目次

―― *プログラミング言語* Boo *チュートリアル*

1. Boo とは何か？

Boo は Python に影響を受けた、CLI(.NET もしくは Mono)をターゲットにしたプログラミング言語です。シンプルで習得しやすい Python の文法でプログラミングができ、.NET 言語なので C#/VB.NET 等で作成したライブラリを呼び出すことも、Boo で作成したライブラリを C#/VB.NET へ公開することも出来ます。Boo にはコンパイラだけでなく、インタプリタも用意されているため、ちょっとしたプログラムなら手軽に開発、実行が出来ます。また、対話型シェルもありますので、ライブラリの調査、テストをする際に役に立つでしょう。

2. Boo のインストール

Boo の公式サイトではバイナリとソースコードが公開されていますが、Boo はまだ開発中なのでインストーラはもちろんのこと、最新バイナリも存在しません。旧バージョンのバイナリは存在しますが、最新のソースコードをチェックアウトしてビルドすることをお勧めします。

2－1. .NET Framework SDKの入手

Visual Studio .NET もしくは、.NET Framework SDK が必要となります。また、ビルドはコマンドプロンプトから行いますので、コマンドプロンプトから C#コンパイラが実行できる環境を用意しておいてください。

2－2. Subversionの入手

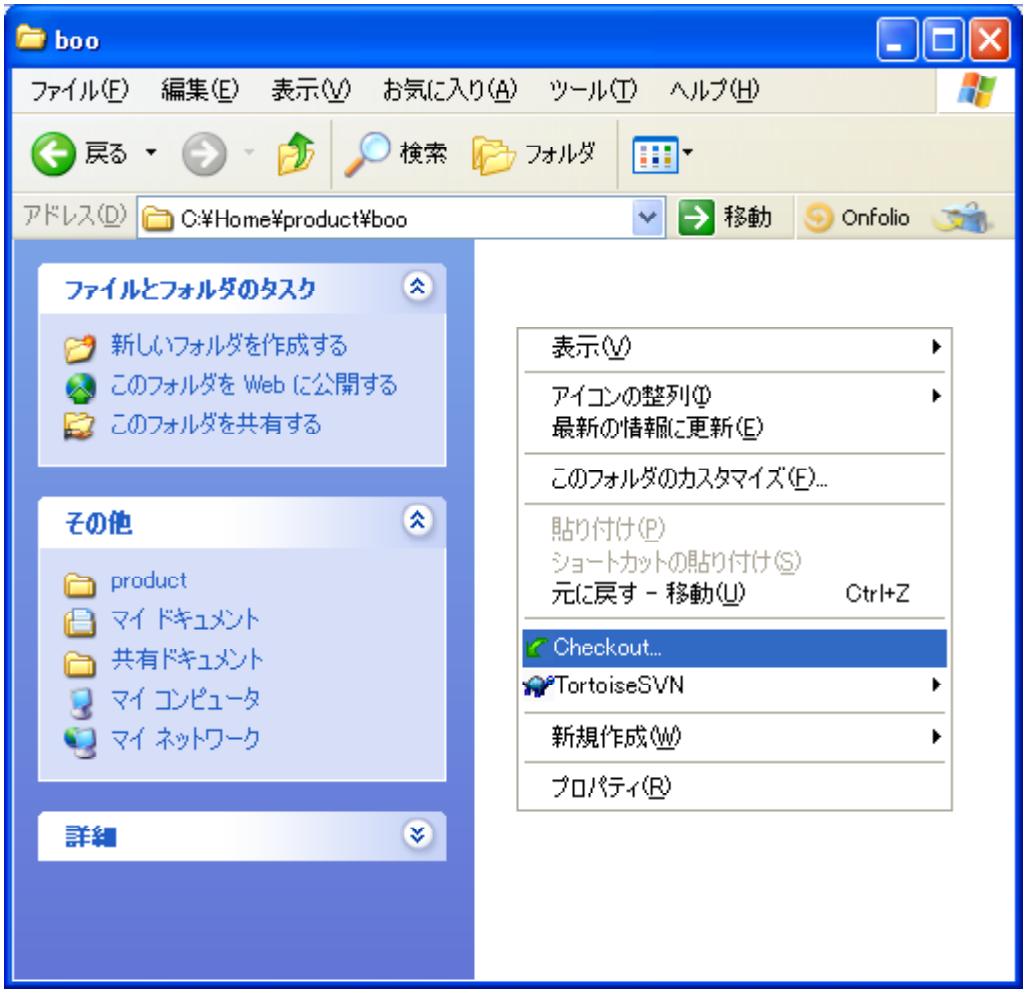
Boo のソースコードは Subversion で管理されています。最新のソースコードをチェックアウトするには、Subversion をインストールする必要があります。公式のダウンロードページ(http://subversion.tigris.org/project_packages.html)から入手してインストールしてください。もし、Windows ユーザでしたら GUI ベースの Tortoise (http://tortoisesvn.tigris.org/)がお勧めです。本ドキュメントでは Tortoise がインストールされていることを前提に説明します。

