Data structures pt II: Dictionaries

Lesson 5 – 8/15/18

Slide by Sarah Middleton

(Please sign-in)

Sponsored by:

Previously on GCB Bootcamp

- Introduction to lists
- Processing files using .split()

Today's schedule

1. File writing
2. Dictionaries

1. File writing

File writing

Opening an output file is almost identical to input, with a small difference:

```
var = open(fileName, 'w')
```

Example:

```
outFile = open("seqs.txt", 'w')
```

File writing

Opening an output file is almost identical to input, with a small difference:

```
var = open(fileName, 'w')
```

Important: opening a file in 'w' mode will overwrite the file if it already exists.

Example:

```
outFile = open("seqs.txt", 'w')
```

Writing to an output file

Once the output file is opened, we use:

```
var.write(someStr)
```

Example:

```
outFile.write("This is output!\n")
```

Writing to an output file

Once the output file is opened, we use:

```
var.write(someStr)
```

Example:

```
outFile.write("This is output!\n")
```

Don't forget the newline!
Unlike `print`, `.write()` does
not insert this for you.

Simple example

Code

```
fileName = "output.txt"
outFile = open(fileName, 'w')
outFile.write("This is me,")
outFile.write("printing to \n a file.")
outFile.close()
```

output.txt

```
This is me,printing to 
 a file.
```

Note the spacing and newline

Only strings can be printed

Code

```
fileName = "output.txt"
outFile = open(fileName, 'w')
outFile.write(25)
outFile.close()
```

Error:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    outFile.write(25)
TypeError: expected a character buffer object
```

Only strings can be printed

Code

```
fileName = "output.txt"
outFile = open(fileName, 'w')
outFile.write(str(25))
outFile.close()
```

A simple fix.

output.txt

```
25
```

Reading and writing can be done at the same time (as long as it's to different files)

Code

```
infile = "genes.txt"
outfile = "output.txt"
inFile = open(infileName, 'r')
outFile = open(outfileName, 'w')
for line in inFile:
    line = line.rstrip('\n')
    outFile.write("Found " + line + "\n")
outFile.close()
inFile.close()
```

genes.txt

```
uc007zzs.1
uc009akk.1
uc009eyb.1
uc008wzq.1
uc007hnl.1
```

output.txt

```
Found uc007zzs.1
Found uc009akk.1
Found uc009eyb.1
Found uc008wzq.1
Found uc007hnl.1
```

2. Dictionaries

Lists vs Dictionaries

Two main differences:

1. You retrieve elements from a dictionary using a "key", rather than an index
2. Dictionaries are unordered

1. Indexing by keys

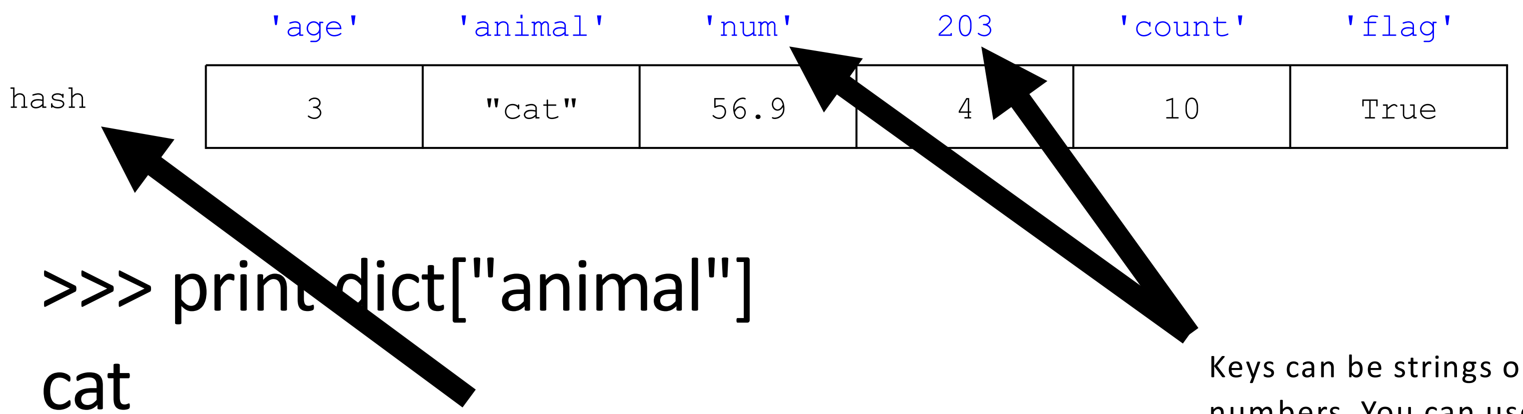
A dictionary is similar to a list, except instead of accessing elements by their index, you access them by a name ("key") that you pick.

	'age'	'animal'	'num'	203	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

