REDUCING FUNCTIONS

Reducing Functions in Python

These are functions that recombine an iterable recursively, ending up with a single return value

Also called accumulators, aggregators, or folding functions.

Example: Finding the maximum value in an iterable

$a_0, a_1, a_2, \ldots, a_{n-1}$

`max(a, b)` → maximum of a and b

```
result = a_0
result = max(result, a_1)
result = max(result, a_2)
…
result = max(result, a_{n-1})
```

→ max value in $a_0, a_1, a_2, \ldots, a_{n-1}$

Because we have not studied iterables in general, we will stay with the special case of sequences.

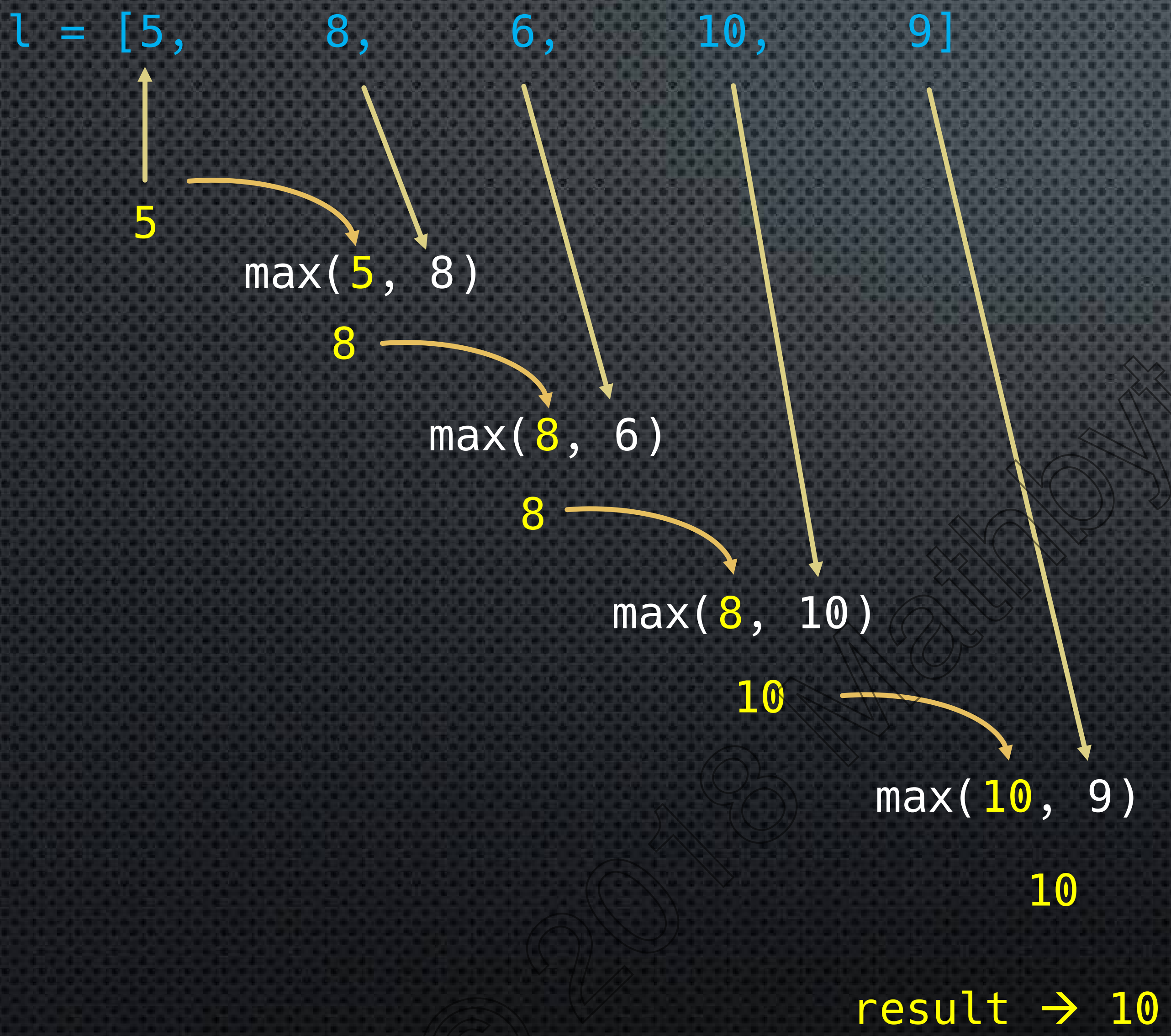
(i.e. we can use indexes to access elements in the sequence)

