Python 101 --- Introduction to Python

Author: Dave Kuhlman
Address:
dkuhlman (at) davekuhlman (dot) org
http://www.davekuhlman.org

Revision: 1.1a
Date: October 05, 2014

Abstract: This document is a self-learning document for a first course in Python programming. This course contains an introduction to the Python language, instruction in the important and commonly used features of the language, and practical exercises in the use of those features.

Contents

1 Introductions Etc

Introductions

Practical matters: restrooms, breakroom, lunch and break times, etc.

Starting the Python interactive interpreter. Also, IPython and Idle.

Running scripts

Editors -- Choose an editor which you can configure so that it indents with 4 spaces, not tab characters. For a list of editors for Python, see: http://wiki.python.org/moin/PythonEditors. A few possible editors:

- SciTE -- http://www.scintilla.org/SciTE.html.
- MS Windows only -- (1) TextPad -- http://www.textpad.com; (2) UltraEdit -- http://www.ultraedit.com/.
- Jed -- See http://www.jedsoft.org/jed/.
- Emacs -- See http://www.gnu.org/software/emacs/ and http://www.xemacs.org/faq/xemacs-faq.html.
- jEdit -- Requires a bit of customization for Python -- See http://jedit.org.
- Vim -- http://www.vim.org/
- Geany -- http://www.geany.org/
- And many more.

Interactive interpreters:

- `python`
- `ipython`
- Idle

IDEs -- Also see http://en.wikipedia.org/wiki/List_of_integrated_development_environments_for_Python:

- PyWin -- MS Windows only. Available at: http://sourceforge.net/projects/pywin32/.
- WingIDE -- See http://wingware.com/wingide/.
- Eclipse -- http://eclipse.org/. There is a plug-in that supports Python.
- Kdevelop -- Linux/KDE -- See http://www.kdevelop.org/.
- Eric -- Linux KDE? -- See http://eric-ide.python-projects.org/index.html
- Emacs and SciTE will evaluate a Python buffer within the editor.

1.1 Resources

Where else to get help:

- Python home page -- http://www.python.org
- Python standard documentation -- http://www.python.org/doc/.

  You will also find links to tutorials there.
- FAQs -- http://www.python.org/doc/faq/.
- The Python Wiki -- http://wiki.python.org/
- The Python Package Index -- Lots of Python packages -- https://pypi.python.org/pypi
- Special interest groups (SIGs) -- http://www.python.org/sigs/

- Other python related mailing lists and lists for specific applications (for example, Zope, Twisted, etc). Try: http://dir.gmane.org/search.php?match=python.
- http://sourceforge.net -- Lots of projects. Search for "python".
- USENET -- comp.lang.python. Can also be accessed through Gmane: http://dir.gmane.org/gmane.comp.python.general.
- The Python tutor email list -- http://mail.python.org/mailman/listinfo/tutor

Local documentation:

- On MS Windows, the Python documentation is installed with the standard installation.
- Install the standard Python documentation on your machine from http://www.python.org/doc/.
- `pydoc`. Example, on the command line, type: `pydoc re`.
- Import a module, then view its `.__doc__` attribute.
- At the interactive prompt, use `help(obj)`. You might need to import it first. Example:

```
>>> import urllib
>>> help(urllib)
```

- In IPython, the question mark operator gives help. Example:

```
In [13]: open?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function open>
Namespace:      Python builtin
Docstring:
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file object.
Constructor Docstring:
    x.__init__(...) initializes x; see x.__class__.__doc__ for
signature
Callable:       Yes
Call def:       Calling definition not available.Call docstring:
    x.__call__(...) <==> x(...)
```

1.2 A general description of Python

Python is a high-level general purpose programming language:

- Because code is automatically compiled to byte code and executed, Python is suitable for use as a scripting language, Web application implementation language, etc.
- Because Python can be extended in C and C++, Python can provide the speed needed for even compute intensive tasks.
- Because of its strong structuring constructs (nested code blocks, functions, classes, modules, and packages) and its consistent use of objects and object-oriented programming, Python enables us to write clear, logical applications for small and large tasks.

Important features of Python:

- Built-in high level data types: strings, lists, dictionaries, etc.
- The usual control structures: if, if-else, if-elif-else, while, plus a powerful collection iterator (for).
- Multiple levels of organizational structure: functions, classes, modules, and packages. These assist in organizing code. An excellent and large example is the Python standard library.
- Compile on the fly to byte code -- Source code is compiled to byte code without a separate compile step. Source code modules can also be "pre-compiled" to byte code files.
- Object-oriented -- Python provides a consistent way to use objects: everything is an object. And, in Python it is easy to implement new object types (called classes in object-oriented programming).
- Extensions in C and C++ -- Extension modules and extension types can be written by hand. There are also tools that help with this, for example, SWIG, sip, Pyrex.
- Jython is a version of Python that "plays well with" Java. See: The Jython Project -- http://www.jython.org/Project/.

Some things you will need to know:

- Python uses indentation to show block structure. Indent one level to show the beginning of a block. Out-dent one level to show the end of a block. As an example, the following C-style code:

```
if (x)
{
    if (y)
    {
        f1()
    }
    f2()
}
```

  in Python would be:

```
if x:
    if y:
        f1()
    f2()
```

And, the convention is to use four spaces (and no hard tabs) for each level of indentation. Actually, it's more than a convention; it's practically a requirement. Following that "convention" will make it so much easier to merge your Python code with code from other sources.

An overview of Python:

- A scripting language -- Python is suitable (1) for embedding, (2) for writing small unstructured scripts, (3) for "quick and dirty" programs.

- *Not* a scripting language -- (1) Python scales. (2) Python encourages us to write code that is clear and well-structured.

Interpreted, but also compiled to byte-code. Modules are automatically compiled (to .pyc) when imported, but may also be explicitly compiled.

- Provides an interactive command line and interpreter shell. In fact, there are several.
- Dynamic -- For example:
    - Types are bound to values, not to variables.
    - Function and method lookup is done at runtime.
    - Values are inspect-able.
    - There is an interactive interpreter, more than one, in fact.
    - You can list the methods supported by any given object.
- Strongly typed at run-time, not compile-time. Objects (values) have a type, but variables do not.
- Reasonably high level -- High level built-in data types; high level control structures (for walking lists and iterators, for example).
- Object-oriented -- Almost everything is an object. Simple object definition. Data hiding by agreement. Multiple inheritance. Interfaces by convention. Polymorphism.
- Highly structured -- Statements, functions, classes, modules, and packages enable us to write large, well-structured applications. Why structure? Readability, locate-ability, modifiability.
- Explicitness
- First-class objects:
    - Definition: Can (1) pass to function; (2) return from function; (3) stuff into a data structure.
    - Operators can be applied to *values* (not variables). Example: `f(x)[3]`
- Indented block structure -- "Python is pseudo-code that runs."
- Embedding and extending Python -- Python provides a well-documented and supported way (1) to embed the Python interpreter in C/C++ applications and (2) to extend Python with modules and objects implemented in C/C++.
    - In some cases, SWIG can generate wrappers for existing C/C++ code automatically. See http://www.swig.org/
    - Cython enables us to generate C code from Python *and* to "easily" create wrappers for C/C++ functions. See http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
    - To embed and extend Python with Java, there is Jython. See http://www.jython.org/
- Automatic garbage collection. (But, there is a `gc` module to allow explicit control of garbage collection.)
- Comparison with other languages: compiled languages (e.g. C/C++); Java; Perl, Tcl, and Ruby. Python excells at: development speed, execution speed, clarity and maintainability.
- Varieties of Python:

- CPython -- Standard Python 2.x implemented in C.
- Jython -- Python for the Java environment -- http://www.jython.org/
- PyPy -- Python with a JIT compiler and stackless mode -- http://pypy.org/
- Stackless -- Python with enhanced thread support and microthreads etc. -- http://www.stackless.com/
- IronPython -- Python for .NET and the CLR -- http://ironpython.net/
- Python 3 -- The new, new Python. This is intended as a replacement for Python 2.x. -- http://www.python.org/doc/. A few differences (from Python 2.x):
  - The `print` statement changed to the `print` function.
  - Strings are unicode by default.
  - Classes are all "new style" classes.
  - Changes to syntax for catching exceptions.
  - Changes to integers -- no long integer; integer division with automatic convert to float.
  - More pervasive use of iterables (rather than collections).
  - Etc.

  For a more information about differences between Python 2.x and Python 3.x, see the description of the various fixes that can be applied with the `2to3` tool: http://docs.python.org/3/library/2to3.html#fixers

  The migration tool, `2to3`, eases the conversion of 2.x code to 3.x.

- Also see The Zen of Python -- http://www.python.org/peps/pep-0020.html. Or, at the Python interactive prompt, type:

```
>>> import this
```

1.3 Interactive Python

If you execute Python from the command line with no script (no arguments), Python gives you an interactive prompt. This is an excellent facility for learning Python and for trying small snippets of code. Many of the examples that follow were developed using the Python interactive prompt.

Start the Python interactive interpreter by typing `python` with no arguments at the command line. For example:

```
$ python
Python 2.6.1 (r261:67515, Jan 11 2009, 15:19:23)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print 'hello'
hello
>>>
```

You may also want to consider using IDLE. IDLE is a graphical integrated development environment for Python; it contains a Python shell. It is likely that Idle was installed for you when you installed Python. You will find a script to start up IDLE in the Tools/scripts directory of your Python distribution. IDLE requires Tkinter.

In addition, there are tools that will give you a more powerful and fancy Python interactive interpreter. One example is IPython, which is available at http://ipython.scipy.org/.

2 Lexical matters

2.1 Lines

- Python does what you want it to do *most* of the time so that you only have to add extra characters *some* of the time.
- Statement separator is a semi-colon, but is only needed when there is more than one statement on a line. And, writing more than one statement on the same line is considered bad form.
- Continuation lines -- A back-slash as last character of the line makes the following line a continuation of the current line. But, note that an opening "context" (parenthesis, square bracket, or curly bracket) makes the back-slash unnecessary.

2.2 Comments

Everything after "#" on a line is ignored. No block comments, but doc strings are a comment in quotes at the beginning of a module, class, method or function. Also, editors with support for Python often provide the ability to comment out selected blocks of code, usually with "##".

2.3 Names and tokens

- Allowed characters: a-z A-Z 0-9 underscore, and must begin with a letter or underscore.
- Names and identifiers are case sensitive.
- Identifiers can be of unlimited length.
- Special names, customizing, etc. -- Usually begin and end in double underscores.
- Special name classes -- Single and double underscores.
    - Single leading single underscore -- Suggests a "private" method or variable name. Not imported by "from module import *".
    - Single trailing underscore -- Use it to avoid conflicts with Python keywords.
    - Double leading underscores -- Used in a class definition to cause name mangling (weak hiding). But, not often used.
- Naming conventions -- Not rigid, but:
    - Modules and packages -- all lower case.
    - Globals and constants -- Upper case.
    - Classes -- Bumpy caps with initial upper.
    - Methods and functions -- All lower case with words separated by underscores.
    - Local variables -- Lower case (with underscore between words) or bumpy caps with initial lower or your choice.
    - Good advice -- Follow the conventions used in the code on which you are working.
- Names/variables in Python do not have a type. Values have types.

2.4 Blocks and indentation

Python represents block structure and nested block structure with indentation, not with begin and end brackets.

The empty block -- Use the `pass` no-op statement.

Benefits of the use of indentation to indicate structure:

- Reduces the need for a coding standard. Only need to specify that indentation is 4 spaces and no hard tabs.
- Reduces inconsistency. Code from different sources follow the same indentation style. It has to.
- Reduces work. Only need to get the indentation correct, not *both* indentation and brackets.
- Reduces clutter. Eliminates all the curly brackets.
- If it looks correct, it is correct. Indentation cannot fool the reader.

Editor considerations -- The standard is 4 spaces (no hard tabs) for each indentation level. You will need a text editor that helps you respect that.

2.5 Doc strings

Doc strings are like comments, but they are carried with executing code. Doc strings can be viewed with several tools, e.g. `help()`, `obj.__doc__`, and, in IPython, a question mark (`?`) after a name will produce help.

A doc string is written as a quoted string that is at the top of a module or the first lines after the header line of a function or class.

We can use triple-quoting to create doc strings that span multiple lines.

There are also tools that extract and format doc strings, for example:

- pydoc -- Documentation generator and online help system -- http://docs.python.org/lib/module-pydoc.html.
- epydoc -- Epydoc: Automatic API Documentation Generation for Python -- http://epydoc.sourceforge.net/index.html
- Sphinx -- Can also extract documentation from Python doc strings. See http://sphinx-doc.org/index.html.

See the following for suggestions and more information on doc strings: Docstring conventions -- http://www.python.org/dev/peps/pep-0257/.

2.6 Program structure

- Execution -- def, class, etc are executable statements that add something to the current name-space. Modules can be both executable and import-able.
- Statements, data structures, functions, classes, modules, packages.
- Functions
- Classes
- Modules correspond to files with a "*.py" extension. Packages correspond to a directory (or folder) in the file system; a package contains a file named "__init__.py". Both modules and packages can be imported (see section import statement).
- Packages -- A directory containing a file named "__init__.py". Can provide additional

initialization when the package or a module in it is loaded (imported).

2.7 Operators

- See: http://docs.python.org/ref/operators.html. Python defines the following operators:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

  The comparison operators `<>` and `!=` are alternate spellings of the same operator. `!=` is the preferred spelling; `<>` is obsolescent.

- Logical operators:

```
and     or      is      not     in
```

- There are also (1) the dot operator, (2) the subscript operator `[]`, and the function/method call operator `()`.

- For information on the precedences of operators, see the table at http://docs.python.org/2/reference/expressions.html#operator-precedence, which is reproduced below.

