.BigInteger [13]
float	System.Double
str, unicode	System.String
bool	System.Boolean

[13] This is true only in CLR 4. In previous versions of the CLR, *long* is implemented by IronPython itself.

---

import clr and builtin types

Since some Python builtin types are implemented as .NET types, the question arises whether the types work like Python types or like .NET types. The answer is that by default, the types work like Python types. However, if a module executes *import clr*, the types work like both Python types and like .NET types. For example, by default, object' does not have the *System.Object* method called *GetHashCode*:

```
>>> hasattr(object, "__hash__")
True
>>> # Note that this assumes that "import clr" has not yet been executed
>>> hasattr(object, "GetHashCode") #doctest: +SKIP
False
```

However, once you do *import clr*, *object* has both *__hash__* as well as *GetHashCode*:

```
>>> import clr
>>> hasattr(object, "__hash__")
True
>>> hasattr(object, "GetHashCode")
True
```

LINQ

Language-integrated Query (LINQ) is a set of features that was added in .NET 3.5. Since it is a scenario rather than a specific feature, we will first compare which of the scenarios work with IronPython:

- LINQ-to-objects

  Python's list comprehension provides similar functionality, and is more Pythonic. Hence, it is recommended to use list comprehension itself.

- DLinq - This is currently not supported.

Feature by feature comparison

LINQ consists of a number of language and .NET features, and IronPython has differing levels of support for the different features:

- C# and VB.NET lambda function - Python supports lambda functions already.
- Anonymous types - Python has tuples which can be used like anonymous types.
- Extension methods - See
- Generic method type parameter inference - See
- Expression trees - This is not supported. This is the main reason DLinq does not work.

Appendix - Type conversion rules

Note that some Python types are implemented as .NET types and no conversion is required in such cases. See *builtin-type-mapping* for the mapping.

Python argument type	.NET method parameter type
int	System.Byte, System.SByte, System.UInt16, System.Int16
User object with __int__ method	*Same as int*
str or unicode of size 1	System.Char
User object with __str__ method	*Same as str*
float	System.Float
tuple with T-typed elements	System.Collections.Generic.IEnumerable<T> or System.Collections.Generic.IList<T>
function, method	System.Delegate and any of its sub-classes
dict with K-typed keys and V-typed values	System.Collections.Generic.IDictionary<K,V>
type	System.Type

Appendix - Detailed method overload resolution rules

TODO: This is old information

Roughly equivalent to VB 11.8.1 with additional level of preferred narrowing conversions

- Start with the set of all accessible members
- Keep only those members for which the argument types can be assigned to the parameter types by a widening conversion
  - If there is one or more member in the set find the best member
    - If there is one best member then call it
    - If there are multiple best members then throw ambiguous

- Add in those members for which the argument types can be assigned to the parameter types by either a preferred narrowing or a widening conversion
    - If there is one applicable member then call it
    - If there is more than one applicable member then throw ambiguous
- Add in those members for which the argument types can be assigned to the parameter types by any narrowing or a widening conversion
    - If there is one applicable member then call it
    - If there is more than one applicable member then throw ambiguous
- Otherwise throw no match

Applicable Members By Number of Arguments – Phase 1

- The number of arguments is identical to the number of parameters
- The number of arguments is less than the number of parameters, but all parameters without an argument are optional – have a non-DbNull default value.
- The method includes a parameter array and the params-expanded form of the method is applicable to the arguments
    - The params-expanded form is constructed by replacing the parameter array in the declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments matches the number of parameters in the expanded form
- The method includes byref parameters and the byref-reduced form of the method is applicable to the arguments
    - The byref-reduced form is constructed by removing all out parameters from the list and replacing all ref parameters with their target type. The return information for such a match will be provided in a tuple of return values.

Applicable Members By Type Of Arguments – Phase 2

- If a conversion of the given type exists from the argument object to the type of the parameter for every argument then the method is applicable
    - For ref or out parameters, the argument must be an instance of the appropriate Reference class – unless the byref-reduced form of the method is being used

Better Member (same as C# 7.4.2.2)

Parameter Types : Given an argument list A with a set of types {A1, A1, ..., An} and type applicable parameter lists P and Q with types {P1, P2, ..., Pn} and {Q1, Q2, ..., Qn} P is a better member than Q if

- For each argument, the conversion from Ax to Px is not worse than the conversion from Ax to Qx, and
- For at least one argument, the conversion from Ax to Px is better than the conversion from Ax to Qx

Parameter Modifications : The method that uses the minimal conversions from the original method is considered the better match. The better member is the one that matches the earliest rule in the list of conversions for applicable methods. If both members use the same rules, then the method that converts the fewest of its parameters is considered best. For example, if multiple params methods have identical expanded forms, then the method with the most parameters prior to params-expanded form will be selected

Static vs. instance methods : When comparing a static method and an instance method that are both applicable, then the method that matches the calling convention is considered better. If the method is called unbound on the type object then the static method is preferred; however, if the method is called bound to an instance than the instance method will be preferred.

