```
#help(str)
#help(set)
#help(list)
#help(tuple)
# Are all a good place to start for more information!
# I am not expecting students to memorize all object.method() calls. I do expect everyone to understand
# the syntax used in all statements!
#Python Shell – List, Tuple, String and Set Examples
#Lists are a collection of objects.
>>> a = [2,3,4,5]
>>> a
[2, 3, 4, 5]
>>> type(a)
<type 'list'>
>>> lst1 = ["Acer", "Dell", "Gateway", "Apple"]
>>> lst1
['Acer', 'Dell', 'Gateway', 'Apple']
>>> type(lst1)
<type 'list'>
>>> a[0]
2
>>> a[1]
3
>>> a[0:2]
[2, 3]
>>> a[2:0]
[]
>>> a[-1]
5
>>> a[-2]
4
>>> a[0:-2]
[2, 3]
>>> a[0:-3]
[2]
>>> type(a[0])
<type 'int'>
>>> type(a[0:2])
<type 'list'>
>>> lst1[2]
'Gateway'
>>> type(lst1[0])
```

```
<type 'str'>
>>> lst2 = []
>>> lst2
[]
>>> type(lst2)
<type 'list'>
>>> a[10]
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
IndexError: list index out of range
>>> lst2 = range(4,10)
>>> lst2
[4, 5, 6, 7, 8, 9]
>>> for i in range(4,10):
...         print i
...
4
5
6
7
8
9
>>> for j in lst2:
...         print j
...
4
5
6
7
8
9
>>> len(lst1)
4
>>> for i in range(0,len(lst1)):
...         print lst1[i]
...
Acer
Dell
Gateway
Apple
>>> max(lst2)
9
```

```
>>> min(a)
2
>>> min(lst2)
4
>>> a
[2, 3, 4, 5]
>>> sum(a)
14
>>> sum(lst1)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> lst3 = lst2 + a
>>> lst3
[4, 5, 6, 7, 8, 9, 2, 3, 4, 5]
>>> lst3 = lst3+lst3
>>> lst3
[4, 5, 6, 7, 8, 9, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 2, 3, 4, 5]
>>> lst3 = [2,3,4]
>>> lst3
[2, 3, 4]
>>> lst4 = lst3*4
>>> lst4
[2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4]
>>> lst4 = 4*lst3
>>> lst4
[2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4]
>>> lst1
['Acer', 'Dell', 'Gateway', 'Apple']
>>> 'Acer' in lst1
True
>>> 'Sony' in lst1
False
>>> 3 in lst4
True
>>> 123456 in lst4
False
>>> lst2
[4, 5, 6, 7, 8, 9]
>>> lst2.append(100)
>>> lst2
[4, 5, 6, 7, 8, 9, 100]
```

```
>>> lst2.extend([110, 120])
>>> lst2
[4, 5, 6, 7, 8, 9, 100, 110, 120]
>>> lst2.reverse()
>>> lst2
[120, 110, 100, 9, 8, 7, 6, 5, 4]
>>> lst2.index(100)
2
>>> lst2[2]
100
>>> lst2.append(4)
>>> lst2.index(4)
8
>>> lst2.pop()
4
>>> lst2.pop()
4
>>> lst2.pop()
5
>>> lst2.pop()
6
>>> lst2
[120, 110, 100, 9, 8, 7]
>>> lst2 = lst2[0:2] + [100] + lst2[2:len(lst2)]
>>> lst2
[120, 110, 100, 100, 9, 8, 7]
>>> lst2.count(100)
2
>>> lst2.insert(3,-999)
>>> lst2
[120, 110, 100, -999, 100, 9, 8, 7]
>>> lst = []
>>> lst = lst + [4]
>>> lst
[4]
>>> lst = lst + [678]
>>> lst
[4, 678]
>>> lst += [456789]
>>> lst
[4, 678, 456789]
>>> del lst[1]
```

```
>>> lst
[4, 456789]
>>> help(list)
>>> lst = [34,25,1234,-45,0,3,2,1]
>>> lst.sort()
>>> lst
[-45, 0, 1, 2, 3, 25, 34, 1234]
>>> lst.reverse()
>>> lst
[1234, 34, 25, 3, 2, 1, 0, -45]
>>> lst.remove(25)
>>> lst
[1234, 34, 3, 2, 1, 0, -45]
>>> lst.remove(1234)
>>> lst
[34, 3, 2, 1, 0, -45]
>>> lst = lst.remove(3)
>>> lst
>>> type(lst)
<type 'NoneType'>
>>> list_final_ex = [3,4,5,6,"Hello Python!"]
