Dictionaries

- A dictionary is an associative array or hash table that contains objects indexed by keys.
- You create a dictionary by enclosing the values in curly braces ({ }), like this:

```
stock = {
        "name"   : "GOOG",
        "shares" : 100,
        "price"  : 490.10
      }
```

Dictionaries

- To access members of a dictionary, use the key-indexing operator as follows:

```
name  = stock["name"]
value = stock["shares"] * shares["price"]
```

Inserting or modifying objects works like this:

```
stock["shares"] = 75
stock["date"]   = "June 7, 2007"
```

Dictionaries

- Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples.
- Some objects, including lists and dictionaries, cannot be used as keys because their contents can change.
- A dictionary is a useful way to define an object that consists of named fields as shown previously.
- However, dictionaries are also used as a container for performing fast lookups on unordered data.

Dictionaries

- Eg:

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM"  : 91.50,
    "MSFT" : 52.13
}
```

An empty dictionary is created in one of two ways:

```
prices = {}      # An empty dict
prices = dict()  # An empty dict
```

Dictionaries

- Eg:

To access members of a dictionary, use the key-indexing operator as follows:

```
name  = stock["name"]
value = stock["shares"] * shares["price"]
```

Inserting or modifying objects works like this:

```
stock["shares"] = 75
stock["date"]   = "June 7, 2007"
```

Dictionaries

- A dictionary is a useful way to define an object that consists of named fields as shown previously.
- However, dictionaries are also used as a container for performing fast lookups on unordered data.

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM"  : 91.50,
    "MSFT" : 52.13
}
```

Dictionaries

- An empty dictionary is created in one of two ways:

```
prices = {}      # An empty dict
prices = dict()  # An empty dict
```

Dictionaries

- Dictionary membership is tested with the in operator, as in the following example:

```
if "SCOX" in prices:
    p = prices["SCOX"]
else:
    p = 0.0
```

- This particular sequ compactly as follows:
- ***p = prices.get("SCOX",0.0)***

Dictionaries

To obtain a list of dictionary keys, convert a dictionary to a list:

```
syms = list(prices)              # syms = ["AAPL", "MSFT", "IBM", "GOOG"]
```

Use the `del` statement to remove an element of a dictionary:

```
del prices["MSFT"]
```

Iteration and Looping

- The most widely used looping construct is the for statement, which is used to iterate over a collection of items.
- The most common form of iteration is to simply loop over all the members of a sequence such as a string, list, or tuple
- ***for n in [1,2,3,4,5,6,7,8,9]:***
  print "2 to the %d power is %d" % (n, 2**n)

Iteration and Looping

- Because looping over ranges of integers is quite common, the following shortcut is often used for that purpose:
- ***for n in range(1,10):***
  print "2 to the %d power is %d" % (n, 2**n)

Iteration and Looping

- The range(i,j [,stride]) function creates an object that represents a range of integers with values i to j-1.
- If the starting value is omitted, it's taken to be zero. An optional stride can also be given as a third argument.

```
a = range(5)          # a = 0,1,2,3,4
b = range(1,8)        # b = 1,2,3,4,5,6,7
c = range(0,14,3)     # c = 0,3,6,9,12
d = range(8,1,-1)     # d = 8,7,6,5,4,3,2
```

Iteration and Looping

- The for statement is not limited to sequences of integers and can be used to iterate over many kinds of objects including strings, lists, dictionaries, and files.

```
a = "Hello World"
# Print out the individual characters in a
for c in a:
    print c

b = ["Dave","Mark","Ann","Phil"]
# Print out the members of a list
for name in b:
    print name
```

Iteration and Looping

- The for statement is not limited to sequences of integers and can be used to iterate over many kinds of objects including strings, lists, dictionaries, and files.

```
c = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# Print out all of the members of a dictionary
for key in c:
    print key, c[key]

# Print all of the lines in a file
f = open("foo.txt")

for line in f:
    print line,
```

Functions

- Use the **def** statement to create a function

```
def remainder(a,b):
    q = a // b        # // is truncating division.
    r = a - q*b
    return r
```

- To invoke a function, simply use the name of the function followed by its arguments enclosed in parentheses, such as ***result = remainder(37,15)***

Functions

- You can use a tuple to return multiple values from a function

```
def divide(a,b):
    q = a // b          # If a and b are integers, q is integer
    r = a - q*b
    return (q,r)
```

- When returning multiple values in a tuple, you can easily unpack the result into separate variables like this:

```
quotient, remainder = divide(1456,33)
```

Functions

- To assign a default value to a function parameter, use assignment:

```
def connect(hostname,port,timeout=300):
    # Function body
```

- When default values are given in a function definition, they can be omitted from subsequent function calls. When omitted, the argument will simply take on the default value.

```
connect('www.python.org', 80)
```

Functions

- You also can invoke functions by using keyword arguments and supplying the arguments in arbitrary order.
- However, this requires you to know the names of the arguments in the function definition

```
connect(port=80,hostname="www.python.org")
```

Functions

- When variables are created or assigned inside a function, their scope is local.
- That is, the variable is only defined inside the body of the function and is destroyed when the function returns.
- To modify the value of a global variable from inside a function, use the global statement as follows:

```
count = 0
...
def foo():
    global count
    count += 1              # Changes the global variable count
```

Generators

- Instead of returning a single value, a function can generate an entire sequence of results if it uses the yield statement.

```
def countdown(n):
    print "Counting down!"
    while n > 0:
        yield n       # Generate a value (n)
        n -= 1
```

Generators

- Any function that uses yield is known as a generator. Calling a generator function creates an object that produces a sequence of results through successive calls to a next() method (or __next__() in Python 3)

```
>>> c = countdown(5)
>>> c.next()
Counting down!
5
>>> c.next()
4
>>> c.next()
3
>>>
```

Generators

- The next() call makes a generator function run until it reaches the next yield statement.
- At this point, the value passed to yield is returned by next(), and the function suspends execution.
- The function resumes execution on the statement following yield when next() is called again.
- This process continues until the function returns.

Generators

- Normally you would not manually call next() as shown. Instead, you hook it up to a for loop like this:

```
>>> for i in countdown(5):
...     print i,
Counting down!
5 4 3 2 1
>>>
```

Generators

- Here's a generator that looks for a specific substring in a sequence of lines:

```
def grep(lines, searchtext):
    for line in lines:
        if searchtext in line: yield line
```