- For information on what the different operators *do*, the section in the "Python Language Reference" titled "Special method names" may be of help: http://docs.python.org/2/reference/datamodel.html#special-method-names

  The following table summarizes the operator precedences in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators on the same line have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators on the same line group left to right (except for comparisons, including tests, which all have the same precedence and chain from left to right -- see section 5.9 -- and exponentiation, which groups from right to left):

```
Operator                  Description
========================  ==================
lambda                    Lambda expression
or                        Boolean OR
and                       Boolean AND
not x                     Boolean NOT
in, not in                Membership tests
is, is not                Identity tests
<, <=, >, >=, <>, !=, ==  Comparisons
|                         Bitwise OR
^                         Bitwise XOR
&                         Bitwise AND
<<, >>                    Shifts
+, -                      Addition and subtraction
*, /, %                   Multiplication, division, remainder
+x, -x                    Positive, negative
~x                        Bitwise not
**                        Exponentiation
x.attribute               Attribute reference
x[index]                  Subscription
```

```
x[index:index]          Slicing
f(arguments...)         Function call
(expressions...)        Binding or tuple display
[expressions...]        List display
{key:datum...}          Dictionary display
`expressions...`        String conversion
```

- Note that most operators result in calls to methods with special names, for example `__add__`, `__sub__`, `__mul__`, etc. See Special method names http://docs.python.org/2/reference/datamodel.html#special-method-names

  Later, we will see how these operators can be emulated in classes that you define yourself, through the use of these special names.

2.8 Also see

For more on lexical matters and Python styles, see:

- Code Like a Pythonista: Idiomatic Python -- http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html.
- Style Guide for Python Code -- http://www.python.org/dev/peps/pep-0008/
- The flake8 style checking program. See https://pypi.python.org/pypi/flake8. Also see the pylint code checker: https://pypi.python.org/pypi/pylint.

2.9 Code evaluation

Understanding the Python execution model -- How Python evaluates and executes your code.

Evaluating expressions.

Creating names/variables -- Binding -- The following all create names (variables) and bind values (objects) to them: (1) assignment, (2) function definition, (3) class definition, (4) function and method call, (5) importing a module, ...

First class objects -- Almost all objects in Python are first class. Definition: An object is first class if: (1) we can put it in a structured object; (2) we can pass it to a function; (3) we can return it from a function.

References -- Objects (or references to them) can be shared. What does this mean?

- The object(s) satisfy the identity test operator `is`.
- The built-in function `id()` returns the same value.
- The consequences for mutable objects are different from those for immutable objects.
- Changing (updating) a mutable object referenced through one variable or container also changes that object referenced through other variables or containers, because *it is the same object*.
- `del()` -- The built-in function `del()` removes a reference, not (necessarily) the object itself.

3 Statements and inspection -- preliminaries

`print` -- Example:

```
print obj
print "one", "two", 'three'
```

`for:` -- Example:

```
stuff = ['aa', 'bb', 'cc']
for item in stuff:
    print item
```

Learn what the *type* of an object is -- Example:

```
type(obj)
```

Learn what attributes an object has and what it's capabilities are -- Example:

```
dir(obj)
value = "a message"
dir(value)
```

Get help on a class or an object -- Example:

```
help(str)
help("")
value = "abc"
help(value)
help(value.upper)
```

In IPython (but not standard Python), you can also get help at the interactive prompt by typing "?" and "??" after an object. Example:

```
In [48]: a = ''
In [49]: a.upper?
Type:       builtin_function_or_method
String Form:<built-in method upper of str object at 0x7f1c426e0508>
Docstring:
S.upper() -> string

Return a copy of the string S converted to uppercase.
```

4 Built-in data-types

For information on built-in data types, see section Built-in Types -- http://docs.python.org/lib/types.html in the Python standard documentation.

4.1 Numeric types

The numeric types are:

- Plain integers -- Same precision as a C long, usually a 32-bit binary number.
- Long integers -- Define with `100L`. But, plain integers are automatically promoted when needed.
- Floats -- Implemented as a C double. Precision depends on your machine. See `sys.float_info`.
- Complex numbers -- Define with, for example, `3j` or `complex(3.0, 2.0)`.

See 2.3.4 Numeric Types -- int, float, long, complex -- http://docs.python.org/lib/typesnumeric.html.

Python does mixed arithmetic.

Integer division truncates. This is changed in Python 3. Use `float(n)` to force coercion to a float. Example:

```
In [8]: a = 4
In [9]: b = 5
In [10]: a / b
Out[10]: 0              # possibly wrong?
In [11]: float(a) / b
Out[11]: 0.8
```

Applying the function call operator (parentheses) to a type or class creates an instance of that type or class.

Scientific and heavily numeric programming -- High level Python is not very efficient for numerical programming. But, there are libraries that help -- Numpy and SciPy -- See: SciPy: Scientific Tools for Python -- http://scipy.org/

4.2 Tuples and lists

List -- A list is a dynamic array/sequence. It is ordered and indexable. A list is mutable.

List constructors: `[]`, `list()`.

`range()` and `xrange()`:

- `range(n)` creates a list of n integers. Optional arguments are the starting integer and a stride.
- `xrange` is like `range`, except that it creates an iterator that produces the items in the list of integers instead of the list itself.

Tuples -- A tuple is a sequence. A tuple is immutable.

Tuple constructors: `()`, but really a comma; also `tuple()`.

Tuples are like lists, but are not mutable.

Python lists are (1) heterogeneous (2) indexable, and (3) dynamic. For example, we can add to a list and make it longer.

Notes on sequence constructors:

- To construct a tuple with a single element, use `(x,)`; a tuple with a single element requires a comma.
- You can spread elements across multiple lines (and no need for backslash continuation character "\").
- A comma can follow the last element.

The length of a tuple or list (or other container): `len(mylist)`.

Operators for lists:

- Try: `list1 + list2`, `list1 * n`, `list1 += list2`, etc.
- Comparison operators: `<`, `==`, `>=`, etc.
- Test for membership with the `in` operator. Example:

```
In [77]: a = [11, 22, 33]
In [78]: a
Out[78]: [11, 22, 33]
In [79]: 22 in a
Out[79]: True
In [80]: 44 in a
Out[80]: False
```

Subscription:

- Indexing into a sequence
- Negative indexes -- Effectively, length of sequence plus (minus) index.
- Slicing -- Example: `data[2:5]`. Default values: beginning and end of list.
- Slicing with strides -- Example: `data[::2]`.

Operations on tuples -- No operations that change the tuple, since tuples are immutable. We can do iteration and subscription. We can do "contains" (the `in` operator) and get the length (the `len()` operator). We can use certain boolean operators.

Operations on lists -- Operations similar to tuples plus:

- Append -- `mylist.append(newitem)`.
- Insert -- `mylist.insert(index, newitem)`. Note on efficiency: The `insert` method is not as fast as the `append` method. If you find that you need to do a large number of `mylist.insert(0, obj)` (that is, inserting at the beginning of the list) consider using a deque instead. See: http://docs.python.org/2/library/collections.html#collections.deque. Or, use `append` and `reverse`.
- Extend -- `mylist.extend(anotherlist)`. Also can use `+` and `+=`.
- Remove -- `mylist.remove(item)` and `mylist.pop()`. Note that `append()` together with `pop()` implements a stack.
- Delete -- `del mylist[index]`.
- Pop -- Get last (right-most) item and remove from list -- `mylist.pop()`.

List operators -- `+`, `*`, etc.

For more operations and operators on sequences, see: http://docs.python.org/2/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange.

Exercises:

- Create an empty list. Append 4 strings to the list. Then pop one item off the end of the list. Solution:

```
In [25]: a = []
In [26]: a.append('aaa')
In [27]: a.append('bbb')
```

```
In [28]: a.append('ccc')
In [29]: a.append('ddd')
In [30]: print a
['aaa', 'bbb', 'ccc', 'ddd']
In [31]: a.pop()
Out[31]: 'ddd'
```

- Use the `for` statement to print the items in the list. Solution:

```
In [32]: for item in a:
   ....:         print item
   ....:
aaa
bbb
ccc
```

- Use the string `join` operation to concatenate the items in the list. Solution:

```
In [33]: '||'.join(a)
Out[33]: 'aaa||bbb||ccc'
```

- Use lists containing three (3) elements to create and show a tree:

```
In [37]: b = ['bb', None, None]
In [38]: c = ['cc', None, None]
In [39]: root = ['aa', b, c]
In [40]:
In [40]:
In [40]: def show_tree(t):
   ....:     if not t:
   ....:         return
   ....:     print t[0]
   ....:     show_tree(t[1])
   ....:     show_tree(t[2])
   ....:
   ....:
In [41]: show_tree(root)
aa
bb
cc
```

  Note that we will learn a better way to represent tree structures when we cover implementing classes in Python.

4.3 Strings

Strings are sequences. They are immutable. They are indexable. They are iterable.

For operations on strings, see http://docs.python.org/lib/string-methods.html or use:

```
>>> help(str)
```

Or:

```
>>> dir("abc")
```

String operations (methods).

String operators, e.g. `+`, `<`, `<=`, `==`, etc..

Constructors/literals:

- Quotes: single and double. Escaping quotes and other special characters with a back-slash.
- Triple quoting -- Use triple single quotes or double quotes to define multi-line strings.
- `str()` -- The constructor and the name of the type/class.
- `'aSeparator'.join(aList)`
- Many more.

Escape characters in strings -- `\t`, `\n`, `\\`, etc.

String formatting -- See: http://docs.python.org/2/library/stdtypes.html#string-formatting-operations

Examples:

```
In [18]: name = 'dave'
In [19]: size = 25
In [20]: factor = 3.45
In [21]: print 'Name: %s  Size: %d  Factor: %3.4f' % (name, size, factor,
)
Name: dave  Size: 25  Factor: 3.4500
In [25]: print 'Name: %s  Size: %d  Factor: %08.4f' % (name, size, factor,
)
Name: dave  Size: 25  Factor: 003.4500
```

If the right-hand argument to the formatting operator is a dictionary, then you can (actually, must) use the names of keys in the dictionary in your format strings. Examples:

```
In [115]: values = {'vegetable': 'chard', 'fruit': 'nectarine'}
In [116]: 'I love %(vegetable)s and I love %(fruit)s.' % values
Out[116]: 'I love chard and I love nectarine.'
```

Also consider using the right justify and left justify operations. Examples: `mystring.rjust(20)`, `mystring.ljust(20, ':')`.

In Python 3, the `str.format` method is preferred to the string formatting operator. This method is also available in Python 2.7. It has benefits and advantages over the string formatting operator. You can start learning about it here: http://docs.python.org/2/library/stdtypes.html#string-methods

Exercises:

- Use a literal to create a string containing (1) a single quote, (2) a double quote, (3) both a single and double quote. Solutions:

```
"Some 'quoted' text."
'Some "quoted" text.'
'Some "quoted" \'extra\' text.'
```

- Write a string literal that spans multiple lines. Solution:

```
"""This string
spans several lines
because it is a little long.
"""
```

- Use the string `join` operation to create a string that contains a colon as a separator. Solution:

```
>>> content = []
>>> content.append('finch')
>>> content.append('sparrow')
>>> content.append('thrush')
>>> content.append('jay')
>>> contentstr = ':'.join(content)
>>> print contentstr
finch:sparrow:thrush:jay
```

- Use string formatting to produce a string containing your last and first names, separated by a comma. Solution:

```
>>> first = 'Dave'
>>> last = 'Kuhlman'
>>> full = '%s, %s' % (last, first, )
>>> print full
Kuhlman, Dave
```

Incrementally building up large strings from lots of small strings -- **the old way** -- Since strings in Python are immutable, appending to a string requires a re-allocation. So, it is faster to append to a list, then use `join`. Example:

```
In [25]: strlist = []
In [26]: strlist.append('Line #1')
In [27]: strlist.append('Line #2')
In [28]: strlist.append('Line #3')
In [29]: str = '\n'.join(strlist)
In [30]: print str
Line #1
Line #2
Line #3
```

Incrementally building up large strings from lots of small strings -- **the new way** -- The `+=` operation on strings has been optimized. So, when you do this `str1 += str2`, even many times, it is efficient.

The `translate` method enables us to map the characters in a string, replacing those in one table by those in another. And, the `maketrans` function in the `string` module, makes it easy to create the mapping table:

```
import string

def test():
    a = 'axbycz'
    t = string.maketrans('abc', '123')
    print a
    print a.translate(t)

test()
```

4.3.1 The new string.format method

The new way to do string formatting (which is standard in Python 3 and *perhaps* preferred for new code in Python 2) is to use the `string.format` method. See here:

- http://docs.python.org/2/library/stdtypes.html#str.format
- http://docs.python.org/2/library/string.html#format-string-syntax
- http://docs.python.org/2/library/string.html#format-specification-mini-language

Some examples:

```
In [1]: 'aaa {1} bbb {0} ccc {1} ddd'.format('xx', 'yy', )
Out[1]: 'aaa yy bbb xx ccc yy ddd'
In [2]: 'number: {0:05d} ok'.format(25)
Out[2]: 'number: 00025 ok'
In [4]: 'n1: {num1}  n2: {num2}'.format(num2=25, num1=100)
Out[4]: 'n1: 100  n2: 25'
In [5]: 'n1: {num1}  n2: {num2}  again: {num1}'.format(num2=25, num1=100)
Out[5]: 'n1: 100  n2: 25  again: 100'
In [6]: 'number: {:05d} ok'.format(25)
Out[6]: 'number: 00025 ok'
In [7]: values = {'name': 'dave', 'hobby': 'birding'}
In [8]: 'user: {name}  activity: {hobby}'.format(**values)
Out[8]: 'user: dave  activity: birding'
```

4.3.2 Unicode strings

Representing unicode:

```
In [96]: a = u'abcd'
In [97]: a
Out[97]: u'abcd'
In [98]: b = unicode('efgh')
In [99]: b
Out[99]: u'efgh'
```

Convert to unicode: `a_string.decode(encoding)`. Examples:

```
In [102]: 'abcd'.decode('utf-8')
Out[102]: u'abcd'
In [103]:
In [104]: 'abcd'.decode(sys.getdefaultencoding())
Out[104]: u'abcd'
```

Convert out of unicode: `a_unicode_string.encode(encoding)`. Examples:

```
In [107]: a = u'abcd'
In [108]: a.encode('utf-8')
Out[108]: 'abcd'
In [109]: a.encode(sys.getdefaultencoding())
Out[109]: 'abcd'
In [110]: b = u'Sel\xe7uk'
In [111]: print b.encode('utf-8')
Selçuk
```

Test for unicode type -- Example:

```
In [122]: import types
In [123]: a = u'abcd'
In [124]: type(a) is types.UnicodeType
Out[124]: True
In [125]:
In [126]: type(a) is type(u'')
Out[126]: True
```

Or better:

```
In [127]: isinstance(a, unicode)
Out[127]: True
```

An example with a character "c" with a hachek:

```
In [135]: name = 'Ivan Krsti\xc4\x87'
In [136]: name.decode('utf-8')
Out[136]: u'Ivan Krsti\u0107'
In [137]:
In [138]: len(name)
Out[138]: 12
In [139]: len(name.decode('utf-8'))
Out[139]: 11
```

You can also create a unicode character by using the `unichr()` built-in function:

```
In [2]: a = 'aa' + unichr(170) + 'bb'
In [3]: a
Out[3]: u'aa\xaabb'
In [6]: b = a.encode('utf-8')
In [7]: b
Out[7]: 'aa\xc2\xaabb'
In [8]: print b
aaªbb
```

Guidance for use of encodings and unicode -- If you are working with a multibyte character set:

1. Convert/decode from an external encoding to unicode *early* (`my_string.decode(encoding)`).
2. Do your work in unicode.
3. Convert/encode to an external encoding *late* (`my_string.encode(encoding)`).