2－3. チェックアウト

Tortoise をインストールするとエクスプローラシェルが拡張され、ポップアップメ

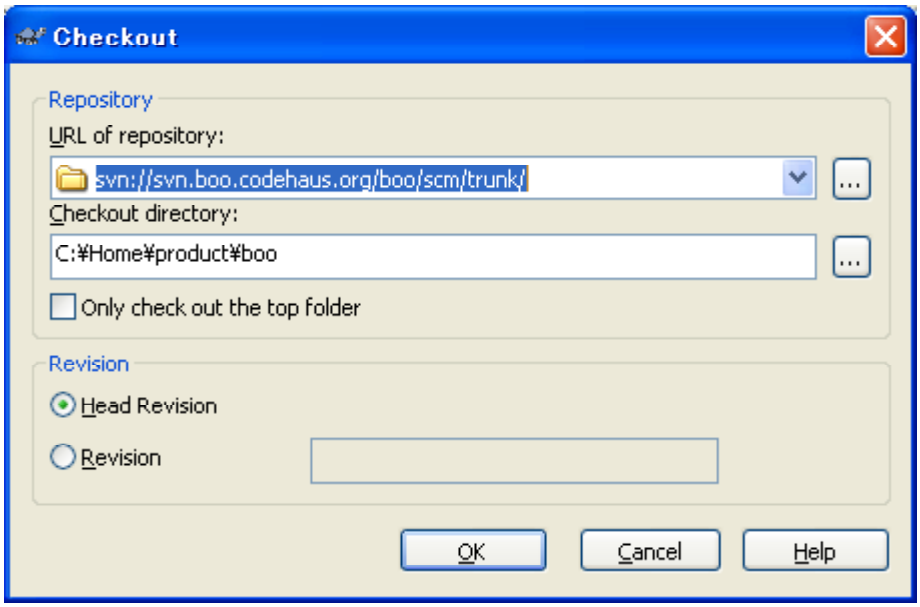
ニューに次のような項目が追加されます。

Subversion インストール後のポップアップメニュー



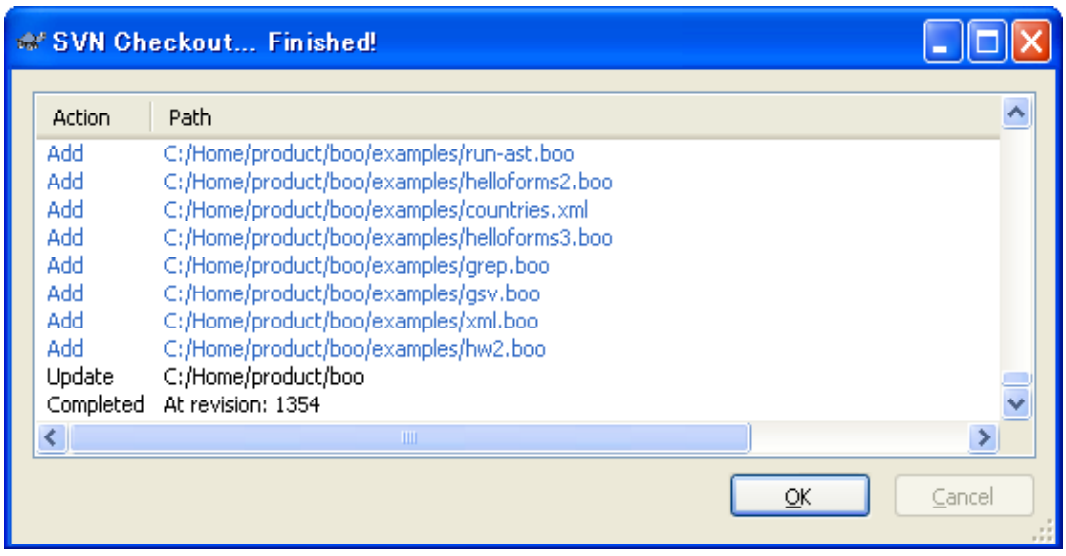
このように「Checkout...」と「TortoiseSVN」が追加されていることがわかりますね。では、「Checkout...」を選択して Boo のソースコードをチェックアウトしてみましょう。例では c:¥home¥product¥boo をチェックアウト先としています。

リポジトリの URL - svn://svn.boo.codehaus.org/boo/scm/trunk/



リポジトリの URL とチェックアウト先ディレクトリを確認したら OK ボタンを押下します。すると、Boo の開発環境一式がチェックアウトされます。

チェックアウト完了



2－4．Javaの入手

Boo のビルドに何故 Java が必要なのか？と疑問に思う人も多いと思います。Boo の構文解析に ALTLR というパーサージェネレータが使用されていて、これが Java で書かれています。

2－5. NAntの入手

Boo のビルドには NAnt が使われています。NAnt はソフトウェアのビルドを自動化するツールです。公式サイト(http://nant.sourceforge.net)からダウンロード出来ます。

2－6. Booのビルド

Java と NAnt をインストールし、パスを通したら Boo をビルドする準備は完了です。コマンドプロンプトを開いて Boo をチェックアウトしたディレクトリ(例の場合は c:¥home¥product¥boo)へ移動します。ここで、次のようなコマンドを実行します。

```
> nant rebuild
```

すると、自動ビルドが実行されます。ビルドに成功したら build ディレクトリに Boo のバイナリが作成されます。ここにあるバイナリを他のディレクトリにコピーしてパスを通せば、Boo を使用出来るようになります。必要なファイルは以下の通りです。

