CMPU 100 · Programming with Data

# Data Sequences

Class 5

# Tomorrow:

Assignment 1 due at 11:59 p.m.

Assignment 2 out 5 p.m.

# Where are we?

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

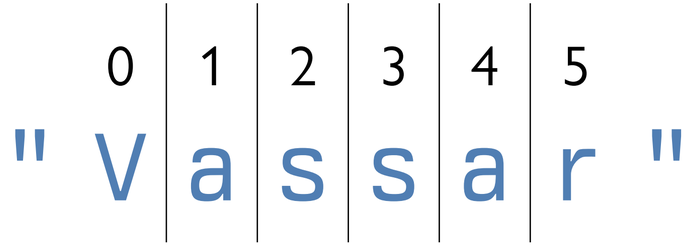
| | |
|---|---|
| *Integers* | `42`, `-3`, `10000` |
| *Floating-point numbers* | `0.0`, `-3.6`, `4.2` |
| *Booleans* | `True`, `False` |
| *Text strings* | `"Alan Turing"`, `"50%"`, `"$3.50"` |

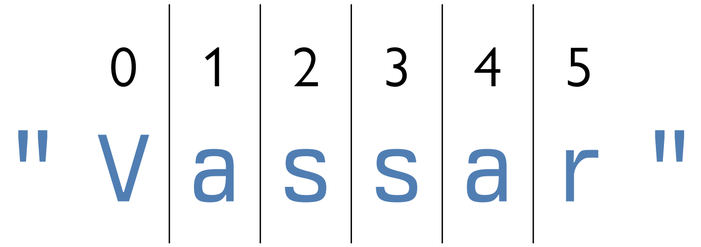
# Notebook: *Where are we?*

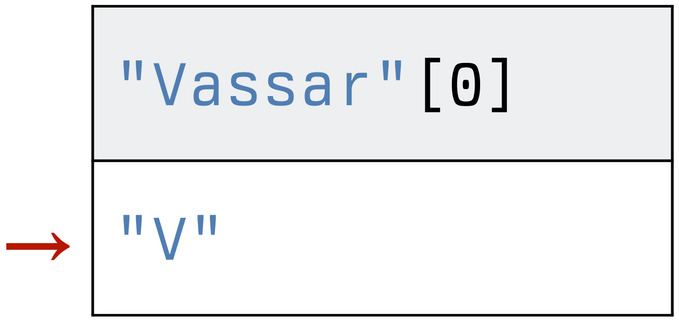
# Strings

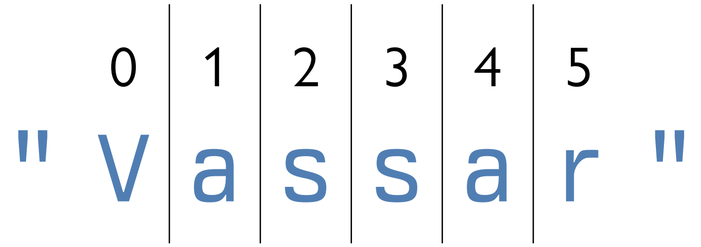
A text string is a *sequence* of *characters* – letters, numbers, punctuation.

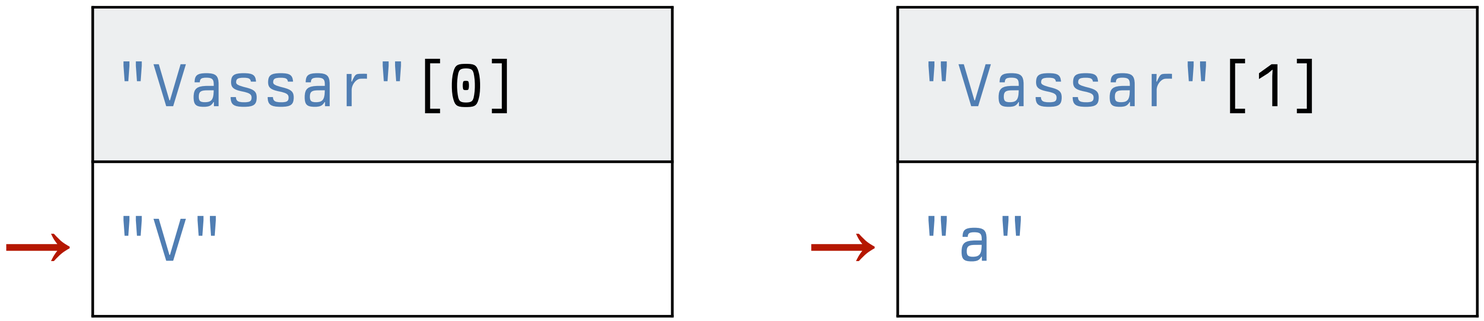
We can access these characters by their position in the sequence.

