

Tomorrow:

Assignment 1 due at 11:59 p.m.

Assignment 2 out 5 p.m.

In Python, we can write code to work with data represented as:

Integers

42, -3, 10000

Floating-point numbers

0.0, -3.6, 4.2

Booleans

True, False

Text strings

"Alan Turing", "50%", "\$3.50"

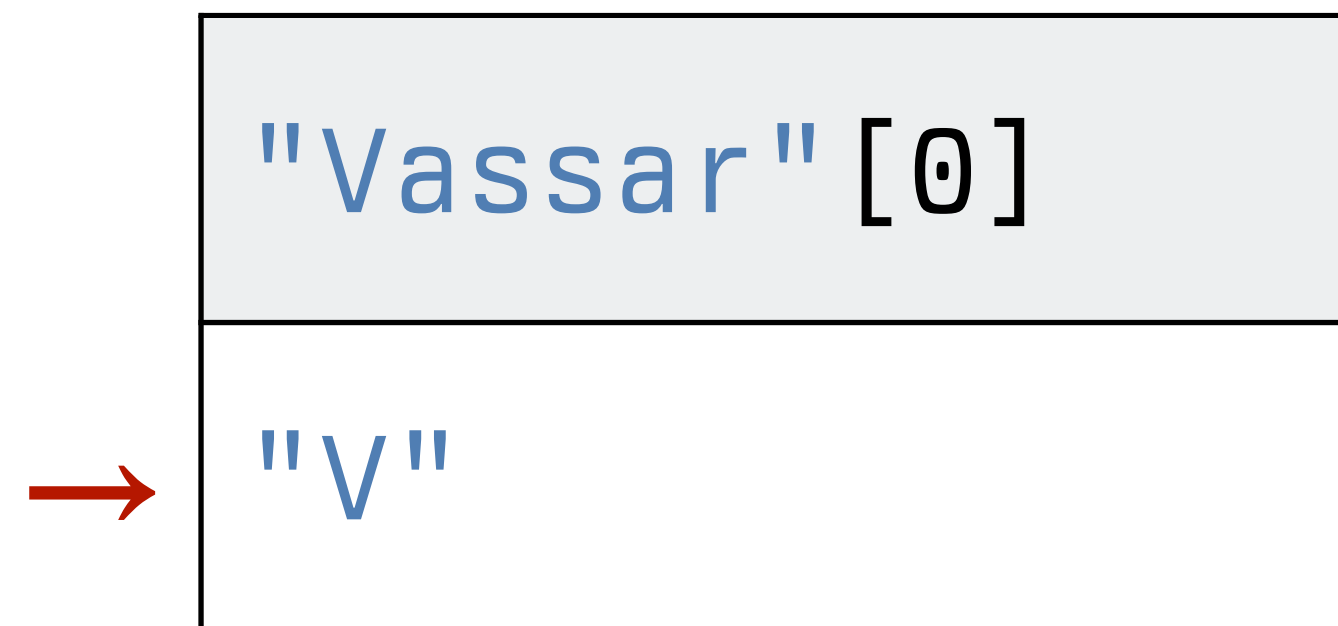
→ Notebook: *Warm-up*

A text string is a *sequence* of *characters* – letters, numbers, punctuation.

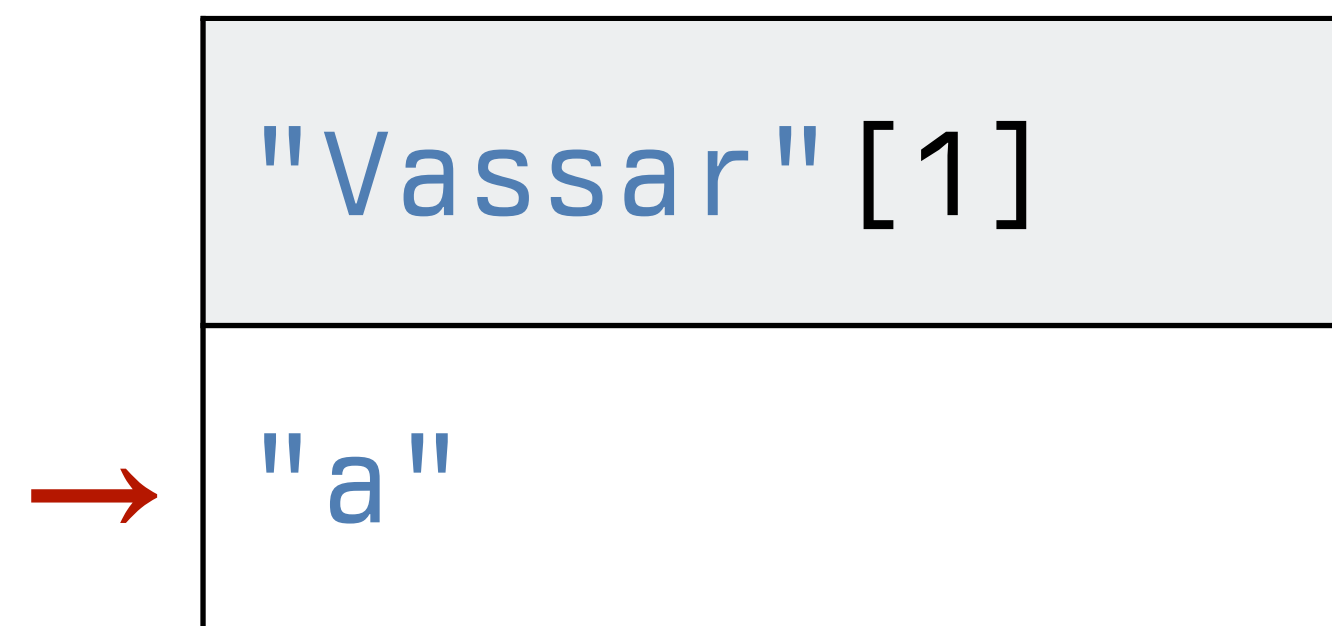
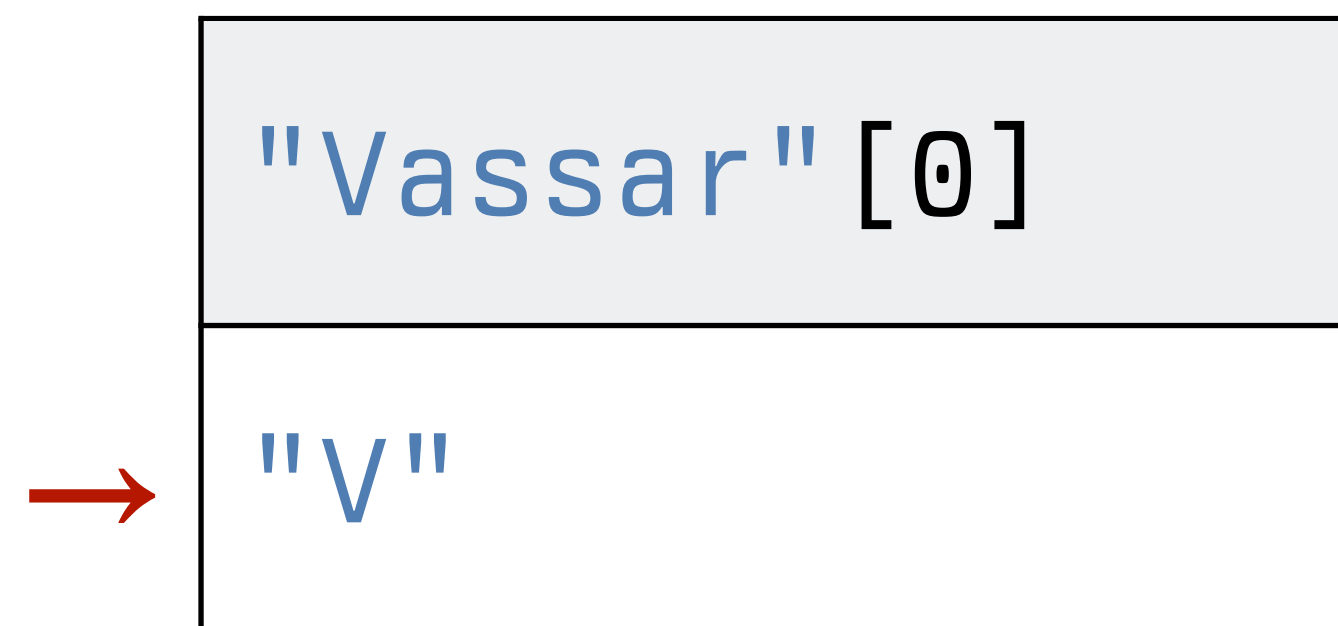
We can access these characters by their position in the sequence.

