A Quick Tour of Python

Perry Greenfield & Richard L. White

March 27, 2002

Space Telescope Science Institute
Email: help@stsci.edu

Abstract

This document serves as a whirlwind overview of the basics of Python. We hope it will give adventurous PyRAF users a better idea of what is going on in the Python environment.

Contents

1 Python is Dynamic **2**

2 Braces? We Don't Need No Stinking Braces! **2**

3 Python Data Structures **3**
3.1 Strings . . . 3
3.2 Lists . . . 4
3.3 Mutability . . . 5
3.4 Tuples . . . 5
3.5 Dictionaries . . . 5

4 Truth and Consequences **6**

5 Control Constructs **7**

6 Modules and Namespaces **7**

7 Objects, Classes and What They Mean to You **8**

8 Defining Functions **8**

9 But Wait! There's More! **9**

10 Python Documentation and Books **9**
10.1 Books . . . 9
10.2 Online Documentation . . . 10

1 Python is Dynamic

Python is dynamically typed. You do not need to declare variables. You may simply assign to them, which creates the variable. Variables may hold simple types (integers, floats, etc.), functions, and objects among other things.

```
x = 1
name = "sample string"
name2 = 'another sample string'
name3 = """a multiline
  string example"""
y = 3.14
longint = 100000000000L
z = None
```

Note the last example. Python has a special value called `None`. It is generally used to represent a null value. Variable names are case sensitive. Variables can change type, simply by assigning them a new value of a different type.

```
x = 1
x = "string value"
```

Typing a variable name by itself at the interactive prompt results in its value or information about it being printed out (unless you are in PyRAF and type the name of an IRAF task, in which case CL emulation mode is entered and the task runs with no command-line arguments). Unlike the IRAF CL, no equal-sign (=) is needed to inspect a Python variable:

```
>>> x = 1
>>> x
1
>>> x+3
4
```

One exception to the rule is that if the value of the expression is `None` then nothing is printed.

A common error is to expect the same behavior in Python scripts. In a script nothing will be printed for the above example; if you want a script to print a value, use the `print` statement. Another common error is to leave off parentheses for a Python function. For example,

```
>>> raw_input()
```

reads a line of input from stdin, but if you type

```
>>> raw_input
<built-in function raw_input>
```

you are simply inspecting the `raw_input` variable, which as the message says is a Python built-in function.

2 Braces? We Don't Need No Stinking Braces!

Indentation counts! (and it is all that counts.)

This is perhaps the most controversial feature of Python for newcomers. Python does not use begin/end statements or braces to denote statement blocks. Instead, indentation is used to delimit blocks. Everything at the same level of indentation is considered in the same block. For example,

```
if x > 0:
   print "x is greater than 0"
   print "but I don't care"
if y > 0:
   print "y is greater than 0"
```

The first if block consists of two print statements, and the second consists of one. The amount of indentation does not matter (as long as it increases for nested blocks), but it must be consistent within a block.

The main source of problems with indentation as a blocking scheme is inconsistent mixing of tabs and spaces. Python treats tabs as 8 spaces. If you use an editor that treats tabs as anything but 8 spaces, and you mix spaces and tabs, you may see indentations in your editor's display that appear identical but are different as far as Python is concerned. So the general rule is: **Don't mix tabs and spaces for indentation.** Use one or the other exclusively.

It may take a little time to adjust to the use of indentation for blocking, but if your experience is like ours, you will soon be wondering why all programming languages don't work this way. It leads to clear, easy-to-read code and is far less likely to lead to errors in blocking than approaches that use braces.

3 Python Data Structures

Python has a few, very useful, built-in data structures that you will see used everywhere.

3.1 Strings

The simplest and most familiar data structure is a string. String constants can be written with either single or double quotes. Python has a number of built-in string operations. Strings can be indexed and slices extracted using a simple subscripting notation. For example,

```
>>> x = 'test string'
>>> x[2]
's'
>>> x[1:3]
'es'
>>> x[-2]
'n'
>>> x[2:]
'st string'
>>> x[:2]
'te'
```

These few examples illustrate some unusual features. First, indexing is 0 based. The first element of a N-character string is number 0, and the last character is number N-1. (This is familiar to C and IDL users but differs from IRAF and Fortran.)

When a range (or "slice" in Python terminology) is specified, the second number represents an index that is not included as part of the range. One should view indexing as working like this

```
 0   1   2   3   4   5   6   7   8   9   10  11 positive indices
   t   e   s   t       s   t   r   i   n   g
-11 -10 -9  -8  -7  -6  -5  -4  -3  -2  -1     negative indices
```

The indices effectively mark the gaps between the items in the string or list. Specifying `2:4` means everything between 2 and 4. Python sequences (including strings) can be indexed in reverse order as well. The index -1 represents the last element, -2 the penultimate element, and so forth. If the first index in a slice is omitted, it defaults to the beginning of the string; if the last is omitted, it defaults to the end of the string. So `s[-4:]` contains the last 4 elements of the string (`'ring'` in the above example).