>>> list_final_ex
[3, 4, 5, 6, 'Hello Python!']
>>> list_final_ex = [[3,"Middle",2.9654]]+list_final_ex
>>> list_final_ex
[[3, 'Middle', 2.9654], 3, 4, 5, 6, 'Hello Python!']
>>> list_final_ex[0][1]
'Middle'
>>> list_final_ex[1]
3
>>> list_final_ex[2]
4
>>> list_final_ex[5][6:13]
'Python!'
>>> list_final_ex[5][12]
'!'
>>> (list_final_ex[0])[1]
'Middle'

#Tuples are very similar to lists. One big difference is that tuples cannot be modified after they are
#created.
```

```
>>> tup1 = 1,2,3
>>> tup1
(1, 2, 3)
>>> tup1[2]
3
>>> tup1[0:3]
(1, 2, 3)
>>> tup1[1:3]
(2, 3)
>>> tup1
(1, 2, 3)
>>> lst1 = [1,2,3]
>>> lst1
[1, 2, 3]
>>> lst1[1] = -4
>>> lst1
[1, -4, 3]
>>> tup1[2]=-4
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'tuple' object does not support item assignment
>>> def return_1_2_3():
...     return 1, 2, 3
...
>>> a, b, c = return_1_2_3()
>>> a
1
>>> b
2
>>> c
3
# How can a function return multiple values at the same time?
# Why can we have several variables on the left side of the assignment
# operator?
# To answer this, we need to explore tuples.
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> type(t)
<type 'tuple'>
# Tuples are like lists, in that we can access their elements by position.
>>> t[0]
```

```
1
>>> t[1]
2
>>> t[2]
3
# Unlike lists, tuples are immutable - we can't change a tuple after it's
# been initialized.

>>> t[0] = 4
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'tuple' object does not support item assignment
# Do + and * work operate on tuples like they do on lists and strings?
>>> t1 = (1, 2, 3)
>>> t2 = (3, 4, 5)
>>> t1 + t2
(1, 2, 3, 3, 4, 5)
>>> t1 * 2
(1, 2, 3, 1, 2, 3)
# How about del?
>>> del t[1]
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'tuple' object doesn't support item deletion
# Can we pass tuples to the len, min and max functions?
>>> len(t)
3
>>> min(t)
1
>>> max(t)
3
# What methods are owned by tuples?
>>> t.append(4)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AttributeError: 'tuple' object has no attribute 'append'
>>> t.count(2)
1
>>> t.remove(2)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AttributeError: 'tuple' object has no attribute 'remove'
```

```
>>> t.pop()
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AttributeError: 'tuple' object has no attribute 'pop'
# Another way of creating a tuple
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
# So, the statement
#
#    return 1, 2, 3
#
# is equivalent to:
#
#    t = 1, 2, 3
#    return t

# This is called "tuple packing"
# Python packs the sequence of values into a tuple, then returns the tuple
# How do we get values from a tuple?
>>> i = t[0]
>>> j = t[1]
>>> k = t[2]
