Warmup (L17)

- Plot `sin(x)` from 0 to 10 with three degrees of precision. That is:
  - One line plotting `sin(x)` but with very few calculated points
  - One line plotting `sin(x)` with more calculated points
  - One line plotting `sin(x)` with a lot of calculated points

Boolean Vector Logic

CS114 M8

Administrata note

- This topic is not covered in *Think Python* (the first textbook on the course web page) but is covered in *Python for Scientists* (the second)
- It's new this term
- The module numbers will diverge from *Think Python* from here on

Arrays of booleans

CS114 M8

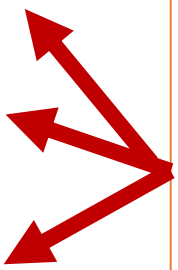
Vectorization

- `np.vectorize` makes a function act over arrays instead of individual values
- All `np.vectorize` is really doing is looping over the array and calling the original function
  - This is a slight oversimplification—you can, e.g., vectorize multi-argument functions—but it's the basic idea

Vectorization

- The thing that was hard was making functions with **`if`**s, because Python will only run one branch

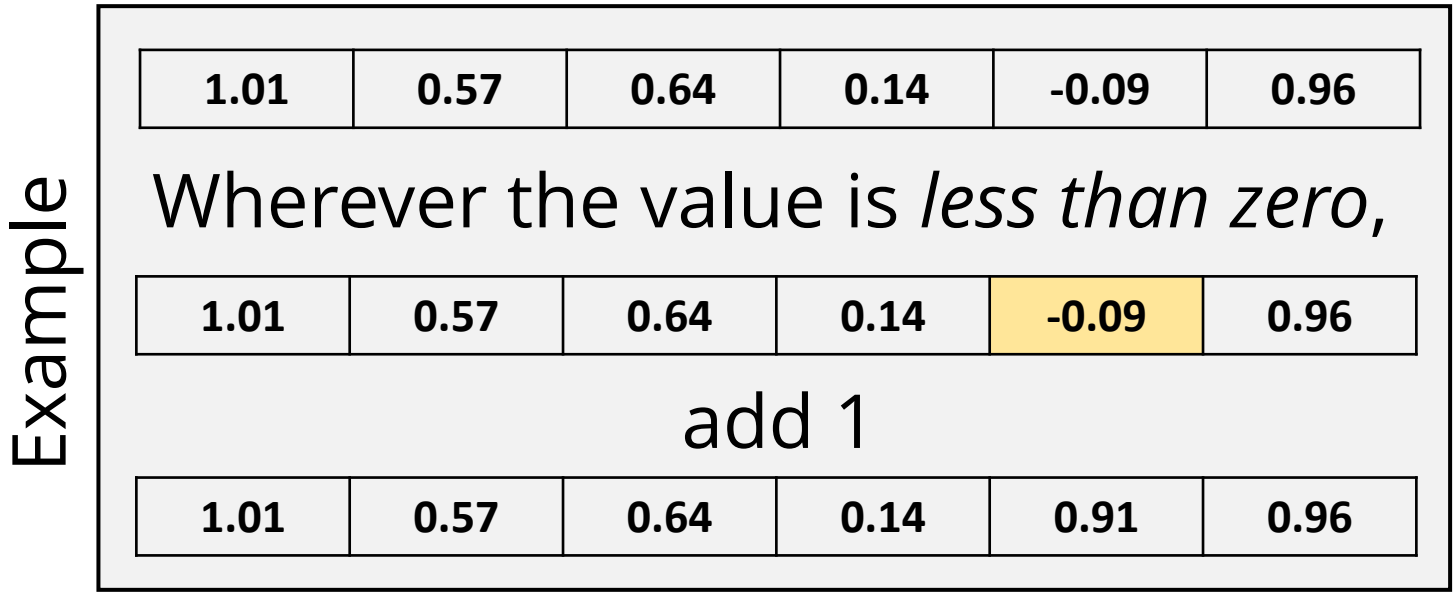
```
def stepFunction(x: float) -> float:
    if x < 0:
        return -1
    elif x < 2:
        return 0
    else:
        return 1
```



Python can't choose to run the "**`if`**" branch for certain parts of the array and the "**`else`**" branch for others

