Warmup (L8)

Write a function that computes pi using the Leibniz formula, taking a callback to decide when to stop. The callback should be a function that takes a float (the current approximation) and returns True to indicate “stop now”, False otherwise.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

M4: Strings and Lists

L8

Sequences

CS114 L8 (M4)

Sequences

- We discussed ranges for **`for`** loops
- I said it's a "grouping", but it's more specific: a *sequence*
- A sequence is a grouping of items with some order
  - `range(1, 10)`: 1, 2, 3, 4, 5, 6, 7, 8, 9
  - prime numbers: 2, 3, 5, 7, 11, ...

We've already seen a sequence!

- Strings are just sequences of characters! ("Character" is a general term for a glyph used in language)
  - You could say the characters have been *strung* together. Yup, that's the etymology.

```
for c in "Hello, world!":
    print(c)
```

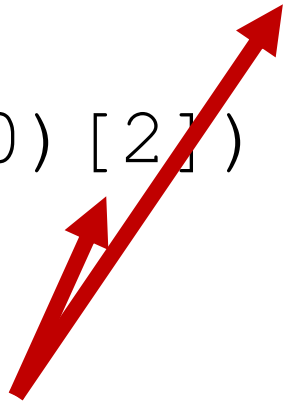
Manipulating sequences

- As we've seen, we can loop over sequences
- We can also get elements from sequences by *indexing*

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```

Manipulating sequences

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```



(some sequence)`[`*index*`]` gets an element from a sequence

Manipulating sequences

```
print("Hello, world!"[1])
e
print(range(1, 10)[2])
3
```

Surprised by the results?
Sequences in Python (and most programming languages) are *0-indexed*. That means that the index for the first element is 0, not 1.

Aside on 0-indexing

- A common error is the *off-by-one* error, which is exactly what it sounds like
- Some people think 0-indexing is the cause of off-by-one errors
- When Julius Caesar was assassinated, Julian leap years were done wrongly for 50 years due to an off-by-one error. Humans just suck at counting.

Sequence length

- Get the length of any sequence with `len`
- We can use ranges to loop over elements in a different way:

```
s = "Hello, world!"
for i in range(len(s)):
    print(s[i])
```

Modifying sequences

- You can *access* the individual characters in a string, but you can't *change* them

```
x = "Hello, world!"
x[1] = "u" # ERROR!
```

- strings are *immutable* (un-changeable)
- So are ranges

Lists

CS114 L8 (M4)

Lists

- Lists are sequences that can contain anything
- Written with square brackets:

```
[2, 4, 6, 0, 1]
```

- Indexed like any sequence

```
x = [2, 4, 6, 0, 1]
x[2] == 6
```

Typing lists

- The type for a list is `list`
- But most of the time, you care what it's a list *of!*
- You can specify what's in the list with, e.g., `list[int]`
- It is always the right style to type as specifically as possible. Don't use `list` when you know what's in it!

List example

- Let's write a function to average a list of numbers

```
def averageOf(l: list[float]) -> float:
    sum = 0.0
    for val in l:
        sum = sum + val
    return sum / len(l)

averageOf([2, 4, 6, 0, 1]) # 2.6
```

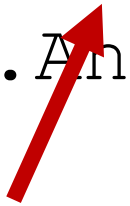
List example

- Let's write a function to check if a value is in a ~~list~~ sequence (any type of sequence!)

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
    for val in haystack:
        if val == needle:
            return True
    return False
```

List example

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
```



The type for a sequence of any sort (string, list, range) is in the typing module.

```
                return True
        return False
```

List example

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
    for val in haystack:
        return False
```

This type means “I don’t care”. In this case, we’re not *doing* anything with the needle, so we don’t actually care what it is.

List example

```
def contains(
    haystack: typing.Sequence,
    needle: typing.Any
) -> bool:
    for val in haystack:
        return False
```

Be wary of this type!
Remember: types are documentation! Don’t just write “any” to make the type checker shut up!

The `in` operator

- We just wrote a `contains` function
- As it turns out, Python has this built in:

```
x = [2, 4, 6, 0, 1]
6 in x # True
"e" in "hello" # True
```

Lists are mutable

- Unlike the other sequences we've seen so far, lists are *mutable* (changeable)