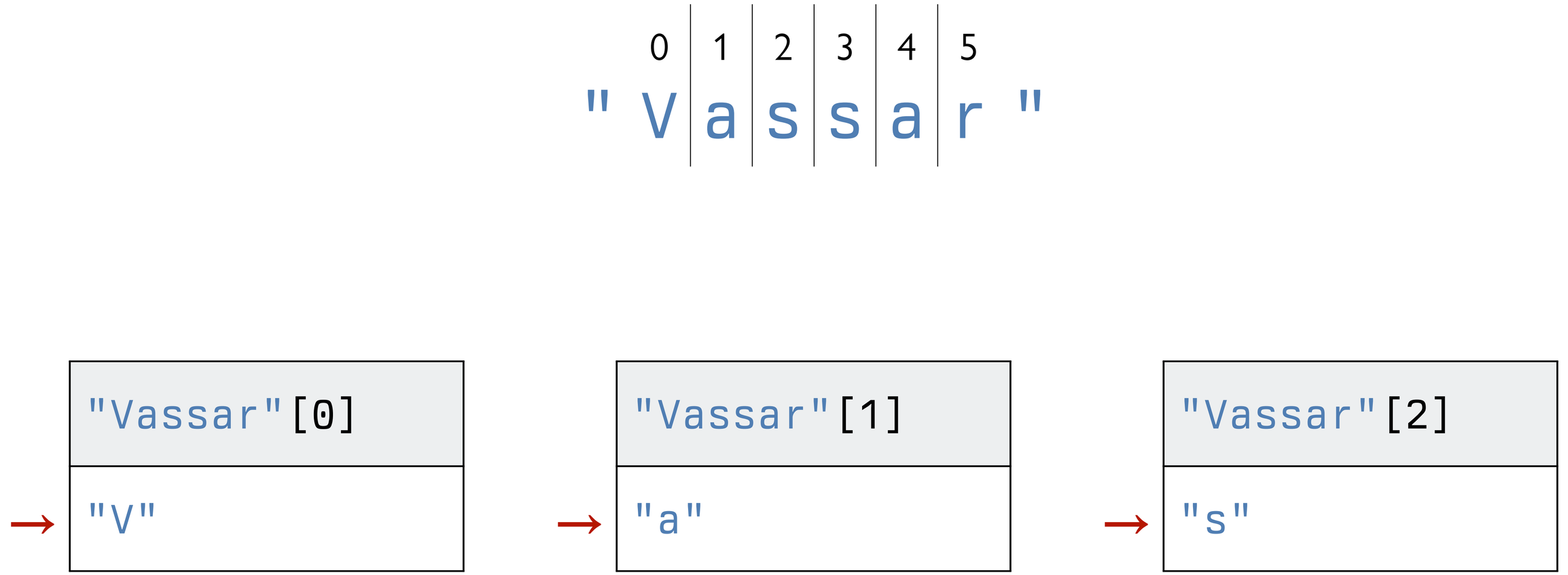

0 1 2 3 4 5
" V a s s a r "

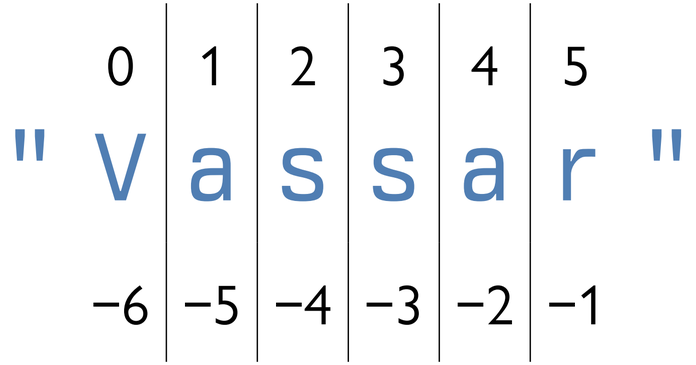

0 1 2 3 4 5
" V a s s a r "


"Vassar"[0]
"V"

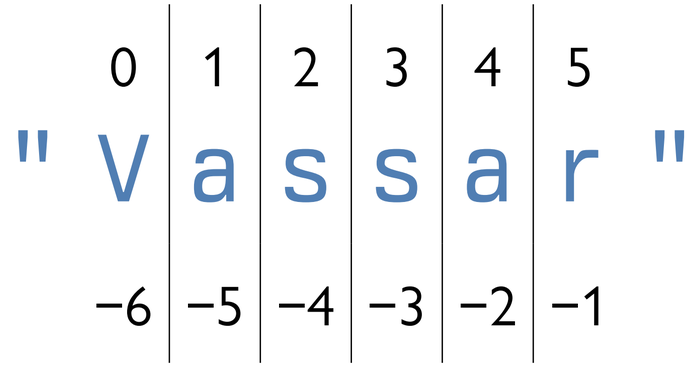

0 1 2 3 4 5
" V a s s a r "