・pt (ディレクトリ)
・Boo.dll
・Boo.Lang.Compiler.dll
・Boo.Lang.Interpreter.dll
・Boo.Lang.Parser.dll
・Boo.NAnt.Tasks.dll
・booc.exe
・booc.rsp
・booi.exe
・booish.exe

3. 対話型シェルを使う

Boo がインストールされており、パスが通っている環境なら、

```
> booish
```

とコマンドを入力すれば、対話型シェルが使えます。ちなみに booc がコンパイラ、booi がインタプリタです。booish はインタプリタのシェル(sh)ということですね。

boosh が起動すると、

```
>>>
```

このようなプロンプトが表示されます。簡単な計算をさせてみましょう。

```
>>> 1 + 2
3
```

このように結果が返ります。結果はプロンプトのような>>>はつかないのですぐ分かります。複数の行からなる構文の場合は、...という二次プロンプトが表示されます。例として次の if 文を見てください。

```
>>> a = 10
10
>>> if a >= 10:
...     print "OK"
...
OK
```

if は複数の行からなるので、二次プロンプトが続きの行を催促します。何も入力せずに改行すると終了し、構文が実行されます。

4. Boo を試してみる

対話型シェルが使えるので、シェル上で色々動かしてみましょう。

4－1. 数

簡単な計算を行います。

```
>>> 2 + 2
4

>>> 2 + 2 # これはコメント
4

>>> 2 + 2 // これもコメント
4
```

```
>>> 2 + 2 /* これもコメント */
4
```

C#と同じく、=を使って変数に値を代入します。

```
>>> a = 10
10
```

複数の変数に同時に値を代入することも出来ます。

```
>>> x = y = z = 10
10
>>> x
10
>>> y
10
>>> z
10
```

