Workshop 2

Strings

Strings are immutable sequences: the characters in the string have a left-to-right order and they cannot be changed in place.

S = ‘’	Empty string
S = “spam’s”	Double or single quotes
S = ‘s\np\ta\x00m’	Escape sequences
S = “”” …*multiline*…”””	Triple quoted block strings
S = r’\temp\spam’	Raw strings (no escapes)
B = b’sp\xc4m’	Byte strings
U = u’sp\u00c4m’	Unicode strings

Concatenation

```
>>>S = ‘spam’
>>>S + ‘ is good’
‘spam is good’
```

Replication (repeat)

```
>>>S * 2
‘spamspam’
```

Length

```
>>> len(S)
4
```

Single and double quoted strings are the same.

```
>>> ‘shrubbery’, “shrubbery”
(‘shrubbery’, ‘shrubbery’)
```

```
>>>’knight”s’, “knight’s”
’knight”s’, “knight’s”
```

Triple Quotes Code Multiline Block Strings

```
>>>mantra = “””Always look
...   on the bright
...side of life.”””
>>>
>>>mantra
‘Always look\n   on the bright\nside of life.’
```

```
>>>print(mantra)
Always look
   on the bright
side of life.
```

Note that any comments embedded in the block will appear as part of the quoted string.

You can use triple quotes to disable a block of code (it makes the code a comment block instead).

Conversions

```
>>>str(42)
‘42’
```

```
>>>int(“42”)
42
```

```
>>>float(“1.34”)
1.34
```

```
>>>str(1.34)
‘1.34’
```

```
>>>S = ‘10’
>>>S + 12
```

Error – mix of string and integer.

Indexing and Slicing

```
>>>S = ‘spam’
>>>S[0], S[-2]
(‘s’, ‘a’)
>>>S[1:3], S[1:], S[:-1]
(‘pa’, ‘pam’, ‘spa’)
```

[start:end]
Indexes refer to places the knife “cuts”

0	**1**	**2**							**-2**	**-1**
S	L	I	C	E	O	F	S	P	A	M
[:										**:]**

Defaults are beginning of sequence and end of sequence.

Indexing (S[i]) fetches components at offsets:

- The first item is at offset 0.
- Negative indexes mean to count backward from the end or right.
- S[0] fetches the first term.
- S[-2] fetches the second item from the end (like S[len(S)-2]).

Slicing (S[i:j]) extracts continuous sections of sequences:

- The upper bound is noninclusive.
- Slice boundaries default to 0 and trhe sequence length, if omitted.
- S[1:3] fetches items at offset 1 up to but not including offset 3.
- S[1:] fetches items at offset 1 through the end of the sequence.
- S[:3] fetches the items at offset 0 up to but not including 3.
- S[:-1] fetches the items at offset 0 up to but not including the last item.
- S[:] fetches the items at offsets 0 through the end – making a top-level copy of S. (Note: makes an object with the same value but located in a different part of memory.)

Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1.

- Allows for skipping items and reversing order.

```
>>>S=’0123456789’
>>>S[1:10:2]
‘13579’
>>>S[::-1]
‘9876543210’
>>>S[5:1:-1]
‘5432’>>>S = S[:4] + ‘Burger’ + S[-1]
>>>S
‘spamBurger!’
```

String Methods

There are 43 string methods (see pages 217-218)
S.find(‘pa’)
S.rstrip()
S.replace(‘pa’, ‘xx’)
S.split(‘,’)
S.isdigit(), S.isalpha()

capitalize, upper, lower
isalnum, isalpha, isdecimal, isdigit, islower, isupper, isspace, isnumeric
lstrip, rstrip
find, replace

```
>>>S=’xxxxSPAMxxxxSPAMxxxx’
>>>S.replace(‘SPAM’, ‘EGGS’)
‘xxxxEGGSxxxxEGGSxxxx’

>>>line=’bob,hacker,40’
>>>line.split(‘,’)
[‘bob’, ‘hacker’, ‘40’]
```

Strings are sequences. That means everything we learn about manipulating a string sequence applies to other object types that are sequences (e.g., lists).

