A Python Book

**A Python Book: Beginning Python, Advanced Python, and Python Exercises**

Author:Dave Kuhlman Contact:

dkuhlman@davekuhlman.org Address:

http://www.davekuhlman.org

Page 1

A Python Book

Revision

1.3a Date

December 15, 2013 Copyright

Copyright (c) 2009 Dave Kuhlman. All Rights Reserved. This document is subject to the provisions of the Open Source MIT License http://www.opensource.org/licenses/mit­license.php. Abstract

This document is a self­learning document for a course in Python programming. This course contains (1) a part for beginners, (2) a discussion of several advanced topics that are of interest to Python programmers, and (3) a Python workbook with lots of exercises.

Page 2

A Python Book

**Contents**

1 Part 1 ­­ Beginning Python...........................................................................................10 1.1 Introductions Etc...................................................................................................10 1.1.1 Resources.......................................................................................................11 1.1.2 A general description of Python....................................................................12 1.1.3 Interactive Python..........................................................................................15 1.2 Lexical matters......................................................................................................15 1.2.1 Lines..............................................................................................................15 1.2.2 Comments......................................................................................................16 1.2.3 Names and tokens..........................................................................................16 1.2.4 Blocks and indentation..................................................................................16 1.2.5 Doc strings.....................................................................................................17 1.2.6 Program structure..........................................................................................17 1.2.7 Operators.......................................................................................................18 1.2.8 Also see.........................................................................................................19 1.2.9 Code evaluation.............................................................................................19 1.3 Statements and inspection ­­ preliminaries...........................................................20 1.4 Built­in data­types.................................................................................................21 1.4.1 Numeric types................................................................................................21 1.4.2 Tuples and lists..............................................................................................21 1.4.3 Strings............................................................................................................24 1.4.3.1 The new string.format method...............................................................26 1.4.3.2 Unicode strings......................................................................................27 1.4.4 Dictionaries....................................................................................................29 1.4.5 Files...............................................................................................................32 1.4.6 Other built­in types........................................................................................35 1.4.6.1 The None value/type..............................................................................35 1.4.6.2 Boolean values.......................................................................................36 1.4.6.3 Sets and frozensets.................................................................................36 1.5 Functions and Classes ­­ A Preview......................................................................36 1.6 Statements.............................................................................................................37 1.6.1 Assignment statement....................................................................................37 1.6.2 import statement............................................................................................39 1.6.3 print statement...............................................................................................41 1.6.4 if: elif: else: statement...................................................................................43 1.6.5 for: statement.................................................................................................44 1.6.6 while: statement.............................................................................................48

Page 3

A Python Book

1.6.7 continue and break statements.......................................................................48 1.6.8 try: except: statement.....................................................................................49 1.6.9 raise statement...............................................................................................51 1.6.10 with: statement.............................................................................................52 1.6.10.1 Writing a context manager...................................................................52 1.6.10.2 Using the with: statement....................................................................53 1.6.11 del................................................................................................................54 1.6.12 case statement..............................................................................................55 1.7 Functions, Modules, Packages, and Debugging....................................................55 1.7.1 Functions.......................................................................................................55 1.7.1.1 The def statement...................................................................................55 1.7.1.2 Returning values....................................................................................55 1.7.1.3 Parameters..............................................................................................56 1.7.1.4 Arguments..............................................................................................56 1.7.1.5 Local variables.......................................................................................57 1.7.1.6 Other things to know about functions....................................................57 1.7.1.7 Global variables and the global statement.............................................58 1.7.1.8 Doc strings for functions.......................................................................60 1.7.1.9 Decorators for functions........................................................................60 1.7.2 lambda...........................................................................................................61 1.7.3 Iterators and generators.................................................................................62 1.7.4 Modules.........................................................................................................67 1.7.4.1 Doc strings for modules.........................................................................68 1.7.5 Packages........................................................................................................68 1.8 Classes...................................................................................................................69 1.8.1 A simple class................................................................................................69 1.8.2 Defining methods..........................................................................................70 1.8.3 The constructor..............................................................................................70 1.8.4 Member variables..........................................................................................70 1.8.5 Calling methods.............................................................................................71 1.8.6 Adding inheritance........................................................................................71 1.8.7 Class variables...............................................................................................72 1.8.8 Class methods and static methods.................................................................72 1.8.9 Properties.......................................................................................................74 1.8.10 Interfaces.....................................................................................................75 1.8.11 New­style classes.........................................................................................75 1.8.12 Doc strings for classes.................................................................................77 1.8.13 Private members..........................................................................................77 1.9 Special Tasks.........................................................................................................77 1.9.1 Debugging tools.............................................................................................77

