Genome Informatics

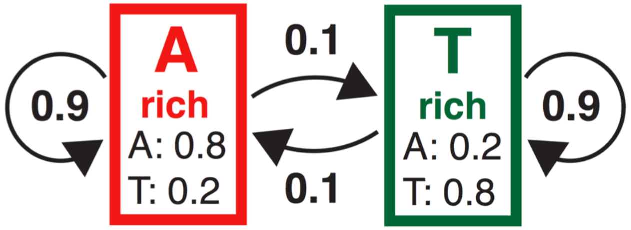
Quiz section 7

May 10, 2018

Housekeeping

- Read assignments carefully!
- Tuesday

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



Observed sequence

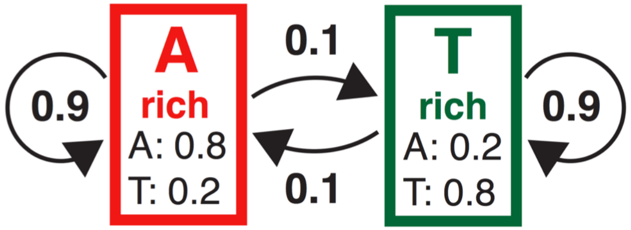
Hidden relationship to states

	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				

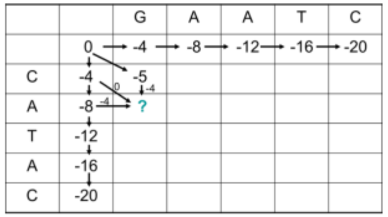
- Likelihood for an “alignment” of hidden state to observed sequence is a function of likelihood of **previous alignment** and **transition & emission probability**

- 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

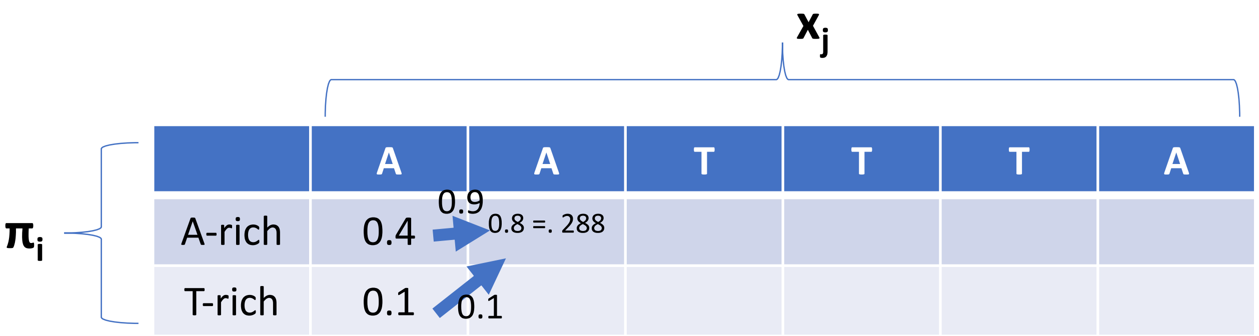


		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

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

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

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

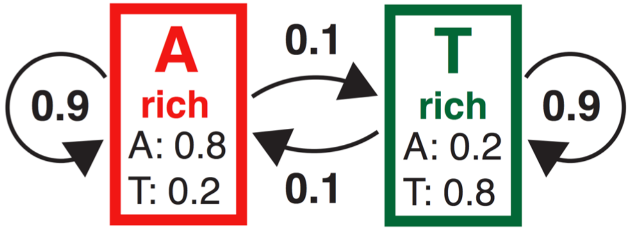


	A	A	T	T	T	A
A-rich	0.4					
T-rich	0.1					

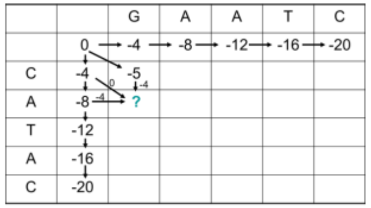
- “Align” observed sequence to state sequence

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

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.

