

# CS114

---

## Module 4: Strings and Lists

# Sequences

---

CS114 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 another 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

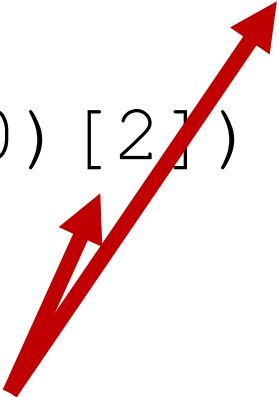
---

```
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(ish) 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 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 just `list` when you know what's in it!

# List example

---

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

# 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!)

# 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:
```



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.

```
    return false
```

# List example

---

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



Be wary of this type!

Remember: types are documentation! Don't just write "any" to make the type checker shut up!

```
        return True
```

# 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)
```

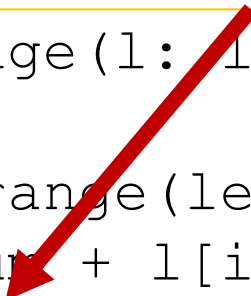
# Using mutation

---

- 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)
```



# 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

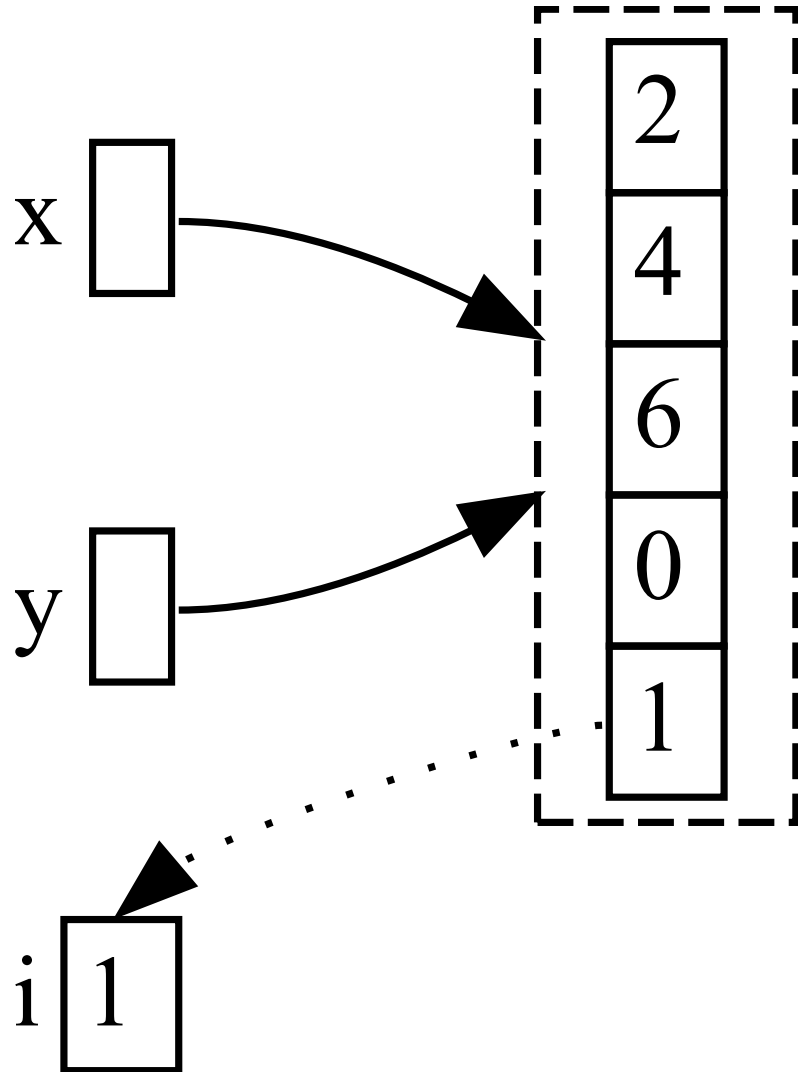
---

```
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
```

