Extracted from:

Practical Programming

An Introduction to Computer Science Using Python

This PDF file contains pages extracted from Practical Programming, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Programmers

Practical Programming

An Introduction to Computer Science Using Python

Jennifer Campbell
Paul Gries
Jason Montojo
Greg Wilson

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Copyright © 2009 Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-27-1

ISBN-13: 978-1-934356-27-2

Printed on acid-free paper.

P1.0 printing, April 2009

Version: 2009-5-9

Function	**Description**
len(L)	Returns the number of items in list L
max(L)	Returns the maximum value in list L
min(L)	Returns the minimum value in list L
sum(L)	Returns the sum of the values in list L

Figure 5.5: List functions

5.3 Built-in Functions on Lists

Section 2.6, *Function Basics*, on page 32 introduced a few of Python's built-in functions. Some of these, such as len, can be applied to lists as well, as can others that we haven't seen before (see Figure 5.5). Here they are in action working on a list of the half-lives[3] of our plutonium isotopes:

Download lists/plu4.cmd

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> len(half_lives)
5
>>> max(half_lives)
376000.0
>>> min(half_lives)
14.4
>>> sum(half_lives)
406749.14000000001
```

We can use the results of the built-in functions in expressions; for example, the following code demonstrates that we can check whether an index is in range:

Download lists/plu5.cmd

```
>>> half_lives = [87.74, 24110.0, 6537.0, 14.4, 376000.0]
>>> i = 2
>>> 0 <= i < len(half_lives)
True
>>> half_lives[i]
6537.0
>>> i = 5
>>> 0 <= i < len(half_lives)
False
```

3. The half-life of a radioactive substance is the time taken for half of it to decay. After twice this time has gone by, three quarters of the material will have decayed; after three times, seven eighths, and so on.

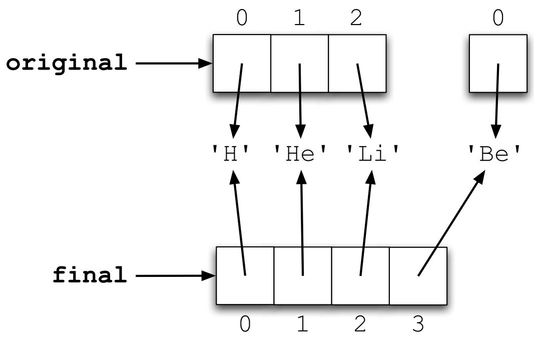

Figure 5.6: List concatenation

```
>>> half_lives[i]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Like all other objects, lists have a particular type, and Python complains if you try to combine types in inappropriate ways. Here's what happens if you try to "add" a list and a string:

Download lists/add_list_str.cmd

```
>>> ['H', 'He', 'Li'] + 'Be'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

That error report is interesting. It hints that we might be able to concatenate lists with lists to create new lists, just as we concatenated strings to create new strings. A little experimentation shows that this does in fact work:

Download lists/concat_lists.cmd

```
>>> original = ['H', 'He', 'Li']
>>> final = original + ['Be']
>>> final
['H', 'He', 'Li', 'Be']
```

As shown in Figure 5.6, this doesn't modify either of the original lists. Instead, it creates a new list whose entries refer to the entries of the original lists.

So if + works on lists, will sum work on lists of strings? After all, if sum([1, 2, 3]) is the same as 1 + 2 + 3, shouldn't sum('a', 'b', 'c') be the same as 'a' + 'b' + 'c', or 'abc'? The following code shows that the analogy can't be pushed that far:

Download lists/sum_of_str.cmd

```
>>> sum(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

On the other hand, you *can* multiply a list by an integer to get a new list containing the elements from the original list repeated a certain number of times:

Download lists/mult_lists.cmd

```
>>> metals = 'Fe Ni'.split()
>>> metals * 3
['Fe', 'Ni', 'Fe', 'Ni', 'Fe', 'Ni']
```

As with concatenation, the original list isn't modified; instead, a new list is created. Notice, by the way, how we use string.split to turn the string 'Fe Ni' into a two-element list ['Fe', 'Ni']. This is a common trick in Python programs.

5.4 Processing List Items

Lists were invented so that we wouldn't have to create 1,000 variables to store a thousand values. For the same reason, Python has a *for loop* that lets us process each element in a list in turn, without having to write one statement per element. The general form of a for loop is as follows:

```
for variable in list:
    block
```

As we saw in Section 2.6, *Function Basics*, on page 32, a block is just a sequence of one or more statements. variable and list are just a variable and a list.

When Python encounters a loop, it executes the loop's block once for each value in the list. Each pass through the block is called an *iteration*, and at the start of each iteration, Python assigns the next value in the list to the specified variable. In this way, the program can do something with each value in turn.

CLICK HERE to purchase this book now.

