

Compatibility

Some basic compatibility:

```
namespace Boo.Compatibility.Python
class Dict(Hash):
    def constructor():
        super()
    def constructor(h as Hash):
        super()
        Update(h)
    def Update(h as Hash):
        for key in h.Keys:
            self[key] = h[key]
    def Get(key, default):
        return self[key] or default
    def SetDefault(key, default):
        if self.Contains(key):
            return self[key]
        else:
            self[key] = default
            return default
```

Recipe 1.1: Swap values without a temporary variable:

```
a = 4
b = 5
c = 6

//DIFF FROM PYTHON ---> print is a fxn,
although can be a macro
//Use print macro.., no ending comma
print a, b, c

a, b, c = b, c, a

print a, b, c
```

Recipe 1.2: Construct a Dictionary without excessive quoting

```
/*Recipe welcome

data = {'red':1, 'green':2, 'blue':3}
#"excessive quoting"
data = makedict((red=1, green=2, blue=3))
//Gives warnings about unused variables
print data["red"]
*/
```

Recipe 1.3: Getting a Value safely from a Dictionary

```

d = {'key': 'value'}

//DIFF FROM PYTHON ---> Contains or
ContainsKey instead of has_key
if d.Contains('key'):
    print d['key']
else:
    print('not found')

//DIFF FROM PYTHON ---> No get with a
default value.
#print(d.get('key', 'not found'))
//Boo equivalent because not found hashes
return null
print d['key'] or 'not found'

//DIFF FROM PYTHON ---> No nested fxns

```

Recipe 1.4: Adding an Entry to a Dictionary

```

def addword1(theIndex as Hash, word,
pagenumber):
    // As of 1/7/05 , autocompletion doesn't
work on theIndex in SharpDevelop
    if theIndex.ContainsKey(word):
        //DIFF FROM Python ---> List instead of
list
        //DIFF FROM Python ---> ArrayList value
must be cast for compiler to work (1/7/05)
        //DIFF FROM PYTHON ---> Add function

```

instead of append function.

```
(theIndex[word] as List).Add(pageneumber)
else:
    theIndex[word] = [pageneumber]
```

#VERSUS

```
def addword2(theIndex as Hash, word,
pageneumber):
    try:
        (theIndex[word] as List).Add(pageneumber)
    except AttributeError:
        theIndex[word] = [pageneumber]
```

//DIFF FROM Python ---> No setdefault.

#VERSUS

```
#def addword(theIndex as Hash, word,
pageneumber):
# theIndex.setdefault(word,
[]).append(pageneumber)
```

```
def wordcount(theIndex as Hash,word):
    //theIndex[word] = 1 + theIndex.get(word,
0)
    i as int= (theIndex[word] or 0)
    theIndex[word] = 1 + i
```

```
theIndex = {}
```

```
addword1(theIndex, "keys", 1)
addword2(theIndex, "job", 2)
```

```
//DIFF FROM Python ---> Hash instead of dict
//DIFF FROM Python ---> Able to declare
types of variables.
```

Recipe 1.5: Associating Multiple Values with Each Key in a Dictionary

```
//Recipe 1.5: Associating Multiple Values
with Each Key in a Dictionary
key = "email"
value = "jbob@bob.net"

#Allows duplicates
d1 = {}
//d1.setdefault(key, []).append(value)
//NO SETDEFAULT
if not d1.ContainsKey(key):
    d1[key] = []
(d1[key] as List).Add(value)

#Doesn't allow duplicates
d2 = {}
//d2.setdefault(key, {})[value] = 1
if not d2.ContainsKey(key):
    d2[key] = {}
(d2[key] as Hash)[value] = 1
print d2[key]
```

Recipe 1.6: Dispatching Using a Dictionary

```
//DIFFERENCE FROM PYTHON --> Referencing
global or module variables not allowed.
```

```
//Globals class per Doug Holton feedback.
class Globals:
    public static number_of_felines = 0

def deal_with_a_cat(animals as List):
    print "meow"
    animals.Add("feline")
    Globals.number_of_felines +=1

def deal_with_a_dog(animals as List):
    print "bark"
    animals.Add("canine")

def deal_with_a_bear(animals as List):
    print "watch out for the *HUG*!"
    animals.Add("ursine")

animals = []
tokenDict = {
    "cat": deal_with_a_cat,
    "dog": deal_with_a_dog,
    "bear": deal_with_a_bear
}
//Doesn't work in boo.. seems like dict
(since it is a hash table) loses the type..
//Casting as callable is necessary to
simulate the Python effect.
for word in ["cat", "bear", "cat", "dog"]:
    functionToCall = tokenDict[word] as
callable
```

```
functionToCall(animals)
#Or
(tokenDict[word] as callable)(animals)
```