	0	1	2	3	4	5	
"	v	a	s	s	a	r	"

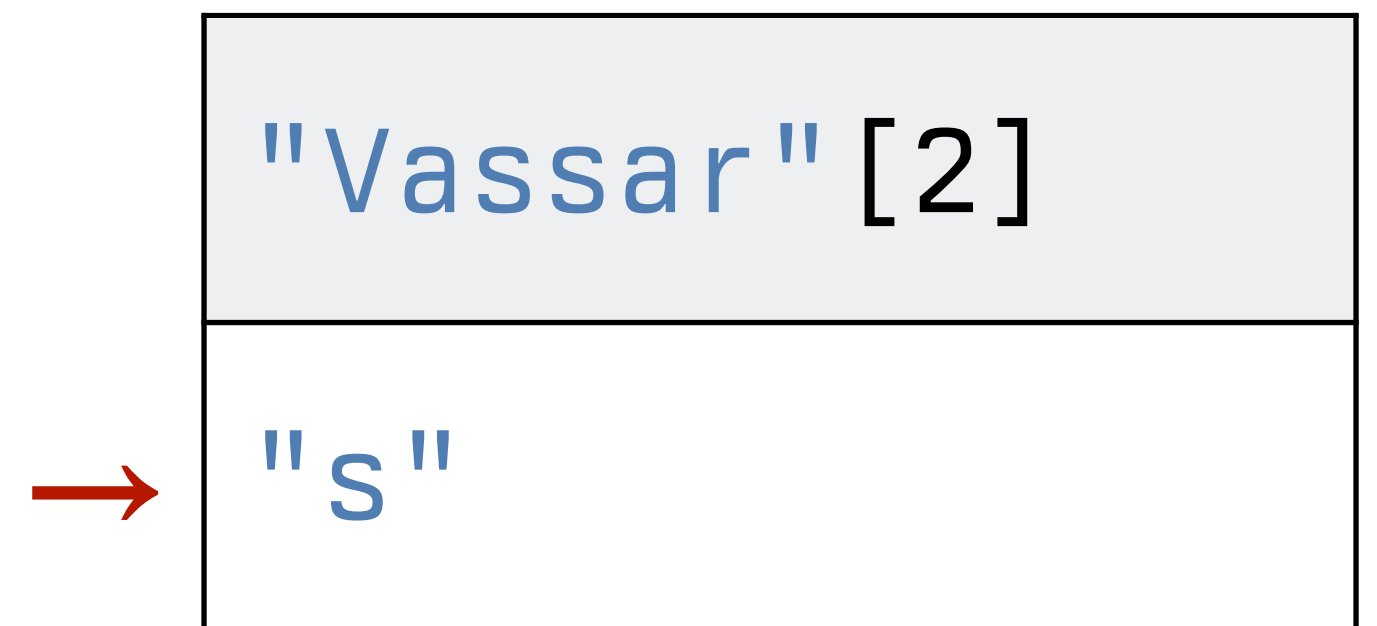
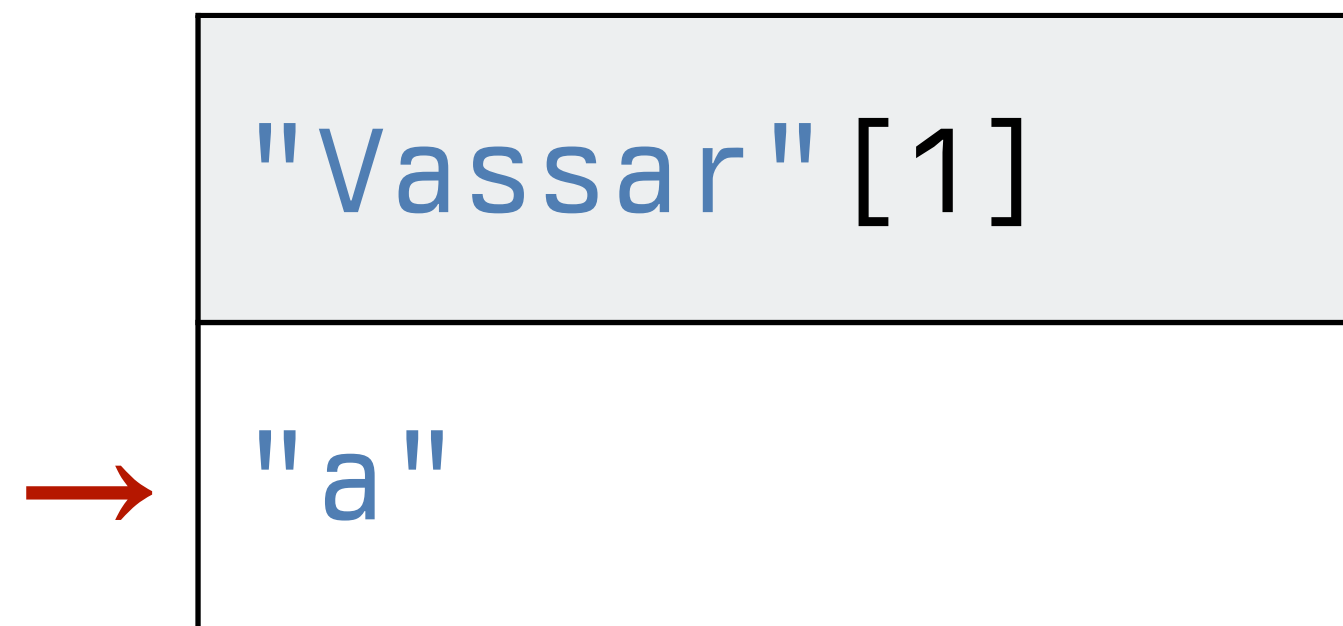
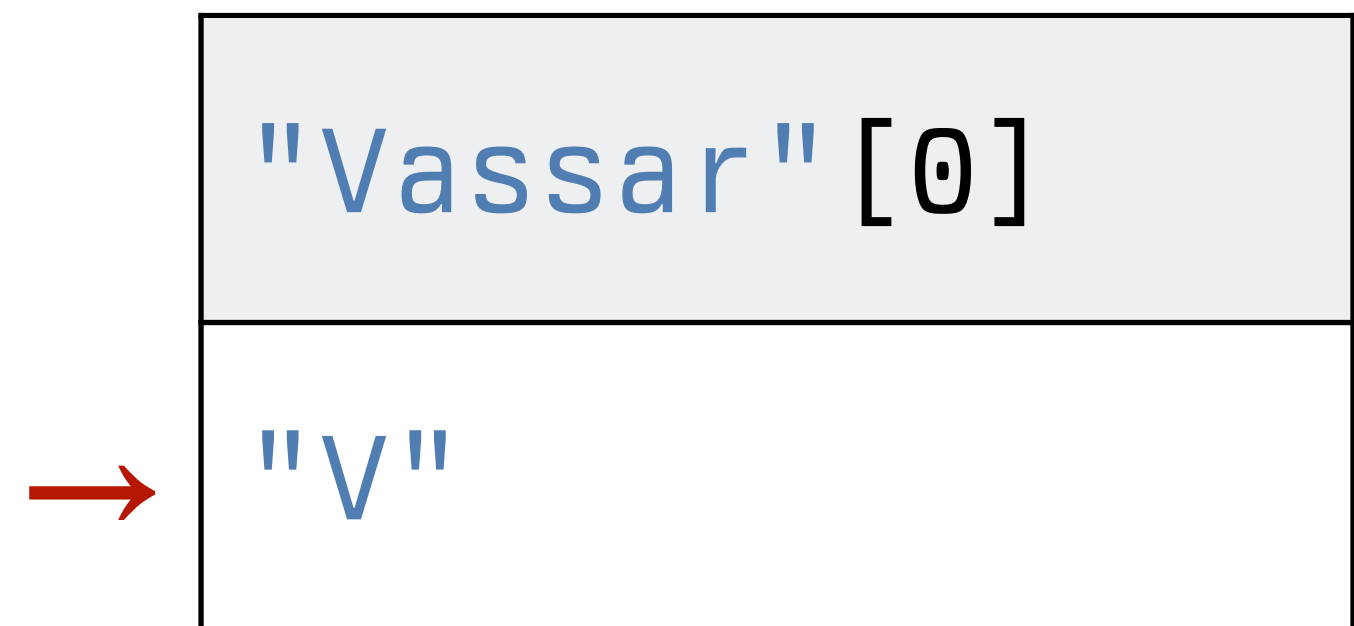
0	1	2	3	4	5		
"	V	a	s	s	a	r	"