```
>>> print dict["animal"]
cat
```

1. Indexing by keys

A dictionary is similar to a list, except instead of accessing elements by their index, you access them by a name ("key") that you pick.



```
>>> print dict["animal"]
cat
```

Dictionaries are similar to what other languages call "hash tables". So I might call them that sometimes.

Keys can be strings or numbers. You can use single quotes or double quotes around the strings; doesn't matter

2. Unordered

Lists are all about keeping elements in some order. Though you may change the ordering from time to time, it's still in *some* order.

You should think of **dictionaries** more like magic grab bags. You mark each piece of data with a key, then throw it in the bag. When you want that data back, you just tell the bag the key and it spits out the data assigned to that key.

2. Unordered

Lists are all about keeping elements in some order. Th

from time der.

Technicality:

Ok, so in reality, there *is* an order to your dictionary. But it is an order that Python picks that obeys complex rules and is essentially unpredictable by us. So for all intents and purposes, it may as well be unordered. Don't worry about it too much... just treat it like a magic grab bag and all will be well.

You shoul like magic grab bags ta with a key, then u want that data back, you just tell the bag the key and it spits out the data assigned to that key.

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash['count']
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash['count']
```

Result:

```
10
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash['num']
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash['num']
```

Result:

```
56.9
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash[age]
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
print hash[age]
```

Result:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'age' is not defined
```

(we didn't put quotes around "age")

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
var = 'animal'
print hash[var]
```