Recipe 1.7 : Collecting a Bunch of Named Items

```
//Modified from Bill Woods Expando example
at
http://svn.boo.codehaus.org/boo/trunk/tests/
testcases/integration/duck-5.boo?view=auto
import System

class Bunch(IQuackFu):

    _attributes = {}

    def constructor(attributes as Hash):
        self._attributes = attributes

    def QuackSet(name as string, value):
        _attributes[name] = value
        return value

    def QuackGet(name as string):
        raise "attribute not found: " + name if
not _attributes.Contains( name)
        return _attributes[name]

    def QuackInvoke(name as string, args as
(object)) as object:
        pass
```

```
data = {'datum':2, 'squared':2*2, 'coord':5}
point as duck = Bunch(data)
print point.datum, point.squared,
point.coord

if point.squared > 4:
    point.isok = true
else:
```

```
    point.isok = false
print point.isok
*/
```

Recipe 1.8: Finding the Intersection of Two Dictionaries

```
#Bad way..
/*
def badslowway(some_dict as Hash,
another_dict as Hash):
    intersect = []

    for item as string in some_dict.Keys:
        if item in another_dict.Keys:
            intersect.Add(item)
    return intersect
*/

#Better:
def simpleway(some_dict as Hash,
another_dict as Hash):
    Globals.lastName = GetMethodName()
    intersects = []
    for k as string in some_dict.Keys:
        if another_dict.Contains(k):
            intersects.Add(k)
    return intersects

#Using List Comprehension
def listcompway(some_dict as Hash,
another_dict as Hash):
```

```

    Globals.lastName = GetMethodName()
    return [k for k as string in
some_dict.Keys if another_dict.Contains(k)]

#Using Filter
//def filterway(some_dict, another_dict):
//    return filter(another_dict.Contains,
some_dict.Keys)

class Globals:
    public static some_dict = {
'zope':'zzz', 'boo':'rocks'}
    public static another_dict = {
'boo':'rocks', 'perl':'$'}
    public static lastName as string

def doNothing(a as Hash, b as Hash):
    Globals.lastName = GetMethodName()
    return ''

def GetMethodName() as string:
    st = System.Diagnostics.StackTrace()
    sf as System.Diagnostics.StackFrame =
st.GetFrame(1)
    return sf.GetMethod().Name

def time(fun as callable, n):
    //Determine fxn call overhead
    a = []
    start = System.DateTime.Now.Ticks
    for i in range(n):

```

```
a.Add(doNothing(Globals.some_dict,Globals.another_dict))
```

```
    end = System.DateTime.Now.Ticks
```

```
    overhead = end-start
```

```
    a = []
```

```
    start = System.DateTime.Now.Ticks
```

```
    for i in range(n):
```

```
a.Add(fun(Globals.some_dict,Globals.another_dict))
```

```
    end = System.DateTime.Now.Ticks
```

```
    duration = end-start-overhead
```

```
    print Globals.lastName,
```

```
System.TimeSpan.FromTicks(duration)
```

```
for f as callable in
```

```
[simpleway,listcompway]:  
//,filterway,badslowway]:  
    time(f,10000)
```

Recipe 1.9: Assigning and Testing with One Statement.

```
//In python the following isn't easy:  
//If x=fxn():  
// doSomething()  
//In boo, it is. Remember == is the test  
for equality for boo.  
def inc(x as int):  
    return x + 1  
  
if y = inc(5):  
    print "Passed"  
else:  
    print "Failed"  
  
//returns "Passed"
```