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

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

$\mathbf{x_j}$

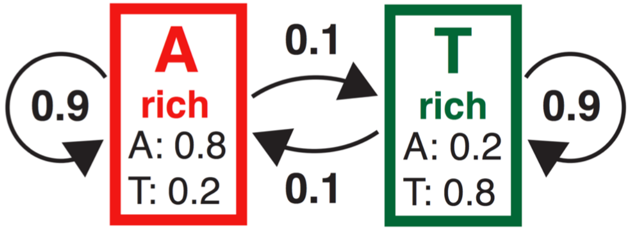
$\boldsymbol{\pi_i}$

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

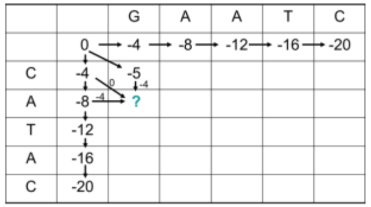
- “Align” observed sequence to state sequence

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

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



DP in equation form

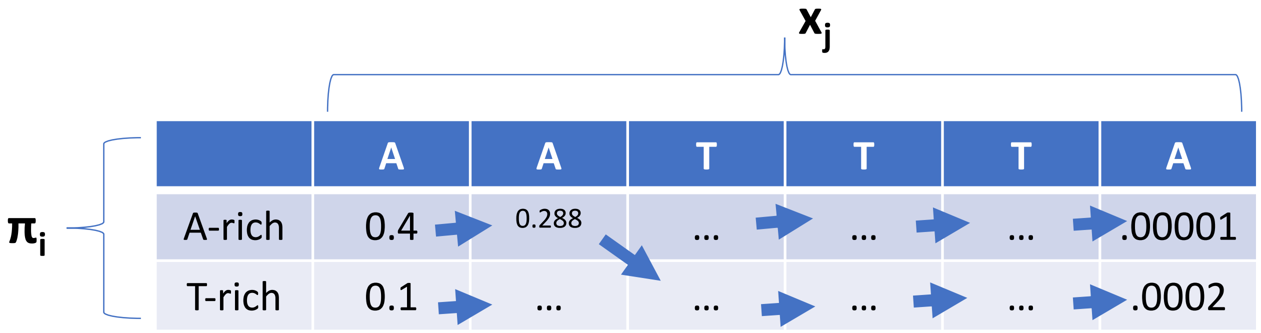


		G	A	A	T	C
	0	-4	-8	-12	-16	-20
C	-4	-5				
A	-8	?				
T	-12					
A	-16					
C	-20					

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

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

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

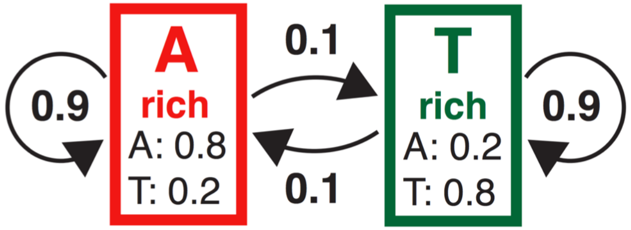


	A	A	T	T	T	A
A-rich	0.4	0.288	…	…	…	.00001
T-rich	0.1	…	…	…	…	.0002

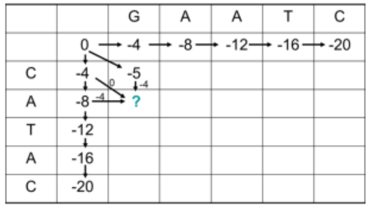
- “Align” observed sequence to state sequence

$$F(i,j) = \max \begin{cases} F(1,j\text{-}1)a(\boldsymbol{\pi}_1, \boldsymbol{\pi}_i)e(x_j, \boldsymbol{\pi}_i) \\ F(2,j\text{-}1)a(\boldsymbol{\pi}_2, \boldsymbol{\pi}_i)e(x_j, \boldsymbol{\pi}_i) \\ \\ \text{etc.} \end{cases}$$