Strings can be concatenated using the addition operator

```
>>> print "hello" + " world"
'hello world'
```

And they can be replicated using the multiplication operator

```
>>> print "hello"*5
'hellohellohellohellohello'
```

There is also a string module (see below for more on modules) in the standard Python library. It provides many additional operations on strings. In the latest version of Python (2.0), strings also have methods that can be used for further manipulations. For example `s.find('abc')` returns the (zero-based) index of the first occurence of the substring `'abc'` in string `s` (-1 if it is not found.) The string module equivalent is `string.find(s,'abc')`.

3.2 Lists

One can view lists as generalized, dynamically sized arrays. A list may contain a sequence of any legitimate Python objects: numbers, strings functions, objects, and even other lists. The objects in a list do not have to be the same type. A list can be specifically constructed using square brackets and commas:

```
x = [1,4,9,"first three integer squares"]
```

Elements of lists can be accessed with subscripts and slices just like strings. E.g.,

```
>>> x[2]
9
>>> x[2:]
[9,"first three integer squares"]
```

They can have items appended to them:

```
>>> x.append('new end of list')
  or
>>> x = x + ['new end of list'] # i.e., '+' concatenates lists
>>> x
[1,4,9,'first three integer squares','new end of list']
```

(Here # is the Python comment delimiter.) Elements can be deleted and inserted:

```
>>> del x[3]
>>> x
[1, 4, 9, 'new end of list']
>>> x.insert(1,'two')
>>> x
[1, 'two', 4, 9, 'new end of list']
```

The product of a list and a constant is the result of repeatedly concatenating the list to itself:

```
>>> 2*x[0:3]
[1, 'two', 4, 1, 'two', 4]
```

There are a number of other operations possible including sorting and reversing:

```
>>> x = [5, 3, 1, 2, 4, 6]
>>> x.sort()
>>> print x
[1, 2, 3, 4, 5, 6]
>>> x.reverse()
>>> print x
[6, 5, 4, 3, 2, 1]
```

A list can have any number of elements, including zero: `[]` represents the empty list.

3.3 Mutability

Python data structures are either mutable (changeable) or not. Strings are not mutable – once created they cannot be modified *in situ* (though it is easy to create new, modified versions). Lists are mutable, which means lists can be changed after being created; a particular element of a list may be modified. Either single elements or slices of lists can be modified by assignment:

```
>>> x
[1,4,9,'new end of list']
>>> x[2] = 16
>>> x
[1,4,16,'new end of list']
>>> x[2:3] = [9,16,25]
[1,4,9,16,25,'new end of list']
```

3.4 Tuples

Tuples are essentially immutable lists. There is a different notation used to construct tuples (parentheses instead of brackets), but otherwise the same operations apply to them as to lists as long as the operation does not change the tuple. (For example, `sort`, `reverse` and `del` cannot be used.)

```
x = (1,"string")
x = ()   # empty tuple
x = (2,) # one element tuple requires a trailing comma (which is legal
         # for any length tuple also) to distinguish them from expressions
```

3.5 Dictionaries

Dictionaries are hash tables that allow elements to be retrieved by a key. The key can be any Python immutable type such as an integer, string, or tuple. An example would be:

```
>>> employee_id = "ricky":11,"fred":12,"ethel":15
>>> print employee_id["fred"]
12
```

Note that braces are used to enclose dictionary definitions. New items are easily added:

```
>>> employee_id["lucy"] = 16
```

will create a new entry.

There are many operations available for dictionaries; for example, you can get a list of all the keys:

```
>>> print employee_id.keys()
['ricky', 'lucy', 'ethel', 'fred']
```

The order of the keys is random, but you can sort the list if you like. You can delete entries:

```
>>> del employee_id['fred']
>>> print employee_id.keys()
['ricky', 'lucy', 'ethel']
```

It is a rare Python program that doesn't use these basic data structures (strings, lists, and dictionaries) routinely and heavily. They can be mixed and matched in any way you wish. Dictionaries can contain lists, tuples, and other dictionaries (and different entries can contain different types of objects), and the same is true of lists and tuples.