Recipe 1.10: Using List Comprehensions instead of map and filter.

```
//Differences:
//Python provides some builtins to do
functional programming: map, filter.
//Boo provides only a map builtin.

//In any case, using list comprehensions is
often a more readable approach.

def square(v as int):
    return v*v

print "Using map:"
results = map((1,3,5,6),square)
for r in results:
    print r

print "Using list comprehensions"
results = [square(x) for x in (1,3,5, 6)]
for r in results:
    print r

//list comprehensions can also filter the
results in a way that map by itself can't
(but python could using filter).
print "Return only the squares of evens"
results = [square(x) for x in (1,3,5,6) if x
% 2 == 0]
for r in results:
    print r
```

```
//A recipe to provide the unzip counterpart  
to the zip builtin.
```

```
/*  
*****  
*****  
*****
```

```
Split a sequence p into a list of n tuples,  
repeatedly taking next unused element of p  
and adding it to the next tuple.
```

```
Each of the resulting tuples is of the same  
length; if p%n != 0, the shorter  
tuples are padded with null (closer to the  
behavior of map than to that of zip--  
in python at least.)
```

Example:

```
>>> unzip(['a','b','c','d','e'],3)  
[('a','d'),('b','e'),('c',null)]
```

```
*****  
*****/  
*****
```

```
def unzip(p as List, n as int) as List:  
    //First, find the length for the longest  
    sublist.
```

```
    lft as int  
    mlen = System.Math.DivRem(p.Count, n,  
lft)
```

```
    if lft != 0:  
        mlen +=1
```

```
    //Then, initialize a list of lists with
```

suitable lengths

```
lst as List = [[null]*mlen for i in  
range(n)]
```

```
//Loop over all items of the input  
sequence (index-wise), and Copy
```

```
// a reference to each into the  
appropriate place.
```

```
k as int
```

```
for i in range(p.Count):
```

```
    j = System.Math.DivRem(i, n, k)
```

```
//Find sublist-index and  
index-within-sublist
```

```
    (lst[k] as List)[j] = p[i]
```

```
//Copy a reference appropriately
```

```
//Finally, turn each sublist into an  
array, since the unzip function
```

```
//is specified to return a list of  
arrays, not a list of lists.
```

```
return [array(val) for val in lst ]
```

```
x = unzip(['a','b','c','d','e'],3)
print x
```

Recipe 1.12: Flattening a nested sequence

```
import System.Collections

def flatten(sequence as List, isScalar as
callable, result as List) as List:
    if result is null:
        result = []
    for item in sequence:
        if isScalar(item):
            result.Add(item)
        else:
            flatten(item,isScalar,result)
    return result

//Using Generators
def flattenWithGenerators(sequence as List,
isScalar as callable) as object:
    for item in sequence:
        if isScalar(item):
            yield item
        else:
            for subitem in flattenWithGenerators(item
as List,isScalar):
                yield subitem

//Checking if an item is loopable
```

```
def canLoopOver(item) as bool:
    ie = item as System.Collections.IEnumerable

    return not ie is null
```

```
def isStringLike(obj) as bool:
    try:
        x = obj + ''
        return true
    except:
        return false
```

```
def isScalar(obj) as bool:
    return isStringLike(obj) or not
    canLoopOver(obj)
```

```
//Provided by Doug Holton:
//In boo you can do it like this:
```

```
def flatten2(seq as List):
    l = []
    for subseq as List in seq:
        l += subseq
    return l
```

```
//or a more general solution perhaps:
```

```
def flatten3(obj) as List:
    l = []
    if obj isa IEnumerable and not (obj
    isa string):
        for item in obj:
```

```
                l += flatten3(item)
else:
    l.Add(obj)
return l

print flatten2([[0], [1,2,3], [4,5],
[6,7,8,9], []]) //->[1,2,3,4,5,6,7,8,9]

print flatten3([[0], [1,2,3], [4,5],
[6,7,8,9], []]) //->[1,2,3,4,5,6,7,8,9]
print flatten3(1) //-> [1]
print flatten3([[[[1]]],[2],3,[[4,5]])] //->
[1,2,3,4,5]
```

```
y as List= [[1,2],2,3,[4,5]]
print flatten(y,isScalar,[])
print flattenWithGenerators(y,isScalar)
```