0 | 1 | 2 | 3 | 4 | 5
" v | a | s | s | a | r "



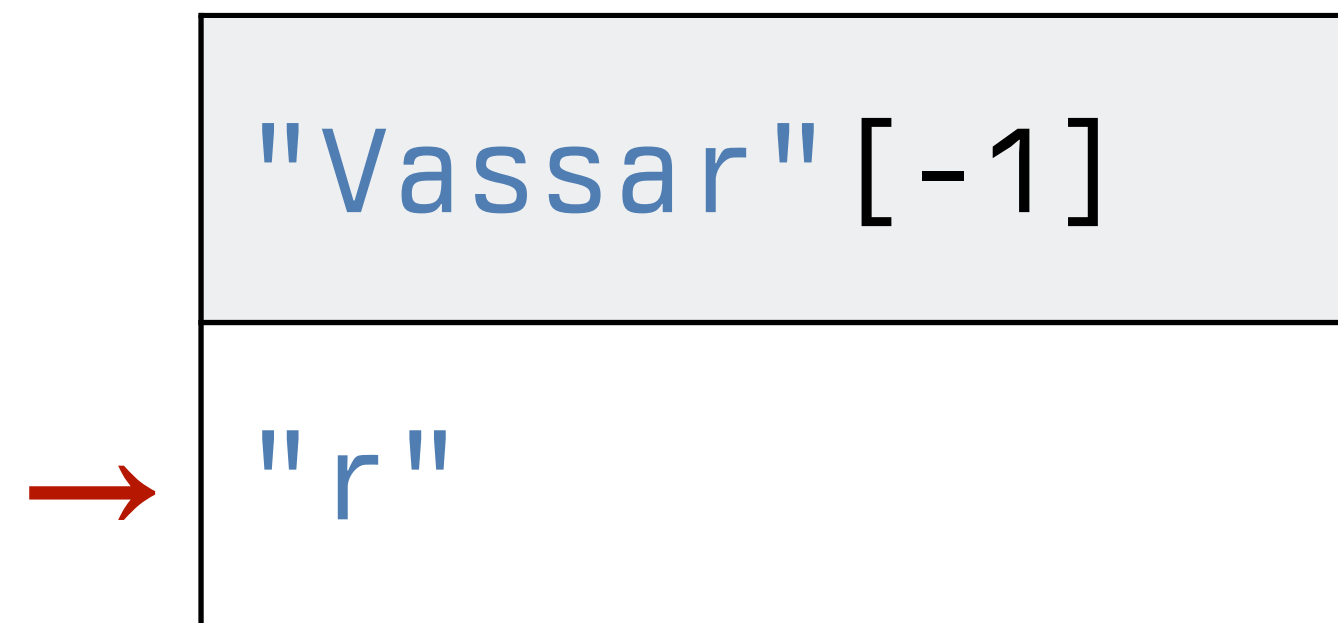
0 | 1 | 2 | 3 | 4 | 5
" v | a | s | s | a | r "



	0	1	2	3	4	5	
"	v	a	s	s	a	r	"
	-6	-5	-4	-3	-2	-1	

We can use negative numbers to index from the end of the string.

