Algorithmic Thinking: Computing with Lists

Announcements

- Tonight (10th):
  - Lab 3
  - PA 3
  - OLI
- Tomorrow (11th)
  - PS3
  - Lab 4

Any Confusion

- Print vs Return:

```
def ?????? (a, b):
    result = a + b
    print (result)
```

```
def ?????? (a, b):
    result = a + b
    return (result)
```

- Between data types:
  “3 + 5” vs 3 + 5 “3” * 3 vs 3 * 3 6 * 5 vs 6 * 5.0

- Variables:
  output = ”hello”

  print(output) vs print(”hello”) vs print(hello)

So Far in Python

- Data types: int, float, Boolean, string
- Assignments, function definitions
- Control structures: For loops, while loops, conditionals
- Accumulating output

Otto’s Farm

This Lecture

- More algorithmic thinking
  - Example: Finding the maximum in a list
- Composite (structured) data type: lists
  - Storing and accessing data in lists
  - Modifying lists
  - Operations on lists
  - Iterating over lists

Reviewing while loops

```
# example to illustrate while loops
def print_yes(num)
    i = 1
    while i < num:
        print(“iteration:”, i, i * “Yes”)
        i = i +1
    return None
```

```
>>> print_yes(10)
iteration: 1 Yes
iteration: 2 YesYes
iteration: 3 YesYesYes
iteration: 4 YesYesYesYes
iteration: 5 YesYesYesYesYes
iteration: 6 YesYesYesYesYesYes
iteration: 7 YesYesYesYesYesYesYes
iteration: 8 YesYesYesYesYesYesYesYes
iteration: 9 YesYesYesYesYesYesYesYesYes
```

Exercise:
Do the same thing with a for loop.

Example: Finding the maximum

How do we find the maximum in a sequence of integers shown to us one at a time?

299

What's the maximum?

Example: Finding the maximum

Input: a non-empty *list* of integers.

1. Set *max_so_far* to the first number in *list*.
2. For each number *n* in *list*:
    a. If *n* is greater than *max_so_far*,
       then set *max_so_far* to *n*.

Loop

Output: *max_so_far* as the maximum of the *list*.

Representing Lists in Python

We will use a list to represent a collection of data values.

```
scores = [78, 93, 80, 68, 100, 94, 85]

colors = [‘red’, ‘green’, ‘blue’]

mixed = [‘purple’, 100, 90.5]
```

A list is an *ordered* sequence of values and may contain values of any data type.

In Python lists may be *heterogeneous* (may contain items of different data types).

Some List Operations

- **Indexing** (think of subscripts in a sequence)
- **Length** (number of items contained in the list)
- **Slicing**
- **Membership** check
- **Concatenation**
- …

Some List Operations

```
>>> names = ["Al", "Jane", "Jill","Mark"]

>>> len(names)
4

>>> Al in names
error … Al is not defined

>>> "Al" in names
True

>>> names + names
["Al", "Jane", "Jill", "Mark", "Al", "Jane", "Jill",
   "Mark"]
```

Accessing List Elements

"Al"	"Jane"	"Jill"	"Mark"	list elements
0	1	2	3	indices

```
>>> names[0]
'Al'
```

```
>>> names[4]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in
    <module> names[4]
IndexError: list index out of range
```

```
>>> names[3]
'Mark'
```

```
>> names[len(names)-1]
'Mark'
```

Slicing Lists

"Al"	"Jane"	"Jill"	"Mark"	list elements
0	1	2	3	indices

```
>>> names[1:3]          <-- slice
['Jane', 'Jill']
>>> names[0:4:2]        <-- incremental slice
[‘Al', 'Jill']
>>> names[:4]
['Al', 'Jane', 'Jill', 'Mark']
>>> names[:2]
['Al', 'Jane']
>>> names[2:]
[‘Jill', 'Mark’]
```

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n, n * s`	*n* shallow copies of *s* concatenated
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(i)`	index of the first occurence of *i* in *s*
`s.count(i)`	total number of occurences of *i* in *s*