Page 4

A Python Book

1.9.2 File input and output......................................................................................78 1.9.3 Unit tests........................................................................................................80 1.9.3.1 A simple example..................................................................................80 1.9.3.2 Unit test suites........................................................................................81 1.9.3.3 Additional unittest features....................................................................83 1.9.3.4 Guidance on Unit Testing......................................................................85 1.9.4 doctest............................................................................................................85 1.9.5 The Python database API..............................................................................87 1.9.6 Installing Python packages............................................................................88 1.10 More Python Features and Exercises..................................................................89 2 Part 2 ­­ Advanced Python............................................................................................90 2.1 Introduction ­­ Python 201 ­­ (Slightly) Advanced Python Topics.......................90 2.2 Regular Expressions..............................................................................................90 2.2.1 Defining regular expressions.........................................................................90 2.2.2 Compiling regular expressions......................................................................91 2.2.3 Using regular expressions..............................................................................91 2.2.4 Using match objects to extract a value..........................................................92 2.2.5 Extracting multiple items..............................................................................93 2.2.6 Replacing multiple items...............................................................................94 2.3 Iterator Objects......................................................................................................96 2.3.1 Example ­ A generator function....................................................................98 2.3.2 Example ­ A class containing a generator method......................................100 2.3.3 Example ­ An iterator class.........................................................................102 2.3.4 Example ­ An iterator class that uses yield.................................................104 2.3.5 Example ­ A list comprehension.................................................................105 2.3.6 Example ­ A generator expression..............................................................105 2.4 Unit Tests............................................................................................................106 2.4.1 Defining unit tests........................................................................................106 2.4.1.1 Create a test class.................................................................................106 2.5 Extending and embedding Python......................................................................109 2.5.1 Introduction and concepts............................................................................109 2.5.2 Extension modules.......................................................................................110 2.5.3 SWIG...........................................................................................................112 2.5.4 Pyrex............................................................................................................115 2.5.5 SWIG vs. Pyrex...........................................................................................120 2.5.6 Cython.........................................................................................................120 2.5.7 Extension types............................................................................................122 2.5.8 Extension classes.........................................................................................122 2.6 Parsing.................................................................................................................122 2.6.1 Special purpose parsers...............................................................................123

Page 5

A Python Book

2.6.2 Writing a recursive descent parser by hand.................................................124 2.6.3 Creating a lexer/tokenizer with Plex...........................................................131 2.6.4 A survey of existing tools............................................................................141 2.6.5 Creating a parser with PLY.........................................................................141 2.6.6 Creating a parser with pyparsing.................................................................148 2.6.6.1 Parsing comma­delimited lines............................................................148 2.6.6.2 Parsing functors...................................................................................149 2.6.6.3 Parsing names, phone numbers, etc.....................................................150 2.6.6.4 A more complex example....................................................................151 2.7 GUI Applications................................................................................................153 2.7.1 Introduction.................................................................................................153 2.7.2 PyGtk...........................................................................................................153 2.7.2.1 A simple message dialog box..............................................................153 2.7.2.2 A simple text input dialog box.............................................................156 2.7.2.3 A file selection dialog box...................................................................158 2.7.3 EasyGUI......................................................................................................160 2.7.3.1 A simple EasyGUI example................................................................161 2.7.3.2 An EasyGUI file open dialog example................................................161 2.8 Guidance on Packages and Modules...................................................................161 2.8.1 Introduction.................................................................................................161 2.8.2 Implementing Packages...............................................................................162 2.8.3 Using Packages............................................................................................162 2.8.4 Distributing and Installing Packages...........................................................162 2.9 End Matter...........................................................................................................164 2.9.1 Acknowledgements and Thanks..................................................................164 2.9.2 See Also.......................................................................................................164 3 Part 3 ­­ Python Workbook.........................................................................................165 3.1 Introduction.........................................................................................................165 3.2 Lexical Structures................................................................................................165 3.2.1 Variables and names....................................................................................165 3.2.2 Line structure...............................................................................................167 3.2.3 Indentation and program structure...............................................................168 3.3 Execution Model.................................................................................................169 3.4 Built­in Data Types.............................................................................................170 3.4.1 Numbers......................................................................................................170 3.4.1.1 Literal representations of numbers......................................................171 3.4.1.2 Operators for numbers.........................................................................173 3.4.1.3 Methods on numbers............................................................................175 3.4.2 Lists.............................................................................................................175 3.4.2.1 Literal representation of lists...............................................................176