"Vassar"[0]
"V"
"Vassar"[1]
"a"


0 1 2 3 4 5
" V a s s a r "
"Vassar"[0]
"V"
"Vassar"[1]
"a"
"Vassar"[2]
"s"

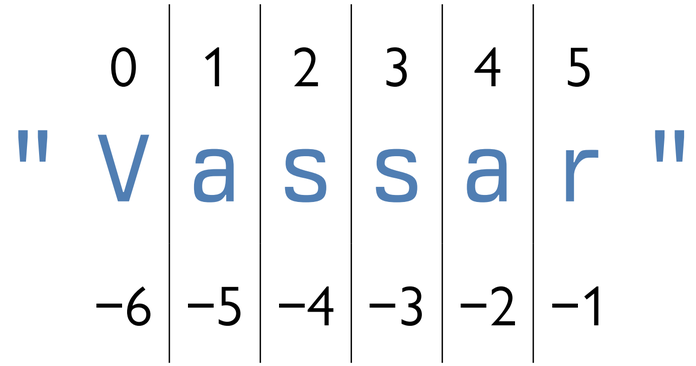



*We can use negative numbers to index from the end of the string.*




| `"Vassar"[-1]` |
| --- |
| → `"r"` |

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



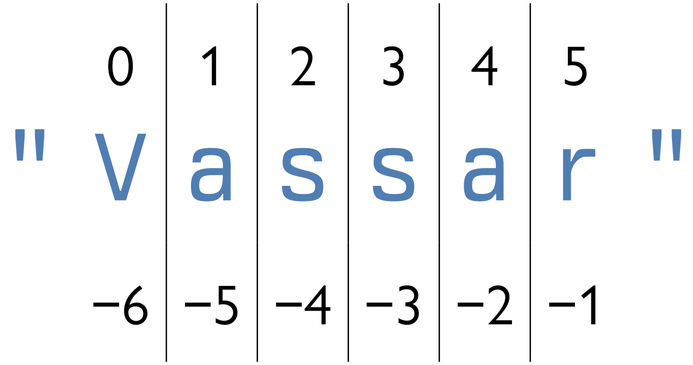

```
"Vassar"[-1]
```

