Python Programming Clarifications

April 20, 2011

1 Overview
A few questions have come up during lecture where I have either given unclear answers or stated that I would look into it and get back to the class. This document serves to clear up any misconceptions or confusion around some of the excellent questions asked during class about syntax, concepts, functions, operations, procedures, etc. Any and all questions about Python programming are welcomed and we strongly encourage students to continue asking questions in class, during office hours, or via email. If someone is confused about something I said or written, it’s likely that other people are too.

2 Code snippets
Many of the sections that appear below have snippets of Python code that students can download and test out for themselves. Below is a list of each section in the document with a link to the specific code file—if available—that is associated with a particular section.

- [:] versus assignment with a list – code03
- rules for how comparison operators work on strings – code04
- append versus extend – code05
- if...if versus if...elif
- mutable versus immutable objects – code07
- while versus for loops – code08
- dictionaries – code09
3  [:] versus assignment with a list

This question came up in lecture 2 when I was discussing methods that modify lists (see pages 63–64 in Model or slide 49 from the lecture 2). During class I gave a partially satisfactory (IMO) answer to this question about how the memory usage is different and it depends on whether you want to retain, for whatever reason, the original reference. In this case you’d use slice operator [:] over the assignment =. This is true, but probably confusing and beyond the scope of the class. Practically speaking, they do the same thing. Conceptually, however, they are quite different. Furthermore, I’ll provide an example that illustrates an issue that arises with copying an item and how assignment produces very different results from slicing.

First, the conceptual difference. Assignment is an binding operation where an object is assigned to a name (basically a reference). The slice operation allows one to extract subsets of collections based on an index (or range). It so happens that slicing without indeces, [:], will operate on the entire list, which is analogous, but not identical to assignment. Now is where the important distinction comes up. Suppose we have a list with 10 sequences from one type of bacteria (e.g. bacAseqs = ['seq1', 'seq2', ..., 'seq10']). What happens if you wanted to run a little computational experiment where you examine sequence drift in transcription by examining different mutational rates. To do this you decide it would be good to make a copy of your original bacAseqs list and then you’ll apply different rates to each list of sequences. How would you do this? A simple approach might be, bacBseqs = bacAseqs. While this might seem like a good solution, you haven’t actually created a true copy in the sense that bacAseqs and bacBseqs are not independent. Instead, they are references to the same object such that any modification to one also modifies the other. To create a true copy of bacAseqs—in the sense that the copy will be independent of its original—you need to use the slice operation, bacBseqs = bacAseqs[:]. Now any change to either list will not modify the other.

While this example was somewhat contrived, I hope it illustrated the point that these two operations are both conceptually and practically different, and furthermore that [:] is a useful approach for creating a copy of a list. As an aside, this notion of copying objects applies to other data types as well. It’s just that most other types are immutable so the copy does not modify the original, whereas, this is not the case with a mutable object such as a list (note that Table 3-5 in Model emphasizes this notion of [:] being a copy operation).

4  rules for how comparison operators work on strings

As we saw in class, the standard comparison operators (<, <=, >, >=, ==, !=) in Python can be applied to strings. The way that these comparisons work is by performing a character-by-character comparison based on using the numerical equivalent of the character (see the built-in function ord() on page 54 in Model). Thus, strings will differ as soon as the first character differs in a pairwise comparison from index 0...len(string). Strings with different lengths are made equal by appending spaces for the missing characters at the end of the string, e.g. when comparing 'dna' and 'dnasequence', the first string becomes 'dna ' for the comparison. In this example, the comparison where these strings differ is at index 3 between ' ' and 's'. An easy rule for string comparisons is, when the characters only consist of alphabetic letters, the order follows alphabetical order. Digits precede letters, and uppercase letters precede lowercase ones. For a complete description of characters and how they are represented numerically, see the ASCII code table: http://www.ascii-code.com/.