Practice with dictionary keys

	'age'	'animal'	'num'	205	'count'	'flag'
hash	3	"cat"	56.9	4	10	True

What will this code print?

```
var = 'animal'
print hash[var]
```

Result:

```
cat
```

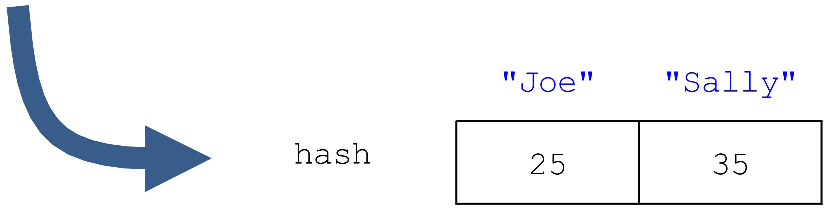
Creating a dictionary

Create an empty dictionary:

```
hash = {}
```

Create a dictionary with elements:

```
hash = {"Joe": 25, "Sally": 35}
```



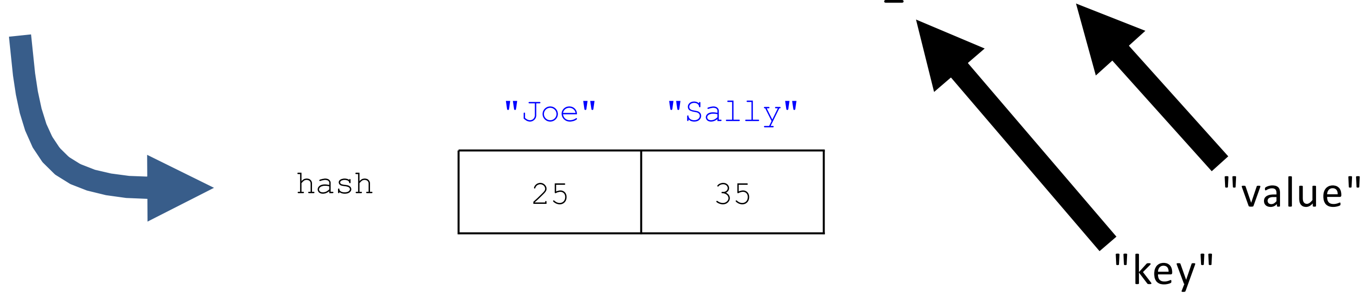
Creating a dictionary

Create an empty dictionary:

```
   hash = {}
```

Create a dictionary with elements:

```
   hash = {"Joe": 25, "Sally": 35}
```



Adding to a dictionary

Add entry:

```
hash[newKey] = newVal
```

Example:

```
>>> hash = {}
>>> hash["Joe"] = 25
>>> hash["Bob"] = 39
>>> print hash
{'Bob': 39, 'Joe': 25}
```

Adding to a dictionary

Add entry:

```
hash[newKey] = newVal
```

Example:

```
>>> hash = {}
>>> hash["Joe"] = 25
>>> hash["Bob"] = 39
>>> print hash
{'Bob': 39, 'Joe': 25}
```

Note that Python printed them in a different order than we entered them.

Removing from a dictionary

Delete entry:

```
    del hash[existingKey]
```

Example:

```
>>> hash = {"name": "Joe", "age": 35, "job": "plumber"}
>>> print hash
{'job': 'plumber', 'age': 35, 'name': 'Joe'}

>>> del hash["age"]
>>> print hash
{'job': 'plumber', 'name': 'Joe'}
```

Removing from a dictionary

Delete entry:

```
del hash[existingKey]
```

Example:

```
>>> hash = {                                                mber"}
>>> print ha
{'job': 'plu

>>> del hash
>>> print ha
{'job': 'plu
```

Side note:

You can use the **.pop()** method to exact the same effect; this additionally returns the removed value

```
>>> hash = {
...         "name": "Joe",
...         "age":  35,
...         "job":  "plumber"
... }
>>> age = hash.pop("age")
>>> print age
35
```

Phonebook example

Code:

```
phonebook = {}
phonebook["Joe Shmo"] = "958-273-7324"
phonebook["Sally Shmo"] = "958-273-9594"
phonebook["George Smith"] = "253-586-9933"

name = raw_input("Lookup number for: ")
print phonebook[name]
```

Output example:

```
Lookup number for: <we enter>Sally Shmo
958-273-9594
```

Phonebook example

Code:

```
phonebook = {}
phonebook["Joe Shmo"] = "958-273-7324"
phonebook["Sally Shmo"] = "958-273-9594"
phonebook["George Smith"] = "253-586-9933"

name = raw_input("Lookup number for: ")
print phonebook[name]
```

Notice that we can store the name of a key in a variable, and then use that variable to access the desired element. In this case, name holds the name that we input in the terminal, Sally Shmo.
What would happen if we entered a name that was not in the phonebook?

Output example:

```
Lookup number for: <we enter>Sally Shmo
958-273-9594
```

Checking if something is in the dict

This is the same as with a list. Use `in`:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

if "Joe" in ages:
    print "Yes, Joe is in the dictionary"
else:
    print "No, Joe is not in the dictionary"
```

Result:

```
Yes, Joe is in the dictionary
```

Dictionary methods

Here are some useful dictionary methods:

- `dict.keys()` - returns a **list** of the keys only
- `dict.values()` - returns a **list** of the values only
- `dict.items()` - returns a **list** of key-value pairs

Example:

```
>>> colors = {"apple": "red", "banana": "yellow", "grape": "purple"}
>>> colors.keys()
['grape', 'apple', 'banana']
>>> print colors.values()
['purple', 'red', 'yellow']
>>> print colors.items()
[('grape', 'purple'), ('apple', 'red'), ('banana', 'yellow')]
```

Using `.keys()`

Code:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for name in ages.keys():
    print name, "is in the dictionary."
```

Output:

```
Sally is in the dictionary.
Joe is in the dictionary.
George is in the dictionary.
```

Once again, notice that things are printed in a seemingly random order.