```
x = [2, 4, 6, 0, 1]
print(x) # [2, 4, 6, 0, 1]
x[1] = 8 # change an element just
         # like you'd change a
         # variable
print(x) # [2, 8, 6, 0, 1]
```

Using mutation

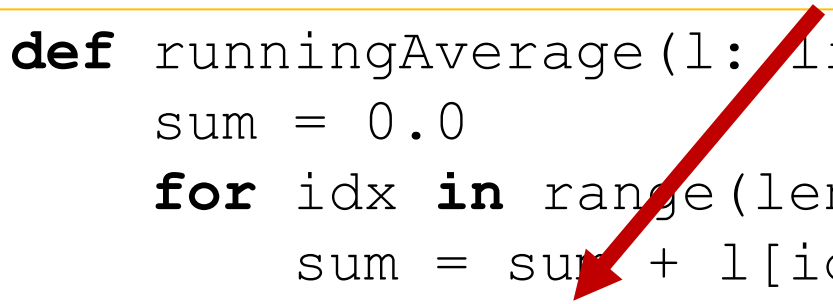
- Let's replace every value in a list with the running average (the average until that point in the list)

```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1) # 0-indexing!
    return sum / len(l)
```

Using mutation

- Let's replace every value in a list with the running average (the average until that point in the list)

Values in the list are replaced (after we used them)



```
def runningAverage(l: list[float]) -> float:
    sum = 0.0
    for idx in range(len(l)):
        sum = sum + l[idx]
        l[idx] = sum / (idx+1) # 0-indexing!
    return sum / len(l)
```

In-lecture quiz (L8)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: How many times does this print “x”?

```
for s in ["Excellent", "text", "box"]:
    for c in s:
        print(c)
```

A. 0 (no times)

B. 1

C. 2

D. 3

E. 4

In-lecture quiz (L8)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What does this print?

```
print(len(["Excellent", "text", "box"]))
```

A. Nothing or an error
B. Excellent text box
C. 3
D. 16
E. 18

Modeling memory

CS114 L8 (M4)

How data is stored

- The association of variable names with values is part of the *memory* of the computer
- Each variable is said to have a *slot* in memory that stores a value
- With mutable lists, we'll find that the arrangement of memory is complicated!
- We need a mental model of how memory works

Why it’s hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

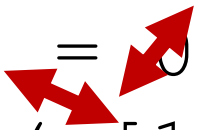
Why it's hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

A change in `x` was visible in `y`

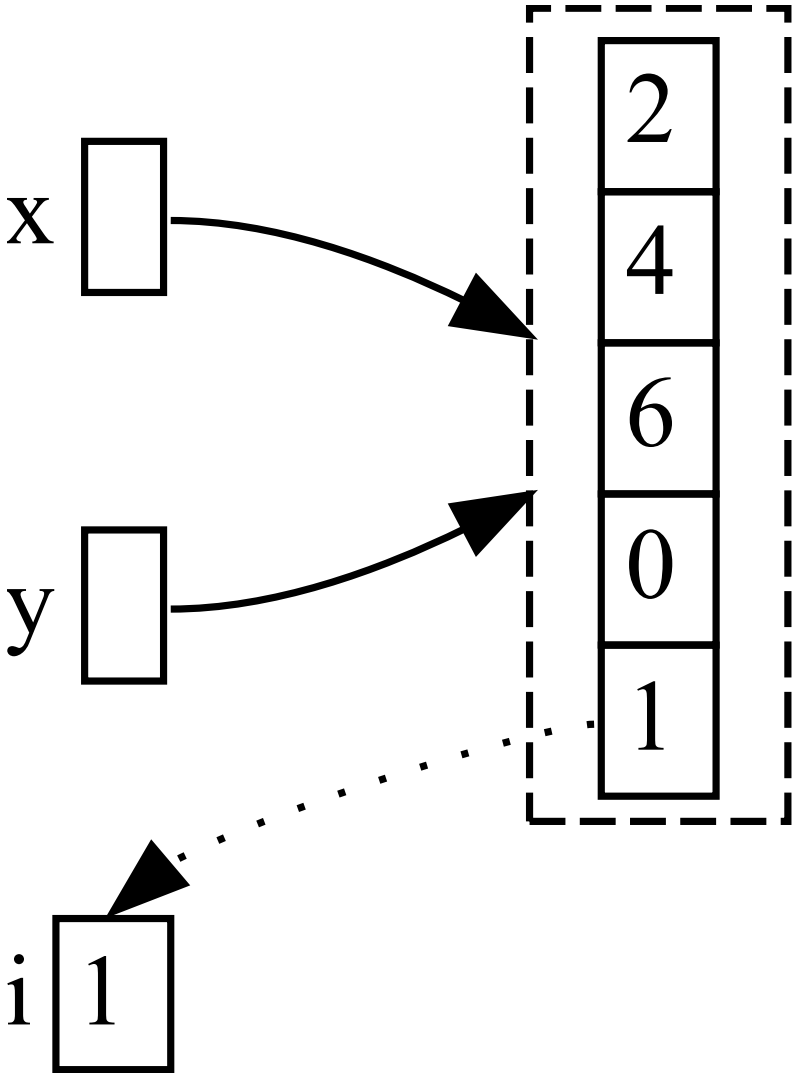
Why it's hard

