Python Programming

Slides Courtesy: C. David Sherrill
School of Chemistry and Biochemistry
Georgia Institute of Technology

List of Chapters

Resources

- These notes are based on information from several sources:
- “Learning Python,” 2nd edition, Mark Lutz and David Ascher (O'Reilly, Sebastopol, CA, 2004) (Thorough. Hard to get into as a quick read)
- “Dive Into Python,” Mark Pilgrim (http://diveintopython.org, 2004)
- “How to Think Like a Computer Scientist: Learning with Python,” 2nd edition, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (http://openbookproject.net//thinkCSpy/)
- “Programming in Python 3: A Complete Introduction to the Python Language,” Mark Summerfeld (Addison-Wesley, Boston, 2009)
- http://www.python.org

Why Python?

- High-level language, can do a lot with relatively little code
- Supposedly easier to learn than its main competitor, Perl
- Fairly popular among high-level languages
- Robust support for object-oriented programming
- Support for integration with other languages
- ✓ Python was developed by **Guido van Rossum** in the **late eighties and early nineties** at the **National Research Institute for Mathematics and Computer Science** in the **Netherlands.**

Chapter 1: Very Basic Stuff

- Running python programs
- Variables
- Printing
- Operators
- Input
- Comments
- Scope

Very Basic Stuff

- You can run python programs from files, just like perl or shell scripts, by typing

python program.py

- at the command line. The file can contain just the python commands.
- Or, one can invoke the program directly by typing the name of the file, “program.py”, if it has as a first line something like “#!/usr/bin/python” (like a shell script... works as long as the file has execute permissions set)

chmod u+x program.py

Hello, world!

- Let's get started! Here's an example of a python program run as a script:

  #!/usr/bin/python

  print “Hello, world”

- If this is saved to a file hello.py, then set execute permissions on this file (**chmod u+x hello.py** in Unix/Linux), and run it with “**./hello.py**”
- When run, this program prints

  Hello, world

More about printing

```
>>> x=2
>>> print 'hello:', x, x**2, x**3
hello: 4 16 64
```

- \t is a tab character
  ```
  >>> print "2**2 =","\t",2**2
  2**2 =  4
  ```
- Ending a print statement with a comma suppresses the newline, so the next print statement continues on the same line

PERL (Practical Extraction and Report Language) script in python

- Example basics.py

```
#!/usr/bin/perl
print 1+3
pi = 3.1415926
print pi
message = "Hello, world"
print message
```

Output:

```
4
3.1415926
Hello, world
```

Variable types

- In the previous example, “pi” and “message” are variables, but one is a floating point number, and the other is a string. Notice we didn't declare the types in our example.
- Python has decided types for the variables
- Actually, “variables” in python are really ***object references*****.** The reason we don't need to declare types is that a reference might point to a different type later.

  references.py:
  x=42
  y=”hello”
  print x,y # prints 42 hello

y=43
  print x,y # prints 42 43

Variable types

- Example types.py:

```
pi = 3.1415926
message = "Hello, world"
i = 2+2

print type(pi)
print type(message)
print type(i)
```

Output:

```
<type 'float'>
<type 'str'>
<type 'int'>
```

Variable names

- Can contain **letters, numbers, and underscores**

*Must **begin** with a **letter**

- Cannot be one of the reserved Python keywords: **and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield**

More on variable names

- Names starting with one underscore (_V) are not imported from the module import * statement
- Names starting and ending with 2 underscores are special, system-defined names (e.g., __V__)
- Names beginning with 2 underscores (but without trailing underscores) are local to a class (__V)
- A single underscore (_) by itself denotes the result of the last expression

Operators

- + addition
- - subtraction
- / division
- ** exponentiation
- % modulus (remainder after division)
- Comparison operators in Chapter 2

Operators

- Example operators.py
  print 2*2
  print 23**
  print 10%3
  print 1.0/2.0
  print 1/2

  Output:
  4
  8
  1
  0.5
  0
- Note the difference between floating point division and integer division in the last two lines

+= but not ++

- Python has incorporated operators like **+=**, but **++ (or --)** do not work in Python

Type conversion

- int(), float(), str(), and bool() convert to integer, floating point, string, and boolean (True or False) types, respectively
- Example typeconv.py:

```
print 1.0/2.0
print 1/2
print float(1)/float(2)
print int(3.1415926)
print str(3.1415926)
print bool(1)
print bool(0)
```

- Output:

```
0.5
0
0.5
3
3.1415926
True
False
```

Operators acting on strings

```
>>> "Amrita!"*3
Amrita!Amrita!Amrita!
```

- ```
  >>> "hello " + "world!"
  'hello world!'
  ```
  ```

Input from keyboard

- Example input.py

```
i = raw_input("Enter a math expression: ")
print i
j = input("Enter the same expression: ")
print j
```

Output:

```
localhost(workshop)% ./input.py
Enter a mathematical expression: 3+2
3+2
Enter the same expression: 3+2
5
```

Comments

- **Anything after a # symbol is treated as a comment**
- This is just like Perl

Chapter 2: Conditionals

- True and False booleans
- Comparison and Logical Operators
- if, elif, and else statements

Booleans: True and False

- >>> **type (True)**
  <type 'bool'>
  >>> **type (true)**
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'true' is not defined
  >>> **2+2==5**
  False
- Note: True and False are of type bool. *The capitalization is required for the booleans!*

Boolean expressions

- A boolean expression can be evaluated as True or False. An expression evaluates to False if it is...
  the constant False, the object None, an empty sequence or collection, or a numerical item of value 0
- Everything else is considered True

Comparison operators

- == : is equal to?
- != : not equal to
- > : greater than
- < : less than
- >= : greater than or equal to
- <= : less than or equal to

More on comparisons

- Can “chain” comparisons:
  >>> **a = 42**
  >>> **0 <= a <= 99**
  True

Logical operators

- and, or, not

