"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:

```
>>> names = ['Larry', 'Moe', 'Curly', 'Shemp']
>>> for name in names:
...   print(name)
...
```

```
Larry
Moe
Curly
Shemp
```

Or the characters in a word:

```
>>> 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. Since it's so annoying to deal with zero-offset counting, we can tell `enumerate` to `start` at 1:

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

```
  4 l
  5 y
```

You can turn a list around with the `reversed` function:

```
>>> reversed(names)
<list_reverseiterator object at 0x109e490f0>
```

What we have here is a failure to communicate. You expected to see the list of names in the reverse order, but what you got was a promise from Python to give you that list when you actually need it. In the REPL, we can use the `list` function:

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

So you can also use that to reverse a word:

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

OK, well, I wanted the word "tac" and not the list of letters in "tac"! 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). Notice that I don't have to use `list` because the `join` function will iterate on the `reversed` result:

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

Identifying Palindromes

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
     8  def main():
     9      """main"""
    10      args = sys.argv[1:]
    11
```

```
    12      if len(args) != 1:
    13          print('Usage: {} STR'.format(os.path.basename(sys.argv[0])))
    14          sys.exit(1)
    15
    16      word = args[0]
    17      rev = ''.join(reversed(word))
    18
    19      print('"{}" is{} a palindrome.'.format(
    20          word, '' if word.lower() == rev.lower() else ' NOT'))
    21
    22
    23  if __name__ == '__main__':
    24      main()
$ ./word_is_palindrome.py
Usage: word_is_palindrome.py STR
$ ./word_is_palindrome.py foobar
"foobar" is NOT a palindrome.
$ ./word_is_palindrome.py foof
"foof" is a palindrome.
```

Finding Palindromes in a File

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
     8  def main():
     9      """main"""
    10      args = sys.argv[1:]
    11      min_length = 2
    12
    13      if len(args) != 1:
    14          print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    15          sys.exit(1)
    16
    17      file = args[0]
    18
    19      if not os.path.isfile(file):
    20          print('"{}" is not a file'.format(file))
```

```
    21          sys.exit(1)
    22
    23      for line in open(file):
    24          for word in line.lower().split():
    25              # skip short words
    26              if len(word) > min_length:
    27                  rev = ''.join(reversed(word))
    28                  if rev == word:
    29                      print(word)
    30
    31
    32  if __name__ == '__main__':
    33      main()
$ cat input.txt
My sister Anna was looking for a kayak around noon when her mom
suggested she might rather enjoy a new racecar instead.
$ ./find_palindromes.py
Usage: find_palindromes.py FILE
$ ./find_palindromes.py foo
"foo" is not a file
$ ./find_palindromes.py input.txt
anna
kayak
noon
mom
racecar
```

Here we've added the ability to take as an argument the name of a file. By include `import os`, we can ask the OS (operating system) if the argument we were given is the name of an existing file. If this returns `False`, then we print an error message and exit with a non-zero exit code. Then we can use a `for` loop to iterate over each line in the `open` file handle, then another `for` loop to iterate over each `word` in the result of lowercasing and splitting the line of text. We skip short words and then print the current `word` if it is a palindrome.

Tetranucleotide Composition

A common operation in bioinformatics is to determine sequence composition and is the first problem (id = `DNA`) to solve at the rosalind.info website. 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
 8  def main():
 9      """main"""
10      args = sys.argv[1:]
11
12      if len(args) != 1:
13          print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
14          sys.exit(1)
15
16      dna = args[0]
17      num_a, num_c, num_g, num_t = 0, 0, 0, 0
18
19      for base in dna:
20          if base == 'a' or base == 'A':
21              num_a += 1
22          elif base == 'c' or base == 'C':
23              num_c += 1
24          elif base == 'g' or base == 'G':
25              num_g += 1
26          elif base == 't' or base == 'T':
27              num_t += 1
28
29      print(' '.join([str(num_a), str(num_c), str(num_g), str(num_t)]))
30
31
32  if __name__ == '__main__':
33      main()
```

```
$ cat dna.txt
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
$ ./dna1.py `cat dna.txt`
3 2 1 1
```

We initialize 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 base in the input DNA string, checking both the upper- and lowercase version of each base to determine which variable to increment.