Strings are immutable. To change a string in place requires changing the string to a mutable object type (e.g., a list), changing that object type in place, and then changing that object type back into a string.

Lists

A list is a mutable sequence. The objects in the list are stored in an order and they can be changed in place.

- Ordered collections of arbitrary objects
- Accessed by offset
- Variable length, heterogeneous, and arbitrarily nestable
- Of the category *mutable sequence*
- Arrays of object references – Technically, Python lists contain zero or more references to other objects.

With respect to sequences, what you learned with strings can be done with lists.

L = []	empty list
L = [123, ‘abc’, 1.23, {}]	Four items: indexes 0 to 3
L = [‘Bob’, 40.0, [‘dev’, ‘mgr’]]	Nested sublists
L = list(‘spam’)	List of iterable items, list of successive integers
L = list(range(-4, 4))	
L[i], L[i][j], L[i:j]	Index, index of index, slicing
len(L)	Length
L1 + L2, L*2	Concatenation and replication (repeat)
For x in L: print(x)	Iteration
3 in L	Membership
L.append, L.extend([5,6,7]), L.insert(I, X)	Methods for growing
L.index(X), L.count(X)	Methods for searching
L.sort(), L.reverse(), L.copy(), L.clear()	Methods for sorting, reversing, copying, clearing
L.pop(i), L.remove(X), del L[i], del L[i:j], L[i:j] = []	Methods, statements for shrinking
L[i] = 3, L[i:j] = [4,5,6]	Index assignment, slice assignment
L = [x**2 for x in range(5)	List comprehension
list(map(ord, ‘spam’))	List map

```
>>> len([1,2,3])
3

>>> [1,2,3] + [4,5,6]
[1,2,3,4,5,6]

>>> [‘spam’] * 2
[‘spam’, ‘spam’]

>>>L = [‘spam’, ‘Spam’, ‘SPAM!’]
>>>L[2]
‘SPAM!’
>>>L[-2]
‘Spam’
```

```
>>>L[1:]
['Spam', 'SPAM!']
```

Index and Slice Assignments

```
>>>L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'
>>>L
['spam', 'eggs, 'SPAM!']

>>>L[0:2] = ['eat', 'more']
>>>L
['eat', 'more', 'SPAM!']
```

Think of slice assignments as a deletion followed by an insertion. The number of items inserted does not have to equal the number of items deleted.

Other tricks:

```
>> L = [1]
>>> L[:0] = [2, 3, 4]       #insert at the beginning
>>>L
[2, 3, 4, 1]

>>>L[len(L):] = [5, 6, 7]   #insert at the end
>>>L
[2, 3, 4, 1, 5, 6, 7]

>>>L.extend([8, 9, 10])   #insert at end
>>>L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]

>>>L.append([11, 12])   #insert at end
>>>L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10, 11, 12]
```

Notice:

```
>>>L = [2, 4, 3, 1]
>>>L.sort()                 #sorted list is a by-product, no need for assignment
>>>L
[1, 2, 3, 4]

>>>L.reverse()              #no need for assignment
>>>L
[4, 3, 2, 1]
```

Also:

```
>>>L = [4, 2, 3, 1]
>>>L.sort(reverse=True)
>>>L
[4, 3, 2, 1]
```

Finally:

```
>>>L.remove(4)
>>>L
[3, 2, 1]

>>>L.insert(1, 4)
>>>L
[3, 4, 2, 1]

>>>L[1:] = []
>>>L
[3]
```

Dictionaries

Dictionaries are unordered, mutable mappings.