浮動小数点もサポートされています。

```
>>> 1.2 + 3.4
4.6
```

また、最後に評価された値は、シェル上では変数_に格納されています。

```
>>> _ * 2
9.2
```

4－2. 文字列

もちろん、数だけでなく文字列も扱えます。文字列は、シングルもしくは、ダブルクォートで括ります。

```
>>> 'Hello, World'
'Hello, World'

>>> "Hello, World"
```

```
'Hello, World'
```

文字列の連結には+を使います。

```
>>> hello = "Hello" + ", " + "World"
'Hello, World'
```

文字列はインデックス表記を使えます。

```
>>> hello[0]
H
```

ただし、Boo の文字列(正確には BCL の String クラス)は変更不可なので、次のような書き換えを行おうとするとエラーが発生します。

```
>>> hello[0] = 'h'
----^
ERROR: Property 'System.String.Chars' is read only.
```

また、インデックス表記だけでなく、スライス表記によって部分文字列を取り出すことも出来ます。インデックス表記は[開始:終了]と:で区切り、終了は境界を含みません。この場合ですと、[開始, 終了)という範囲になります。

```
>>> hello[1:5]
'ello'
```

スライスには省略値があります。開始を省略すると 0、終了を省略すると文字列の長さ(つまり、文字列の最後の要素の次)となります。

```
>>> hello[:2]
'He'

>>> hello[2:]
'llo, World'
```

また、スライスの開始、終了には負の値を使用出来ます。その場合、文字列を後ろから数えた位置となります。

```
>>> hello[:-1]
'Hello, Worl'
```

-1 は文字列の先頭を 0 として後ろに 1 文字なので文字'd'を指します。スライスでは終了は境界を含まないので最後の文字'd'が削除されます。これを利用すると改行'¥n'を含む文字列から改行を削除することも簡単に出来ますね。

4ー3. リスト

Boo で最もよく使われるコレクション型です。コンマで区切ったリストを鉤括弧で括ることで書き表します。

```
>>> c = [1, 'two', 3.0, "four"]
[1, 'two', 3, 'four']
```

リストは文字列と違って変更可能です。

```
>>> c[2] = 3.3
>>> c
[1, 'two', 3.3, 'four']
```

文字列で使用したスライスはリストでも使用できます。

```
>>> c[2:]
[3.3, 'four']
```

リストへの追加には Add メソッドを使用します。

```
>>> c.Add(5)
[1, 'two', 3.3, 'four', 5]
>>> c.Add('six')[1, 'two', 3.3, 'four', 5, 'six']
```

AddUnique メソッドを使うと重複した値は追加されません。

```
>>> c.AddUnique(3.3)
[1, 'two', 3.3, 'four', 5, 'six']
>>> c.AddUnique(7.7)
[1, 'two', 3.3, 'four', 5, 'six', 7,7]
```

要素の削除には Remove を使用します。

```
>>> c.Remove(1)
```

```
['two', 3.3, 'four', 5, 'six', 7,7]
>>> c.Remove(5)
['two', 3.3, 'four', 'six', 7,7]
```

RemoveAt で位置による削除が出来ます。

```
>>> c.RemoveAt(2)
['two', 3.3, 'six', 7,7]
```

組み込み関数 len でリストの要素数が取得できます。

```
>>> len(c)
4
```

Clear で全削除です。

```
>>> c.Clear()
>>> len(c)
0
```

4－4. プログラミングっぽいこと

単純な計算や、文字列操作を見てきましたが、Boo はプログラミング言語なのでもちろん、もっと複雑なこともできます。試しに Fibonacci 級数を求めてみましょう。

```
>>> a, b = 0, 1 # a = 0, b = 1 を同時に行う
1
>>> while b < 10: # while 文 条件を満たす間、ループする
...     print '*', b # b の内容を表示
...     a, b = b, a + b # 次の Fibonacci 数を求める
...
* 1
* 1
* 2
* 3
* 5
* 8
13
```