```
>>> 2+2==5 or 1+1==2
True
>>> 2+2==5 and 1+1==2
False
>>> not(2+2==5) and 1+1==2
True
```

- **Note: We do NOT use &&, ||, !, as in C!**

If statements

- Example ifelse.py

```
if (1+1==2):
    print "1+1==2"
    print "I always thought so!"
else:
    print "My understanding of math must be faulty!"
```

- Simple one-line if:
  if (1+1==2): print “I can add!”

elif statement

- Equivalent of “else if” in C
- Example elif.py:

```
x = 3
if (x == 1):
    print "one"
elif (x == 2):
    print "two"
else:
    print "many"
```

Chapter 3: Functions

- Defining functions
- Return values
- Local variables
- Built-in functions
- Functions of functions
- Passing lists, dictionaries, and keywords to functions

Functions

- Define them in the file above the point they're used
- Body of the function should be indented consistently (4 spaces is typical in Python)
- Example: square.py

```
def square(n):
    return n*n

print "The square of 3 is ",
print square(3)
```

Output:
The square of 3 is 9

The def statement

- The **def statement is *excecuted*** (that's why functions have to be defined before they're used)
- **def creates an object and assigns a name to reference it;** the function could be assigned another name, function names can be stored in a list, etc.

More about functions

- Arguments are optional. **Multiple arguments are separated by commas.**
- If there's no return statement, then “None” is returned. Return values can be simple types or tuples. Return values may be ignored by the caller.
- Functions are **“typeless.”** Can call with arguments of any type, so long as the operations in the function can be applied to the arguments.

Function variables are local

- Variables declared in a function do not exist outside that function
- Example square2.py

```
def square(n):
    m = n*n
    return m

print "The square of 3 is ",
print square(3)
print m
```

Output:

```
  File "./square2.py", line 9, in <module>
    print m
NameError: name 'm' is not defined
```

Scope

- Variables assigned within a function are l**ocal to that function call**

Variables assigned at the top of a module are global to that module; **there's only “global” within a module**

- Within a function, Python will try to match a variable name to one assigned locally within the function; if that fails, it will try within enclosing function-defining (def) statements (if appropriate);
- if that fails, it will try to resolve the name in the global scope (but the variable must be declared global for the function to be able to change it).
- If none of these match, Python will look through the list of built-in names

Scope example

- scope.py

```
a = 5            # global

def func(b):
  c = a + b
  return c

print func(4)        # gives 4+5=9
print c               # not defined
```

Scope example

- scope.py

```
a = 5            # global

def func(b):
  global c
  c = a + b
  return c

print func(4)      # gives 4+5=9
print c            # now it's defined (9)
```

By value / by reference

- Everything in Python is a reference.
- However, note also that immutable objects are not changeable --- so changes to **immutable** objects within a function only change what object the name points to (and do not affect the caller, unless it's a global variable)
- For immutable objects (e.g., integers, strings, tuples), **Python acts like C's pass by value**
- For **mutable objects (e.g., lists), Python acts like C's pass by pointer;** in-place changes to mutable objects can affect the caller

Example

- passbyref.py

```
def f1(x,y):
    x = x * 1
    y = y * 2
    print x, y                    # 0 [1, 2, 1, 2]

def f2(x,y):
    x = x * 1
    y[0] = y[0] * 2
    print x, y                    # 0 [2, 2]

a = 0
b = [1,2]
f1(a,b)
print a, b                        # 0 [1, 2]
f2(a,b)
print a, b                        # 0 [2, 2]
```

Multiple return values

- Can return multiple values by packaging them into a tuple

```
def onetwothree(x):
  return x*1, x*2, x*3

print onetwothree(3)
```

3, 6, 9

print onetwothree(3.2)

(3.2, 6.4, 9.600000000000001)

>>> print onetwothree("hello")

Built-in Functions

- Several useful built-in functions. Example math.py

  print pow(2,3)
  print abs(-14)
  print max(1,-5,3,0)

  Output:
  8
  14
  3

Functions of Functions

- Example funcfunc.py

```
def iseven(x,f):
    if (f(x) == f(-x)):
        return True
    else:
        return False

def square(n):
    return(n*n)

def cube(n):
    return(n*n*n)

print iseven(2,square)
print iseven(2,cube)
```

- Output:
  True
  False

Default arguments

- Like C or Java, can define a function to supply a default value for an argument if one isn't specified

```
def print_error(lineno, message=”error”):
    print “%s at line %d” % (message, lineno)

print_error(42)
```

error at line 42

Functions without return values

- All functions in Python return something. If a return statement is not given, then by default, Python returns None
- Beware of assigning a variable to the result of a function which returns None. For example, the list append function changes the list but does not return a value:

```
a = [0, 1, 2]
b = a.append(3)
print b
None
```

Chapter 4: Iteration

- while loops
- for loops
- range function
- Flow control within loops: break, continue, pass, and the “loop else”

while

- Example while.py

```
i=1
while i < 4:
    print i
    i += 1
```

Output:
1
2
3

for

- Example for.py

  ```
  for i in range(3):
      print i,

  output:
    0, 1, 2
  ```

- range(n) returns a list of integers from 0 to n-1.
  range(0,10,2) returns a list 0, 2, 4, 6, 8

Flow control within loops

- General structure of a loop:

```
while <statement> (or for <item> in <object>):
    <statements within loop>
    if <test1>: break       # exit loop now
    if <test2>: continue  # go to top of loop now
    if <test3>: pass        # does nothing!
else:
    <other statements> # if exited loop without
                                # hitting a break
```

Using the “**loop else**”

- An else statement after a loop is useful for taking care of a case where an item isn't found in a list. Example: search_items.py:

```
for i in range(3):
    if i == 4:
        print "I found 4!"
        break
    else:
        print "Don't care about",i
else:
    print "I searched but never found 4!"
```

for ... in

- Used with collection data types (see Chapter 6) which can be iterated through (“iterables”):

```
for name in [“Mutasem”, “Micah”, “Ryan”]:
    if name[0] == “M”:
        print name, “starts with an M”
    else:
        print name, “doesn't start with M”
```

- More about lists and strings later on

Parallel traversals

- If we want to go through 2 lists (more later) in parallel, can use zip:

```
A = [1, 2, 3]
B = [4, 5, 6]
for (a,b) in zip(A,B):
  print a, “*”, b, “=”, a*b
```

output:

```
1 * 4 = 4
2 * 5 = 10
3 * 6  = 18
```

Chapter 5: Strings

- String basics
- Escape sequences
- Slices
- Block quotes
- Formatting
- String methods

String basics

- Strings can be delimited by single or double quotes
- Python uses Unicode, so strings are not limited to ASCII characters
- An empty string is denoted by having nothing between string delimiters (e.g., '')
- Can access elements of strings with [], with indexing starting from zero:
  >>> “snakes”[3]
  'k'
- Note: can't go other way --- can't set “snakes”[3] = 'p' to change a string; strings are *immutable*
- a[-1] gets the *last* element of string a (negative indices work through the string backwards from the end)
- Strings like a = r'c:\home\temp.dat' (starting with an r character before delimiters) are “raw” strings (interpret literally)

More string basics

- Type conversion:
  >>> int(“42”)
  42
  >>> str(20.4)
  '20.4'
- Compare strings with the is-equal operator, == (like in C and C++):
  >>> a = “hello”
  >>> b = “hello”
  >>> a == b
  True

>>>location = “Coimbatore “ + “Trichy”
>>>location

'CoimbatoreTrichy'

Escape sequences

- Escape Meaning
  \\ \
  \' '
  \” “
  \
  \n newline
  \t tab

  \x Hex digits value hh
  \0 Null byte (unlike C, doesn't end string)

Block quotes

- Multi-line strings use triple-quotes:
  >>> lincoln = “””Four score and seven years
  ... ago our fathers brought forth on this
  ... continent, a new nation, conceived in
  ... Liberty, and dedicated to the proposition
  ... that all men are created equal.”””

String formatting

- Formatting syntax:
  format % object-to-format

  >>> greeting = “Hello”
  >>> “%s. Welcome to python.” % greeting
  'Hello. Welcome to python.'
- Note: formatting creates a new string (because strings are immutable)
- The advanced printf-style formatting from C works. Can format multiple variables into one string by collecting them in a tuple (comma-separated list delimited by parentheses) after the % character:

  >>> “The grade for %s is %4.1f” % (“Tom”, 76.051)
  'The grade for Tom is 76.1'

- String formats can refer to dictionary keys (later):
  >>> “%(name)s got a %(grade)d” % {“name”:”Bob”, “grade”:82.5}
  'Bob got a 82'

Stepping through a string

- A string is treated as a collection of characters, and so it has some properties in common with other collections like lists and tuples (see below).
- >>> for c in “snakes”: print c,
  ...
  s n a k e s
- >>> 'a' in “snakes”
  True

Slices

- Slice: get a substring from position i to j-1:
  >>> “snakes”[1:3]
  'na'
- Both arguments of a slice are optional; a[:] will just provide a copy of string a

String methods

- Strings are classes with many built-in methods. Those methods that create new strings need to be assigned (since strings are immutable, they cannot be changed in-place).
- S.capitalize()
- S.center(width)
- S.count(substring [, start-idx [, end-idx]])
- S.find(substring [, start [, end]]))
- S.isalpha(), S.isdigit(), S.islower(), S.isspace(), S.isupper()
- S.join(sequence)
- And many more!

replace method

- Doesn't really replace (strings are immutable) but makes a new string with the replacement performed:
  >>> a = “abcdefg”
  >>> b = a.replace('c', 'C')
  >>> b
  abCdefg
  >>> a
  abcdefg

More method examples

- methods.py:

```
a = “He found it boring and he left”
loc = a.find(“boring”)
a = a[:loc] + “fun”
print a

b = ' and '.join([“cars”, “trucks”, “motorcycles”])
print b

c = b.split()
print c

d = b.split(“ and “)
print d
```

Output:

```
He found it fun
cars and trucks and motorcycles
['cars', 'and', 'trucks', 'and', 'motorcycles']
['cars', 'trucks', 'motorcycles']
```

Regular Expressions

- Regular expressions are a way to do pattern-matching.
- The basic concept (and most of the syntax of the actual regular expression) is the same in Java or Perl

Regular Expression Syntax

- Common regular expression syntax:
  . Matches any char but newline (by default)
  ^ Matches the start of a string
  $ Matches the end of a string

  * Any number of what comes before this
  + One or more of what comes before this
  | Or
  \w Any alphanumeric character
  \d Any digit
  \s Any whitespace character

  (Note: \W matches NON-alphanumeric, \D NON digits, etc)

  [aeiou] matches any of a, e, i, o, u
  junk Matches the string 'junk'

Reading from files

