**Introduction to Bioinformatics – AS 250.265**
**Lecture 2 Supplement**

Since we didn't get to cover all of the Python basics I would have liked during class, this document may be helpful if you find yourself more confused than ever about what a computer program is. In this document, we'll go through some basics about evaluation in the Python language and then talk about what we missed on conditional expressions and looping on Friday. Of course, programming isn't an easy thing to pick up quickly, and if necessary I'll adjust to reflect this.

## 1. Evaluation in Python

Recall from class that a computer program is simply a list of instructions for a computer to perform. Whether you're using a word processor, web browser, or video game, at some level the computer is just following directions: copy the report you're writing to the printer, open up the CNN web page, or blast that alien scum. Because of copyrights and intellectual property concerns, often you're not allowed to see the actual programming that goes into the applications you use. That's because most programming languages are *compiled*. The programmer writes instructions, and then another program translates those instructions to something that the machine can read but that you would find incomprehensible.

Python is different, because it is an *interpreted* language. The programs you write will be read directly by the Python interpreter, and then it will perform the translation to machine instructions in real time. This is slower than compiling a program, but it has the benefit of allowing more direct interaction with the code. In fact, we can actually run Python directly and type our programs in line by line. The following is a copy of my terminal window when I simply type in "`python`" at the command prompt. The text I type is in red—everything else is what the terminal displays.

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello class!'
Hello class!
>>> print 2+2
4
>>> a = 6
>>> b = 5
>>> print a + b
11
>>>
[nfitzkee@frodo nfitzkee]$
```

As you can see, the same statements that are allowable in a program file are allowed in the Python "shell" itself. You simply type in the command you want Python to run after the ">>>" and it will oblige. This behavior highlights the fact that Python is only following one command at a time as it goes through your programs. The only reason we use programs files and not the interpreter (by editing them as a single file in Xemacs) is because entering our

instructions one line at a time is rather cumbersome. To exit the Python shell, I typed <CTRL-D>; that is, I held down the control key and pressed D.

What happens when we give Python an invalid command? If someone came up to you and said "Bellows orange brick stuff exam said," you would probably say (politely), "What?" Similarly, if you type something into your program or the interpreter that Python cannot understand, it will display an error. Many times, it will try to diagnose what was wrong with your statements. Below is another transcript that shows some Python error messages.

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> moo shoo pork
  File "<stdin>", line 1
    moo shoo pork
            ^
SyntaxError: invalid syntax
>>>
>>> print b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'b' is not defined
>>> 5. / 0.
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: float division
>>> a = 'Hello class
  File "<stdin>", line 1
    a = 'Hello class
                    ^
SyntaxError: EOL while scanning single-quoted string
>>>
[nfitzkee@frodo nfitzkee]$
```

In the first example, when you simply type gobbledygook into the interpreter it complains of a "syntax error." This is generally what is displayed as a last resort when Python can't understand what you're telling it to do. The second example displays what happens when you try to reference an undefined variable (b). The third example is what happens when you try to perform an invalid mathematical operation. In this case, it's division by zero, but a similar error would occur if you were to take the logarithm of a negative number. The final example shows what would happen if you forgot to terminate a string with a closing quotation mark. Python doesn't always know when you want to stop a string, so if you forget to close a string, it may think that the rest of the program is part of your DNA sequence!

What does Python actually do when it attempts to follow your directions? When you started doing math, you had to be very explicit in how you worked through problems. However, as you got better, you were able to perform multiple steps at a time. Computers cannot do this. Consider the following expression:

```
sin( (3) / (1+3) )
```

You can look at this expression and instantly know to type sin(0.75) in to your calculator and get the right answer. A computer, however, has to go through the following simplification process, one step at a time.

```
sin( (3.0) / (1.0+3.0) )
sin( (3.0) / (4.0) )
sin( 0.75 )
0.68163876002333423
```

When you're in the Python interpreter, it has to do all this before it can give you the final answer, even though you don't see it (and it takes next to no additional time). Instead, this is what you see:

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import sin
>>> sin( (3.0) / (1.0 + 3.0) )
0.68163876002333423
>>>
```

Now, suppose you didn't type in the extra parentheses? Let's take a look to see what Python would do then.

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import sin
>>> sin( 3.0 / 1.0 + 3.0)
-0.27941549819892586
>>>
```

A little work would show that Python in this case is actually evaluating sin(6) and not sin(0.75). This shows that Python, like you, has a pre-determined order of operations that it must follow. In this case, it performs division before addition, just like it's supposed to. If you're in doubt as to what Python will do with a set of operations, I'd recommend using the parenthesis.

**Real Numbers versus Integers**

In class, I alluded to the need to specify the decimal place for when you perform operations like division. I said that the reason for this is beyond the scope of our course, but the curious are welcome to stop by office hours for an explanation. Since this is something you may need to be aware of, let's take a look at some examples.