最後の 13 は、インタプリタが最後に評価した値です。求めた Fibonacci 数と区別するために、計算した値には'*'をつけています。ところで複数行からなる構文では字下げを行っていますが、これは見やすくするためではなく、ブロック(C#で{...}で括ることと同様)を構成しているのです。例えば、Boo は同一の字下げを行っているところをブロックと見なしますなので、

```
>>> while b < 10:
...     print '*' b
... a, b = b, a + b
...
```

と書いてしまうと、while ブロックは print 関数までとなり、ひたすら b の値が出力されることになってしまいます。さらっと書いてしまいましたが、Boo ではとても重要なことなので忘れないようにしてください。

5. フロー制御及び、関数

5－1. if文

条件による分岐をさせます。分岐が増えると見難くなりますが、現状、if-elif を使うことになります。将来のバージョンでは given-when(C#における switch-case)がサポートされる予定です。

```
>>> if x < 0:
...     print 'Negative'
... elif x == 0:
...     print 'Zero'
... else:
...     print 'Positive'
```

5－2. for文

要素に対する反復処理を行います。C#の for では無く、foreach に相当します。

```
>>> cities = ['Tokyo', 'Sapporo', 'Osaka']
['Tokyo', 'Sapporo', 'Osaka']
>>> for city in cities:
...     print city, len(city)
...
```

```
Tokyo 5
Sapporo 7
Osaka 5
```

5－3. range関数

C#の for のようにループカウンタを使ってループさせる場合には、range 関数を使うと便利です。

```
>>> cnt = len(cities)
3
>>> for i in range(cnt):
...     print cities[i], len(cities[i])
...
Tokyo 5
Sapporo 7
Osaka 5
```

この例ですと、あえてループカウンタを使う必要はありませんね。ところで、range 関数の戻り値は何でしょうか？ ちなみに Python では range(5)とかすると、0 〜 4 のリストが返されました。Boo の場合は、

```
>>> r = range(5)
Boo.Lang.Builtins+RangeEnumerator
```

と RangeEnumerator が返されます。名前から想像できるように IEnumearator インタフェースを実装しているので、

```
>>> r.MoveNext()
true
>>> r.Current
0
>>> r.MoveNext()
true
>>> r.Current
1
```

このように呼び出すことも出来ます。

5－4. break文とcontinue文

C#の break/continue そのまんま。

```
>>> x = 0
0
>>> while true
...     ++x # x をインクリメント ちなみに x++はサポートしていない
...     if x < 3:
...         continue # 次の反復処理へ
...     if x > 8:
...         break # ループを終了する
...     print '*' * x # '*'を x 個表示
...
***
****
*****
******
*******
********
9
```

5－5. pass

何もしない文。文が構文上必要だけど処理を行う必要がない場合に使います。

```
>>> if x > 10:
...     pass
... else:
...     print x
```

ブロック内に処理がないことを明示的にしてしています。

5－6. 関数を定義する

先ほど書いた fibonacci 級数を求めるコードを関数にしてみましょう。

```
>>> def fib(n as int):
...     a, b = 0, 1
...     while b < n:
...         print '*', b
...         a, b = b, a + b
...
>>> fib(10)
* 1
* 1
* 2
* 3
* 5
* 8
```

このように関数定義は

```
def 関数名(引数):
        関数本体
```

のように書きます。もし、戻り値がある場合は、

```
def 関数名(引数) as 型:
```

とします。

5－7. 匿名関数

Boo では匿名関数をサポートしています。

```
>>> def make_incrementor(n as int):
...     return def (x as int):
...         return x + n
...
>>> f = make_incrementor(42)
Input11Module+___closure1.Invoke
>>> f(0)
42
>>> f(1)
43
```

このように def の後に関数名を指定しなければ匿名関数となります。また、匿名関数がブロックを必要としなければ、次のように書くこともできます。

```
>>> def make_incrementor(n as int):
...     return { x as int | x + n }
...
```

6. データ構造

リスト、ハッシュなどのビルトインデータ型について説明します。

6－1. リスト型について

リスト型については 3.3 でもふれましたが、メンバについてもう少し詳しく見てみます。

６－１－１．メソッド

List Add(object item)
要素をリストの最後に追加します。

```
>>> list = []
[]
>>> list.Add(1)
[1]
>>> list.Add("two")
[1, 'two']
```

