Quiz Section Week 7
May 9, 2017

HMM calculations by hand

A few quick Python notes

Recursion

Random numbers

Remember Needleman-Wunsch: determine the best "hidden" evolutionary relationship between two sequences

Observed sequence

"Hidden" relationship to 2nd sequence

		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

- Alignment score for a position is a function of a previous alignment score and a "transition" score
- Find the path through the matrix that has the best score

Viterbi: determine the likeliest hidden state sequence for an observed sequence

Observed sequence

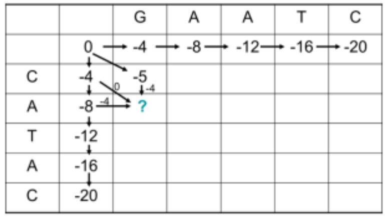
Hidden relationship to states

	A	A	T	T	T	A
A-rich	0.5*0.8= 0.4	0.9 *0.8 = ?				
T-rich	0.5*0.2= 0.1	0.1				

- Likelihood for an "alignment" of hidden state to observed sequence is a function of likelihood of **previous alignment** and **transition & emission probability**
- Find the path through this matrix that has the highest probability

Dynamic programming to find the best path for Needleman-Wunsch and Viterbi

DP in equation form



- Align sequence ***x*** and ***y***.
- ***F*** is the DP matrix; ***s*** is the substitution matrix; ***d*** is the linear gap penalty.

$$F(0,0) = 0$$

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) + d \\ F(i, j-1) + d \end{cases}$$

$\mathbf{x_j}$

$\boldsymbol{\pi_i}$

	A	A	T	T	T	A
A-rich	0.4	0.9 0.8 =. 288				
T-rich	0.1	0.1				

- "Align" observed sequence to state sequence

$$F(i,j) = \max \begin{cases} F(1,j\text{-}1)a(\boldsymbol{\pi_1}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ F(2,j\text{-}1)a(\boldsymbol{\pi_2}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ \\ \text{etc.} \end{cases}$$

Dynamic programming to find the best path for Needleman-Wunsch and Viterbi

DP in equation form

		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

- Align sequence ***x*** and ***y***.
- ***F*** is the DP matrix; ***s*** is the substitution matrix; ***d*** is the linear gap penalty.

$$F(0,0) = 0$$

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) + d \\ F(i, j-1) + d \end{cases}$$

$\overbrace{\hphantom{xxxxxxxxxxxx}}^{x_j}$

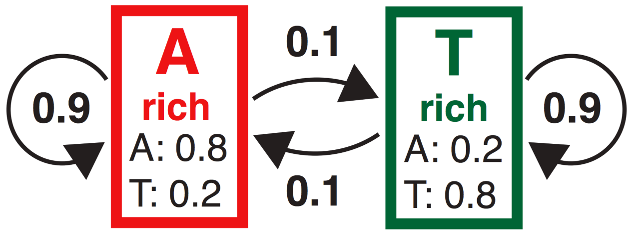
π_i

	A	A	T	T	T	A
A-rich	0.4	0.288				
T-rich	0.1					

- "Align" observed sequence to state sequence

$$F(i,j) = \max \begin{cases} F(1,j-1)a(\boldsymbol{\pi_1}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ F(2,j-1)a(\boldsymbol{\pi_2}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ \\ \text{etc.} \end{cases}$$

Dynamic programming to find the best path for Needleman-Wunsch and Viterbi



DP in equation form

		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

- Align sequence ***x*** and ***y***.
- ***F*** is the DP matrix; ***s*** is the substitution matrix; ***d*** is the linear gap penalty.

$$F(0,0) = 0$$

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) + d \\ F(i, j-1) + d \end{cases}$$

x_j

π_i	A	A	T	T	T	A
A-rich	0.4	0.288	…	…	…	.00001
T-rich	0.1	…	…	…	…	.0002

- "Align" observed sequence to state sequence

$$F(i,j) = \max \begin{cases} F(1,j\text{-}1)\,a(\boldsymbol{\pi_1}, \boldsymbol{\pi_i})\,e(x_j, \boldsymbol{\pi_i}) \\ F(2,j\text{-}1)\,a(\boldsymbol{\pi_2}, \boldsymbol{\pi_i})\,e(x_j, \boldsymbol{\pi_i}) \\ \\ \text{etc.} \end{cases}$$

Dynamic programming to find the best path for Needleman-Wunsch and Viterbi

DP in equation form

		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

- Align sequence ***x*** and ***y***.
- ***F*** is the DP matrix; ***s*** is the substitution matrix; ***d*** is the linear gap penalty.