To save 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  def main():
 8      """main"""
 9      args = sys.argv[1:]
10
11      if len(args) != 1:
12          print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
13          sys.exit(1)
14
15      dna = args[0]
16
17      num_a, num_c, num_g, num_t = 0, 0, 0, 0
18
19      for base in dna.lower():
20          if base == 'a':
21              num_a += 1
22          elif base == 'c':
23              num_c += 1
24          elif base == 'g':
25              num_g += 1
26          elif base == 't':
27              num_t += 1
28
29      print(' '.join([str(num_a), str(num_c), str(num_g), str(num_t)]))
30
31  if __name__ == '__main__':
32      main()
```

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  def main():
 8      """main"""
 9      args = sys.argv[1:]
10
11      if len(args) != 1:
12          print('Usage: {} DNA'.format(os.path.basename(sys.argv[0])))
```

```
    13          sys.exit(1)
    14
    15      dna = args[0]
    16
    17      num_a, num_c, num_g, num_t = 0, 0, 0, 0
    18
    19      for base in dna.lower():
    20          if base == 'a':
    21              num_a += 1
    22          elif base == 'c':
    23              num_c += 1
    24          elif base == 'g':
    25              num_g += 1
    26          elif base == 't':
    27              num_t += 1
    28
    29      print('{} {} {} {}'.format(num_a, num_c, num_g, num_t))
    30
    31  if __name__ == '__main__':
    32      main()
```

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

```
$ diff dna2.py dna3.py
29c29
<     print(' '.join([str(num_a), str(num_c), str(num_g), str(num_t)]))
---
>     print('{} {} {} {}'.format(num_a, num_c, num_g, num_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. Note that this program shows how the argument can be a string or the name of a file.

```
$ cat -n run_length.py
     1  #!/usr/bin/env python3
     2  """Compress text/DNA by marking repeated letters"""
     3
     4  import os
     5  import sys
     6
     7
     8  def main():
```

```
 9      """main"""
10      args = sys.argv[1:]
11
12      if len(args) != 1:
13          print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
14          sys.exit(1)
15
16      # If the argument is a file, the text should be the file contents
17      arg = args[0]
18      text = ''
19      if os.path.isfile(arg):
20          text = ''.join(open(arg).read().split())
21      else:
22          text = arg.strip()
23
24      # Make sure we have something
25      if len(text) == 0:
26          print('No usable text')
27          sys.exit(1)
28
29      counts = []
30      count = 0
31      prev = None
32      for letter in text:
33          # We are at the start
34          if prev is None:
35              prev = letter
36              count = 1
37          # This letter is the same as before
38          elif letter == prev:
39              count += 1
40          # This is a new letter, so record the count
41          # of the previous letter and reset the counter
42          else:
43              counts.append((prev, count))
44              count = 1
45              prev = letter
46
47      # get the last letter after we fell out of the loop
48      counts.append((prev, count))
49
50      for letter, num in counts:
51          print('{}{}'.format(letter, '' if num == 1 else num), end='')
52
53      print()
54
```

```
    55
    56  if __name__ == '__main__':
    57      main()
$ ./run_length.py AAAAACCCGATT
A5C3GAT2
$ ./run_length.py dna.txt
AGCT4CAT2CTGACTGCA2CG3CA2TATGTCTCTGTGTG2AT2A7GAGTGTCTGATAGCAGC
```

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 -n tac1.py
     1  #!/usr/bin/env python3
     2  """print a file in reverse"""
     3
     4  import sys
```

```
 5  import os
 6
 7
 8  def main():
 9      """main"""
10      args = sys.argv[1:]
11      if len(args) != 1:
12          print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
13          sys.exit(1)
14
15      file = args[0]
16      if not os.path.isfile(file):
17          print('"{}" is not a file'.format(file))
18          sys.exit(1)
19
20      lines = []
21      for line in open(file):
22          lines.append(line)
23
24      lines.reverse()
25
26      for line in lines:
27          print(line, end='')
28
29
30  if __name__ == '__main__':
31      main()
```

```
$ cat quick.txt
The quick brown
fox jumps
over the
lazy dog.
$ ./tac1.py quick.txt
lazy dog.
over the
fox jumps
The quick brown
```

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 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
```

```
 8  def main():
 9      """main"""
10      args = sys.argv[1:]
11      if len(args) != 1:
12          print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
13          sys.exit(1)
14
15      file = args[0]
16      if not os.path.isfile(file):
17          print('"{}" is not a file'.format(file))
18          sys.exit(1)
19
20      lines = open(file).readlines()
21      lines.reverse()
22
23      for line in lines:
24          print(line, end='')
25
26
27  if __name__ == '__main__':
28      main()
```

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