>>> print i, j, k
1 2 3
# We can also do this:
>>> x, y, z = t
>>> print x, y, z
1 2 3
# This is called "tuple unpacking".
# So, when Python executes a, b, c = return_1_2_3(),
# Python packs the sequence of values in the return statement into a tuple,
# returns the tuple, then unpacks that tuple, binding the values to a, b and c.
# More unpacking experiments
>>> len(t)
3
>>> p, q = t
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
ValueError: too many values to unpack
>>> p, q, r, s = t
Traceback (most recent call last):
```

```
  File "<string>", line 1, in <fragment>
ValueError: need more than 3 values to unpack
# Clearly, the number of variables on the left side of = must be the
# same as the number of values in the tuple.
# How do we swap the values bound to two variables? That is, if
#   a = 5
#   b = 10
# how do we bind a to the value b is bound to, and b to the value a is bound to?
# This doesn't work:
#   a = b
#   b = a
>>> a = 5
>>> b = 10
>>> a = b
>>> b = a
>>> print a, b
10 10
# In most programming languages, you'd need to use a temporary variable;
# for example,
#   temp = a
#   a = b
#   b = temp
>>> a = 5
>>> b = 10
>>> temp = a
>>> print a, b, temp
5 10 5
>>> a = b
>>> print a, b, temp
10 10 5
>>> b = temp
>>> print a, b, temp
10 5 5
# In Python, you can do this:
>>> a = 5
>>> b = 10
>>> a, b = b, a  # swap the values bound to a and b
>>> print a, b
10 5
#Why does the above work to swap the variables?

# A string containing leading and trailing whitespace (spaces and
```

```
# newline characters):
>>> greeting = "    \n   Hello,    world!    \n"
>>> greeting
'    \n   Hello,    world!    \n'
# When print the string, there are 3 newlines in the output: the two
# \n's in the string, and the extra newline that print outputs.
>>> print greeting


  Hello,    world!


>>>

# Use , to prevent print from printing an extra newline.
>>> print greeting,

  Hello,    world!
>>>

# All string objects own several methods that provide often-required
# operations on strings.
# Strings are immutable, so none of the methods change the strings
# on which they are called. Several methods return a string (the
# result of performing the requested operation on the original string).
#
# Use Python's built-in help function to learn more:
>>> help(str)
# strip removes the leading and trailing whitespace, but doesn't remove
# the extra spaces between the words.
>>> s = greeting.strip()
>>> s
'Hello,    world!'
>>> print s
Hello,    world!
>>>
# strip takes an optional argument: a string containing the characters it
# should strip from the start and end of the string on which it is called.
# Whitespace isn't stripped, unless it is included in the string of
# characters to strip.
>>> s = "@$%   First @$ Second @%%$"
>>> s.strip("@")
'$%   First @$ Second @%%$'
>>> s.strip("$")
```

```
'@$%   First @$ Second @%%'
>>> s.strip("%")
'@$%   First @$ Second @%%$'
>>> s.strip("@$")
'%   First @$ Second @%%'
>>> s.strip("@%")
'$%   First @$ Second @%%$'
>>> s.strip("%$")
'@$%   First @$ Second @'
>>> s.strip("@%$")
'   First @$ Second '
>>> help(str)
>>> s = greeting.strip()
>>> s
>>> print s
>>>
>>> s = "@$%   First @$ Second @%%$"
>>> s.strip("@")
>>> s.strip("$")
>>> s.strip("%")
>>> s.strip("@$")
>>> s.strip("@%")
>>> s.strip("%$")
>>> s.strip("@%$")
>>>
>>> s = "123  First 98 Second  566  "
>>> s.strip(" 0123456789")
>>>
# Whitespace isn't stripped, unless it is included in the string of
# characters to strip.
# Example: remove leading and trailing spaces and digits
>>> s = "123  First 98 Second  566  "
>>> s.strip(" 0123456789")
'First 98 Second'
>>>
# split returns a list of the words in the string.
# By default, all whitespace around each word is removed, but any
# punctuation symbols (comma, semicolon, colon, etc.) are not removed.
>>> greeting
'   \n   Hello,    world!     \n'
>>> lst = greeting.split()
>>> lst
```

```
['Hello,', 'world!']