For more information, see:

- Unicode In Python, Completely Demystified -- http://farmdev.com/talks/unicode/
- PEP 100: Python Unicode Integration -- http://www.python.org/dev/peps/pep-0100/
- In the Python standard library:
  - codecs -- Codec registry and base classes -- http://docs.python.org/2/library/codecs.html#module-codecs
  - Standard Encodings -- http://docs.python.org/2/library/codecs.html#standard-encodings

If you are reading and writing multibyte character data from or to a *file*, then look at the `codecs.open()` in the codecs module -- http://docs.python.org/2/library/codecs.html#codecs.open.

Handling multi-byte character sets in Python 3 is easier, I think, but different. One hint is to

use the `encoding` keyword parameter to the `open` built-in function. Here is an example:

```
def test():
    infile = open('infile1.txt', 'r', encoding='utf-8')
    outfile = open('outfile1.txt', 'w', encoding='utf-8')
    for line in infile:
        line = line.upper()
        outfile.write(line)
    infile.close()
    outfile.close()

test()
```

4.4 Dictionaries

A dictionary is a collection, whose values are accessible by key. It is a collection of name-value pairs.

The order of elements in a dictionary is undefined. But, we can iterate over (1) the keys, (2) the values, and (3) the items (key-value pairs) in a dictionary. We can set the value of a key and we can get the value associated with a key.

Keys must be immutable objects: ints, strings, tuples, ...

Literals for constructing dictionaries:

```
d1 = {}
d2 = {key1: value1, key2: value2, }
```

Constructor for dictionaries -- `dict()` can be used to create instances of the class `dict`. Some examples:

```
dict({'one': 2, 'two': 3})
dict({'one': 2, 'two': 3}.items())
dict({'one': 2, 'two': 3}.iteritems())
dict(zip(('one', 'two'), (2, 3)))
dict([['two', 3], ['one', 2]])
dict(one=2, two=3)
dict([(['one', 'two'][i-2], i) for i in (2, 3)])
```

For operations on dictionaries, see http://docs.python.org/lib/typesmapping.html or use:

```
>>> help({})
```

Or:

```
>>> dir({})
```

Indexing -- Access or add items to a dictionary with the indexing operator `[ ]`. Example:

```
In [102]: dict1 = {}
In [103]: dict1['name'] = 'dave'
In [104]: dict1['category'] = 38
In [105]: dict1
Out[105]: {'category': 38, 'name': 'dave'}
```

Some of the operations produce the keys, the values, and the items (pairs) in a dictionary. Examples:

```
In [43]: d = {'aa': 111, 'bb': 222}
In [44]: d.keys()
Out[44]: ['aa', 'bb']
In [45]: d.values()
Out[45]: [111, 222]
In [46]: d.items()
Out[46]: [('aa', 111), ('bb', 222)]
```

When iterating over large dictionaries, use methods `iterkeys()`, `itervalues()`, and `iteritems()`. Example:

```
In [47]:
In [47]: d = {'aa': 111, 'bb': 222}
In [48]: for key in d.iterkeys():
   ....:     print key
   ....:
   ....:
aa
bb
```

To test for the existence of a key in a dictionary, use the `in` operator or the `mydict.has_key(k)` method. The `in` operator is preferred. Example:

```
>>> d = {'tomato': 101, 'cucumber': 102}
>>> k = 'tomato'
>>> k in d
True
>>> d.has_key(k)
True
```

You can often avoid the need for a test by using method `get`. Example:

```
>>> d = {'tomato': 101, 'cucumber': 102}
>>> d.get('tomato', -1)
101
>>> d.get('chard', -1)
-1
>>> if d.get('eggplant') is None:
...     print 'missing'
...
missing
```

Dictionary "view" objects provide dynamic (automatically updated) views of the keys or the values or the items in a dictionary. View objects also support set operations. Create views with `mydict.viewkeys()`, `mydict.viewvalues()`, and `mydict.viewitems()`. See: http://docs.python.org/2/library/stdtypes.html#dictionary-view-objects.

The dictionary `setdefault` method provides a way to get the value associated with a key from a dictionary and to set that value if the key is missing. Example:

```
In [106]: a
Out[106]: {}
In [108]: a.setdefault('cc', 33)
```

```
Out[108]: 33
In [109]: a
Out[109]: {'cc': 33}
In [110]: a.setdefault('cc', 44)
Out[110]: 33
In [111]: a
Out[111]: {'cc': 33}
```

Exercises:

- Write a literal that defines a dictionary using both string literals and variables containing strings. Solution:

```
>>> first = 'Dave'
>>> last = 'Kuhlman'
>>> name_dict = {first: last, 'Elvis': 'Presley'}
>>> print name_dict
{'Dave': 'Kuhlman', 'Elvis': 'Presley'}
```

- Write statements that iterate over (1) the keys, (2) the values, and (3) the items in a dictionary. (Note: Requires introduction of the `for` statement.) Solutions:

```
>>> d = {'aa': 111, 'bb': 222, 'cc': 333}
>>> for key in d.keys():
...   print key
...
aa
cc
bb
>>> for value in d.values():
...   print value
...
111
333
222
>>> for item in d.items():
...   print item
...
('aa', 111)
('cc', 333)
('bb', 222)
>>> for key, value in d.items():
...   print key, '::', value
...
aa :: 111
cc :: 333
bb :: 222
```

Additional notes on dictionaries:

- You can use `iterkeys()`, `itervalues()`, `iteritems()` to obtain iterators over keys, values, and items.
- A dictionary itself is iterable: it iterates over its keys. So, the following two lines are equivalent:

```
for k in myDict: print k
for k in myDict.iterkeys(): print k
```

The `in` operator tests for a key in a dictionary. Example:

```
In [52]: mydict = {'peach': 'sweet', 'lemon': 'tangy'}
In [53]: key = 'peach'
In [54]: if key in mydict:
   ....:     print mydict[key]
   ....:
sweet
```

4.5 Files

Open a file with the `open` factory method. Example:

```
In [28]: f = open('mylog.txt', 'w')
In [29]: f.write('message #1\n')
In [30]: f.write('message #2\n')
In [31]: f.write('message #3\n')
In [32]: f.close()
In [33]: f = file('mylog.txt', 'r')
In [34]: for line in f:
   ....:     print line,
   ....:
message #1
message #2
message #3
In [35]: f.close()
```

Notes:

- Use the (built-in) `open(path, mode)` function to open a file and create a file object. You could also use `file()`, but `open()` is recommended.
- A file object that is open for reading a text file supports the iterator protocol and, therefore, can be used in a `for` statement. It iterates over the *lines* in the file. This is most likely only useful for text files.
- `open` is a factory method that creates file objects. Use it to open files for reading, writing, and appending. Examples:

```
infile = open('myfile.txt', 'r')    # open for reading
outfile = open('myfile.txt', 'w')   # open for (over-) writing
log = open('myfile.txt', 'a')       # open for appending to existing
content
```

- When you have finished with a file, close it. Examples:

```
infile.close()
outfile.close()
```

- You can also use the `with:` statement to automatically close the file. Example:

```
with open('tmp01.txt', 'r') as infile:
    for x in infile:
        print x,
```

The above works because a file is a context manager: it obeys the context manager protocol. A file has methods `__enter__` and `__exit__`, and the `__exit__` method automatically closes the file for us. See the section on the with: statement.

- In order to open multiple files, you can nest `with:` statements, or use a single `with:` statement with multiple "expression as target" clauses. Example:

```
def test():
    #
    # use multiple nested with: statements.
    with open('small_file.txt', 'r') as infile:
        with open('tmp_outfile.txt', 'w') as outfile:
            for line in infile:
                outfile.write('line: %s' % line.upper())
    print infile
    print outfile
    #
    # use a single with: statement.
    with open('small_file.txt', 'r') as infile, \
            open('tmp_outfile.txt', 'w') as outfile:
        for line in infile:
            outfile.write('line: %s' % line.upper())
    print infile
    print outfile

test()
```

- `file` is the file type and can be used as a constructor to create file objects. *But*, `open` is preferred.
- Lines read from a text file have a newline. Strip it off with something like: `line.rstrip('\n')`.
- For binary files you should add the binary mode, for example: `rb`, `wb`. For more about modes, see the description of the `open()` function at Built-in Functions -- http://docs.python.org/lib/built-in-funcs.html.
- Learn more about file objects and the methods they provide at: 2.3.9 File Objects -- http://docs.python.org/2/library/stdtypes.html#file-objects.

You can also append to an existing file. Note the "a" mode in the following example:

```
In [39]: f = open('mylog.txt', 'a')
In [40]: f.write('message #4\n')
In [41]: f.close()
In [42]: f = file('mylog.txt', 'r')
In [43]: for line in f:
   ....:     print line,
   ....:
message #1
message #2
message #3
message #4
In [44]: f.close()
```

For binary files, add "b" to the mode. Not strictly necessary on UNIX, but needed on MS Windows. And, you will want to make your code portable across platforms. Example:

```
In [62]: import zipfile
In [63]: outfile = open('tmp1.zip', 'wb')
In [64]: zfile = zipfile.ZipFile(outfile, 'w', zipfile.ZIP_DEFLATED)
In [65]: zfile.writestr('entry1', 'my heroes have always been cowboys')
In [66]: zfile.writestr('entry2', 'and they still are it seems')
In [67]: zfile.writestr('entry3', 'sadly in search of and')
In [68]: zfile.writestr('entry4', 'on step in back of')
In [69]:
In [70]: zfile.writestr('entry4', 'one step in back of')
In [71]: zfile.writestr('entry5', 'themselves and their slow moving ways')
In [72]: zfile.close()
In [73]: outfile.close()
In [75]:
$
$ unzip -lv tmp1.zip
Archive:  tmp1.zip
 Length   Method    Size  Ratio   Date   Time   CRC-32    Name
--------  ------  ------- -----   ----   ----   ------    ----
      34  Defl:N       36  -6%  05-29-08 17:04  f6b7d921  entry1
      27  Defl:N       29  -7%  05-29-08 17:07  10da8f3d  entry2
      22  Defl:N       24  -9%  05-29-08 17:07  3fd17fda  entry3
      18  Defl:N       20 -11%  05-29-08 17:08  d55182e6  entry4
      19  Defl:N       21 -11%  05-29-08 17:08  1a892acd  entry4
      37  Defl:N       39  -5%  05-29-08 17:09  e213708c  entry5
--------          -------  ---                            -------
     157              169  -8%                            6 files
```

Exercises:

- Read all of the lines of a file into a list. Print the 3rd and 5th lines in the file/list.
  Solution:

```
In [55]: f = open('tmp1.txt', 'r')
In [56]: lines = f.readlines()
In [57]: f.close()
In [58]: lines
Out[58]: ['the\n', 'big\n', 'brown\n', 'dog\n', 'had\n', 'long\n',
'hair\n']
In [59]: print lines[2]
brown

In [61]: print lines[4]
had
```

More notes:

- Strip newlines (and other whitespace) from a string with methods `strip()`, `lstrip()`, and `rstrip()`.
- Get the current position within a file by using `myfile.tell()`.
- Set the current position within a file by using `myfile.seek()`. It may be helpful to use `os.SEEK_CUR` and `os.SEEK_END`. For example:
  - `f.seek(2, os.SEEK_CUR)` advances the position by two
  - `f.seek(-3, os.SEEK_END)` sets the position to the third to last.
  - `f.seek(25)` sets the position relative to the beginning of the file.

4.6 Other built-in types

Other built-in data types are described in section Built-in Types -- http://docs.python.org/lib/types.html in the Python standard documentation.

4.6.1 The None value/type

The unique value `None` is used to indicate "no value", "nothing", "non-existence", etc. There is only one `None` value; in other words, it's a singleton.

Use `is` to test for `None`. Example:

```
>>> flag = None
>>>
>>> if flag is None:
...     print 'clear'
...
clear
>>> if flag is not None:
...     print 'hello'
...
>>>
```

4.6.2 Boolean values

`True` and `False` are the boolean values.

The following values also count as false, for example, in an `if:` statement: `False`, numeric zero, `None`, the empty string, an empty list, an empty dictionary, any empty container, etc. All other values, including `True`, act as true values.

4.6.3 Sets and frozensets

A set is an unordered collection of immutable objects. A set does not contain duplicates.

Sets support several set operations, for example: union, intersection, difference, ...

A frozenset is like a set, except that a frozenset is immutable. Therefore, a frozenset is hash-able and can be used as a key in a dictionary, and it can be added to a set.

Create a set with the set constructor. Examples:

```
>>> a = set()
>>> a
set([])
>>> a.add('aa')
>>> a.add('bb')
>>> a
set(['aa', 'bb'])
>>> b = set([11, 22])
>>> b
set([11, 22])
>>> c = set([22, 33])
>>> b.union(c)
set([33, 11, 22])
>>> b.intersection(c)
set([22])
```

For more information on sets, see: Set Types -- set, frozenset -- http://docs.python.org/lib/types-set.html

5 Functions and Classes -- A Preview

Structured code -- Python programs are made up of expressions, statements, functions, classes, modules, and packages.

Python objects are first-class objects.

Expressions are evaluated.

