CSC326 Python Sequences

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1	Agenda	1
2	while Statement	1
3	Sequence Overview	2
4	String	2
5	Lists	4
6	Dictionary	5
7	Tuples	7
8	What We have Learned for Sequences	9
9	Sequences are Objects	10
10	List Comprehension	10
11	Cross product of two sets	11
12	Accessing Matrics	12
13	Prime Number with List Comprehension	12
14	Quick Sort	12
15	Recap	12

1 Agenda

- · iteration construct
- sequences
 - string
 - list
 - dictionary
 - tuple

2 while Statement

- recall recursive construction of countdown
- cost of function calls
- good old while loop

```
def countdown( n ) :
    while n > 0 :
        print n
        n = n - 1
    print 'Blastoff!'
```

• forever loop

```
while True :
    line = raw_input('>')
    if line == 'done' :
        break
    print line
print 'Done'
```

• square root approximation algorithm

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

- Newton's algorithm
- · A few caveats

```
epsilon = 0.01
def sra( a ) :
    x = a / 2
    while True:
        print x
        y = (x + a/x) / 2
        if abs(y-x) < epsilon :
            break
        x = y
    return x</pre>
```

- Recap
 - Multiple assignment: vars are initialized and assigned more than once
 - loop often has loop index, incremented by one each iteration
 - conditional evaluation and branch forming cycle

3 Sequence Overview

- an ordered collection (set) of values
 - membership test
 - subset
 - eumeration
- another angle: a mapping from an index to value
 - lookup
 - reverse lookup (searching, content address memory)
- data structures in C
- native data types with native operators in Python
 - make python enjoyable to read and write!

4 String

- We have already seen one: string is a sequence!
- sequence of characters
- indexed lookup using []

```
>>> fruit = 'banana'
>>> print fruit[0]
>>> print fruit[1]
```

• index has to be an integer

```
>>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

• builtin function len

```
>>> len(fruit)
6
```

• index has to be within range

```
>>>> letter = fruit[len(fruit)]
IndexError: string index out of range
```

• enumeration: for statement

```
for char in fruit print char
```

• compare with C

```
for( i = 0; i < strlen(fruit); i ++ )
    printf( "%c", fruit[i] );</pre>
```

- subsetting: string slices
 - [first:last]: from first to last, excluding last
 - [first:]
 - [:last]
 - **-** [:] ?

```
print fruit[0:3]
print fruit[0:]
print fruit[:3]
print fruit[3:3]
```

- strings are "immutable"
 - element cannot be assigned (NOT an Lvalue)
 - has to create new string of wants to modify

```
greeting = 'hello, world'
greeting[0] = 'J'  # wrong!
new_greeting = 'J' + greeting[1:]
```

• searching (reverse lookup)

```
def find( word, letter ) :
   index = 0
   while index < len(word) :
        if word[index] == letter :
            return index
        index = index + 1
   return -1</pre>
```

• membership/subset test: in operator

```
>>>> 'a' in 'banana'
True
>>>> 'seed' in 'banana'
False
```

• print all letters appears in both word1 and word2

```
def in_both( word1, word2 ) :
    for letter in word1 :
        if letter in word2 :
            print letter
```

• comparison operator ==

```
>>> string1 = 'hello'
>>> string2 = 'he' + 'llo'
>>> print string1 == string2
True
```

• Python strings are values C strings are references

```
strcpy( string1, "hello1" );
strcpy( string2, "he" );
strcat( string2, "llo" );
printf( "%d", string1 == string2 )
```

• How do we compare by reference? is operator

```
>>> string1 = 'hello'
>>> string2 = 'he' + 'llo'
>>> print string1 is string2
False
```

• !=, <, >, \(\infty \), works too!

5 Lists

- Strings are sequence of values of fixed type: characters
- Lists are sequence of values (element, item) of arbitary types
- Form:

```
[ a, b, c, ... ][]: empty list
```

• Examples

```
[10, 20, 30, 40] #uniform types
['spam', 2.0, 5, [10, 20]] #non-uniform types, even nested!
```

· Indexed lookup

• Mutable