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 3.0 / 4.0
0.75
>>> print 3. / 4.
0.75
>>> print 4. / 3.
1.33333333333
>>> print 3 / 4
0
>>> print 4 / 3
1
>>> print 4. / 3
1.33333333333
>>>
```

In the first three cases, Python seems to get the right answer. Specifing ".0" or even simply a "." tells python to use "floating point" mathematical operations. Leaving off the decimal point tells Python to use "integer" or whole number mathematical operations. When this is done, the division is performed, but all the remaining values behind the decimal place are ignored. 0.75 becomes simply 0, and 1.333 becomes 1. To be safe, simply use the decimal places when you want to print out a mathematical result. In the final example, we can see that Python assumes floating point operations when one of the two numbers are given with a decimal place, but it is generally bad form to do this.

**Conditional Expressions**

Now that we've learned about mathematical evaluation, we can look at some Boolean logic. A Boolean is simply a value that can either be true (1) or false (0). Consider the following five questions with their yes/no answers:

1. Is an apple an orange? No.
2. Is the earth round? Yes.
3. Does the protein have an amino acid sequence? Yes.
4. Is the one-letter code for Alanine Q? No.
5. Is five greater than four? Yes.

Above, we answered the following questions either yes or no based on the truth content of the statements. In Python, conditional expressions are simply stated, and then the interpreter determines, not yes or no, but true or false. The five statements above, stated in Boolean terms, would be:

1. An apple is an orange. False.
2. The earth is round. True.
3. The protein has an amino acid sequence. True.
4. The one letter code for Alanine is Q. False

5. Five is greater than four. True.

Because writing sentences like those above would be difficult for the computer to understand, Python has several standard *conditional operators* that can be used to making such comparisons. These operators are > (greater than), < (less than), == (equal to, note that this is two equal signs), <= (less than or equal to), and >= (greater than or equal to). In additon, Python understands the keywords and, or, and not. Now we can look at some actual Python code.

```
>>> print True
True
>>> print False
False
>>> 1 < 2
True
>>> 'apple' == 'orange'
False
>>> 1 >= 4
False
>>> 1 > 4 or 5 < 6
True
>>> not (1 > 4)
True
>>>
```

As you can see, Python recognizes `True` and `False` as built in keywords, and then it evaluates the conditional expressions to either True or False. Since we are in interactive mode, we don't need to use "print" all the time—Python will show us what the expression we typed evaluated to.

There is one other handy Boolean operator worth knowing in Python which has to do with lists. Suppose you would like a quick and easy way to determine whether a particular element is a member of a particular list. This would be done using the `in` keyword, illustrated below.

```
>>> 'a' in ['b', 'c', 'dog', 'a', 'cat']
True
>>> b = [1, 3, 5, 7, 9, 11]
>>> 4 in b
False
>>>
```

**Summary**

With all this, we've pretty much covered the basics of all the expressions you will need for the course. All of the things we've talked about so far evaluate to a value, either a numeric value, a true/false value, or another type of value (string, etc.) Thus, it would be fair to assign any real value to a variable, as we discussed in class.

```
>>> a = 1 > 2 and not 1 < 4
>>> b = 5*6
>>> print a
```

```
False
>>> print b
30
>>>
```

Once you've mastered the expressions, and their evaluation, there are only two other topics that you'll need to know before you can start writing very powerful programs.

## 2. Conditional Program Control

In the last section, we learned about statements that can evaluate to either true or false. It would be nice if Python could change what it does based the values of these expressions. Consider the following instructions you may give to your roommate as he or she heads out the door to Super Fresh.

Grocery List
1. Skim milk
2. Bagels
3. If it's on sale, get some Breyers Vanilla Ice Cream, otherwise get the store brand.
4. Cereal
5. If you still have money, left over, pick up some pretzels.

Remember that a computer needs to be told everything explicitly, so this list probably wouldn't work for a simple computer program. We need a simple and concise way to tell a computer to do something based on the value of a conditional expression. Enter the `if` statement. This statement along with `else` and `elif` (short for "else if") allows you to control what your program does based on the values of other expressions.

Conditional statements have a unique syntax in Python. Below is an example of how to use the `if` … `elif` … `else` construct. Note that this is what you would see in Xemacs, not at the interpreter. We'll get to that later.

```
a = 4
b = 6

if a == 3:
      print 'The variable a is three'
      (other code can be inserted here)

elif a == 4 and b == 5:
      print 'The variable a is four and b is five'
      (again, other code can be placed here)

elif a == 4 and b == 6:
      print 'The variable a is four and b is six'
      (more code, if necessary)

else:
      print 'Neither of the conditions were met'
      (other statements)
```

The general form is `if`, then some expression, followed by a colon (`:`). Then the code that you want to execute based on that condition follows, indented by a tab. The expression can be any expression we learned about in the previous section, or it can be a variable containing a result of one of those expressions. There is quite a bit of variability in formatting if statements. You can include one `if`, with no `else` or `elif`. In this case, the indented code would only execute based on the expression contained in the `if` statement. Consider the following example, which is typed directly into the Python interpreter.