Explicitly implemented interface methods: Methods implemented as public methods on a class are considered better than methods that are private on the declaring class which explicitly implement an interface method.

Generic methods: Non-generic methods are considered better than generic methods.

Better Conversion (same as C# 7.4.2.3)

- If T1 == T2 then neither conversion is better
- If S is T1 then C1 is the better conversion (and vice-versa)
- If a conversion from T1 to T2 exists, and no conversion from T2 to T1 exists, then C1 is the better conversion (and vice versa)
- Conversion to a signed numeric type is preferred over conversion to a non-signed type of equal or greater size (this means that sbyte is preferred over byte)

Special conversion rule for ExtensibleFoo: An ExtensibleFoo has a conversion to a type whenever there is an appropriate conversion from Foo to that type.

Implicit Conversions

- Implicit numeric conversions (C# 6.1.2)
- Implicit reference conversions (C# 6.1.4) == Type.IsAssignableFrom
- null -> Nullable<T>
- COM object to any interface type
- User-defined implicit conversions (C# 6.1.7)
- Conversion from DynamicType -> Type

Narrowing Conversions (see VB 8.9 but much more restrictive for Python) are conversions that cannot be proved to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit narrowing notation. The following conversions are classified as narrowing conversions:

Preferred Narrowing Conversions

- BigInteger -> Int64 – because this is how Python represents numbers larger than 32 bits
- IList<object> -> IList<T>
- IEnumerator<object> -> IEnumerator<T>
- IDictionary<object,object> -> IDictionary<K,V>

<Need to edit from here on down>

Narrowing Conversions

- Bool -> int
- Narrowing conversions of numeric types when overflow doesn't occur
- String(length == 1) -> char and Char -> string(length == 1)
- Generic Python protocols to CLS types
  - Callable (or anything?) -> Delegate
  - Object (iterable?) -> IEnumerator?
  - __int__ to int, __float__, __complex__
- Troubling conversions planning to keep
  - Object -> bool (__nonzero__)
  - Double -> int – this is standard Python behavior, albeit deprecated behavior
  - Tuple -> Array<T>

All of the below will require explicit conversions

- Enum to numeric type – require explicit conversions instead
- From numeric types to char (excluded by C#)
- Dict -> Hashtable
- List -> Array<T>, List<T> and ArrayList
- Tuple -> List<T> and ArrayList

Rules for going the other direction when C# methods are overridden by Python or delegates are implemented on the Python side:

- This change alters our rules for how params and by ref parameters are handled for both overridden methods and delegates.

  1. by ref (ref or out) parameters are always passed to Python as an instance of clr.Reference. The Value property on these can be used to get and set the underlying value and on return from the method this will be propogated back to the caller.
  2. params parameters are ignored in these cases and the underlying array is passed to the Python function instead of splitting out all of the args.

- The principle behind this change is to present the most direct reflection of the CLS signature to the Python programmer when they are doing something where the signature could be ambiguous. For calling methods with by ref parameters we support both explicit Reference objects and the implicit skipped parameters. When overriding we want to support the most direct signature to remove ambiguity. Similarly for params methods we support both calling the method with an explicit array of args or with n-args. To remove the ambiguity when overriding we only support the explicit array.

- I'm quite happy with this principle in general. The one part that sucks for me is that these methods are now not callable from Python in the non-explict forms any more. For example, if I have a method void Foo(params object[] args) then I will override it with a Python method Foo(args) and not Foo(*args). This means that the CLS base type's method can be called as o.Foo(1,2,3) but the Python subclass will have to be called as o.Foo((1,2,3)). This is somewhat ugly, but I can't come up with any other relatively simple and clear option here and I think that because overriding overloaded methods can get quite complicated we should err on the side of simplicity.

Appendix - Rules for Type parameter inference while invoking generic methods

TODO