Lecture 5: Conditionals and loops

Often we don't know ahead of time what sort of data will be entered into the program. For example, if the user is allowed to enter DNA sequence into the program, we don't know ahead of time whether they will enter valid information. In order to run additional analysis on the DNA sequence, we first want to know if the DNA sequence is valid. If it is valid, then we want to do one thing. If it isn't valid we want to do something else. The way we accomplish this in Python is through if statements.

If statements

The basic structure of an if statement is:

```
if conditional:
        code
```

The if statement tells the interpreter to evaluate the conditional. A colon (:) MUST be present at the end of the conditional followed by indented code (use a tab to indent). If the conditional is True, then the indented code (represented by code) is run. If the conditional is false, then we continue on in the program without running code. For example:

```
>>> x = 0
>>> y = 0
>>> x
0
>>> y
0
>>> if True:
...     x = 1
...     y = 1
...
>>> x
1
>>> y
1
>>> if False:
...     x = 0
...     y = 0
...
>>> x
1
>>> y
1
```

This code shows that the indented code only ran when the conditional was True and not when the conditional was False. True and False are built-in booleans representing True or False. In general, you will not know ahead of time whether the conditional should be True or False (which is what makes the if statement useful). In most cases, you use comparision operators to determine on the fly if a conditional is true. The following comparison operators return either True or False:

== is equal to

!= is not equal

< is less than

> is greater than

<= is less than or equal to

>= is greater than or equal to

You can use these comparison operators outside the context of an if statement:

```
>>> 1==1
True
>>> 1==1+1
False
>>> 1!=1
False
>>> 1!=1+1
True
```

But these are most often used as part of if statements:

```
>>> x = 1
>>> if x==1:
...    print("x is 1!")
...
x is 1!
```

You can also append multiple if statements in a row using the elif statement. elif is equivielent to

```
>>> if x==2:
...    print("x equals 2!")
... elif x!=1:
       print(“x doesn’t equal 1!”)
... else:
...    print("x is 1!")
...
x is 1!
```

This code demonstrates how do use an if statement to ascertain whether a variable fits some condition. It also shows the elif and else statement. There is no conditional for the else statement, but rather executes if the preceeding if does not. elif is Python’s version of else if.

Combining Booleans with “and”/”or”

Sometimes you will want to combine to logical statements together. For example, you might be interested in whether a number is between 0 and 100. Instead of combining two if statements together, you can combine both conditionals using the and statement:

```
>>> x = 1
>>> if x > 0 and x < 100:
...    print("YES!")
...
YES!
```

The and and or statements combines two Booleans and returns True or False based upon the following truth table:

Input 1	Input 2	and	or
True	True	True	True
False	True	False	True
True	False	False	True
False	False	False	False

In statements

In statements are also really useful for determining if an element exists within a container:

```
>>> good_letters = 'acgtACGT'
>>> 'a' in good_letters
True
>>> 'b' in good_letters
False
>>> list_letters = ['a','c','g','t','A','C','G','T']
>>> 'a' in list_letters
True
>>> 'b' in list_letters
False
```

One thing that is a bit confusing is what the conditional actually has to equal in order for the indented code to execute. Let’s try a few things out. Put this in your python file and execute it:

```
if True:
        print("True")
if False:
        print("False")
if 1:
        print("1")
if 0:
        print("0")
if -1:
        print("-1")
if "Python":
        print("Python")
if ":
        print("Empty string")
if None:
        print("None")
```

You should see something like this:

```
[pmcgrath7@bioebb301301:~]$python3 first_script.py
True
1
-1
Python
```

This shows that the exact things that trigger the if statement are complicated. While True triggers it, any non-zero number does as well as any non-empty string. Best practice: Always make sure there is a conditional in your if statement that returns True or False

Loops

Sometimes you want to run the same code over and over for a known or unknown number of times. For example, let's say the user input a string that's supposed to be DNA sequence. The possible values for each character are A, C, G, T, a, c, g, or t. If you the user enters sequence, you want to make sure all the values that they enter are one of these 8 letters. The most straightforward way to accomplish this is by analyzing each letter that in the string that is created. You can do this using a loop. There are two main types of loops in Python: for and while.