source: docs.python.org

Modifying Lists

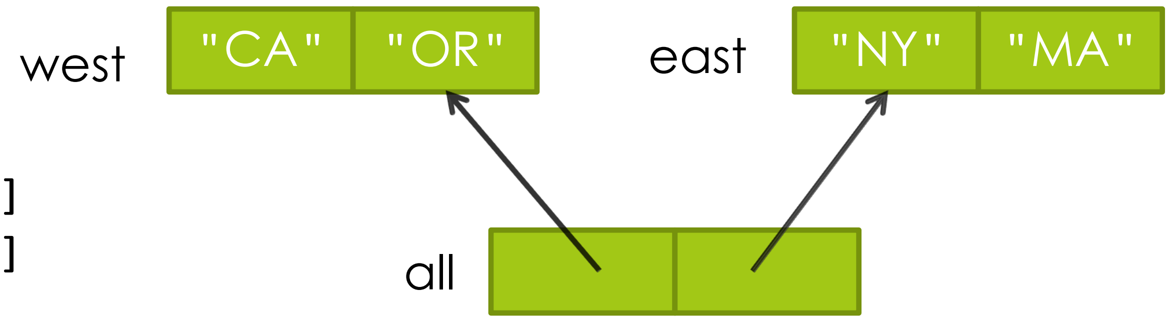
```
>>> names = ['Al', 'Jane', 'Jill', 'Mark']

>>> names[1] = "Kate"

>>> names
['Al', 'Kate', 'Jill', 'Mark']

>>> names[1:3] = ["Me",”You"]

>>> names
['Al', 'Me', 'You', 'Mark']

>>> names[1:3] = ["Me","Me","Me","Me"]
['Al', 'Me', 'Me', 'Me', 'Me', 'Mark']
```

The list grew in length, we could make it shrink as well.

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	same as `s[len(s):len(s)] = [x]`
`s.extend(x)`	same as `s[len(s):len(s)] = x`
`s.count(x)`	return number of *i*'s for which `s[i] == x`
`s.index(x[, i[, j]])`	return smallest *k* such that `s[k] == x` and `i <= k < j`
`s.insert(i, x)`	same as `s[i:i] = [x]`
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`
`s.remove(x)`	same as `del s[s.index(x)]`
`s.reverse()`	reverses the items of *s* in place
`s.sort([key[, reverse]])`	sort the items of *s* in place

source: docs.python.org

Aliasing



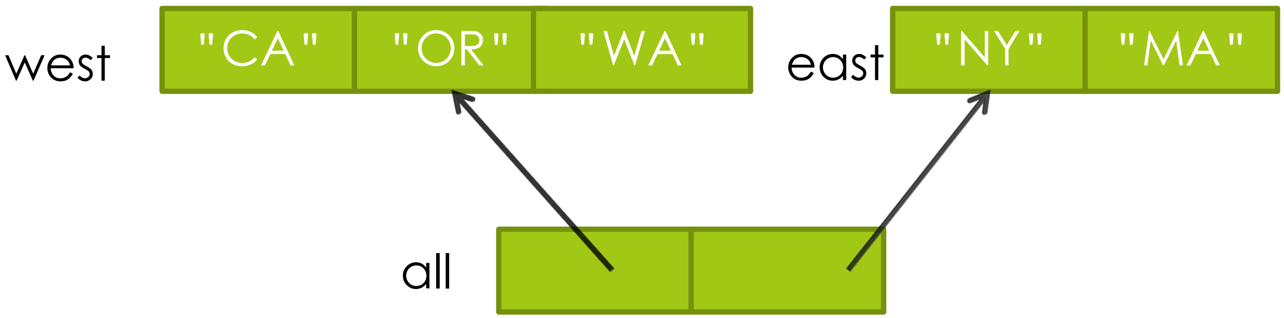

```
>>> west = ["CA", "OR"]
>>> east = ["NY", "MA"]
>>> all = [west, east]
>>> all
[["CA", "OR"],["NY", "MA"]]
```

There are two paths to the list containing state names in the West Coast.

- One through the variable **`west`**.
- The other through the variable **`all`** (namely, `all[0]`).

This is called **aliasing**.

