**Advantages of Generators over lists**

Basic Rundown

Generators functions allow you to declare a function that behaves like an iterator, that is it can be used in a for loop. Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets.

Iterating over the *generator expression* or the *list comprehension* will do the same thing. However, the *list comprehension* will create the entire list in memory first while the *generator expression* will create the items on the fly, so you are able to use it for very large (and also infinite!) sequences.

Elongated rundown

The important point is that the list comprehension creates a new list. The generator creates an iterable object that will "filter" the source material on-the-fly as you consume the bits.

Imagine you have a 2.5TB log file called "**hugefile.txt**", and you want the content and length for all the lines that start with the word "ENTRY".

Ex. start out by writing a list comprehension:

logfile = open("hugefile.txt","r")

entry\_lines = [(line,len(line)) for line in logfile if line.startswith("ENTRY")]

This slurps up the whole file, processes each line, and stores the matching lines in your array. This array could therefore contain up to 2.5TB of content. That's alot of RAM, and probably not practical for your purposes.

So instead we can use a generator to apply a "filter" to our content. No data is actually read until we start iterating over the result.

# logfile = open("hugefile.txt","r")

entry\_lines = ((line,len(line)) for line in logfile if line.startswith("ENTRY"))

Not even a single line has been read from our file yet. In fact, say we want to filter our result even further:

long\_entries = ((line,length) for (line,length) in entry\_lines if length > 80)

Still nothing has been read, but we've specified now two generators that will act on our data as you wish.

Ex. Write out our filtered lines to another file:

outfile = open("filtered.txt","a")

for entry,length in long\_entries:

outfile.write(entry)

*Next* read the input file. As the **for** loop continues to request additional lines, the **long\_entries**generator demands lines from the **entry\_lines** generator, returning only those whose length is greater than 80 characters. And in turn, the **entry\_lines**generator requests lines (filtered as indicated) from the **logfile**iterator, which in turn reads the file.

So instead of "pushing" data to your output function in the form of a fully-populated list, you're giving the output function a way to "pull" data only when its needed!! This case is more efficient, but not so much as flexible. Generators are exhbitied as a one way, one pass mentality; the data from the log file we've read gets immediately *discarded*, so we one go back to a previous line. On the other hand, you don't have to worry about keeping data around once we're done with it!