Warmup (L9)

Write a function `reverse(lst)` to reverse a list *in place*.

Hint: Swap two values using a temporary variable, e.g.

```
tmp = lst[a]
lst[a] = lst[b]
lst[b] = tmp
```

(If you're familiar with slicing or list.reverse, don't use them. Just indexing.)

Modeling memory

CS114 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
```

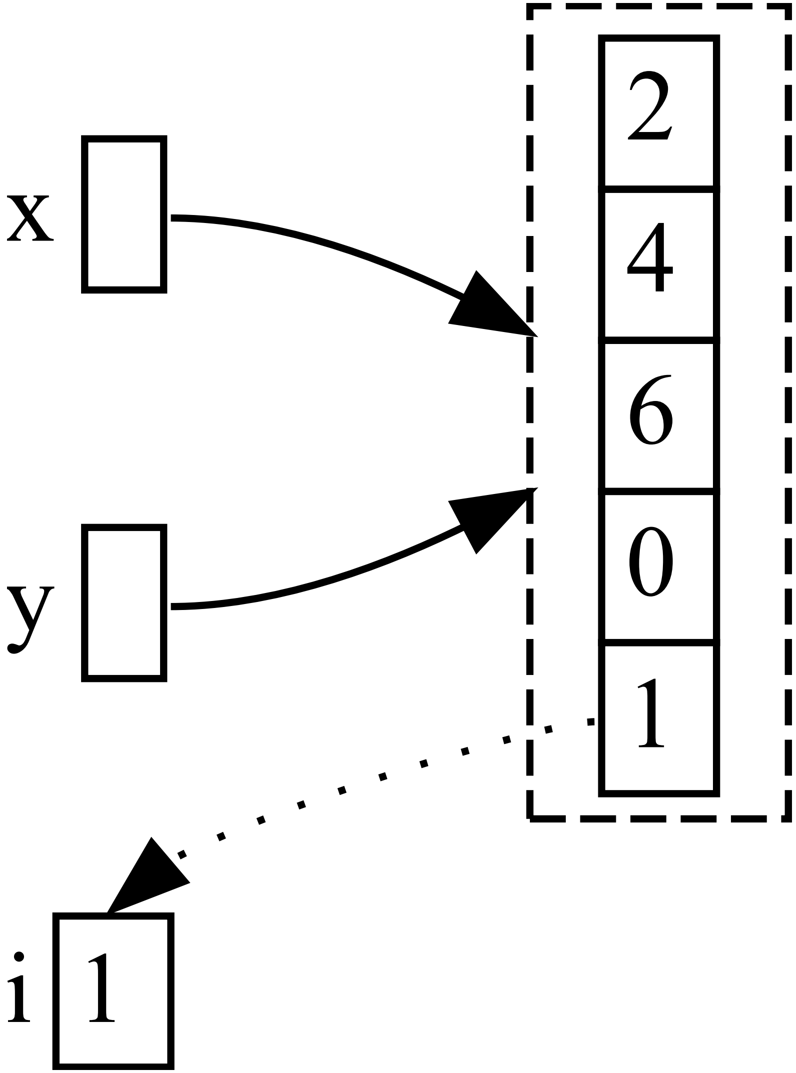
A change in `x` was visible in `y`

Why it’s hard

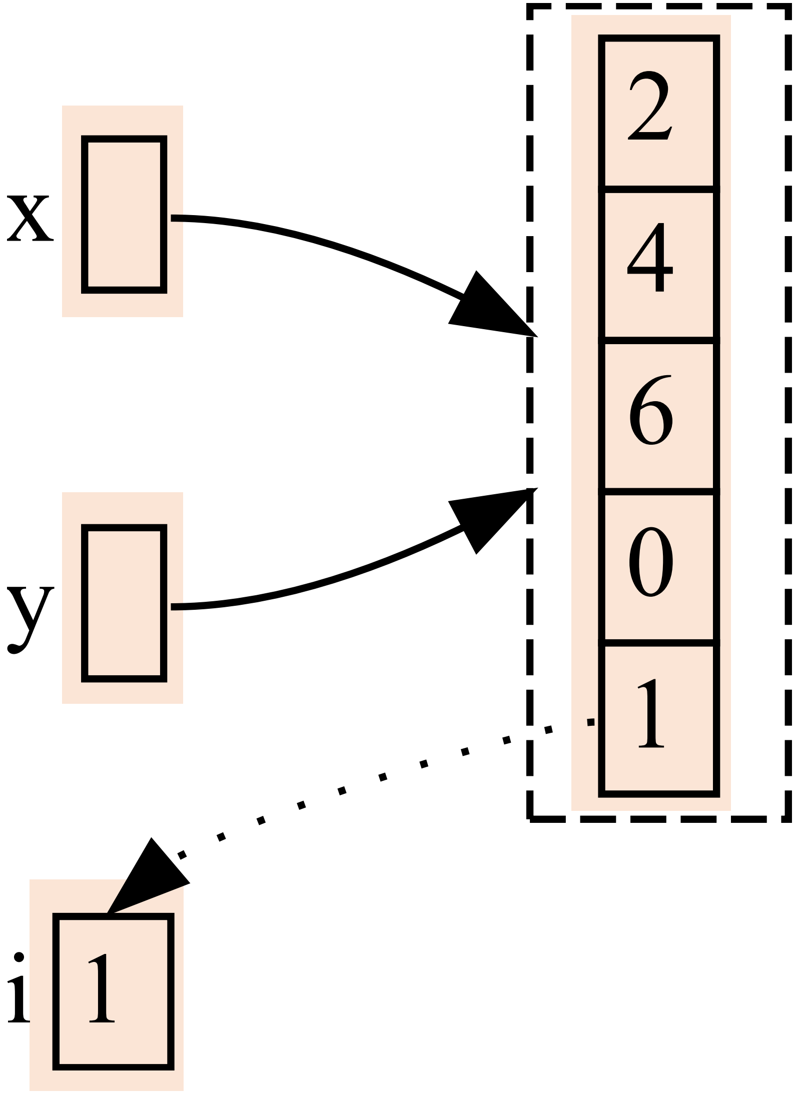
```
x = [2, 4, 6, 0, 1]
y = x
x[1] = 8
print(y[1]) # prints 8
for i in y:
    i = 0
print(y[1]) # prints 8
```

And yet this changed nothing!

The graph model of memory