- Accessed by key, not offset position
- Unordered collection of arbitrary objects
- Variable-length, heterogeneous, and arbitrarily nestable
- Of the category ‘mutable mapping’
- Tables of object references

Dictionary basics

D = {}	Empty dictionary
D = {‘name’: ‘Bob’, ‘age’: 40}	Two-item dictionary
E = {cto: {‘name’: ‘Bob’, ‘age’: 40}}	Nesting
D = dict(name=’Bob’, age = 40)	Ways to create dictionaries
D = dict([(‘name’, ‘age’), (‘Bob’, 40)])	
D = dict(zip(keyslist, valueslist))	
D = dict.fromkeys([‘name’, ‘age’])	
D[‘name’]	Indexing by key
E[‘cto’][‘age’]	
‘age’ in D	Membership
D.keys()	Keys
D.values()	Values
D.items()	Items
D.copy()	Copy
D.clear()	Clear
D.update(D2)	Update (change existing and merge new values)
D.get(key, default)	Get a value, default if not there
D.pop(key, default)	Get a key, default if not there
D.setdefault(key, default)	Set the default for a key value that does not exist
D.popitem()	Fetch item and remove from dictionary
len(D)	Length
D[key] = 42	Add a key and value or change an existing value
del D[key]	Delete a key
list(D.keys())	Create a list of keys
D1.keys() & D2.keys()	Returns keys common to both lists
D.viewkeys(), D.viewvalues()	Dictionary views – did not work in 3.X
D = {x: x*2 for x in range(10)}	Dictionary comprehensions

```
>>> D={'name': 'Bob', 'age': 40}
>>> E={'name':'David', 'age':63, 'addr': '1834 Jacobs Ln'}

>>> D.keys()
['age', 'name']
>>> E.keys()
['age', 'name', 'addr']


>>> D.update(E)
>>> D
{'age': 63, 'name': 'David', 'addr': '1834 Jacobs Ln'}
>>> E
{'age': 63, 'name': 'David', 'addr': '1834 Jacobs Ln'}

>>> 'age' in D
True
>>> D.pop('age')
63
>>> D
{'name': 'David', 'addr': '1834 Jacobs Ln'}

>>> D.get('name')
'David'
>>>D[‘name’]
‘David’
>>> D
{'name': 'David', 'addr': '1834 Jacobs Ln'}
>>> E
{'age': 63, 'name': 'David', 'addr': '1834 Jacobs Ln'}

>>> D.values()
['David', '1834 Jacobs Ln']
>>> D.keys()
['name', 'addr']
>>> D['phone'] = '8502127014'
>>> D
{'phone': '8502127014', 'name': 'David', 'addr': '1834 Jacobs Ln'}
>>> D['name']='Philip'
>>> D
{'phone': '8502127014', 'name': 'Philip', 'addr': '1834 Jacobs Ln'}
>>> del D[‘phone’]        #like pop, except does not return the value
>>> D
{'name': 'David', 'addr': '1834 Jacobs Ln'}
```

```
>>> Ks = D.keys()
>>> Ks
['name', 'addr']
>>> Ks.sort()
>>> Ks
['addr', 'name']
>>> for k in Ks: print(k, D[k])                    #sorts the keys, not the values stored
...
('addr', '1834 Jacobs Ln')
('name', 'Philip')
>>>
```

Simulate sparse lists by using a dictionary:

```
>>> L = {12 : ‘Alabama’}
>>> L[12]
‘Alabama’

>>> L[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 10
```

Dictionaries can look like records:

```
>>> Employee = {‘ID’: 901, ‘jobs’: [‘developer’, ‘programmer’],
...                     ‘hiredate’: ‘06/01/2010’, ‘salary’: 50,000}
...
>>>
```

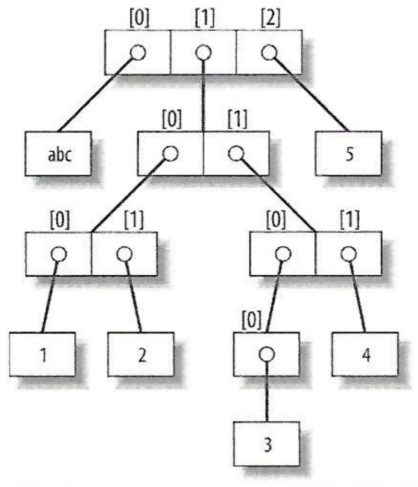
Sequence operations do not work on dictionaries.
Assigning to new indexes adds entries.
Keys need not be strings.