```
    23          print(line, end='')
    24
    25
    26  if __name__ == '__main__':
    27      main()
```

And finally I will introduce the `with/open` convention that you will see in Python. Notice that I will call the variable `fh` to stand for "file handle" which is not the same as the file name (which is a string):c

```
$ cat -n tac4.py
     1  #!/usr/bin/env python3
     2  """print a file in reverse"""
     3
     4  import sys
     5  import os
     6
     7
     8  def main():
     9      """main"""
    10      args = sys.argv[1:]
    11      if len(args) != 1:
    12          print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    13          sys.exit(1)
    14
    15      file = args[0]
    16      if not os.path.isfile(file):
    17          print('"{}" is not a file'.format(file))
    18          sys.exit(1)
    19
    20      with open(file) as fh:
    21          lines = fh.readlines()
    22          for line in reversed(lines):
    23              print(line, end='')
    24
    25
    26  if __name__ == '__main__':
    27      main()
```

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? [! to quit] chips
We'll have chips.
What else are you bringing? [! to quit] ham sammich
We'll have chips and ham sammich.
What else are you bringing? [! to quit] co-cola
We'll have chips, ham sammich, and co-cola.
What else are you bringing? [! to quit] cupcakes
We'll have chips, ham sammich, co-cola, and cupcakes.
What else are you bringing? [! to quit] apples
We'll have chips, ham sammich, co-cola, cupcakes, and apples.
What else are you bringing? [! to quit] !
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 and an infinite loop:

```
$ cat -n picnic.py
     1  #!/usr/bin/env python3
     2  """What are you bringing to the picnic?"""
     3
     4
     5  # --------------------------------------------------
     6  def joiner(items):
     7      """properly conjuct items"""
     8      num_items = len(items)
     9      if num_items == 0:
    10          return ''
    11      elif num_items == 1:
    12          return items[0]
    13      elif num_items == 2:
    14          return ' and '.join(items)
    15      else:
    16          items[-1] = 'and ' + items[-1]
    17          return ', '.join(items)
    18
    19
    20  # --------------------------------------------------
    21  def main():
    22      """main"""
    23      items = []
    24
    25      while True:
    26          item = input('What {}are you bringing? [! to quit] '.format(
    27              'else ' if items else ''))
    28          if item == '!':
    29              break
```

```
30          elif len(item.strip()) > 0:
31              if item in items:
32                  print('You said "{}" already.'.format(item))
33              else:
34                  items.append(item)
35                  # pass a copy because it gets mutated!
36                  print("We'll have {}.".format(joiner(items.copy())))
37
38      print('Bye.')
39
40
41  # -----------------------------------------------
42  if __name__ == '__main__':
43      main()
```

One bug that got me in writing this program was in calling my `joiner` with the list of `items`. I was surprised to learn that in mutating the list (by adding "and" before the last item when there are more than 2 items), I was mutating the *original list*! I learned to pass `items.copy()` so as to work on a copy of the data and not the actual list.

Insults

Sometimes (especially 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
     9  corrupt foul loathsome irksome heedless unmannered whoreson cullionly
    10  false filthsome toad-spotted caterwauling wall-eyed insatiate vile
    11  peevish infected sodden-witted lecherous ruinous indistinguishable
    12  dishonest thin-faced slanderous bankrupt base detestable rotten
    13  dishonest lubbery
    14  """.strip().split()
    15
    16  nouns = """
    17  knave coward liar swine villain beggar slave scold jolthead whore
```

```
    18  barbermonger fishmonger carbuncle fiend traitor block ape braggart
    19  jack milksop boy harpy recreant degenerate Judas butt cur Satan ass
    20  coxcomb dandy gull minion ratcatcher maw fool rogue lunatic varlet
    21  worm
    22  """.strip().split()
    23
    24
    25  def main():
    26      """main"""
    27      args = sys.argv[1:]
    28      num = int(args[0]) if len(args) > 0 and args[0].isdigit() else 5
    29
    30      for _ in range(num):
    31          adjs = [random.choice(adjectives) for _ in range(3)]
    32          noun = random.choice(nouns)
    33          print('You {} {}!'.format(', '.join(adjs), noun))
    34
    35
    36  if __name__ == '__main__':
    37      main()
$ ./insult.py
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.

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
Usage: recomb.py NUM_GENES
$ ./recomb.py foo
"foo" is not an integer
$ ./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. Note that I replaced repeated calls to `print`/`sys.exit(1)` with a new function I call `die`. I use the `product` function from `itertools` to get all the combinations of the promoter, coding, and terminator regions.