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



DP in equation form

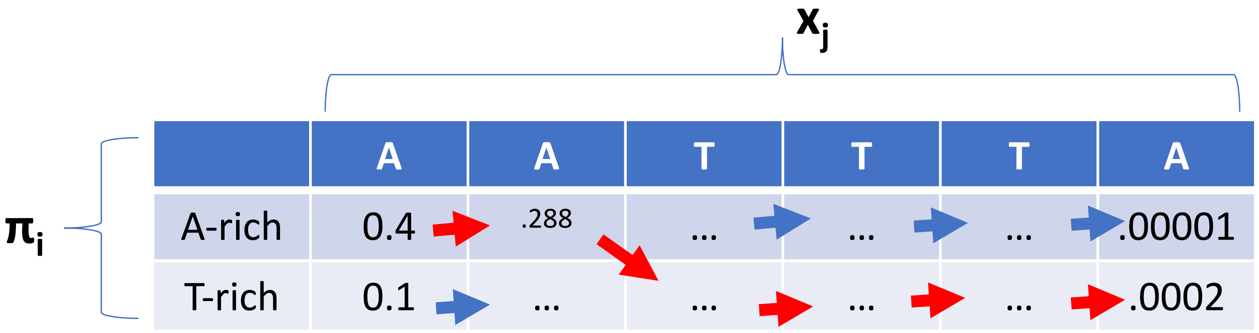


		G	A	A	T	C
	0 →	-4 →	-8 →	-12 →	-16 →	-20
C	-4	-5				
A	-8 →	?				
T	-12					
A	-16					
C	-20					

- 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

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

Programming