<table>
<thead>
<tr>
<th>Character</th>
<th>Numerical Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>4</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>&lt;=</td>
<td>10</td>
</tr>
<tr>
<td>t</td>
<td>116</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>Two</td>
<td>86</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>T</td>
<td>84</td>
</tr>
<tr>
<td>composer</td>
<td>o</td>
</tr>
<tr>
<td>computer</td>
<td>u</td>
</tr>
</tbody>
</table>

*Technically this is called a lexicographical comparison, but this is beyond the scope of the class so I leave it as a research problem for students who are interested in these details.

†A space has a numeric value of 32, whereas, s has the value 115 so 'dna' is less than 'dnasequence'.
5 append versus extend

Both of these methods act on lists. On page 64 in the textbook, we see a brief description of each:

- `lst.append(x)` Adds `x` to the end of `lst`
- `lst.extend(x)` Adds the elements of `x` at the end of `lst`

This description is concise and correct, however, this difference may not be obvious for people when first starting off so here’s another way of thinking about the distinction between these two methods and some examples of how they can be used.

- **append** adds the entire object (can be a number, string, list, range, set, dictionary, tuple, etc.) to a single element at the end of the list.

- **extend** iterates through the entire collection, adding each of the elements in the collection to its own index at the end of a list—the order follows the iteration.

The difference between these two methods can be clearly highlighted by the following toy examples. *Note that in each example **append** adds the entire object, whereas **extend** iterates through each item in the object.*

### 5.1 two lists

```python
lst1 = list('acgt')
lst2 = list('tgca')
lst1.append(lst2)
    ['a', 'c', 'g', 't', ['t', 'g', 'c', 'a']]
len(lst1)
    5
lst1 = list('acgt')
lst1.extend(lst2)  # iterate through each element in lst2
    ['a', 'c', 'g', 't', 't', 'g', 'c', 'a']
len(lst1)
    8
```

### 5.2 list and a string

```python
lst1 = list('acgt')
dnaStr = 'tgcaactg'
lst1.append(dnaStr)
    ['a', 'c', 'g', 't', 'tgcaacgt']
len(lst1)
    5
lst1 = list('acgt')
lst1.extend(dnaStr)  # iterate through each character in dnaStr
    ['a', 'c', 'g', 't', 't', 'g', 'c', 'a', 'a', 'c', 'g', 't']
len(lst1)
    12
```

### 5.3 list and a dictionary

```python
lst1 = list('acgt')
d = {'ATG': 'Met', 'TAG': '---'}
lst1.append(d)
    ['a', 'c', 'g', 't', {'ATG': 'Met', 'TAG': '---'}]
len(lst1)
    5
lst1 = list('acgt')
lst1.extend(d)  # iterate through each key in d
    ['a', 'c', 'g', 't', 'ATG', 'TAG']
len(lst1)
    6
```
6 if...if versus if...elif

The question came up as to what is the difference between using a series of if statements (i.e. if...if...) rather than the if...elif... construct. While these two programming constructs may seem functionally equivalent at first glance, there is an important distinction between them that makes it worth discussing.

First, let’s return to the definition of a conditional statement. An if statement checks an expression and then executes the indentation block below if the expression is true. If the expression is false, the indentation block is not executed. Now, if we an elif statement, then we create a dependency between the two test expressions. When the expression for the if statement is true, the code in the indentation block is executed as before, however, once the indentation block is complete, the code will skip over the elif statement and its indentation block is not executed. If the expression for the if statement is false, only then is the elif expression examined and, if the indentation block will only be executed when the elif expression is True.

The cartoon shown in Figure 1 highlights this dependency.

Figure 1: Control flow diagram for an if...elif pair versus an if...if pair of conditional statements. In this diagram, there are 4 possible paths from b → b', whereas, there are only 3 possible paths from a → a'. In this schematic, the squares indicate junctions at the code will always execute.