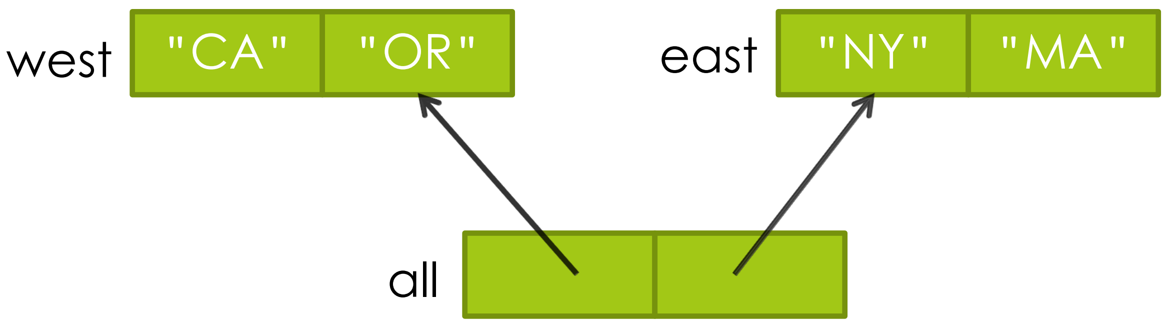
Mutability Requires Caution

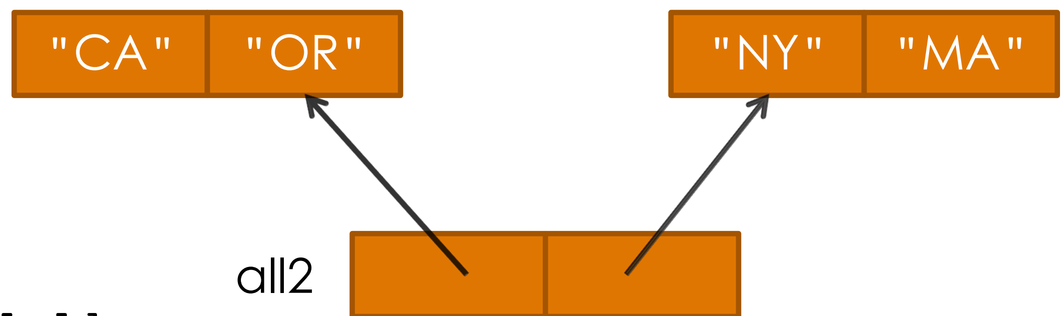



```
>>> west = ["CA", "OR"]
>>> east = ["NY", "MA"]
>>> all = [west, east]
>>> west.append("WA")
>>> all
[['CA', 'OR', 'WA'], ['NY', 'MA']]
```

All variables that are bound to the modified object change in value.

Creating Copies






```
>>> west = ["CA", "OR"]
>>> east = ["NY", "MA"]
>>> all2 = [west[:], east[:]]
>>> all2
>>> [["CA", "OR"], ["NY", "MA"]
```

Creates a ***shallow* copy**.
If list items were mutable objects, as opposed to strings as we have here, we would have needed something more.
Don't worry about it now.

No matter how I modify `west`,
`all2` will not see it.

Iterating over Lists

```
def print_colors(colors):
    for i in range(0, len(colors)):
        print(colors[i])


>>> print_colors([“red”, “blue”, “green”])
red
blue
green
```

Alternative Version

```
def print_colors(colors):
    for c in colors:
        print(c)
```

```
def print_colors(colors):
    for i in range(0, len(colors)):
        print(colors[i])
```

Python binds c to the first item in colors,
then execute the statement in the loop body,
binds c to the next item in the list colors etc.

Finding the max using Python

```
def findmax(list):
    max_so_far = list[0]                # set max_so_far to the first item
    for i in range(1,len(list)):        #check all the following items
        if list[i] > max_so_far:        # if you find a bigger value
            max_so_far = list[i]        #    update max_so_far
    return max_so_far
```

Alternative Version

```
def findmax(list):
    max_so_far = list[0]
    for item in list:
        if item > max_so_far:
            max_so_far = item
    return max_so_far
```

"For each item in the list..."

Summary

- The list data type (ordered and dynamic collections of data)
  - Creating lists
  - Accessing elements
  - Modifying lists
- Iterating over lists

Algorithmic Thinking:
Sieve of Erathosthenes

A 2000 year old algorithm (procedure) for generating a table of prime numbers.

2, 3, 5, 7, 11, 13, 17, 23, 29, 31, …

Prime Numbers

- An integer is “prime” if it is not divisible by any smaller integers except 1.

- 10 is **not** prime because 10 = 2 × 5
- 11 **is** prime
- 12 is **not** prime because 12 = 2 × 6 = 2 × 2 × 3
- 13 **is** prime
- 15 is **not** prime because 15 = 3 × 5

Testing Divisibility in Python

- x is "divisible by" y if the remainder is 0 when we divide x by y
- 15 is divisible by 3 and 5, but not by 2:

```
>>> 15 % 3

0
>>> 15 % 5

0
>> 15 % 2

1
```

What Is a “Sieve” or “Sifter”?

Separates stuff you want from stuff you don’t:

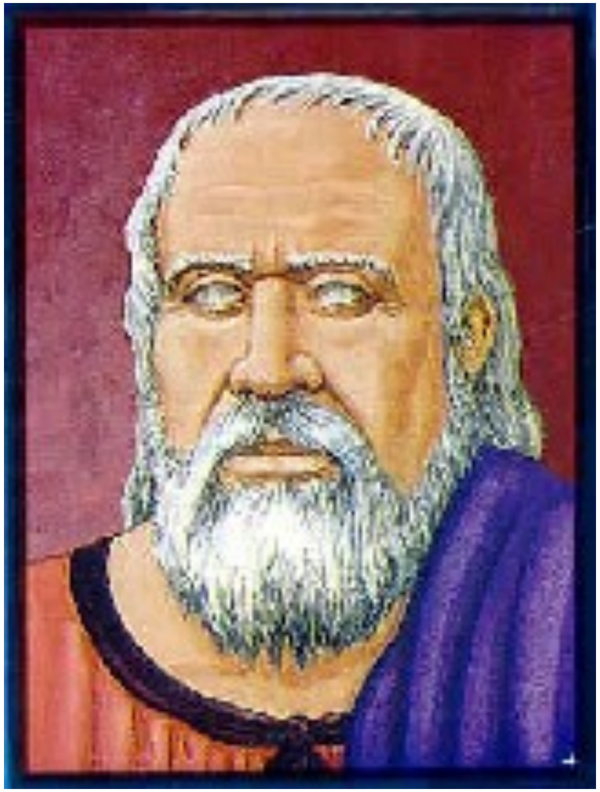
We want to separate prime numbers.