Statements are executed.

Functions (1) are objects and (2) are callable.

Object-oriented programming in Python. Modeling "real world" objects. (1) Encapsulation; (2) data hiding; (3) inheritance. Polymorphism.

Classes -- (1) encapsulation; (2) data hiding; (3) inheritance.

An overview of the structure of a typical class: (1) methods; (2) the constructor; (3) class (static) variables; (4) super/subclasses.

6 Statements

6.1 Assignment statement

Form -- `target = expression`.

Possible targets:

- Identifier
- Tuple or list -- Can be nested. Left and right sides must have equivalent structure. Example:

```
>>> x, y, z = 11, 22, 33
>>> [x, y, z] = 111, 222, 333
>>> a, (b, c) = 11, (22, 33)
>>> a, B = 11, (22, 33)
```

  This feature can be used to simulate an enum:

```
In [22]: LITTLE, MEDIUM, LARGE = range(1, 4)
In [23]: LITTLE
Out[23]: 1
In [24]: MEDIUM
Out[24]: 2
```

- Subscription of a sequence, dictionary, etc. Example:

```
In [10]: a = range(10)
In [11]: a
Out[11]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [12]: a[3] = 'abc'
```

```
In [13]: a
Out[13]: [0, 1, 2, 'abc', 4, 5, 6, 7, 8, 9]
In [14]:
In [14]: b = {'aa': 11, 'bb': 22}
In [15]: b
Out[15]: {'aa': 11, 'bb': 22}
In [16]: b['bb'] = 1000
In [17]: b['cc'] = 2000
In [18]: b
Out[18]: {'aa': 11, 'bb': 1000, 'cc': 2000}
```

- A slice of a sequence -- Note that the sequence must be mutable. Example:

```
In [1]: a = range(10)
In [2]: a
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [3]: a[2:5] = [11, 22, 33, 44, 55, 66]
In [4]: a
Out[4]: [0, 1, 11, 22, 33, 44, 55, 66, 5, 6, 7, 8, 9]
```

- Attribute reference -- Example:

```
>>> class MyClass:
...     pass
...
>>> anObj = MyClass()
>>> anObj.desc = 'pretty'
>>> print anObj.desc
pretty
```

There is also augmented assignment. Examples:

```
>>> index = 0
>>> index += 1
>>> index += 5
>>> index += f(x)
>>> index -= 1
>>> index *= 3
```

Things to note:

- Assignment to a name creates a new variable (if it does not exist in the namespace) and a binding. Specifically, it binds a value to the new name. Calling a function also does this to the (formal) parameters within the local namespace.
- In Python, a language with dynamic typing, the data type is associated with the value, not the variable, as is the case in statically typed languages.
- Assignment can also cause sharing of an object. Example:

```
obj1 = A()
obj2 = obj1
```

Check to determine that the same object is shared with `id(obj)` or the `is` operator. Example:

```
In [23]: a = range(10)
```

```
In [24]: a
Out[24]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [25]: b = a
In [26]: b
Out[26]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [27]: b[3] = 333
In [28]: b
Out[28]: [0, 1, 2, 333, 4, 5, 6, 7, 8, 9]
In [29]: a
Out[29]: [0, 1, 2, 333, 4, 5, 6, 7, 8, 9]
In [30]: a is b
Out[30]: True
In [31]: print id(a), id(b)
31037920 31037920
```

- You can also do multiple assignment in a single statement. Example:

```
In [32]: a = b = 123
In [33]: a
Out[33]: 123
In [34]: b
Out[34]: 123
In [35]:
In [35]:
In [35]: a = b = [11, 22]
In [36]: a is b
Out[36]: True
```

- You can interchange (swap) the value of two variables using assignment and packing/unpacking:

```
>>> a = 111
>>> b = 222
>>> a, b = b, a
>>> a
222
>>> b
111
```

6.2 import statement

Make module (or objects in the module) available.

What `import` does:

- Evaluate the content of a module.
- Likely to create variables in the local (module) namespace.
- Evaluation of a specific module only happens once during a given run of the program. Therefore, a module is shared across an application.
- A module is evaluated from top to bottom. Later statements can replace values created earlier. This is true of functions and classes, as well as (other) variables.
- Which statements are evaluated? Assignment, `class`, `def`, ...
- Use the following idiom to make a module both run-able and import-able:

```
if __name__ == '__main__':
    # import pdb; pdb.set_trace()
    main()        # or "test()" or some other function defined in
module
```

Notes:

- The above condition will be true *only when* the module is run as a script and will *not* be true when the module is imported.
- The line containing `pdb` can be copied any place in your program and un-commented, and then the program will drop into the Python debugger when that location is reached.

Where `import` looks for modules:

- `sys.path` shows where it looks.
- There are some standard places.
- Add additional directories by setting the environment variable `PYTHONPATH`.
- You can also add paths by modifying `sys.path`, for example:

```
import sys
sys.path.insert(0, '/path/to/my/module')
```

- Packages need a file named `__init__.py`.
- Extensions -- To determine what extensions import looks for, do:

```
>>> import imp
>>> imp.get_suffixes()
[('.so', 'rb', 3), ('module.so', 'rb', 3), ('.py', 'U', 1), ('.pyc',
'rb', 2)]
```

Forms of the `import` statement:

- `import A` -- Names in the local (module) namespace are accessible with the dot operator.
- `import A as B` -- Import the module A, but bind the module object to the variable B.
- `import A1, A2` -- Not recommended
- `from A import B`
- `from A import B1, B2`
- `from A import B as C`
- `from A import *` -- Not recommended: clutters and mixes name-spaces.
- `from A.B import C` -- (1) Possibly import object `C` from module `B` in package `A` or (2) possibly import module `C` from sub-package `B` in package `A`.
- `import A.B.C` -- To reference attributes in `C`, must use fully-qualified name, for example use `A.B.C.D` to reference `D` inside of `C`.

More notes on the `import` statement:

- The import statement and packages -- A file named `__init__.py` is required in a package. This file is evaluated the first time either the package is imported or a file in the package is imported. Question: What is made available when you do `import aPackage`?

Answer: All variables (names) that are global inside the `__init__.py` module in that package. But, see notes on the use of `__all__`: The import statement -- http://docs.python.org/ref/import.html

- The use of `if __name__ == "__main__":` -- Makes a module both import-able and executable.
- Using dots in the import statement -- From the Python language reference manual:

> "Hierarchical module names:when the module names contains one or more dots, the module search path is carried out differently. The sequence of identifiers up to the last dot is used to find a `package`; the final identifier is then searched inside the package. A package is generally a subdirectory of a directory on sys.path that has a file __init__.py."

See: The import statement -- http://docs.python.org/ref/import.html

Exercises:

- Import a module from the standard library, for example `re`.
- Import an element from a module from the standard library, for example import `compile` from the `re` module.
- Create a simple Python package with a single module in it. Solution:
  1. Create a directory named `simplepackage` in the current directory.
  2. Create an (empty) `__init__.py` in the new directory.
  3. Create an `simple.py` in the new directory.
  4. Add a simple function name `test1` in `simple.py`.
  5. Import using any of the following:

```
>>> import simplepackage.simple
>>> from simplepackage import simple
>>> from simplepackage.simple import test1
>>> from simplepackage.simple import test1 as mytest
```

6.3 print statement

`print` sends output to `sys.stdout`. It adds a newline, unless an extra comma is added.

Arguments to `print`:

- Multiple items -- Separated by commas.
- End with comma to suppress carriage return.
- Use string formatting for more control over output.
- Also see various "pretty-printing" functions and methods, in particular, `pprint`. See 3.27

pprint -- Data pretty printer -- http://docs.python.org/lib/module-pprint.html.

String formatting -- Arguments are a tuple. Reference: 2.3.6.2 String Formatting Operations -- http://docs.python.org/lib/typesseq-strings.html.

Can also use `sys.stdout`. Note that a carriage return is *not* automatically added. Example:

```
>>> import sys
>>> sys.stdout.write('hello\n')
```

Controlling the destination and format of print -- Replace `sys.stdout` with an instance of any class that implements the method `write` taking one parameter. Example:

```
import sys

class Writer:
    def __init__(self, file_name):
        self.out_file = file(file_name, 'a')
    def write(self, msg):
        self.out_file.write('[[%s]]' % msg)
    def close(self):
        self.out_file.close()

def test():
    writer = Writer('outputfile.txt')
    save_stdout = sys.stdout
    sys.stdout = writer
    print 'hello'
    print 'goodbye'
    writer.close()
    # Show the output.
    tmp_file = file('outputfile.txt')
    sys.stdout = save_stdout
    content = tmp_file.read()
    tmp_file.close()
    print content

test()
```

There is an alternative form of the `print` statement that takes a file-like object, in particular an object that has a `write` method. For example:

```
In [1]: outfile = open('tmp.log', 'w')
In [2]: print >> outfile, 'Message #1'
In [3]: print >> outfile, 'Message #2'
In [4]: print >> outfile, 'Message #3'
In [5]: outfile.close()
In [6]:
In [6]: infile = open('tmp.log', 'r')
In [7]: for line in infile:
   ...:     print 'Line:', line.rstrip('\n')
   ...:
Line: Message #1
Line: Message #2
Line: Message #3
In [8]: infile.close()
```

Future deprecation warning -- There is no print *statement* in Python 3. There is a print built-in *function*.

6.4 if: elif: else: statement

A template for the `if:` statement:

```
if condition1:
    statements
elif condition2:
    statements
elif condition3:
    statements
else:
    statements
```

The `elif` and `else` clauses are optional.

Conditions -- Expressions -- Anything that returns a value. Compare with `eval()` and `exec`.

Truth values:

- False -- `False`, `None`, numeric zero, the empty string, an empty collection (list or tuple or dictionary or ...).
- True -- `True` and everything else.

Operators:

- `and` and `or` -- Note that both `and` and `or` do short circuit evaluation.
- `not`
- `is` and `is not` -- The identical object. Cf. `a is b` and `id(a) == id(b)`. Useful to test for `None`, for example:

  ```
  if x is None:
      ...
  if x is not None:
      ...
  ```

- `in` and `not in` -- Can be used to test for existence of a key in a dictionary or for the presence of a value in a collection.

  The `in` operator tests for equality, not identity.

  Example:

  ```
  >>> d = {'aa': 111, 'bb': 222}
  >>> 'aa' in d
  True
  >>> 'aa' not in d
  False
  >>> 'xx' in d
  False
  ```

- Comparison operators, for example `==`, `!=`, `<`, `<=`, ...

There is an `if` expression. Example:

```
>>> a = 'aa'
>>> b = 'bb'
>>> x = 'yes' if a == b else 'no'
>>> x
'no'
```

Notes:

- The `elif:` clauses and the `else:` clause are optional.
- The `if:`, `elif:`, and `else:` clauses are all header lines in the sense that they are each followed by an indented block of code and each of these header lines ends with a colon. (To put an empty block after one of these, or any other, statement header line, use the `pass` statement. It's effectively a no-op.)
- Parentheses around the condition in an `if:` or `elif:` are not required and are considered bad form, unless the condition extends over multiple lines, in which case parentheses are preferred over use of a line continuation character (backslash at the end of the line).

Exercises:

- Write an `if` statement with an `and` operator.
- Write an `if` statement with an `or` operator.
- Write an `if` statement containing both `and` and `or` operators.

6.5 for: statement

Iterate over a sequence or an "iterable" object.

Form:

```
for x in y:
    block
```

Iterator -- Some notes on what it means to be iterable:

- An iterable is something that can be used in an iterator context, for example, in a `for:` statement, in a list comprehension, and in a generator expression.
- Sequences and containers are iterable. Examples: tuples, lists, strings, dictionaries.
- Instances of classes that obey the iterator protocol are iterable. See http://docs.python.org/lib/typeiter.html.
- We can create an iterator object with built-in functions such as `iter()` and `enumerate()`. See Built-in Functions -- http://docs.python.org/lib/built-in-funcs.html in the Python standard library reference.
- Functions that use the `yield` statement, produce an iterator, although it's actually called a generator.
- An iterable implements the iterator interface and satisfies the iterator protocol. The iterator protocol: `__iter__()` and `next()` methods. See 2.3.5 Iterator Types -- (http://docs.python.org/lib/typeiter.html).

Testing for "iterability":

- If you can use an object in a `for:` statement, it's iterable.
- If the expresion `iter(obj)` does not produce a `TypeError` exception, it's iterable.

Some ways to produce iterators:

- `iter()` and `enumerate()` -- See: http://docs.python.org/lib/built-in-funcs.html.

- `some_dict.iterkeys()`, `some_dict.itervalues()`, `some_dict.iteritems()`.
- Use a sequence in an iterator context, for example in a `for` statement. Lists, tuples, dictionaries, and strings can be used in an iterator context to produce an iterator.
- Generator expressions -- Latest Python only. Syntactically like list comprehensions, but (1) surrounded by parentheses instead of square brackets and (2) use lazy evaluation.
- A class that implements the iterator protocol -- Example:

```
class A(object):
    def __init__(self):
        self.data = [11,22,33]
        self.idx = 0
    def __iter__(self):
        return self
    def next(self):
        if self.idx < len(self.data):
            x = self.data[self.idx]
            self.idx +=1
            return x
        else:
            raise StopIteration

def test():
    a = A()
    for x in a:
        print x

test()
```

  Note that the iterator protocol changes in Python 3.

- A function containing a yield statement. See:
  - Yield expressions -- http://docs.python.org/2/reference/expressions.html#yield-expressions
  - The yield statement -- http://docs.python.org/2/reference/simple_stmts.html#the-yield-statement
- Also see `itertools` module in the Python standard library for much more help with iterators: itertools — Functions creating iterators for efficient looping -- http://docs.python.org/2/library/itertools.html#module-itertools

The `for:` statement can also do unpacking. Example:

```
In [25]: items = ['apple', 'banana', 'cherry', 'date']
In [26]: for idx, item in enumerate(items):
   ....:     print '%d.  %s' % (idx, item, )
   ....:
0.  apple
1.  banana
2.  cherry
3.  date
```

The `for` statement can also have an optional `else:` clause. The `else:` clause is executed if the `for` statement completes normally, that is if a `break` statement is *not* executed.

