Algorithmic Thinking: Computing with Lists
Announcements

- Tonight:
  - Lab 3
  - PA 3
  - OLI

- Tomorrow
  - Lab 4
Any Confusion

- **Print vs Return:**
  ```python
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:**
  ```python
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 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...
def plant_cabbage():
    print("@")

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

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?

What’s the maximum?

299
Example: Finding the maximum

Input: a non-empty list of integers.

1. Set \( \text{max}_\text{so_far} \) to the first number in list.
2. For each number \( n \) in list:
   a. If \( n \) is greater than \( \text{max}_\text{so_far} \),
      then set \( \text{max}_\text{so_far} \) to \( n \).

Output: \( \text{max}_\text{so_far} \) 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**
- ...
>>> names = ["Al", "Jane", "Jill","Mark"]

>>> len(names)
4

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

>>> "Al" in names
True

>>> names + names
Accessing List Elements

<table>
<thead>
<tr>
<th>&quot;Al&quot;</th>
<th>&quot;Jane&quot;</th>
<th>&quot;Jill&quot;</th>
<th>&quot;Mark&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

list elements
indices

>>> names[0]
'Al'

>>> names[3]
'Mark'

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

```
>>> 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']
```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of <code>s</code> is equal to <code>x</code>, else <code>False</code></td>
</tr>
<tr>
<td><code>x not in s</code></td>
<td><code>False</code> if an item of <code>s</code> is equal to <code>x</code>, else <code>True</code></td>
</tr>
<tr>
<td><code>s + t</code></td>
<td>the concatenation of <code>s</code> and <code>t</code></td>
</tr>
<tr>
<td><code>s * n</code>, <code>n * s</code></td>
<td><code>n</code> shallow copies of <code>s</code> concatenated</td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td><code>i</code>th item of <code>s</code>, origin 0</td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code></td>
</tr>
<tr>
<td><code>s[i:j:k]</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code></td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>length of <code>s</code></td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>smallest item of <code>s</code></td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>largest item of <code>s</code></td>
</tr>
<tr>
<td><code>s.index(i)</code></td>
<td>index of the first occurrence of <code>i</code> in <code>s</code></td>
</tr>
<tr>
<td><code>s.count(i)</code></td>
<td>total number of occurrences of <code>i</code> in <code>s</code></td>
</tr>
</tbody>
</table>

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

```python
>>> 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"]]
```

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

```
>>> west = ["CA", "OR"]
```

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.`
Iterating over Lists

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

>>> print_colors(['red', 'blue', 'green'])
red
blue
green
```
def print_colors(colors):
    for c in colors:
        print(c)

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)):
        if list[i] > max_so_far:
            max_so_far = list[i]  # if you find a bigger value

    return max_so_far
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...”
Exercise

Write a function that returns how many of numbers in `num_list` are odd:

```python
def num_odd(num_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 Eratosthenes
A 2000 year old algorithm (procedure) for generating a table of prime numbers.

2, 3, 5, 7, 11, 13, 17, 23, 29, 31, …
An integer is “prime” if it is not divisible by any smaller integers except 1.

- 10 is not prime because $10 = 2 \times 5$
- 11 is prime
- 12 is not prime because $12 = 2 \times 6 = 2 \times 2 \times 3$
- 13 is prime
- 15 is not prime because $15 = 3 \times 5$
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 is the first prime
Finding Primes Between 2 and 50

Filter out everything divisible by 2.
Now we see that 3 is the next prime.
Filter out everything divisible by 3.
Now we see that 5 is the next prime.
Filter out everything divisible by 5.

Now we see that 7 is the next prime.
Filter out everything divisible by 7.

Now we see that 11 is the next prime.
Finding Primes Between 2 and 50

Since $11 \times 11 > 50$, 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

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
Append the current number in numlist to the end of primes.
Step 3b

numlist

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

primes

2

Cross out all the multiples of the last number in primes.
Append the current number in numlist to the end of primes.
Cross out all the multiples of the **last** number in **primes**.
Append the current number in numlist to the end of primes.
Cross out all the multiples of the last number in primes.
An Algorithm for Sieve of Eratosthenes

**Input**: A number \( n \):

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

2. Create an empty list \( \textit{primes} \).

3. For each element in \( \textit{numlist} \)
   
   a. If element is not marked, copy it to the end of \( \textit{primes} \).

   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 \( n \)
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  
  `>`  greater than  
  `==`  equal to  
  `<=`  less than or equal to  
  `>=`  greater than or 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

```python
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).
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.
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

A Working Sieve

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

Helper function that we defined before

We could have used primes[len(primes)-1] instead.
We stopped at 11 because all the remaining entries must be prime since $11 \times 11 > 50$. 
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])
            sift(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.
Exercise

Write a function that returns how many of the three integers $n_1, n_2, \text{ and } n_3$ are odd:

```python
def num_odd(n1, n2, n3):
    cnt = 0  # initialize a counter
    if (n1 % 2 == 1):
        cnt = cnt + 1
    if (n2 % 2 == 1):
        cnt = cnt + 1
    if (n3 % 2 == 1):
        cnt = cnt + 1
    return cnt
```
Exercise

Write a function that prints whether die1 and die2 are doubles, cat's eyes (two 1's) or neither of these.

def print_doubles(die1, die2):
    if (die1 == 1 and die2 == 1):
        print("cat's eyes")
    elif (die1 == die2):
        print("doubles")
    else:
        print("not doubles")