```
$ cat -n recomb.py
     1  #!/usr/bin/env python3
     2  """Show recominations"""
     3
     4  import os
     5  import sys
     6  from itertools import product
     7
     8  def die(msg):
     9      """print and exit with an error"""
    10      print(msg)
    11      sys.exit(1)
    12
    13  def main():
    14      """main"""
    15      args = sys.argv[1:]
    16
    17      if len(args) != 1:
    18          die('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
    19
    20      if not args[0].isdigit():
    21          die('"{}" is not an integer'.format(args[0]))
    22
    23      num_genes = int(args[0])
    24      min_num = 2
```

```
25      max_num = 10
26      if not min_num <= num_genes <= max_num:
27          die('NUM_GENES must be between {} and {}'.format(min_num, max_num))
28
29      promotors, coding, term = ([], [], [])
30      for n in [str(i) for i in range(1, num_genes + 1)]:
31          promotors.append('P' + n)
32          coding.append('C' + n)
33          term.append('T' + n)
34
35      print('N = "{}"'.format(num_genes))
36      for i, combo in enumerate(product(promotors, coding, term), start=1):
37          print('{:4}: {}'.format(i, ' - '.join(combo)))
38
39
40  if __name__ == '__main__':
41      main()
```

Line 30 is an example of a list comprehension. They read awkwardly because the `for` loop that introduces the variable happens after you start seeing the variable:

```
>>> [n for n in range(1, 4)]
[1, 2, 3]
>>> [n ** n for n in range(1, 4)]
[1, 4, 27]
```

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

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

That `%` operator is the remainder after division, so if the remainder after dividing a number by 2 is 0, then the number is even.

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
     7
     8
     9  def die(msg):
    10      """print and exit with an error"""
    11      print(msg)
    12      sys.exit(1)
```

```
13
14
15  def main():
16      """main"""
17      args = sys.argv[1:]
18
19      if len(args) != 1:
20          die('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
21
22      if not args[0].isdigit():
23          die('"{}" does not look like an integer'.format(args[0]))
24
25      num_genes = int(args[0])
26      min_num = 2
27      max_num = 10
28      if not min_num <= num_genes <= max_num:
29          die('NUM_GENES must be between {} and {}'.format(min_num, max_num))
30
31      ranger = range(1, num_genes + 1)
32      promotors = ['P' + str(n + 1) for n in ranger]
33      coding = ['C' + str(n + 1) for n in ranger]
34      term = ['T' + str(n + 1) for n in ranger]
35
36      print('N = "{}"'.format(num_genes))
37      for i, combo in enumerate(product(promotors, coding, term), start=1):
38          print('{:4}: {}'.format(i + 1, ' - '.join(combo)))
39
40
41  if __name__ == '__main__':
42      main()
```

But lines 32-34 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
     7
     8
     9  def die(msg):
    10      """print and exit with an error"""
    11      print(msg)
    12      sys.exit(1)
```

```
13
14
15  def main():
16      """main"""
17      args = sys.argv[1:]
18
19      if len(args) != 1:
20          die('Usage: {} NUM_GENES'.format(os.path.basename(sys.argv[0])))
21
22      if not args[0].isdigit():
23          die('"{}" does not look like an integer'.format(args[0]))
24
25      num_genes = int(args[0])
26      if not 2 <= num_genes <= 10:
27          die('NUM_GENES must be greater than 1, less than 10')
28
29      def gen(prefix):
30          return [prefix + str(n) for n in range(1, num_genes + 1)]
31
32      print('N = "{}"'.format(num_genes))
33      combos = product(gen('P'), gen('C'), gen('T'))
34      for i, combo in enumerate(combos, start=1):
35          print('{:4}: {}'.format(i, ' - '.join(combo)))
36
37
38  if __name__ == '__main__':
39      main()
```

Now all the repeated code is in the `gen` function, and I simply call that for each character I want.

Exercise: head.py

Create a Python program called `head.py` that expects one or two arguments. If there are no arguments, print a "Usage" statement. The first argument is required and much be a regular file; if it is not, print " is not a file" and exit *with an error code*. The second argument is optional. If given, it must be a positive number (non-zero); if it is not, then print "lines () must be a positive number". If no argument is provided, use a default value of 3. You can expect that the test will only give you a value that can be safely converted to a number using the `int` function. If given good input, it should act like the normal `head` utility and print the expected number of lines from the given file.