Helpful functions with `for`:

- `enumerate(iterable)` -- Returns an iterable that produces pairs (tuples) containing count (index) and value. Example:

```
>>> for idx, value in enumerate([11,22,33]):
...     print idx, value
...
0 11
1 22
2 33
```

- `range([start,] stop[, step])` and `xrange([start,] stop[, step])`.

List comprehensions -- Since list comprehensions create lists, they are useful in `for` statements, although, when the number of elements is large, you should consider using a generator expression instead. A list comprehension looks a bit like a `for:` statement, but is inside square brackets, and it is an expression, not a statement. Two forms (among others):

- `[f(x) for x in iterable]`
- `[f(x) for x in iterable if t(x)]`

Generator expressions -- A generator expression looks similar to a list comprehension, except that it is surrounded by parentheses rather than square brackets. Example:

```
In [28]: items = ['apple', 'banana', 'cherry', 'date']
In [29]: gen1 = (item.upper() for item in items)
In [30]: for x in gen1:
   ....:     print 'x:', x
   ....:
x: APPLE
x: BANANA
x: CHERRY
x: DATE
```

Exercises:

- Write a list comprehension that returns all the keys in a dictionary whose associated values are greater than zero.
  - The dictionary: `{'aa': 11, 'cc': 33, 'dd': -55, 'bb': 22}`
  - Solution: `[x[0] for x in my_dict.iteritems() if x[1] > 0]`
- Write a list comprehension that produces even integers from 0 to 10. Use a `for` statement to iterate over those values. Solution:

```
for x in [y for y in range(10) if y % 2 == 0]:
    print 'x: %s' % x
```

- Write a list comprehension that iterates over two lists and produces all the combinations of items from the lists. Solution:

```
In [19]: a = range(4)
In [20]: b = [11,22,33]
In [21]: a
```

```
Out[21]: [0, 1, 2, 3]
In [22]: b
Out[22]: [11, 22, 33]
In [23]: c = [(x, y) for x in a for y in b]
In [24]: print c
[(0, 11), (0, 22), (0, 33), (1, 11), (1, 22), (1, 33),
(2, 11), (2, 22), (2, 33), (3, 11), (3, 22), (3, 33)]
```

But, note that in the previous exercise, a generator expression would often be better. A generator expression is like a list comprehension, except that, instead of creating the entire list, it produces a generator that can be used to produce each of the elements.

The `break` and `continue` statements are often useful in a `for` statement. See continue and break statements

The `for` statement can also have an optional `else:` clause. The `else:` clause is executed if the `for` statement completes normally, that is if a `break` statement is *not* executed. Example:

```
for item in data1:
    if item > 100:
        value1 = item
        break
else:
    value1 = 'not found'
print 'value1:', value1
```

When run, it prints:

```
value1: not found
```

6.6 while: statement

Form:

```
while condition:
    block
```

The `while:` statement is not often used in Python because the `for:` statement is usually more convenient, more idiomatic, and more Pythonic.

Exercises:

- Write a `while` statement that prints integers from zero to 5. Solution:

```
count = 0
while count < 5:
    count += 1
    print count
```

The `break` and `continue` statements are often useful in a `while` statement. See continue and break statements

The `while` statement can also have an optional `else:` clause. The `else:` clause is executed if

the `while` statement completes normally, that is if a `break` statement is *not* executed.

6.7 continue and break statements

The `break` statement exits from a loop.

The `continue` statement causes execution to immediately continue at the start of the loop.

Can be used in `for:` and `while:`.

When the `for:` statement or the `while:` statement has an `else:` clause, the block in the `else:` clause is executed only if a `break` statement is *not* executed.

Exercises:

- Using `break`, write a `while` statement that prints integers from zero to 5. Solution:

```
count = 0
while True:
    count += 1
    if count > 5:
        break
    print count
```

  Notes:

  - A `for` statement that uses `range()` or `xrange()` would be better than a `while` statement for this use.

- Using `continue`, write a `while` statement that processes only even integers from 0 to 10. Note: `%` is the modulo operator. Solution:

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print count
```

6.8 try: except: statement

Exceptions are a systematic and consistent way of processing errors and "unusual" events in Python.

Caught and un-caught exceptions -- Uncaught exceptions terminate a program.

The `try:` statement catches an exception.

Almost all errors in Python are exceptions.

Evaluation (execution model) of the `try` statement -- When an exception occurs in the `try` block, even if inside a nested function call, execution of the `try` block ends and the `except` clauses are searched for a matching exception.

Tracebacks -- Also see the `traceback` module: http://docs.python.org/lib/module-traceback.html

Exceptions are classes.

Exception classes -- subclassing, args.

An exception class in an `except:` clause catches instances of that exception class and all subclasses, but *not* superclasses.

Built-in exception classes -- See:

- Module `exceptions`.
- Built-in exceptions -- http://docs.python.org/lib/module-exceptions.html.

User defined exception classes -- subclasses of `Exception`.

Example:

```
try:
    raise RuntimeError('this silly error')
except RuntimeError, exp:
    print "[[[%s]]]" % exp
```

Reference: http://docs.python.org/lib/module-exceptions.html

You can also get the arguments passed to the constructor of an exception object. In the above example, these would be:

```
exp.args
```

Why would you define your own exception class? One answer: You want a user of your code to catch your exception and no others.

Catching an exception by exception class catches exceptions of that class and all its subclasses. So:

```
except SomeExceptionClass, exp:
```

matches and catches an exception if SomeExceptionClass is the exception class or a base class (superclass) of the exception class. The exception object (usually an instance of some exception class) is bound to `exp`.

A more "modern" syntax is:

```
except SomeExceptionClass as exp:
```

So:

```
class MyE(ValueError):
    pass
```

```
try:
    raise MyE()
except ValueError:
    print 'caught exception'
```

will print "caught exception", because `ValueError` is a base class of `MyE`.

Also see the entries for "EAFP" and "LBYL" in the Python glossary: http://docs.python.org/3/glossary.html.

Exercises:

- Write a *very* simple, empty exception subclass. Solution:

```
class MyE(Exception):
    pass
```

- Write a `try:except:` statement that raises your exception and also catches it. Solution:

```
try:
    raise MyE('hello there dave')
except MyE, e:
    print e
```

6.9 raise statement

Throw or raise an exception.

Forms:

- `raise instance`
- `raise MyExceptionClass(value)` -- preferred.
- `raise MyExceptionClass, value`

The `raise` statement takes:

- An (instance of) a built-in exception class.
- An instance of class `Exception` or
- An instance of a built-in subclass of class `Exception` or
- An instance of a user-defined subclass of class `Exception` or
- One of the above classes and (optionally) a value (for example, a string or a tuple).

See http://docs.python.org/ref/raise.html.

For a list of built-in exceptions, see http://docs.python.org/lib/module-exceptions.html.

The following example defines an exception subclass and throws an instance of that subclass. It also shows how to pass and catch multiple arguments to the exception:

```
class NotsobadError(Exception):
    pass

def test(x):
    try:
        if x == 0:
```

```
            raise NotsobadError('a moderately bad error', 'not too bad')
    except NotsobadError, e:
        print 'Error args: %s' % (e.args, )

test(0)
```

Which prints out the following:

```
Error args: ('a moderately bad error', 'not too bad')
```

Notes:

- In order to pass in multiple arguments with the exception, we use a tuple, or we pass multiple arguments to the constructor.

The following example does a small amount of processing of the arguments:

```
class NotsobadError(Exception):
    """An exception class.
    """
    def get_args(self):
        return '::::'.join(self.args)

def test(x):
    try:
        if x == 0:
            raise NotsobadError('a moderately bad error', 'not too bad')
    except NotsobadError, e:
        print 'Error args: {{{%s}}}' % (e.get_args(), )

test(0)
```

6.10 with: statement

The `with` statement enables us to use a context manager (any object that satisfies the context manager protocol) to add code before (on entry to) and after (on exit from) a block of code.

6.10.1 Writing a context manager

A context manager is an instance of a class that satisfies this interface:

```
class Context01(object):
    def __enter__(self):
        pass
    def __exit__(self, exc_type, exc_value, traceback):
        pass
```

Here is an example that uses the above context manager:

```
class Context01(object):
    def __enter__(self):
        print 'in __enter__'
        return 'some value or other'    # usually we want to return self
    def __exit__(self, exc_type, exc_value, traceback):
        print 'in __exit__'
```

Notes:

- The `__enter__` method is called *before* our block of code is entered.
- Usually, but not always, we will want the `__enter__` method to return `self`, that is, the instance of our context manager class. We do this so that we can write:

```
with MyContextManager() as obj:
    pass
```

  and then use the instance (`obj` in this case) in the nested block.

- The `__exit__` method is called when our block of code is exited either normally or because of an exception.
- If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.
- If the block exits normally, the value of `exc_type`, `exc_value`, and `traceback` will be `None`.

For more information on the `with:` statement, see Context Manager Types -- http://docs.python.org/2/library/stdtypes.html#context-manager-types.

See module `contextlib` for strange ways of writing context managers: http://docs.python.org/2/library/contextlib.html#module-contextlib

6.10.2 Using the with: statement

Here are examples:

```
# example 1
with Context01():
    print 'in body'

# example 2
with Context02() as a_value:
    print 'in body'
    print 'a_value: "%s"' % (a_value, )
    a_value.some_method_in_Context02()

# example 3
with open(infilename, 'r') as infile, open(outfilename, 'w') as outfile:
    for line in infile:
        line = line.rstrip()
        outfile.write('%s\n' % line.upper())
```

Notes:

- In the form `with ... as val`, the value returned by the `__enter__` method is assigned to the variable (`val` in this case).
- In order to use more than one context manager, you can nest `with:` statements, or separate uses of of the context managers with commas, which is usually preferred. See example 3 above.

6.11 del

The `del` statement can be used to:

- Remove names from namespace.
- Remove items from a collection.

If name is listed in a `global` statement, then `del` removes name from the global namespace.

Names can be a (nested) list. Examples:

```
>>> del a
>>> del a, b, c
```

We can also delete items from a list or dictionary (and perhaps from other objects that we can subscript). Examples:

```
In [9]:d = {'aa': 111, 'bb': 222, 'cc': 333}
In [10]:print d
{'aa': 111, 'cc': 333, 'bb': 222}
In [11]:del d['bb']
In [12]:print d
{'aa': 111, 'cc': 333}
In [13]:
In [13]:a = [111, 222, 333, 444]
In [14]:print a
[111, 222, 333, 444]
In [15]:del a[1]
In [16]:print a
[111, 333, 444]
```

And, we can delete an attribute from an instance. Example:

```
In [17]:class A:
   ....:    pass
   ....:
In [18]:a = A()
In [19]:a.x = 123
In [20]:dir(a)
Out[20]:['__doc__', '__module__', 'x']
In [21]:print a.x
123
In [22]:del a.x
In [23]:dir(a)
Out[23]:['__doc__', '__module__']
In [24]:print a.x
---------------------------------------------
exceptions.AttributeError     Traceback (most recent call last)

/home/dkuhlman/a1/Python/Test/<console>

AttributeError: A instance has no attribute 'x'
```

6.12 case statement

There is no case statement in Python. Use the `if:` statement with a sequence of `elif:` clauses. Or, use a dictionary of functions.

7 Functions, Modules, Packages, and Debugging

7.1 Functions

7.1.1 The def statement

The `def` statement is used to define functions and methods.

The `def` statement is evaluated. It produces a function/method (object) and binds it to a variable in the current name-space.

Although the `def` statement is evaluated, the code in its nested block is not executed. Therefore, many errors may not be detected until each and every path through that code is tested. Recommendations: (1) Use a Python code checker, for example `flake8` or `pylint`; (2) Do thorough testing and use the Python `unittest` framework. Pythonic wisdom: If it's not tested, it's broken.

7.1.2 Returning values

The `return` statement is used to return values from a function.

The `return` statement takes zero or more values, separated by commas. Using commas actually returns a single tuple.

The default value is `None`.

To return multiple values, use a tuple or list. Don't forget that (assignment) unpacking can be used to capture multiple values. Returning multiple items separated by commas is equivalent to returning a tuple. Example:

```
In [8]: def test(x, y):
   ...:     return x * 3, y * 4
   ...:
In [9]: a, b = test(3, 4)
In [10]: print a
9
In [11]: print b
16
```

7.1.3 Parameters

Default values -- Example:

```
In [53]: def t(max=5):
   ....:     for val in range(max):
   ....:         print val
   ....:
   ....:
In [54]: t(3)
0
1
2
In [55]: t()
0
1
2
```

```
3
4
```

Giving a parameter a default value makes that parameter optional.

Note: If a function has a parameter with a default value, then all "normal" arguments must proceed the parameters with default values. More completely, parameters must be given from left to right in the following order:

1. Normal arguments.
2. Arguments with default values.
3. Argument list (`*args`).
4. Keyword arguments (`**kwargs`).

List parameters -- `*args`. It's a tuple.

Keyword parameters -- `**kwargs`. It's a dictionary.

7.1.4 Arguments

When calling a function, values may be passed to a function with positional arguments or keyword arguments.

Positional arguments must placed before (to the left of) keyword arguments.

Passing lists to a function as multiple arguments -- `some_func(*aList)`. Effectively, this syntax causes Python to unroll the arguments. Example:

```
def fn1(*args, **kwargs):
    fn2(*args, **kwargs)
```

7.1.5 Local variables

Creating local variables -- Any binding operation creates a local variable. Examples are (1) parameters of a function; (2) assignment to a variable in a function; (3) the `import` statement; (4) etc. Contrast with accessing a variable.

Variable look-up -- The LGB/LEGB rule -- The local, enclosing, global, built-in scopes are searched in that order. See: http://www.python.org/dev/peps/pep-0227/

The `global` statement -- Inside a function, we must use `global` when we want to set the value of a global variable. Example:

```
def fn():
    global Some_global_variable, Another_global_variable
    Some_global_variable = 'hello'
    ...
```

7.1.6 Other things to know about functions

- Functions are first-class -- You can store them in a structure, pass them to a function, and return them from a function.

- Function calls can take keyword arguments. Example:

```
>>> test(size=25)
```

- Formal parameters to a function can have default values. Example:

```
>>> def test(size=0):
        ...
```

  Do *not* use mutable objects as default values.

- You can "capture" remaining arguments with `*args`, and `**kwargs`. (Spelling is not significant.) Example:

```
In [13]: def test(size, *args, **kwargs):
   ....:     print size
   ....:     print args
   ....:     print kwargs
   ....:
   ....:
In [14]: test(32, 'aa', 'bb', otherparam='xyz')
32
('aa', 'bb')
{'otherparam': 'xyz'}
```

- Normal arguments must come before default arguments which must come before keyword arguments.

- A function that does not explicitly return a value, returns `None`.

- In order to *set* the value of a global variable, declare the variable with `global`.

Exercises:

- Write a function that takes a single argument, prints the value of the argument, and returns the argument as a string. Solution:

```
>>> def t(x):
...     print 'x: %s' % x
...     return '[[%s]]' % x
...
>>> t(3)
x: 3
'[[3]]'
```

- Write a function that takes a variable number of arguments and prints them all. Solution:

```
>>> def t(*args):
...     for arg in args:
...         print 'arg: %s' % arg
...
>>> t('aa', 'bb', 'cc')
arg: aa
arg: bb
arg: cc
```

- Write a function that prints the names and values of keyword arguments passed to it. Solution:

```
>>> def t(**kwargs):
...     for key in kwargs.keys():
...         print 'key: %s  value: %s' % (key, kwargs[key], )
...
>>> t(arg1=11, arg2=22)
key: arg1  value: 11
key: arg2  value: 22
```

7.1.7 Global variables and the global statement

By default, assignment in a function or method creates local variables.

Reference (not assignment) to a variable, accesses a local variable if it has already been created, else accesses a global variable.

In order to assign a value to a global variable, declare the variable as global at the beginning of the function or method.

If in a function or method, you both reference and assign to a variable, then you must either:

1. Assign to the variable first, or
2. Declare the variable as global.

The `global` statement declares one or more variables, separated by commas, to be global.

Some examples:

```
In [1]:
In [1]: X = 3
In [2]: def t():
   ...:       print X
   ...:
In [3]:
In [3]: t()
3
In [4]: def s():
   ...:     X = 4
   ...:
In [5]:
In [5]:
In [5]: s()
In [6]: t()
3
In [7]: X = -1
In [8]: def u():
   ...:     global X
   ...:     X = 5
   ...:
In [9]:
In [9]: u()
In [10]: t()
5
In [16]: def v():
   ....:     x = X
   ....:     X = 6
   ....:     return x
```

```
   ....:
In [17]:
In [17]: v()
---------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
  File "<ipython console>", line 2, in v
UnboundLocalError: local variable 'X' referenced before assignment
In [18]: def w():
   ....:     global X
   ....:     x = X
   ....:     X = 7
   ....:     return x
   ....:
In [19]:
In [19]: w()
Out[19]: 5
In [20]: X
Out[20]: 7
```

7.1.8 Doc strings for functions

Add docstrings as a triple-quoted string beginning with the first line of a function or method. See epydoc for a suggested format.

7.1.9 Decorators for functions

A decorator performs a transformation on a function. Examples of decorators that are built-in functions are: `@classmethod`, `@staticmethod`, and `@property`. See: http://docs.python.org/2/library/functions.html#built-in-functions

A decorator is applied using the "@" character on a line immediately preceeding the function definition header. Examples:

```
class SomeClass(object):

    @classmethod
    def HelloClass(cls, arg):
        pass
    ## HelloClass = classmethod(HelloClass)

    @staticmethod
    def HelloStatic(arg):
        pass
    ## HelloStatic = staticmethod(HelloStatic)

#
# Define/implement a decorator.
def wrapper(fn):
    def inner_fn(*args, **kwargs):
        print '>>'
        result = fn(*args, **kwargs)
        print '<<'
        return result
    return inner_fn

#
# Apply a decorator.
@wrapper
def fn1(msg):
    pass
## fn1 = wrapper(fn1)
```

Notes:

- The decorator form (with the "@" character) is equivalent to the form (commented out) that calls the decorator function explicitly.
- The use of `classmethods` and `staticmethod` will be explained later in the section on object-oriented programming.
- A decorator is implemented as a function. Therefore, to learn about some specific decorator, you should search for the documentation on or the implementation of that function. Remember that in order to use a function, it must be defined in the current module or imported by the current module or be a built-in.
- The form that explicitly calls the decorator function (commented out in the example above) is equivalent to the form using the "@" character.

7.2 lambda

Use a lambda, as a convenience, when you need a function that both:

- is anonymous (does not need a name) and
- contains only an expression and no statements.

Example:

```
In [1]: fn = lambda x, y, z: (x ** 2) + (y * 2) + z
In [2]: fn(4, 5, 6)
Out[2]: 32
In [3]:
In [3]: format = lambda obj, category: 'The "%s" is a "%s".' % (obj,
category, )
In [4]: format('pine tree', 'conifer')
Out[4]: 'The "pine tree" is a "conifer".'
```

A lambda can take multiple arguments and can return (like a function) multiple values.
Example:

```
In [79]: a = lambda x, y: (x * 3, y * 4, (x, y))
In [80]:
In [81]: a(3, 4)
Out[81]: (9, 16, (3, 4))
```

Suggestion: In some cases, a lambda may be useful as an event handler.

Example:

```
class Test:
    def __init__(self, first='', last=''):
        self.first = first
        self.last = last
    def test(self, formatter):
        """
        Test for lambdas.
        formatter is a function taking 2 arguments, first and last
          names.  It should return the formatted name.
        """
        msg = 'My name is %s' % (formatter(self.first, self.last),)
        print msg

def test():
    t = Test('Dave', 'Kuhlman')
```

```
    t.test(lambda first, last: '%s %s' % (first, last, ))
    t.test(lambda first, last: '%s, %s' % (last, first, ))

test()
```

A `lambda` enables us to define "functions" where we do not need names for those functions.

Example:

```
In [45]: a = [
   ....: lambda x: x,
   ....: lambda x: x * 2,
   ....: ]
In [46]:
In [46]: a
Out[46]: [<function __main__.<lambda>>, <function __main__.<lambda>>]
In [47]: a[0](3)
Out[47]: 3
In [48]: a[1](3)
Out[48]: 6
```

Reference: http://docs.python.org/2/reference/expressions.html#lambda

7.3 Iterators and generators

Concepts:

iterator
: And iterator is something that satisfies the iterator protocol. Clue: If it's an iterator, you can use it in a `for:` statement.

generator
: A generator is a class or function that implements an iterator, i.e. that implements the iterator protocol.

the iterator protocol
: An object satisfies the iterator protocol if it does the following:

- It implements a `__iter__` method, which returns an iterator object.
- It implements a `next` function, which returns the next item from the collection, sequence, stream, etc of items to be iterated over
- It raises the `StopIteration` exception when the items are exhausted and the `next()` method is called.

`yield`
: The `yield` statement enables us to write functions that are generators. Such functions may be similar to coroutines, since they may "yield" multiple times and are resumed.

For more information on iterators, see the section on iterator types in the Python Library Reference -- http://docs.python.org/2/library/stdtypes.html#iterator-types.

For more on the `yield` statement, see:
http://docs.python.org/2/reference/simple_stmts.html#the-yield-statement

Actually, `yield` is an expression. For more on yield expressions and on the `next()` and `send()`

generator methods, as well as others, see: Yield expression -- http://docs.python.org/2/reference/expressions.html#yield-expressions in the Python language reference manual.

A function or method containing a `yield` statement implements a generator. Adding the `yield` statement to a function or method turns that function or method into one which, when called, returns a generator, i.e. an object that implements the iterator protocol.

A generator (a function containing `yield`) provides a convenient way to implement a filter. But, also consider:

- The `filter()` built-in function
- List comprehensions with an `if` clause

Here are a few examples:

```
def simplegenerator():
    yield 'aaa'                         # Note 1
    yield 'bbb'
    yield 'ccc'

def list_tripler(somelist):
    for item in somelist:
        item *= 3
        yield item

def limit_iterator(somelist, max):
    for item in somelist:
        if item > max:
            return                      # Note 2
        yield item

def test():
    print '1.', '-' * 30
    it = simplegenerator()
    for item in it:
        print item
    print '2.', '-' * 30
    alist = range(5)
    it = list_tripler(alist)
    for item in it:
        print item
    print '3.', '-' * 30
    alist = range(8)
    it = limit_iterator(alist, 4)
    for item in it:
        print item
    print '4.', '-' * 30
    it = simplegenerator()
    try:
        print it.next()                 # Note 3
        print it.next()
        print it.next()
        print it.next()
    except StopIteration, exp:          # Note 4
        print 'reached end of sequence'

if __name__ == '__main__':
    test()
```

Notes:

1. The `yield` statement returns a value. When the next item is requested and the iterator is "resumed", execution continues immediately after the `yield` statement.
2. We can terminate the sequence generated by an iterator by using a `return` statement with no value.
3. To resume a generator, use the generator's `next()` or `send()` methods. `send()` is like `next()`, but provides a value to the yield expression.
4. We can alternatively obtain the items in a sequence by calling the iterator's `next()` method. Since an iterator is a first-class object, we can save it in a data structure and can pass it around for use at different locations and times in our program.

6. When an iterator is exhausted or empty, it throws the `StopIteration` exception, which we can catch.

And here is the output from running the above example:

```
$ python test_iterator.py
1. ------------------------------
aaa
bbb
ccc
2. ------------------------------
0
3
6
9
12
3. ------------------------------
0
1
2
3
4
4. ------------------------------
aaa
bbb
ccc
reached end of sequence
```

An instance of a class which implements the `__iter__` method, returning an iterator, is iterable. For example, it can be used in a `for` statement or in a list comprehension, or in a generator expression, or as an argument to the `iter()` built-in method. But, notice that the class most likely implements a generator method which can be called directly.

Examples -- The following code implements an iterator that produces all the objects in a tree of objects:

```
class Node:
    def __init__(self, data, children=None):
        self.initlevel = 0
        self.data = data
        if children is None:
            self.children = []
        else:
            self.children = children
    def set_initlevel(self, initlevel): self.initlevel = initlevel
    def get_initlevel(self): return self.initlevel
    def addchild(self, child):
        self.children.append(child)
    def get_data(self):
        return self.data
    def get_children(self):
```

```
        return self.children
    def show_tree(self, level):
        self.show_level(level)
        print 'data: %s' % (self.data, )
        for child in self.children:
            child.show_tree(level + 1)
    def show_level(self, level):
        print '   ' * level,
    #
    # Generator method #1
    # This generator turns instances of this class into iterable objects.
    #
    def walk_tree(self, level):
        yield (level, self, )
        for child in self.get_children():
            for level1, tree1 in child.walk_tree(level+1):
                yield level1, tree1
    def __iter__(self):
        return self.walk_tree(self.initlevel)


#
# Generator method #2
# This generator uses a support function (walk_list) which calls
#   this function to recursively walk the tree.
# If effect, this iterates over the support function, which
#   iterates over this function.
#
def walk_tree(tree, level):
    yield (level, tree)
    for child in walk_list(tree.get_children(), level+1):
        yield child

def walk_list(trees, level):
    for tree in trees:
        for tree in walk_tree(tree, level):
            yield tree


#
# Generator method #3
# This generator is like method #2, but calls itself (as an iterator),
#   rather than calling a support function.
#
def walk_tree_recur(tree, level):
    yield (level, tree,)
    for child in tree.get_children():
        for level1, tree1 in walk_tree_recur(child, level+1):
            yield (level1, tree1, )


def show_level(level):
    print '   ' * level,


def test():
    a7 = Node('777')
    a6 = Node('666')
    a5 = Node('555')
    a4 = Node('444')
    a3 = Node('333', [a4, a5])
    a2 = Node('222', [a6, a7])
    a1 = Node('111', [a2, a3])
    initLevel = 2
    a1.show_tree(initLevel)
    print '=' * 40
    for level, item in walk_tree(a1, initLevel):
        show_level(level)
        print 'item:', item.get_data()
```

```
    print '=' * 40
    for level, item in walk_tree_recur(a1, initLevel):
        show_level(level)
        print 'item:', item.get_data()
    print '=' * 40
    a1.set_initlevel(initLevel)
    for level, item in a1:
        show_level(level)
        print 'item:', item.get_data()
    iter1 = iter(a1)
    print iter1
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
    print iter1.next()
##    print iter1.next()
    return a1

if __name__ == '__main__':
    test()
```

Notes:

- An instance of class `Node` is "iterable". It can be used directly in a `for` statement, a list comprehension, etc. So, for example, when an instance of `Node` is used in a `for` statement, it produces an iterator.
- We could also call the `Node.walk_method` directly to obtain an iterator.
- Method `Node.walk_tree` and functions `walk_tree` and `walk_tree_recur` are generators. When called, they return an iterator. They do this because they each contain a `yield` statement.
- These methods/functions are recursive. They call themselves. Since they are generators, they must call themselves in a context that uses an iterator, for example in a `for` statement.

7.4 Modules

A module is a Python source code file.

A module can be imported. When imported, the module is evaluated, and a module object is created. The module object has attributes. The following attributes are of special interest:

- `__doc__` -- The doc string of the module.
- `__name__` -- The name of the module when the module is imported, but the string "__main__" when the module is executed.
- Other names that are created (bound) in the module.

A module can be run.

To make a module both import-able and run-able, use the following idiom (at the end of the module):