$$F(0,0) = 0$$

$$F(i,j) = \max\begin{cases} F(i-1,j-1)+s(x_i,y_j) \\ F(i-1,j)+d \\ F(i,j-1)+d \end{cases}$$

x_j

π_i

	A	A	T	T	T	A
A-rich	0.4	.288	...	...	...	.00001
T-rich	0.1	...	...	...	...	.0002

- "Align" observed sequence to state sequence

$$F(i,j) = \max\begin{cases} F(1,j\text{-}1)a(\boldsymbol{\pi_1}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ F(2,j\text{-}1)a(\boldsymbol{\pi_2}, \boldsymbol{\pi_i})e(x_j, \boldsymbol{\pi_i}) \\ \\ \text{etc.} \end{cases}$$

For forward-backward, we account for all paths instead of just the best path

- What's the probability that the T at position 3 was emitted by the T-rich state?
- What's the probability that any path goes through the T-rich state at the third position?
- Combine all paths that pass through that position/state pair

k=3

i=T-rich

	A	A	T	T	T	A
A-rich	…	…	…	…	…	…
T-rich	…	…	$\boxed{…}$	…	…	…

$$P(\pi_i = k|x) = \frac{P(x, \pi_i = k)}{P(x)}$$

$$P(x, \pi_i = k) = \sum_{\pi_{i=k}} P(\pi|x)$$

Programming

Reminder: Tons of resources online for extra programming practice

- I still recommend this one:
  - http://interactivepython.org/runestone/static/thinkcspy/index.html

- You can use the help() function to learn about what other functions do, what arguments they need, etc.:

```
>>> help(len)
>>> help(open)
>>> my_list = []
>>> help(my_list.append)
```

A quick note on some useful list functions

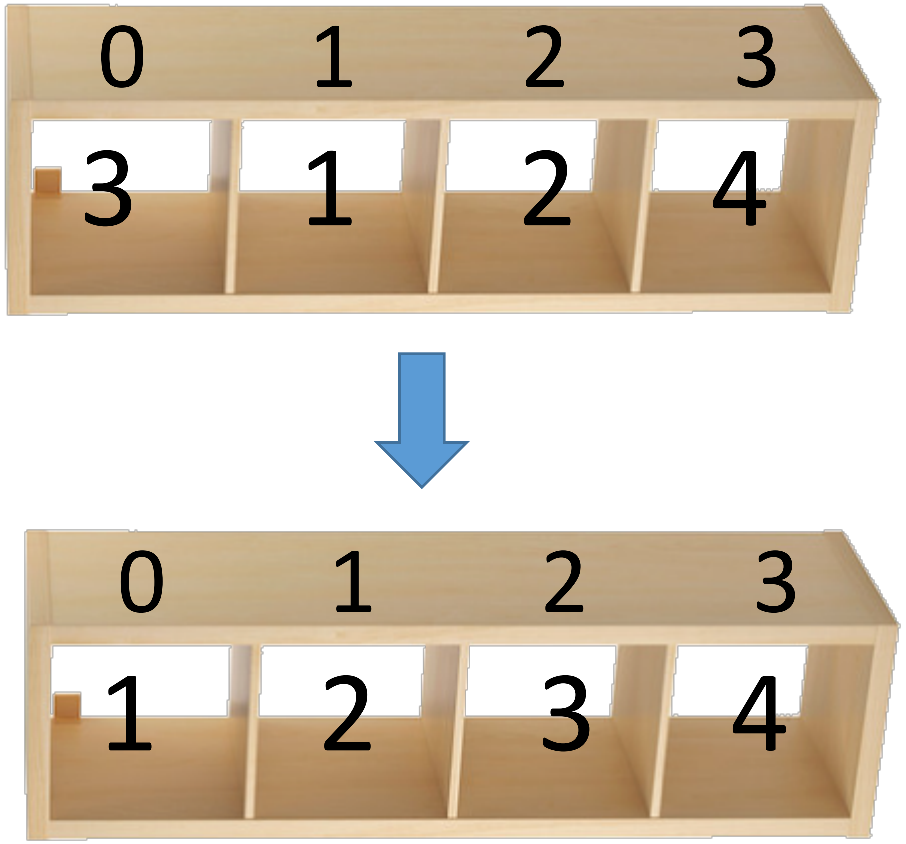
```
>>> my_list = [3,1,2]
>>> my_list[2]
>>> my_list[2:4]
>>> my_list[1:]
>>> my_list[-1]
```

Retrieve data

```
>>> my_list.append(4)
>>> my_list.remove(4)
>>> my_list.extend([4,5,6])
#compare with .append([4,5,6]
>>> my_list.sort()
>>> my_list.sort(reverse = True)
>>> my_list.pop()
```

Modify data

Both!