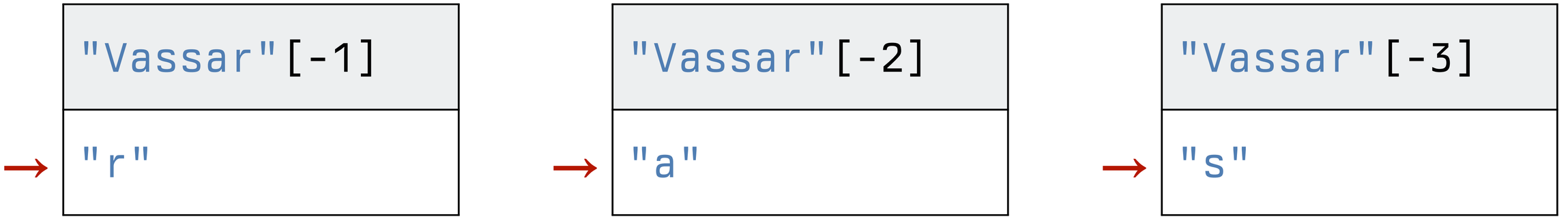
→ `"r"`

```
"Vassar"[-2]
```

→ `"a"`

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

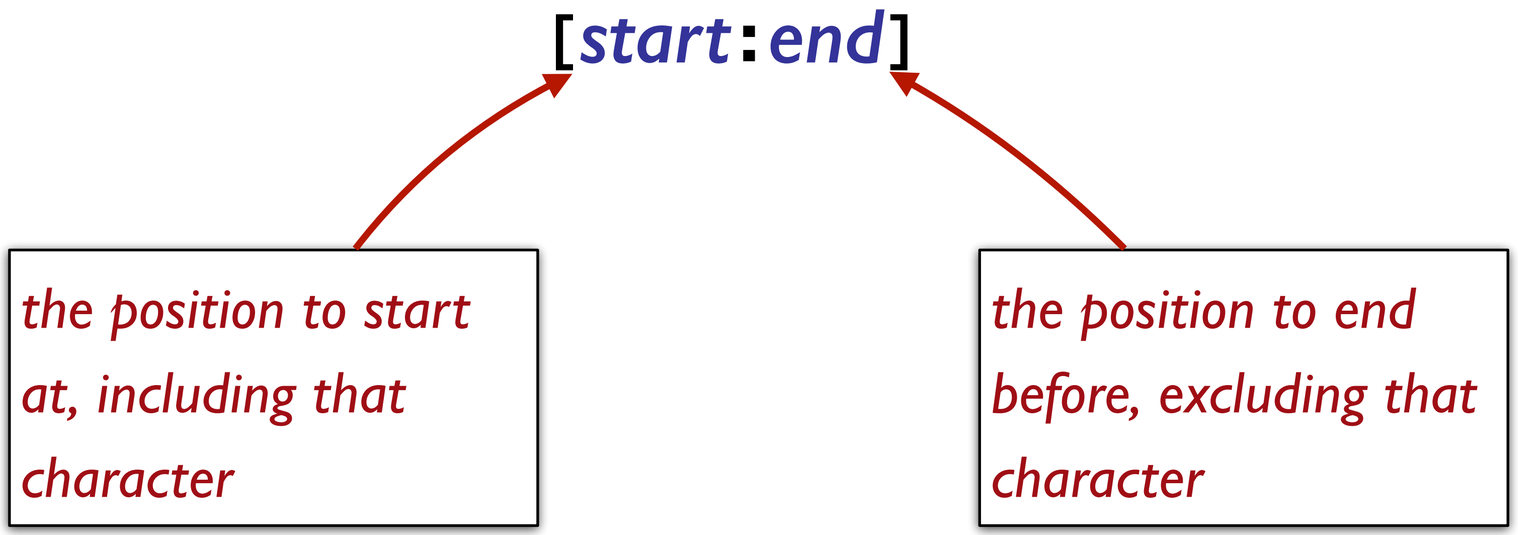


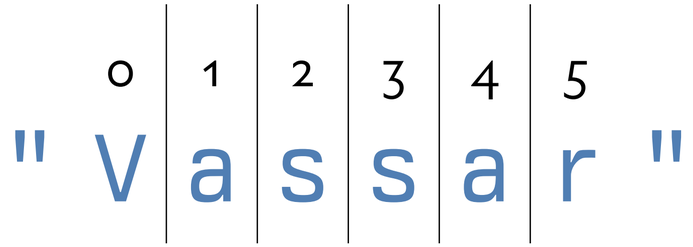

*We can use negative numbers to index from the end of the string.*

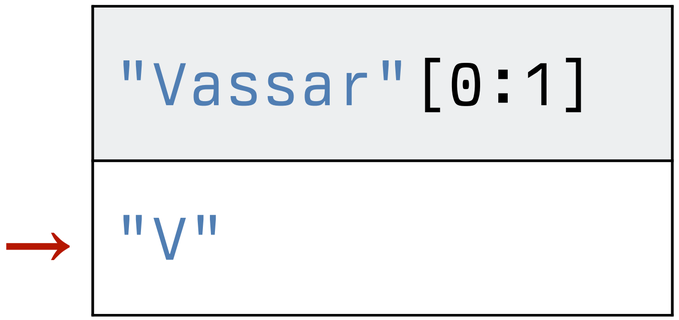
We can also *slice* a sequence to access a subsequence.