The Sieve of Eratosthenes

Start with a table of integers from 2 to N.

Cross out all the entries that are divisible by the primes known so far.

The first value remaining is the *next* prime.



Finding Primes Between 2 and 50

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

2 is the first prime

Finding Primes Between 2 and 50

	2	**3**	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 2.

Now we see that 3 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 3.

Now we see that 5 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 5.

Now we see that 7 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Filter out everything divisible by 7.

Now we see that11 is the next prime.

Finding Primes Between 2 and 50

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Since 11 x 11 > 50, all remaining numbers must be primes. Why?

An Algorithm for Sieve of Eratosthenes

Input: A number *n*:

1. Create a list *numlist* with every integer from 2 to n, in order. (Assume $n > 1$.)

2. Create an empty list *primes*.

3. For each element in *numlist*

   a. If element is not marked, copy it to the end of *primes*.

   b. Mark every number that is a multiple of the most recently discovered prime number.

Output: The list of all prime numbers less than or equal to *n*

Automating the Sieve

numlist	primes
2 3 4 5 6 7 8 9 10 11 12 13 …	

Use *two* lists: candidates, and confirmed primes.

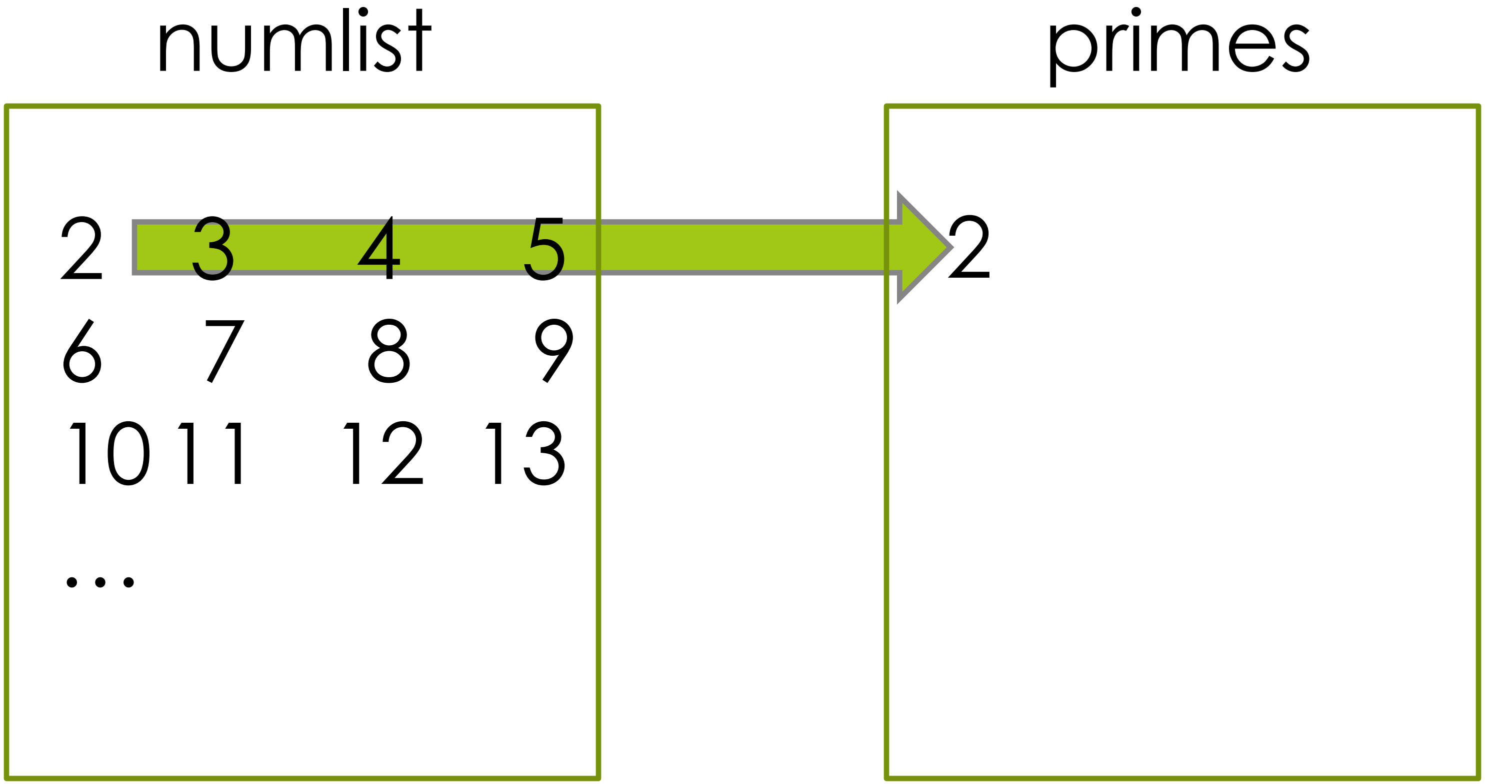
Steps 1 and 2

numlist

2	3	4	5
6	7	8	9
10	11	12	13
…			

primes

Step 3a



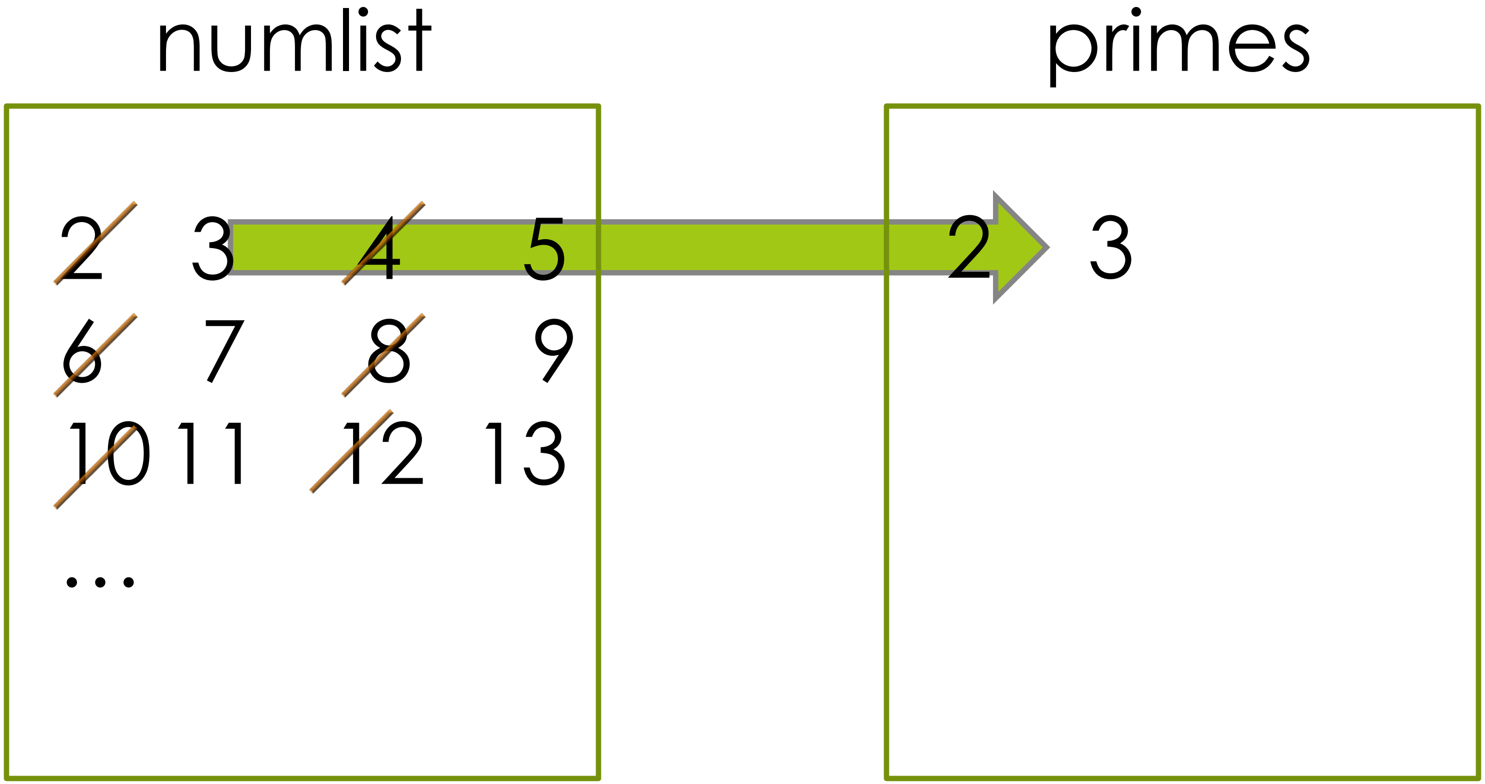
Append the current number in numlist to the end of primes.

