Python Strings, Lists, and Tuples

> "Good programming is good writing." - John Shore

There's some overlap among Python's strings, lists, and tuples. In a way, you could think of strings as lists of characters. Many list operations work exactly the same over strings like subscripting to get a particular item. We can ask for the first (or "zeroth") element from a string:

```
>>> name = 'Curly'
>>> name[0]
'C'
```

Or from a list:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> names[0]
'Larry'
```

"Slice" operations let you take a range of items. Notice that we can operate on a string literal (in quotes):

```
>>> names[2:4]
['Curly', 'Shemp']
>>> 'Curly'[2:4]
'rl'
```

Functions like `join` that take lists can also work on strings:

```
>>> ', '.join(names)
'Larry, Moe, Curly, Shemp'
>>> ', '.join(names[0])
'L, a, r, r, y'
```

You can ask if a list contains a certain member, and you can also ask if a string contains a certain character or substring:

```
>>> 'Moe' in names
True
>>> 'r' in 'Larry'
True
>>> 'url' in 'Curly'
True
>>> 'x' in 'Larry'
False
>>> 'Joe' in names
False
```

You can iterate with a `for` loop over both the items in a list or the characters in a word:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> for name in names:
...   print(name)
...
Larry
Moe
Curly
Shemp
>>> for letter in 'Curly':
...   print(letter)
...
C
u
r
l
y
```

Just as in bash, we can create a counter, increment it inside our loop, and print the element number before the element:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> i = 0
>>> for name in names:
...   i += 1
...   print(i, name)
...
1 Larry
2 Moe
3 Curly
4 Shemp
```

Because we so often want this behavior, there is a function called `enumerate` that takes a list/string and returns the index/position along with the item/character:

```
>>> for i, name in enumerate(names):
...   print('{:3} {}'.format(i, name))
...
  0 Larry
  1 Moe
  2 Curly
  3 Shemp
>>> for i, letter in enumerate('Curly'):
...   print('{:3} {}'.format(i, letter))
...
  0 C
  1 u
  2 r
  3 l
```

```
  4 y
```

You can use the `reversed` function on both strings and lists. Try it:

```
>>> reversed('cat')
<reversed object at 0x10bdc8358>
```

Probably you expected to see "tac"? Yeah, me, too. What is a "reversed object"? For now, think of it as a promise to give you the reversed string when you actually need it. We can force it into a list that we can look at by using the `list` function:

```
>>> list(reversed('cat'))
['t', 'a', 'c']
```

OK, closer, but I wanted to see "tac" and not a list containing those letters. We can put them back into a word by calling the `join` *function* of the *string element* that we want to put between the letter (which is an empty string):

```
>>> ''.join(list(reversed('cat')))
'tac'
```

Hmm, quite a bit of work to turn a word around. Still, could be useful, for example in finding CRISPR (clustered regularly interspaced short palindromic repeats) sequences? Here is a simple program to determine if a given string is a palindrome which is a string that is the same forwards and backwards.

```
$ cat -n word_is_palindrome.py
     1    #!/usr/bin/env python3
     2    """Report if the given word is a palindrome"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} STR'.format(os.path.basename(args[0])))
    11        sys.exit(1)
    12
    13    word = args[0]
    14    rev = ''.join(reversed(word))
    15    print('"{}" is{} a palindrome.'.format(word, '' if word.lower() == rev.lower() els
```

As we discussed earlier, `sys.argv` returns exactly what the operating system thinks of as "the program" it's running, namely that the program name is in the first (zeroth) position, and anything else you type on the command line follows. If you run this as `./word_is_palindrome.py foo` then `sys.argv` looks like `['./word_is_palindrome.py', 'foo']`. Because the program name is in

the [0] position, it makes more sense to get the arguments for the program like this (skipping over the 0th element):

```
args = sys.argv[1:]
```

You can say more logical things like:

```
if len(args) == 0:
    print('Usage: blah blah blah')
    sys.exit(1)
```

Note that Python will throw an exception if you try to reference an index position in a list that doesn't exist:

```
>>> 'foo'[0]
'f'
>>> 'foo'[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python will **not** blow up, however, if you take a slice of an array starting or ending at non-existent positions:

```
>>> 'foo'[1:10]
'oo'
>>> 'foo'[5:]
''
```

Which is why it's safe to say `sys.argv[1:]` to slice out everything starting at position 1 even if there is nothing there.