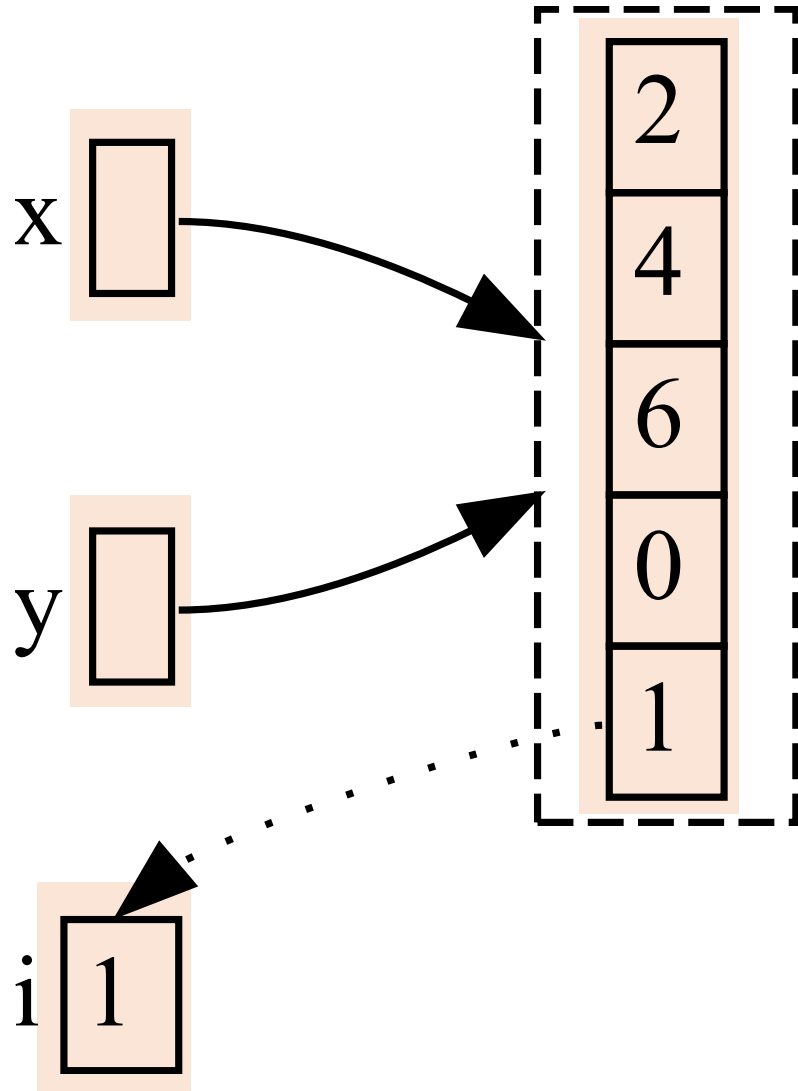


And yet this changed nothing!

