

Python Tutorial Notes

Jacob Lee, SigPLAN

February 21, 2004

1 Introduction

Python is a modern programming language with a wide range of applications, from simple scripts to large-scale software development. Python's most widespread niche is scripting and rapid prototyping, but it is also used for everything from web application frameworks (such as Zope) to graphics programming (such as PyGame). Well-written Python programs are readable, elegant, and simple; they also tend to be shorter than implementations in other, less dynamic, languages.

Whitespace

The most immediate difference most programmers notice when seeing python code for the first time is that blocks are delimited not by braces (like in C) or by keywords (like in BASIC), but by whitespace and indentation. Many people never get over this fact and categorically dismiss Python because of its syntax. Pythonistas point out that good programmers indent anyway, and eliminating braces eliminates the class of errors that occur when braces and indentation disagree. Personally, I find that Python's syntax is clean and elegant; I encourage you to simply try writing Python and find out if the indentation is actually a hurdle.

Types

Python's type system is *strong* and *dynamic*. Strong typing means that, for example, objects will not be converted from one type to another implicitly. Dynamic typing means that the type of an object is not explicitly specified: it is implied by the code.

```
>>> s = 'spam'
>>> type(s)
<type 'str'>
>>> s = 5 # This is ok! (but possibly bad form for a real program)
<type 'int'>
>>> 5 + '14' # Should this be '514' or 19?
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(5) + '14'
'514'
>>> 5 + int('14')
19

```

To identify where this fits into the grand scheme of type systems, here is a comparative chart:

	Static	Dynamic
Weak	C	PHP, Perl
Strong	Java	Python

2 Lists

Lists are similar to arrays in C, but they are of arbitrary length and can hold arbitrary data.

```

>>> l = [0,1,2,3]
>>> l.append(4)
>>> l
[0, 1, 2, 3, 4]
>>> l = ['spam', 2, 3, 'eggs'] # Observe the dynamic types at work
>>> l + [1,2]
['spam', 2, 3, 'eggs', 1, 2]
>>> range(5) # Why does this stop at 4? See below on ‘slices’
[0, 1, 2, 3, 4]
>>> len(l)
5

```

List elements are indexed beginning with zero. Negative indices will count backwards from the end of a list.

```

>>> l[0]
'spam'
>>> l[-1]
2

```

Slices

Slice notation is a way to return segments of a larger list.

```

>>> l[2:4]
[3, 'eggs']

```

If you omit the start or end element, it will default to the beginning or end of the list.

```
>>> l[:3]
['spam', 2, 3]
>>> l[3:]
['eggs']
```

Note that the start element is inclusive and the end element is exclusive. This allows slice ranges to obey the properties $l[:n] + l[n:] = l$ and $\text{len}(\text{range}(n)) = n$.

If you omit both the start and end of a list (by specifying $l[:]$), you get a copy of the list. This will become relevant later.

Useful List Functions

```
>>> l.insert(0, 'more spam')
>>> l
['more spam', 'spam', 2, 3, 'eggs', 1, 2]
>>> l2 = [3,4]
>>> l.extend(l2)
>>> l
['more spam', 'spam', 2, 3, 'eggs', 1, 2, 3, 4]
>>> l.append(l2) # Observe the difference!
>>> l
['more spam', 'spam', 2, 3, 'eggs', 1, 2, 3, 4, [3, 4]]
```

Note that nothing special needs to be done to allow a list to store a list (or indeed any other data type). What else can we do with lists? The interpreter can help us out:

```
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delslice__', '__doc__', '__eq__', '__ge__', '__getattr__',
 ... <removed for clarity> ...
 '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
>>> help(l.count)
Help on built-in function count:
```

```
count(...)
    L.count(value) -> integer -- return number of occurrences of value
```

The items surrounded by double-underscores are special methods of lists that you don't need to worry about for now.

3 Iteration (i.e. Looping)

```
for item in l:
    print item
```

```

# len is the built-in function to return the length of a sequence
for index in range(len(l)):
    print index

# Despamify the program's input
import sys
for line in sys.stdin.readlines(): # This technically uses iterators
    if line[:4] == 'spam':
        print line[4:]

while not done:
    # do some stuff
    if some_condition():
        done = 1

```

Like C, Python has break and continue statements.

```

while 1:
    # do stuff
    if some_condition():
        continue # Go back to the beginning of the loop
    # do more stuff
    if other_condition():
        break # Leave the loop

```

We can loop over any sequence type: strings '012', lists [0,1,2], tuples (0,1,2), dictionaries (associative arrays) {'a':5, 'b':4, 'c':1}, and iterators.

```

>>> for i in 'spam':
...     print i, # the comma prevents print from ending the line
s p a m
>>> t = (0,1,2,3) # t is a tuple
>>> for i in t:
...     print t,
0 1 2 3

```

Before we get to the other data structures, we need to take a detour and talk about mutability.