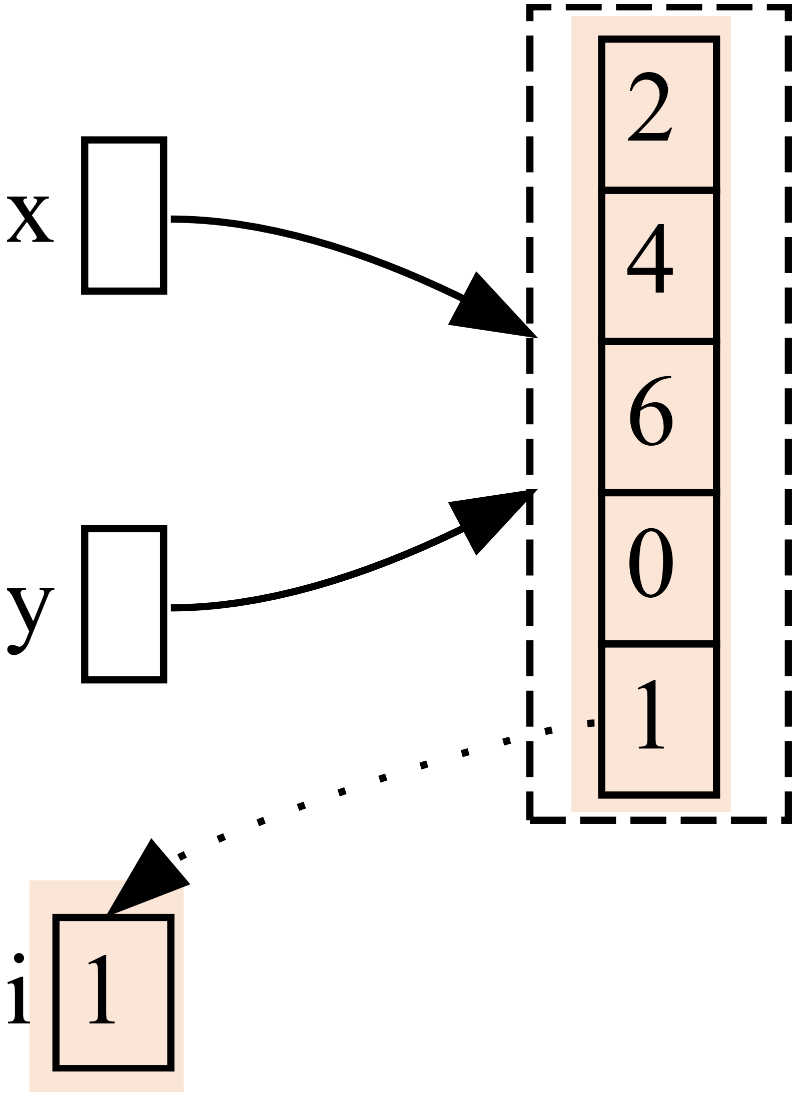


The graph model of memory



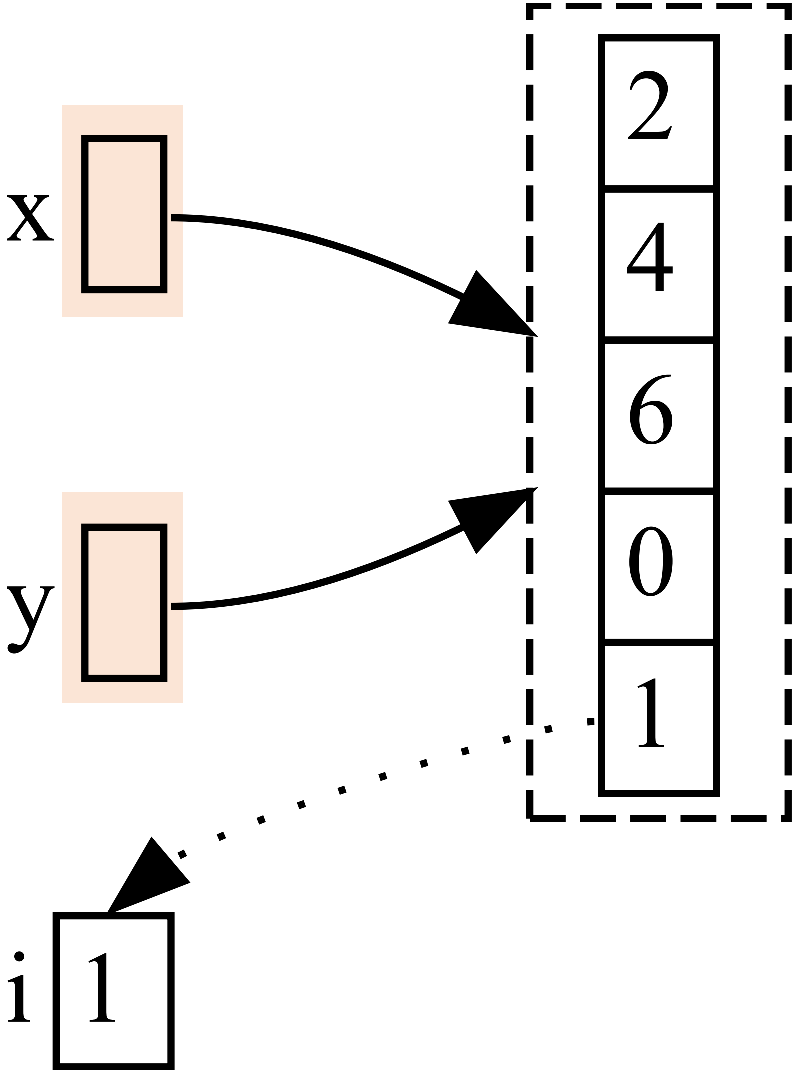
Memory *slots* store *values*

The graph model of memory



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

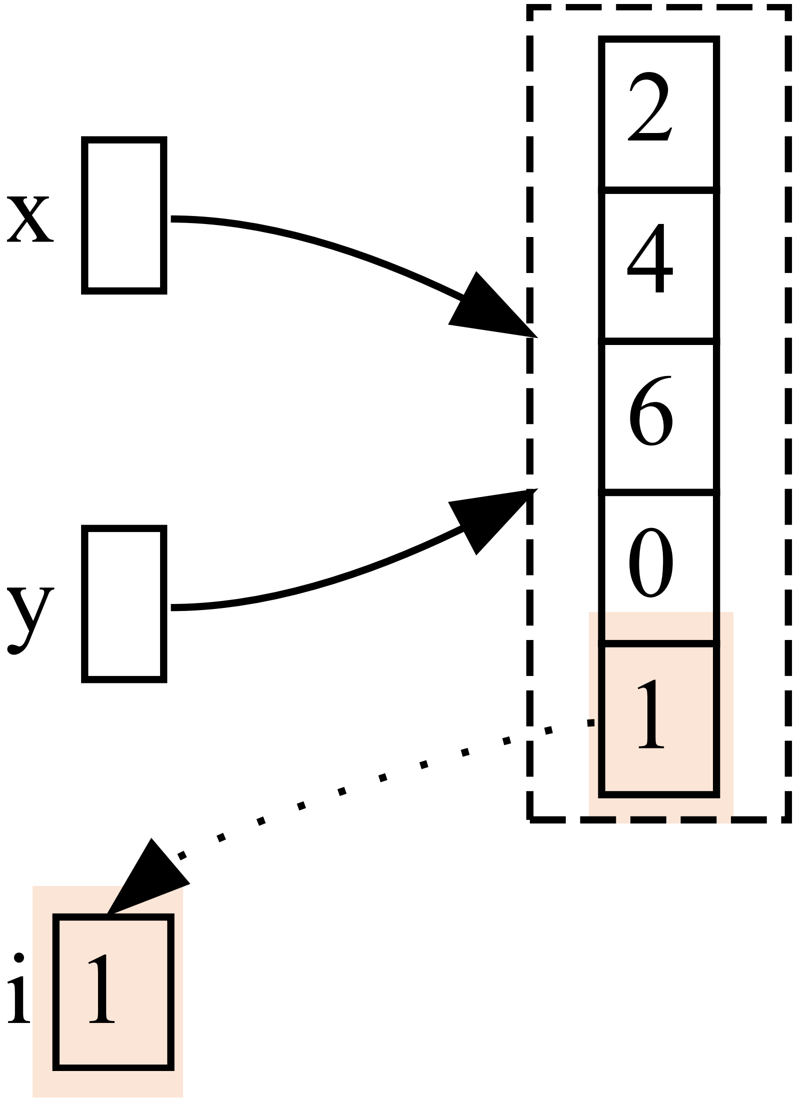
The graph model of memory