```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

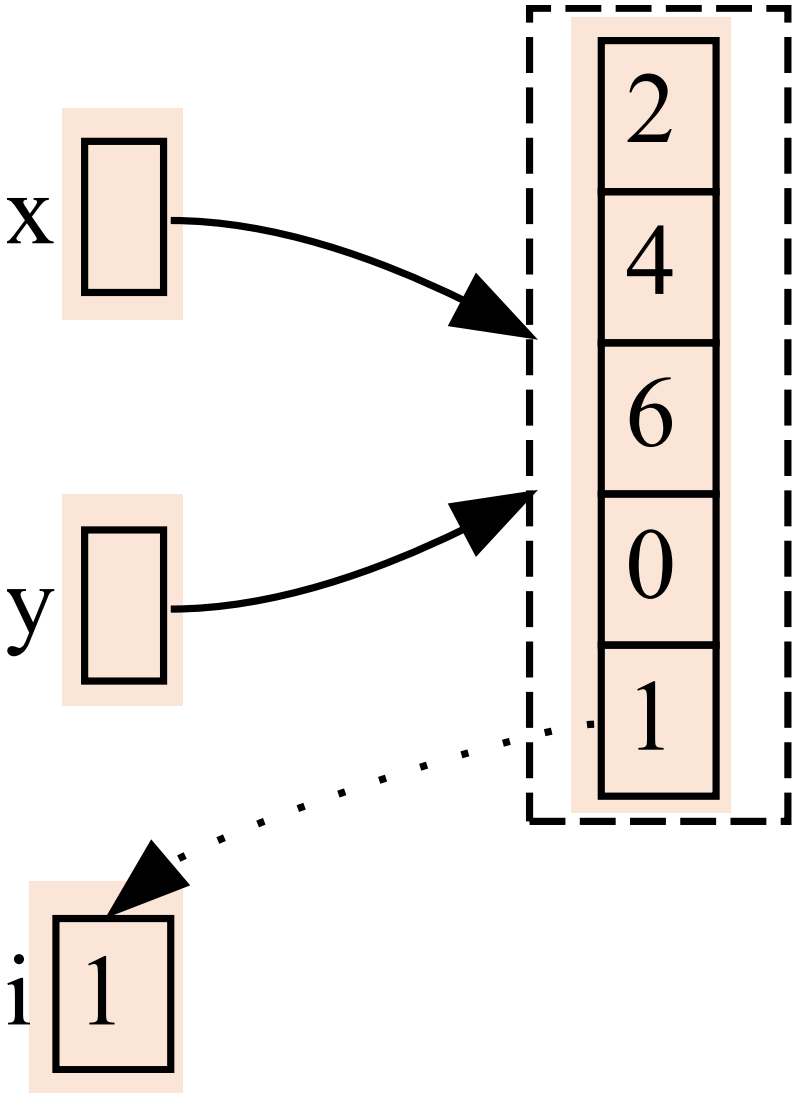


And yet this changed nothing!