Structuring a program, keeping your code organized

- Remember last week we talked about how you can import functions from another script

intersection_function.py:

```
def my_intersection(list1, list2):
      new_list = []
      for x in list1:
        if x in list2:
            new_list.append(x)
      return new_list
```

calc_intersections.py:

```
#This line imports all function
#definitions from the file
#intersection_function.py
from intersection_function import *
#This runs all the code in
intersection_function.py!

list_num =
my_intersection([1,3,5,21],[5,4,19,21])
print list_num
```

We can put code to test or apply defined functions in its own section

Recall HW4

```
from __future__ import division

def euclidean_distance(point1, point2):
	#Calculate and return Euclidean distance here
	return(dist)
def manhattan_distance(point1, point2):
	#Calculate and return Manhattan distance here
	return sum_dist
```

Recursion

- Doug mentioned on Friday that the Forward-Backward algorithm is a *recursive* algorithm

- What does that mean?

Here's a puzzle: how to calculate the sum of a list (of any length) without for and while loops?

```
def sumList(list1):
    #List sum calculation here


my_list = [0,5,3,4,8]
```

Hint: One way to show this mathematically

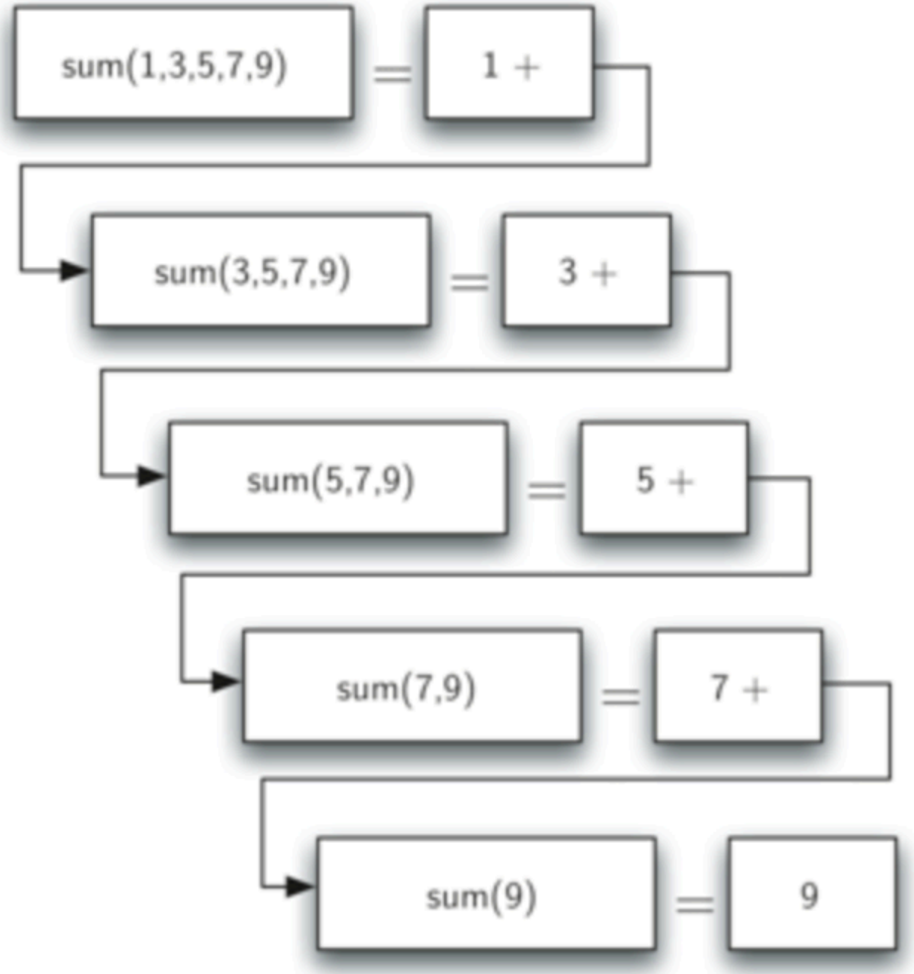
sum = (0 + (5 + (3 + (4 + (8)))))