Using `.keys()`

Code:

```
ages = {}

ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for name in ages.keys():
    print name, "is", ages[name]
```

This gets the value associated with the name

Output:

```
Sally is 36
Joe is 35
George is 39
```

Using `.keys()`

Code:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for name in ages:
    print name, "is", ages[name]
```

Note that in a `for` loop, you can actually leave off the `.keys()`, because this is what python loops over by default when a dict is the iterable.

Output:

```
Sally is 36
Joe is 35
George is 39
```

Using `.values()`

Code:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for age in ages.values():
    print "There is a person who is", age
```

Output:

```
There is a person who is 36
There is a person who is 35
There is a person who is 39
```

The order is still random-seeming, but note that it's the same order as when we printed the keys.

Using `.items()`

Code:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for (name, age) in ages.items():
    print name, "is", age
```

`.items()` returns two variables each time it is called: a key and its value. This is why we can simultaneously assign the result to two variables

Output:

```
Sally is 36
Joe is 35
George is 39
```

Sorting a dictionary

You can **not** sort a dictionary. However, you can emulate sorting in the following way:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for name in sorted(ages.keys()):
    print name, "is", ages[name]
```

Output:

```
George is 39
Joe is 35
Sally is 36
```

← Sorted based on person's name

Sorting by values

Occasionally, you'll also want to sort the keys of your dictionary based on their *value*, rather than the key itself. Here's one way to do it:

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39

for name in sorted(ages, key=ages.get):
    print name, "is", ages[name]
```

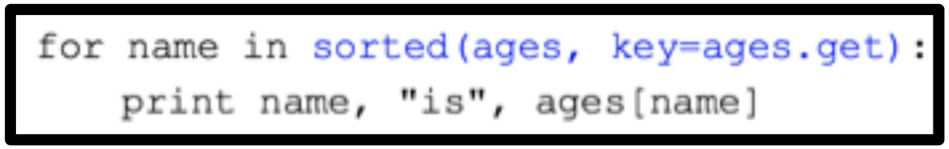
Output:

```
Joe is 35
Sally is 36
George is 39
```

Sorted based on age rather than name

Safe lookup using .get

You may have noticed the use of `ages.get` on the previous slide.



```
for name in sorted(ages, key=ages.get):
    print name, "is", ages[name]
```

The method **dict.get** has the following signature:

```
dict.get(self, key, default=None)
```

This method returns the equivalent of dict[key] if key is in dict.

Otherwise, it returns the specified default value.

This is effectively a way to look up an arbitrary element in a dict that may not have that element, without raising KeyError.

Terminology quiz

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39
```

"Joe" is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

Terminology quiz

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39
```

"Joe" is most accurately referred to as...

a. an element

b. an index

<u>c. a key</u>

d. a value

Terminology quiz

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39
```

35 is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

Terminology quiz

```
ages = {}
ages["Joe"] = 35
ages["Sally"] = 36
ages["George"] = 39
```

35 is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

"an element" is OK too, but value is the more common terminology

Terminology quiz

```
ages = []        #this is a list
ages[0] = 35
ages[1] = 36
ages[2] = 39
```

0 is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

Terminology quiz

```
ages = []       #this is a list
ages[0] = 35
ages[1] = 36
ages[2] = 39
```

0 is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

Terminology quiz

```
ages = []        #this is a list
ages[0] = 35
ages[1] = 36
ages[2] = 39
```

39 is most accurately referred to as...

a. an element

b. an index

c. a key

d. a value

Terminology quiz

```
ages = []       #this is a list
ages[0] = 35
ages[1] = 36
ages[2] = 39
```

39 is most accurately referred to as...