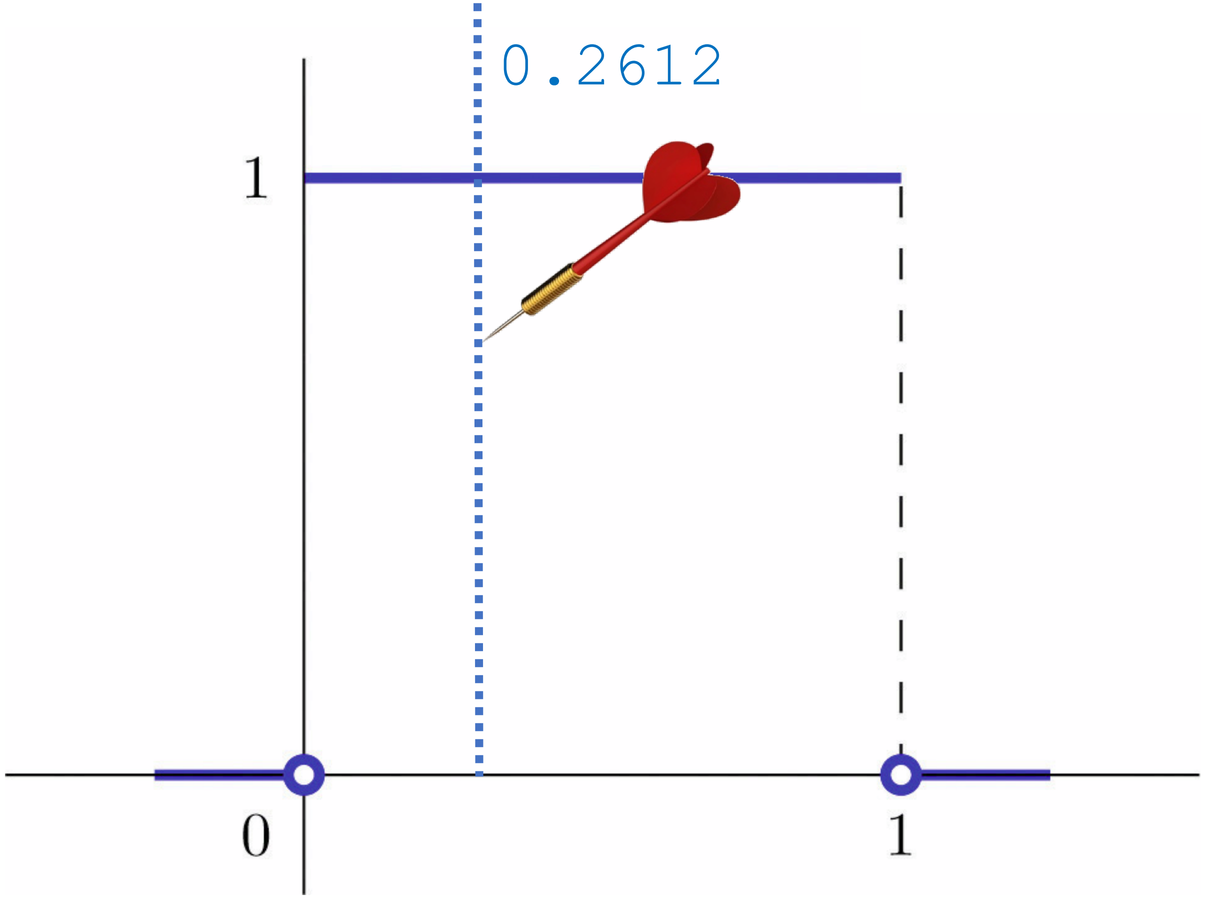
Generating random numbers in Python

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



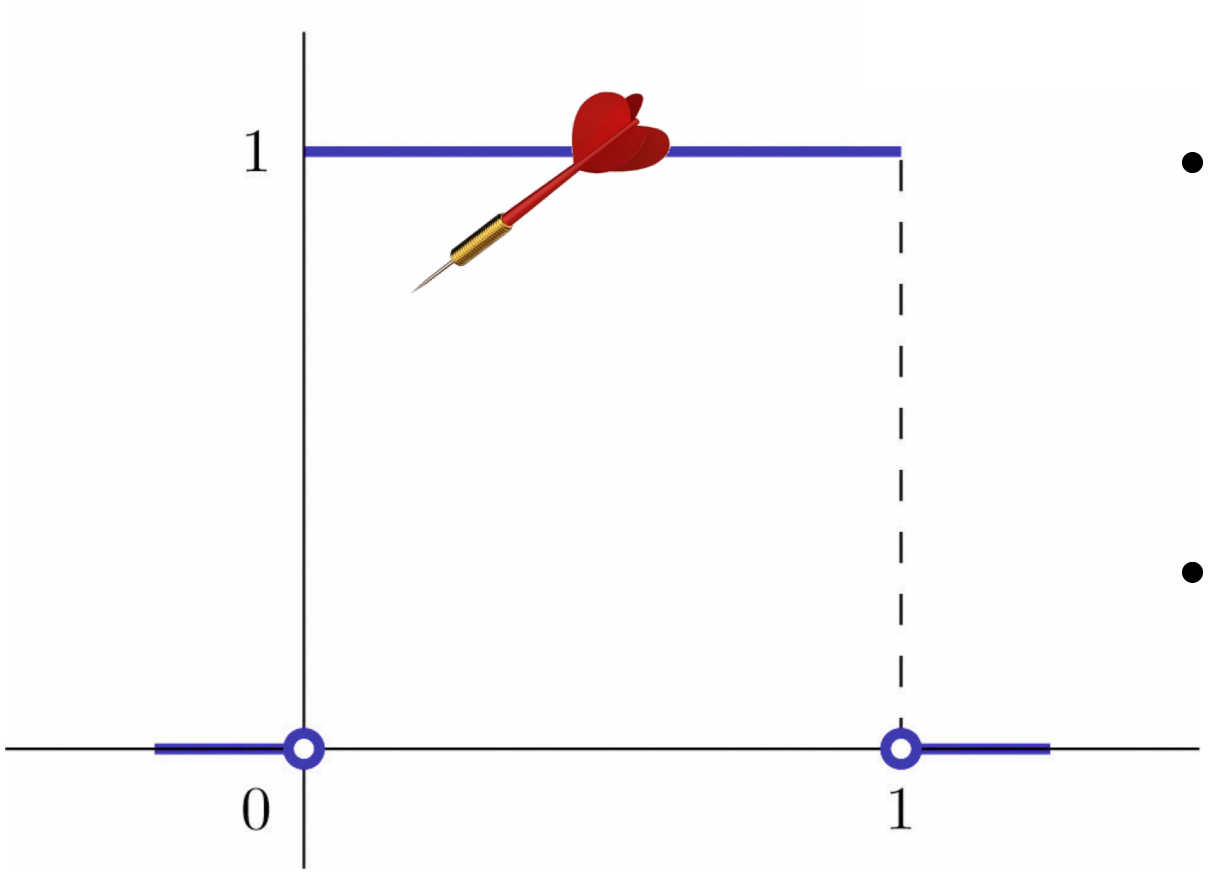
- How can you convert this into a random coin flip with heads or tails?

```
import random
r = random.random()
print r
0.26125636123
```

*Not actually random!