Page 6

A Python Book

3.4.2.2 Operators on lists.................................................................................178 3.4.2.3 Methods on lists...................................................................................178 3.4.2.4 List comprehensions............................................................................180 3.4.3 Strings..........................................................................................................182 3.4.3.1 Characters............................................................................................183 3.4.3.2 Operators on strings.............................................................................184 3.4.3.3 Methods on strings...............................................................................185 3.4.3.4 Raw strings..........................................................................................187 3.4.3.5 Unicode strings....................................................................................188 3.4.4 Dictionaries..................................................................................................190 3.4.4.1 Literal representation of dictionaries...................................................190 3.4.4.2 Operators on dictionaries.....................................................................191 3.4.4.3 Methods on dictionaries.......................................................................192 3.4.5 Files.............................................................................................................195 3.4.6 A few miscellaneous data types..................................................................197 3.4.6.1 None.....................................................................................................197 3.4.6.2 The booleans True and False...............................................................197 3.5 Statements...........................................................................................................198 3.5.1 Assignment statement..................................................................................198 3.5.2 print statement.............................................................................................200 3.5.3 if: statement exercises..................................................................................201 3.5.4 for: statement exercises...............................................................................202 3.5.5 while: statement exercises...........................................................................205 3.5.6 break and continue statements.....................................................................206 3.5.7 Exceptions and the try:except: and raise statements...................................207 3.6 Functions.............................................................................................................210 3.6.1 Optional arguments and default values.......................................................211 3.6.2 Passing functions as arguments...................................................................213 3.6.3 Extra args and keyword args.......................................................................214 3.6.3.1 Order of arguments (positional, extra, and keyword args)..................216 3.6.4 Functions and duck­typing and polymorphism...........................................216 3.6.5 Recursive functions.....................................................................................217 3.6.6 Generators and iterators...............................................................................219 3.7 Object­oriented programming and classes..........................................................223 3.7.1 The constructor............................................................................................224 3.7.2 Inheritance ­­ Implementing a subclass.......................................................225 3.7.3 Classes and polymorphism..........................................................................227 3.7.4 Recursive calls to methods..........................................................................228 3.7.5 Class variables, class methods, and static methods.....................................230 3.7.5.1 Decorators for classmethod and staticmethod.....................................233

Page 7

A Python Book

3.8 Additional and Advanced Topics........................................................................234 3.8.1 Decorators and how to implement them......................................................234 3.8.1.1 Decorators with arguments..................................................................235 3.8.1.2 Stacked decorators...............................................................................236 3.8.1.3 More help with decorators...................................................................238 3.8.2 Iterables.......................................................................................................239 3.8.2.1 A few preliminaries on Iterables..........................................................239 3.8.2.2 More help with iterables......................................................................240 3.9 Applications and Recipes....................................................................................240 3.9.1 XML ­­ SAX, minidom, ElementTree, Lxml..............................................241 3.9.2 Relational database access...........................................................................249 3.9.3 CSV ­­ comma separated value files...........................................................255 3.9.4 YAML and PyYAML..................................................................................256 3.9.5 Json..............................................................................................................258 4 Part 4 ­­ Generating Python Bindings for XML.........................................................260 4.1 Introduction.........................................................................................................260 4.2 Generating the code.............................................................................................261 4.3 Using the generated code to parse and export an XML document.....................263 4.4 Some command line options you might want to know.......................................263 4.5 The graphical front­end.......................................................................................264 4.6 Adding application­specific behavior.................................................................265 4.6.1 Implementing custom subclasses................................................................265 4.6.2 Using the generated "API" from your application......................................266 4.6.3 A combined approach..................................................................................267 4.7 Special situations and uses..................................................................................269 4.7.1 Generic, type­independent processing.........................................................269 4.7.1.1 Step 1 ­­ generate the bindings............................................................270 4.7.1.2 Step 2 ­­ add application­specific code................................................270 4.7.1.3 Step 3 ­­ write a test/driver harness.....................................................274 4.7.1.4 Step 4 ­­ run the test application..........................................................276 4.8 Some hints...........................................................................................................276 4.8.1 Children defined with maxOccurs greater than 1........................................276 4.8.2 Children defined with simple numeric types...............................................277 4.8.3 The type of an element's character content..................................................277 4.8.4 Constructors and their default values..........................................................277