List AddUnique(object item)
要素がリストに含まれて居なければ、リストの最後に追加します。

```
>>> list = []
[]
>>> list.AddUnique(1)
[1]
>>> list.AddUnique(1)
[1]
```

```
>>> list.AddUnique("two")
[1, 'two']
```

void Clear()
リストの要素を null で初期化し、要素数を 0 とします。キャパシティはそのままです。

```
>>> list = [1, 2, 3]
[1, 2, 3]
>>> list.Clear()
>>> list
[]
```

List Collect(Predicate condition)
条件を満たす要素の集合をリストとして返します。Predicate には object を引数として bool を返す関数を指定します。

```
>>> list = [i for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sub = list.Collect({x as int | x < 5})
[0, 1, 2, 3, 4]
```

List Collect(List target, Predicate condition)
条件を満たす要素を target に追加します。戻り値は target となります。

```
>>> list = [i for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sub = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Collect(sub, {x as int | <= 3})
['one', 'two', 'three', 0, 1, 2, 3]
```

bool Contains(object item)
item がリスト中に存在すれば、true を返します。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Contains('one')
true
>>> list.Contains('four')
```

```
false
```

bool Contains(Predicate condition)
条件を満たす要素がリスト中に存在すれば、true を返します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list .Contains({x as int | return x < 3})
true
>>> list .Contains({x as int | return x >= 10})
false
```

void CopyTo(Array target, int index)
リストの要素を配列の index 以降へにコピーします。配列に領域が足りない場合は例外が発生します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr = array(range(0, 20))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19)
>>> list.CopyTo(arr, 10)
>>> arr
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

List Extend(IEnumerable enumerable)
リストに要素を追加します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Extend(range(10, 20))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19]
```

object Find(Predicate condition)
条件を満たす最初の要素を取得します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Find({x as int | return x < 7})
```

```
0
```

List GetRange(int begin)
begin 以降の要素を取得します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.GetRange(3)
[3, 4, 5, 6, 7, 8, 9]
```

List GetRange(int begin, int end)
[begin, end)の範囲にある要素を取得します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.GetRange(3, 7)
[3, 4, 5, 6]
```

int IndexOf(Predicate condition)
指定した条件を満たす最初の要素のインデックスを取得します。見つからない場合は-1 を返します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.IndexOf({x as int | return x > 5})
6
```

int IndexOf(object item)
指定した要素のインデックスを取得します。見つからない場合は-1 を返します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.IndexOf(7)
7
```

List Insert(int index, object item)
指定した位置に要素を挿入します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list.Insert(5, 4.5)
[0, 1, 2, 3, 4, 4.5, 5, 6, 7, 8, 9]
```

string Join(string separator)
要素を separator で区切って文字列にします。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Join('|')
'0|1|2|3|4|5|6|7|8|9'
```

List Multiply(int count)
リストを count 分繰り返したリストを返します。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Multiply(3)
['one', 'two', 'three', 'one', 'two', 'three', 'one', 'two',
'three']
```

object Pop()
リストの最後の要素を取り出します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Pop()
9
>>> list
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

object Pop(int index)
index で指定した要素を取り出します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Pop(0)
0
>>> list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Push(object item)
リストの最後に要素を追加します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list.Push(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

List Remove(object item)
リストから要素を削除します。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Remove('two')
['one', 'three']
```

List RemoveAt(int index)
index で指定した要素を削除します。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.RemoveAt(1)
['one', 'three']
```

List Sort()
リストをソートします。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Sort()
['one', 'three', 'two']
```

List Sort(Comparer comparer)
comparer で指定した順にソートします。comparer には２つの object を引数にとって int を返す関数を指定します。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Sort({x as string, y as string | return y.CompareTo(x)
 })
```

```
['two', 'three', 'one']
```

Array ToArray(Type targetType)
リストを targetType 型の配列に変換します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr = list.ToArray(typeof(int))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> arr.GetType()
System.Int32[]
```

object[] ToArray()
リストを配列に変換します。

```
>>> list = [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr = list.ToArray()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> arr.GetType()
System.Object[]
```