This is actually a pseudorandom number generator – it's *approximates* random number generation based on a starting point – a seed. If you want to reproducibly produce the same "random" set of numbers twice, you can set the seed with random.seed(100)

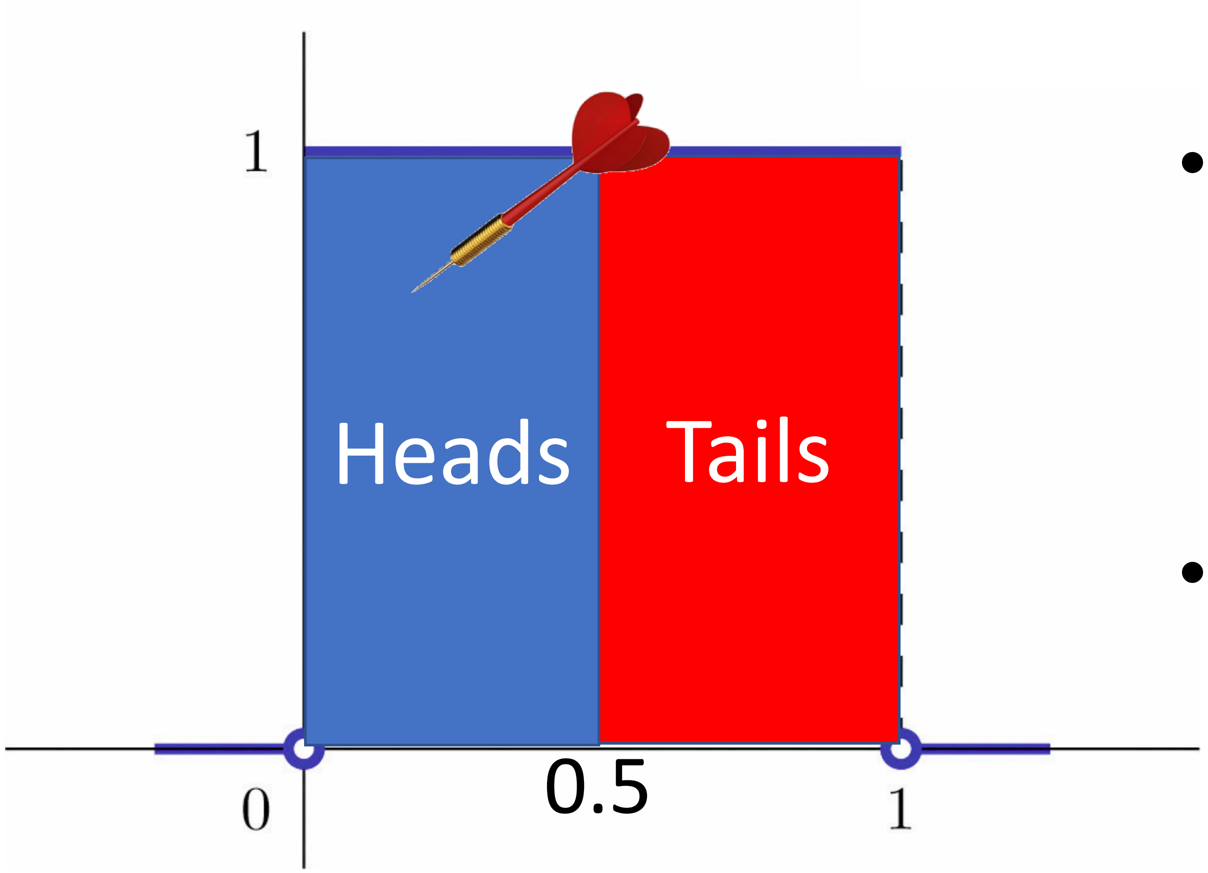
random() returns a uniformly distributed random value from [0,1)