Figure 1 raises the question, when would you pick one construct over the other? The answer to this question depends on what you are testing for. If you have a situation where you are testing whether a string matches a certain type of sequence (e.g. DNA versus RNA), then this is a dependent test in that sequences are mutually exclusive (i.e. if it’s DNA, it’s not RNA) so there’s no point in testing for the alternative if the first test turns out to be true. If on the other hand, you have a situation where the two tests are independent, e.g. a DNA sequence has the recognition sites for two separate restriction enzymes, then you don’t care about the result of the first test as you will still want to perform the second test.

7 mutable versus immutable objects (from quiz 2)

Question 8 from Quiz 2 was really about how methods act different on an object depending on whether it is mutable (e.g. a list) or immutable (e.g. a string). Since a number of people seemed confused by this question on the quiz, I thought it would be a good idea to pass out a snippet of Python code that performs the operations from question 8 and encourage students to study this code as you will see something like this again.
8 while versus for loops

According to Edward, a number of students were confused about the differences between while and for loops so he has graciously written up this section to clarify these ideas and show where they can be used.

The main difference between while and for loops is the number of times they will run. A for loop has a fixed number of iterations based on the number of elements in a collection. This looping scheme differs from a number of other languages where the for loop has beginning, step and ending conditions. Instead, the for loop in Python is given an “iterable” object—such as a set, string, range, tuple, list or dictionary—and iterates over that sequence. This concept of iteration makes for loop ideal when you want to examine each character of a string, each element of a list, each key in a dictionary, each element of a tuple, etc. The format of a for loop looks like:

```
for values in iterable:
    ...do something
```

An example:
```
dna = “acgtacgt”
for base in dna:
    if base == ‘a’:
        print “found %s” % base
```

This will print “found a” 2 times. If we go through this iteration step by step, the value of base at each step looks like the following:

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>t</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>t</td>
</tr>
<tr>
<td>print</td>
<td>“found a”</td>
<td></td>
<td>“found a”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you have an iterable that has more than one item in each element (e.g. a tuple of tuples), then you will need to have more than one value to unpack from the iterable. For example,
```
coordinates = ((2, 3), (8, 2), (5, 10)) # A tuple of tuples reflecting pairs of points
for (x, y) in coordinates:
    print “x = %d, y = %d” % (x, y)
```

Will print:
```
x = 2, y = 3
x = 8, y = 2
x = 5, y = 10
```

A while loop, on the other hand, runs until a condition is met: this means the loop does not have a fixed number of iterations, unless of course you make a condition that relies on a counter. That said, any while loop that stops when a counter reaches a certain value can be better done with the for loop with the range iterable such as for i in range(0, x):
where x is the number of times you want your loop to run.

Back to while loops, the condition for whether to continue the loop or not must return a Boolean (i.e. True or False) value (see pages 7-9 in Model about Logical Operators and comparisons). Evaluation statements, like x > y or setA.issubset(setB) or string.isspace() all return Boolean values, which you can check with the Python interpreter:

```
>>> 6 > 7
False
>>> {5, 6}.issubset({5, 6, 7, 8})
True
>>> ““.isspace()
True
```
To implement a while loop in Python, the expression that you use should evaluate to True while you want the loop to run, and False when you want it to stop. Think carefully about this when you write your loop, or you may end up with an infinite loop. You may need to combine different conditions using the Boolean operators and, or, or not to get a while loop that does what you actually want:

# Let's create a while loop that only stops when the user has entered valid input.
# We need a value to start with before we can evaluate it in the while expression,
# which is the initial condition.
dnaseq = raw_input("Please enter a DNA sequence: ")

# Now, compare this value to the set of DNA characters
while not set(dnaseq).issubset({"A", "T", "C", "G")}):
    # If the entered sequence contains characters that are not in the set,
    # then keep asking until the sequence is in the set of DNA characters.
    dnaseq = raw_input("Input is not a DNA sequence, please enter a DNA sequence: ")

One problem with the above loop is that it will run on forever if the user never inputs a DNA sequence. This type of loop is perfectly valid and preferred in some cases. Suppose, however, that we wanted to set a maximum number of times that someone could enter a sequence. In this situation, we would create the following type of while loop.

