COMBINATORICS

The `itertool` module contains a few functions for generating

permutations combinations

It also has a function to generate the Cartesian product of multiple iterables

All these functions return lazy iterators

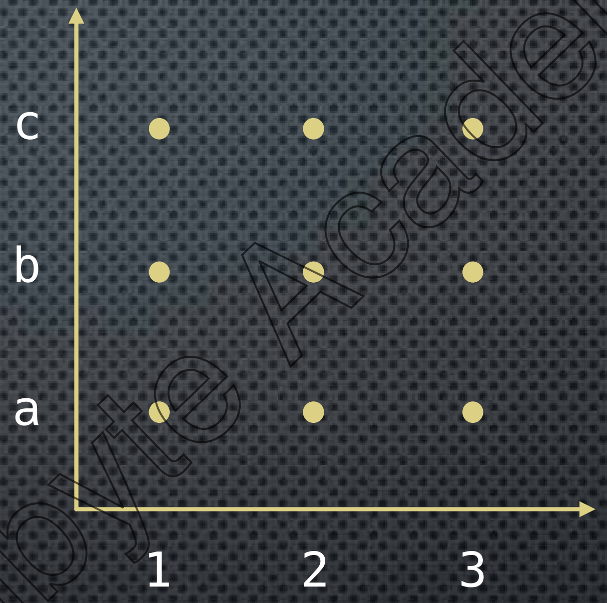
Cartesian Product

{1, 2, 3} x {a, b, c}

(1, a)
(2, a)
(3, a)

(1, b)
(2, b)
(3, b)

(1, c)
(2, c)
(3, c)



2-dimensional: { | }

n-dimensional: { | }

Cartesian Product

Let's say we wanted to generate the Cartesian product of two lists:

```
l1 = [1, 2, 3]     l2 = ['a', 'b', 'c', 'd']
```

→ notice not same length

```
def cartesian_product(l1, l2):
    for x in l1:
        for y in l2:
            yield (x, y)
```

```
cartesian_product(l1, l2)
```

```
→ (1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), …, (3,'d')
```

```
itertools.product(*args)        → lazy iterator


l1 = [1, 2, 3]     l2 = ['a', 'b', 'c', 'd']

product(l1, l2)  → (1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), …, (3,'d')

l3 = [100, 200]

product(l1, l2, l3)    → (1, 'a', 100), (1, 'a', 200),
                         (1, 'b', 100), (1, 'b', 200),
                         (1, 'c', 100), (1, 'c', 200),
                         …
                         (3, 'd', 100), (3, 'd', 200)
```

Permutations

This function will produce all the possible permutations of a given iterable

In addition, we can specify the length of each permutation

→ maxes out at the length of the iterable

```
itertools.permutations(iterable, r=None)
```

→ r is the size of the permutation

→ r = None means length of each permutation is the length of the iterable

Elements of the iterable are considered unique based on their position, not their value

→ if iterable produces repeat values
then permutations will have repeat values too

Combinations

Unlike permutations, the order of elements in a combination is not considered

→ OK to always sort the elements of a combination

Combinations of length r, can be picked from a set

- without replacement → once an element has been picked from the set it cannot be picked again

- with replacement → once an element has been picked from the set it can be picked again

```
itertools.combinations(iterable, r)

itertools.combinations_with_replacement(iterable, r)
```

Just like for permutations:

the elements of an iterable are unique based on their position, not their value

The different combinations produced by these functions are sorted based on the original ordering in the iterable

Code Exercises