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

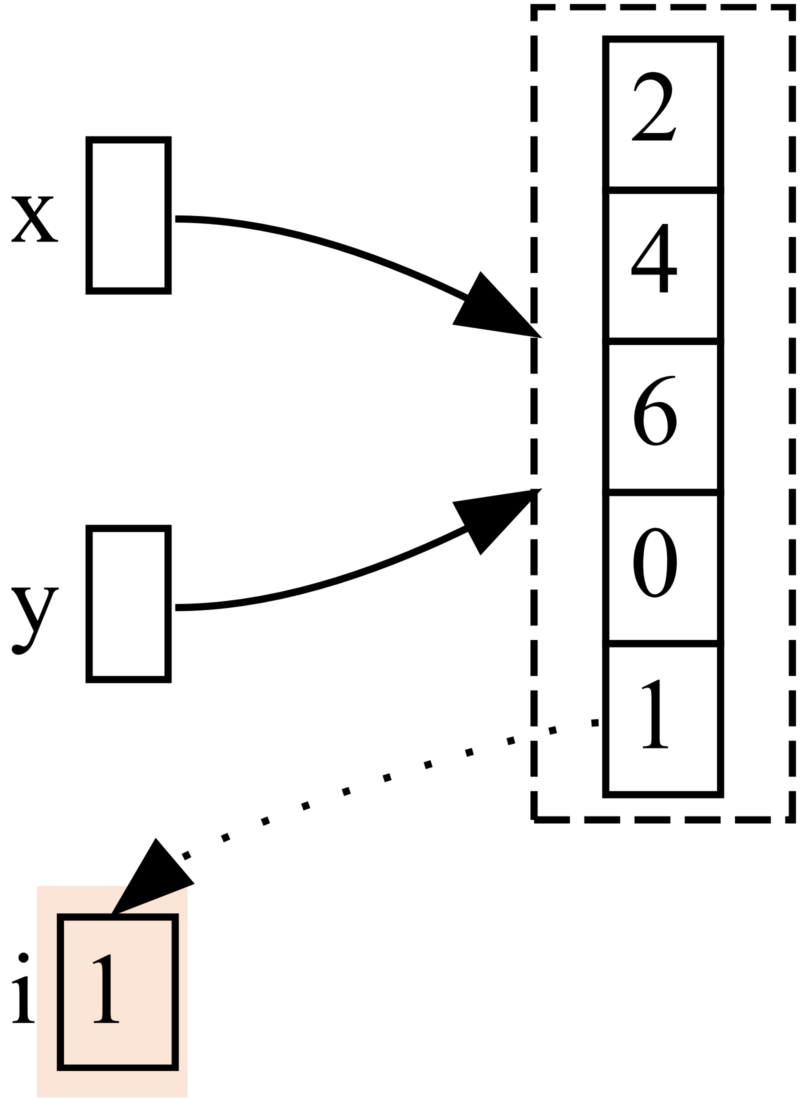


When we index, we copy the value out of the slot, so

`i = y[4]`

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

The graph model of memory



A for loop is just shorthand 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
```