# The graph model of memory

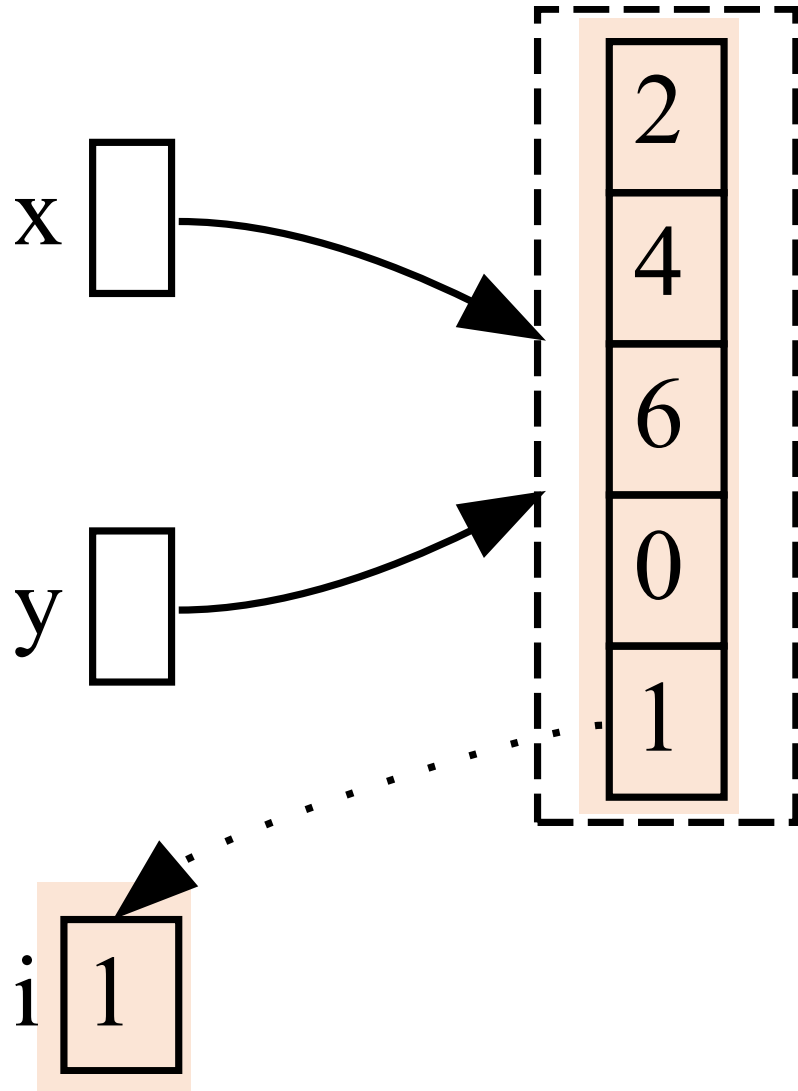


# The graph model of memory



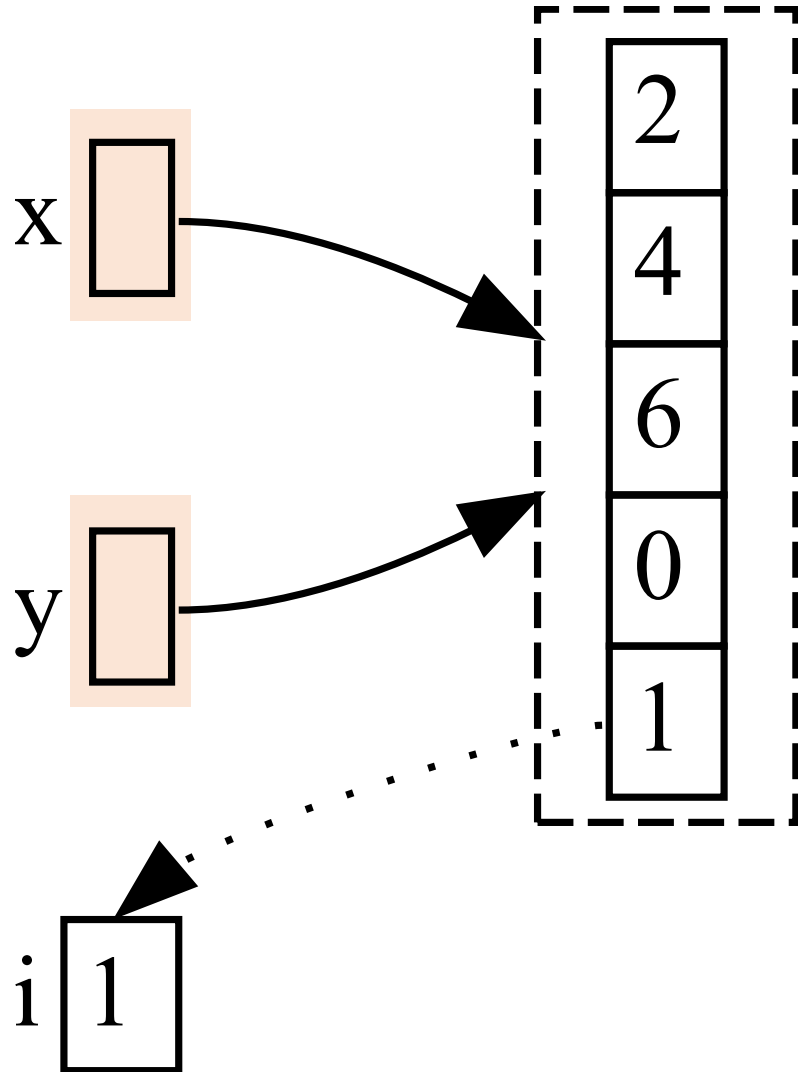
Memory *slots* store *values*

# The graph model of memory



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

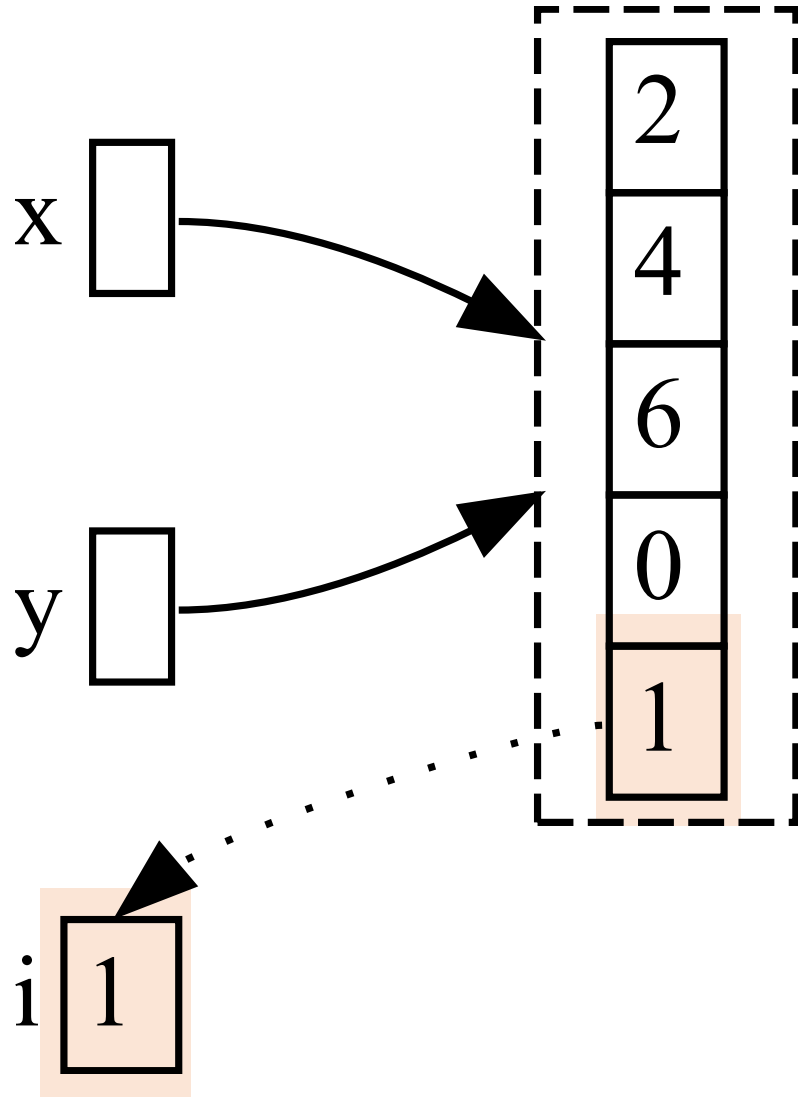
# The graph model of memory



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



