Python for Informatics

Exploring Information

Version 0.0.6

Charles Severance
8.3 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:

```python
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`:

```python
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:

```python
for x in empty:
    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 four:

```python
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```
8.9 Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```python
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid `l` because it looks too much like 1. So that's why I use `t`. 
The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```python
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

Once you have used `split` to break the string into a list of tokens, you can use the index operator (square bracket) to look at a particular word in the list.

You can call `split` with an optional argument called a **delimiter** which specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```python
>>> s = 'spam-spam-spam'
>>> delimiter = '-

>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```python
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = '

>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, `''`, as a delimiter.

### 8.10 Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the “interesting lines” and then parse the line to find some interesting part of the line. What if we wanted to print out the day of the week from those lines that start with “From”.

```text
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

The `split` method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with “From” and then split those lines and then print out the third word in the line:
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print words[2]

We also use the contracted form of the if statement where we put the continue on the same line as the if. This contracted form of the if functions the same as if the continue were on the next line and indented.

The program produces the following output:

Sat
Fri
Fri
Fri
...

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.
4. Lists, split, and files

When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the guardian pattern when it comes writing programs that read through a file and look for a “needle in the haystack”.

Let’s revisit our program that is looking for the day of the week on the from lines of our file:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don’t have “From” as the first word as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From': continue
    print words[2]
```

This looks much simpler and we don’t even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
python search8.py
```

```
Sat
```

```
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From': continue
IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat) but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix “Debug:” to the line so we can keep our regular output separate from our debug output.

```
for line in fhand:
    words = line.split()
    print 'Debug:', words
    if words[0] != 'From': continue
    print words[2]
```
When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```python
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From': continue
IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we split the line into words. When the program fails the list of words is empty []. If we open the file in a text editor and look at the file, at that point it looks as follows:

```plaintext
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan 5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772

The error occurs when our program encounters a blank line! Of course there are “zero words” on a blank line. Why didn’t we think of that when we were writing the code. When the code looks for the first word (word[0]) to check to see if it matches “From”, we get an “index out of range” error.

This of course is the perfect place to add some **guardian** code to avoid checking the first word if the first word is not there. There are many ways to protect this code, we will choose to check the number of words we have before we look at the first word:

```python
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print words[2]
```

First we commented out the debug print statement instead of removing it in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are “interesting” to us and which we want to process some more. A line which has no
words is “uninteresting” to us so we skip to the next line. A line which does not have “From” as its first word is uninteresting to us so we skip it.

The program as modified runs successfully so perhaps it is correct. Our guardian statement does make sure that the words[0] will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, “What might go wrong?”.

**Exercise 8.2** Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

**Exercise 8.3** Rewrite the guardian code in the above example without two if statements. Instead use a compound logical expression using the and logical operator with a single if statement.
Chapter 9

Dictionaries

A dictionary is like a list, but more general. In a list, the positions (a.k.a. indices) have to be integers; in a dictionary the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

```python
>>> eng2sp = dict()
>>> print eng2sp
{}
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```python
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value 'uno'. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```python
>>> print eng2sp
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```python
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```
But if you print `eng2sp`, you might be surprised:

```python
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```python
>>> print eng2sp['two']
'dos'
```

The key ‘two’ always maps to the value ‘dos’ so the order of the items doesn’t matter.

If the key isn’t in the dictionary, you get an exception:

```python
>>> print eng2sp['four']
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```python
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

```python
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```python
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table that has a remarkable property; the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won’t explain why hash functions are so magical, but you can read more about it at `wikipedia.org/wiki/Hash_table`. 
Exercise 9.1 Write a function that reads the words in words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.

9.1 Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An implementation is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don’t have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

```python
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c] + 1
    return d
```

The name of the function is histogram, which is a statistical term for a set of counters (or frequencies).

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].
Here’s how it works:

```python
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters ‘a’ and ‘b’ appear once; ‘o’ appears twice, and so on.

**Exercise 9.2** Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```python
>>> h = histogram('a')
>>> print h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement.

(skipping section 9.2)
9.3 Looping and dictionaries

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. For example, `print_hist` prints each key and the corresponding value:
def print_hist(h):
    for c in h:
        print c, h[c]

Here’s what the output looks like:

```python
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order.

If you want to print the keys in alphabetical order, you first make a list of the keys in the
dictionary using the `keys` method available in dictionary objects, and then sort that list and
loop through the sorted list, looking up each key printing out key/value pairs in sorted order as
follows as follows:

```python
def print_sorted_hist(h):
    lst = h.keys()
    lst.sort()
    for c in lst:
        print c, h[c]
```

Here’s what the output looks like:

```python
>>> h = histogram('parrot')
>>> print_sorted_hist(h)
a 1
o 1
p 1
r 2
t 1
```

So now the keys are in alphabetical order.

(skipping section 9.4)
9.5 Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

**Scale down the input:** If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first \( n \) lines.

If there is an error, you can reduce \( n \) to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

**Check summaries and types:** Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “insane.” Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check.”

Pretty print the output: Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debugging.

9.6 Glossary

dictionary: A mapping from a set of keys to their corresponding values.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

histogram: A set of counters.

implementation: A way of performing a computation.

item: Another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

key-value pair: The representation of the mapping from a key to a value.

lookup: A dictionary operation that takes a key and finds the corresponding value.

nested loops: When there is one or more loops “inside” of another loop. The inner loop runs to completion each time the outer loop runs once.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value.”

9.7 Exercises

Exercise 9.3 Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines which start with “From”, then look for the third word and then keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}

Exercise 9.4 Write a program to read through a mail log, and figure out who had the most messages in the file. The program looks for “From” lines and takes the second parameter on those lines as the person who sent the mail.

The program creates a Python dictionary that maps the sender’s address to the total number of messages for that person.

After all the data has been read the program looks through the dictionary using a maximum loop (see Section 5.7.2) to find who has the most messages and how many messages the person has.

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195

Exercise 9.5 This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e. the whole e-mail address). At the end of the program print out the contents of your dictionary.

python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7, 'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
Chapter 10

Tuples

10.1 Tuples are immutable

A tuple\(^1\) is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```python
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```python
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```python
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```python
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

---

\(^1\)Fun fact: The word "tuple" comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quintuple, sextuple, septuple, etc.
Another way to construct a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```python
>>> t = tuple()
```

If the argument is a sequence (string, list or tuple), the result of the call to `tuple` is a tuple with the elements of the sequence:

```python
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```python
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

And the slice operator selects a range of elements.

```python
>>> print t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```python
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```python
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

(skipping section 10.2)
10.3 Tuple assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left hand side of an assignment statement. This allows you to assign more than one variable at a time when the left hand side is a sequence.

In this example we have a two element list (which is a sequence) and assign the first and second elements of the sequence to the variables \( x \) and \( y \) in a single statement.

```python
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
```

```
'have'
>>> y
'fun'

It is not magic. Python roughly translates the tuple assignment syntax to be the following:\footnote{Python does not translate the syntax literally. For example if you try this with a dictionary it will not work as might expect.}

```python
>>> m = ['have', 'fun']
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
```

Stylistically when we use a tuple on the left hand side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```python
>>> m = ['have', 'fun']
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
```

A particularly clever application of tuple assignment allows us to \textbf{swap} the values of two variables in a single statement:

```python
>>> a, b = b, a
```

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```python
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```python
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```
10.4. Dictionaries and tuples

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```python
>>> print uname
monty
>>> print domain
python.org
```

(SKIPPING THE REST OF CHAPTER 10)