```
infile = open("test.txt", 'r')
lines = infile.readlines()
infile.close()

lines
```

Simple Regexp Example

- In the previous example, we open a file and read in all the lines (see upcoming chapter), and we replace all PSI3 instances with PSI4
- The regular expression module needs to be imported
- We need to “compile” the regular expression with re.compile(). The “r” character here and below means treat this as a “raw” string (saves us from having to escape backslash characters with another backslash)
- re.IGNORECASE (re.I) is an optional argument. Another would be re.DOTALL (re.S; dot would match newlines; by default it doesn't). Another is re.MULTILINE (re.M), which makes ^ and $ match after and before each line break in a string

More about Regexps

- The re.compile() step is optional (more efficient if doing a lot of regexps
- Can do re.search(regex, subject) or re.match(regex, subject) as alternative syntax
- re.match() only looks for a match at the beginning of a line; does not need to match the whole string, just the beginning
- re.search() attempts to match throughout the string until it finds a match
- re.findall(regex, subject) returns an array of all non-overlapping matches; alternatively, can do
  for m in re.finditer(regex, subject)
- Match(), search(), finditer(), and findall() do not support the optional third argument of regex matching flags; can start regex with (?i), (?s), (?m), etc, instead

re.split()

- re.split(regex, subject) returns an array of strings. The strings are all the parts of the subject besides the parts that match. If two matches are adjacent in the subject, then split() will include an empty string.
- Example:
  line = “Upgrade the PSI3 program to PSI4. PSI3 was an excellent program.”
  matchstr = re.split(“PSI3”, line)
  for i in matchstr:
  print i
  >>> Upgrade the
  >>> program to PSI4.
  >>> was an excellent program.

Match Object Funtions

- Search() and match() return a MatchObject. This object has some useful functions:
  group(): return the matched string
  start(): starting position of the match
  end(): ending position of the match
  span(): tuple containing the (start,end) positions of the match

Match Object Example

```
line = "Now we are upgrading the PSI3 program"
matchstr = re.search("PSI3", line)
print "Match starts at character ", matchstr.start()
print "Match ends at character ", matchstr.end()
print "Before match: ", line[0:matchstr.start()]
print "After match:", line[matchstr.end():]
```

Capture Groups

- Parentheses in a regular expression denote “capture groups” which can be accessed by number or by name to get the matching (captured) text
- We can name the capture groups with (?P<name>)
- We can also take advantage of triple-quoted strings (which can span multiple lines) to define the regular expression (which can include comments) if we use the re.VERBOSE option

Capture Groups Example

```
infile = open("test.txt", 'r')

lines = infile.readlines()

infile.close()

# triple quoted string can span multiple lines

restring = """[ \t]*          # optional whitespace at beginning

           (?P<key>\w+)    # name the matching word 'key'

           [ \t]*=[ \t]*   # equals sign w/ optional whitespace

           (?P<value>.+)   # some non-whitespace after = is 'value'

           """

matchstr = re.compile(restring, re.IGNORECASE | re.VERBOSE)

for line in lines:

  for match in matchstr.finditer(line):

    print "key =", match.group("key"),

    print ", value =", match.group("value")
```

Chapter 6: Collection Data Types

- Tuples
- Lists
- Dictionaries

Tuples

- Tuples are a collection of data items. They may be of different types. Tuples are *immutable* like strings. Lists are like tuples but are mutable.

  >>>“Tony”, “Pat”, “Stewart”
  ('Tony', 'Pat', 'Stewart')
  Python uses () to denote tuples; we could also use (), but if we have only one item, we need to use a comma to indicate it's a tuple: (“Tony”,).
- An empty tuple is denoted by ()
- Need to enclose tuple in () if we want to pass it all together as one argument to a function

Lists

- Like tuples, but mutable, and designated by square brackets instead of parentheses:
  >>> [1, 3, 5, 7, 11]
  [1, 3, 5, 7, 11]
  >>> [0, 1, 'boom']
  [0, 1, 'boom']
- An empty list is []
- Append an item:
  >>> x = [1, 2, 3]
  >>> x.append(“done”)
  >>> print x
  [1, 2, 3, 'done']

Lists and Tuples Contain Object References

- Lists and tuples contain *object references*. Since lists and tuples are also objects, they can be nested

```
>>> a=[0,1,2]
>>> b=[a,3,4]
>>> print b
[[0, 1, 2], 3, 4]
>>> print b[0][1]
1
>>> print b[1][0]
... TypeError: 'int' object is unsubscriptable
```

List example

- list-example.py:

```
x=[1,3,5,7,11]
print x
print "x[2]=",x[2]
x[2] = 0
print "Replace 5 with 0, x = ", x
x.append(13)
print "After append, x = ", x
x.remove(1) # removes the 1, not the item at position 1!!
print "After remove item 1, x = ", x
x.insert(1,42)
print "Insert 42 at item 1, x = ", x
```

Output:

```
[1, 3, 5, 7, 11]
x[2]= 5
Replace 5 with 0, x =  [1, 3, 0, 7, 11]
After append, x =  [1, 3, 0, 7, 11, 13]
After remove item 1, x =  [3, 0, 7, 11, 13]
Insert 42 at item 1, x =  [3, 42, 0, 7, 11, 13]
```

Indexing

- Indexing for a list
  ```
  >>> x=[1,3,5,7,11]
  >>> x[2]
  5
  >>> x[2]=0
  >>> x
  [1, 3, 0, 7, 11]
  ```
- Can index a tuple the same way, but can't change the values because tuples are immutable
- Slices work as for strings (recall last index of slice is not included)
  ```
  >>> x[2:4]
  [0,7]
  ```

The += operator for lists

- >>> a = [1, 3, 5]
  >>> a += [7] # a += 7 fails
  >>> a
  [1, 3, 5, 7]
  >>> a += [“the-end”] # put this in [] also!
  >>>a
  [1, 3, 5, 7, 'the-end']

Dictionaries

- Unordered collections where items are accessed by a key, not by the position in the list
- Like a hash in Perl
- Collection of arbitrary objects; use object references like lists
- Nestable
- Can grow and shrink in place like lists
- Concatenation, slicing, and other operations that depend on the order of elements do not work on dictionaries

Dictionary Construction and Access

- Example:
  >>> jobs = {'David':'Professor', 'Sahan':'Postdoc', 'Shawn':'Grad student'}
  >>> jobs['Sahan']
  >>> 'Postdoc'
- Can change in place
  >>> jobs['Shawn'] = 'Postdoc'
  >>> jobs['Shawn']
  'Postdoc'
- Lists of keys and values
  >>> jobs.keys()
  ['Sahan', 'Shawn', 'David'] # note order is diff
  >>> jobs.values()
  ['Postdoc', 'Postdoc', 'Professor']
  >>> jobs.items()
  [('Sahan', 'Postdoc'), ('Shawn', 'Postdoc'), ('David', 'Professor')]

Common Dictionary Operations

- Delete an entry (by key)
  del d['keyname']
- Add an entry
  d['newkey'] = newvalue
- See if a key is in dictionary
  d.has_key('keyname') or 'keyname' in d
- get() method useful to return value but not fail (return None) if key doesn't exist (or can provide a default value)
  d.get('keyval', default)
- update() merges one dictionary with another (overwriting values with same key)
  d.update(d2) [the dictionary version of concatenation]

Dictionary example

- Going through a dictionary by keys:

```
bookauthors = {'Gone with the Wind':
'Margaret Mitchell',
                'Aeneid': 'Virgil',
                'Odyssey': 'Homer'}

for book in bookauthors:
    print book, 'by', bookauthors[book]

output:
Gone with the Wind by Margaret Mitchell
Aeneid by Virgil
Odyssey by Homer
```

Constructing dictionaries from lists

- If we have separate lists for the keys and values, we can combine them into a dictionary using the zip function and a dict constructor:

```
keys = ['david', 'chris', 'stewart']
values = ['504', '637', '921']
D = dict(zip(keys, vals))
```

More general keys

- Keys don't have to be strings; they can be any immutable data type, including tuples
- This might be good for representing sparse matrices, for example

```
Matrix = {}          # start a blank dictionary
Matrix[(1,0,1)]  = 0.5 # key is a tuple
Matrix[(1,1,4)]  = 0.8
```

Length of collections

- len() returns the length of a tuple, list, or dictionary (or the number of characters of a string):

```
>>>len((“Tony”,))
1
>>>len(“Tony”)
4
>>>len([0, 1, 'boom'])
3
```

The “is” operator

- Python “variables” are really object references. The “is” operator checks to see if these references refer to the *same* object (note: could have two identical objects which are not the same object...)
- References to integer constants should be identical. References to strings may or may not show up as referring to the same object. Two identical, mutable objects are not necessarily the same object

is-operator.py

- x = "hello"
  y = "hello"
  print x is y True here, not nec.

  x = [1,2]
  y = [1,2]
  print x is y False (even though ident)

  x = (1,2)
  y = (1,2)
  print x is y False (even though identical, immutable)

  x = []
  print x is not None True (list, even though empty)

“in” operator

- For collection data types, the “in” operator determines whether something is a member of the collection (and “not in” tests if not a member):
  >>> team = (“David”, “Robert”, “Paul”)
  >>> “Howell” in team
  False
  >>> “Stewart” not in team
  True

Iteration: for ... in

- To traverse a collection type, use for ... in
  ```
  >>> numbers = (1, 2, 3)
  >>> for i in numbers: print i,
  ...
  1 2 3
  ```

Copying collections

- Using assignment just makes a new reference to the same collection, e.g.,

```
A = [0,1,3]
B = A          # B is a ref to A.  Changing B will
               # change A
C = A[:]       # make a copy
B[1] = 5
C[1] = 7
A, B, C
( [0, 5, 3], [0, 5, 3], [0, 7, 3])
```

- Copy a dictionary with A.copy()

Chapter 7: Advanced Functions

- Passing lists and keyword dictionaries to functions
- Lambda functions
- apply()
- map()
- filter()
- reduce()
- List comprehensions

Passing lists as arguments

- Lists can be passed in cases where there may be a variable number of arguments

  listarg.py:

```
def sum(*args):
  result = 0
  for arg in args:
    result += arg
  return result

print sum(1,2,3)
6
```

Keyword arguments

- The caller of a function can place the arguments in the correct order (“positional arguments”), but alternatively, some arguments can be designated by keywords
- Note: this example calls the function with the arguments in the wrong order, but the arguments are passed with keyword syntax so it doesn't matter:

```
def print_hello(firstname="", lastname=""):
  print "Hello, %s %s" % (firstname, lastname)

print_hello(lastname="Sherrill",
  firstname="David")
```

Hello, David Sherrill

Mixing keyword and positional arguments

- Another use of the * operator is to designate that all arguments after it must be keyword arguments
- This is new to Python 3?

```
def area(x, y, *, units="inches"):
  print x*y, "square %s" % units

area(2,3,"centimeters")
6 square centimeters
```

Passing dictionaries to functions

- Dictionaries will be discussed later

dict-args.py:

```
def print_dict(**kwargs):
  for key in kwargs.keys():
    print "%s = %s" % (key, kwargs[key])

user_info = dict(name="David", uid=593,
homedir="/home/users/david")

print_dict(**user_info) # note: need ** here!!
```

output: (note: dictionary entries are unordered)

```
uid = 593
name = David
homedir = /home/users/david
```

Dictionary arguments

- Note: dictionaries can be passed as arguments even to “normal” functions wanting positional arguments!

  dict-args.py:

```
def area(x, y, units="inches"):
  print x*y, "square %s" % units

area_info = dict(x=2, y=3, units="centimeters")
area(**area_info)
```

  output: 6 square centimeters

Lambda functions

- Shorthand version of def statement, useful for “inlining” functions and other situations where it's convenient to keep the code of the function close to where it's needed
- Can only contain an expression in the function definition, not a block of statements (e.g., no if statements, etc)
- A lambda returns a function; the programmer can decide whether or not to assign this function to a name

Lambda example

- Simple example:

```
>>> def sum(x,y): return x+y
>>> ...
>>> sum(1,2)
3
>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```

apply()

- In very general contexts, you may not know ahead of time how many arguments need to get passed to a function (perhaps the function itself is built dynamically)
- The apply() function calls a given function with a list of arguments packed in a tuple:
  def sum(x, y): return x+y
  apply(sum, (3, 4))
  7
- Apply can handle functions defined with def or with lambda

More general apply()

- Apply can take a third argument which is a dictionary of keyword arguments

```
def nameargs(*args, **kwargs): print args, kwargs
args = (1.2, 1.3)
kargs = {'a': 2.1, 'b': 3.4}
apply(nameargs, args, kwargs)
(1.2, 1.3) {'a': 2.1000, 'b': 3.4000}
```

map

- Map calls a given function on every element of a sequence
- map.py:
  def double(x): return x*2
  a = [1, 2, 3]
  print map(double, a)
  [2, 4, 6]
- Alternatively:
  print map((lambda x: x*2), a)
  [2, 4, 6]

More map

- Can also apply a map to functions taking more than one argument; then work with two or more lists

  print map((lambda x, y: x+y), [1,2,3], [4,5,6])
  [5,7,9]

filter

- Like map, the filter function works on lists. Unlike map, it returns only a selected list of items that matches some criterion.
- Get only even numbers:
  filter((lambda x: x%2==0), range(-4,4))
  [-4, -2, 0, 2] # recall range doesn't include last
  # value given in range

reduce

- Reduce is like map but it reduces a list to a single value (each operation acts on the result of the last operation and the next item in the list):

  ```
  reduce((lambda x, y: x+y), [0, 1, 2, 3, 4])
  10
  ```

List comprehensions

- These aren't really functions, but they can replace functions (or maps, filters, etc)

```
>>> [x**2 for x in range(9)]
[0, 1, 4, 9, 16, 25, 36, 49, 64]
>>> [x**2 for x in range(10) if x%2 == 0]
[0, 4, 16, 36, 64]
>>> [x+y for x in [1,2,3] for y in [100,200,300]]
[101, 201, 301, 102, 202, 302, 103, 203, 303]
```

Generators and Iterators

- Sometimes it may be useful to return a single value at a time, instead of an entire sequence. A generator is a function which is written to return values one at a time. The generator can compute the next value, save its state, and then pick up again where it left off when called again.
- Syntactically, the generator looks just like a function, but it “yields” a value instead of returning one (a return statement would terminate the sequence)

Generator example

- >>> def squares(x):
  ... for i in range(x):
  ... yield i**2
  >>> for i in squares(4):
  ... print i,
  ...
  0 1 4 9
- Can get next element in sequence from the next() function:

  z = squares(4)
  z.next()
  0
  z.next()
  1

Persistence of mutable default arguments

- Mutable default arguments persist between calls to the function (like static variables in C)
- This may not be the behavior desired
- If not, copy the default at the start of the function body to another variable, or move the default value expression into the body of the function

Chapter 8: Exception Handling

- Basics of exception handling

Basic Exception Handling

- An “exception” is a (recognized type of) error, and “handling” is what you do when that error occurs
- General syntax:

```
try:
    code-you-want-to-run
except exception1 [as variable1]:
    exception1 block
...
except exceptionN [as variableN]:
    exceptionN block
```

- If an error occurs, if it's of exception type 1, then variable1 becomes an alias to the exception object, and then exception1 block executes. Otherwise, Python tries exception types 2 ... N until the exception is caught, or else the program stops with an unhandled exception (a traceback will be printed along with the exception's text)
- The optional [as variable] will not work with older Python

Exception example

- value-error.pl

```
try:
    i = int("snakes")
    print "the integer is", i
except ValueError:
    print "oops!  invalid value"
```

Other exceptions

- EOFError is raised at the end of a file
- IndexError happens if we use an invalid index for a string/collection, e.g., if we try to get argv[1] if there is only one command-line argument (counting starts from zero)
- TypeError happens when, e.g., comparing two incomparable types

Chapter 9: Python Modules

- Basics of modules
- Import and from … import statements
- Changing data in modules
- Reloading modules
- Module packages
- __name__ and __main__
- Import as statement

Module basics

- Each file in Python is considered a module. Everything within the file is encapsulated within a namespace (which is the name of the file)
- To access code in another module (file), import that file, and then access the functions or data of that module by prefixing with the name of the module, followed by a period
- To import a module:
  import sys
  (note: no file suffix)
- Can import user-defined modules or some “standard” modules like sys and random
- Any python program needs one “top level” file which imports any other needed modules

Python standard library

- There are over 200 modules in the Standard Library
- Consult the Python Library Reference Manual, included in the Python installation and/or available at http://www.python.org

What import does

- An import statement does three things:
  - Finds the file for the given module
  - Compiles it to bytecode
  - Runs the module's code to build any objects (top-level code, e.g., variable initialization)
- The module name is only a simple name; Python uses a module search path to find it. It will search: (a) the directory of the top-level file, (b) directories in the environmental variable PYTHONPATH, (c) standard directories, and (d) directories listed in any .pth files (one directory per line in a plain text file); the path can be listed by printing sys.path

The sys module

- Printing the command-line arguments, print-argv.pl

```
import sys

cmd_options = sys.argv
i = 0
for cmd in cmd_options:
    print "Argument ", i, "=", cmd
    i += 1
```

output:

```
localhost(Chapter8)% ./print-argv.pl test1 test2
Argument  0 = ./print-argv.pl
Argument  1 = test1
Argument  2 = test2
```

The random module

- import random

  guess = random.randint(1,100)
  print guess
  dinner = random.choice([“meatloaf”, “pizza”, “chicken pot pie”])
  print dinner

