Think Python (and other useful things)

Allen B. Downey (and Laurel Farris)

May 8, 2016

http://www.greenteapress.com/thinkpython/html/index.html

8 Strings

8.1 A string is a sequence

8.2 `len`

8.3 Traversal with a `for` loop

8.4 String slices

8.5 Strings are immutable

8.6 Searching

8.7 Looping and counting

8.8 String methods

A **method** is similar to a function - it takes arguments and returns a value, but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking `upper` on the `word`.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2` (not including `2`).

8.9 The `in` operator

8.10 String comparison

8.11 Debugging

9 Case study: word play

10 Lists

10.1 A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements** or sometimes **items**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets:

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an *empty list*; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2 Lists are mutable

The syntax for *accessing* the elements of a list is the same as for accessing the characters of a string - the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print cheeses[0]
Cheddar
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

The one-eth element of numbers, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a **mapping**; each index "maps to" one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 Traversing a list

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A for `loop` over an *empty* list never executes the body:

```
for x in []:
    print 'This never happens.'
```

Although a list can contain another list, *the nested list still counts as a single element.* The length of this list is 4:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 List operations

IMPORTANT: these are not mathematical operations for lists. Convert to numpy arrays: `a = np.array(a)`

The `+` operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the `*` operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1,2,3]` three times.

10.5 List slices

10.6 List methods

10.7 Map, filter, and reduce

10.8 Deleting elements

10.9 Lists and strings

10.10 Objects and values

10.11 Aliasing

10.12 List arguments

10.13 Debugging

11 Dictionaries

12 Tuples

13 Case study: data structure selection

14 Files

http://www.greenteapress.com/thinkpython/html/thinkpython015.html

14.1 Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2 Reading and writing

```
file:
f=open('file.txt','[rwa]') [read,write,append]
s=f.readline()
f.close(),
alternatively:
for line in f:
  print line
```

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode 'w' as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method puts data into the file.

```
>>> line1 = ‘‘This here’s the wattle,\n’’
>>> fout.write(line1)
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
```

When you are done writing, you have to close the file.

```
>>> fout.close()
```

14.3 Format operator

```
’’’
Formatted output
 > print ’the number is {:.#e|:nd|:n.nf|:ns}’.format(x)
    exponential, integer, float, string
 General syntax:
’’’
template.format(var_1,var_2,...var_n)
# template:
#’{[field][!conversion]:[spec]}’
field = index of variables listed in .format()
conversion = int, float, string, etc.
spec = specifier
[[fill]align][sign][#][0][minwidth][.prec][type]
align:
<(left,default)
>(right)
=(padding after sign, before digits)
^(center)
0: zero padding (same as ’=’ and fill char of 0)
type: e,f,g(general)

# Including text:
print ’Variable 2 is {1} and variable 1 is {0}’.format(var_1,var_2)

’{:8d}asdfad{:8.2f}’.format(i,f)
formats: {:nd},{:n.nf},{:ns}
#    n-number (width.precision), d-integer, f-float, s-string
#-------------- How to use a variable for width?? ------------------#
```

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as an integer (`d` stands for "decimal"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at http://docs.python.org/2/library/stdtypes.html#string-formatting.

14.4 Filenames and paths

15 Classes and objects

16 Classes and functions

17 Classes and methods

18 Inheritance

19 Case study: Tkinter

20 Debugging

21 Analysis of Algorithms

22 Lumpy

Glossary

accumulator A variable used in a loop to add up or accumulate a result.

aliasing A circumstance where two or more variables refer to the same object.

augmented assignment A statement that updates the value of a variable using an operator like +=.

delimiter A character or string used to indicate where a string should be split.

element One of the values in a list (or other sequence), also called items.

equivalent Having the same value.

filter A processing pattern that traverses a list and selects the elements that satisfy some criterion.

identical Being the same object (which implies equivalence).

index An integer value that indicates an element in a list.

list A sequence of values.

list traversal The sequential accessing of each element in a list.

map A processing pattern that traverses a sequence and performs an operation on each element.

mapping A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.

nested list A list that is an element of another list.

object Something a variable can refer to. An object has a type and a value.

reduce A processing pattern that traverses a sequence and accumulates the elements into a single result.