- 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

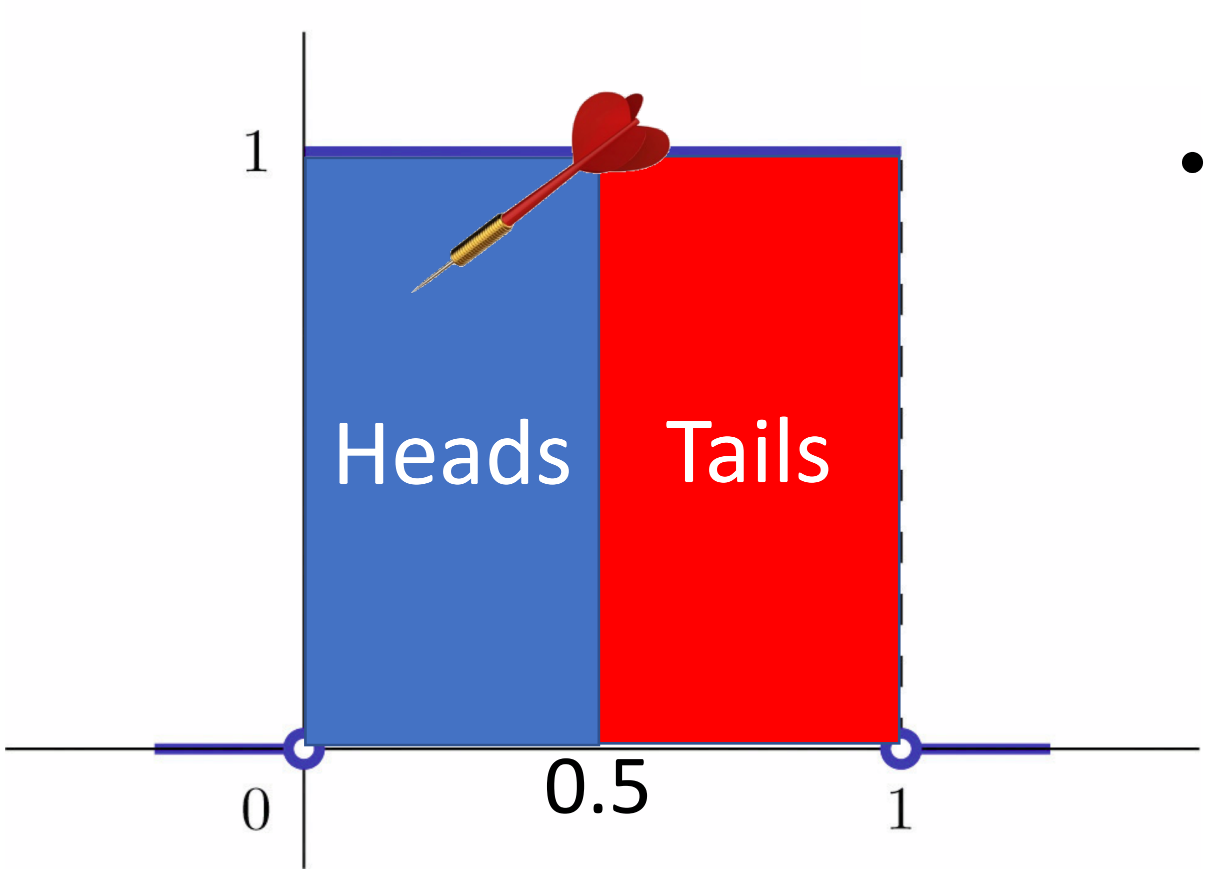
Exercise: write a function to simulate a coin flip using random()

```
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?

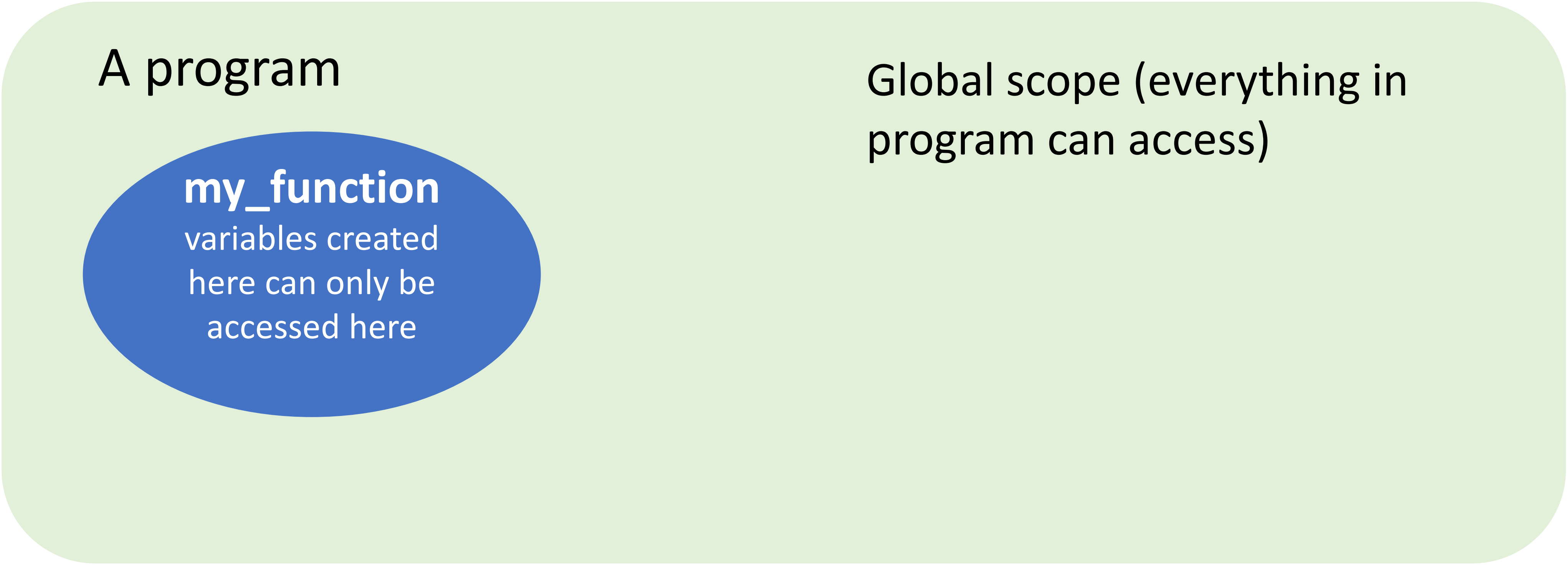
Exercise: write a function to simulate a die roll using random()

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

The nitty gritty of scope and functions

Scope of a variable

- Variables created in the main part of your program can be accessed anywhere (**global** scope)
- Variables created within functions are only accessible within that function (**local** scope)



A program

Global scope (everything in program can access)

my_function

variables created here can only be accessed here

Scope of a variable