Step 3b

numlist	primes
~~2~~ 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 …	2

Cross out all the multiples of the last number in primes.

Iterations



Append the current number in numlist to the end of primes.

Iterations

numlist

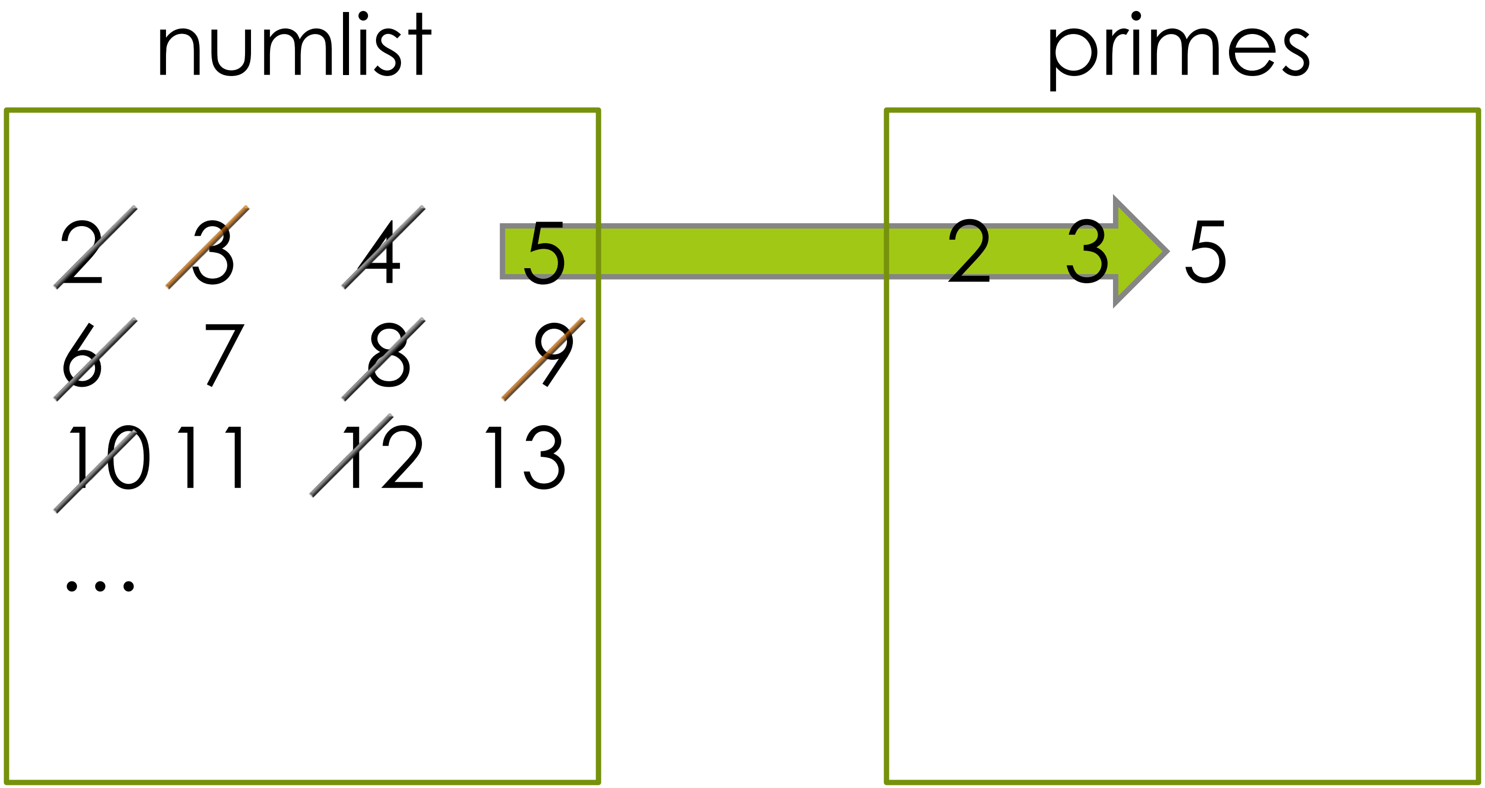
~~2~~ ~~3~~ ~~4~~ 5
~~6~~ 7 ~~8~~ ~~9~~
~~10~~ 11 ~~12~~ 13
…

primes

2 3

Cross out all the multiples of the <u>last</u> number in primes.

Iterations



Append the current number in numlist to the end of primes.

Iterations

numlist

~~2~~ ~~3~~ ~~4~~ ~~5~~
~~6~~ 7 ~~8~~ ~~9~~
~~10~~ 11 ~~12~~ 13
…

primes

2 3 5

Cross out all the multiples of the <u>last</u> number in primes.