Other Topics

Objects:

- Lists, dictionaries, and tuples can hold any kind of object.
- Sets can contain any type of immutable object.
- Lists, dictionaries, and tuples can be arbitrarily nested.
- Lists, dictionaries, and sets can dynamically grow and shrink.

```
>>>L = [‘abc’, [(1,2), ([3], 4)], 5]
>>>L[1]
[(1,2), ([3], 4)]
>>>L[1][1]
([3], 4)
>>>L[1][1][0]
[3]
>>>L[1][1][0][0]
3
```



References vs Copies

```
>>>X = [1, 2, 3]
>>>L = [‘a’, X, ‘b’]
>>>D = {‘x’: X, ‘y’:2}
```

X has a reference to the list [1, 2, 3]
L references a list that references X as the second item in the list.
D references a key that references X as the value associated with the key.

Thus:

```
>>>L
[‘a’, [1, 2, 3], ‘b’]
>>>D
{‘x’: [1, 2, 3], ‘y’:2}
```

When an object is referenced, any change to the object will appear through any reference to the object. Thus:

```
>>>X[1] = ‘surprise’
```

Results in

```
>>>L
[‘a’, [1, ‘surprise’, 3]

>>>D
{‘x’: [1, ‘surprise’, 3], ‘y’: 2}
```

- If you truly want a copy, you can:
  Use a slice expression with empty limits, e.g., L[:] creates a copy
- Use the dictionary, set, or list copy method (X.copy()) to make a copy
- Remember that some built-in functions make copies (list(L), dict(D), set(S))
- The copy standard library module makes full copies when needed.

For example,

```
>>>L = [1, 2, 3], D = {‘a’: 1, ‘b’: 2}
>>>A = L[:]                    #creates a copy of L
>>>B = D.copy()                #creates a copy of D
```

Changes to A and B will not change L and D. And note:

```
>>>L = [‘a’, X[:], ‘b’]        #changes the result of our earlier sequence of statements by
#making a copy of X in L
```

Note:

Empty index slices and the copy method only make a high-level copy. They will not make copies of deeper, nested structures. To get a fully independent copy of a deeply nested data structure:

```
Import copy
X = copy.deepcopy(Y)
```

Comparisons, Equality, and Truth

Due to the nested structures that can be built in Python, determining whether two objects are “equal” requires looking at everything in the object.

We can think of equality in these terms: Two objects are equal if the first item in the first object is equal to the first item in the second object and the rest of the two objects are equal.

The “rest of the two objects are equal” if the first item in the rest of the first object is equal to the first item in the rest of the second object and the rest of the two objects are equal.

We can continue along this process until we get to the last item in both objects (if we haven’t already found two items that are not equal).

Python defines

- The == operator tests value equivalence. Python performs an equivalence test, comparing all nested objects recursively.
- The *is* operator tests object identity. Python tests whether the two are really the same object (i.e., live at the same address in memory).

Boolean Considerations

False is 0; True is 1
Numbers are False if 0, True otherwise
Objects are False if empty; True otherwise

Python has a special object: **None**. It is like NULL in other languages or in database environments.
None is False.

None is an object, so it is something, not nothing!

Gotchas

```
>>>L = [4, 5, 6]
>>>X = L * 4
>>>Y = [L] * 4

>>>X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>>Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

>>>L[1] = 0
>>>X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>>Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]

>>>L = [‘grail’]
>>>L.append(L)
>>>L
[‘grail’, …]
```

Python knows that L has been appended to itself. Instead of looping (perhaps infinitely), Python uses an ellipsis to indicate a *cyclic data structure*.

Let’s write some code!

Write a program that will build a list of five names that you enter using the keyboard. Print the list of names. Sort the list and print it again. Ask for a name to remove from the list, remove it, and print the list again. Swap the second and third names in the list and print the list again. Extra for experts: take the last name in the list, change it so that it is spelled backwards, and print the list again.

Write a program that will build a list of 5 dictionaries that have the following keys: name, age, salary. Print the list of dictionaries. Ask for a name and print the dictionary containing that name. Modify you code so that the contents of the dictionary for the name you input is printed in a nice format. Choose a dictionary in your list (use the name key), add a key to that dictionary that is named ‘job’ with the value ‘manager’, and print the list of dictionaries again. Extra for experts: use the name key to specify a dictionary to remove from the list.