```
$ ./head.py
Usage: head.py FILE [NUM_LINES]
$ ./head.py foo
```

```
foo is not a file
$ ./head.py files/issa.txt
Selected Haiku by Issa

Don't worry, spiders,
$ ./head.py files/issa.txt 5
Selected Haiku by Issa

Don't worry, spiders,
I keep house
casually.
```

Exercise: cat_n.py

Create a Python program called `cat_n.py` that expects exactly one argument which is a regular file and prints usage statement if either condition fails. It should print each line of the file argument preceeded by the line number which is right-justified in spaces and a colon. You may the format '{:5}: {}' to make it look exactly like the output below, but the test is just checking for a leading space, some number(s), a colon, and the line of text.

```
$ ./cat_n.py
Usage: cat_n.py FILE
$ ./cat_n.py foo
foo is not a file
$ ./cat_n.py files/sonnet-29.txt
    1: Sonnet 29
    2: William Shakespeare
    3:
    4: When, in disgrace with fortune and men's eyes,
    5: I all alone beweep my outcast state,
    6: And trouble deaf heaven with my bootless cries,
    7: And look upon myself and curse my fate,
    8: Wishing me like to one more rich in hope,
    9: Featured like him, like him with friends possessed,
   10: Desiring this man's art and that man's scope,
   11: With what I most enjoy contented least;
   12: Yet in these thoughts myself almost despising,
   13: Haply I think on thee, and then my state,
   14: (Like to the lark at break of day arising
   15: From sullen earth) sings hymns at heaven's gate;
   16: For thy sweet love remembered such wealth brings
   17: That then I scorn to change my state with kings.
```

Exercise: gc.py

Write a Python program called `gc.py` that takes a single positional argument which should be a file. Die with a warning if the argument is not a file. For each line in the file, print the line number and the percentage of the characters on that line that are a "G" or "C" (case-insensitive).

```
$ ./gc.py
usage: gc.py [-h] FILE
gc.py: error: the following arguments are required: FILE
$ ./gc.py foo
"foo" is not a file
$ ./gc.py samples/sample1.txt
  1:   9%
  2:  19%
  3:  19%
  4:  22%
  5:  32%
  6:  21%
```

Exercise: vowel_counter.py

Write a Python program called `vowel_counter.py` that counts the number of vowels in a single string. Be sure your subject and verb agree in number, and use proper plurals.

```
$ ./vowel_counter.py
Usage: vowel_counter.py STRING
$ ./vowel_counter.py for
There is 1 vowel in "for."
$ ./vowel_counter.py elliptical
There are 4 vowels in "elliptical."
$ ./vowel_counter.py YYZ
There are 0 vowels in "YYZ."
```

Exercise: grid.py

Write a Python program called `grid.py` that will create a square grid from the argument given on the command line. The program will expect one positional argument; if the number of arguments is not exactly one, print a "usage" statement and exit *with an error code*. The test suite will only provide integer values, so you can assume it is safe to use `int` to convert the input from a string to an integer. The number provided must be in the range of 2 to 9 (inclusive). If it is not, print "NUM () must be between 1 and 9" and exit *with an error code*. You

will square the given number to create a grid (so think of the number as how many rows and columns).

```
$ ./grid.py
Usage: grid.py NUM
$ ./grid.py 1
NUM (1) must be between 1 and 9
$ ./grid.py -1
NUM (-1) must be between 1 and 9
$ ./grid.py 10
NUM (10) must be between 1 and 9
$ ./grid.py 2
  1  2
  3  4
$ ./grid.py 5
  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15
 16 17 18 19 20
 21 22 23 24 25
$ ./grid.py 8
  1  2  3  4  5  6  7  8
  9 10 11 12 13 14 15 16
 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32
 33 34 35 36 37 38 39 40
 41 42 43 44 45 46 47 48
 49 50 51 52 53 54 55 56
 57 58 59 60 61 62 63 64
```

Exercise: histy.py

Write a Python program called `histy.py` that takes one or more integer values as positional argumements and creates a histogram of the values in their sorted order using the `-c|--character` (default "|") repeated by the number. Also accept a `-m|--minimum` option that is the minimum value to print a number as well as a `-s|--scale` option that will be used to scale the number of characters printed so that large numbers won't line wrap in your terminal.