Page 8

A Python Book

Preface

This book is a collection of materials that I've used when conducting Python training and also materials from my Web site that are intended for self­instruction.

You may prefer a machine readable copy of this book. You can find it in various formats here:● HTML – http://www.davekuhlman.org/python\_book\_01.html

● PDF ­­ http://www.davekuhlman.org /python\_book\_01.pdf

● ODF/OpenOffice ­­ http://www.davekuhlman.org /python\_book\_01.odt And, let me thank the students in my Python classes. Their questions and suggestions were a great help in the preparation of these materials.

Page 9

A Python Book

**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/xemacs­faq.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 plug­in that supports Python.

● Kdevelop ­­ Linux/KDE ­­ See http://www.kdevelop.org/.

● Eric ­­ Linux KDE? ­­ See http://eric­ide.python­projects.org/index.html

● Emacs and SciTE will evaluate a Python buffer within the editor.

Page 10

A Python Book

**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: <built­in 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

Page 11

A Python Book

Callable: Yes Call def: Calling definition not available.Call docstring:

x.\_\_call\_\_(...) <==> x(...)

**1.1.2 A general description of Python**

Python is a high­level 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 object­oriented programming, Python enables us to write clear, logical applications for small and large tasks. Important features of Python:

● Built­in high level data types: strings, lists, dictionaries, etc.

● The usual control structures: if, if­else, if­elif­else, 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 "pre­compiled" to byte code files.

● Object­oriented ­­ 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 object­oriented 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. Out­dent one level to show the end of a block. As an example, the following C­style code:

if (x) {

Page 12

A Python Book

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 well­structured.

● Interpreted, but also compiled to byte­code. 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 inspect­able.

○ There is an interactive interpreter, more than one, in fact.

○ You can list the methods supported by any given object.

● Strongly typed at run­time, not compile­time. Objects (values) have a type, but variables do not.

● Reasonably high level ­­ High level built­in data types; high level control structures (for walking lists and iterators, for example).

● Object­oriented ­­ 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, well­structured applications. Why structure? Readability, locate­ability, modifiability.

● Explicitness

Page 13

A Python Book

● First­class 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 pseudo­code that runs."

● Embedding and extending Python ­­ Python provides a well­documented 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:

Page 14

A Python Book

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/pep­0020.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 semi­colon, 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 back­slash as last character of the line makes the

Page 15

A Python Book

following line a continuation of the current line. But, note that an opening "context" (parenthesis, square bracket, or curly bracket) makes the back­slash 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: a­z A­Z 0­9 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 no­op statement.

Benefits of the use of indentation to indicate structure:

Page 16

A Python Book

● 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 triple­quoting 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/module­pydoc.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://sphinx­doc.org/index.html. See the following for suggestions and more information on doc strings: Docstring conventions ­­ http://www.python.org/dev/peps/pep­0257/.

**1.2.6 Program structure**

● Execution ­­ def, class, etc are executable statements that add something to the current name­space. Modules can be both executable and import­able.

● 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

Page 17

A Python Book

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#operator­precedence, 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#special­method­names 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

Page 18

A Python Book

+, ­ 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#special­method­names 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/pep­0008/

● 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.

Page 19

A Python Book

● The built­in 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 built­in 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:<built­in method upper of str object at 0x7f1c426e0508> Docstring: S.upper() ­> string

Page 20

A Python Book

Return a copy of the string S converted to uppercase.