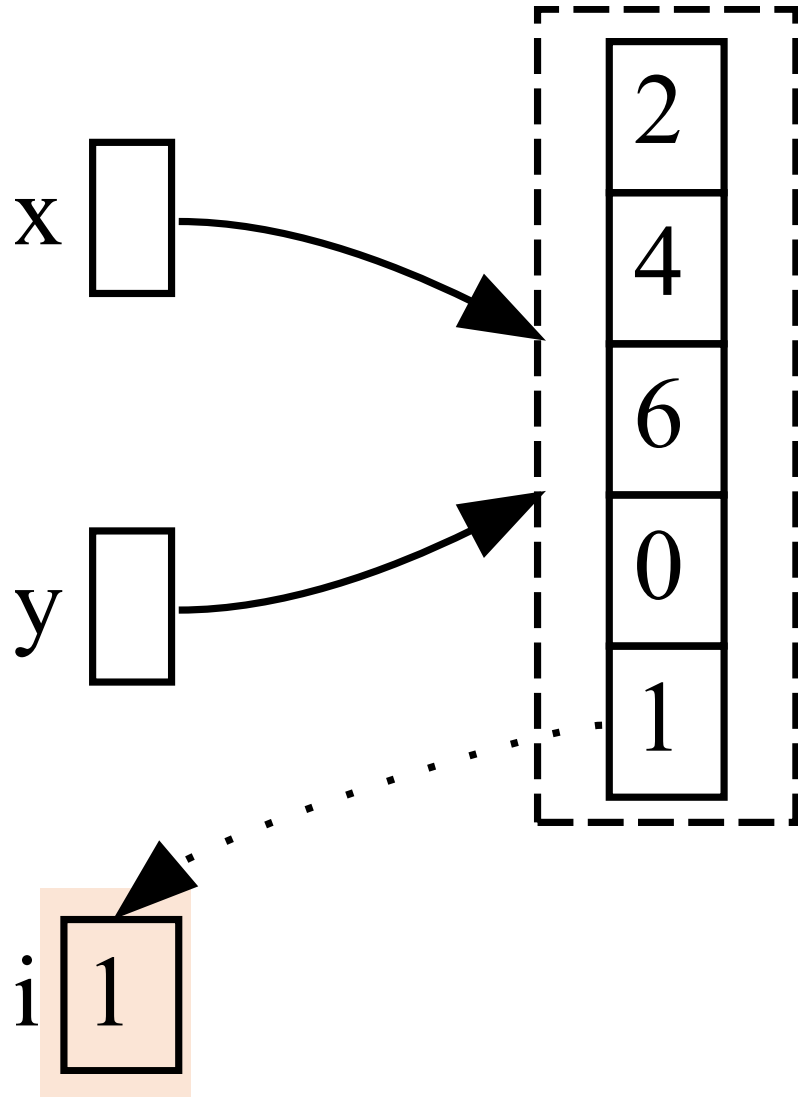
When we index, we copy  
the value out of the slot,

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!

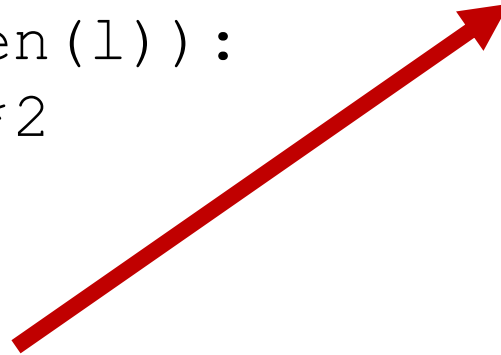
# The graph model of memory

- 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.



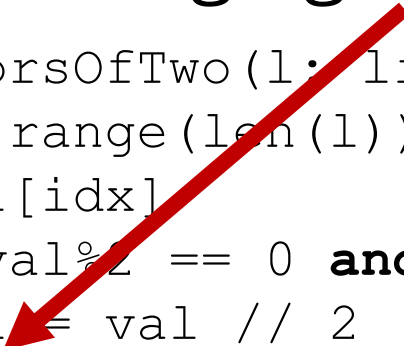
```
def removeFactorsOfTwo(l: list[int]) -> None:
    for idx in range(len(l)):
        val = l[idx]
        while val%2 == 0 and val > 1:
            val = val // 2
        l[idx] = val
```

# 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 and val > 1:
            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)