We can expand our palindrome program to one that searches in a file:

```
$ cat -n find_palindromes.py
     1  #!/usr/bin/env python3
     2  """Report if the given word is a palindrome"""
     3
     4  import sys
     5  import os
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  file = args[0]
    14
    15  if not os.path.isfile(file):
    16      print('"{}" is not a file'.format(file))
```

```
    sys.exit(1)

for line in open(file):
    for word in line.lower().split():
        if len(word) > 2:
            rev = ''.join(reversed(word))
            if rev == word:
                print(word)
```

Lines 19-20 read each `line` and then lowercase and `split` (on spaces) into each `word`. You could compress this like so (see "find_palindromes2.py"):

```
for word in open(file).read().lower().split():
```

This will call `read` on the opened file handle to bring the entire file contents into memory, lowercase, and `split` into words. The first way is probably more efficient with memory, but you will likely see files being read in this way. Another common idiom to read all the lines of a file while removing the newlines is:

```
all_lines = open(file).read().splitlines()
```

Tetranucleotide Composition

A common operation in bioinformatics is to determine sequence composition. Here is a program to find the frequencies of the DNA bases (A, C, T, G):

```
$ cat -n dna1.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count_a, count_c, count_g, count_t = 0, 0, 0, 0
    16
    17    for letter in dna:
    18        if letter == 'a' or letter == 'A':
    19            count_a += 1
```

```
    20        elif letter == 'c' or letter == 'C':
    21            count_c += 1
    22        elif letter == 'g' or letter == 'G':
    23            count_g += 1
    24        elif letter == 't' or letter == 'T':
    25            count_t += 1
    26
    27    print(' '.join([str(count_a), str(count_c), str(count_g), str(count_t)]))
$ ./dna1.py AACCTAG
3 2 1 1
```

On line 15, we initiate four variables to count each DNA base. Just as we can use a `for` loop to iterate through a list, we can iterate through each letter in a string on line 17. We need to check for both upper- and lowercase strings to determine which counter to increment. Line 27 points out that the "count_*" variables are numbers that must be converted to strings in order to `print` them.

To save quite a bit of typing, let's force the input sequence to lowercase:

```
$ cat -n dna2.py
     1    #!/usr/bin/env python3
     2    """Tetra-nucleotide counter"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    dna = args[0]
    14
    15    count_a, count_c, count_g, count_t = 0, 0, 0, 0
    16
    17    for letter in dna.lower():
    18        if letter == 'a':
    19            count_a += 1
    20        elif letter == 'c':
    21            count_c += 1
    22        elif letter == 'g':
    23            count_g += 1
    24        elif letter == 't':
    25            count_t += 1
    26
    27    print(' '.join([str(count_a), str(count_c), str(count_g), str(count_t)]))
```

There are better ways than this to count the characters, but we'll save this until we talk about dictionaries.