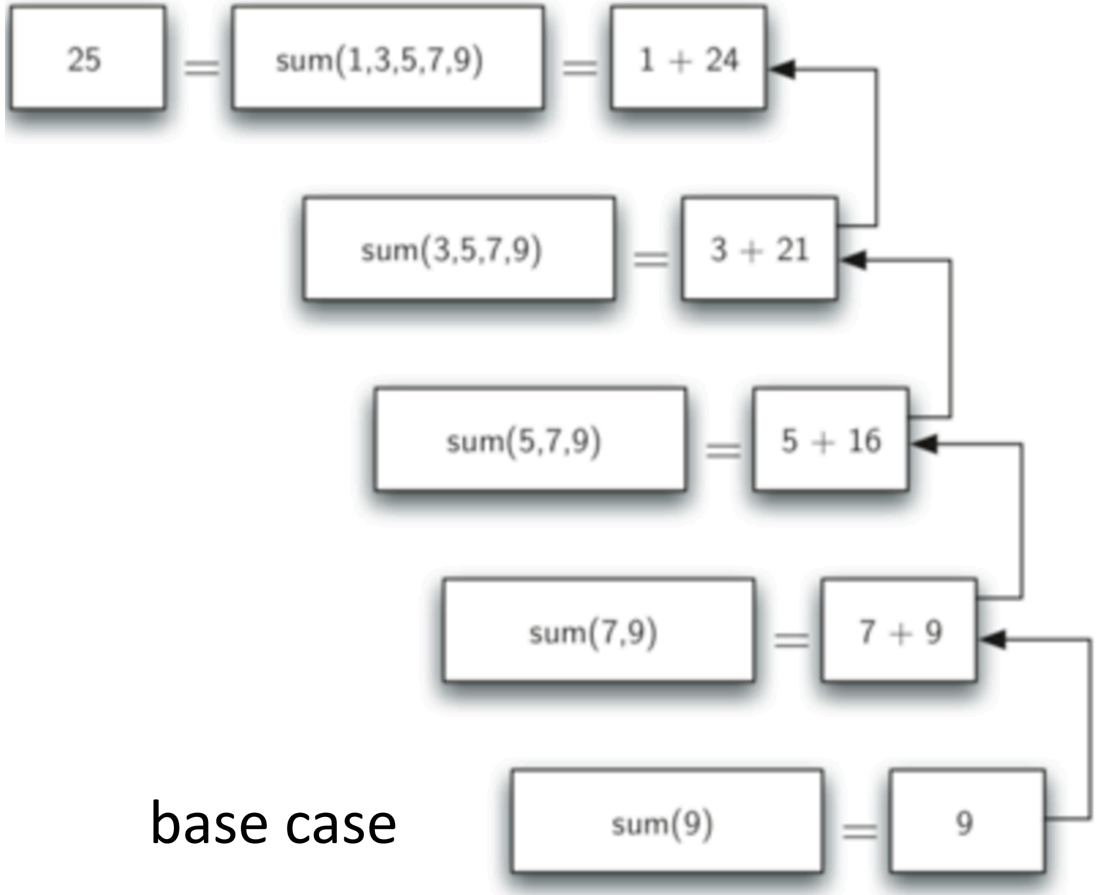
Recursively calculating the sum of a list

```
def sumList(list1):
    if len(list1) > 1:
        return list1[0] + listsum(list1[1:])
    else:
        return list1[0]
```