4 Mutability

Put simply, a mutable object can be changed, and an immutable object cannot. Lists and dictionaries are mutable; strings and tuples are not.

```

>>> l = [0,1,2]

```

```

>>> l[1] = 5 # ok
>>> t = (0,1,2)
>>> t[1] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment

```

Importantly, mutable types cannot be used as the key in a dictionary. Tuples can be used as dictionary keys, and they are often preferable to lists in instances when the size is known (e.g. to specify (x,y) coordinates).

Also note that the equals sign does not make a copy of an object. For example:

```

>>> l = [0,1,2]
>>> l2 = l
>>> l2[1] = 100
>>> l
[0, 100, 2]

```

This is because Python assignment works by *object reference*. That is, the equal sign in this case means “l2 now refers to the same object as l”. To demonstrate this, use the `id(obj)` function to observe that l and l2 refer to the same place in memory. In this example, if we wanted to make a copy of l, we could either create a slice of the entire list: `l2 = l[:]`, or we could explicitly create a new list object: `l2 = list(l)`.

5 Dictionaries

Dictionaries are like unordered lists with arbitrary keys. They are often called associative arrays, mappings, or hashtables (indeed, they are internally implemented using a hashtable).

```

>>> d = {} # Create an empty dictionary
>>> d = {'eggs':2, 5:3, 4:4} # Again, observe the dynamic typing
>>> d['eggs']
2
>>> d['eggs'] = 5
>>> d[(1,2)] = 'foo'
>>> l = [1,2,3]
>>> d[l] = 'foo' # No!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
>>> d.keys() # Observe that the order here is arbitrary
['eggs', (1, 2), 4, 5]
>>> d.values()

```

```
[2, 'foo', 4, 3]
>>> d.items()
[('eggs', 2), ((1, 2), 'foo'), (4, 4), (5, 3)]
```

You can use the `.keys()`, `.items()`, and `.values()` methods to iterate over a dictionary. The following snippets of code behave identically, and both are equally Pythonic idioms:

```
for i in d.keys():
    print i, d[i]

for key, value in d.items():
    print key, value
```

6 Functions

```
>>> def f(x):
...     return x**2 # The double-asterisk means ‘to the power of’
>>> f(5)
25
>>> def sum(x, y=5): # Default arguments
...     return x + y
>>> sum(10, 15)
25
>>> sum(10)
15
>>> sum(x=10, y=15) # Keyword arguments
25
```

7 The Standard Library

Python comes with a large number of modules built-in. Use “import” to import modules from the standard library and to organize larger projects (see `derive.py`, below)

```
>>> import math
>>> math.cos(2*math.pi)
1.0
>>> from random import randrange, random
>>> randrange(10)
3
>>> random()
0.53700871252175475
>>> from threading import *
>>> my_lock = Lock() # From threading
```

See <http://www.python.org/doc> for a description of the entire standard library; in a pinch, `dir()` and `help()` are your friends.

8 Classes

File 'classes.py':

```
class C:
    def __init__(self, x, y=5):
        self.x = x
        self.y = 5

    def value(self):
        return self.x, self.y

>>> from classes import C
>>> c = C(10)
>>> print c.value()
(10, 5)
>>> d = C()
>>> d.xzyzzy = 1 # Dynamic typing, yet again
>>> d.xzyzzy
1
>>> c.xzyzzy
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: C instance has no attribute 'xzyzzy'
```

Inheritance

File 'derive.py':

```
from classes import C # This looks for classes.py in the same directory
```

```
class D(C):
    def __init__(self, x, y=5, z=10):
        C.__init__(self, x, y)
        self.z = 10

    def value(self):
        return x, y, z
```

A derived class can override all, some, or none of the parent class' methods. Python has multiple inheritance, too:

```
class AnotherClass(D, SomeOtherClass):
    pass # A real class would do a little more here
```

"pass", by the way, is a statement that means "do nothing".

9 Advanced Topics

Functional Programming

Functions are variables that can be assigned, passed to functions, etc.

```
>>> def f(x,y):
...     return x+y
>>> g = f
>>> g(5,3)
8
>>> map (f, [0,1,2,3], [5,6,7,8])
[5, 7, 9, 11]
>>> f = lambda(x,y): x+y # Lambdas are unnamed ('anonymous') functions
>>> reduce (lambda x,y: x+y, [0,1,2,3]) # Reduce a list to a single value
6
```

The following code snippet demonstrates using `apply` to call a function given as a parameter.

```
import time

# args is a list and kwargs is a dictionary (for keyword arguments)
def timer (f, args, kwargs):
    t = time.clock()
    val = apply (f, args, kwargs)
    print time.clock() - t
    return val
```

(Instead of `apply`, you can also use a syntax meant to deal with arbitrary parameter lists: `f(*args, **kwargs)`.) Now we'll extend this concept a bit further by implementing `time()` as a wrapper function, then using that wrapper function to compare the running times of two different functions, `square1` and `square2`.