4 Truth and Consequences

There are no explicit boolean types in Python. Any variable or expression can be tested for truth in an if statement. The rules for whether something is considered true or false are:

- `None` is false.
- Numeric types:
  - Nonzero values are true.
  - Zero values (`0`, `0L`, `0.0`, complex `0+0j`) are false.
- Strings, lists, tuples, dictionaries:
  - Empty values are false, so `""`, `[]`, `()`, and `{}` are all false.
  - Non-empty values are true (even if filled with zeros or `None`'s, for example, `[None]` is true. `""` is false, but the string `"false"` is true).
- Other objects:
  - All other objects are considered true (unless the programmer has done some fancy stuff to make them test as false). For example, after `fh = open('filename')`, the filehandle `fh` is true (even if the file is closed with `fh.close()`).

One cannot (mistakenly or deliberately) put an assignment statement inside a conditional as one can in C. For example the following is not legal Python:

```
if x = 0:
   print "did I really mean to write a never used print statement?"
```

But this is legal:

```
if x == 0:
   print "much better"
```

5 Control Constructs

Python has if, for, and while control statements. The for statement is different than that in most other languages; it is more like the `foreach` loop in `csh`. It takes the following form:

```
for item in itemlist:
  print item
```

The loop body is executed with a new value for the variable `item` for each iteration. The values are taken from a sequence-like object (such as a string, list, or tuple ... but other possibilities exist), and `item` is set to each of the values in the sequence.

Note the colon after the statement – all statements that control the execution of a following block of statements (including for, if, while, etc.) end with a colon.

To get the common loop over consecutive integers, use the built-in `range` function:

```
for i in range(100): print i
```

`range(100)` constructs a list of values from 0 to 99 (yes, 99!) and the for loop repeats 100 times with values of `i` starting at 0 and ending at 99. The argument to `range` is the number of elements in the returned list, not the maximum value. There are additional arguments to `range` if you want a loop that starts at a value other than zero or that increments by a value other than 1.

6 Modules and Namespaces

There are different ways of loading modules (the Python equivalent of libraries.) How you load them affects how you use them, and there are some important details to remember. For example one may load the standard string module by

```
>>> import string
>>> x = 'my test string'
>>> print string.capitalize(x)
MY TEST STRING
```

If you are testing a module you have written and have changed it, importing the revised module a second time, even from within a completely different program, has no effect (Python notices that it has already been imported and doesn't bother to read and execute it again). To reload a module that was already imported, type:

```
>>> reload(mmm)
```

where `mmm` (with no quotation marks) is the name of your Python module that you wish to reload.

You can import using an alternate form:

```
>>> from string import *
>>> print capitalize(s)
```

Note that with this form of import, you do not prepend the module name to the function name, which makes using the `capitalize` function a bit more convenient. But this approach does have drawbacks. All the string module names appear in the user namespace. Importing many modules this way greatly increases the possibility of name collisions. If you import a module you are developing and want to reload an edited version, importing this way makes it very difficult to reload (it's possible, but usually too tedious to be worthwhile). So, **when debugging Python modules or scripts interactively, don't use `from mmm import *`!**

A better way to use the `from ... import ...` form of the import statement is to specify explicitly which names you want to import:

```
>>> from string import capitalize
>>> print capitalize(s)
```

This avoids cluttering your namespace with functions and variables that you do not use. Namespaces in general are a large and important topic in Python but are mainly beyond the scope of this quick overview. They are ubiquitous – there are namespaces associated with functions, modules, and class instances, and each such object has a local and a global namespace.

7 Objects, Classes and What They Mean to You

If you are an object-oriented programmer, you'll find Python a pleasure to use: it provides a well-integrated object model with pretty much all of the tools you expect in an object-oriented language. But nothing forces you to write classes in Python. You can write traditional procedural code just fine if you think OO stuff is for namby pamby dweebs. Nonetheless, it is helpful to know a bit about it since many of the the standard libraries are written with objects in mind. Learning to use objects is much easier than learning how to write good classes.

Typically an object is created by calling a function (or at least something that looks like a function) and assigning the return value to a variable. Thereafter you may access and modify attributes of the object (its local variables, in effect). You can perform operations on the object (or have it perform actions) by calling its 'methods'. File handles provide a good example of what we mean.

```
>>> fin = open('input.txt','r') # open file in read mode and return file handle
>>> lines = fin.readlines()     # call a method that returns a list of lines
>>> for line in lines: print line
>>> print fin.name              # print name of the file for the fin object
                                # 'name' is an attribute of the file object
>>> fin.close()                 # close the file
```

This approach is used by many Python libraries.