***1.4 Built­in data­types***

For information on built­in data types, see section Built­in 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 32­bit 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():

Page 21

A Python Book

● 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).

Page 22

A Python Book

● 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 (right­most) 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#sequence­types­str­unicode­list­tuple­byte array­buffer­xrange.

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]

Page 23

A Python Book

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) aabbcc 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/string­methods.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 back­slash.

● Triple quoting ­­ Use triple single quotes or double quotes to define multi­line 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#string­formatting­operations

Examples:

In [18]: name = 'dave'

Page 24

A Python Book

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 right­hand 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#string­methods

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,

Page 25

A Python Book

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 re­allocation. 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#format­string­syntax

● http://docs.python.org/2/library/string.html#format­specification­mini­language Some examples:

Page 26

A Python Book

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('utf­8') 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('utf­8') Out[108]: 'abcd' In [109]: a.encode(sys.getdefaultencoding()) Out[109]: 'abcd' In [110]: b = u'Sel\xe7uk' In [111]: print b.encode('utf­8') 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]:

Page 27

A Python Book

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('utf­8') Out[136]: u'Ivan Krsti\u0107' In [137]: In [138]: len(name) Out[138]: 12 In [139]: len(name.decode('utf­8')) Out[139]: 11 You can also create a unicode character by using the unichr() built­in function:

In [2]: a = 'aa' + unichr(170) + 'bb' In [3]: a Out[3]: u'aa\xaabb' In [6]: b = a.encode('utf­8') 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/pep­0100/

● In the Python standard library:

○ codecs ­­ Codec registry and base classes ­­ http://docs.python.org/2/library/codecs.html#module­codecs

○ Standard Encodings ­­ http://docs.python.org/2/library/codecs.html#standard­encodings If you are reading and writing multibyte character data from or to a *file*, then look at the

Page 28

A Python Book

codecs.open() in the codecs module ­­ http://docs.python.org/2/library/codecs.html#codecs.open.

Handling multi­byte character sets in Python 3 is easier, I think, but different. One hint is to use the encoding keyword parameter to the open built­in function. Here is an example:

def test():

infile = open('infile1.txt', 'r', encoding='utf­8') outfile = open('outfile1.txt', 'w', encoding='utf­8') 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 name­value 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 (key­value 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'][i­2], i) for i in (2, 3)]) For operations on dictionaries, see http://docs.python.org/lib/typesmapping.html or use:

>>> help({}) Or:

Page 29

A Python Book

>>> 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 ....: ....: aabb 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'

Page 30

A Python Book

... 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#dictionary­view­objects.

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 ... aaccbb>>> for value in d.values(): ... print value ... 111 333 222 >>> for item in d.items(): ... print item

Page 31

A Python Book

... ('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

**1.4.5 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 (built­in) open(path, mode) function to open a file and create a file object. You could also use file(), but open() is recommended.

Page 32

A Python Book

● 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*,

Page 33

A Python Book

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 Built­in Functions ­­ http://docs.python.org/lib/built­in­funcs.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#file­objects. 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 CRC­32 Name ­­­­­­­­ ­­­­­­ ­­­­­­­ ­­­­­ ­­­­ ­­­­ ­­­­­­ ­­­­

34 Defl:N 36 ­6% 05­29­08 17:04 f6b7d921 entry1 27 Defl:N 29 ­7% 05­29­08 17:07 10da8f3d entry2 22 Defl:N 24 ­9% 05­29­08 17:07 3fd17fda entry3 18 Defl:N 20 ­11% 05­29­08 17:08 d55182e6 entry4

Page 34

A Python Book

19 Defl:N 21 ­11% 05­29­08 17:08 1a892acd entry4 37 Defl:N 39 ­5% 05­29­08 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.

**1.4.6 Other built­in types**

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

***1.4.6.1 The None value/type***

The unique value None is used to indicate "no value", "nothing", "non­existence", 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'

Page 35

A Python Book

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

***1.4.6.2 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.

***1.4.6.3 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 hash­able 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/types­set.html

***1.5 Functions and Classes ­­ A Preview***

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

Page 36

A Python Book

Python objects are first­class objects.