An Algorithm for Sieve of Eratosthenes

Input: A number *n*:

1. Create a list *numlist* with every integer from 2 to n, in order. (Assume *n* > 1.)

2. Create an empty list *primes*.

3. For each element in *numlist*

    a. If element is not marked, copy it to the end of *primes*.

    b. Mark every number that is a multiple of the most recently discovered prime number.

Output: The list of all prime numbers less than or equal to *n*

Implementation Decisions

- How to implement *numlist* and *primes?*
  - *For numlist* we will use a list in which crossed out elements are marked with the special value `None`. For example,

    `[None, 3, None, 5, None, 7, None]`

- Use a helper function to mark the multiples, step 3.b. We will call it sift.

Relational Operators

- If we want to compare two integers to determine their relationship, we can use these **relational operators**:

<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to
==	equal to	!=	not equal to

- We can also write compound expressions using the **Boolean operators** and and or.

x >= 1 and x <= 1

Sifting: Removing Multiples of a Number

```
def sift(lst,k):
    # marks multiples of k with None
    i = 0
    while i < len(lst):
        if lst[i] != None and lst[i] % k == 0:
            lst[i] = None
        i = i + 1
    return lst
```

Filters out the multiples of the number k from list by marking them with the special value None (greyed out ones).

Sifting: Removing Multiples of a Number (Alternative version)

```
def sift2(lst,k):
    i = 0
    while i < len(lst):
        if lst[i] % k == 0:
            lst.remove(lst[i])
        else:
            i = i + 1
    return lst
```

Filters out the multiples of the number k from list by modifying the list. **Be careful** in handling indices.

A Working Sieve

Use the first version of sift in this function, which does the filtering using Nones.

```
def sieve(n):
    numlist = list(range(2, n+1))
    primes = []
    for i in range(0, len(numlist)):
        if numlist[i] != None:
            primes.append(numlist[i])
            sift(numlist, numlist[i])
    return primes
```

We could have used `primes[len(primes)-1]` instead.

Helper function that we defined before

Observation for a Better Sieve

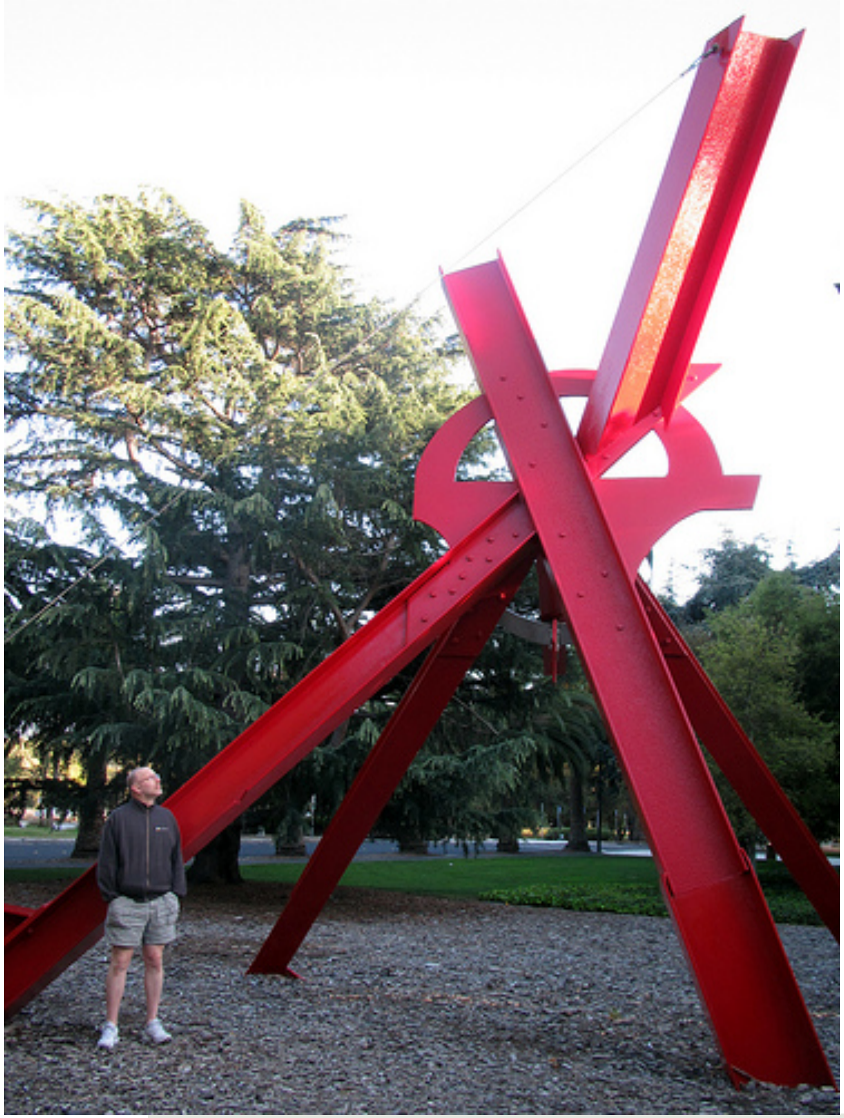
We stopped at 11 because all the remaining entries must be prime since $11 \times 11 > 50$.