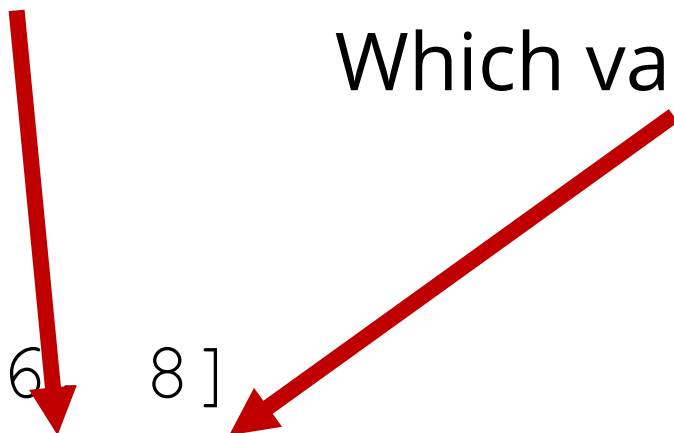
```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

# Insertion

---

Where to insert the value.

Which value to insert.

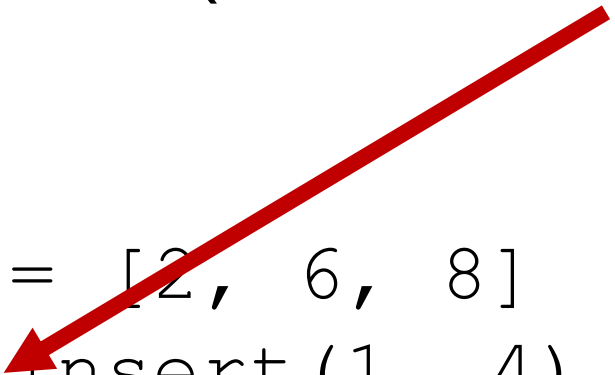


```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

# 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).

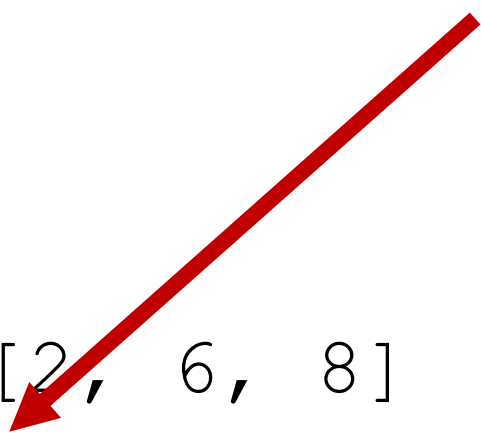


```
e = [2, 6, 8]  
e.insert(1, 4)  
e.insert(0, 0)
```

# 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 *popping* 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]
```

# New syntax!

---

- `(sequence) [from:to]`
- Makes a *copy* of the sequence, from `from` to `to`
- Like `range`, `from` is inclusive, `to` is exclusive
- The `from` and `to` are the indices, not values!

# Basic slices

---

- Let's split a list in half using slicing

```
midpoint = len(lst) // 2  
left = lst[0:midpoint]  
right = lst[midpoint:len(lst)]
```

# More advanced slicing

---

- Where did `[ : ]` come from?
- Both from and to are optional!
  - By default, `from = 0`
  - By default, `to = len(the sequence)`
  - With both defaults, you copy the whole list
- There's a third part, also optional: the step (just like `range`)
  - `lst[::2]` gets every second element
  - `lst[::-1]` reverses a sequence

# More advanced indexing

---

- It's common to want the last element in a list (or other sequence)
- Obvious way: `lst[len(lst) - 1]`
- Python lets you shorthand by using a negative index: `lst[-1]`
- Same works with slicing, and insert, and pop, and everything else!

# Joining

---

- You can also join (called *concatenate*) lists or other sequences with +

```
"Hello, " + "world!" == "Hello, world!"
```

```
[3, 1, 4] + [1, 5, 9] == [3, 1, 4, 1, 5, 9]
```

# The reverse interleave

---

- Let's write a function to *reverse interleave* a list
  - What this means is, for instance, turn  
[1, 2, 3, 4, 5, 6] into  
[1, 3, 5, 2, 4, 6]
  - Perfectly interleaving or reverse interleaving playing cards is a basic magicians' trick

# The reverse interleave

---