- **a. an element**
- b. an index
- c. a key
- d. a value

Appendix

A longer example, with graphing!

Why use a dictionary?

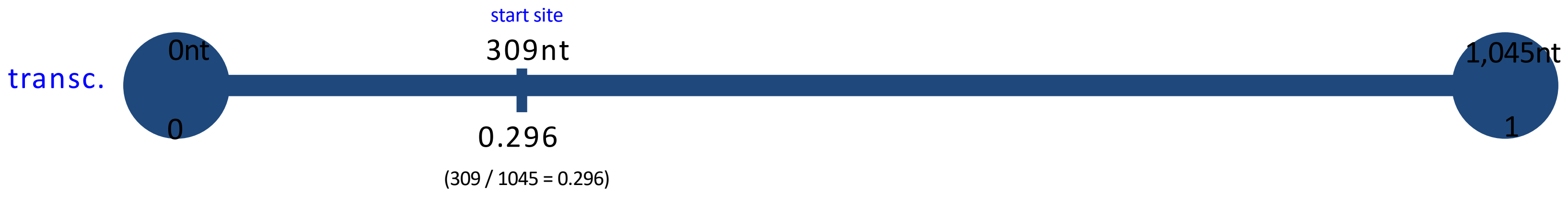
Technically, anything you can do with a dictionary you could also just do with a list instead. But dictionaries make coding certain tasks much easier.

Example: matching across files

One problem I encounter a lot is where I have two files with different information about a transcript, and I need to integrate the info.

REAL LIFE SITUATION (!!)

I have a file with transcript ids and translation start sites. I need to normalize these start positions by transcript length so that I can graph the distribution of start sites across all transcripts in my dataset. To do this, I need to divide the start site position by the full length of the transcript. Unfortunately, I have transcript lengths stored in a separate file, so I need to **match up start sites with their full transcript lengths.**



Ex. cont.: data files

Here are the formats of my data files:

Start site file:

```
knownGene       Gene    InitCodon       DistCDS         Frame   InitContext     CDSLen  PeakSt  PeakWidth       #Reads  PeakScore       Codon   Product
uc007afd.1      Mrpl15 248      79      1       AATATGG         15      247     2       368     2.61    aug     internal-out-of-frame

uc007afh.1      Lypla1 36       5       0       AACATGT         225     34      4       783     3.27    aug     n-term-trunc

uc007afi.1      Tcea1   28      -24     0       GGCTTGT         325     27      3       446     1.43    nearcog         n-term-ext
uc007afi.1      Tcea1   100     0       0       GCCATGG         301     99      3       3852    3.79    aug     canonical

uc007afn.1      Atp6v1h         100     -13     -1      GCTATCC         10      99      3       728     0.77    nearcog         uorf
uc007afn.1      Atp6v1h         149     3       0       AAGATGG         480     147     3       1407    1.36    aug     n-term-trunc

uc007agb.1      Pcmtd1 120      -97     -1      GCGCTGG         45      119     3       65      0.75    nearcog         uorf
uc007agb.1      Pcmtd1 265      -49     0       GCGCTGC         42      264     3       133     0.86    nearcog         uorf
…
```

Transc. length file:

```
SeqID           Len
uc009gmc.1      4900

uc008mue.1      459

uc007hzr.1      4578
uc007gtm.1      1257

uc007axo.1      2311
uc007wps.1      2694

uc007gqc.1      30
uc009smc.1      1530
...
```

The common piece of information between these files is the **transcript ID**, so this is what we will use to match up start sites to transcript lengths.

Ex. cont.: Plan

When working with more than one file, it sometimes helps to write down a step-by-step plan before you start coding.