```
[nfitzkee@frodo nfitzkee]$ python
Python 2.4.2 (#2, Nov 23 2005, 16:43:13)
[GCC 4.0.0 (Apple Computer, Inc. build 5026)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> seq = "AUGGCATTACCC"
>>> if seq[0] == 'A' and seq[1] == 'U' and seq[2] == 'G':
...     print 'Sequence begins with a start codon (AUG)'
...
Sequence begins with a start codon (AUG)
>>>
```

Note that Python doesn't care if I only have an if statement, with no other else or elif statements. It will happily process the indented code if the statement is true (it is here), and if not, it will simply continue processing input as before, ignoring the indented code. Also note that, in the interpreter, when it sees an if statement, it changes the prompt from ">>>" to "…" letting you know that it's still expecting you to continue the if statement. After an if statement, there can be zero or more elif's which are executed in mutual exclusion, and zero or one else blocks, which are executed if none of the other statements evaluate to true.

In your own programs, when you're ready to end the if statement *block*, simply start typing at the previous indent level. The following program shows two if statements nested within one another. Python can tell you are ending the if statement by the level of indentation used.

```
codons = ['AUG', 'GCA', 'UAU', 'CCA', 'UAG', 'CCU']

if codons[0] == 'AUG':
      print 'Sequence begins with a start codon (AUG)'

      if ('UAG' in codons) or ('UAA' in codons) or ('UGA' in codons):
            print 'Sequence contains a stop codon'

      print 'Sequence is', (3*len(codons)), 'bases'

print 'Sequence has', len(codons), codons
```

In the above example, the number of bases will only appear if the sequence begins with a start codon. Similarly, the notification about the stop codon will only appear if one of the three stop codons are found in the list of codons.

## 3. While loops

The final topic that we will cover here are loops. Fortunately, loops and if statements are very similar. Both require the evaluation of a true/false expression, and in Python the extents of both are offset by indentation. They can also both be nested within other loops and if statements. Generally, loops are used to process the elements of a list individually. Since the same commands are processed over and over again, loops save time and space by allowing a single set of commands to be executed many times while you only must type them once. The example of a loop we are using here is a while loop. The syntax of a while loop is given below:

```
while {expression}:
      {command 1}
      {command 2}
      ...

{rest of code}
```

The first thing Python does when it encounters a while loop is evaluate the {expression}. If the expression is true, the commands within the loop block of code are evaluated. Once Python reaches the end of the block of code, it evaluates the expression again. If the expression is false, the loop ends, and Python resumes execution at the end of the loop. If the expression is still true, Python enters the loop again.

Now that we understand how a while loop works, we can examine the following example, which performs a more thorough analysis of our RNA transcript. The following example includes many of the concepts presented here as well, including if statements, etc., and you can copy it to a program file in Xemacs and run it yourself.

```
rna = ['aug', 'cca', 'cua', 'aac', 'uaa', 'uuu', 'cgc']
counter = 0

while counter < len(rna):
      if counter == 0 and rna[counter] == 'aug':
            print 'Sequence starts with a start codon'

      if rna[counter] == 'uag':
            print 'Sequence has stop codon (uag) at codon #', counter
      elif rna[counter] == 'uaa':
            print 'Sequence has stop codon (uaa) at codon #', counter
      elif rna[counter] == 'uga':
            print 'Sequence has stop codon (uga) at codon #', counter
      else:
            print 'Codon', counter, 'is not a stop codon'

      # Perform other analysis here, maybe translating the codon

      counter = counter + 1
```

The code above goes through the RNA sequence, one codon at a time, and performs a simple analysis to see whether a start codon appears at the being, and whether the sequence contains any stop codons.

What would happen if the final statement (which increments counter) were left out? Clearly, the condition for the while loop would never be false, since the size of counter would always remain at zero and never increase to the size of the list. In this case, Python has encountered what is called an *infinite loop*. And it will appear to hang in your window. In some situations, infinite loops can be difficult to spot, especially if you expect the program to take a long time anyway. Indeed, there is no algorithm that will tell you whether in general a program contains an infinite loop. Fortunately, at this stage, all of your programs should terminate very quickly, and if it hangs for more than a few seconds you're likely stuck in a loop. To terminate an infinite loop, simply press <CTRL-C>, which will tell Python to stop processing its current code. Then, you can examine where your loop may be occurring.

Hopefully, this tutorial has provided you with what you'll need to interpret and begin writing your own Python programs. I encourage you to practice and immerse yourself in learning the language, and I believe you'll find the learning curve quickly becomes reasonable. Ultimately, the benefits of knowing a programming language—both in terms of marketability for future careers as well as the discipline in thinking that it requires—will benefit you tremendously more than the effort it takes to learn the language initially. If you are struggling, though, please come in for office hours.