```
$ ./histy.py
usage: histy.py [-h] [-c str] [-m int] [-s int] int [int ...]
histy.py: error: the following arguments are required: int
$ ./histy.py -h
usage: histy.py [-h] [-c str] [-m int] [-s int] int [int ...]

Histogrammer
```

```
positional arguments:
  int                   Inputs

optional arguments:
  -h, --help            show this help message and exit
  -c str, --character str
                        Character to represent (default: |)
  -m int, --minimum int
                        Minimum value to print (default: 1)
  -s int, --scale int   Scale inputs (default: 1)
$ ./histy.py 3 1 2
  1 |
  2 ||
  3 |||
$ ./histy.py 300 100 200 -s 100
100 |
200 ||
300 |||
$ ./histy.py 300 100 200 -s 100 -c '#'
100 #
200 ##
300 ###
$ ./histy.py 300 100 200 -s 100 -c '#' -m 150
200 ##
300 ##
```

Exercise: column.py

Create a Python program called `column.py` that takes a list of words and creates a columnar output of each word and their length. If given no words as positional, command-line arguments, print a usage statement. For the output, first print a header of "word" and "len", then lines which are the width of the longest word and the longest numbers with a minimum for each of the column headers themselves. The words should be left-justified in the first column and the numbers should be right-justified in the second column.

```
$ ./column.py
Usage: column.py WORD [WORD...]
$ ./column.py a an the
word  len
----  ---
a       1
an      2
the     3
```

```
$ ./column.py `cat out/1.in`
word               len
-----------------  ---
Iphis                5
cyclone              7
dare                 4
umbraculiferous     15
indescribableness   17
prattling            9
pediculine          10
pondwort             8
lava                 4
adipoma              7
```

Exercise: rot13.py

Write a Python program called `rot13.py` that will encrypt/decrypt input text by shifting the text by a given `-s|--shift` argument or will move each character halfway through the alphabet, e.g., "a" becomes "n," "b" becomes "o," etc. The text to rotate should be provided as a single positional argument to your program and can either be a text file, text on the command line, or `-` to indicate STDIN so that you can round-trip data through your program to ensure you are encrypting and decrypting properly.

```
$ ./rot13.py
usage: rot13.py [-h] [-s int] str
rot13.py: error: the following arguments are required: str
$ ./rot13.py -h
usage: rot13.py [-h] [-s int] str

Argparse Python script

positional arguments:
  str                   Input text, file, or "-" for STDIN

optional arguments:
  -h, --help            show this help message and exit
  -s int, --shift int   Shift arg (default: 0)
$ ./rot13.py AbCd
NoPq
$ ./rot13.py AbCd -s 2
CdEf
$ ./rot13.py fox.txt
Gur dhvpx oebja sbk whzcf bire gur ynml qbt.
$ ./rot13.py fox.txt | ./rot13.py -
```

```
The quick brown fox jumps over the lazy dog.
$ ./rot13.py -s 3 fox.txt | ./rot13.py -s -3 -
The quick brown fox jumps over the lazy dog.
```

Exercise: transpose.py

Write a Python program called `transpose.py` that will read a file in ABC notation (https://en.wikipedia.org/wiki/ABC_notation) and transpose the melody line up or down by a given `-s|--shift` argument. Like the `rot13` exercise, it might be helpful to think of the space of notes (`ABCDEFG`) as a list which you can roll through. For instance, if you have the note `c` and want to transpose up a (minor) third (`-s 3`), you would make the new note `e`; similarly if you have the note `F` and you go up a (major) third, you get `A`. You will not need to worry about the actual number of semitones that you are being asked to shift, as the previous example showed that we might be shifting by a major/minor/augmented/diminished/pure interval. The purpose of the exercise is simply to practice with lists.