Masking

- There's another way to think about vectorization: *masking*
- The idea of masking: wherever (something) is true, do (something)



Etymology

- The idea is that you're "masking out" all the other values so they're not touched
- More like masking tape (tape you put on a wall where you don't want to paint) than a mask you wear

Masking in NumPy

- NumPy uses arrays of booleans to do masking
- This makes for some new and surprising behavior

Example: Discard out of range

- Basic case of masking: rather than doing some operation with masked values, we just want to get rid of them
- We have an array of numbers that are supposed to be between 0 and 1, but due to sampling error, some are outside that range
- Discard the ones that are outside that range

Example: Discard out of range

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

mask = np.logical_and(a >= 0, a <= 1)
```

Unfortunately, **`and`** doesn't work on arrays, but NumPy provides a vectorized **`and`**: `np.logical_and`

```
d = a[mask]
```

Example: Discard out of range

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

mask = (a >= 0) & (a <= 1)
```

This is sufficiently common that there's a shorter operator, &

```
d = a[mask]
```

Example: Discard out of range

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

mask = (a >= 0) & (a <= 1)
```

`mask` is just an array of booleans. We could've achieved the same with `np.array([False, True, …])`

```
d = a[mask]
```

Example: Discard out of range

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

mask = (a >= 0) & (a <= 1)



d = a[mask]
```

This is similar to slicing, but slicing doesn't let you use an array of booleans! It's masking! Copy out only the parts of the array where the mask says "`True`"

Note

- The shorthand for `np.logical_and` is `&`
- The shorthand for `np.logical_or` is `|` (the pipe symbol)

In-lecture quiz (L17)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: What is x after this code?

```
a = np.array([1.01, -0.09, 0.96, 0.0])
x = a < 0
```

A. `False`

B. `True`

C. `np.array([1])`

D. `np.array([False, True, False, False])`

In-lecture quiz (L17)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: I tried to mask out-of-range values, but it didn't work! Why?

```
d = a[a >= 0]
d = d[a <= 1] # This didn't do what I want
```

A. The mask is for the wrong array
B. `d` is not an array, so can't be masked
C. There is no `mask` variable
D. You can't overwrite `d`

Left-slicing

CS114 (M8)

Left-slicing

- We've shown slicing to copy out part of a list (or array, or other sequence)
- With mutable sequences (lists, arrays), you can slice on the left of an assignment to mutate whole subparts
  - The idea is the same as indexing on the left to change a given slot
- We never did this with lists because it's super-awkward if you can't do element-wise operations

Terminology

This is usually called “slice assignment”, but “left-slicing” or the related “LHS-slicing” are also used

Example: Triangle building

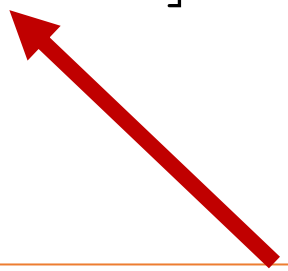
- We have an array of numbers, and want to replace all the numbers with a triangle pattern
  - E.g., [0, 1, 2, 3, 2, 1, 0] or [0, 1, 2, 2, 1, 0]
- This isn't especially difficult to do with loops, so let's do that version first

Example: Triangle building

```
a = np.zeros(41)
midpoint = len(a)//2 + len(a)%2
for idx in range(0, midpoint):
    a[idx] = idx
    a[-1-idx] = idx
```

Slice version

```
a = np.zeros(41)
midpoint = len(a)//2 + len(a)%2
for idx in range(0, midpoint):
    a[idx:-idx] = idx
```



This is `-idx` instead of `-1-idx` because the upperbound is exclusive

Slice version

```
a = np.zeros(41)
midpoint = len(a)//2 + len(a)%2
for idx in range(0, midpoint):
    a[idx:-idx] = idx
```

We are “oversetting” here, since we’re setting the subsequence, but only the first and last will be kept.

Triangle building

- The slice version wasn’t easier, and set more than it needed to
- So let’s try something a bit trickier

Example: Pyramid building

- Now, let's bump it up. You have a 2D array of numbers which you want to replace with a pyramid of numbers.
- Now this is much more annoying to do with loops: you need to loop over both *x* and *y*, and there's a whole square you need to fill in!
- But with slicing on the left-hand side...