	0	1	2	3	4	5	
"	V	a	s	s	a	r	"
	-6	-5	-4	-3	-2	-1	



We can use negative numbers to index from the end of the string.

	0	1	2	3	4	5	
"	V	a	s	s	a	r	"
	-6	-5	-4	-3	-2	-1	

→

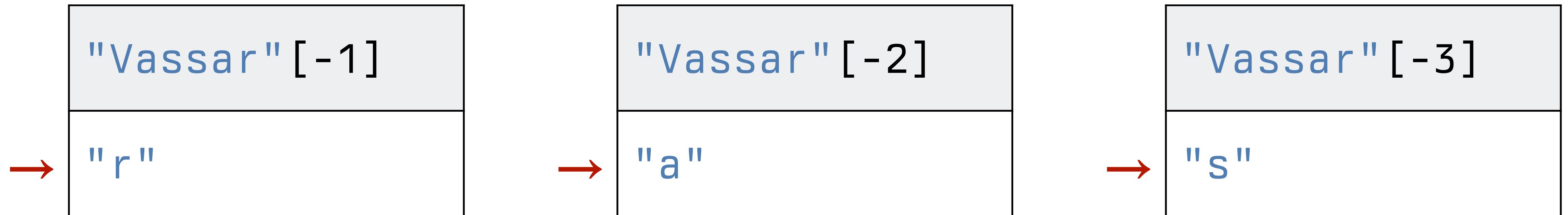
"Vassar" [-1]
"r"

→

"Vassar" [-2]
"a"

We can use negative numbers to index from the end of the string.

	0	1	2	3	4	5	
"	V	a	s	s	a	r	"
	-6	-5	-4	-3	-2	-1	



We can use negative numbers to index from the end of the string.

We can also *slice* a sequence to access a subsequence.

Instead of a single index, we specify

[*start:end*]

*the position to start
at, including that
character*

*the position to end
before, excluding that
character*

Think–pair–share

What does

`"Vassar"[2:4]`

give you?