reference The association between a variable and its value.

```
#export PYTHONPATH="${PYTHONPATH}:/new/path

# not compiled, execute with cl> python program_name.py
#   or cl> ./program_name.py IF you have the top line in your program
#     (must be the first line, literally)

# Resources
astropy.org
stackoverflow
rgex
```

```
# 'pickle' is the equivalent of IDL's 'save'

# To allow import:
def main():
    # program statements
    print "hello world."
if __name__=="__main__":  # flexible way of running routines
    main()


# Things to import (packages?)
import math [as shortername]
print math.sqrt(4)
circumference = 2*math.pi*radius
print math.exp(2) --> get e^2

import pdb
pdb.set_trace() #equivalent of 'STOP' in IDL

# line continuation:
implicit continuation using expression in parentheses...?

# variables: not declared

# lists... similar to arrays, but can't do mathematical operations
x = [0,1,2,3,4]
x = range(5) # Same as above... note that there are 5 elements, not
             # including the number 5

# numerical arrays
import numpy as np
np.zeros(n,m)
np.array(n) # creates an array that consists of n, NOT LENGTH n
np.array([[a,b,c,d][e,f,g]])
np.arange(n) # [0,1,2,...,n-1]
np.ndarray(...) # two-dimensional array
np.argmax(x) # returns INDEX of max value of x array
np.append(array,what_to_append)
np.linspace
np.logspace
np.sum(array)
np.roll(x,2) # (~IDL's SHIFT); shifts array elements 2 to the right
  --> [3,4,0,1,2] # note x is still the same, unless do x = np.roll(...)

# lists
array.append(newValue)
A = [1,2,3] # --> [1,2,3]
```

```
print A*2 # --> [1,2,3,1,2,3]... not [2,4,6] as expected
B = np.array([1,2,3]) # --> [1 2 3]
B*2  # --> [2 4 6]
<<<<<<< HEAD
=======
np.ndarray(...) # two-dimensional array
np.append...?
x = np.linspace(0,9,100) #0-9 with 100 increments
y1 = x**2
# just like with the math package, can't simply do sqrt(x). Need
# np.sqrt (or math.sqrt if not dealing with arrays).
y2 = np.sqrt(x)
# Also this doesn't work:
x = [1,2,3]
y = x**2
# because x is a list, not an array... even though it's just numbers.

>>>>>>> e65b08cb7892365af863d0202be5fe03115aa38c

num_elements = len(Array)

# dynamic memory allocation

# structures
(dictionaries, see also lists)
var={'name':'test', ra:1.}
print var['name'], var['test']
dicname = { 'A':[1,2,3], 'B':[2,4,6], 'C':[3,9,15] }
print dicname
# dictionaries are indexed by keys, rather than numbers.
# Keys can be any immutable type. Can't use lists, since they can be modified.
profile = {'density':rho_r, 'mass':mass, ...}
density = np.array(profile['density'])

# structure arrays
sarray = np.zeros(nelem,
 dtype=[('name','a12'),('ra','f4'),('dec','f4')])
sarray=np.zeros(nelem,
 dtype={'names': ('name', 'ra', 'dec'),
  'formats': ('S12','f4','f4')})

# operators (add, subtract, multiply, divide, exponent)
+,-,*,/,**,+=,-=,*=,/=,++,--
# bitwise operators (and, or, xor, not)
&,|,^,!
# string operators
+, str[i1:i2] (string slice, but string is immutable)
```

```
#matrix operations

# conditionals
if (condition):
    statements
else:
    statements
    (extent of conditional statements specified by indentation)
    pass # need to have 'something' here.
if (condition):
    break
else:
    continue
''' Conditions (logical operators) '''
==, !=, >, >=, <, <=, and, or, not
```

Looping

`for i in (list):` i = value in list, NOT the index of each value

`for i in range(1,x): ...` ~ `IDL> for i=1,x-1...` i IS the index here

PYTHON IS NOT INCLUSIVE!

```
''' Looping with condition '''
while (condition):
    statements

''' Simple output '''
print 'characters', string, variable


# simple reading from terminal and file
terminal:
s=raw_input('prompt')

# higher level file reading routines
data = numpy.loadtxt(file,
 dtype=[('name','a12'),('ra',f4),('dec','f4')])

 data = astropy.io.ascii(file)
```

Plotting

http://latexcolor.com/

???

```
>>> import matplotlib.pyplot as plt
.csv file --> ?
pyplot.show()
pyplot.savefig('tmp.pdf')
  --> rewritten each time!
plt.xlabel('labelname',fontsize=14,color='red')
color='#eeefff'
```

???

axes objects

Whole figure:

```
fig = plt.figure()
fig.suptitle('bold figure suptitle',fontsize=14,fontweight='bold')
plt.xlabel('xlabel') # whole figure
plt.ylabel('xlabel') # whole figure
```

Simpler?:

```
fig, ax = plt.subplots()
```

Individual axes:

```
ax.set_xlabel('labelname',
    fontsize = 14,
    )
ax.tick_params(axis=['x'|'y'|'both'],
    labelsize='large')
ax.ticklabel_format(style='sci',


for i in range(0,some_number):
    ax = fig.add_subplot(n,m,i+1)
    # n - vertical; m - horizontal; i+1 - plot being defined as 'ax'
```

Legend

```
    ax.plot(x,y1, 'k--', label='line 1')
    ax.plot(x,y2, 'k:', label='line 2')
    ax.plot(x,y3, 'k', label='line 3')
    legend = ax.legend(loc='upper center', shadow=True)
    legend = ax.legend(loc='lower left')
```

'k' is black.