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`.

Pedantry corner

- This is a *mental model*, not a literal description of what's going on in memory
- If a type is immutable (think strings), there's no way for you to tell if what's in the slot is a reference or a value
- Values are easier to reason about, so in our mental model, we think of all immutable things as values

A note on equality

- Two lists are equal (== says "`True`") if they contain equal elements, even if they're not the same reference in memory
- If you want to know if they're the same reference in memory, there's another comparison for it, "**`is`**"

```
x = [2, 4, 6, 0, 1]
y = x
z = [2, 4, 6, 0, 1]
x == y and x == z # True
x is y # True
x is z # False
y[1] = 8
x == z # False, spooky action at a distance!
```

A note on equality

- Two lists are equal (== says “`True`”) if they contain equal elements, even if they’re not the same reference in memory
- If you want to know if they’re the same reference in memory, there’s another comparison for it, “**`is`**”
- It’s pretty rare to need **`is`**, and **`is`** can reveal surprising details about Python’s *real* memory model, so usually use ==

Example break

- Let’s find the greatest value in a list

```
def greatest(lst: list[float]) -> float:
    assert len(lst) > 0, "No greatest in
                          an empty list"
    r = lst[0] # Need to start with
               # something!
    for val in lst:
        if val > r:
            r = val
    return r
```