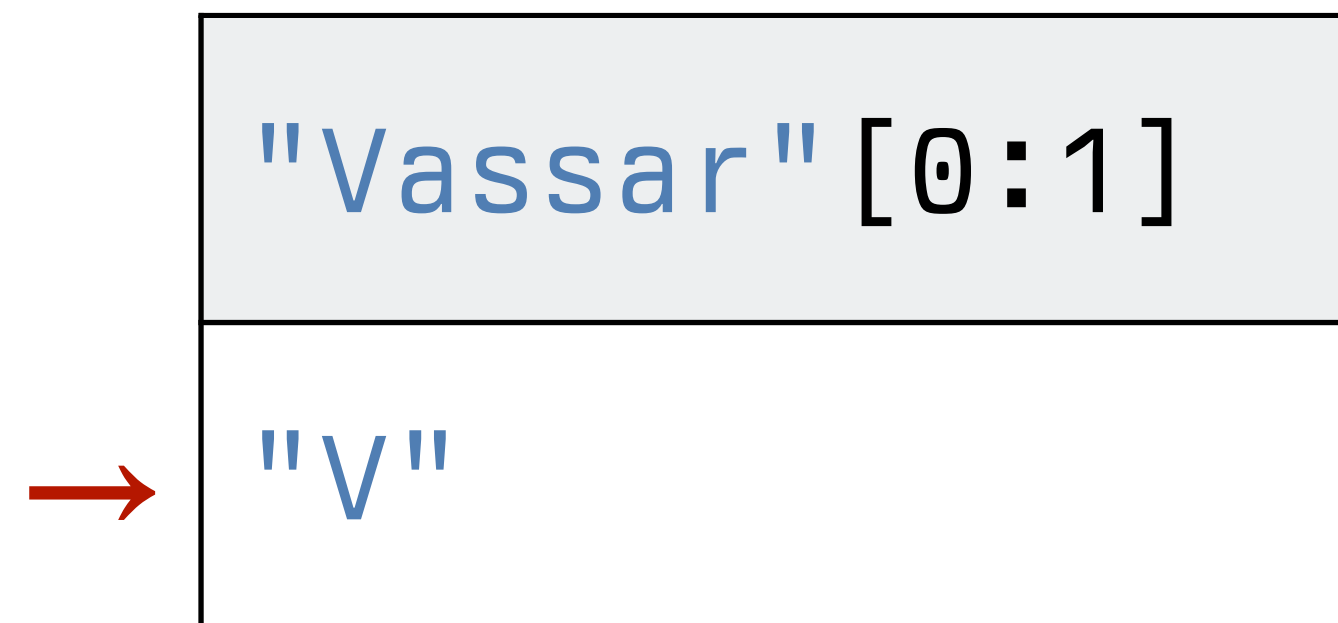
How many characters does

`s[a:b]`

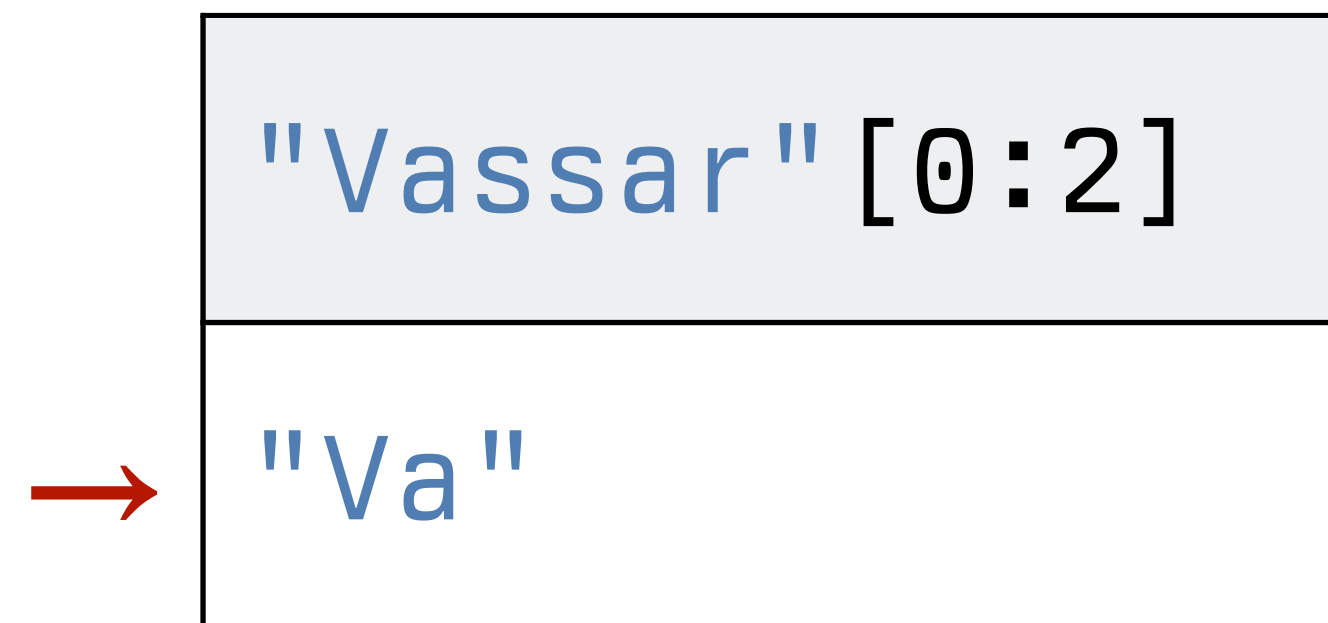
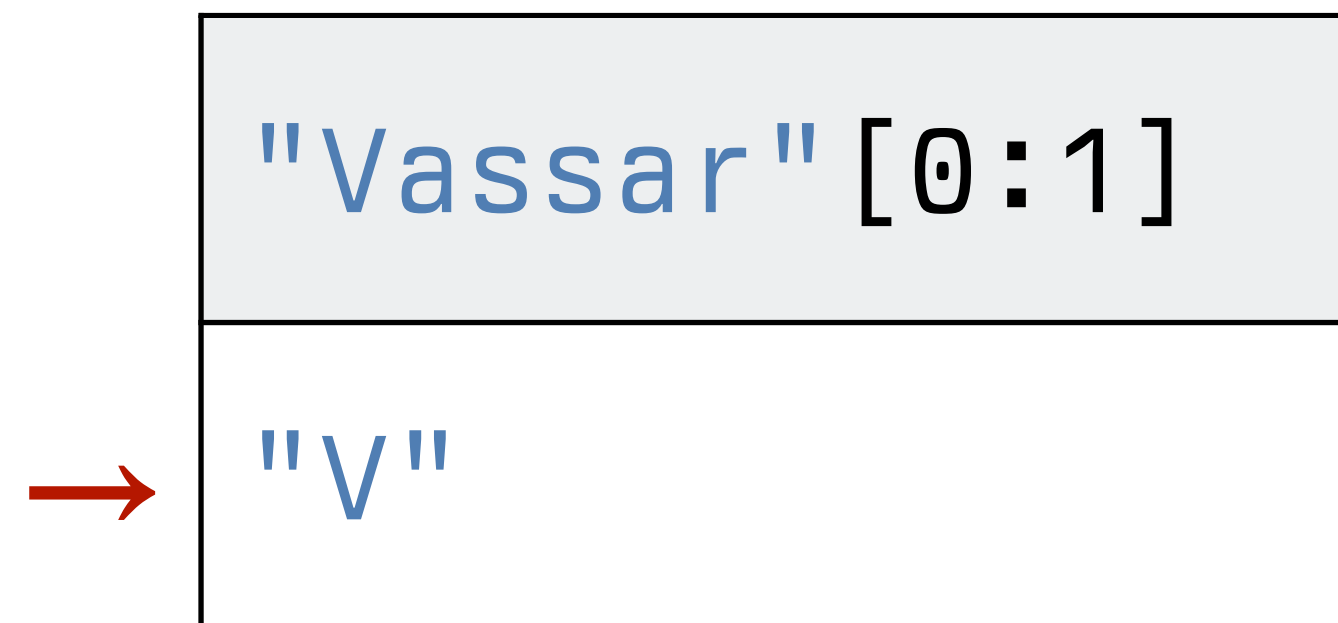
give you in general?

	0	1	2	3	4	5	
"	v	a	s	s	a	r	"