Lastly, let's use the `format` method to get rid of those pesky `str` calls:

```
$ cat -n dna3.py
     1  #!/usr/bin/env python3
     2  """Tetra-nucleotide counter"""
     3
     4  import sys
     5  import os
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  dna = args[0]
    14
    15  count_a, count_c, count_g, count_t = 0, 0, 0, 0
    16
    17  for letter in dna.lower():
    18      if letter == 'a':
    19          count_a += 1
    20      elif letter == 'c':
    21          count_c += 1
    22      elif letter == 'g':
    23          count_g += 1
    24      elif letter == 't':
    25          count_t += 1
    26
    27  print('{} {} {} {}'.format(count_a, count_c, count_g, count_t))
$ ./dna3.py AACCTAG
3 2 1 1
```

If you're having trouble seeing the differences from `dna2.py` to `dna3.py`, try using `diff`:

```
$ diff dna2.py dna3.py
27c27
< print(' '.join([str(count_a), str(count_c), str(count_g), str(count_t)]))
---
> print('{} {} {} {}'.format(count_a, count_c, count_g, count_t))
```

Run-length Encoding

Along the lines of counting characters in a string, we can write a very simple string compression program that encodes repetitions of characters:

```
$ ./compress.py AAACAATTTTGGGGGAC
A3CA2T4G5AC
$ cat -n compress.py
     1    #!/usr/bin/env python3
     2    """Compress text/DNA by marking repeated letters"""
     3
     4    import os
     5    import sys
     6
     7    args = sys.argv[1:]
     8
     9    if len(args) != 1:
    10        print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
    11        sys.exit(1)
    12
    13    arg = args[0]
    14    text = ''
    15    if os.path.isfile(arg):
    16        text = ''.join(open(arg).read().split())
    17    else:
    18        text = arg.strip()
    19
    20    if len(text) == 0:
    21        print('No usable text')
    22        sys.exit(1)
    23
    24    counts = []
    25    count = 0
    26    prev = None
    27    for letter in text:
    28        if prev is None:
    29            prev = letter
    30            count = 1
    31        elif letter == prev:
    32            count += 1
    33            prev = letter
    34        else:
    35            counts.append((prev, count))
    36            count = 1
    37            prev = letter
    38
```

```
39    # get the last letter after we fell out of the loop
40    counts.append((prev, count))
41
42    for letter, count in counts:
43        print('{}{}'.format(letter, '' if count == 1 else count), end='')
44
45    print('')
```

Line 15 uses the `os.path.isfile` function to determine if the argument is a file; if so, line 16 uses the code from earlier to `split` the entire file into "words" and then `join`s them back together on the empty string. This would concatenate all sequence lines into one long sequence. If the argument is not a file, then we use `rstrip` to get rid of any spaces on the right-hand side.

This program makes use of a `counts` list to keep track of each letter we saw. We add a "tuple" to the list:

```
>>> counts = []
>>> counts.append(('A', 3))
>>> counts
[('A', 3)]
>>> counts.append(('C', 1))
>>> counts
[('A', 3), ('C', 1)]
```

Tuples are similar to lists, but they are immutable:

```
>>> tup = ('white', 'dog')
>>> tup[1]
'dog'
>>> tup[1] = 'cat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You see they are subscripted like strings and lists, but you cannot change a value inside a tuple. Tuples are not limited to pairs:

```
>>> tup = ('white', 'dog', 'bird')
>>> tup[-1]
'bird'
```

tac

We all know and love the venerable `cat` program, but do you know about `tac`? It prints a file in reverse. We can use lists in Python to read a file into list and `reverse` it:

```
$ cat input.txt
first line
second line
third line
fourth line
$ ./tac1.py input.txt
fourth line
third line
second line
first line
```

Here is the code:

```
$ cat -n tac1.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
     8    if len(args) != 1:
     9        print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    10        sys.exit(1)
    11
    12    file = args[0]
    13    if not os.path.isfile(file):
    14        print('"{}" is not a file'.format(file))
    15        sys.exit(1)
    16
    17    lines = []
    18    for line in open(file):
    19        lines.append(line)
    20
    21    lines.reverse()
    22
    23    for line in lines:
    24        print(line, end='')
```

We initialize a new list on line 17, then read through the file line-by-line and call the `append` method to add the line to the end of our list. Then we call `reverse` *function* on the list to mutate the list **IN PLACE**:

```
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
>>> names.reverse()
>>> names
['Shemp', 'Curly', 'Moe', 'Larry']
```

After `reverse` we see that the `names` are permanently changed. We can put them back with another call:

```
>>> names.reverse()
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
```

If we had simply wanted to use them in a reversed order **WITHOUT ALTERING THE ACTUAL LIST**, we could call the `reversed` function:

```
>>> list(reversed(names))
['Shemp', 'Curly', 'Moe', 'Larry']
>>> names
['Larry', 'Moe', 'Curly', 'Shemp']
```

It's really easy to read an entire file directly into a list with `readlines` (this preserves newlines), but you should be sure that you have at least as much memory on your machine as the file is big. Compare these various ways to read an entire file. `read` will give you the contents as one string, and newlines will be present to denote the end of each line:

```
>>> open('input.txt').read()
'first line\nsecond line\nthird line\nfourth line\n'
```

Whereas `readlines` will return a list of strings broken on the newlines (but not removing them):

```
>>> open('input.txt').readlines()
['first line\n', 'second line\n', 'third line\n', 'fourth line\n']
```

Calling `read().splitlines()` will suck in the whole file, then break on the newlines, removing them in the process:

```
>>> open('input.txt').read().splitlines()
['first line', 'second line', 'third line', 'fourth line']
```

Similarly, you can `read().split()` to break all the input on spaces to get the words:

```
>>> open('input.txt').read().split()
['first', 'line', 'second', 'line', 'third', 'line', 'fourth', 'line']
```

Here is a version that uses `readlines()`:

```
$ cat -n tac2.py
     1    #!/usr/bin/env python3
     2    """print a file in reverse"""
     3
     4    import sys
     5    import os
     6
     7    args = sys.argv[1:]
```

```
 8    if len(args) != 1:
 9        print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
10        sys.exit(1)
11
12    file = args[0]
13    if not os.path.isfile(file):
14        print('"{}" is not a file'.format(file))
15        sys.exit(1)
16
17    lines = open(file).readlines()
18    lines.reverse()
19
20    for line in lines:
21        print(line, end='')
```

This version uses the `reversed` function:

```
$ cat -n tac3.py
 1    #!/usr/bin/env python3
 2    """print a file in reverse"""
 3
 4    import sys
 5    import os
 6
 7    args = sys.argv[1:]
 8    if len(args) != 1:
 9        print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
10        sys.exit(1)
11
12    file = args[0]
13    if not os.path.isfile(file):
14        print('"{}" is not a file'.format(file))
15        sys.exit(1)
16
17    lines = open(file).readlines()
18
19    for line in reversed(lines):
20        print(line, end='')
```

And finally I will introduce the `with/open` convention that you will see in Python:

```
$ cat -n tac4.py
 1    #!/usr/bin/env python3
 2    """print a file in reverse"""
 3
 4    import sys
 5    import os
 6
```

```
 7    args = sys.argv[1:]
 8    if len(args) != 1:
 9        print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
10        sys.exit(1)
11
12    file = args[0]
13    if not os.path.isfile(file):
14        print('"{}" is not a file'.format(file))
15        sys.exit(1)
16
17    with open(file) as fh:
18        lines = fh.readlines()
19        for line in reversed(lines):
20            print(line, end='')
```

Picnic

Here is a little memory game you might have played with your bored siblings on family car trips:

```
$ ./picnic.py
What are you bringing? [q to quit] chips
We'll have chips.
What else are you bringing? [q to quit] ham sammich
We'll have chips and ham sammich.
What else are you bringing? [q to quit] Coke
We'll have chips, ham sammich, and Coke.
What else are you bringing? [q to quit] cupcakes
We'll have chips, ham sammich, Coke, and cupcakes.
What else are you bringing? [q to quit] apples
We'll have chips, ham sammich, Coke, cupcakes, and apples.
What else are you bringing? [q to quit] q
Bye.
```

Each person introduces a new item, and the other person has to remember all the previous items and add a new one. This is a classic "stack" that can be implemented with lists:

```
$ cat -n picnic.py
     1    #!/usr/bin/env python3
     2    """What are you bringing to the picnic?"""
     3
     4    # --------------------------------------------------
     5    def joiner(items):
     6        """properly conjuct items"""
```

```
 7        num_items = len(items)
 8        if num_items == 0:
 9            return ''
10        elif num_items == 1:
11            return items[0]
12        elif num_items == 2:
13            return ' and '.join(items)
14        else:
15            items[-1] = 'and ' + items[-1]
16            return ', '.join(items)
17
18    # -----------------------------------------------
19    def main():
20        """start here"""
21        items = []
22
23        while True:
24            item = input('What {}are you bringing? [q to quit] '.format('else ' if ite
25            if item == 'q':
26                break
27            elif len(item.strip()) > 0:
28                if item in items:
29                    print('You said "{}" already.'.format(item))
30                else:
31                    items.append(item)
32                    print("We'll have {}.".format(joiner(items.copy())))
33
34        print('Bye.')
35
36    # -----------------------------------------------
37    if __name__ == '__main__':
38        main()
```

One bug that got me in writing this program was line 32. Because I mutate the last item in the list in my `joiner` function, I was actually mutating the original list! I had to learn to pass `items.copy()` so as to work on a copy of the data and not the actual list.

Insults

Sometimes (esp when writing games) you may want a random selection from a list of items. Here is an insult generator that draws from the fabulous vocabulary of Shakespeare:

```
$ cat -n insult.py
```

```
 1  #!/usr/bin/env python3
 2  """Shakespearean insult generator"""
 3
 4  import sys
 5  import random
 6
 7  ADJECTIVES = """
 8  scurvy old filthy scurilous lascivious foolish rascaly gross rotten corrupt
 9  foul loathsome irksome heedless unmannered whoreson cullionly false filthsome
10  toad-spotted caterwauling wall-eyed insatiate vile peevish infected
11  sodden-witted lecherous ruinous indistinguishable dishonest thin-faced
12  slanderous bankrupt base detestable rotten dishonest lubbery
13  """.split()
14
15  NOUNS = """
16  knave coward liar swine villain beggar slave scold jolthead whore barbermonger
17  fishmonger carbuncle fiend traitor block ape braggart jack milksop boy harpy
18  recreant degenerate Judas butt cur Satan ass coxcomb dandy gull minion
19  ratcatcher maw fool rogue lunatic varlet worm
20  """.split()
21
22  args = sys.argv[1:]
23  num = 5
24  if len(args) > 0 and args[0].isdigit():
25      num = int(args[0])
26
27  for i in range(0, num):
28      adjs = []
29      for j in range(0, 3):
30          adjs.append(random.choice(ADJECTIVES))
31
32      print('You {} {}!'.format(', '.join(adjs), random.choice(NOUNS)))
```

```
$ ./insult.py foo
You bankrupt, cullionly, detestable milksop!
You foul, indistinguishable, false Satan!
You lascivious, scurilous, bankrupt villain!
You lascivious, lecherous, rotten jack!
You toad-spotted, base, foolish Satan!
$ ./insult.py 3
You detestable, cullionly, wall-eyed scold!
You peevish, caterwauling, caterwauling traitor!
You thin-faced, foul, dishonest Judas!
```

Notice how the program takes an optional argument that I expect to be an integer. On line 24, I test both that there is an argument present and that it `isdigit()` before attempting to use it as a number. The real work is done by

the `random.choice` function to grab my adjectives and noun. The `"""` operator lets us write strings with newlines, then we `split` the long string into words. This is a common idiom in Python. Notice the use of `append` to grow the list of adjectives on line 30, then we `join` them on line 32.

Synthetic Biology

Lists could represent biological entities such as promotor, coding, and terminator regions. Let's say we wanted to design synthetic microbes where we tested all possible permutations of these regions with each other to see if we were able to increase production of a desired enzyme. Since the operation is N^3, I will only show the output for 2 genes:

```
$ ./recomb.py 2
N = "2"
  1: ('P1', 'C1', 'T1')
  2: ('P1', 'C1', 'T2')
  3: ('P1', 'C2', 'T1')
  4: ('P1', 'C2', 'T2')
  5: ('P2', 'C1', 'T1')
  6: ('P2', 'C1', 'T2')
  7: ('P2', 'C2', 'T1')
  8: ('P2', 'C2', 'T2')
```

Here is the Python code:

```
$ cat -n recomb.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
```

```
19    if not 2 <= num_genes <= 10:
20        print('NUM_GENES must be greater than 1, less than 10')
21        sys.exit(1)
22
23    promotors = []
24    coding = []
25    terminators = []
26    for i in range(0, num_genes):
27        n = str(i + 1)
28        promotors.append('P' + n)
29        coding.append('C' + n)
30        terminators.append('T' + n)
31
32    print('N = "{}"'.format(num_genes))
33    for i, combo in enumerate(chain(product(promotors, coding, terminators))):
34        print('{:3}: {}'.format(i + 1, combo))
```

The heavy lifting is being done on line 33 by the `product` function we get from the `itertools` module. Because this function is given three lists to cross, it returns a list of three sub-lists which I want to combine into one list with `chain`. Then I call the `enumerate` function (shown in the first section) to get the list index and the list member in one loop so I don't have to keep up with a counter variable.

I don't like lines 26-30, so I tried rewriting using a list comprehension (one of the most useful things you can do with lists). Here's an example of using list comprehensions to square the numbers from 1 to 4:

```
>>> [x ** 2 for x in range(1, 5)]
[1, 4, 9, 16]
```

You can add a predicate for item selection to the end:

```
>>> [x ** 2 for x in range(1, 5) if x % 2 == 0]
[4, 16]
```

Here is the comprehensions in the program (lines 23-25):

```
$ cat -n recomb2.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
```

```
    11        print('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
    19    if not 2 <= num_genes <= 10:
    20        print('NUM_GENES must be greater than 1, less than 10')
    21        sys.exit(1)
    22
    23    promotors = ['P' + str(n + 1) for n in range(0, num_genes)]
    24    coding = ['C' + str(n + 1) for n in range(0, num_genes)]
    25    terminators = ['T' + str(n + 1) for n in range(0, num_genes)]
    26
    27    print('N = "{}"'.format(num_genes))
    28    for i, combo in enumerate(chain(product(promotors, coding, terminators))):
    29        print('{:3}: {}'.format(i + 1, combo))
```

But these lines are identical with the exception of the character I'm using, so I can put that code into a little function:

```
$ cat -n recomb3.py
     1    #!/usr/bin/env python3
     2    """Show recominations"""
     3
     4    import os
     5    import sys
     6    from itertools import product, chain
     7
     8    args = sys.argv[1:]
     9
    10    if len(args) != 1:
    11        print('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
    12        sys.exit(1)
    13
    14    if not args[0].isdigit():
    15        print('"{}" does not look like an integer'.format(args[0]))
    16        sys.exit(1)
    17
    18    num_genes = int(args[0])
    19    if not 2 <= num_genes <= 10:
    20        print('NUM_GENES must be greater than 1, less than 10')
    21        sys.exit(1)
    22
    23    def gen(prefix):
```

```
24          return [prefix + str(n + 1) for n in range(0, num_genes)]
25
26      promotors = gen('P')
27      coding = gen('C')
28      terminators = gen('T')
29
30      print('N = "{}"'.format(num_genes))
31      for i, combo in enumerate(chain(product(promotors, coding, terminators))):
32          print('{:3}: {}'.format(i + 1, combo))
```

Now all the repeated code is in the `gen` function (line 23-24), and I simply call that for each character I want.