8 Defining Functions

Defining functions is easy:

```
def factorial(n):
    """a simple factorial function with no overflow protection"""
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

Note that, like other code blocks, indentation is used to decide what belongs in the function body. Also note that Python allows recursive functions.

The string at the beginning of the function is called a "doc string" and contains documentation information on the function. It is available as the `__doc__` attribute of the function, so `print factorial.__doc__` will print the string for the example function. The doc string can have multiple lines and is by convention enclosed in triple quotes (which allow strings that extend across lines) even when it contains only a single line, as in the sample.

Python provides a great deal of flexibility in handling function arguments. Check books or references to see how to handle default values, keyword arguments and a variable number of arguments.

9 But Wait! There's More!

There is a lot more to Python than shown in this brief overview. For more of our introductory material, see these PDF versions of a Python tutorial we presented at STScI:

- *PDF Slides*
- *PDF Slides (b&w)*

Python's object system provides great flexibility and allows operator overloading, special syntax enhancement (for example, making subscripting behave in a special way for a new class), introspection features (allowing run-time examination of objects to determine their characteristics), and all sorts of other goodies. But explaining these really does require a book.

Python also has many modules (libraries if you will) available to do string manipulation, regular expression matching, operating system access, etc. See the Python library reference and books listed below for documentation of the standard library. One interesting module is Numeric, which adds a numeric array data type to the language and allows easy array computations. We are not currently using many of Numeric's capabilities but intend to make it a focus of much of our future enhancements of PyRAF. In fact, we are in the process of rewriting the Numeric module (as *numarray*) to make it more flexible and maintainable. We are also developing a Python FITS module, *pyfits*, that provides powerful but intuitive access to FITS files.

10 Python Documentation and Books

10.1 Books

At the beginning of 1999 there was little to choose from in the way of Python books. There were really only two English language books and neither was an ideal introduction to the language. Two years later the situation is quite different. Numerous books have been published and more are on the way. We will only mention some of the available books here. Those wishing to see the whole list should visit the Python web site "bookstore".

Currently we feel that the following two books are the best introductions to Python:

- *Learning Python* by Lutz and Ascher. Published by O'Reilly (ISBN: 1-5659-2464-9).

- *Quick Python* by Harms and McDonald. Published by Manning (ISBN: 1-8847-7774-0).

Do not confuse *Learning Python* with *Programming in Python* by Lutz (also an O'Reilly book) which is older and much longer. In our view, the older book is poorly organized and not very convenient to use.

The standard Python documentation may also be purchased as bound books (but is available for free online in html, postscript or pdf form). Check the PSA bookstore link provided above to see how to order the hardcopy versions.

People may find the following books useful:

- *Python Essential Reference* by Beazley. Published by New Riders (ISBN: 0-7357-0901-7). Condenses the standard documentation into a compact book. Very nice reference book.
- *(the eff-bot guide to) The Standard Python Library* by Lundh. (On-line only, see above PSA bookstore link to find how to obtain it.) Provides many short examples of how to use the standard library, but is not an alternate reference to the standard library.
- *Python Pocket Reference* by Lutz. Published by O'Reilly (ISBN: 1-5659-2500-9). A very compact, inexpensive book that serves as a basic reference for Python (and does fit in your pocket). Not nearly as complete as *Python Essential Reference*, however.

10.2 Online Documentation

The Standard Python Distribution is generally documented quite well. The documentation may be obtained from www.python.org/doc/ in various formats. Available are the following:

Document	**Comment**
Tutorial	Introduction to Python
Library Reference	Essential reference
Language Reference	Should not be needed by most
Extending and Embedding	For those interested in accessing C or C++ code from Python, or Python from C or C++.
Python/C API	More information on the Python/C interface.

There are also a number of Topic guides and How-to's available. Those making extensive use of regular expressions may find *Mastering Regular Expressions* by Friedl (O'Reilly, ISBN: 1-5659-2257-3) useful, though its references to Python regular expressions are now obsolete (Python regular expressions now behave much more like those of Perl).

Tkinter is currently the most widely used GUI toolkit library available for Python, and there is now a good book that describes how to use it: *Python and Tkinter Programming* by Grayson, published by Manning (ISBN: 1-8847-7781-3). There is also on-line documentation by Fredrik Lundh available, though it is not as complete as the book. Most will also benefit from Tk documentation or books (particularly *Practical Programming in Tcl and Tk* by Welch).

wxPython is another cross-platform GUI library that is gaining momentum. Although a book is in the works, it will not be available until next year. See the wxPython web site for details.

Numeric, the module that provides arrays for efficient numerical programming, is not part of the standard distribution, but we expect to use it heavily in the future. PyRAF already makes use of Numeric for some array conversion functions. The current documentation for Numeric may be found at http://numpy.sourceforge.net. Our rewrite of Numeric, numarray, is available at http://stsdas.stsci.edu/numarray, and our Python FITS module pyfits is available at http://stsdas.stsci.edu/pyfits.