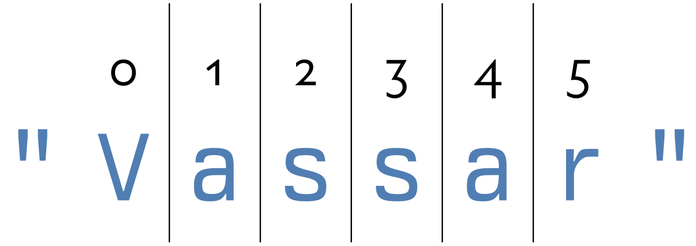
Instead of a single index, we specify

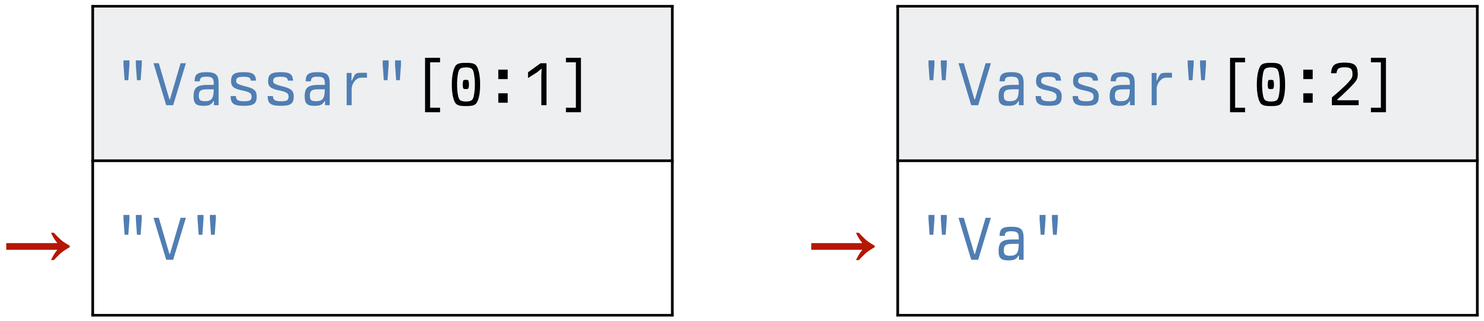


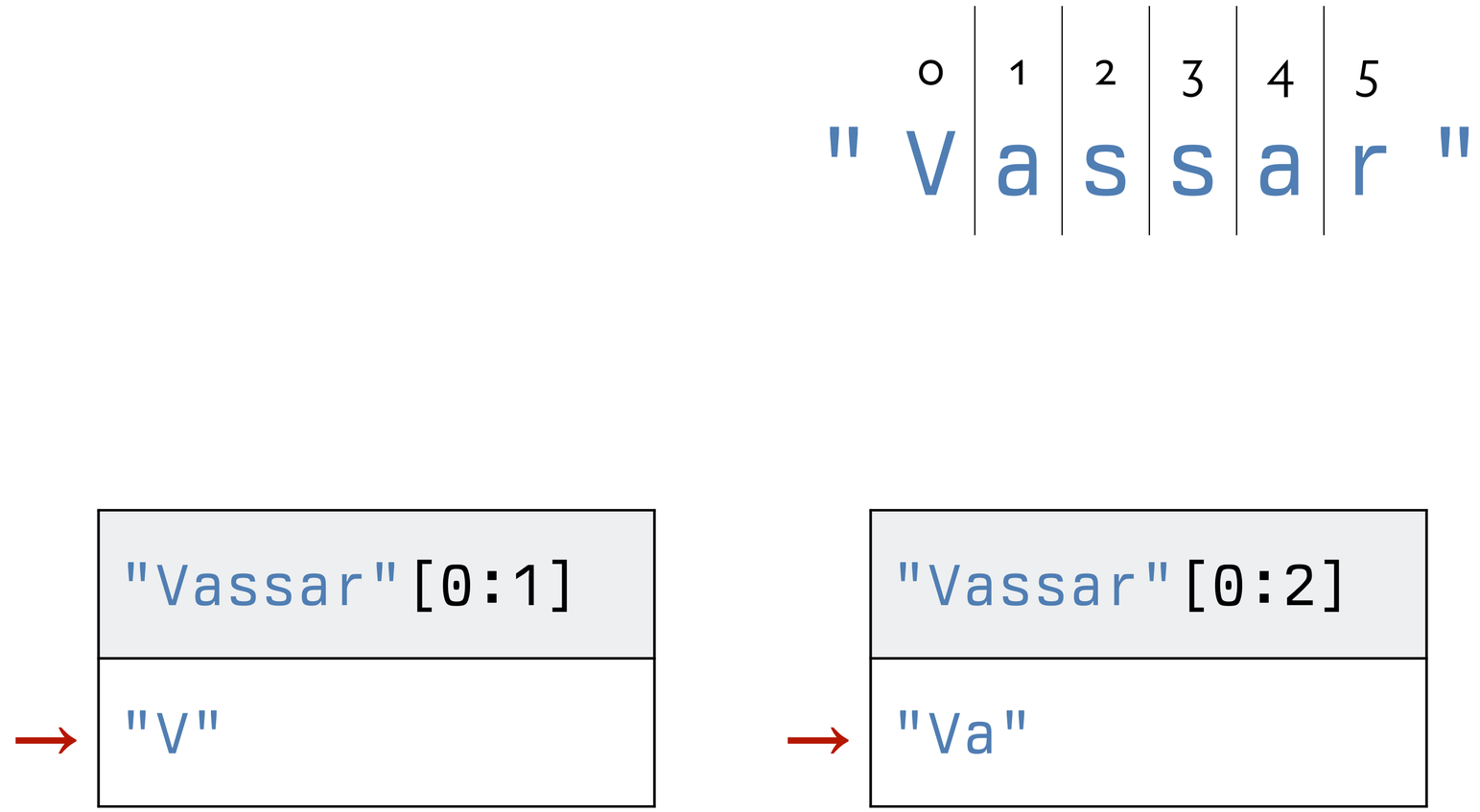
| | 0 | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|---|
| " | V | a | s | s | a | r | " |


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


"Vassar"[0:1]
"V"


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


"Vassar"[0:1]
"V"
"Vassar"[0:2]
"Va"


