1 Part 1 -- Beginning Python

1.1 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.

1.1.1 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(...)
```

1.1.2 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
```

1.1.3 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/.

1.2 Lexical matters

1.2.1 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.

1.2.2 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 "##".

1.2.3 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.

1.2.4 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.

1.2.5 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/.

1.2.6 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).

1.2.7 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.

1.2.8 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.

1.2.9 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.

1.3 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.
```

1.4 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.

1.4.1 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/

1.4.2 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.

1.4.3 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()
```

1.4.3.1 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'
```

1.4.3.2 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()
```

1.4.4 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
```