Strings to Numeric value

- To perform mathematical calculations, strings first have to be converted into a numeric value using a function such as int() or float()

```
z = int(x) + int(y)    # z = 79   (Integer +)
```

Non Strings to Strings

- Non-string values can be converted into a string representation by using the str(), repr(), or format() function.

```
s = "The value of x is " + str(x)
s = "The value of x is " + repr(x)
s = "The value of x is " + format(x,"4d")
```

Non Strings to Strings

- Although str() and repr() both create strings, their output is usually slightly different
- str() produces the output that you get when you use the print statement, whereas repr() creates a string that you type into a program to exactly represent the value of an object

```
>>> x = 3.4
>>> str(x)
'3.4'
>>> repr(x)
'3.3999999999999999'
>>>
```

Strings

- The inexact representation of 3.4 in the previous example is not a bug in Python.
- It is an artifact of double-precision floating-point numbers, which by their design can not exactly represent base-10 decimals on the underlying computer hardware.
- The format() function is used to convert a value to a string with a specific formatting applied.

Strings

- The format() function is used to convert a value to a string with a specific formatting applied.

```
>>> format(x,"0.5f")
'3.40000'
>>>
```

Lists

- Lists are sequences of arbitrary objects. You create a list by enclosing values in square brackets, as follows:
- **names = ["Dave", "Mark", "Ann", "Phil"]**
- Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:
- a = names[2] # Returns the third item of the list, "Ann"
- names[0] = "Jeff" # Changes the first item to "Jeff"

Lists

- To append new items to the end of a list, use the append() method:
- **names.append("Paula")**
- To insert an item into the middle of a list, use the insert() method:
- **names.insert(2, "Thomas")**

Lists

- You can extract or reassign a portion of a list by using the slicing operator:

```
b = names[0:2]                          # Returns [ "Jeff", "Mark" ]
c = names[2:]                           # Returns [ "Thomas", "Ann", "Phil", "Paula" ]
names[1] = 'Jeff'                       # Replace the 2nd item in names with 'Jeff'
names[0:2] = ['Dave','Mark','Jeff'] # Replace the first two items of
                                        # the list with the list on the right.
```

Lists

- Use the plus (+) operator to concatenate lists:
- a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]

- An empty list is created in one of two ways:
- names = [] # An empty list
- names = list() # An empty list

Lists

- Lists can contain any kind of Python object, including other lists, as in the following example:
- **a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]**
- Items contained in nested lists are accessed by applying more than one indexing operation, as follows:

```
a[1]              # Returns "Dave"
a[3][2]           # Returns 9
a[3][3][1]        # Returns 101
```

Lists

- Lists can contain any kind of Python object, including other lists, as in the following example:
- **a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]**
- Items contained in nested lists are accessed by applying more than one indexing operation, as follows:

```
a[1]              # Returns "Dave"
a[3][2]           # Returns 9
a[3][3][1]        # Returns 101
```

Advanced List Features

Listing 1.2 **Advanced List Features**

```
import sys                          # Load the sys module
if len(sys.argv) != 2               # Check number of command line arguments :
     print "Please supply a filename"
     raise SystemExit(1)
f = open(sys.argv[1])               # Filename on the command line
lines = f.readlines()               # Read all lines into a list
f.close()

# Convert all of the input values from strings to floats
fvalues = [float(line) for line in lines]

# Print min and max values
print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

Tuples

- To create simple data structures, you can pack a collection of values together into a single object using a tuple.
- You create a tuple by enclosing a group of values in parentheses like this:
- stock = ('GOOG', 100, 490.10)
- address = ('www.python.org', 80)
- person = (first_name, last_name, phone)

Tuples

- Python often recognizes that a tuple is intended even if the parentheses are missing:
- stock = 'GOOG', 100, 490.10
- address = 'www.python.org',80
- person = first_name, last_name, phone

Tuples

- Python often recognizes that a tuple is intended even if the parentheses are missing:
- stock = 'GOOG', 100, 490.10
- address = 'www.python.org',80
- person = first_name, last_name, phone

Tuples

- The values in a tuple can be extracted by numerical index just like a list.
- However, it is more common to unpack tuples into a set of variables like this:

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

Tuples

- Although tuples support most of the same operations as lists (such as indexing, slicing, and concatenation), **the contents of a tuple cannot be modified after creation** (that is, you cannot replace, delete, or append new elements to an existing tuple).
- This reflects the fact that a tuple is best viewed as a single object consisting of several parts, not as a collection of distinct objects to which you might insert or remove items.

Tuples

- Some programmers are inclined to ignore tuples altogether and simply use lists because they seem to be more flexible.
- Although this works, it wastes memory if your program is going to create a large number of small lists (that is, each containing fewer than a dozen items).
- This is because lists slightly overallocate memory to optimize the performance of operations that add new items.
- Because tuples are immutable, they use a more compact representation where there is no extra space.

Sets

- A set is used to contain an unordered collection of objects.
- To create a set, use the set() function and supply a sequence of items such as follows:

```
s = set([3,5,9,10])          # Create a set of numbers
t = set("Hello")             # Create a set of unique characters
```

- Unlike lists and tuples, sets are unordered and cannot be indexed by numbers.
- Moreover, the elements of a set are never duplicated.

Sets

- eg: if you inspect the value of t from the preceding code, you get the following:

```
>>> t
set(['H', 'e', 'l', 'o'])
```

- Notice that only one 'l' appears.
- Sets support a standard collection of operations, including union, intersection, difference, and symmetric difference.

Sets

```
a = t | s          # Union of t and s
b = t & s          # Intersection of t and s
c = t - s          # Set difference (items in t, but not in s)
d = t ^ s          # Symmetric difference (items in t or s, but not both)
```

New items can be added to a set using `add()` or `update()`:

```
t.add('x')            # Add a single item
s.update([10,37,42])  # Adds multiple items to s
```

An item can be removed using `remove()`:

```
t.remove('H')
```