# Make a counter for the number of times it has run
num_loops = 0
max_tries = 3 # this will limit the number of tries to 3 inside the loop
dnaseq = raw_input("Please enter a DNA sequence: ")
while not set(dnaseq).issubset({"A", "T", "C", "G")} and num_loops < max_tries:
    dnaseq = raw_input("Input is not a DNA sequence, please enter a DNA sequence: ")
    # Now we need to increment our counter so that it's counting each loop
    num_loops += 1

9 dictionaries

I wanted to follow up on the dictionary discussion from lecture and clarify some of the details of the method get(). For this, let’s assume that we have created an empty dictionary d = {}. Remember that you can read the value of a key from a dictionary item with the syntax d[k], where d is the name of the dictionary and k is the key. You can also write values to a key (technically associating a value to a key) with the assignment d[k] = v, where v is any data type you like (e.g. string, number, Boolean, list, tuple, dictionary, etc.).

Let's suppose that we want to create a dictionary that will 1-letter amino acid letters as the keys and we're going to use this dictionary to store the counts of how many of each type of amino acids are in a string. Why would we use a dictionary for this rather than a list? Simply because it's a clean and convenient way to represent the data, both for putting data into the dictionary and for reading data out. Trying to use a list would actually be a headache to keep track of the indices (a dictionary does this for us) and we would have to use two lists—one for the amino acids and the other for its corresponding count—or create some scheme where a particular index represents an amino acid count or use a list of tuples, or something else much more complicated. The bottom line is that dictionaries were designed for handling this type of data structure.

Remember that a string is an iterable so we can use a for loop to go through each letter and add it as a key to the dictionary, as well as, adding a count for the fact we found a particular amino acid. How would this work? Two solutions are listed below and in the example code.
# create a dictionary where the keys are the 1-letter amino acids and the
# values are the number of times that amino acid appears in a string. use a for
# loop to go through the amino acid string. remember to create an empty
# dictionary before starting the loop.
# first: create a dictionary with keys for every amino acid, where the
# initial values are 0.

# init option 1: create empty dictionary (need to start with something)
d = {}

# initialize with dictionary with setdefault() method, which will first check to
# see if the key already exists in the dictionary. if the key exists, it will not
# modify the dictionary, otherwise, it will create a key with the default value,
# which in this case is zero.
natAA = 'ACDEFGHIKLMNPQRSTVWY'
for aa in natAA:
    d.setdefault(aa, 0)

# init option 2: create empty dictionary (need to start with something)
d = {}

# initialize with dictionary with explicit assignment, which will create a key and
# assign the value to zero whether or not that key already exists.
natAA = 'ACDEFGHIKLMNPQRSTVWY'
for aa in natAA:
    d[aa] = 0

# add count to dictionary option 1:
# loop through a string of amino acids and add a count to the key. in this option
# we assume that the key exists and we simply add 1 to the current value and
# assign that addition to the key.
aaSeq = 'VLSPADKTNVKAAW'
for aa in aaSeq:
    d[aa] = d[aa] + 1

# add count to dictionary option 2:
# loop through a string of amino acids and add a count to the key. in this option
# we use the get() method, which doesn’t assume the key exists. in fact, if the
# key doesn’t exist, it will create that key and assign a value to it, which in
# this case is 0. then we simply add 1 to the current value and assign that
# addition to the key. this option is more tolerant of whether or not a key is
# already in the dictionary and the preferred method.
aaSeq = 'VLSPADKTNVKAAW'
for aa in aaSeq:
    d[aa] = d.get(aa, 0) + 1

Remember that dictionaries are iterables so we can easily read out the keys, the values, or both using
the dictionary methods keys(), values(), items() (see page 72 in Model).