The graph model of memory

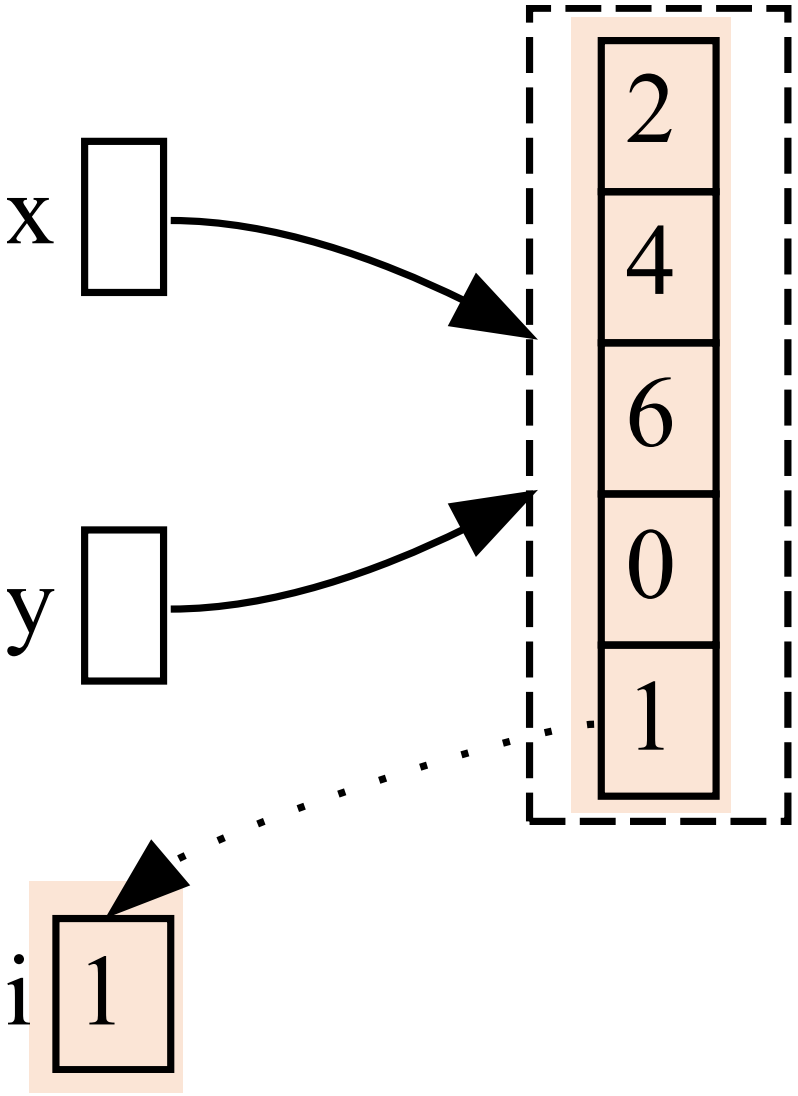


The graph model of memory



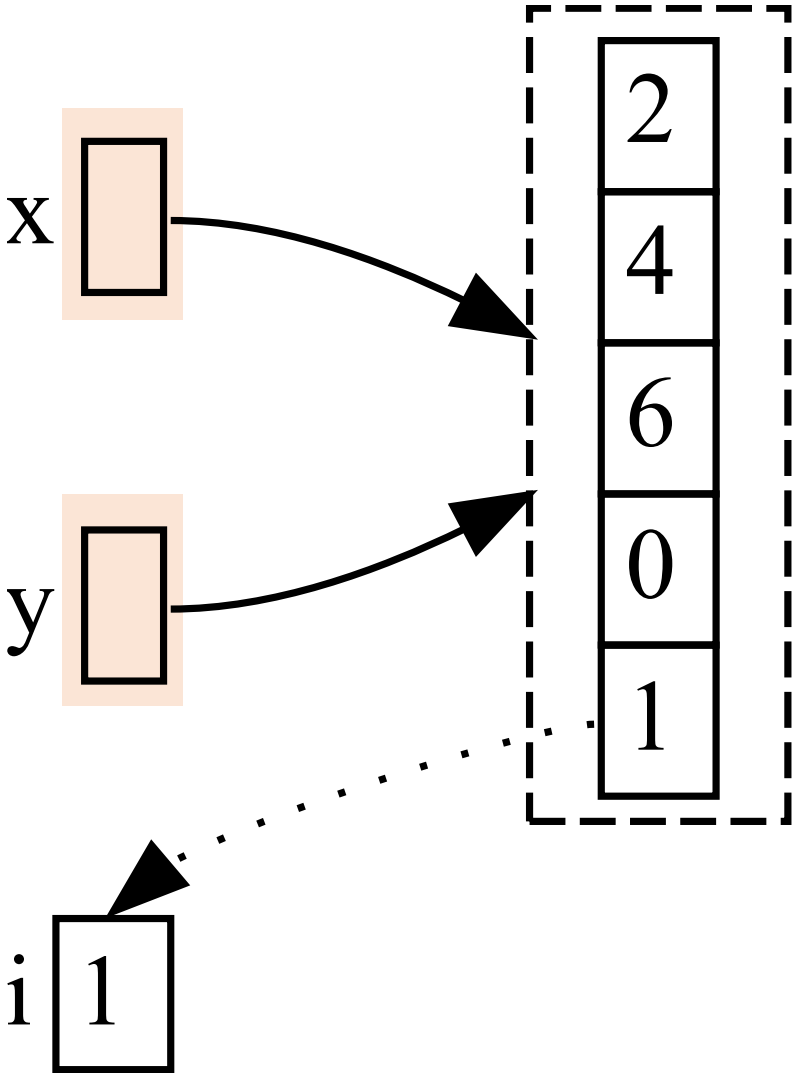
Memory *slots* store *values*

The graph model of memory