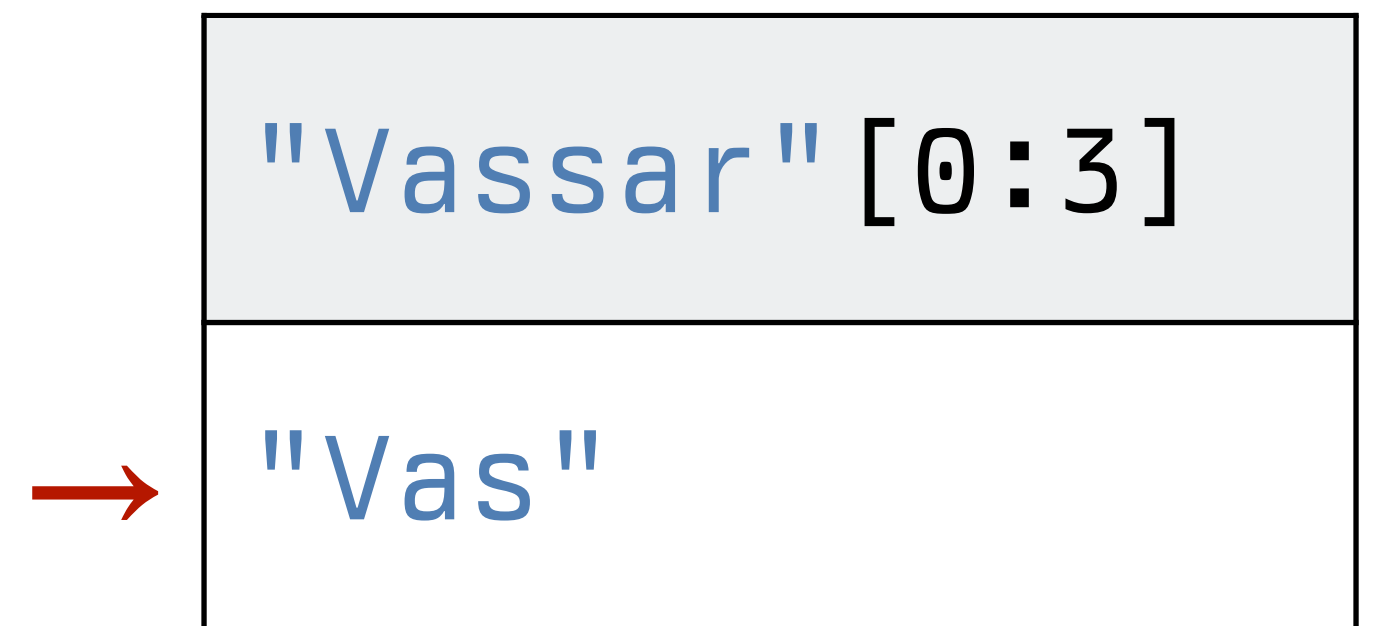
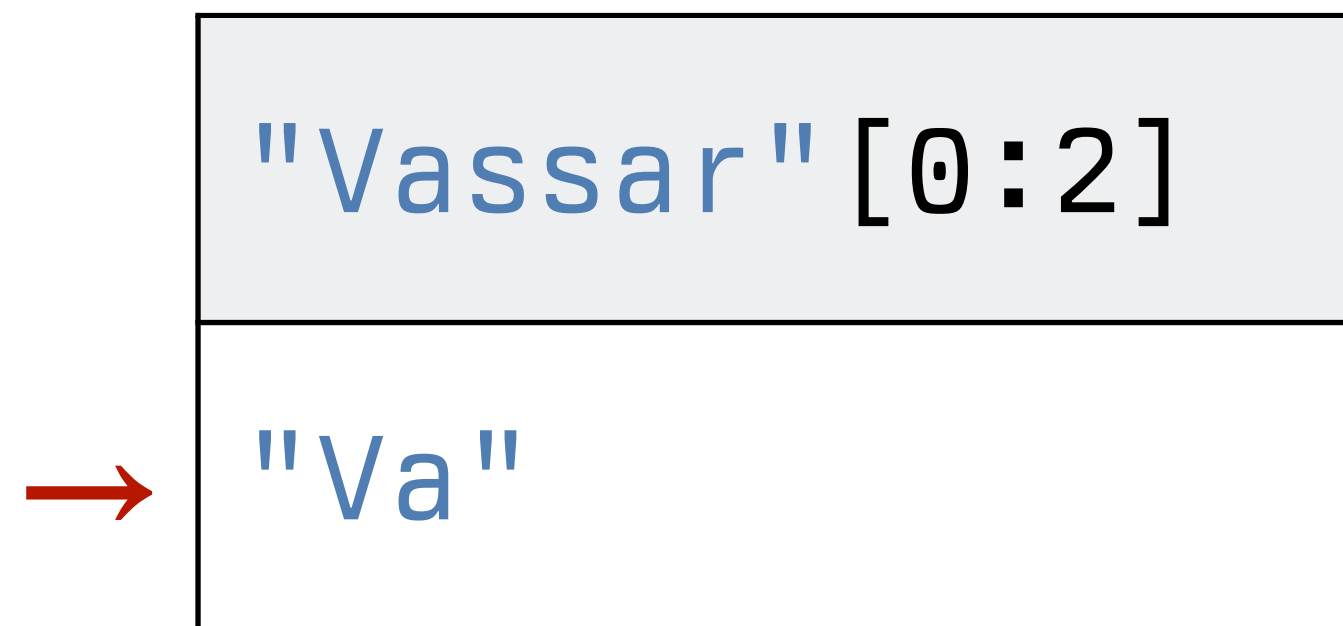
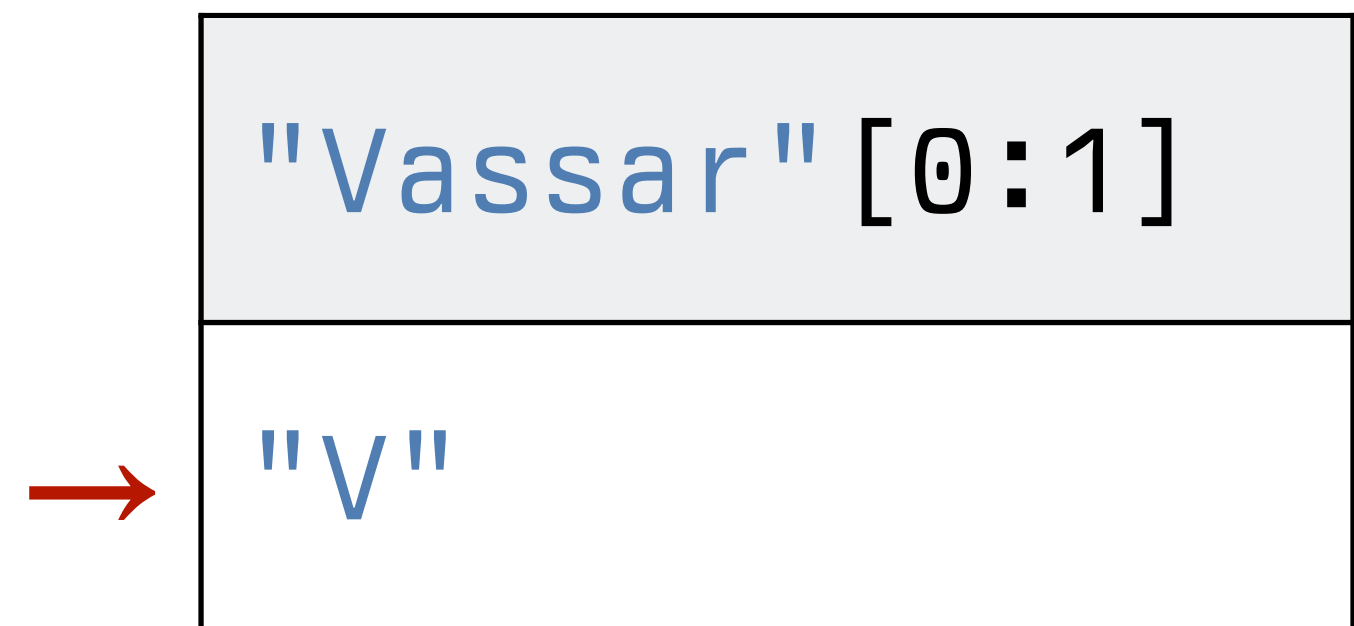
0 1 2 3 4 5
" v a s s a r "



0 1 2 3 4 5
" v a s s a r "



0 1 2 3 4 5
" v a s s a r "



→ Notebook: *Strings*

While a string is a sequence of characters, a *list* is a sequence of any data we want!

