MAPPING AND ACCUMULATION

Mapping and Accumulation

Mapping → applying a callable to each element of an iterable

→ `map(fn, iterable)`

Accumulation → reducing an iterable down to a single value

→ `sum(iterable)` calculates the sum of every element in an iterable

→ `min(iterable)` returns the minimal element of the iterable

→ `max(iterable)` returns the maximal element of the iterable

→ `reduce(fn, iterable, [initializer])`

→ `fn` is a function of two arguments

→ applies `fn` cumulatively to elements of iterable

map

You should already be familiar with `map` → quick review

`map(fn, iterable)` applies `fn` to every element of `iterable`, and returns an iterator (lazy)

→ `fn` must be a callable that requires a single argument

```
map(lambda x: x**2, [1, 2, 3, 4])
```
→ `1, 4, 9, 16` → lazy iterator

Of course, we can easily do the same thing using a generator expression too

```
maps = (fn(item) for item in iterable)
```

reduce

You should already be familiar with `reduce` → quick review

Suppose we want to find the sum of all elements in an iterable: `l = [1, 2, 3, 4]`

```
sum(l)    → 1 + 2 + 3 + 4 = 10
```

```
reduce(lambda x, y: x + y, l)  → 1
                               → 1 + 2 = 3
                               → 3 + 3 = 6
                               → 6 + 4 = 10
```

To find the product of all elements:

```
reduce(lambda x, y: x * y, l)  → 1
                               → 1 * 2 = 2
                               → 2 * 3 = 6
                               → 6 * 4 = 24
```

We can specify a different "start" value in the reduction

```
reduce(lambda x, y: x + y, l, 100)   → 110
```

itertools.starmap

starmap is very similar to map

→ it unpacks every sub element of the iterable argument, and passes that to the map function

→ useful for mapping a multi-argument function on an iterable of iterables

```
l = [ [1, 2], [3, 4] ]        map(lambda item: item[0] * item[1], l)   → 2, 12
```

We can use starmap: `starmap(operator.mul, l)` → 2, 12

we could also just use a generator expression to do the same thing:

```
(operator.mul(*item) for item in l)
```

We can of course use iterables that contain more than just two values:

```
l = [ [1, 2, 3], [10, 20, 30], [100, 200, 300] ]
starmap(lambda: x, y, z: x + y + z, l)   → 6, 60, 600
```

`itertools.accumulate(iterable, fn)` → lazy iterator

The `accumulate` function is very similar to the `reduce` function

But it returns a (lazy) iterator producing all the intermediate results

→ `reduce` only returns the final result

Unlike `reduce`, It does not accept an initializer

Note the argument order is not the same!

```
reduce(fn, iterable)
accumulate(iterable, fn)
```

→ in `accumulate`, `fn` is optional

→ defaults to addition

Example

```
l = [1, 2, 3, 4]
```

```
functools.reduce(operator.mul, l)
```

- 1
- 1 * 2 = 2
- 2 * 3 = 6
- 6 * 4 = 24

→ 24

```
itertools.accumulate(l, operator.mul)
```

→ 1, 2, 6, 24

Coding Exercises