Expanding lists

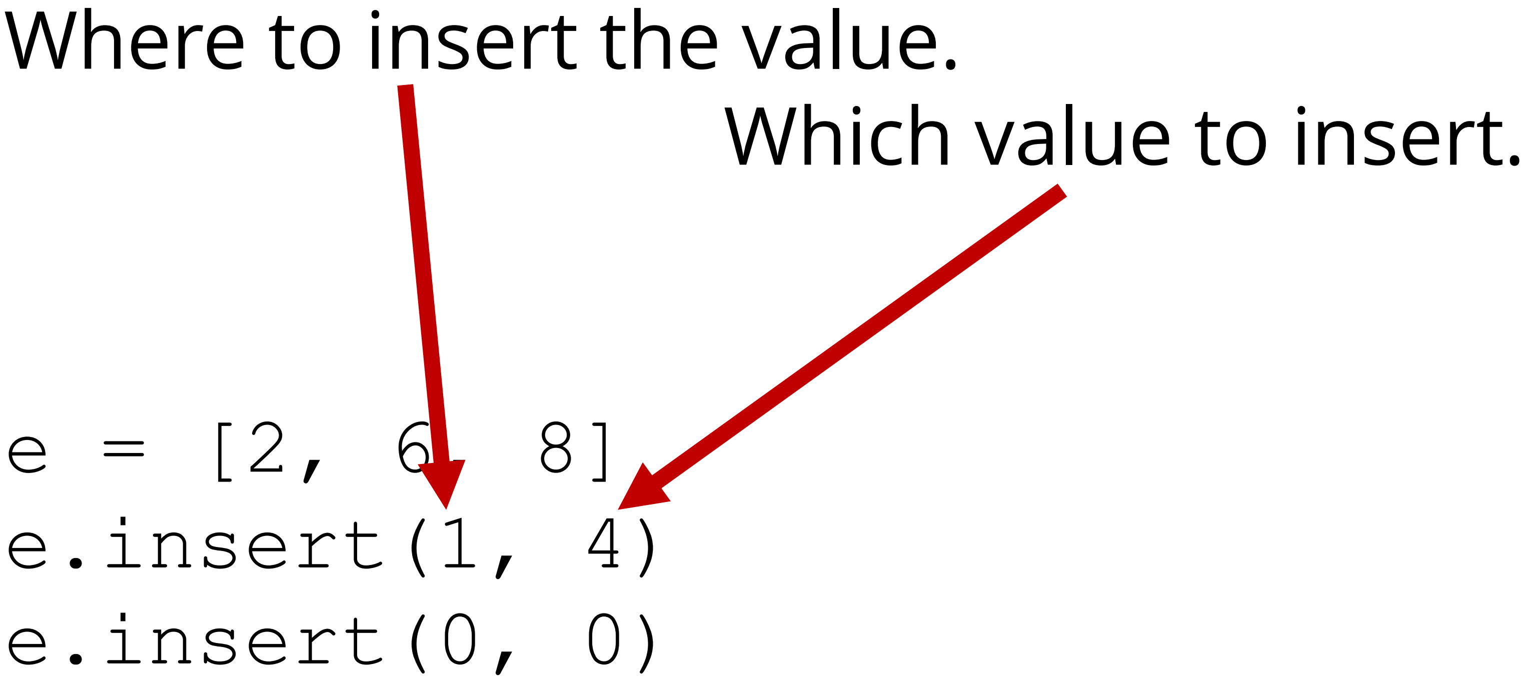
CS114 M4

Insertion

- As well as *changing* values in the list, we can *insert* slots into the list (and put values there)

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion



Where to insert the value.

Which value to insert.

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

Remember the dot (as in `math.sqrt`)? It's also how you get special functions that act on lists (and other things we'll see later).

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

These functions you get with dot ("on" the list) are called "methods".

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

Let’s add some prints to understand our lists as the code runs.

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Appending

- There’s a special version of `insert` for the common case of inserting at the end

```
e = [0, 2, 4, 6, 8]
e.append(10)
```

Using `append`

- Let’s collect all the common divisors of two integers into a list (sort of “all-cd” instead of gcd)

```
def divisors(x: int, y: int) -> list[int]:
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    return r
```

Shrinking lists

- Just like we can add items, we can remove items by *pop*ping them out of the list:

```
e = [0, 2, 4, 6, 8]
e.pop(0) # Removes element at index 0
# e is now [2, 4, 6, 8]
e.pop() # By default, removes the last element
# e is now [2, 4, 6]
```

How does this affect loops?

- Nothing in Python stops you from changing the length of a list while you loop through it
- But, the behavior is *hugely* confusing, so best avoided

```
x = [2, 4, 6, 0, 1]
for v in x:
    print(v) # Which values will actually print here???
    x.pop(0)
```

Slicing and joining

CS114 M4

References are Hell!

- Remember, lists are a reference type: if you pass a list to a function and it modifies it, you will see the changes!
- This was done because copying is slow
- But sometimes you *want* to copy!

```
x = [2, 4, 6, 0, 1]
y = x[:]
y[0] = 4
# x is still [2, 4, 6, 0, 1]
```

But what’s this thing???

New syntax!

- `x[:]` was a *slice* of `x`
- Why did we call it a slice when it was a copy? Because that ~~hamburger~~ operator is so much more powerful!

```
x = [2, 4, 6, 0, 1]
x[1:3] == [4, 6]
```