>>>
# Numbers in strings are considered to be words.
>>> s = "1 2 3"
>>> lst = s.split()
>>> lst
['1', '2', '3']
>>> lst[0]
'1'
>>> type(lst[0])
<type 'str'>
>>> lst[1]
'2'
>>> type(lst[1])
<type 'str'>
>>> type(lst[2])
<type 'str'>
>>> type(lst[2])
<type 'str'>
>>>
# Notice that the numbers in the list are strings.
# split does not convert numeric strings to ints or floats
# To do that, we need to call int or float.
>>> i = int(lst[2])
>>> i
3
>>> type(i)
<type 'int'>
>>>
# By default, split treats whitespace as the delimiter between words.
# We can change this by passing a delimiter string to split.
>>> url = 'http://www.sce.carleton.ca'
>>> url.split('.')
['http://www', 'sce', 'carleton', 'ca']
>>> url.split('//')
['http:', 'www.sce.carleton.ca']
>>>
# When we change the delimiter to something other than whitespace,
# whitespace is included in the word strings returned by split.
>>> s = "First. Second.Third.   Fourth.   "
>>> s.split('.')
['First', ' Second', 'Third', '   Fourth', '   ']
```

```
# Some other useful methods
>>> url.endswith('.ca')
True
>>> url.startswith('http://')
True
>>> s1 = 'abcdef'
>>> s2 = 'abc123'
>>> s3 = '123456'
>>> s1.isalpha()
True
>>> s2.isalpha()
False
>>> s3.isalpha()
False
>>> s1.isalnum()
True
>>> s2.isalnum()
True
>>> s3.isalnum()
True
>>> s1.isdigit()
False
>>> s2.isdigit()
False
>>> s3.isdigit()
True
>>> s1.islower()
True
>>> s2.islower()
True  # Why???
>>> help(str.islower)
Help on method_descriptor:
islower(...)
    S.islower() -> bool
    Return True if all cased characters in S are lowercase and there is
    at least one cased character in S, False otherwise.
>>> s3.islower()
False
>>> s1.isupper()
False
>>> s2.isupper()
```

```
False
>>> s3.isupper()
False

# Python's built-in set type implements the mathematical concept of a set (an
# unordered collection of unique elements).

>>> s = {7, 3, 1, 9, 3, 5}

>>> type(s)
<type 'set'>

>>> s
set([1, 3, 5, 7, 9]) # Notice that the duplicate 3 was not inserted

# Also notice that when the set is evaluated, its elements aren't necessarily
# listed in the same order that was used when the list was created.

# The {elem-1, elem-2, ..., elem-n} notation was introduced in Python 2.7,
# which won't work with older releases.
#
# Code written prior to Python 2.7 will likely do this:

>>> lst = [7, 3, 1, 9, 3, 5]
>>> s = set(lst)  # Create a set from the elements in list lst
>>> s
set([1, 3, 5, 7, 9])

# or this:

>>> s = set()  # Create an empty set

# Call the add method to insert elements into the set, one at a time.

>>> s.add(7)
>>> s.add(3)
>>> s.add(1)
>>> s.add(9)
>>> s.add(3)
>>> s.add(5)
>>> s
set([1, 3, 5, 7, 9])
```

```
# Can we create an empty set this way?

>>> s = {}
>>> type(s)
<type 'dict'>  # s is bound to a dictionary, not a set

# What types of values can be elements in sets? They must be hashable.
# You'll learn about hashing in your data structures course in Second year,
# but in this course, we can assume that just about any immutable value
# we're likely to use is hashable. This means that ints, floats and strings
# can be stored in sets, as well as tuples containing values of these types.

# Some of the operators and functions we use with lists, tuples and strings
# can be used with sets.

>>> 3 in s
True
>>> 8 in s
False

>>> for item in s:
...     item
...