```
def reverseInterleave(deck: list) -> list:  
    return deck[::2] + deck[1::2]
```

- (Yup, that's it! Slicing is powerful, eh?)

# Sequence conversions

---

- You can convert any other sequence into a list by using `list` as a function:

```
list("Hello!") ==  
["H", "e", "l", "l", "o", "!"]
```

# The second dimension

---

CS114 M4

# The second dimension

---

- To represent anything two-dimensional, it's common to use lists within lists (nested lists)
- E.g., a tic-tac-toe board might look like this:

```
board = [ [ " ", "x", " " ],  
          [ " ", "x", "o" ],  
          [ " ", " ", " " ] ]
```

# The second dimension

---

- Think carefully about memory! Let's draw our tic-tac-toe board on the blackboard.

- It's easy to make surprising mistakes

```
board[2] = board[1]
```

```
board[2][1] = "o"
```

**# Now board[1][1] is also "o"!**

# Tuples

---

CS114 M4

# Tuples are immutable lists

- One more sequence type: tuples
- Tuples are just immutable lists
- Created with parentheses:

```
x = (2, 4, 6, 0, 1)
```

# Why tuples?

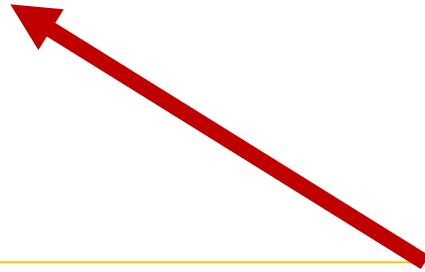
---

- Why would we want immutable lists when we already have lists?
- Usually to box together multiple values of disparate types that are related in meaning
  - The type for a list can only have one “what’s in the list” type, because it’s expandable
  - The type for a tuple can list every type in it, because it’s fixed

# Typing tuples

---

```
x: tuple[int, str] = (24601, "Jean Valjean")
```



You can name each individual type in a tuple, but for a list, they all have to be the same!

# Returning tuples

---

- Most common use is returning multiple things from a function
- We'll do this with an example in a moment

# Fun with lists

---

CS114 M4

# Longest string

---

- Find the longest string in a list
- Instead of returning a string, return a *list* of all strings of the same length, *and* the length

# Longest string

---

```
def longest(strs: list[str]) -> tuple[list[str], int]:  
    r = [strs[0]]  
    for s in strs[1:]:  
        if len(s) > len(r[0]):  
            r = [s]  
        elif len(s) == len(r[0]):  
            r.append(s)  
    return (r, len(r[0]))
```

[...]

```
(res, length) = longest(["a", "blue", "duck"])
```



You can assign to a tuple of variables to get the values out of a tuple. We could've also used indexing.

# Another version of pi

---

- One (bad?) way to calculate pi is the dartboard technique: throw darts at a square board with a quarter circle in it, and the proportion that land in the circle can tell us pi
- We'll need one new feature to do this: random numbers:

```
import random
random.random()  # A random number
                  # between
                  # 0.0 and 1.0
                  # (inclusive, exclusive)
```

# Monte Carlo pi

---

```
import random
def monteCarloPi(ounds: int) -> float:
    xs = []
    ys = []
    for _ in range(ounds):
        xs.append(random.random())
        ys.append(random.random())

    # How many fall in the quarter circle?
    inside = 0
    for idx in range(ounds):
        x = xs[idx]
        y = ys[idx]
        if x*x + y*y <= 1:
            inside = inside + 1

    return 4 * inside / ounds
```

# Module summary

---

CS114 M4

# Module summary

---

- Strings and ranges are *sequences*
- **for** loops loop over sequences
- Lists are *mutable* sequences
- Lists are *reference types*
- Memory model
- Expanding/contracting lists
- Slicing and joining lists
- Tuples are immutable lists