Let's start with a string:

```
>>> DNA = 'ACGCGGGXCCGT'
```

To loop through this string we can do

```
>>> for letter in DNA:
...    print (letter)
...
A
C
G
C
G
G
G
X
C
C
G
T
```

So how does this work? There are two new keywords, for and in. for tells Python to loop through some elements. What are the elements? In this case there are the letters of the string. The in statement describes how to find each element and works like x in y where an element in y is stored in x. The semicolon tells Python to expect an indented block of code that will be run on every letter. Each time you go through the loop you will run the block of indented code.

You can now do more fancy things like:

```
>>> for letter in DNA:
...    if letter.upper()!='A' and letter.upper()!='C' and 
letter.upper()!='G' and letter.upper()!='T':
...       print(letter + " isn't a valid nucleotide!")
...
X isn't a valid nucleotide!
```

You can also do this on lists. For example:

```
>>> phrase = 'This is a bunch of different words'
>>> words = phrase.split()
>>> words
['This', 'is', 'a', 'bunch', 'of', 'different', 'words']
>>> for word in words:
...    print(word)
...
This
is
a
bunch
of
```

```
different
words
```

In both cases, the for loop ‘knows’ how to loop through the elements of the string or the list. How does it do this? It relies on the __iter__() method of the string and list classes, which returns a class of str_iterator or list_iterator.

Iterators are really important in Python, so it’s worth covering these in a little bit more depth. An iterator needs to do four things:

1. Identify the first element of the array
2. Return the value of the element
3. ‘Know’ how to find the next element of the array
4. ‘Know’ once it hits the end of the array

Let’s see how we can do this for a string:

```
>>> iterator = DNA.__iter__()
>>> type(iterator)
<class 'str_iterator'>
>>> iterator = iter(DNA)
>>> type(iterator)
<class 'str_iterator'>
>>> dir(iterator)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__length_hint__', '__lt__', '__ne__', '__new__',
'__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setstate__', '__sizeof__', '__str__', '__subclasshook__']
>>> #Use the __next__ method or next() built-in function to move through
the string
...
>>> iterator.__next__()
'A'
>>> iterator.__next__()
'C'
>>> iterator.__next__()
'G'
>>> next(iterator)
'C'
>>> next(iterator)
'G'
>>> next(iterator)
'G'
>>> next(iterator)
'G'
>>> iterator.__next__()
'X'
>>> iterator.__next__()
'C'
>>> iterator.__next__()
'C'
>>> next(iterator)
'G'
>>> next(iterator)
```

```
'T'
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Ok, what happened. First we created an string iterator instance from DNA using either the built-in function iter() or the method __iter__(). This class contains a method called __next__(). By calling this method (or the built-in function next(), we could move through the characters of DNA until we reached the end of the string (which resulted in a StopIteration error.

This is essentially what the for loop does setting the value of letter to what is returned by the next function (i.e. letter = next(iterator). The for loop then executes whatever code on letter that you've told it to.

You also might want to run a loop a set number of times. For example, let's say you wanted to create a list containing the first 10 powers of 2. You can use the range function for that, which can be iterated through by the for loop. For example:

```
>>> for i in range(0,10):
...    print (2**i)
...
1
2
4
8
16
32
64
128
256
512
```

Finally the while loop can be very useful:

while True:

   code

```
>>> x = 0
>>> while x < 10:
...    print(x)
...    x += 1
...
0
1
2
3
4
5
6
7
8
9
```

You can also use a break statement to exit a loop, even if the conditional hasn't changed to false:

```
>>> while x < 10:
```

```
...     print(x)
...     x += 1
...     if x > 8:
...        break
...
0
1
2
3
4
5
6
7
8
```