Building and Breaking Down Lists

Outline

Pattern: Accumulating a Result

We often iterate over a full list, doing a computation over each element in turn.

```
lst_sum = 0
for elt in [1,2,3,4,5]:
    lst_sum += elt
```

What does the loop above do?

The General Pattern

```
# initialize aggregator variable(s)
var1 = init_var1()
for elt in lst:
    # aggregate elt into aggregator
    var1 = process(var1, elt)
```

We sometimes call this a *left fold*, since it processes elements from the left of the list to the right.

Exercise: Compute the Maximum

Using the left fold idea, compute the greatest element of the list `lst`. You can assume the list only contains non-negative numbers, for simplicity.

Folding with an Index

We saw `enumerate` last time. Our `find_greatest` could be implemented to return the index of the greatest element like so:

```
def find_greatest_index(lst):
    greatest = -1
    index = None
    for i, elt in enumerate(lst):
        if elt > greatest:
            index = i
            greatest = elt
    return index
```

Exercise: Find an Element

Using the left fold idea and `enumerate`, write a function that takes a list and a number and finds the index of the last occurrence of the number in the list, returning None if it's not found.

Going Backwards

We have seen `enumerate`, but there are other iterator adapters as well. We can use `reversed` to simplify that last function and avoid the need to accumulate anything:

```
def find_last_occurrence(lst, num):
    for i, elt in reversed(enumerate(lst)):
        if elt == num:
            return i
    return None
```

Right Folds

If we have a left fold, it stands to reason there should be a right fold that starts from the last element and goes backwards. We can use `reversed`—right folds often show up when implementing programming languages!

Example: Stack-Based Calculator

```
def calculator(instructions):
    stack = []
    for inst in reversed(instructions):
        if inst == "+":
            stack.append(stack.pop() + stack.pop())
        else:
            stack.append(inst)
    return stack[0]
# (9 + 12) + (2 + 5)
calculator(['+', '+', 12, 9, '+', 5, 2])
```

Folds are Very Powerful

As the last example shows, folds are very flexible. We can aggregate into numbers, lists, strings, or any other type!

We can specialize folds into two common cases; for now it's important to recognize these patterns, but in the future you will see tools like `filter` and `map` to avoid the need for the loop altogether.

Folds

Folds take in a list, apply some operation over the elements from an initial value, and return a new value. In today's class we'll mostly be discussing functions that create and return new lists, rather than modifying the original input list in-place.

Mapping a Computation Over a List

We often want to transform a list of things by transforming each thing individually and in order:

```
def double_all(lst):
    output = []
    for elt in lst:
        output.append(elt*2)
    return output
```

Mapping

In mathematical terms, we are *mapping* the *doubling* operation over `lst`.

The output list will always have the same number of elements as the input, in the same corresponding order.

Many natural uses of map occur in programming: uppercasing or lowercasing a string, gathering email addresses of users, etc.

Mapping Recipe

The general mapping recipe is a special kind of fold:

```
def mapping_fn(lst):
    output = []
    for elt in lst:
        output.append(f(elt))
    return output
```

Exercise

Write a mapping function that squares each number in a list.

Write a mapping function that replaces space characters in a string with dashes (-).

Filtering a List by a Computation

Another special case of folding is *filtering*: accepting only those elements which pass some test.

```
def keep_fives(lst):
    output = []
    for elt in lst:
        if elt == 5:
            output.append(elt)
    return output
```

Filtering Recipe

```
def filter_fn(lst):
    output = []
    for elt in lst:
        if test(elt):
            output.append(elt)
    return output
```

Exercise

Write a filtering function that removes even numbers from a list of numbers.

Write a filtering function that removes the odd-numbered indices from a list (use `enumerate`).

Filtering and Mapping

We can also combine mapping and filtering. The following composes map and filter:

```
def keep_and_double_threes(lst):
    output = []
    for elt in lst:
        # filter step
        if elt % 3 == 0:
            # map step
            output.append(elt * 2)
    return output
```

Exercise: How would the code be different if we wanted to first double elements and keep the ones that became multiples of 3? In other words, to compose filter and map?

Working on Lists in Parallel

We have already seen exercises where we work on elements of two lists side by side:

```
lst1 = [1,2,3]
lst2 = [4,5,6]
for i in range(len(lst1)):
    lst1[i] += lst2[i]
```

Working on Lists in Parallel

We want to avoid direct indexing as much as possible because it's easy to introduce bugs. `enumerate` isn't super useful here since we still do need to access the second list. We'll wrap up today by introducing `zip`:

```
lst1 = [1,2,3]
lst2 = [4,5,6]
output = []
for elt1, elt2 in zip(lst1, lst2):
    output.append(elt1+elt2)
```

`zip`

`zip` is a bit like `enumerate`, but instead of producing a sequence of numbers and a sequence of elements, it produces a sequence of pairs of elements drawn from each of the input sequences.

Exercise

Use `zip` to implement a function `get_fullnames(firstnames, lastnames)`, where `firstnames` and `lastnames` are each lists of strings.
The output should be a list of full names; e.g. `get_fullnames(["Alice", "Bob"], ["Adams", "Bobsleigh"])` should produce `["Alice Adams", "Bob Bobsleigh"]`.

Quiz