	2	**3**	4	**5**	6	**7**	8	9	10
11	12	**13**	14	15	16	**17**	18	**19**	20
21	22	**23**	24	25	26	27	28	**29**	30
31	32	33	34	35	36	**37**	38	39	40
41	42	**43**	44	45	46	**47**	48	49	50

A Better Sieve

```
def sieve(n):
    numlist = list(range(2, n + 1))
    primes = []
    i = 0 # index 0 contains number 2
    while (i+2) <= math.sqrt(n):
       if numlist[i] != None:
         primes.append(numlist[i])
         sift2(numlist, numlist[i])
         i = i + 1
    return primes + numlist
```

Algorithm-Inspired Sculpture



The Sieve of Eratosthenes, 1999 sculpture by Mark di Suvero. Displayed at Stanford University.

Otto’s Farm

Otto's new farm

Otto has found a new passion: growing heritage variety, organic cabbage. He saves his money and is finally able to purchase a small, narrow 37 x 1 track of land just outside the city—now he can devote himself full time to farming! So he packs up his skinniest overalls, mounts his trusty fixie and leaves his native homeland of Lawrenceville-- off to begin a new career as a farmer.

Otto quickly discovers that farming's tough work – especially in tight overalls. So he decides to program a simple robot to plant his cabbage for him…

```python
def plant_cabbage():
    print("@")
```

- Why a function planting individual cabbage?
- What does the rest of the problem require?
- Keeping count?

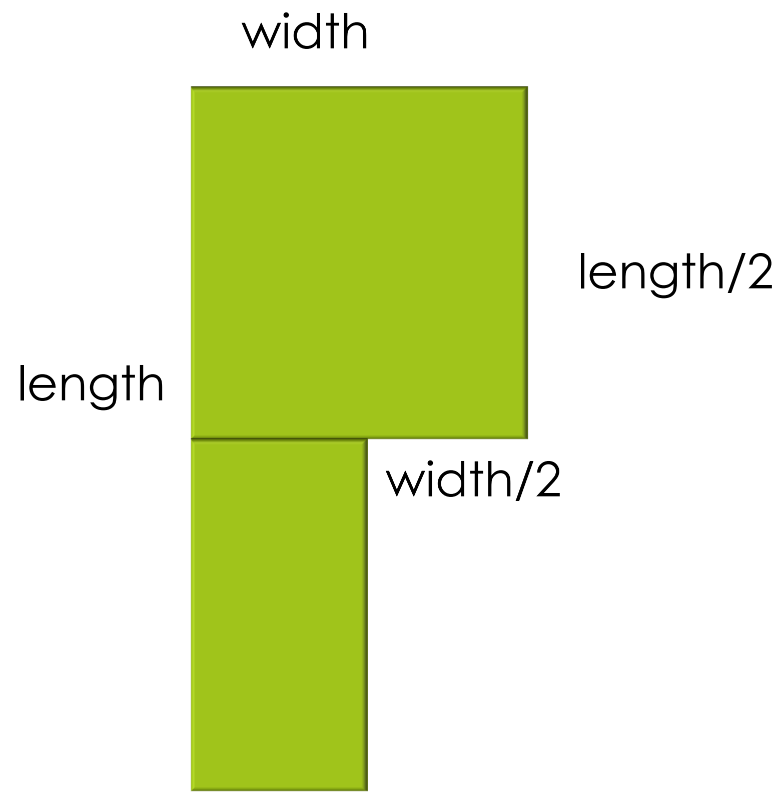
A little more space.

Otto's first crop is successful, although a little stunted. He reminds himself to leave some space between his cabbage next time. After carefully grooming his beard, he heads to the farmer's market and sells his cabbage; he's able to buy a little more land, expanding his track to 37 x 20.

Success!

Otto’s cabbages grow well and become the hit of the farm to table circuit, and his labor-saving robot allows him to devote more of his time to listening to bands you’ve probably never heard of.

With the extra income, he’s managed to increase his patch of land again. Time to add more functionality to the robot to accommodate the new field



New varieties of cabbage

Otto buys some new heritage varieties of green, purple, rainbow and yellow cabbage from the Picture and Thief Seed Co of Williamsburg. He plants an early row of the seeds to better understand how they grow: ["G", "G", "G", "G", "G", "G", "G", "G", "G", "P", "R", "R", "R", "R", "R", "R", "Y", "Y", "Y", "Y", "Y", "Y"]

Picture & Thief