9
3
1
5
7

>>> len(s)
5
>>> min(s)
1
>>> max(s)
9

# Sets are unordered (they don't record the position of the elements or
# the order in which elements were inserted).
# This means we can't index into a set:

>>> s[0]
```

```
>>> s[0]
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing

# Exercise - remove all duplicate elements from a list, storing the result
# in a new list

>>> lst = [2, 5, 9, 2, 9, 8, 3, 2]

# Build a new set from the items in the list (duplicate values are discarded)
>>> s = set(lst)
>>> s
set([2, 3, 5, 8, 9])

# Now, build a new list from the elements in the set
>>> lst2 = list(s)
>>> lst2
[8, 9, 2, 3, 5]

# Caveat - the order of the items in the new list is not the same as the
# original list. This isn't a problem if, for example, we start with a
# list containing items sorted in ascending order, and remember to sort the
# new list.

# Aside - we can't create a list from a set this way.
# Instead, it creates a list containing one element: the entire set.

>>> lst2 = [s]
>>> len(lst2)
1
>>> lst2
[set([8, 9, 2, 3, 5])]

# Some other set operators

# First, create a few sets.

>>> s1 = {3, 9, 4}
>>> s2 = s1.copy()  # s2 is a duplicate of s1
>>> s3 = {9, 3}
```

```
>>> s1
set([3, 4, 9])
>>> s2
set([3, 4, 9])
>>> s3
set([3, 9])

# == returns True if both sets contain the same elements

>>> s1 == s2
True
>>> s1 == s3
False

# != tests for set inequality

>>> s1 != s2
False
>>> s1 != s3
True

# s <= t returns True if s is a subset of t (every element in s is also in t)

>>> s3 <= s1
True
>>> s2 <= s1
True

# s < t returns True if s is a proper subset of t (s <= t and s != t)

>>> s3 < s1
True
>>> s2 < s1
False

# Similarily, s >= t determines if s is a superset of t, and s > t determines
# if s is a proper supeset of t

>>> s4 = {9, 2, 7, 3}
>>> s4
set([2, 3, 7, 9])
```

```
>>> s1
set([3, 4, 9])

# s | t returns a new set containing the union of s and t

>>> s1 | s4
set([2, 3, 4, 7, 9])

# s & t returns a new set containing the intersection of s and t

>>> s1 & s4
set([3, 9])

# s - t returns a new set containing the difference of s and t (elements that
# are in s but not in t)

>>> s4 - s1
set([2, 7])
>>> s1 - s4
set([4])

# s ^ t returns a new set containing the symmetric difference of s and t
# (elements that are in either s or t but not both)

>>> s1 ^ s4
set([2, 4, 7])
>>> s4 ^ s1
set([2, 4, 7])

# Both operands of the |, &, - and ^ operators must be sets.
#
# The union, intersection, difference and symmetric_difference methods are
# similar to |, &, - and ^, except they can be passed any iterable collection
# (e.g., a list or a tuple) as an argument.

# Sets are mutable. In addition to add, other methods modify sets:

>>> s5 = s4.copy()
>>> s6 = s4.copy()

# Remove a specified element from a set, if it is present
```

```
>>> s4
set([2, 3, 7, 9])

>>> s4.discard(3)
>>> s4
set([2, 7, 9])

>>> s4.discard(8)
>>> s4
set([2, 7, 9])

# The remove method is similar, but raises an error if the element
# isn't present.

>>> s5
set([2, 3, 7, 9])

>>> s5.remove(3)
>>> s5
set([2, 7, 9])

>>> s5.remove(8)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
KeyError: 8

# Call pop to remove and return an arbitrary element

>>> s6.pop()
9
>>> s6
set([2, 3, 7])

# Call clear to remove all elements

>>> s6.clear()
>>> s6
set()
```