0 1 2 3 4 5
" V a s s a r "
"Vassar"[0:1]
"V"
"Vassar"[0:2]
"Va"
"Vassar"[0:3]
"Vas"

# Notebook: *Strings*

# Lists

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.,

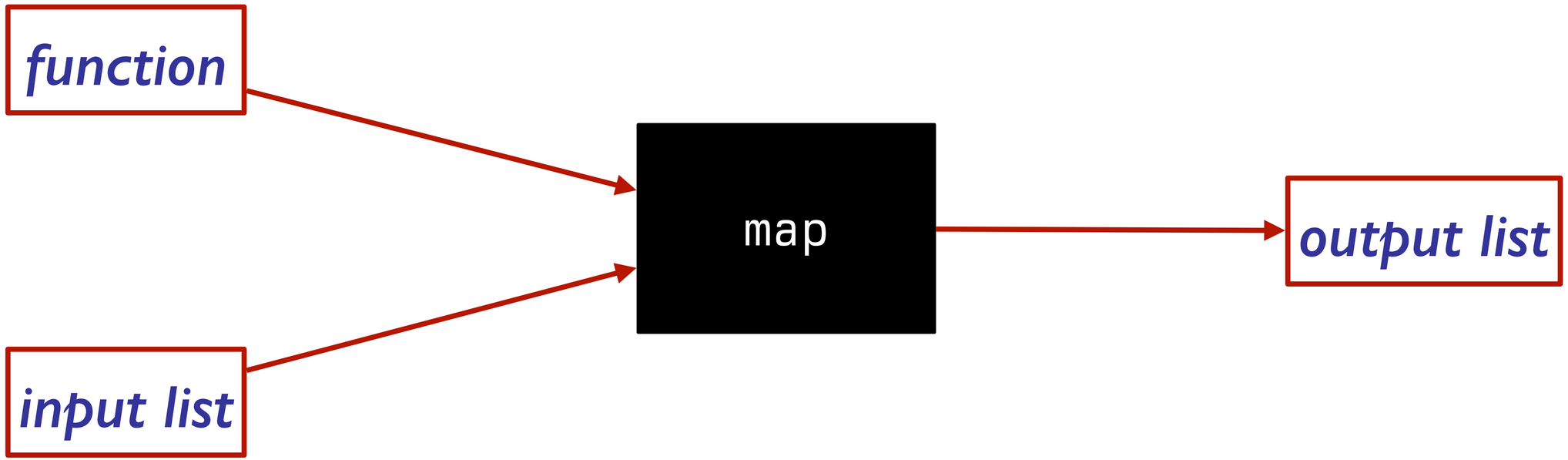
```
max(10, 12)
```

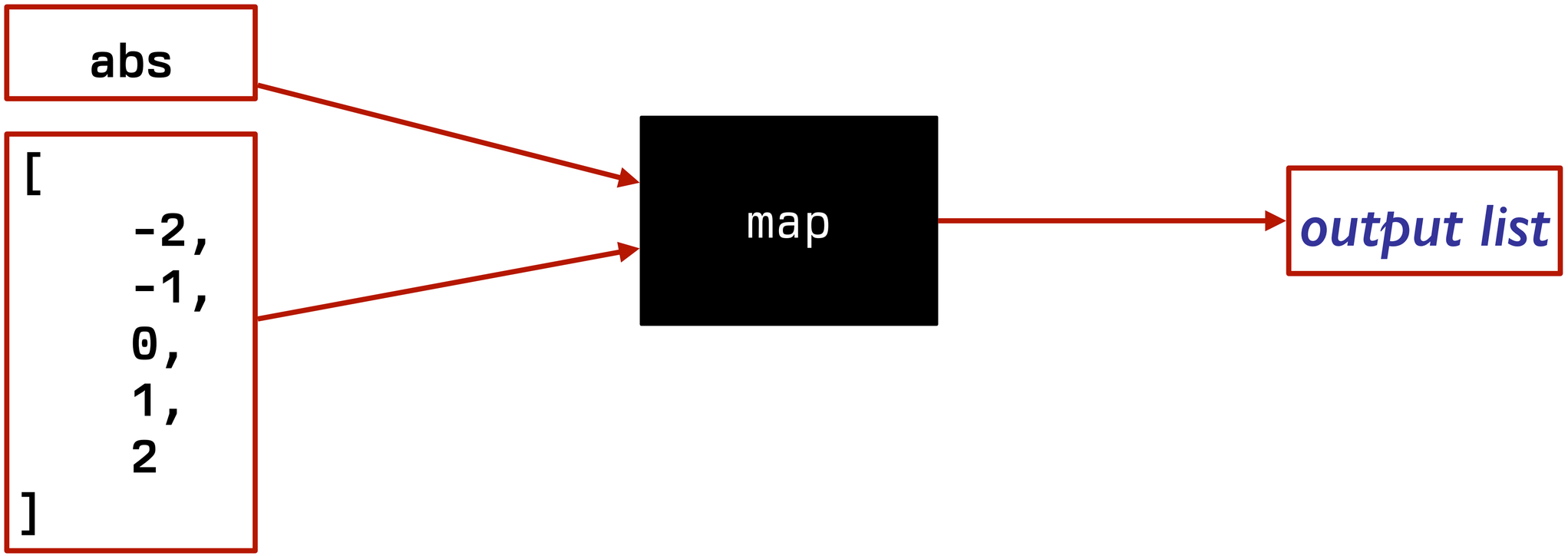
but we can also pass one function as an input to another function to tell it what to do.