To make a list, just enclose the contents in square brackets, e.g.,

```
["Frodo", "Sam", "Merry", "Pippin"]
```

```
[1, 2, 3]
```

Lists can hold a mix of different data types, e.g.,

```
[1.0, -3, True, "Spring"]
```

and even other lists:

```
[[1, 2, 3], [4, 5, 6]]
```

Just like with strings, we can access individual elements or subsequences, we can ask `in` questions, and we can concatenate lists using `+`.

→ Notebook: *Lists*

Processing lists with `map` and `filter`

We've seen that we can call a function on data, e.g.,

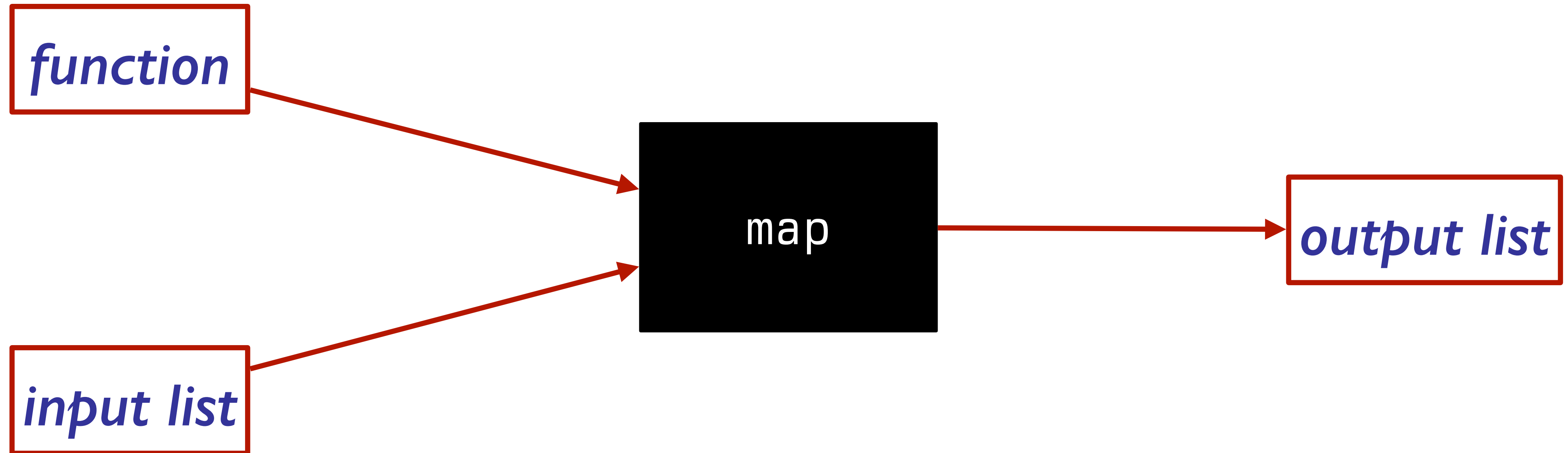
```
max(10, 12)
```

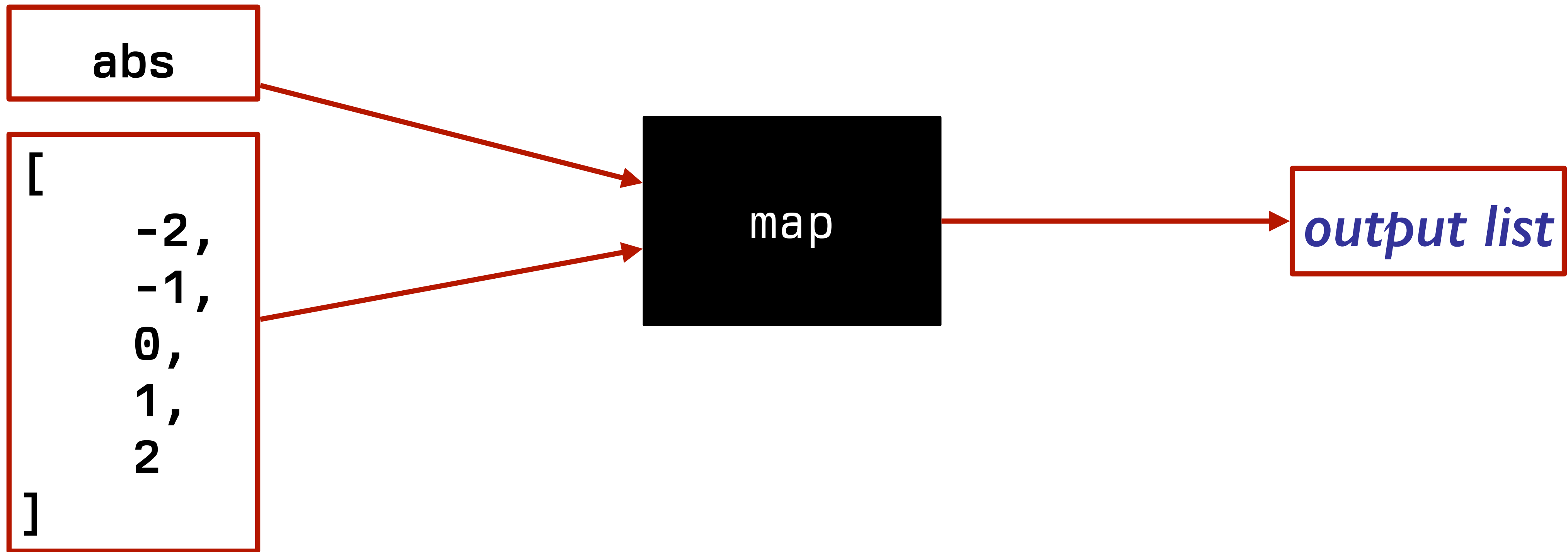
but we can also pass one function as an input to another function to tell it what to do.

