9 Iteration

Key terms: `for` `while`

Reading: Daw-Ran Liou's "You (Probably) Don't Need For-Loops"[13]

Exercise: Solve Gauss's problem: find the sum of the integers from 1 to 100.

As in other procedural languages, explicit loops are offered here in two styles. `while` is the same as in other languages: it continues until a condition becomes `False`. `for` touches on each object in an iterable collections and for that reason is inherently safer than `while`: it prevents infinite loops and off-by-one errors, as well as freeing the programmer from counting the elements in a collection.

Nonetheless, explicit loops are not often needed, and avoiding them saves code and the risk of bugs. The functional model of programming uses implicit loops and more nested expressions to make execution flow more implicit, and relieve the programmer of having to remember as much. Consider the task of combining the command line arguments around copies of the word "and": `' and '.join(sys.argv)` is more robust and succinct than a loop that assembles the string. We can rely on certain built-in features that do iteration implicitly, including functions like `range()`, `max()`, `any()`, `sum()`, and the slice notation (`[::]`).

Let's review several ways to solve a simple iterative problem, counting to 100. First comes a painful C-style example, which counts and increments as long as `i < 100`. Because the code is stateful, unexpected states could occur and create bugs. Since Python lists are iterable, the work maintaining and checking the index adds nothing.

Counting iteration (don't do this):

```
i = 0
while i < 100:
 print (i)
 i += 1
```

Second is iteration over the members of a collection. This one is not so bad. The code is shorter and more robust because it does leaves out the fiddliest bits. It does still have a stateful loop where surprises could occur.

For-each iteration (use sparingly):

```
for i in range(100):
 print(i)
```

The most robust programs are those where the least can go wrong. These two implicit iteration styles rely on the built-in function `range()` to create a sequence of numbers. It may take you some concentration to understand these styles of programming, because they are not *imperative*, like most languages, but *functional*. Their elegance is in the fact that there is no loop at all. Expect to see Pythonic code that uses this style, because it tends to produce shorter code that is more resistant to bugs.

The first uses the "splat", lstinline|*|, to explode a collection into its member elements; `print()` receives 100 arguments to print, plus one named argument asking it to separate those elements with newlines. The second maps the numbers into strings, then joining them with newlines in between to create one long string.

Implicit iteration (prefer this):

```
print(*range(100),sep='\n')

print('\n'.join(map(str,range(100))))
```

Next, we will address exception handling.

[13]`https://medium.com/python-pandemonium/never-write-for-loops-again-91a5a4c84baf`