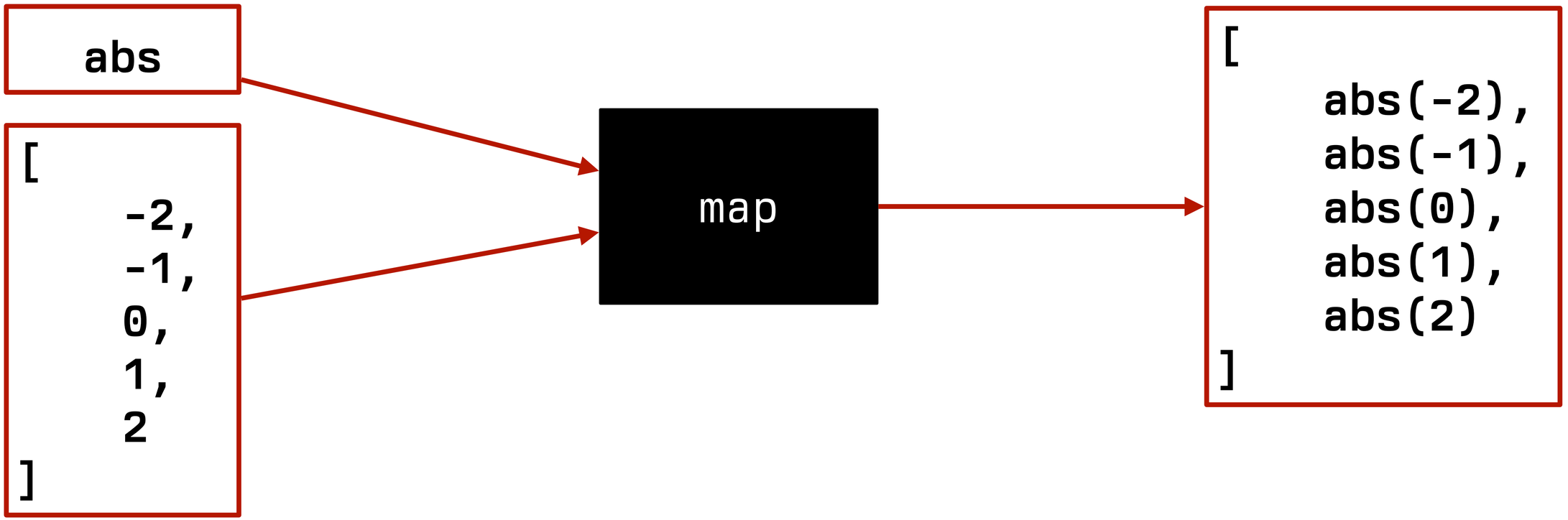
These are called *higher-order functions*, and they'll let us work with the individual elements of a list.