```
fruit[0] = 'grape'
```

• Eumeration

```
for cheese in cheeses :
    print cheese
```

• range(.) create a integer sequence from a scalar

```
for i in range( len(numbers) ) :
   numbers[i] = numbers[i] * 2
```

• Concatenation and Repetitiong (just like strings)

```
print [1, 2, 3] + [4, 5, 6]
print [0] * 4
print [1, 2, 3] * 4
```

• Subsetting: list slices

```
t = ['a', 'b', 'c', 'd']
t[1.3] = ['x', 'y']
```

This gives you:

```
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not float
```

- Comparison (same as string)
 - By value: == !=
 - By reference: is

NOTE - assignment is always by reference - assignment create alias

6 Dictionary

- The index of list has to be integers
- The index of dictionary can be arbitary types
- Form:
 - { key:val, key:val, ... }
 - {}: empty dictionary
 - dict(): empty dictionary

```
>>> print { 0: 2, 1: 3, 2: 5}
>>> eng2sp = { 'one':'uno', 'two':'dos', 'three':'tres' }
>>> print type(eng2sp)
>>> print eng2sp['one']
```

• Lookup: index of any types

```
>>> print eng2sp['one']
```

- Membership: in operator (constant time algorithm)
 - use key to test
- Example

```
def histogram( s ) :
    assert isinstance(s,str) # Do we need it?
    d = {}
    for c in s :
        if c not in d :
            d[c] = 1
        else :
            d[c] += 1
    return d
```

- histogram(good)?
- histogram([good, good, bad])?

Note

Do we need the assertion? polymorphic programming!

- Enumeration
 - of keys (indices)
 - list/string enumerate values

```
def print_histogram( h ) :
   for c in h :
      print c, h[c]
```

• Search

```
def reverse_lookup( h, v ) :
    for k in h :
        if( h[k] == v ) :
            return k
    return None
```

- I lied that key can be arbitary types
 - they have to be immutable!

- Why? (hint: think about hash table implementation)
- Example: cache/dynamic programming
 - We use divide/conquer to solve problem
 - typically map well to recursion
 - Recall fibonacci

```
def fibonacci(n) :
    if n in [0,1] :
        result = n
    else :
        result = fibonacci(n-1) + fibonacci(n-2)
    return result
```

- · Souce of inefficiency
- Dictionary comes to the rescue

```
cache = {}
def fibonacci(n) :
    if( n in cache ) :
        return cache[n]
    if n in [0,1] :
        result = n
    else :
        result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result
    return result
```

• Enhancement

```
cache = { 0:0, 1:1 }
def fibonacci(n) :
    if( n in cache ) :
        return cache[n]
    result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result
    return result
```

Note

We have used global variable cache if a variable is reassigned in a function, by default it is considered as a local variable (scope is within function). To tell Python it is a global variable, use global foo

7 Tuples

- Sequence of values (just like list)
- Immutables!
- Forms
 - a, b, c, ...

```
- (a, b, c, ...)
- ()
```

- tuple()
- Lookup(Index), Subsetting(Slice), Enumeration (for) works as usual
- Tuple variable assignment
 - In C

```
temp = a;
a = b;
b = temp;
```

• In Python

```
a, b = b, a
```

• Multiple return values

```
quot, rem = divmod( 7, 3 )
print quot  # gives 2
print rem  # gives 1

def min_max( t ):  # t is a sequence
    return min(t), max(6)  # min/max are builtin functions
```

- Variable argument: Gather
 - In C

va_end(va);

```
void printall( ... ) {
    va_list    va;

va_start( va, ... );
for( ... ) {
    arg = va_arg( va, int );
    printf( "%d ", arg );
}
```

• In Python (* operator): Gather all arguments into a tuple

```
def printall(*args) :
    print args
```

```
>>> print printall( 1, 2.0, '3') (1, 2.0, '3')
```

• Scatter: expand tuple into arguments

```
>>> t = (7,3)
>>> print divmod( *t )
(2,1)
```

- Zip
 - Builtin function
 - From: two or more sequences
 - To: a list of tuples, each element of tuple is taking from respective element from sequence

```
>>> s = 'abc'  # a string
>>> t = [0, 1, 2]  # a list
>>> zip(s, t)
[('a',0'), ('b',1), ('c,2)]
```

• Interesting to see how we can enumerate

```
for letter, number in t
    print number, letter
```

• Did you ever need to travese two sequences at the same time?