```
def main():
    o
    o
    o

if __name__ == '__main__':
```

```
    main()
```

Where Python looks for modules:

- See `sys.path`.
- Standard places.
- Environment variable `PYTHONPATH`.

Notes about modules and objects:

- A module is an object.
- A module (object) can be shared.
- A specific module is imported only once in a single run. This means that a single module object is shared by all the modules that import it.

7.4.1 Doc strings for modules

Add docstrings as a triple-quoted string at or near the top of the file. See epydoc for a suggested format.

7.5 Packages

A package is a directory on the file system which contains a file named `__init__.py`.

The `__init__.py` file:

- Why is it there? -- It makes modules in the directory "import-able".
- Can `__init__.py` be empty? -- Yes. Or, just include a comment.
- When is it evaluated? -- It is evaluated the first time that an application imports anything from that directory/package.
- What can you do with it? -- Any code that should be executed exactly once and during import. For example:
    - Perform initialization needed by the package.
    - Make variables, functions, classes, etc available. For example, when the package is imported rather than modules in the package. You can also expose objects defined in modules contained in the package.
- Define a variable named `__all__` to specify the list of names that will be imported by `from my_package import *`. For example, if the following is present in `my_package/__init__.py`:

    ```
    __all__ = ['func1', 'func2',]
    ```

    Then, `from my_package import *` will import `func1` and `func2`, but not other names defined in `my_package`.

    Note that `__all__` can be used at the module level, as well as at the package level.

For more information, see the section on packages in the Python tutorial:

http://docs.python.org/2/tutorial/modules.html#packages.

Guidance and suggestions:

- "Flat is better" -- Use the `__init__.py` file to present a "flat" view of the API for your code. Enable your users to do `import mypackage` and then reference:
  - `mypackage.item1`
  - `mypackage.item2`
  - `mypackage.item3`
  - Etc.

  Where `item1`, `item2`, etc compose the API you want your users to use, even though the implementation of these items may be buried deep in your code.
- Use the `__init__.py` module to present a "clean" API. Present only the items that you intend your users to use, and by doing so, "hide" items you do *not* intend them to use.

8 Classes

Classes model the behavior of objects in the "real" world. Methods implement the behaviors of these types of objects. Member variables hold (current) state. Classes enable us to implement new data types in Python.

The `class:` statement is used to define a class. The `class:` statement creates a class object and binds it to a name.

8.1 A simple class

```
In [104]: class A:
   .....:     pass
   .....:
In [105]: a = A()
```

To define a new style class (recommended), inherit from `object` or from another class that does. Example:

```
In [21]: class A(object):
   ....:     pass
   ....:
In [22]:
In [22]: a = A()
In [23]: a
Out[23]: <__main__.A object at 0x82fbfcc>
```

8.2 Defining methods

A method is a function defined in class scope and with first parameter `self`:

```
In [106]: class B(object):
   .....:     def show(self):
   .....:         print 'hello from B'
   .....:
```

```
In [107]: b = B()
In [108]: b.show()
hello from B
```

A method as we describe it here is more properly called an *instance method*, in order to distinguish it from *class methods* and *static methods*.

8.3 The constructor

The constructor is a method named `__init__`.

Exercise: Define a class with a member variable `name` and a `show` method. Use `print` to show the name. Solution:

```
In [109]: class A(object):
   .....:       def __init__(self, name):
   .....:           self.name = name
   .....:       def show(self):
   .....:           print 'name: "%s"' % self.name
   .....:
In [111]: a = A('dave')
In [112]: a.show()
name: "dave"
```

Notes:

- The `self` variable is explicit. It references the current object, that is the object whose method is currently executing.
- The spelling ("self") is optional, but *everyone* spells it that way.

8.4 Member variables

Defining member variables -- Member variables are created with assignment. Example:

```
class A(object):
    def __init__(self, name):
        self.name = name
```

A small gotcha -- Do this:

```
In [28]: class A(object):
   ....:     def __init__(self, items=None):
   ....:         if items is None:
   ....:             self.items = []
   ....:         else:
   ....:             self.items = items
```

Do *not* do this:

```
In [29]: class B:
   ....:     def __init__(self, items=[]):   # wrong.  list ctor evaluated
only once.
   ....:         self.items = items
```

In the second example, the `def` statement and the list constructor are evaluated only once. Therefore, the item member variable of all instances of class B, will share the same value,

which is most likely *not* what you want.

8.5 Calling methods

- Use the instance and the dot operator.
- Calling a method defined in the same class or a superclass:
    - From outside the class -- Use the instance:

```
some_object.some_method()
an_array_of_of_objects[1].another_method()
```

    - From within the same class -- Use `self`:

```
self.a_method()
```

    - From with a subclass when the method is in the superclass and there is a method with the same name in the current class -- Use the class (name) or use `super`:

```
SomeSuperClass.__init__(self, arg1, arg2) super(CurrentClass,
self).__init__(arg1, arg2)
```

- Calling a method defined in a specific superclass -- Use the class (name).

8.6 Adding inheritance

Referencing superclasses -- Use the built-in `super` or the explicit name of the superclass. Use of `super` is preferred. For example:

```
In [39]: class B(A):
   ....:     def __init__(self, name, size):
   ....:         super(B, self).__init__(name)
   ....:         # A.__init__(self, name)    # an older alternative form
   ....:         self.size = size
```

The use of `super()` may solve problems searching for the base class when using multiple inheritance. A better solution is to not use multiple inheritance.

You can also use multiple inheritance. But, it can cause confusion. For example, in the following, class C inherits from both A and B:

```
class C(A, B):
    ...
```

Python searches superclasses MRO (method resolution order). If only single inheritance is involved, there is little confusion. If multiple inheritance is being used, the search order of super classes can get complex -- see here:
http://www.python.org/download/releases/2.3/mro

For more information on inheritance, see the tutorial in the standard Python documentation

set: 9.5 Inheritance and 9.5.1 Multiple Inheritance.

Watch out for problems with inheriting from multiple classes that have a common base class.

8.7 Class variables

- Also called static data.
- A class variable is shared by instances of the class.
- Define at class level with assignment. Example:

```
class A(object):
    size = 5
    def get_size(self):
        return A.size
```

- Reference with `classname.variable`.
- Caution: `self.variable = x` creates a new member variable.

8.8 Class methods and static methods

Instance (plain) methods:

- An instance method receives the instance as its first argument.

Class methods:

- A class method receives the class as its first argument.
- Define class methods with built-in function `classmethod()` or with decorator `@classmethod`.
- See the description of `classmethod()` built-in function at "Built-in Functions": http://docs.python.org/2/library/functions.html#classmethod

Static methods:

- A static method receives neither the instance nor the class as its first argument.
- Define static methods with built-in function `staticmethod()` or with decorator `@staticmethod`.
- See the description of `staticmethod()` built-in function at "Built-in Functions": http://docs.python.org/2/library/functions.html#staticmethod

Notes on decorators:

- A decorator of the form `@afunc` is the same as `m = afunc(m)`. So, this:

```
@afunc
def m(self): pass
```

  is the same as:

```
def m(self): pass
m = afunc(m)
```

- You can use decorators `@classmethod` and `@staticmethod` (instead of the `classmethod()` and `staticmethod()` built-in functions) to declare class methods and static methods.

Example:

```
class B(object):

    Count = 0

    def dup_string(x):
        s1 = '%s%s' % (x, x,)
        return s1
    dup_string = staticmethod(dup_string)

    @classmethod
    def show_count(cls, msg):
        print '%s  %d' % (msg, cls.Count, )

def test():
    print B.dup_string('abcd')
    B.show_count('here is the count: ')
```

An alternative way to implement "static methods" -- Use a "plain", module-level function. For example:

```
In [1]: def inc_count():
   ...:         A.count += 1
   ...:
In [2]:
In [2]: def dec_count():
   ...:         A.count -= 1
   ...:
In [3]:
In [3]: class A:
   ...:         count = 0
   ...:     def get_count(self):
   ...:             return A.count
   ...:
In [4]:
In [4]: a = A()
In [5]: a.get_count()
Out[5]: 0
In [6]:
In [6]:
In [6]: inc_count()
In [7]: inc_count()
In [8]: a.get_count()
Out[8]: 2
In [9]:
In [9]: b = A()
In [10]: b.get_count()
Out[10]: 2
```

8.9 Properties

The property built-in function enables us to write classes in a way that does not require a user of the class to use getters and setters. Example:

```
class TestProperty(object):
    def __init__(self, description):
        self._description = description
```

```
    def _set_description(self, description):
        print 'setting description'
        self._description = description
    def _get_description(self):
        print 'getting description'
        return self._description
    description = property(_get_description, _set_description)
```

The `property` built-in function is also a decorator. So, the following is equivalent to the above example:

```
class TestProperty(object):
    def __init__(self, description):
        self._description = description

    @property
    def description(self):
        print 'getting description'
        return self._description

    @description.setter
    def description(self, description):
        print 'setting description'
        self._description = description
```

Notes:

- We mark the instance variable as private by prefixing it with and underscore.
- The name of the instance variable and the name of the property must be different. If they are not, we get recursion and an error.

For more information on properties, see Built-in Functions -- properties -- http://docs.python.org/2/library/functions.html#property

8.10 Interfaces

In Python, to implement an interface is to implement a method with a specific name and a specific arguments.

"Duck typing" -- If it walks like a duck and quacks like a duck ...

One way to define an "interface" is to define a class containing methods that have a header and a doc string but no implementation.

Additional notes on interfaces:

- Interfaces are not enforced.
- A class does not have to implement *all* of an interface.

8.11 New-style classes

A new-style class is one that subclasses `object` or a class that subclasses `object` (that is, another new-style class).

You can subclass Python's built-in data-types.

- A simple example -- the following class extends the list data-type:

```
class C(list):
  def get_len(self):
    return len(self)

c = C((11,22,33))
c.get_len()

c = C((11,22,33,44,55,66,77,88))
print c.get_len()
# Prints "8".
```

- A slightly more complex example -- the following class extends the dictionary data-type:

```
class D(dict):
    def __init__(self, data=None, name='no_name'):
        if data is None:
            data = {}
        dict.__init__(self, data)
        self.name = name
    def get_len(self):
        return len(self)
    def get_keys(self):
        content = []
        for key in self:
            content.append(key)
        contentstr = ', '.join(content)
        return contentstr
    def get_name(self):
        return self.name

def test():
    d = D({'aa': 111, 'bb':222, 'cc':333})
    # Prints "3"
    print d.get_len()
    # Prints "'aa, cc, bb'"
    print d.get_keys()
    # Prints "no_name"
    print d.get_name()
```

Some things to remember about new-style classes:

- In order to be new-style, a class must inherit (directly or indirectly) from `object`. Note that if you inherit from a built-in type, you get this automatically.
- New-style classes unify types and classes.
- You can subclass (built-in) types such as dict, str, list, file, etc.
- The built-in types now provide factory functions: `dict()`, `str()`, `int()`, `file()`, etc.
- The built-in types are introspect-able -- Use `x.__class__`, `dir(x.__class__)`, `isinstance(x, list)`, etc.
- New-style classes give you properties and descriptors.
- New-style classes enable you to define static methods. Actually, all classes enable you to do this.
- A new-style class is a user-defined type. For an instance of a new-style class x, `type(x)` is the same as `x.__class__`.

For more on new-style classes, see: http://www.python.org/doc/newstyle/

Exercises:

- Write a class and a subclass of this class.

- Give the superclass one member variable, a name, which can be entered when an instance is constructed.
- Give the subclass one member variable, a description; the subclass constructor should allow entry of both name and description.
- Put a `show()` method in the superclass and override the `show()` method in the subclass.

Solution:

```
class A(object):
    def __init__(self, name):
        self.name = name
    def show(self):
        print 'name: %s' % (self.name, )

class B(A):
    def __init__(self, name, desc):
        A.__init__(self, name)
        self.desc = desc
    def show(self):
        A.show(self)
        print 'desc: %s' % (self.desc, )
```

8.12 Doc strings for classes

Add docstrings as a (triple-quoted) string beginning with the first line of a class. See epydoc for a suggested format.

8.13 Private members

Add an leading underscore to a member name (method or data variable) to "suggest" that the member is private.

9 Special Tasks

9.1 Debugging tools

`pdb` -- The Python debugger:

- Start the debugger by running an expression:

```
pdb.run('expression')
```

Example:

```
if __name__ == '__main__':
    import pdb
    pdb.run('main()')
```

- Start up the debugger at a specific location with the following:

```
import pdb; pdb.set_trace()
```

Example:

```
if __name__ == '__main__':
    import pdb
    pdb.set_trace()
    main()
```

- Get help from within the debugger. For example:

```
(Pdb) help
(Pdb) help next
```

Can also embed IPython into your code. See http://ipython.scipy.org/doc/manual/manual.html.

`ipdb` -- Also consider using `ipdb` (and `IPython`). The `ipdb` debugger interactive prompt has some additional features, for example, it does tab name completion.

Inspecting:

- `import inspect`
- See http://docs.python.org/lib/module-inspect.html.
- Don't forget to try `dir(obj)` and `type(obj)` and `help(obj)`, first.

Miscellaneous tools:

- `id(obj)`
- `globals()` and `locals()`.
- `dir(obj)` -- Returns interesting names, but list is not necessarily complete.
- `obj.__class__`
- `cls.__bases__`
- `obj.__class__.__bases__`
- `obj.__doc__`. But usually, `help(obj)` is better. It produces the doc string.
- Customize the representation of your class. Define the following methods in your class:
  - `__repr__()` -- Called by (1) `repr()`, (2) interactive interpreter when representation is needed.
  - `__str__()` -- Called by (1) `str()`, (2) string formatting.

`pdb` is implemented with the `cmd` module in the Python standard library. You can implement similar command line interfaces by using `cmd`. See: cmd -- Support for line-oriented command interpreters -- http://docs.python.org/lib/module-cmd.html.

9.2 File input and output

Create a file object. Use `open()`.

This example reads and prints each line of a file:

```
def test():
    f = file('tmp.py', 'r')
    for line in f:
        print 'line:', line.rstrip()
    f.close()
```

```
test()
```

Notes:

- A text file is an iterable. It iterates over the lines in a file. The following is a common idiom:

```
infile = file(filename, 'r')
for line in infile:
    process_a_line(line)
infile.close()
```

- `string.rstrip()` strips new-line and other whitespace from the right side of each line. To strip new-lines only, but not other whitespace, try `rstrip('\n')`.
- Other ways of reading from a file/stream object: `my_file.read()`, `my_file.readline()`, `my_file.readlines()`,

This example writes lines of text to a file:

```
def test():
    f = file('tmp.txt', 'w')
    for ch in 'abcdefg':
        f.write(ch * 10)
        f.write('\n')
    f.close()

test()
```

Notes:

- The `write` method, unlike the `print` statement, does not automatically add new-line characters.
- Must close file in order to flush output. Or, use `my_file.flush()`.

And, don't forget the `with:` statement. It makes closing files automatic. The following example converts all the vowels in an input file to upper case and writes the converted lines to an output file:

```
import string

def show_file(infilename, outfilename):
    tran_table = string.maketrans('aeiou', 'AEIOU')
    with open(infilename, 'r') as infile, open(outfilename, 'w') as
outfile:
        for line in infile:
            line = line.rstrip()
            outfile.write('%s\n' % line.translate(tran_table))
```

9.3 Unit tests

For more documentation on the unit test framework, see unittest -- Unit testing framework -- http://docs.python.org/2/library/unittest.html#module-unittest

For help and more information do the following at the Python interactive prompt:

```
>>> import unittest
>>> help(unittest)
```

And, you can read the source: `Lib/unittest.py` in the Python standard library.

9.3.1 A simple example

Here is a very simple example. You can find more information about this primitive way of structuring unit tests in the library documentation for the `unittest` module Basic example -- http://docs.python.org/lib/minimal-example.html

```
import unittest

class UnitTests02(unittest.TestCase):

    def testFoo(self):
        self.failUnless(False)

class UnitTests01(unittest.TestCase):

    def testBar01(self):
        self.failUnless(False)

    def testBar02(self):
        self.failUnless(False)

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

Notes:

- The call to `unittest.main()` runs all tests in all test fixtures in the module. It actually creates an instance of class `TestProgram` in module `Lib/unittest.py`, which automatically runs tests.
- Test fixtures are classes that inherit from `unittest.TestCase`.
- Within a test fixture (a class), the tests are any methods whose names begin with the prefix "test".
- In any test, we check for success or failure with inherited methods such as `failIf()`, `failUnless()`, `assertNotEqual()`, etc. For more on these methods, see the library documentation for the `unittest` module TestCase Objects -- http://docs.python.org/lib/testcase-objects.html.
- If you want to change (1) the test method prefix or (2) the function used to sort (the order of) execution of tests within a test fixture, then you can create your own instance of class `unittest.TestLoader` and customize it. For example:

```
def main():
    my_test_loader = unittest.TestLoader()
    my_test_loader.testMethodPrefix = 'check'
```

```
    my_test_loader.sortTestMethodsUsing = my_cmp_func
    unittest.main(testLoader=my_test_loader)

if __name__ == '__main__':
    main()
```

But, see the notes in section Additional unittest features for instructions on a (possibly) better way to do this.