Using a loop

```
l = [5, 8, 6, 10, 9]

max_value = lambda a, b: a if a > b else b

def max_sequence(sequence):
    result = sequence[0]
    for e in sequence[1:]:
        result = max_value(result, e)
    return result
```

```
result = 5

result = max(5, 8) = 8

result = max(8, 6) = 8

result = max(8, 10) = 10

result = max(10, 9) = 10

      result → 10
```

Notice the sequence of steps:



```
l = [5, 8, 6, 10, 9]

5
max(5, 8)
8
max(8, 6)
8
max(8, 10)
10
max(10, 9)
10

result → 10
```

To calculate the min:

```
l = [5, 8, 6, 10, 9]

min_value = lambda a, b: a if a < b else b

def min_sequence(sequence):
    result = sequence[0]
    for e in sequence[1:]:
        result = min_value(result, e)
    return result
```

All we really needed to do was to change the function that is repeatedly applied.

In fact we could write:

```
def _reduce(fn, sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = fn(result, x)
    return result
```

```
_reduce(lambda a, b: a if a > b else b, l) → maximum

_reduce(lambda a, b: a if a < b else b, l) → minimum
```

Adding all the elements in a list

```
add = lambda a, b: a+b


l = [5, 8, 6, 10, 9]


def _reduce(fn, sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = fn(result, x)
    return result


_reduce(add, l)
```

```
result = 5

result = add(5, 8) = 13

result = add(13, 6) = 19

result = add(19, 10) = 29

result = add(29, 9) = 38

      result → 38
```

The functools module

Python implements a reduce function that will handle any iterable, but works similarly to what we just saw

```
from functools import reduce

l = [5, 8, 6, 10, 9]

 reduce(lambda a, b: a if a > b else b, l)        → max → 10

 reduce(lambda a, b: a if a < b else b, l)        → min → 5

 reduce(lambda a, b: a + b, l)                    → sum → 38
```

reduce works on any iterable

```
reduce(lambda a, b: a if a < b else b, {10, 5, 2, 4})    → 2

reduce(lambda a, b: a if a < b else b, 'python')          → h

reduce(lambda a, b: a + ' ' + b, ('python', 'is', 'awesome!'))
                                                           → 'python is awesome'
```

Built-in Reducing Functions

Python provides several common reducing functions:

`min`	`min([5, 8, 6, 10, 9])`	→ 5
`max`	`max([5, 8, 6, 10, 9])`	→ 10
`sum`	`sum([5, 8, 6, 10, 9])`	→ 38
`any`	`any(l)` →	`True` if any element in `l` is truthy `False` otherwise
`all`	`all(l)` →	`True` if every element in `l` is truthy `False` otherwise

Using reduce to reproduce any

```
l = [0, '', None, 100]
```

```
result = bool(0) or bool('') or bool(None) or bool(100)
```

Note: `0 or '' or None or 100 → 100` but we want our result to be True/False
so we use `bool( )`

Here we just need to repeatedly apply the or operator to the truth values of each element

```
result = bool(0)                      → False

result = result or bool('')           → False

result = result or bool(None)         → False

result = result or bool(100)          → True
```

```
reduce(lambda a, b: bool(a) or bool(b), l)    → True
```

Calculating the product of all elements in an iterable

No built-in method to do this

But very similar to how we added all the elements in an iterable or sequence:

```
[1, 3, 5, 6]   → 1 * 3 * 5 * 6
```

```
reduce(lambda a, b: a * b, l)
```

```
res = 1

res = res * 3 = 3

res = res * 5 = 3 * 5 = 15

res = res * 6 = 15 * 6 = 90     = 1 * 3 * 5 * 6
```

Special case: Calculating n!

n! = 1 * 2 * 3 * … * n   5! = 1 * 2 * 3 * 4 * 5

```
range(1, 6)        → 1, 2, 3, 4, 5

range(1, n+1)      → 1, 2, 3, …, n
```

To calculate `n!` we need to find the product of all the elements in `range(1, n+1)`

`reduce(lambda a, b: a * b, range(1, 5+1))` → 5!

The reduce initializer

The reduce function has a third (optional) parameter: initializer (defaults to None)

If it is specified, it is essentially like adding it to the front of the iterable.

It is often used to provide some kind of default in case the iterable is empty.

```
l = []
reduce(lambda x, y: x+y, l)          → exception

l = []
reduce(lambda x, y: x+y, l, 1)       → 1

l = [1, 2, 3]
reduce(lambda x, y: x+y, l, 1)       → 7

l = [1, 2, 3]
reduce(lambda x, y: x+y, l, 100)     → 106
```

Code