Lecture 2

if Statements: The `if` statement allows you to only conditionally execute some code block, conditioned on some expression evaluating to `True`.

```
if BOOL_EXPR:
    CODE_BLOCK
elif BOOL_EXPR:
    CODE_BLOCK
...
elif BOOL_EXPR:
    CODE_BLOCK
else:
    CODE_BLOCK
```

In the above code, exactly one of the code blocks is executed, corresponding to the first `BOOL_EXPR` which evaluates to `True` (or the final code block corresponding to the `else` in the case that none of the `BOOL_EXPR` evaluates to `True`). The `elif` and `else` statements are optional.

Example:

```
def printSign(n):
    if n < 0:
        print 'Negative'
    elif n > 0:
        print 'Positive'
```

Now, if we were to execute `printSign`(-1), 'Negative' would be printed, and similarly `printSign`(1) would print 'Positive'. Calling `printSign`(0) would result in nothing being printed.

for Statements: The `for` statement allows you to iterate over data in Python (for example, iterating over items in a `list`, or characters in `str`).

```
for var in v:
   CODE_BLOCK
```

The expression `v` above should evaluate to something iterable.

Example:

```
fruits = ['orange', 'pineapple', 'banana', 'mango']
pluralFruits = []
for x in fruits:
    plural = x + 's'
    pluralFruits += [plural]
```

The above code segment would cause `favoriteFruits` to equal ['oranges', 'pineapples', 'bananas', 'mangos'].

while Statements: The `while` statement allows you to repeatedly execute a code block as long as some `bool` expression evaluates to `True`.

```
while BOOL_EXPR:
   CODE_BLOCK
```

Example:

```
x = []
y = 0
while y < 10:
    x += [y]
    y += 1
```

The above code segment would cause `x` to equal [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

break and continue: Sometimes you might want to stop iterating in a `for` or `while` early, or just skip some particular iteration. The `break` and `continue` statements are useful for this. `break` exits the loop early, and `continue` moves back to the beginning of the loop.

Example: Both of these code examples print only the odd numbers between 0 and 5.

```
# Example with for loop
myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for x in myList:
    if x > 5:
        break
    elif x%2 == 0:
        continue
    else:
        print x

# Example with while loop
x = 0
while True:
    if x > 5:
        break
    elif x%2 == 0:
        x += 1
        continue
    else:
        print x
    x += 1
```

Other useful functions: It will be helpful for today's lab to know the following functions.

- `len`(x) returns the length of an iterable data type (such as a `str` or `list`) as an `int`. For example, `len`('abc') and `len`(['a', 'b', 'c']) both evaluate to 3. `len`(['a', ['b', 'c', 'd']]) evaluates to 2.
- `range`(x) returns a list of `int`s from 0 to $x - 1$. For example, `range`(5) returns [0, 1, 2, 3, 4]. You can also give `range` a starting value (`range`(2, 5) returns [2, 3, 4]) and a "skip-by" value (`range`(0, 10, 2) returns [0, 2, 4, 6, 8]).
- `xrange`(x) is similar to `range`(x), except that it does not actually return a list, but rather returns an object which can be iterated over and has the same values as if `range` had been called. We have not spoken about objects yet, so do not worry too much about what that means, but the main point is that `xrange` only generates the next value as you need it without ever explicitly storing the entire `list` in memory, whereas `range` would explicitly store the list. Thus, `xrange` can be helpful if you know your code will `break` early in a long sequence. Consider the following examples:

```
# Example 1
for x in range(1000000000):
    if x == 6:
        break
    print x

# Example 2
for x in xrange(1000000000):
    if x == 6:
        break
    print x
```

In both cases the numbers $0, 1, 2, 3, 4, 5$ are printed, but the first code example is almost a billion times slower because `range` will actually generate a list of size one billion at the beginning of the loop, whereas `xrange` just generates the next number as it is needed.