9.3.2 Unit test suites

Here is another, not quite so simple, example:

```
#!/usr/bin/env python

import sys, popen2
import getopt
import unittest


class GenTest(unittest.TestCase):

    def test_1_generate(self):
        cmd = 'python ../generateDS.py -f -o out2sup.py -s out2sub.py
people.xsd'
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        self.failUnless(len(result) == 0)

    def test_2_compare_superclasses(self):
        cmd = 'diff out1sup.py out2sup.py'
        outfile, infile = popen2.popen2(cmd)
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        #print 'len(result):', len(result)
        # Ignore the differing lines containing the date/time.
        #self.failUnless(len(result) < 130 and result.find('Generated') >
-1)
        self.failUnless(check_result(result))

    def test_3_compare_subclasses(self):
        cmd = 'diff out1sub.py out2sub.py'
        outfile, infile = popen2.popen2(cmd)
        outfile, infile = popen2.popen2(cmd)
        result = outfile.read()
        outfile.close()
        infile.close()
        # Ignore the differing lines containing the date/time.
        #self.failUnless(len(result) < 130 and result.find('Generated') >
-1)
        self.failUnless(check_result(result))


def check_result(result):
    flag1 = 0
    flag2 = 0
    lines = result.split('\n')
    len1 = len(lines)
    if len1 <= 5:
        flag1 = 1
    s1 = '\n'.join(lines[:4])
```

```
    if s1.find('Generated') > -1:
        flag2 = 1
    return flag1 and flag2


# Make the test suite.
def suite():
    # The following is obsolete.  See Lib/unittest.py.
    #return unittest.makeSuite(GenTest)
    loader = unittest.TestLoader()
    # or alternatively
    # loader = unittest.defaultTestLoader
    testsuite = loader.loadTestsFromTestCase(GenTest)
    return testsuite


# Make the test suite and run the tests.
def test():
    testsuite = suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    runner.run(testsuite)


USAGE_TEXT = """
Usage:
    python test.py [options]
Options:
    -h, --help      Display this help message.
Example:
    python test.py
"""

def usage():
    print USAGE_TEXT
    sys.exit(-1)


def main():
    args = sys.argv[1:]
    try:
        opts, args = getopt.getopt(args, 'h', ['help'])
    except:
        usage()
    relink = 1
    for opt, val in opts:
        if opt in ('-h', '--help'):
            usage()
    if len(args) != 0:
        usage()
    test()


if __name__ == '__main__':
    main()
    #import pdb
    #pdb.run('main()')
```

Notes:

- `GenTest` is our test suite class. It inherits from `unittest.TestCase`.
- Each method in `GenTest` whose name begins with "test" will be run as a test.
- The tests are run in alphabetic order by method name.
- Defaults in class `TestLoader` for the test name prefix and sort comparison function can be overridden. See 5.3.8 TestLoader Objects -- http://docs.python.org/lib/testloader-objects.html.
- A test case class may also implement methods named `setUp()` and `tearDown()` to be run before and after tests. See 5.3.5 TestCase Objects --

http://docs.python.org/lib/testcase-objects.html. Actually, the first test method in our example should, perhaps, be a `setUp()` method.
- The tests use calls such as `self.failUnless()` to report errors. These are inherited from class `TestCase`. See 5.3.5 TestCase Objects -- http://docs.python.org/lib/testcase-objects.html.
- Function `suite()` creates an instance of the test suite.
- Function `test()` runs the tests.

9.3.3 Additional unittest features

And, the following example shows several additional features. See the notes that follow the code:

```
import unittest

class UnitTests02(unittest.TestCase):
    def testFoo(self):
        self.failUnless(False)
    def checkBar01(self):
        self.failUnless(False)


class UnitTests01(unittest.TestCase):
    # Note 1
    def setUp(self):
        print 'setting up UnitTests01'
    def tearDown(self):
        print 'tearing down UnitTests01'
    def testBar01(self):
        print 'testing testBar01'
        self.failUnless(False)
    def testBar02(self):
        print 'testing testBar02'
        self.failUnless(False)

def function_test_1():
    name = 'mona'
    assert not name.startswith('mo')

def compare_names(name1, name2):
    if name1 < name2:
        return 1
    elif name1 > name2:
        return -1
    else:
        return 0

def make_suite():
    suite = unittest.TestSuite()
    # Note 2
    suite.addTest(unittest.makeSuite(UnitTests01,
sortUsing=compare_names))
    # Note 3
    suite.addTest(unittest.makeSuite(UnitTests02, prefix='check'))
    # Note 4
    suite.addTest(unittest.FunctionTestCase(function_test_1))
    return suite

def main():
    suite = make_suite()
    runner = unittest.TextTestRunner()
    runner.run(suite)

if __name__ == '__main__':
    main()
```

Notes:

1. If you run this code, you will notice that the `setUp` and `tearDown` methods in class `UnitTests01` are run before and after each test in that class.
2. We can control the order in which tests are run by passing a compare function to the `makeSuite` function. The default is the `cmp` built-in function.
3. We can control which methods in a test fixture are selected to be run by passing the optional argument `prefix` to the `makeSuite` function.
4. If we have an existing function that we want to "wrap" and run as a unit test, we can create a test case from a function with the `FunctionTestCase` function. If we do that, notice that we use the `assert` statement to test and possibly cause failure.

9.3.4 Guidance on Unit Testing

Why should we use unit tests? Many reasons, including:

- Without unit tests, corner cases may not be checked. This is especially important, since Python does relatively little compile time error checking.
- Unit tests facilitate a frequent and short design and implement and release development cycle. See ONLamp.com -- Extreme Python -- http://www.onlamp.com/pub/a/python/2001/03/28/pythonnews.html and What is XP -- http://www.xprogramming.com/what_is_xp.htm.
- Designing the tests before writing the code is "a good idea".

Additional notes:

- In a test class, instance methods `setUp` and `tearDown` are run automatically before each and after each individual test.
- In a test class, class methods `setUpClass` and `tearDownClass` are run automatically *once* before and after *all* the tests in a class.
- Module level functions `setUpModule` and `tearDownModule` are run before and after any tests in a module.
- In some cases you can also run tests directly from the command line. Do the following for help:

```
$ python -m unittest --help
```

9.4 doctest

For simple test harnesses, consider using `doctest`. With `doctest` you can (1) run a test at the Python interactive prompt, then (2) copy and paste that test into a doc string in your module, and then (3) run the tests automatically from within your module under `doctest`.

There are examples and explanation in the standard Python documentation: 5.2 doctest -- Test interactive Python examples -- http://docs.python.org/lib/module-doctest.html.

A simple way to use `doctest` in your module:

1. Run several tests in the Python interactive interpreter. Note that because `doctest` looks for the interpreter's ">>>" prompt, you must use the standard interpreter, and not, for example, IPython. Also, make sure that you include a line with the ">>>" prompt after each set of results; this enables `doctest` to determine the extent of the test results.

2. Use copy and paste, to insert the tests and their results from your interactive session into the docstrings.

3. Add the following code at the bottom of your module:

```
def _test():
    import doctest
    doctest.testmod()

if __name__ == "__main__":
    _test()
```

Here is an example:

```
def f(n):
    """
    Print something funny.

    >>> f(1)
    10
    >>> f(2)
    -10
    >>> f(3)
    0
    """
    if n == 1:
        return 10
    elif n == 2:
        return -10
    else:
        return 0


def test():
    import doctest, test_doctest
    doctest.testmod(test_doctest)

if __name__ == '__main__':
    test()
```

And, here is the output from running the above test with the `-v` flag:

```
$ python test_doctest.py -v
Running test_doctest.__doc__
0 of 0 examples failed in test_doctest.__doc__
Running test_doctest.f.__doc__
Trying: f(1)
Expecting: 10
ok
Trying: f(2)
Expecting: -10
ok
Trying: f(3)
Expecting: 0
ok
0 of 3 examples failed in test_doctest.f.__doc__
```

```
Running test_doctest.test.__doc__
0 of 0 examples failed in test_doctest.test.__doc__
2 items had no tests:
    test_doctest
    test_doctest.test
1 items passed all tests:
   3 tests in test_doctest.f
3 tests in 3 items.
3 passed and 0 failed.
Test passed.
```

9.5 The Python database API

Python database API defines a standard interface for access to a relational database.

In order to use this API you must install the database adapter (interface module) for your particular database, e.g. PostgreSQL, MySQL, Oracle, etc.

You can learn more about the Python DB-API here: http://www.python.org/dev/peps/pep-0249/

The following simple example uses sqlite3 -- http://docs.python.org/2/library/sqlite3.html

```
#!/usr/bin/env python

"""
Create a relational database and a table in it.
Add some records.
Read and display the records.
"""

import sys
import sqlite3

def create_table(db_name):
    con = sqlite3.connect(db_name)
    cursor = con.cursor()
    cursor.execute('''CREATE TABLE plants
    (name text, desc text, cat int)''')
    cursor.execute(
        '''INSERT INTO plants VALUES ('tomato', 'red and juicy', 1)''')
    cursor.execute(
        '''INSERT INTO plants VALUES ('pepper', 'green and crunchy',
2)''')
    cursor.execute('''INSERT INTO plants VALUES ('pepper', 'purple',
2)''')
    con.commit()
    con.close()

def retrieve(db_name):
    con = sqlite3.connect(db_name)
    cursor = con.cursor()
    cursor.execute('''select * from plants''')
    rows = cursor.fetchall()
    print rows
    print '-' * 40
    cursor.execute('''select * from plants''')
    for row in cursor:
        print row
    con.close()

def test():
    args = sys.argv[1:]
    if len(args) != 1:
```

```
        sys.stderr.write('\nusage: test_db.py <db_name>\n\n')
        sys.exit(1)
    db_name = args[0]
    create_table(db_name)
    retrieve(db_name)

test()
```

9.6 Installing Python packages

Simple:

```
$ python setup.py build
$ python setup.py install    # as root
```

More complex:

- Look for a `README` or `INSTALL` file at the root of the package.
- Type the following for help:

  ```
  $ python setup.py cmd --help
  $ python setup.py --help-commands
  $ python setup.py --help [cmd1 cmd2 ...]
  ```

- And, for even more details, see Installing Python Modules -- http://docs.python.org/inst/inst.html

`pip` is becoming popular for installing and managing Python packages. See: https://pypi.python.org/pypi/pip

Also, consider using `virtualenv`, especially if you suspect or worry that installing some new package will alter the behavior of a package currently installed on your machine. See: https://pypi.python.org/pypi/virtualenv. `virtualenv` creates a directory and sets up a Python environment into which you can install and use Python packages without changing your usual Python installation.

10 More Python Features and Exercises

[As time permits, explain more features and do more exercises as requested by class members.]

Thanks to David Goodger for the following list or references. His "Code Like a Pythonista: Idiomatic Python" (http://python.net/~goodger/projects/pycon/2007/idiomatic/) is worth a careful reading:

- "Python Objects", Fredrik Lundh, http://www.effbot.org/zone/python-objects.htm
- "How to think like a Pythonista", Mark Hammond, http://python.net/crew/mwh/hacks/objectthink.html
- "Python main() functions", Guido van Rossum, http://www.artima.com/weblogs/viewpost.jsp?thread=4829
- "Python Idioms and Efficiency", http://jaynes.colorado.edu/PythonIdioms.html
- "Python track: python idioms",

http://www.cs.caltech.edu/courses/cs11/material/python/misc/python_idioms.html
- "Be Pythonic", Shalabh Chaturvedi, http://shalabh.infogami.com/Be_Pythonic2
- "Python Is Not Java", Phillip J. Eby, http://dirtsimple.org/2004/12/python-is-not-java.html
- "What is Pythonic?", Martijn Faassen, http://faassen.n--tree.net/blog/view/weblog/2005/08/06/0
- "Sorting Mini-HOWTO", Andrew Dalke, http://wiki.python.org/moin/HowTo/Sorting
- "Python Idioms", http://www.gungfu.de/facts/wiki/Main/PythonIdioms
- "Python FAQs", http://www.python.org/doc/faq/

View document source. Generated on: 2014-10-05 20:00 UTC. Generated by Docutils from reStructuredText source.