```
new_list = [0,1,2]

def less_than(myList, num = 4):
     new_list = []
     for x in myList:
          if x < num:
               new_list.append(x)
     return new_list

print new_list
anotherList = [3,7,12]
print less_than(anotherList)
```

Scope of a variable

```
new_list = [0,1,2]

def less_than(myList, num = 4):
     #new_list = []
     for x in myList:
          if x < num:
               new_list.append(x)
     return new_list

print new_list
anotherList = [3,7,12]
print less_than(anotherList)
```

Don't do this!! You'll confuse yourself

Define all your functions at the beginning of your program or in another file

Returning values

- Check the following function:

```
# This function …
# …
def CalcSum(a_list):
    sum = 0
    for item in a_list:
        sum += item
    return sum
```

- What does this function do?

From Elhanan

Returning values

- Check the following function:

```
# This function calculates the sum
# of all the elements in a list
def CalcSum(a_list):
    sum = 0
    for item in a_list:
        sum += item
    return sum
```

- What does this function do?

```
>>> my_list = [1, 3, 2, 9]
>>> print CalcSum(my_list)
15
```

Returning more than one value

- Let’s be more ambitious:

```
# This function calculates the sum
# AND the product of all the
# elements in a list
def CalcSumAndProd(a_list):
    sum = 0
    prod = 1
    for item in a_list:
        sum += item
        prod *= item
    return ???
```

- How can we return both values?

Returning more than one value

- We can use a list as a return value:

```
# This function calculates the sum
# AND the product of all the
# elements in a list
def CalcSumAndProd(a_list):
    sum = 0
    prod = 1
    for item in a_list:
        sum += item
        prod *= item
    return [sum, prod]
```

```
>>> my_list = [1, 3, 2, 9]
>>> print CalcSumAndProd(my_list)
[15, 54]
```

```
>>> res = CalcSumAndProd(my_list)
```

List assignment

```
>>> [s,p] = CalcSumAndProd(my_list)
```

multiple assignment

Returning lists

- An increment function:

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    new_list = []
    for item in a_list:
        new_list.append(item+1)
    return new_list

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
print my_list
my_incremended_list = incrementEachElement(my_list)
Print my_incremended_list
```

```
[1, 20, 34, 8]
[2, 21, 35, 9]
```

- Is this good practice?