Import vs from ... import

- Import brings in a whole module; you need to qualify the names by the module name (e.g., sys.argv)
- “import from” copies names from the module into the current module; no need to qualify them (note: these are copies, not links, to the original names)

```
from module_x import junk
junk()  # not module_x.junk()

from module_x import * # gets all top-level
                       # names from module_x
```

Changing data in modules

- Reassigning a new value to a fetched name from a module does not change the module, but changing a mutable variable from a module does:

  ```
  from module_x import x,y

  x = 30   # doesn't change x in module_x
  y[0] = 1 # changes y[0] in module_x
  ```

- This works just like functions
- To actually change a global name in another file, could use import (without “from”) and qualify the variable: module_x.x = 30 (but this breaks data encapsulation)

Reloading modules

- A module's top-level code is only run the first time the module is imported. Subsequent imports don't do anything.
- The reload function forces a reload and re-run of a module; can use if, e.g., a module changes while a Python program is running
- reload is passed an existing module object

```
reload(module_x) # module_x must have been
                 # previously imported
```

- reload changes the module object in-place
- reload does not affect prior from..import statements (they still point to the old objects)

Module Packages

- When using import, we can give a directory path instead of a simple name. A directory of Python code is known as a “package”:
  import dir1.dir2.module
  or
  from dir1.dir2.module import x
  will look for a file dir1/dir2/module.py
- Note: dir1 must be within one of the directories in the PYTHONPATH
- Note: dir1 and dir2 must be simple names, not using platform-specific syntax (e.g., no C:\)

Package __init__.py files

- When using Python packages (directory path syntax for imports), each directory in the path needs to have an __init__.py file
- The file could be blank
- If not blank, the file contains Python code; the first time Python imports through this directory, it will run the code in the __init__.py file
- In the dir1.dir2.module example, a namespace dir1.dir2 now exists which contains all names assigned by dir2's __init__.py file
- The file can contain an "__all__" list which specifies what is exported by default when a directory is imported with the from* statement

Data encapsulation

- By default, names beginning with an underscore will not be copied in an import statement (they can still be changed if accessed directly)
- Alternatively, one can list the names to be copied on import by assigning them to a list called __all__:
  __all__ = [“x1”, “y1”, “z1”] # export only these
  this list is only read when using the from * syntax

__name__ and __main__

- When a file is run as a top-level program, it's __name__ is set to “__main__” when it starts
- If a file is imported, __name__ is set to the name of the module as the importer sees it
- Can use this to package a module as a library, but allow it to run stand-alone also, by checking

```
if __name__ == '__main__':
    do_whatever() # run in stand-alone mode
```

Import as

- You can rename a module (from the point of view of the importer) by using:
  import longmodulename as shortname,
  where shortname is the alias for the original module name
- Using this syntax requires you to use the short name thereafter, not the long name
- Can also do
  from module import longname as name

Reload may not affect “from” imports

- From copies names, and does not retain a link back to the original module
- When reload is run, it changes the module, but not any copies that were made on an original from module import XX
  statement
- If this is a problem, import the entire module and use name qualification (module.XX) instead

Chapter 10: Files

- Basic file operations

Opening a file

- open(filename, mode)

  where filename is a Python string, and mode is a Python string, 'r' for reading, 'w' for writing, or 'a' for append

Basic file operations

- Basic operations:

```
outfile = open('output.dat', 'w')
infile = open('input.dat', r')
```

Basic file operations

- Basic operations

```
output = open('output.dat', 'w')
input = open('input.dat', 'r')

A = input.read()         # read whole file into string
A = input.read(N)        # read N bytes
A = input.readline()     # read next line
A = input.readlines()    # read file into list of
# strings

output.write(A)  # Write string A into file
output.writelines(A)  # Write list of strings
output.close()           # Close a file
```

Redirecting stdout

- Print statements normally go to stdout ("standard output," i.e., the screen)
- stdout can be redirected to a file:

```
import sys
sys.stdout = open('output.txt', 'w')
print message   # will show up in output.txt
```

- Alternatively, to print just some stuff to a file:
  print >> logfile, message # if logfile open

Examples of file parsing

Whole thing at once:

```
infile = open(“test.txt”, 'r')
lines = infile.readlines()
infile.close()

for line in lines:
  print line,
```

Line-by-line (shortcut syntax avoiding readline calls):

```
infile = open(“test.txt”, 'r')

for line in infile
  print line,

infile.close()
```

Chapter 11: Documentation

- Comments
- dir
- Documentation strings

Comments

- As we've seen, anything after a # character is treated as a comment

dir

- The dir function prints out all the attributes of an object (see later)

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__',
'__excepthook__','__name__', '__stderr__',
'__stdin__', '__stdout__', '_current_frames',
'_getframe', 'api_version', 'argv',
...
>>> dir([])  # a list
['__add__', '__class__', '__contains__', ... ,
'append', 'count', 'extend', 'index', ...]
```

docstrings

- Strings at the top of module files, the top of functions, and the top of classes become “docstrings” which are automatically inserted into the __doc__ attribute of the object
- Can use regular single or double quotes to delimit the string, or can use triple quotes for a multi-line string
- e.g.,:

```
def sum(x, y):
  “This function just adds two numbers”
  return x+y

print sum.__doc__
This function just adds two numbers
```

Docstrings for built-in objects

- Can print the docstrings for built-in Python objects

```
>>> import sys
>>> print sys.__doc__
```

PyDoc

- PyDoc is a tool that can extract the docstrings and display them nicely, either via the help command, or via a GUI/HTML interface
  >>> help(list)

Documentation on the web

- Check out www.python.org

Chapter 12: Classes

- Introduction to classes
- stuff
- junk

Introduction to Classes

- A “class” is a user-defined data type. It contains (or encapsulates) certain “member data,” and it also has associated things it can do, or “member functions.”
- If a class is a specialization (or sub-type) of a more general class, it is said to inherit from that class; this can give a class access to a parent class' member data/functions
- These general concepts are basically the same in C++, Java, etc.
- Classes are built into Python from the beginning. For object-oriented programming (i.e., programming using classes), Python is superior in this sense to Perl

Defining Classes

- A class is defined by:

```
class classname:
    suite-of-code
```

or

```
class classname(base_classes):
    suite-of-code
```

- The class definition suite (code block) can contain member function definitions, etc.
- base_classes would contain any classes that this class inherits from

Member Data Scope

- Class attributes (member data) defined in the class definition itself are common to the class (note: the code block in a class definition actually runs the first time the class definition is encountered, and not when class objects are instantiated)
- Class attributes defined for a specific object of a certain class are held only with that particular object (often set using the “self” keyword, see below)

Class Example

```
class Student:
    course = "CHEM 3412"
    def __init__(self, name, test1=0, test2=0):
      self.name = name
      self.test1 = test1
      self.test2 = test2
    def compute_average(self):
      return ((self.test1 + self.test2)/2)
    def print_data(self):
      print "The grade for %s in %s is %4.1f" %  \
        (self.name, self.course, self.compute_average())


David = Student("David", 90, 100)
Bob = Student("Bob", 60, 80)
David.print_data()
Bob.print_data()
```

Comments on Student Example

- Calling a function with the same name as the name of the class creates an object of that class; the actual function called is the __init__ function within the class (__init__ is the class *constructor*)
- Each time the constructor is called, a different object of that type is created; these different objects will generally have different member data

Comments on Student Example

- Note that the course name is kept common to all students (all instances of the class)
- However, the two test grades are specific to each student
- The “self” keyword has the same role as “this” in C++/Java --- it points to a specific instance of the class. It is automatically supplied as the first argument to any class member function, and it must be used when referring to any member data/function of a class (even if it's member data shared among all instances of a class)

Operator Overloading

- Certain operators (like ==, +, -, *) can be *overloaded* to work on user-defined datatypes (classes) as well as built-in types like integers
- To see if two objects are equal, overload the == operator and implement a function which can compare two instances of your user-defined datatype (class). This is done by defining the __eq__ function for the class; see next example

Point example

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __eq__(self, other):
    return self.x == other.x and self.y == other.y

a = Point()
b = Point(1,1)

print a==b
```

Fancier equals checking

- We could be a little safer and make sure the comparison is actually of two objects of the same type, otherwise return a flag indicating that this comparison isn't implemented by our Point class:

```
if not isinstance(other, Point):
  return notImplemented
else
  return self.x == other.x and self.y == other.y
```

Overloading other comparisons

- There are other comparison operators we can overload for user-defined datatypes:
  __lt__(self, other) x < y
  __le__(self, other) x<= y
  __eq__(self, other) x == y
  __ne__(self, other) x != y
  __ge__(self, other) x >= y
  __gt__(self, other) x > y

No Overloading Call Signatures

- In other languages like C++, we can overload member functions to behave differently depending on how many arguments (and what types of arguments) are passed to the function
- Python doesn't work like this: the last function defined with a given name is the one that will be used:

```
class A:
   def method(self, x):
     …
   def method(self, x, y):
     # this definition overrides the previous one
```

Inheritance

- A class can *inherit* from another class, allowing a *class hierarchy*
- If class A inherits from class B, then class B is a *superclass* of class A, and class A automatically has access to all the member data and member functions of class B
- If class A redefines some member function or member data which is also present in class B, then the definition in A takes precedence over that in class B

Inheritance Example

```
class Class1:
  def __init__(self, data=0):
    self.data = data

  def display(self):
    print self.data

class Class2(Class1):
  def square(self):
    self.data = self.data * self.data

a = Class2(5)
a.square()      # member function specific to Class2
a.display()     # inherited member function from Class1
```

Alternative Syntax for Method Calls

- Instead of calling methods through their objects, we can also call them through the class name:

```
x = SomeClass()
x.method1()
-or-
SomeClass.method1(x)
```

- Useful if we need to guarantee that a superclass constructor runs as well as the subclass constructor:

```
class Class2(Class1):
  def __init__(self, data):
    Class1.__init__(self, data)
    … new code for Class2 here...
```

__getitem__

- We can overload array indexing syntax with __getitem__

```
class testit:
   def __getitem__(self, n):
    return self.data[n]

A = testit()
A.data = “junk”

A[1]
'u'
```

__getattr__ and __setattr_

- These functions catch attribute references for a class. __getattr__ catches all attempts to get a class attribute (via, e.g., x.name, etc.), and __setattr__ catches all attempts to set a class attribute (e.g., x.name = “Bob”)
- Must be careful with __setattr__, because all instances of self.attr=value become self.__setattr__('attr', value), and this can prevent one from writing code such as self.attr=value in the definition of __setattr__!

Using __setattr__

```
Class SomeClass:
   def __setattr__(self, attr, value):
      self.__dict__[attr] = value

# this avoids syntax like self.attr = value
   # which we can't have in the definition
```

__str__ and __repr__

- __str__ is meant to provide a “user friendly” representation of the class for printing
- __repr__ is meant to provide a string which could be interpreted as code for reconstructing the class instance

```
class Number:
   def __init__(self, data):
     self.data = data
   def __repr__(self):
     return 'Number(%s)' % self.data

a = Number(5)
print a
>> Number(5)
```

destructors

- __del__ is a destructor, which is called automatically when an instance is being deleted (it is no longer being used, and it's space is reclaimed during “garbage collection”)
- Python normally cleans up the memory used by an instance automatically, so this kind of thing isn't usually necessary to put in a destructor; hence, they may be less necessary than in, for example, C++