```
def has_match( t1, t2 ) :
    for x, y in zip( t1, t2 ) :
        if x == y :
            return True
    return False
```

• Do you need the index and value at the same time?

```
for index, val in enumerate('abc'):
    print index, element
```

· Fast way to create dictionary

```
d = dict( zip('abc', range(3)) )
print d
```

• Tuple can be key for dictionary

8 What We have Learned for Sequences

• We have seen different types of sequences

```
string ..., "..."list [...]dictionary { ... }tuple (...)
```

- Common set of operation
 - indexing and slicing: [...]
 - search
 - eumeration: for / in

9 Sequences are Objects

- Constructors
- · string methods
 - s.upper()
 - s.find(a)
 - s.split()
- list methods
 - l.append(e)
 - 1.extend(12)
 - 1.sort()
 - l.pop(index)
 - l.pop(): remove last element
 - l.remove(val)
 - del l[index]
- Simple Stack

```
def stack_init() :
    return []
def stack_push( s, e ) :
        s.append( e )
        return s
def stack_pop( s ) :
        e = s.pop()
    return e
```

• Simple FIFO?

```
def fifo_init() :
    return []
def fifo_enqueue( s, e ) :
    s.append( e )
    return s
def fifo_dequeue( s ) :
    e = s.pop( 0 )
    return e
```

10 List Comprehension

• Common pattern of deriving sequence from other sequences

- So common that Python devise a **construct** for it
- List comprehension! (Since Python 2.0)
 - Contributors: Greg Ewing, Skip Montanaro and Thomas Wouters

```
12 = [ i * 2 for i in 11]
```

· Look like math notation

```
S = {x$^2$ : x in {0 ... 9}}
V = (1, 2, 4, 8, ..., 2$^1$$^2$)
M = {x | x in S and x even}

>>> S = [x**2 for x in range(10)]
>>> V = [2**i for i in range(13)]
>>> M = [x for x in S if x % 2 == 0]
>>> print S; print V; print M
```

• Applying to any element type

```
>>> words = 'The quick brown fox jumps over the lazy dog'.split()
>>> print words
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> stuff = [[w.upper(), w.lower(), len(w)] for w in words]
```

· General syntax

```
L = [F(x) \text{ for } x \text{ in } S \text{ if } P(x)]
```

- Function F(x) is **mapped** to every x
- Function P(x) is the **filter**
- F(x) and P(x) are really just expressions
- Nested: there could be multiple for with its own predicte
- Big deal:
 - Expression, not statement
 - Almost like Math notation

11 Cross product of two sets

```
>>> colours = [ "red", "green", "yellow", "blue" ]
>>> things = [ "house", "car", "tree" ]
>>> coloured_things = [ (x,y) for x in colours for y in things ]
>>> print coloured_things
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), \( 'green', 'tree'), ('yellow', 'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', \( 'house'), ('blue', 'car'), ('blue', 'tree')]
>>>
```

12 Accessing Matrics

• No easy way to find column!

```
>>> [r[2] for r in M1]
[3, 6, 9]
```

13 Prime Number with List Comprehension

• sieve of Eratosthenes

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>> primes = [x for x in range(2, 50) if x not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

14 Quick Sort

```
def qsort(s) :
    if( len(s) <= 2 ) : return s
    pivot = s[0]
    less = [e for e in s if e < pivot]
    grt = [e for e in s if e > pivot]
    eq = [e for e in s if e == pivot]
    result = [qsort(v) for v in [less, grt]]
    return result[0] + eq + result[1]
print qsort( [2, 1, 5, 3, 4] )
```

15 Recap

- Data Structures are native types
- Common set of operations
 - []
 - **-** [:]
 - in
 - for
- Polymorphic programming
 - Canadian value: care what you can do, not who you are
- Immutables