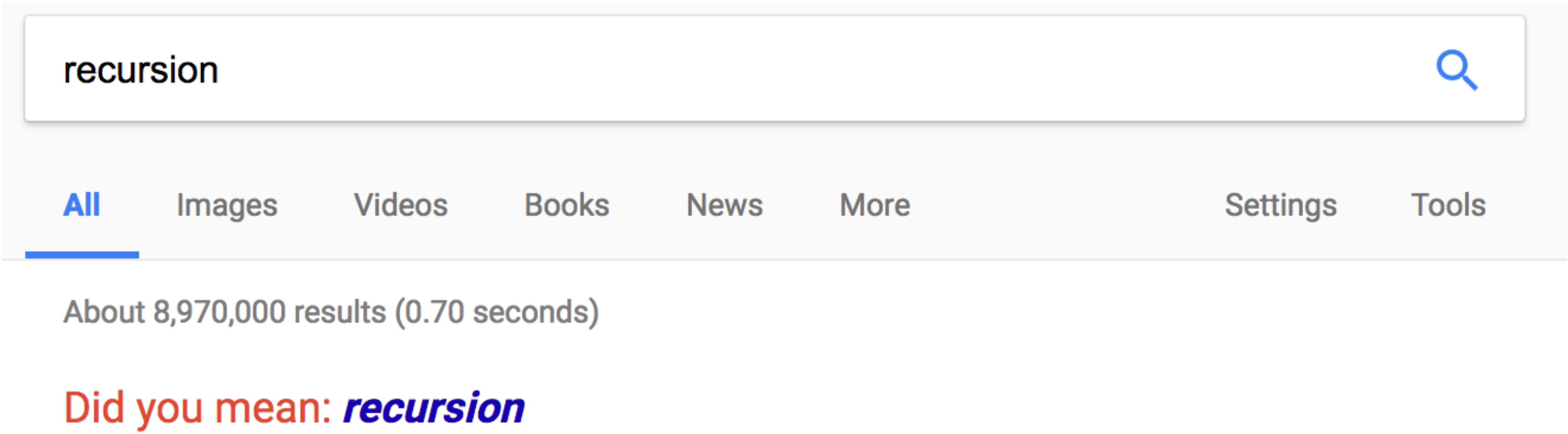
A *recursive* algorithm refers to itself – it calls itself iteratively *until reaching a base case*

In what order does our sum_list function actually run?





A bad computer scientist joke



What's wrong with this recursive "algorithm"?

Write a recursive function to calculate a factorial

```
def factorial(num):
    if ##something:
        #recursion
    else:
        # base case
    return prod
```

Write a recursive function to calculate a factorial

```
def factorial(num):
    if num > 1:
        prod = num*factorial(num - 1)
    else:
        prod = num
    return prod
```

In what way(s) is the forward-backward algorithm recursive?

We build on all the forward-backward probabilities

$$P(\pi_i = k|x) = \frac{P(x, \pi_i = k)}{P(x)} = \frac{f_{k,i} * b_{k,i}}{P(x)}$$

$$f_{k,i} = e_k(x_i) \sum_l f_{l,i-1} a_{lk} \qquad b_{k.i} = \sum_l e_l(x_{i+1}) b_{l,i+1} a_{kl}$$