Write a program that will read a CSV file named tdata (tdata.csv) and output the data in the following forms: raw form (the way it is read in), as a list of rows, as a list of dictionaries, and as a list of sets. Also, write the data to a text file named tdataout.txt.

For the dictionary, the keys are: 'state', 'yearinUS', and 'pop'

You can create a CSV file by opening Excel, and entering this data

Alabama	1819	4887871
Arkansas	1836	3013825
Georgia	1788	10519475

Save the file as tdata.csv in the same folder where your program is stored.

```
"""List practice program"""

#Initialize counter
namecount = 0
namelist = []

#Start interaction with user.
while namecount < 5:
   name_in = input('Please enter a name: ')
   namelist.append(name_in)
   namecount += 1


print('Original list:', namelist)

namelist.sort()
print('Sorted name list:', namelist)

name_in = input('Which name should be removed?')
namelist.remove(name_in)
print('Revised name list:', namelist)

namelist[1], namelist[2] = namelist[2], namelist[1]
print('Swapped names:', namelist)

lastname = list(namelist[-1])          #Make the last name a list of characters
lastname.reverse()                                             #Reverse the list of characters
reversename = ''
for c in lastname: reversename += c    #Make the list into a string
namelist[-1] = reversename             #Put the string in the list at the last location
print('List with reversed last name:', namelist)
```

```
"""Dictionary practice program"""


#Initialize counter
namecount = 0
dictlist, valueslist = [], []

#Start interaction with user.

while namecount < 5:
   values_in = input('Please enter a name, age, and salary, separated by commas: ')
   valueslist = list(values_in.split(','))
   dictionaryentry = dict(zip(['name', 'age', 'salary'],valueslist))
   dictlist.append(dictionaryentry)
   namecount += 1

print('Original list:', dictlist)
keyname = input('Which person do you want to view?')
for d in dictlist:
   if d['name'] == keyname:
      print(d)
      print('Name:',d['name'],'Age:',d['age'],'Salary:',d['salary'])
      break

keyname = input('Which person do you want to make manager?')
for d in dictlist:
   if d['name'] == keyname:
      d['job'] = 'manager'
      break
print(dictlist)

keyname = input('Which person do you want to remove?')
for d in dictlist:
   if d['name'] == keyname:
      target = d
      break

dictlist.remove(d)
print(dictlist)
```

```
"""
This program reads the data in tdata.csv, creates lists of different object types,
   prints the objects to the terminal, and writes the objects to tdataout.txt
"""

import csv

rowlist, dictlist, tuplelist, setlist = [], [], [], []

#Read the data from the file to be read
reader = csv.reader(open('tdata.csv', 'r'), delimiter=',')

#Process each row
for row in reader:
   rowlist.append(list(row))   #builds a list of records; each record is a list itself

print('Here is the data in one statement:', rowlist)

print('Here is the data in a loop:')
for r in rowlist: print(r)

print('Here is the data in dictionaries:')
for r in rowlist:
   D = dict(zip(['state', 'yearinUS', 'pop'], r))
   dictlist.append(D)                 #builds a list of records; each record is a dictionary
print(dictlist)

#  A good test: make the prior loop more general by putting column names in the first row of the file.

print('Here is the data in tuples:')
for r in rowlist:
   T = tuple(r)
   tuplelist.append(T)                #builds a list of records; each record is a tuple
print(tuplelist)

print('Here is the data in sets:')
for r in rowlist:
   S = set(r)                         #another way:  for i in l: S.add(i)
   setlist.append(S)                  #builds a list of records; each record is a set
print(setlist)
```

```
#output to text files must be strings, \n creates an end-of-line

outfile = open('tdataout.txt','w')
line = 'Data in: ' + str(rowlist) + '\n'
outfile.write(line)
line = 'Dictionaries: ' + str(dictlist) + '\n'
outfile.write(line)
line = 'Tuples: ' + str(tuplelist) + '\n'
outfile.write(line)
line = 'Sets: ' + str(setlist) + '\n'
outfile.write(line)
```