６－１－２．プロパティ

int Count
リストの要素数。

```
>>> list = ['one', 'two', 'three']
['one', 'two', 'three']
>>> list.Count
3
```

６－１－３．リストをスタックとして使う

リスト型のメソッドを使用するとリストをスタックとして使えます。

```
>>> stack = [3, 4, 5]
[3, 4, 5]
>>> stack.Push(6)
[3, 4, 5, 6]
```

```
>>> stack.Push(7)
[3, 4, 5, 6, 7]
>>> stack.Pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.Pop()
6
>>> stack.Pop()
5
>>> stack
[3, 4]
```

６－１－４．リストをキューとして使う

リストをキューとして使うこともできます。

```
>>> queue = [3, 4, 5]
[3, 4, 5]
>>> queue.Push(6)
[3, 4, 5, 6]
>>> queue.Push(7)
[3, 4, 5, 6, 7]
>>> queue.Pop(0)
3
>>> queue.Pop(0)
4
>>> queue
[5, 6, 7]
```

６－１－５．ビルトイン関数との組み合わせ

ビルドイン関数"map(function, enumerable)"は enumerable のすべての要素 x に対し、function(x)を行った結果を enumerable として返します。例えば、1 ～ 9 を二乗した値のリストを作成するには次のようにします。

```
>>> list = List(map({x as int | return x * x}, range(1, 10)))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

６－１－６．リスト内包表記

前節の 1 〜 9 を二乗したリストはリスト内包表記を使うと次のように簡潔に書くことが出来ます。

```
>>> list = [x * x for x in range(1, 10)]
```

リスト内包表記はそれだけでなく、次のような柔軟性もあります。

```
>>> list = [x * x for x in range(1, 10) if x % 2 != 0]
[1, 9, 25, 49, 81]
>>> list = [x * x for x in range(1, 10) unless x % 2 == 0]
[1, 9, 25, 49, 81]
```

6－2. 配列

Boo の配列型は BCL の Array をそのまま使用しています。BCL のマニュアルを参照ください。配列はビルトイン関数 array や、要素を,(カンマ)で区切って括弧で括ることで作成することが出来ます。

```
>>> arr1 = array(typeof(int), 10) # int 型、要素数 10 の配列を作成
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
>>> arr2 = (1, 2, 3, 4, 5) # 要素 1, 2, 3, 4, 5 からなる配列を作成
(1, 2, 3, 4, 5)
```

6－3. ハッシュ

Boo の Hash 型は BCL の Hashtable を継承したものです。BCL のマニュアルを参照ください。ハッシュは Key:Value の組を,(カンマ)で区切って中括弧で括ることで作成することが出来ます。

```
>>> hash = {'one': 1, 'two': 2}
{'two': 2, 'one': 1}
>>> hash['one']
1
>>> hash['two']
2
```

ハッシュへの追加は、キーを指定して代入するだけです。

```
>>> hash['three'] = 3
>>> hash
{'three': 3, 'two': 2, 'one': 1}
```

すべての要素に対する処理は次のように行います。

```
>>> for e in hash:
...     print e.Key, e.Value
...
three 3
two 2
one 1
```

また、キー、値それぞれの列挙は次のようになります。

```
>>> for k in hash.Keys:
...     print k
...
three
two
one

>>> for v in hash.Values:
...     print v
...
3
2
1
```

7. ソースファイルとアセンブリ

7－1. ソースファイルからの実行

今まで、booish シェルでコードを打ち込んできましたが、シェルを終了するとすべての内容が消えてしまいます。より長いプログラムを作成するにはエディタでファイルにコードを打ち込んで保存することになります。試しに、フィボナッチ級数を求めるプログラムを作成し、fib.boo というファイルに保存してみましょう。