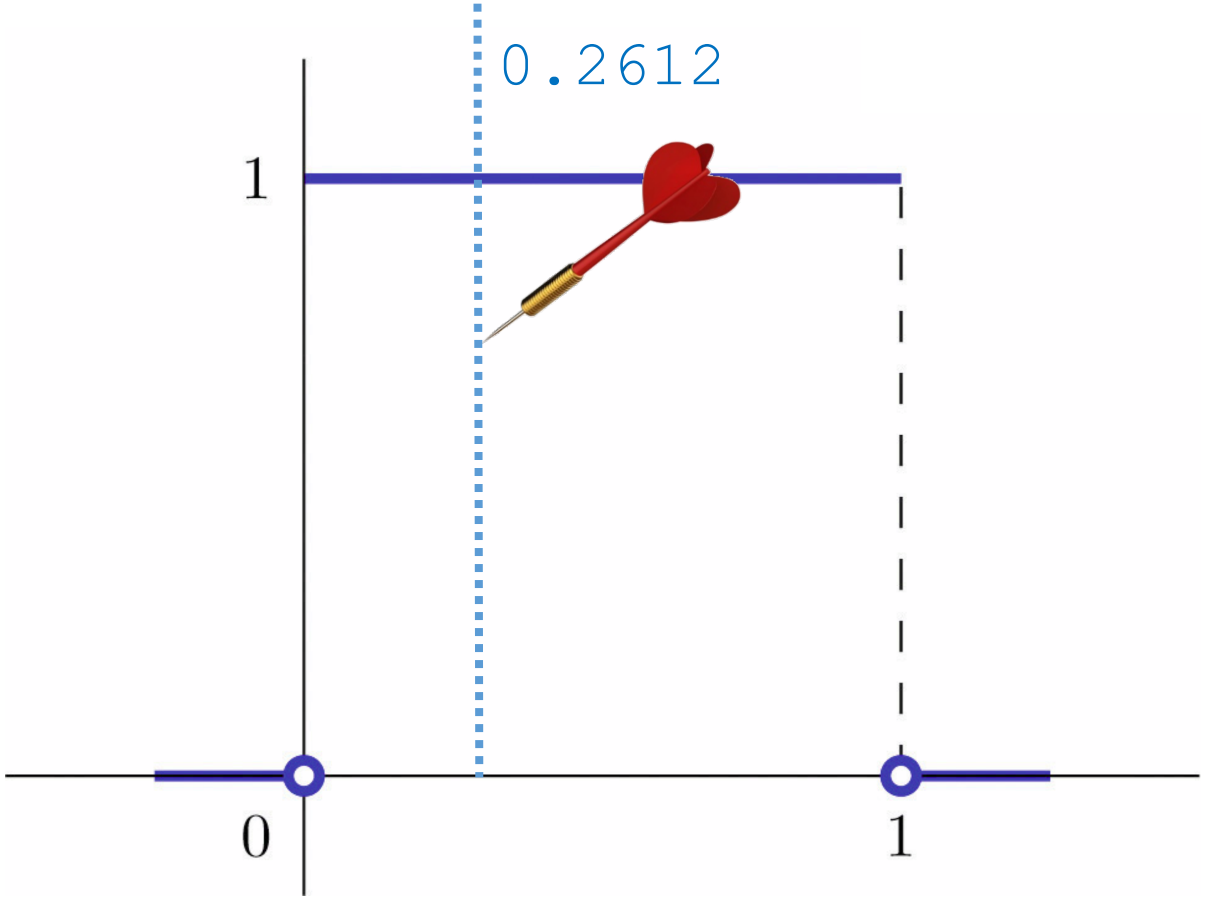
Generating random numbers in Python

What are some situations where you'd want to generate random numbers?

In-class examples?

- Generating random sequences to create null distribution for sequence alignment
- A Markov chain that changes states probabilistically

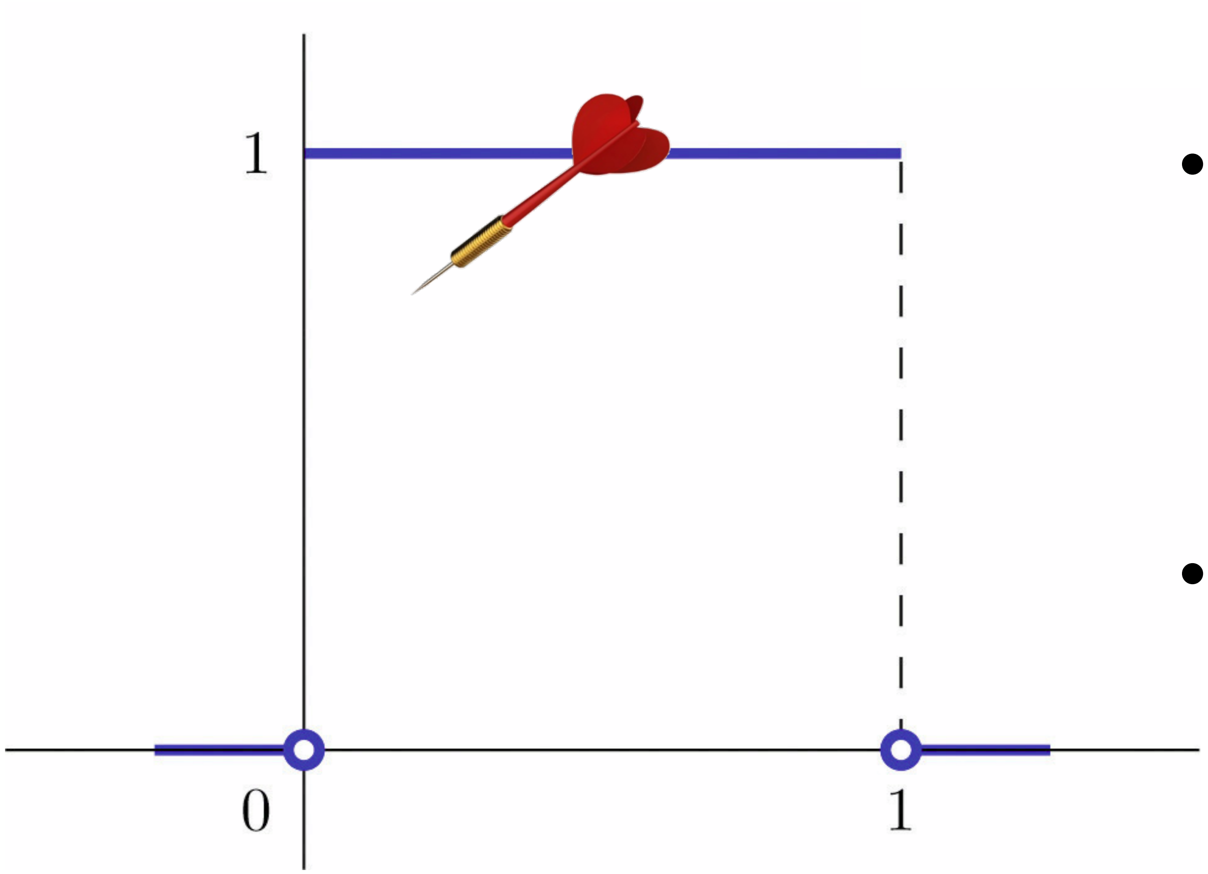
random() returns a uniformly distributed random value between 0 and 1



- How can you convert this into a random coin flip with heads or tails?