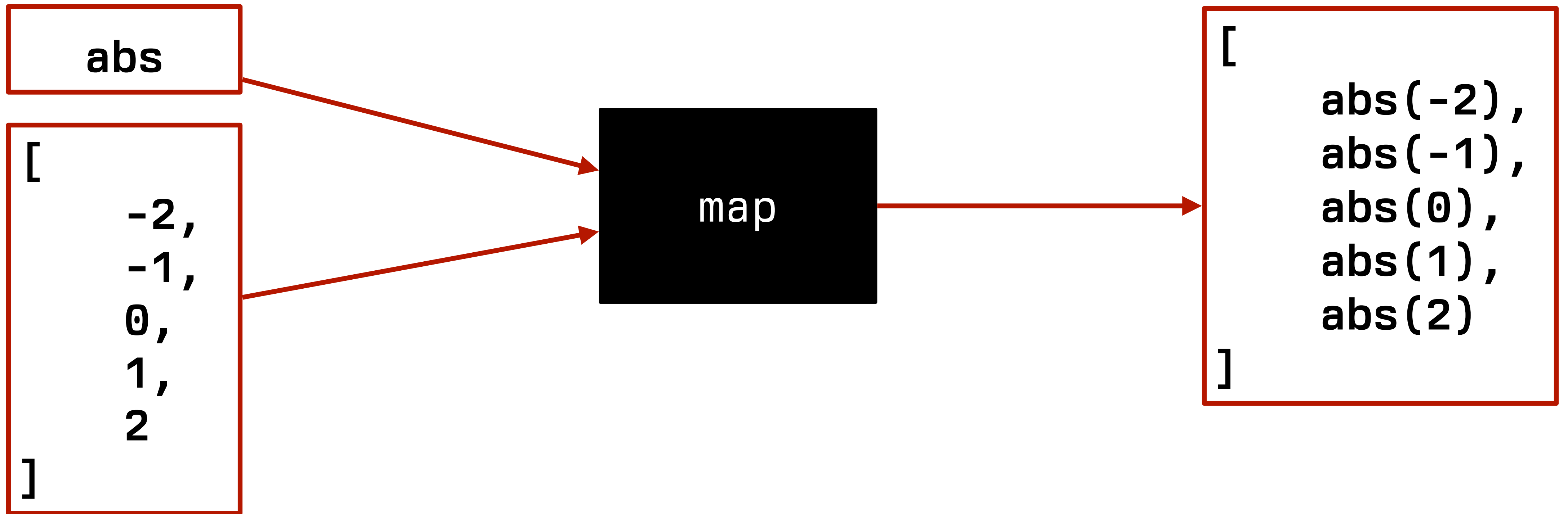
These are called *higher-order functions*, and they'll let us work with the individual elements of a list.

`map(function, input list)`

*When you call map, it returns a new sequence,
where each item in **input list** has been
transformed by running **function** on it.*





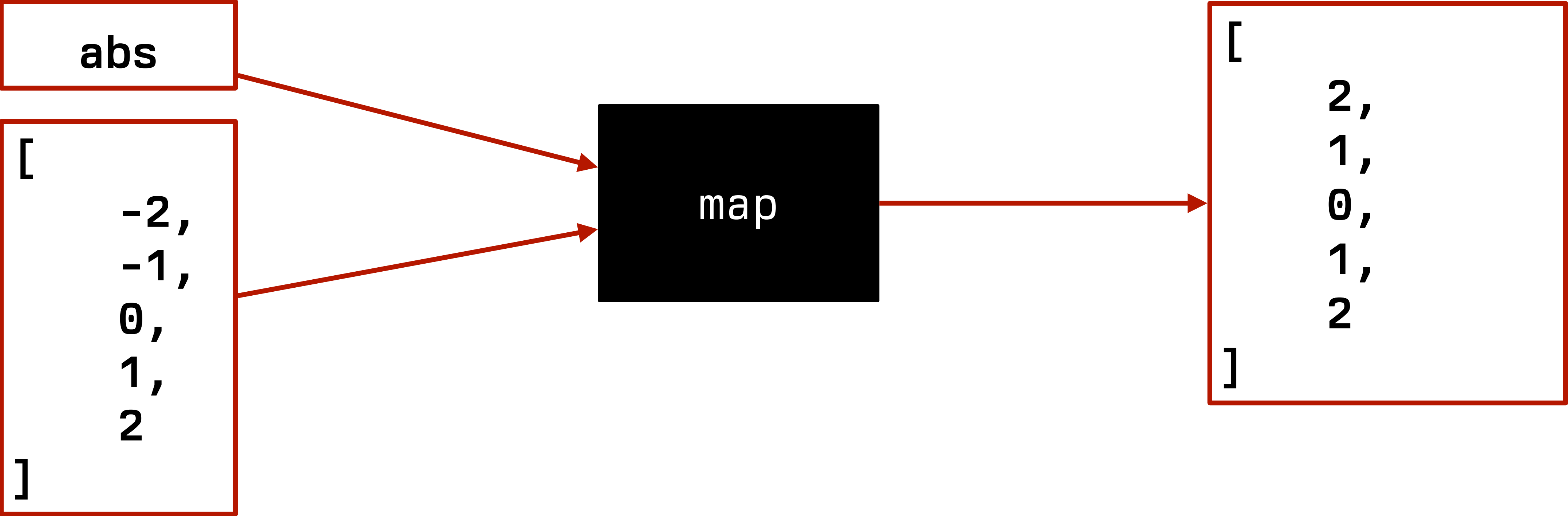


abs

**[
-2,
-1,
0,
1,
2
]**

map

**[
2,
1,
0,
1,
2
]**



Think–pair–share

There's another high-order function called **filter**.

- 1 What do you think it does?
- 2 What would the function you give it as input have to return?

→ Notebook:
Processing lists with **map** *and* **filter**

An *array* is like a list, but designed for efficient computations, especially when they contain numbers.

Specifically, we'll be using the arrays from the popular NumPy library, which we load using an import statement:

```
import numpy as np
```



Think–pair–share

```
np.array([1, 2, 3]) +  
np.array([1, 2, 3])
```



What do you think this does?

For strings and lists, + joined two sequences together, one after another.

For arrays, it's element-wise addition:

```
np.array([1, 2, 3]) +  
np.array([1, 2, 3])
```

→ `array([2, 4, 6])`

```
np.array([-2, 1, 0]) +  
np.array([ 2, -1, 0])
```

→ `array([0, 0, 0])`

We can also easily scale the elements of an array by multiplying them by a single number:

	<code>np.array([1, 2, 3]) * 2</code>
→	<code>array([2, 4, 6])</code>

Or add a single number to each element:

	<code>np.array([1, 2, 3]) + 2</code>
→	<code>array([3, 4, 5])</code>

NumPy provides convenient built-in functions, e.g.,

```
np.mean(np.array([1, 2, 3]))
```

→ 2.0

NumPy provides convenient built-in functions, e.g.,

```
np.mean(np.array([1, 2, 3]))
```

→

```
np.float64(2.0)
```

*Ok – actually it will look like this,
using NumPy's own version of a float*

→ Notebook: *Arrays*

Next week, we'll see how we can build on arrays to work with tables of data!