Numbers (both `int` and `float`) and strings are values.

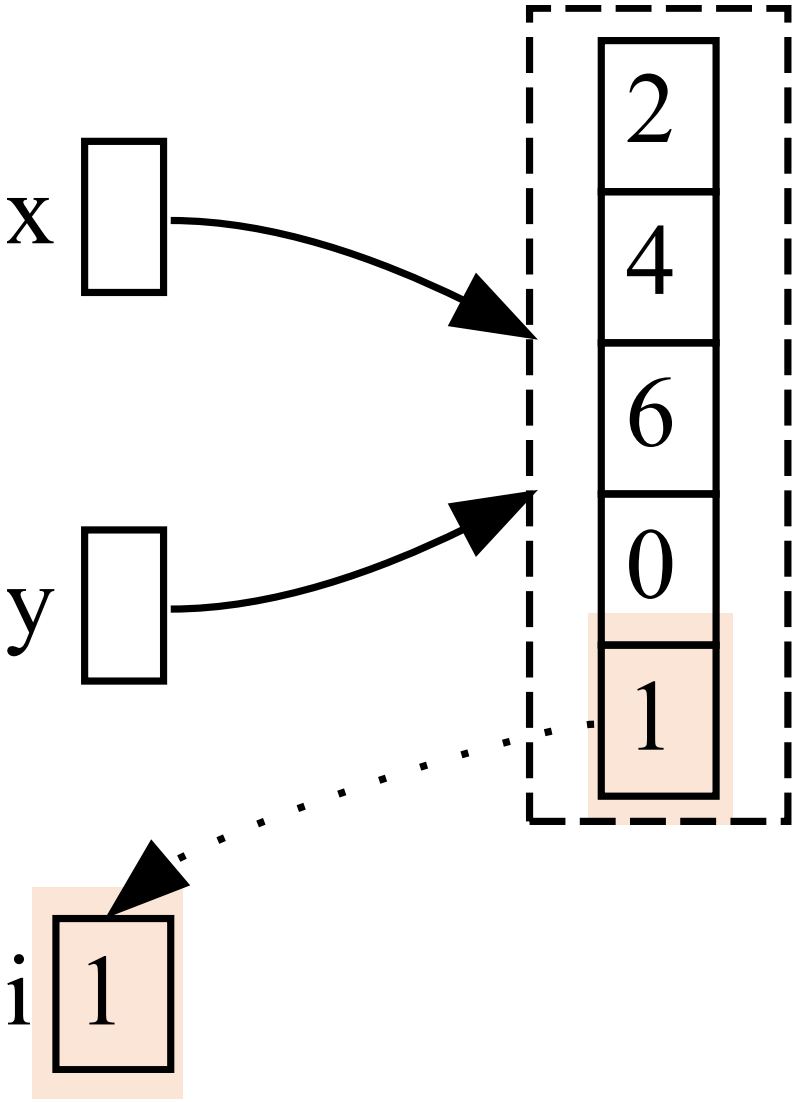
The graph model of memory



List *references* are values! The thing in the memory slot is not the list, it is a *reference* to the list!