For example, this code prints every velocity of a falling object in metric and imperial units:

Download lists/velocity_loop.cmd

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for v in velocities:
...     print "Metric:", v, "m/sec;",
...     print "Imperial:", v * 3.28, "ft/sec"
...
Metric: 0.0 m/sec; Imperial: 0.0 ft/sec
Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec
Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec
Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec
```

Here are two other things to notice about this loop:

- In English we would say "for each velocity in the list, print the metric value, and then print the imperial value." In Python, we said roughly the same thing.
- As with function definitions, the statements in the loop block are indented. (We use four spaces in this book; check with your instructors to find out whether they prefer something else.)

In this case, we created a new variable v to store the current value taken from the list inside the loop. We could equally well have used an existing variable. If we do this, the loop still starts with the first element of the list—whatever value the variable had before the loop is lost:

Download lists/velocity_recycle.cmd

```
>>> speed = 2
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for speed in velocities:
...     print "Metric:", speed, "m/sec;",
...
Metric: 0.0 m/sec
Metric: 9.81 m/sec
Metric: 19.62 m/sec
Metric: 29.43 m/sec
>>> print "Final:", speed
Final: 29.43
```

Either way, the variable is left holding its last value when the loop finishes. Notice, by the way, that the last print statement in this program is not indented, so it is not part of the for loop. It is executed after the for loop has finished and is executed only once.

Nested Loops

We said earlier that the block of statements inside a loop could contain anything. This means that it can also contain another loop.

This program, for example, loops over the list inner once for each element of the list outer:

Download lists/nested_loops.cmd

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
...     for gas in inner:
...         print metal + gas
...
...
LiF
LiCl
LiBr
NaF
NaCl
NaBr
KF
KCl
KBr
```

If the outer loop has N_o iterations and the inner loop executes N_i times for each of them, the inner loop will execute a total of N_oN_i times. One special case of this is when the inner and outer loops are running over the same list of length N, in which case the inner loop executes N^2 times. This can be used to generate a multiplication table; after printing the header row, we use a nested loop to print each row of the table in turn, using tabs to make the columns line up:

Download lists/multiplication_table.py

```
def print_table():
    '''Print the multiplication table for numbers 1 through 5.'''

    numbers = [1, 2, 3, 4, 5]

    # Print the header row.
    for i in numbers:
        print '\t' + str(i),

    print # End the header row.

    # Print the column number and the contents of the table.
    for i in numbers:
        print i,
        for j in numbers:
            print '\t' + str(i * j),
        print # End the current row.
```

Here is print_table's output:

Download lists/multiplication_out.txt

```
>>> from multiplication_table import *
>>> print_table()
        1       2       3       4       5
1       1       2       3       4       5
2       2       4       6       8       10
3       3       6       9       12      15
4       4       8       12      16      20
5       5       10      15      20      25
```

Notice when the two different kinds of formatting are done: the print statement at the bottom of the program prints a new line when outer loop advances, while the inner loop includes a tab in front of each item.

5.5 Slicing

Geneticists describe *C. elegans* (nematodes, or microscopic worms) using three-letter short-form markers. Examples include Emb (embryonic lethality), Him (High incidence of males), Unc (Uncoordinated), Dpy (dumpy: short and fat), Sma (small), and Lon (long). We can thus keep a list:

Download lists/celegans.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

It turns out that Dpy worms and Sma worms are difficult to distinguish from each other, so they are not as useful as markers in complex strains. We can produce a new list based on celegans_markers, but without Dpy or Sma, by taking a *slice* of the list:

Download lists/celegans1.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> useful_markers = celegans_markers[0:4]
```

This creates a new list consisting of only the four distinguishable markers (see Figure 5.7, on the following page).

The first index in the slice is the starting point. The second index is *one more than* the index of the last item we want to include. More rigorously,

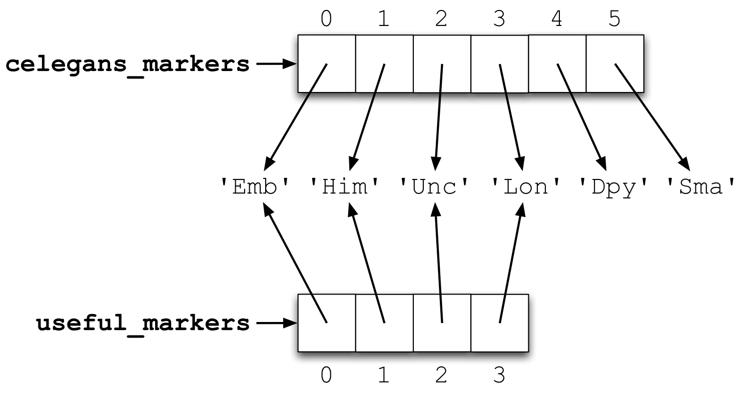

Figure 5.7: Slicing doesn't modify lists.

list[i:j] is a slice of the original list from index i (inclusive) up to, but not including, index j (exclusive).[4]

The first index can be omitted if we want to slice from the beginning of the list, and the last index can be omitted if we want to slice to the end:

Download lists/celegans2.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_markers[:4]
['Emb', 'Him', 'Unc', 'Lon']
>>> celegans_markers[4:]
['Dpy', 'Sma']
```

To create a copy of the entire list, we just omit both indices so that the "slice" runs from the start of the list to its end:

Download lists/celegans3.cmd

```
>>> celegans_markers = ['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
>>> celegans_copy = celegans_markers[:]
>>> celegans_markers[5] = 'Lvl'
>>> celegans_markers
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Lvl']
>>> celegans_copy
['Emb', 'Him', 'Unc', 'Lon', 'Dpy', 'Sma']
```

4. Python uses this convention to be consistent with the rule that the legal indices for a list go from 0 up to one less than the list's length.

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Practical Programming's Home Page
http://pragprog.com/titles/gwpy
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/gwpy.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	`support@pragprog.com`
Non-English Versions:	`translations@pragprog.com`
Pragmatic Teaching:	`academic@pragprog.com`
Author Proposals:	`proposals@pragprog.com`
Contact us:	1-800-699-PROG (+1 919 847 3884)