Returning lists

- An increment function (modified):

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    new_list = []
    for item in a_list:
        new_list.append(item+1)
    return new_list

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
print my_list
my_list = incrementEachElement(my_list)
Print my_list
```

```
[1, 20, 34, 8]
[2, 21, 35, 9]
```

- What about this?

Returning lists

- What will happen if we do this?

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
print my_list
incrementEachElement(my_list)
print my_list
```

- **(note: no return value!!!)**

Returning lists

- What will happen if we do this?

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1

# Now, create a list and use the function
my_list = [1, 20, 34, 8]
print my_list
incrementEachElement(my_list)
print my_list
```

- **(note: no return value)**

```
[2, 21, 35, 9]
[2, 21, 35, 9]
```

WHY IS THIS WORKING?

Pass-by-reference vs. pass-by-value

- Two fundamentally different function calling strategies:
- **Pass-by-Value:**
  - The value of the argument is copied into a local variable inside the function
  - C, Scheme, C++
- **Pass-by-reference:**
  - The function receives an implicit reference to the variable used as argument, rather than a copy of its value
  - Perl, VB, C++
- **So, how does Python pass arguments?**

Python passes arguments by reference

(almost)

- So … this will work!

```
# This function increment every element in
# the input list by 1
def incrementEachElement(a_list):
    for index in range(len(a_list)):
        a_list[index] +=1
```

```
>>> my_list = [1, 20, 34, 8]
>>> incrementEachElement(my_list)
>>> my_list
[2, 21, 35, 9]
>>> incrementEachElement(my_list)
>>> my_list
[3, 22, 36, 10]
```

Python passes arguments by reference

(almost)

- How about this?

```
def addQuestionMark(word):
    print “word inside function (1):”, word
    word = word + “?”
    print “word inside function (2):”, word

my_word = “really”
addQuestionMark(my_word)
print “word after function:”, my_word
```

Python passes arguments by reference

(almost)

- How about this?

```
def addQuestionMark(word):
    print “word inside function (1):”, word
    word = word + “?”
    print “word inside function (2):”, word

my_word = “really”
addQuestionMark(my_word)
print “word after function:”, my_word
```

```
word inside function (1): really
word inside function (2): really?
word after function: really
```

- Remember:
  1. Strings/numbers are immutable
  2. The assignment command often creates a new object

Passing by reference: the bottom line

- **You can (and should) use this option when**:
  - Handling large data structures
  - “In place” changes make sense

- **Be careful** (a double-edged sword):
  - Don’t lose the reference!
  - Don’t change an argument by mistake

- When we learn about objects and methods we will see yet an additional way to change variables

Required Arguments

- How about this?

```
def printMulti(text, n):
    for i in range(n):
        print text
```

```
>>> printMulti(“Bla”,4)
Bla
Bla
Bla
Bla
```

- What happens if I try to do this:

```
>>> printMulti("Bla")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: printMulti() takes exactly 2
arguments (1 given)
```

Default Arguments

- Python allows you to define defaults for various arguments:

```
def printMulti(text, n=3):
    for i in range(n):
        print text
```

```
>>> printMulti(“Bla”,4)
Bla
Bla
Bla
Bla
```

```
>>> printMulti(“Yada”)
Yada
Yada
Yada
```

Default Arguments

- This is very useful if you have functions with numerous arguments/parameters, most of which will rarely be changed by the user:

```
def runBlast(fasta_file, costGap=10, E=10.0, desc=100,
   max_align=25, matrix=“BLOSUM62”, sim=0.7, corr=True):
   <runBlast code here>
```

- You can now simply use:

```
>>> runBlast(“my_fasta.txt”)
```

- Instead of:

```
>>> runBlast(“my_fasta.txt”,10,10.0,100,25,“BLOSUM62”,0.7,
True)
```

Keyword Arguments

- You can still provide values for specific arguments using their label:

```
def runBlast(fasta_file, costGap=10, E=10.0, desc=100,
  max_align=25, matrix=“BLOSUM62”, sim=0.7, corr=True):
    <runBlast code here>
    …



>>> runBlast(“my_fasta.txt”, matrix=“PAM40”)
```