File 'clock.py':

```
import time

def print_time(f):
    def wrapper(*args, **kwargs):
        t = time.clock()
        val = f(*args, **kwargs)
        # f.__name__ is a magic member of the function object, containing
        # the function's name
        # The % operator substitutes a tuple of variables into a string,
        # using a syntax similar to C's printf function
        print '%s took %f seconds' % (f.__name__, time.clock() - t)
    return val
```

```

        return wrapper

def square(x):
    return x*x

def square1(max):
    return map(square, range(max))
square1 = print_time(square1)

def square2(max):
    l = []
    for i in range(max):
        l.append(square(i))
    return l
square2 = print_time(square2)

if __name__ == '__main__':
    print_time(square)(500)
    square1(1000000)
    square2(1000000)

```

Decorators Starting in Python 2.4, we can implement the same code with a slightly more convenient syntax:

```

from clock.py import print_time, square # Use the above functions

@print_time
def square1(max):
    return map(square, range(max))

@print_time
def square2(max):
    l = []
    for i in range(max):
        l.append(square(i))
    return l

```

The `@print_time` syntax is equivalent to inserting the line `square2 = print_time(square2)` after the function definition. Decorators and wrapper functions can be used for all sorts of benign purposes, such as the above, as well as for far more subtle and fiendish behavior. For example, it would not be too difficult to write wrapper functions that allow the following function definition:

```

@require(int, int, list)
@return(int)

```

```
def some_function_that_is_now_type_checked(var1, var2, var3):
    return var3.append(var1 + var2)
```

Generators

The `yield` statement acts like a return statement; however, when the function is called again, it will resume its flow at the line following the `yield` statement, with its local variables intact.

File 'gen.py':

```
from __future__ import generators # Necessary in Python 2.3

def fib(max):
    '''Generator for the sequence of fibonacci numbers f_0 through f_max'''
    a, b, i = 1, 1, 0

    # Base cases
    yield 1
    yield 1

    while i < max - 2:
        c = a + b
        yield c
        a = b
        b = c
        i += 1
```

Iterators

A class is an iterator if it implements the methods `next()` and `__iter__()`. `__iter__` returns an iterator object, so if a class is its own iterator, `__iter__` should return `self`.

File 'reverse.py':

```
class reverse:
    def __init__(self, list):
        self._l = list
        self._pos = 0

    def next(self):
        if self._pos == -len(self._l):
            raise StopIteration
        else:
            self._pos -= 1
            return self._l[self._pos]

# __iter__ should return an iterator over the class;
```

```

    # this class is its own iterator.
    def __iter__ (self):
        return self

>>> l = range(4)
>>> for i in reverse(l):
...     print i,
3 2 1 0

```

If we want, we can use this code to implement a list class that behaves identically to a normal list except that it iterates in reverse.

```

class rlist (list):
    def __iter__ (self):
        return reverse (self)

>>> l = rlist([0,1,2,3])
>>> for i in l:
...     print i,
3 2 1 0

```

Magic Methods and Members

Names surrounded by double-underscores have special meaning in Python classes. Many are supplied to an object by default, and almost all can be overridden by the user. Here is a partial implementation of a complex number class. Note that Python already has complex numbers as a built-in datatype, so this is probably a bad example (although it is a traditional example for other languages).

```

class Complex:
    def __init__ (self, a=0, b=0):
        self._a = a
        self._b = b

    # Overload the '+' operator
    def __add__ (self, other):
        return Complex (self._a + other._b,
                        self._a + other._b)

    # Return a string representation of a class
    def __str__ (self):
        return '%f + %f' % (self._a, self._b)

```

You can also override functions like `__getattr__` and `__setattr__` to create user-defined data types that act like lists or dictionaries.

List Comprehensions and Generator Expressions

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> [x**2 for x in range(5) if x%2 == 0]
[0, 4, 16]
```

List comprehensions are a more powerful replacement for functions like `map` that apply a function across an entire list. For example, the first statement above could also have been written as `map(lambda(x): x**2, range(5))`.

List comprehensions build an entire list in memory, which is wasteful if you are only interested in a single value. If that is the case, you can use generator expressions, which achieve similar results using generators instead of lists. Generator expressions do for functions like `sum` and `reduce`, which collapse a list (or technically any iterable object, which is why this example works) into a single value what list comprehensions do for functions like `map` and `filter`. For example:

```
>>> sum(x**2 for x in range(5) if x%2 == 0)
20
>>> gen = (x**2 for x in range(5))
>>> gen # Find out what exactly generator expressions are
<generator object at 0x401e508c>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
4
```