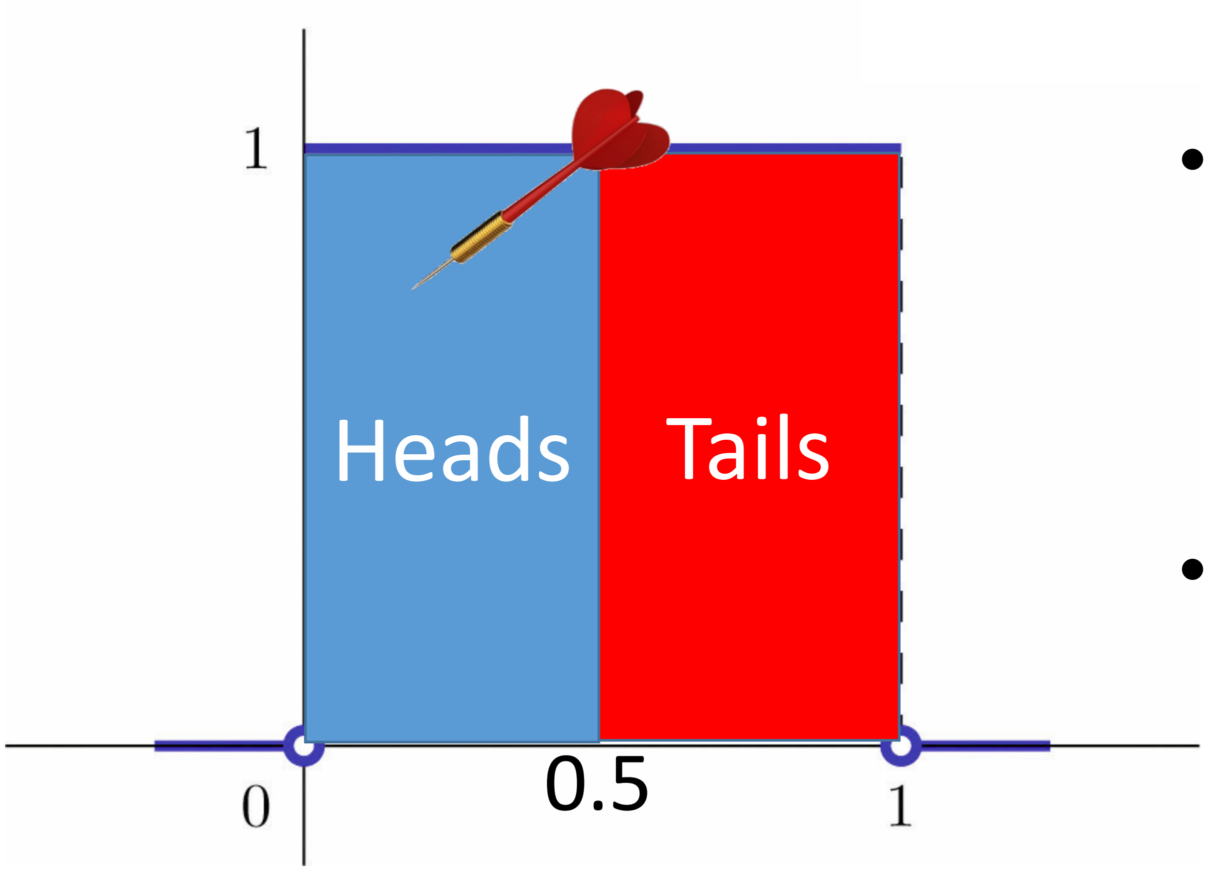
```
import random
r = random.random()
print r
0.261256363123
```

random() returns a uniformly distributed random value from [0,1)



- How can you convert this into a random coin flip with heads or tails?
- Throw a dart, call heads if dart lands between 0 and 0.5, tails if between 0.5 and 1

random() returns a uniformly distributed random value between 0 and 1



- How can you convert this into a random coin flip with heads or tails?
- Throw a dart, call heads if dart lands between 0 and 0.5, tails if between 0.5 and 1