(Shown as an arrow here. “Pointer” usually has the same meaning.)

The graph model of memory

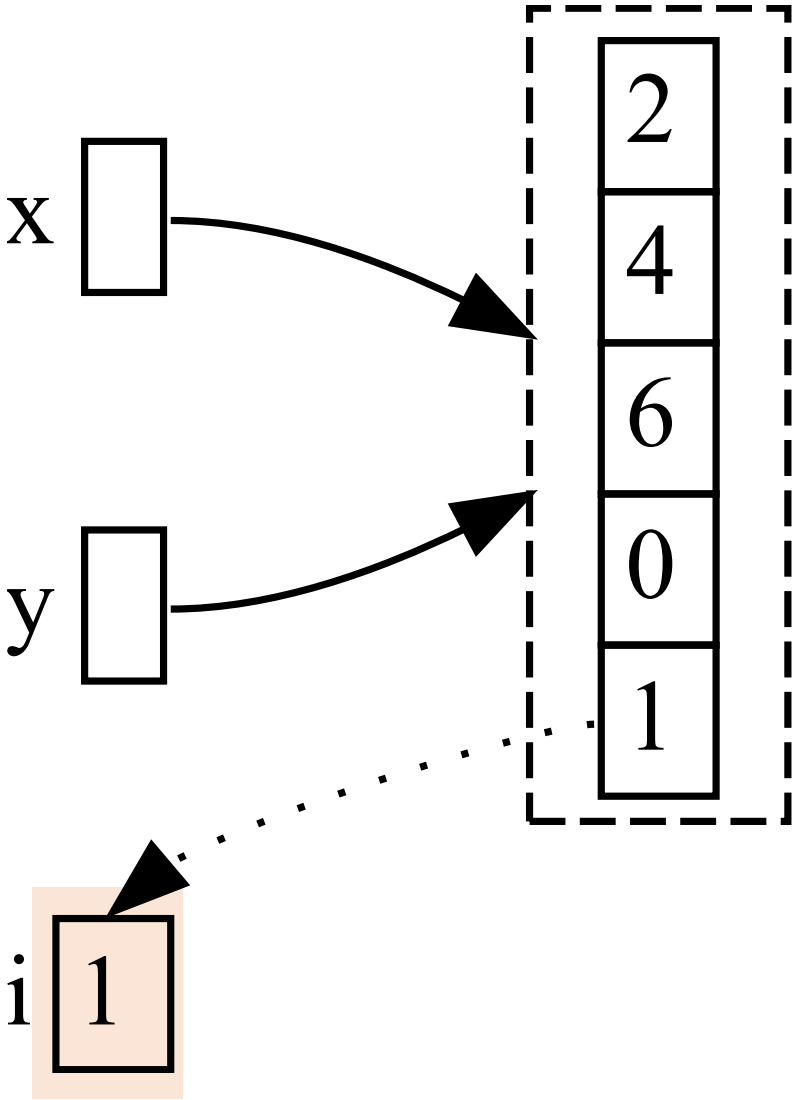


When we index, we copy the value out of the slot, so

`i = y[4]`

copies the 1 from `y[4]` to `i`

The graph model of memory



A for loop is just short-hand for copying the values out of the array, so

```
for i in y:
    …
```

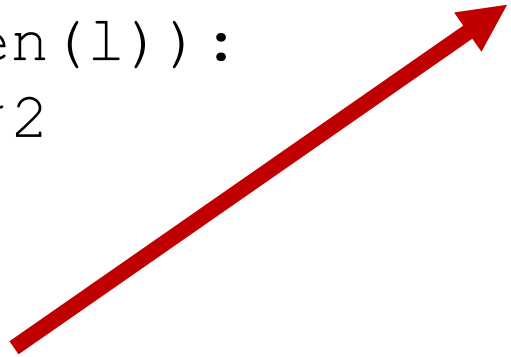
does the same. Changing `i` doesn't change `y[4]`, because it was a copy!