```
$ ./transpose.py
usage: transpose.py [-h] [-s int] FILE
transpose.py: error: the following arguments are required: FILE
$ ./transpose.py -h
usage: transpose.py [-h] [-s int] FILE

Tranpose ABC notation

positional arguments:
  FILE                  Input file

optional arguments:
  -h, --help            show this help message and exit
  -s int, --shift int   Interval to shift (default: 2)
$ ./transpose.py foo
"foo" is not a file
$ ./transpose.py songs/legacy.abc -s 1
--shift "1" must be between 2 and 8
$ ./transpose.py songs/legacy.abc
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:A
AGA CBC | aga abc | AGA CBC | e2B BGE |
```

```
AGA CBC | aga abc | baf feC |1 eCB BGE :|2 eCB BCe |:
fgf feC | eCB BCe | fgf feC | aeC BCe |
fgf feC | e2e efg | agf feC |1 eCB BCe :|2 eCB BGE |]
</score>
```

Exercise: tictactoe.py

Write a Python program named `tictactoe.py` that accepts four named arguments:

- `-s|--state`: The state of the board (type `str`, default "." [9 dots])
- `-p|--player`: The player to modify the state (type `str`, valid "X" or "O", no default)
- `-c|--cell`: The cell to alter (type `int`, valid 1-9, default None)
- `-h|--help`: Indication to print "usage" and exit (no error)

Your program must do the following:

- Print a usage with the `-h` or `--help` argument and exit
- Ensure the `--state` of the board is exactly 9 characters (for the 9 cells) and comprised only of ".", "X", and "O"
- Print a board with the given state where each character of the `--state` is the contents of each cell; when the state for a cell is ".", print the number of the cell
- If provided only `--player` or `--cell`, print the message "Must provide both –player and –cell" and exit *with an error code*
- If provided `--player`, ensure it is only an "X" or "O"; if not, print the expected message and exit *with an error code*
- If provided `--cell`, ensure it is in 1-9; if not, print the expected message and exit *with an error code*
- If provided both `--player` and `--cell`, modify the `--state` to change the given "cell" to the value of "player"; e.g., if `--state` is "." `--cell` is 1, and `--player` is "X", the state should change to "X........" and then print the board with the new state

```
$ ./tictactoe.py -h
usage: tictactoe.py [-h] [-s str] [-p str] [-c str]

Tic-Tac-Toe board

optional arguments:
  -h, --help            show this help message and exit
  -s str, --state str   Board state (default: .........)
  -p str, --player str  Player (default: )
  -c str, --cell str    Cell to apply -p (default: )
```

Print an "empty" (no cells selected) grid with no arguments:

```
$ ./tictactoe.py
-------------
| 1 | 2 | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
```

Die if given a bad `--state` (too short, not made entirely of "-XO"). Note the feedback where the bad input from the user is repeated in the error message:

```
$ ./tictactoe.py -s abcdefghi
State "abcdefghi" must be 9 characters of only ., X, O
$ ./tictactoe.py --state XXO
State "XXO" must be 9 characters of only ., X, O
```

Die on invalid player:

```
$ ./tictactoe.py -p A
Invalid player "A", must be X or O
```

Die on invalid cell:

```
$ ./tictactoe.py -c 10
Invalid cell "10", must be 1-9
$ ./tictactoe.py -c 0
Invalid cell "0", must be 1-9
```

Die on a cell argument that is not an `int` (note that `argparse` will create this if you indicate the "type" as `int` for the argument):

```
$ ./tictactoe.py -c foo
usage: tictactoe.py [-h] [-s str] [-p str] [-c int]
tictactoe.py: error: argument -c/--cell: invalid int value: 'foo'
```

Print a valid state:

```
$ ./tictactoe.py -s XXO..X.OO
-------------
| X | X | O |
-------------
| 4 | 5 | X |
-------------
| 7 | O | O |
-------------
```

Mutate an initial state as described by `--cell` and `--player`. For instance, starting from the default state, change cell 2 to "X":

```
$ ./tictactoe.py -p X -c 2
-------------
| 1 | X | 3 |
-------------
| 4 | 5 | 6 |
-------------
| 7 | 8 | 9 |
-------------
```

From a state of "XXO..X.OO", change cell 7 to an "O":

```
$ ./tictactoe.py -s XXO..X.OO -p O -c 7
-------------
| X | X | O |
-------------
| 4 | 5 | X |
-------------
| O | O | O |
-------------
```

Die if the `--cell` is already taken:

```
$ ./tictactoe.py -s XXO..X.OO -p O -c 1
Cell 1 already taken
```