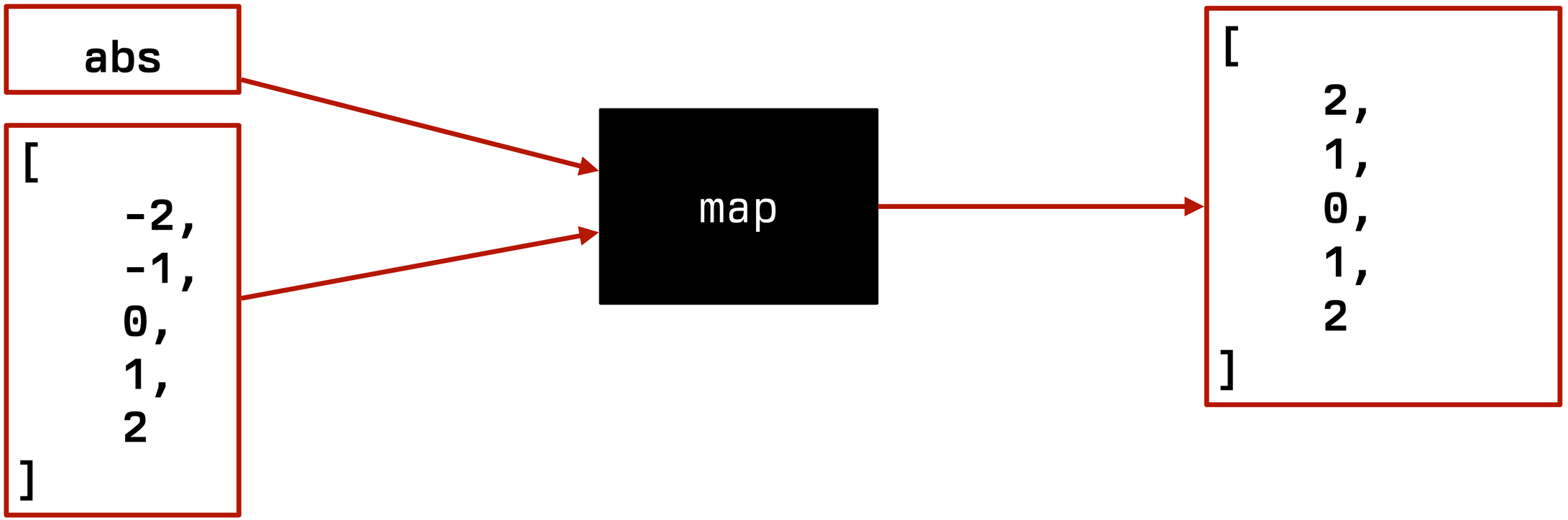
`map(`***function***`,  `***input list***`)`

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


function
input list
map
output list


abs
[
-2,
-1,
0,
1,
2
]
map
output list


abs
[
-2,
-1,
0,
1,
2
]
map
[
abs(-2),
abs(-1),
abs(0),
abs(1),
abs(2)
]


abs
[
-2,
-1,
0,
1,
2
]
map
[
2,
1,
0,
1,
2
]

# Notebook:
# *Processing lists with* `map` *and* `filter`

# Arrays

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
```

We can make an array out of a list by calling the **`np.array`** function on it:

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

→

```
array([1, 2, 3])
```

Values in an array must all be of the same data type, and Python will attempt to convert (cast) them as appropriate:

```
np.array([5, -1, 0.3, 5])
```

→

```
array([5.0, -1.0, 0.3, 5.0])
```

```
np.array([4, -4.5, "not a number"])
```

→

```
array(["4", "-4.5", "not a number"])
```

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:

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

→

```
array([2, 4, 6])
```

Or add a single number to each element:

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

→

```
array([3, 4, 5])
```

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

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

→ `2.0`

# Example

We can measure how much the radius of a tree grows in a given year by measuring the width of tree ring for that year

Suppose we have the ring widths (in mm) for a tree for five years:

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

What was the total growth?

```
np.sum(ring_widths)
```

What was the average growth?

```
np.mean(ring_widths)
```

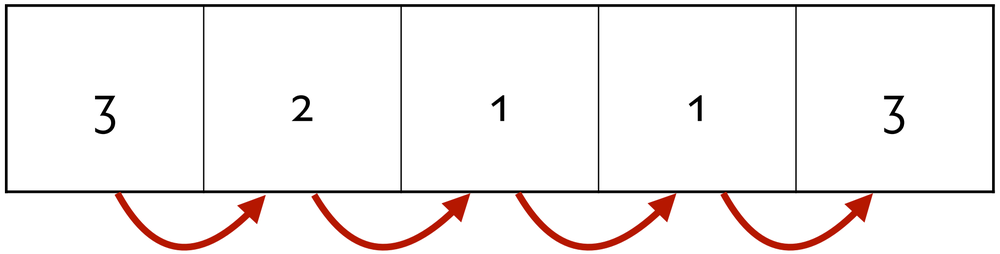
And **`np.diff`** produces an array of the differences between adjacent elements in the input, letting us see how the ring widths changed rom year to year

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

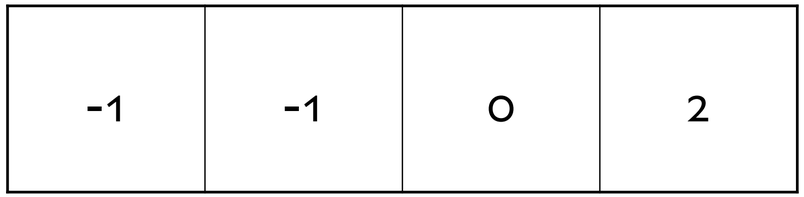
| | | | | |
|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 3 |

And **`np.diff`** produces an array of the differences between adjacent elements in the input, letting us see how the ring widths changed rom year to year

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




```
np.diff(ring_widths)
```



# Notebook: *Arrays*

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