Expressions are evaluated.

Statements are executed.

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

Object­oriented 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.

***1.6 Statements***

**1.6.1 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

Page 37

A Python Book

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

Page 38

A Python Book

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

**1.6.2 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 run­able and import­able:

if \_\_name\_\_ == '\_\_main\_\_':

Page 39

A Python Book

# 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 un­commented, 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 name­spaces.

● from A.B import C ­­ (1) Possibly import object C from module B in package A or (2) possibly import module C from sub­package B in package A.

● import A.B.C ­­ To reference attributes in C, must use fully­qualified 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

Page 40

A Python Book

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 import­able 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

**1.6.3 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 "pretty­printing" functions and methods, in particular, pprint. See 3.27 pprint ­­ Data pretty printer ­­

Page 41

A Python Book

http://docs.python.org/lib/module­pprint.html. String formatting ­­ Arguments are a tuple. Reference: 2.3.6.2 String Formatting Operations ­­ http://docs.python.org/lib/typesseq­strings.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 file­like 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')

...:

Page 42

A Python Book

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 built­in *function*.

**1.6.4 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

Page 43

A Python Book

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 no­op.)

● 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.

**1.6.5 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

Page 44

A Python Book

http://docs.python.org/lib/typeiter.html.

● We can create an iterator object with built­in functions such as iter() and enumerate(). See Built­in Functions ­­ http://docs.python.org/lib/built­in­funcs.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/built­in­funcs.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:

Page 45

A Python Book

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#yield­expressions

○ The yield statement ­­ http://docs.python.org/2/reference/simple\_stmts.html#the­yield­statement

● 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#module­itertools 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]Page 46

A Python Book

● [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:

Page 47

A Python Book

for item in data1:

if item > 100: value1 = item

break else: value1 = 'not found' print 'value1:', value1 When run, it prints:

value1: not found

**1.6.6 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.

**1.6.7 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:

Page 48

A Python Book

● 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

**1.6.8 try: except: statement**

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

Caught and un­caught 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/module­traceback.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.

Built­in exception classes ­­ See:

● Module exceptions.

Page 49

A Python Book

● Built­in exceptions ­­ http://docs.python.org/lib/module­exceptions.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/module­exceptions.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):

Page 50

A Python Book

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

**1.6.9 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 built­in exception class.

● An instance of class Exception or

● An instance of a built­in subclass of class Exception or

● An instance of a user­defined 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 built­in exceptions, see http://docs.python.org/lib/module­exceptions.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')

Page 51

A Python Book

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)

**1.6.10 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.

***1.6.10.1 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:

Page 52

A Python Book

● 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#context­manager­types.

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

***1.6.10.2 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

Page 53

A Python Book

preferred. See example 3 above.

**1.6.11 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)

Page 54

A Python Book

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

AttributeError: A instance has no attribute 'x'

**1.6.12 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.

***1.7 Functions, Modules, Packages, and Debugging***

**1.7.1 Functions**

***1.7.1.1 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 name­space.

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.

***1.7.1.2 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

Page 55

A Python Book

In [11]: print b 16

***1.7.1.3 Parameters***

Default values ­­ Example:

In [53]: def t(max=5):

....: for val in range(max): ....: print val ....: ....: In [54]: t(3) 012In [55]: t() 01234 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.

***1.7.1.4 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)

Page 56

A Python Book

***1.7.1.5 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 look­up ­­ The LGB/LEGB rule ­­ The local, enclosing, global, built­in scopes are searched in that order. See: http://www.python.org/dev/peps/pep­0227/

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' ...

***1.7.1.6 Other things to know about functions***

● Functions are first­class ­­ 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:

Page 57

A Python Book

● 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

***1.7.1.7 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.

Page 58

A Python Book

Some examples:

In [1]: In [1]: X = 3 In [2]: def t(): ...: print X ...: In [3]: In [3]: t() 3In [4]: def s():

...: X = 4 ...: In [5]: In [5]: In [5]: s() In [6]: t() 3In [7]: X = ­1 In [8]: def u():

...: global X ...: X = 5 ...: In [9]: In [9]: u() In [10]: t() 5In [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

Page 59

A Python Book

***1.7.1.8 Doc strings for functions***

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

***1.7.1.9 Decorators for functions***

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

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 object­oriented programming.