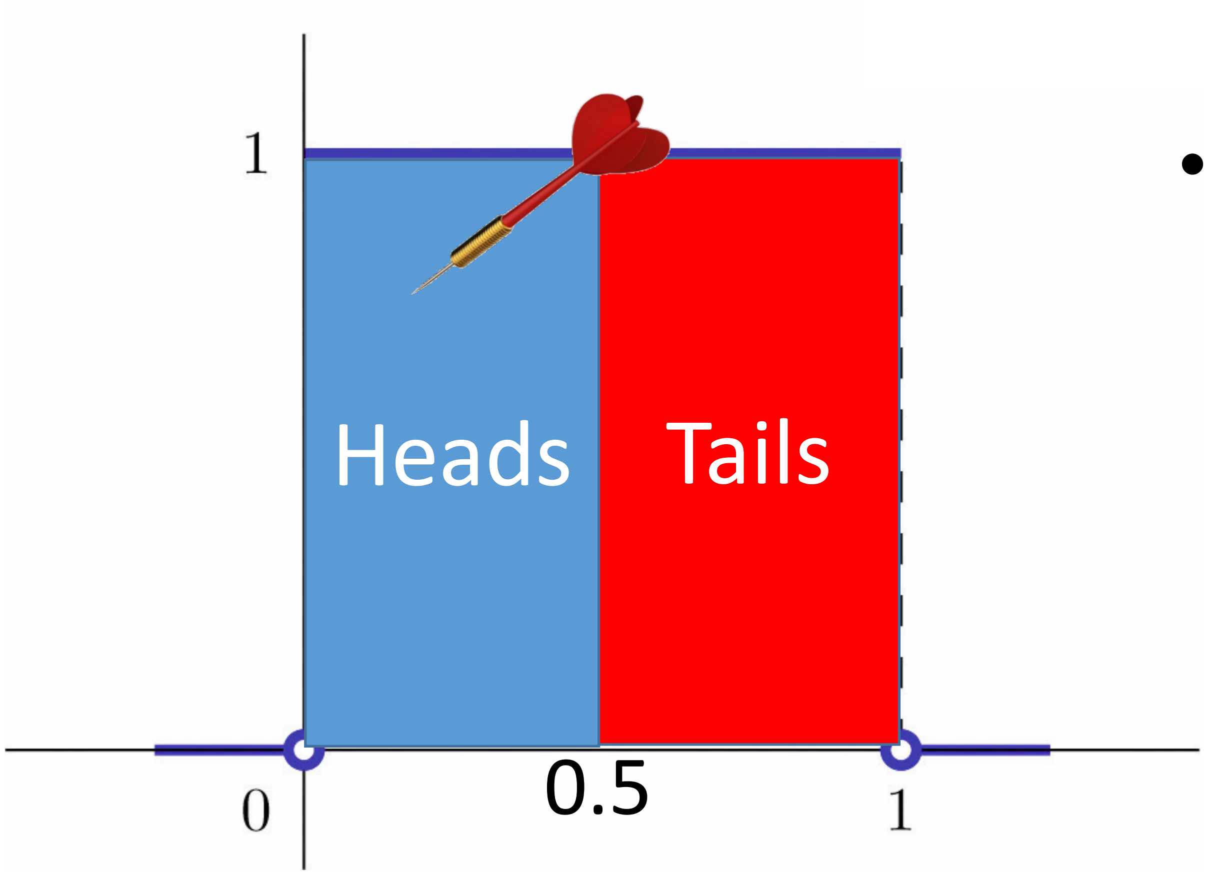
Exercise: write a function to simulate a coin flip using random()

```
import random
# return ‘heads’ or ‘tails’ with 50/50 odds
def coinflip():
```

Exercise: write a function to simulate a coin flip using random()

```
import random
# return heads or tails
def coinflip():
    v = random()
    if f > 0.5:
        return 'Tails'
    else:
        return 'Heads'
```

random() returns a uniformly distributed random value between 0 and 1



- How can you convert this into a die roll?

Exercise: write a function to simulate a die roll using random()

```
import random
# return 1,2,3,4,5, or 6 with equal odds
def dieroll():
```

Randomly shuffling a sequence of letters

How would you generate a random permutation of this sequence?

ATCGTCCTTAAGGATTACCATTTGGCCTAGA

Randomly shuffling a sequence of letters

How would you generate a random permutation of this sequence?