```
# fib.boo
```

```
def fib(n as int): # n 以下のフィボナッチ数を表示する
  a, b = 0, 1
  while b < n:
    System.Console.Write(b + " ")
    a, b = b, a + b
  print

def fib2(n as int): # n 以下のフィボナッチ数のリストを返す
  result = []
  a, b = 0, 1
  while b < n:
    result.Add(b)
    a, b = b, a + b
  return result

fib(1000)

print fib2(1000)
```

ソースファイルをコードを実行するには以下のコマンドを使用します。

```
> booi fib.boo
```

fib.boo は実行されて、次のような結果が表示されます。

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

対話型シェルで打ち込んでいた時とは違って、booi を使うことで fib.boo をいつでも実行できるようになりました。

7－2. 関数の再利用

関数 fib、fib2 は fib.boo 内で使用出来ますが、他のプログラムからも呼び出したい場合、どうすればいいでしょうか？ fib.boo からコピー＆ペーストで関数を持ってくることも出来ますが、もし、不具合があった場合にはコピーした箇所すべてを修正することになるので好ましくありません。Boo では関数をダイナミックリンクライブラリ(以降、DLL)として再利用します。DLL を作成するには booc.exe(Boo コン

パイラ)でコンパイルすることになります。コンパイルといっても難しくは無く、次のようなコマンドを実行すればいいです。

```
> booc -t:library fib.boo
```

-t オプションで作成するアセンブリの種類を DLL にしています。コンパイルが成功すると fib.dll が作成されます。ソースファイルが複数ある場合は、スペースで区切って列挙すればいいです。

```
> booc -t:library foo.boo bar.boo
```

コンパイルが成功すると foo.dll が作成されます。ソースファイルが複数ある場合は最初のソースファイル名で DLL が作成されます。もし、ソースファイルとは異なる DLL 名を付けたい場合は、-o オプションで出力ファイル名を明示的に指定することもできます。

```
> boo -t:library -o:mydll foo.boo bar.boo
```

この場合、mydll.dll が作成されます。

7ー3. 名前空間とimport

DLL によって他の人が作った関数を利用出来ることは良いことですが、それによって新たな問題が発生することがあります。名前の衝突です。例えば、他の人が作ったライブラリ、your.dll があったとします。ここに含まれている fib 関数を使いたいのですが、このライブラリには calc 関数も含まれていました。ところが、自分のライブラリ(my.dll)でも calc という名前の関数を作っていたら、関数名が衝突してどっちの calc か区別がつかなくなってしまいます。my.dll と your.dll を使ったアプリケーションをコンパイルすると次のようなエラーが発生します。

```
BCE0004: Ambiguous reference 'calc': Your Module.calc(),
MyModule.calc().
1 error(s).
```

どちらかが関数名を変更すればとりあえず、問題を解決できますが、影響が大きすぎて変更できない場合も当然あります。このような問題を解決するために、名前空間が用意されています。

```
# your.boo
```

```
namespace Your

def calc():
	print "your.calc"
```

```
# my.boo
namespace My

def calc():
	print "my.calc"
```

このようにファイルの先頭に"namespace　名前空間名"を入れることで、関数名の衝突を避けられます。また、これを呼び出す app.boo は次のようになります。

```
# app.boo

Your.calc()
My.calc()
```

このように"名前空間.関数"という書き方になります。いちいち"名前空間.関数"と打ち込むのが面倒な場合は、import を使うことで名前空間を指定しなくても良くなります。

```
# app.boo
import Your

calc() # Your.calc が呼ばれる
My.calc()
```

もちろん、名前空間 My、Your 両方を import してしまうと関数名は衝突してしまいます。

7－4. コンパイルについて

コンパイルで作成できるのは DLL だけでなく、EXE(実行モジュール)も作成することが出来ます。booi で実行する代わりに EXE に変換すれば、配布時にインタプリタを含める必要がなくなるので、最終的にはコンパイルすることになります。たとえば、fib.dll を使ったアプリケーションとして app.boo を開発したとします。開発時には、

```
> booi -r:fib.dll app.boo
```

このように、インタプリタでテストしたりします。コーディングが終われば、コンパイルを行って EXE に変換します。

```
> booc -t:exe -r:fib.dll app.boo
```

EXE になるとインタプリタは不要なので、

```
> app.exe
```

と実行することが出来ます。

索引