The graph model of memory

- We've just added a major complication to Python: *reference types*
- A reference type is a kind of value that is stored as a *reference*, rather than the content being stored directly in a slot
- Reference types allow spooky action at a distance
- Let's write a function to square every value in a list

Reference types

```
def squareList(l: list[float]) -> None:
    for i in range(len(l)):
        l[i] = l[i]**2
```



No return??? Then how does this do anything?

Reference types

```
def squareList(l: list[float]) -> None:
    for i in range(len(l)):
        l[i] = l[i]**2
```

```
a = [2, 4, 6, 0, 1]
squareList(a)
a[0] == 4
a[1] == 16
…
```

Let's draw what the memory in this program looks like on the board.

Example of mutating a list

- Let's make a function to remove all the 2s from the factors of a list of numbers

```
def removeFactorsOfTwo(l: list[int]) -> None:
    for idx in range(len(l)):
        val = l[idx]
        while val%2 == 0 and val > 1:
            val = val // 2
        l[idx] = val
```

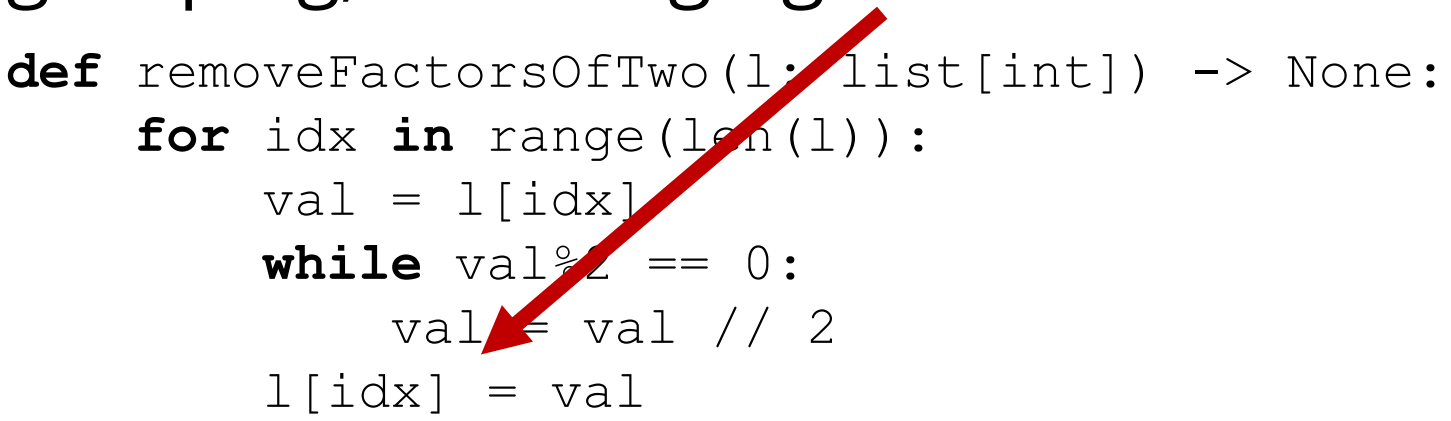
Example of mutating a list

This function doesn't return anything, because it only *modifies* the list.

```
def removeFactorsOfTwo(l: list[int]) -> None:
    for idx in range(len(l)):
        val = l[idx]
        while val%2 == 0:
            val = val // 2
        l[idx] = val
```

Example of mutating a list

Why did we loop by indices instead of **`for`** `val` **`in`** `l`? The value is copied out of the grouping, so changing `val` does nothing!



```
def removeFactorsOfTwo(l: list[int]) -> None:
    for idx in range(len(l)):
        val = l[idx]
        while val%2 == 0:
            val = val // 2
        l[idx] = val
```

Aside on plurals

The plural of “index” is “indices” because English was designed by sociopaths.

Again with feeling!

```
a = [1, 2, 3]
for v in a:
    v = v * 2
# a is still [1, 2, 3]
```

```
a = [1, 2, 3]
for i in range(len(a)):
    a[i] = a[i] * 2
# a is now [2, 4, 6]
```

Again, let's draw what memory in these programs looks like on the board

Isn’t this confusing?

- Yup!

- … what, you thought I was going to have a justification here?

Isn’t this confusing?

- Yup!
- Usual justification: copying things takes time, so don’t. A list can be millions of slots!
- However, using things by reference can be helpful. Think of `runningAverage` or `squareList`.