*args

Recall from iterable unpacking

```
a, b, c = (10, 20, 30)    →    a = 10    b = 20    c = 30
```

Something similar happens when positional arguments are passed to a function:

```
def func1(a, b, c):
    # code
```

```
func1(10, 20, 30)    →    a = 10    b = 20    c = 30
```

*args

Recall also: `a, b, *c = 10, 20, 'a', 'b'` → `a=10` `b=20` `c=['a', 'b']`

Something similar happens when positional arguments are passed to a function:

```
def func1(a, b, *c):
    # code
```

this is a tuple, not a list

`func1(10, 20, 'a', 'b')` → `a=10` `b=20`

`c=('a', 'b')`

The * parameter name is arbitrary – you can make it whatever you want

It is customary (but not required) to name it `*args`

```
def func1(a, b, *args):
    # code
```

*args exhausts positional arguments

You cannot add more positional arguments after *args

this is actually OK -- covered in next lecture

```
def func1(a, b, *args, d):
    # code
```

This will not work!

```
func1(10, 20, 'a', 'b', 100)
```

❌

Unpacking arguments

```
def func1(a, b, c):
    # code
```

```
l = [10, 20, 30]
```

This will not work: `func1(l)` ✗

But we can unpack the list first and then pass it to the function

```
func1(*l)      →  a = 10     b = 20     c = 30
```

Code