● A decorator is implemented as a function. Therefore, to learn about some specific

Page 60

A Python Book

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 built­in.

● The form that explicitly calls the decorator function (commented out in the example above) is equivalent to the form using the "@" character.

**1.7.2 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')Page 61

A Python Book

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

**1.7.3 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. yieldThe 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.

Page 62

A Python Book

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

For more on the yield statement, see: http://docs.python.org/2/reference/simple\_stmts.html#the­yield­statement

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#yield­expressions 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() built­in 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)

Page 63

A Python Book

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 first­class object, we can save it in a data structure and can pass it around for use at different locations and times in our program. 1. 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. ­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­ 0369123. ­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­ 0123

Page 64

A Python Book

44. ­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­ 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() built­in 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)

Page 65

A Python Book

## 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()

Page 66

A Python Book

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.

**1.7.4 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 import­able and run­able, use the following idiom (at the end of

Page 67

A Python Book

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.

***1.7.4.1 Doc strings for modules***

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

**1.7.5 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 "import­able".

● 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:

Page 68

A Python Book

\_\_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.

***1.8 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.

**1.8.1 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 ....:

Page 69

A Python Book

In [22]: In [22]: a = A() In [23]: a Out[23]: <\_\_main\_\_.A object at 0x82fbfcc>

**1.8.2 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*.

**1.8.3 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.

**1.8.4 Member variables**

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

class A(object):

def \_\_init\_\_(self, name):

Page 70

A Python Book

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.

**1.8.5 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).

**1.8.6 Adding inheritance**

Referencing superclasses ­­ Use the built­in super or the explicit name of the superclass. Use of super is preferred. For example:

In [39]: class B(A):

Page 71

A Python Book

....: 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.

**1.8.7 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.

**1.8.8 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 built­in function classmethod() or with decorator

Page 72

A Python Book

@classmethod.

● See the description of classmethod() built­in function at "Built­in 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 built­in function staticmethod() or with decorator @staticmethod.

● See the description of staticmethod() built­in function at "Built­in 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() built­in 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", module­level function. For example:

In [1]: def inc\_count():

...: A.count += 1 ...: In [2]:

Page 73

A Python Book

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

**1.8.9 Properties**

The property built­in 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 built­in 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.\_descriptionPage 74

A Python Book

@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 Built­in Functions ­­ properties ­­ http://docs.python.org/2/library/functions.html#property

**1.8.10 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.

**1.8.11 New­style classes**

A new­style class is one that subclasses object or a class that subclasses object (that is, another new­style class).

You can subclass Python's built­in data­types.

● A simple example ­­ the following class extends the list data­type:

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

Page 75

A Python Book

data­type:

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 new­style classes:

● In order to be new­style, a class must inherit (directly or indirectly) from object. Note that if you inherit from a built­in type, you get this automatically.

● New­style classes unify types and classes.

● You can subclass (built­in) types such as dict, str, list, file, etc.

● The built­in types now provide factory functions: dict(), str(), int(), file(), etc.

● The built­in types are introspect­able ­­ Use x.\_\_class\_\_, dir(x.\_\_class\_\_), isinstance(x, list), etc.

● New­style classes give you properties and descriptors.

● New­style classes enable you to define static methods. Actually, all classes enable you to do this.

● A new­style class is a user­defined type. For an instance of a new­style class x, type(x) is the same as x.\_\_class\_\_. For more on new­style 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

Page 76

A Python Book

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, )

**1.8.12 Doc strings for classes**

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

**1.8.13 Private members**

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

***1.9 Special Tasks***

**1.9.1 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:

Page 77

A Python Book

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/module­inspect.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 line­oriented command interpreters ­­ http://docs.python.org/lib/module­cmd.html.

**1.9.2 File input and output**

Create a file object. Use open().

This example reads and prints each line of a file:

Page 78

A Python Book

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 new­line and other whitespace from the right side of each line. To strip new­lines 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 new­line 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))

Page 79

A Python Book

**1.9.3 Unit tests**

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

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.

***1.9.3.1 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/minimal­example.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

Page 80