Example: Pyramid building

```
a = np.zeros((7, 7))
midpoint = len(a)//2 + len(a)%2
for idx in range(0, midpoint):
    a[idx:-idx, idx:-idx] = idx
```

Aside: Left-slicing lists

- Slice-assignment/left-slicing on lists works too… sort of
- Because nothing on lists is element-wise, you need to assign a whole list

```
lst = [1, 2, 3, 4, 5]
lst[1:] = 1 # CRASH
lst[1:] = [1, 1, 1, 1] # OK
```

I thought we were talking about masking?

CS114 (M8)

Left-masking

- I said masks act sort of like slicing
- Like slices, they can also go on the left!
- So, you can use a mask to choose what parts of an array you change

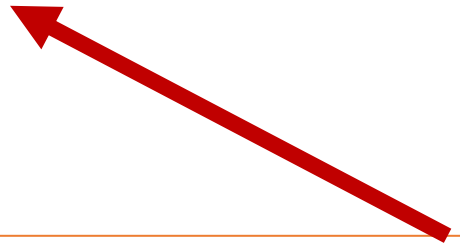
Original example: Add 1 to negatives

- Our original example was "Wherever the value is less than zero, add one"
- Let's do that!

Original example: Add 1 to negatives

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

a[a < 0] += 1
```



A *lot* is happening in this innocuous little line!
Let's break it down.

Step one: Get the mask

- `a < 0` gets the mask
- It becomes

```
np.array([False, False, False, False, True, False])
```

Step two: Mask the array

- When you index by a mask, you get only the True parts
  (in this case, `np.array([-0.09])`)
- `a[a < 0]` should be read like "`a` where the value in `a` is less than zero"
- `a` is used in both the array to mask and the mask, but it's taking different roles
  - You can use any mask that's the right size

Step three: Mask assignment

- A left-slice (slice assignment) lets you update part of an array
- A mask behaves like a slice…
- so, we can update a masked part of an array!

```
a[a < 0] += 1
```

- We’re back to our original reading: “Wherever a is less than zero, add one.”

Multiple changes

- Let's say for negative numbers we wanted to add one and multiply by two
- This *won't work as expected*:

```
a[a < 0] += 1
a[a < 0] *= 2
```

- Why not? We computed the mask twice, and which numbers are negative changed!

Multiple changes

- Solution: save your masks!
- Remember, it's not the "`a`" in the index that made this work, it's the mask

```
mask = a < 0
a[mask] += 1
a[mask] *= 2
```

`where`

CS114 (M7)

`if` but no **`else`**

- In effect, mask assignment is "**`if`**", not "**`else`**"
- You can get "**`else`**" by "**`not`**"-ing the mask (wherever this mask is not true):

```
mask = a < 0
a[mask] += 1
a[np.logical_not(mask)] *= 2
```

`if` but no **`else`**

- In effect, mask assignment is "**`if`**", not "**`else`**"
- You can get "**`else`**" by "**`not`**"-ing the mask (wherever this mask is not true):

```
mask = a < 0
a[mask] += 1
a[~mask] *= 2
```

This is sufficiently common that there's a shorter operator for it

`where`

- Everything I've shown modifies the original array
- This is sometimes what you want, but not always
- Having both an "**`if`**" and an "**`else`**" branch *and* not modifying the array is cumbersome, so NumPy has a function for it

where

```
b = np.where(
    a < 0,  # Condition
    a * -1, # Values to use where true
    a * 2   # Values to use where false
)
```

Clamping

- My original problem was an array of numbers that were supposed to be between 0 and 1, but some were out of bounds due to sampling issues
- Let’s clamp them to the range 0 to 1
  - (“Clamping” in this context means restricting, so values less than 0 become 0, and values greater than 1 become 1)

Clamping step one: < 0

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

clamped = np.where(
    a < 0, # if val < 0:
    0,     # use 0
    a      # else: original value
)
```

Clamping step two: > 1

```
a = np.array([
    1.01, 0.57, 0.64, 0.14, -0.09, 0.96
])

clamped = np.where(
    a < 0, # if val < 0:
    0,     # use 0
    np.where(
        a > 1, # elif val > 1:
        1,     # use 1
        a      # else: original value
    )
)
```