Recipe 1.13: Looping in Parallel over Index and Sequence Items

```
//Looping over a list and knowing the index
of a particular item.
import System.Reflection

//This (A):
[DefaultMember("Item")]
class Indexed(List):
  seq as List
  def constructor(seq as List):
    self.seq = seq

  def Item():
    for idx in range(seq.Count):
      yield seq[idx],idx

def something(item, index):
  print "something + " + item + " + " + index

sequence = ["Cat", "Dog", "Elephant"]
indices = range(System.Int32.MaxValue)

//(A) Cont'd
```

```
//(A) Cont'd
for item, index in Indexed(sequence).Item():
    something(item, index)
```

```
//or this (B):
for item, index in zip(sequence, indices):
    something(item, index)
```

```
//is approximately equivalent to this (C):  
for index in range(len(sequence)):  
    something(sequence[index], index)
```

Recipe 1.14: Loop through every item of multiple lists

```
//To loop through multiple lists .. one  
element at a time..
```

```
def loop(a as List, b as List):  
    for i in range(a.Count):  
        if i < b.Count:  
            yield (a[i], b[i])  
        else:  
            yield (a[i],null)
```

```
//data
```

```
a = ['a1','a2','a3']
```

```
b = ['b1','b2']
```

```
//Python map approach is not valid.
```

```
print "Zip:"
```

```
for x,y in zip(a,b):
```

```
    print x,y
```

```
print "3rd iteration: a3 is not done"
```

```
print "List Comprehension:"  
for x,y in [(x,y) for x in a for y in b]:  
    print x,y
```

```
print "Generators: Methods"  
for x,y in loop(a,b):  
    print x,y
```

```
print "Generators: Expressions"  
for x,y in ((a[i],b[i]) for i in
```

```
range(a.Count)):
    print x,y
//Exception at the last one.
```

Recipe 1.15 Spanning a Range Defined by Floats

```
def frange(start as double, end as double,
inc as double):
//A range-like function that does accept
float increments...
    assert inc>0, "Increment must be greater
than 0."
    L = []
    while 1:
        next = start + len(L) * inc
        if next >=end:
            break
        L.Add(next)

    return L

def frange(start as double, inc as double):
    end = start
    start = 0.0
    return frange(start, end, inc)

def frange2(start as double, end as double,
inc as double):
//A faster range-like function that does
accept float increments..
    assert inc>0, "Increment must be greater
```

than 0."

```
count as int = (end-start) / inc
if start + count * inc != end:
    //Need to adjust the count. It comes up
one short.
    count +=1

L = [start] * count
for i in range(1, count):
    L[i] = start + i * inc

return L
```

```
def frange2(start as double, inc as double):
    end = start
    start = 0.0
    return frange2(start,end, inc)
```

```
def frangei(start as double, end as double,
inc as double):
    //A generator version of xrange that accepts
floats
    assert inc>0, "Increment must be greater
than 0."
    i = 0
    while 1:
        next = start + i * inc
        if next >= end:
            break
        yield next
```

```
i+=1
```

```
def frangei(start as double, inc as double):  
    end = start  
    start = 0.0  
    return frangei(start,end, inc)
```

```
print "frange: "  
print frange(-5,5,1.7)  
print "frange2: "
```

```
print frange2(-5,5,1.7)
print "frangei: "
print frangei(-5,5,1.7)
```

Recipe 1.16:

```
//Recipe 1.16 - Transposing Two-Dimensional
Arrays
arr = [[1,2,3], [4,5,6],[7,8,9],[10,11,12]]

//Simple route: List comprehension:
x = [[r[col] for r as duck in arr] for col
in range(len(arr[0]))]
print x

//or if one line matters:
print( [[r[col] for r as duck in arr] for
col in range(len(arr[0]))])
```

Recipe 1.17 - Creating Lists of Lists Without Sharing References

```
//Dont use macro when first character is
left-bracket. Causes callable does not
support slicing error.
print( [0]*5)
multi = [[0] * 5]*3
print multi
i as List = multi[0]
i[0] = -999999
print multi //Note that all 3 arrays had
their first value changed!!

print string.Empty
//To prevent this:
//Use list comprehension:
multilist = [[0 for col in range(5)] for row
in range(3)]
print multilist
i = multilist[0]
i[0] = -999999
print multilist //Note that the data is
protected.
```