Here's my plan:

```
1. Open length file
2. For each line in length file:
    a. Extract the id (1st column) and length (2nd column)
    b. Store lengths in hash based on id
        hash[id] --> length
3. Open output file
4. Open tss file
5. For each line in tss file:
    a. Extract the id (1st column) and start site (3rd column)
    b. Using the id, lookup the length of the transcript from the hash
    c. Divide the start position by the length of the transc.
    d. Print the result to the output file
    e. Also store the result in a list for graphing
```

Ex. cont.: Code pt. 1

```
# input files
tssFile = "all_start_sites.txt"
lenFile = "transc_lengths.txt"

# output files
normOut = "normalized_tss.txt"

# data
lengths = {}
normLengths = []

# read in lengths, store in hash
ins = open(lenFile, 'r')
ins.readline()                 #don't forget to skip the header!
for line in ins:
    line = line.rstrip('\n')
    (id, len) = line.split()   #split on whitespace
    lengths[id] = len          #store len in hash labeled by the id
ins.close()
```

... continued on next slide

Ex. cont.: Code pt. 2

```
# read in TSS, use stored lengths to normalize,
# then print. No need to store TSS.
output = open(normOut, 'w')
output.write("RelativePos\n")             #header line for output file
ins = open(tssFile, 'r')
ins.readline()                            #skip header in input file
for line in ins:
    line = line.rstrip('\n')
    data = line.split()                   #data is now a LIST
    id = data[0]                          #id is the first data value in the list
    tss = float(data[2])                  #start site is third data value in the list
    if id in lengths:                     #make sure there is an entry in hash for this id**
          fullLen = int(lengths[id])      #using hash, lookup the length of this transc
          norm = tss / fullLen            #divide start position by full length
          output.write(str(norm) + "\n")  #output the result to a file
          normLengths.append(norm)        # also add the result to a list for graphing later
ins.close()
output.close()
```

** If there are transcript ids in the TSS file that were not in the length file, then we will get an error when we try to look up that id in the hash (because it's just not there). If you think there is a chance this might happen (and there almost always is) just add this quick `if` statement to skip over any ids that would cause an error.

Ex. cont.: Output

Here's what the output file looks like:

```
RelativePos
0.0147118921128
0.0506072874494
0.0754048582996
0.0229226361032
0.0506208213945
0.0787010506208
0.140783190067
0.170009551098
0.0329929300864
0.0675569520817
0.0523469608479
0.0627298291153
0.0752757949384
0.100800346096
...
```

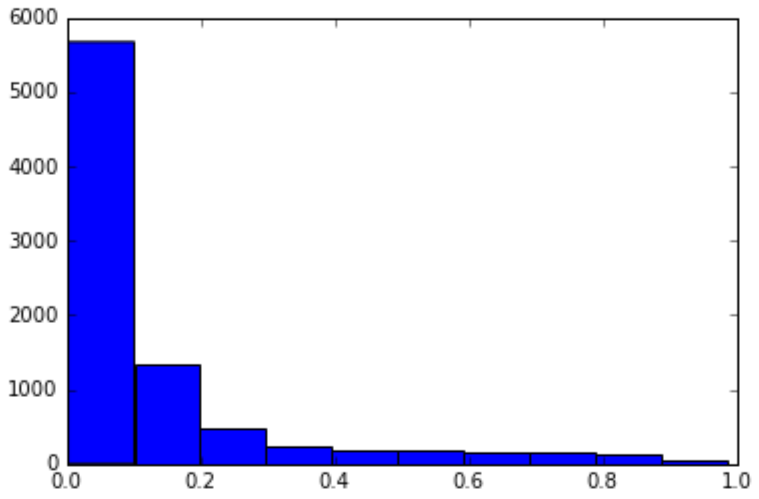
Using matplotlib

This type of data is best represented using a histogram. Here's how you can easily create a histogram using the package matplotlib:

```
%matplotlib inline
import matplotlib.pyplot as plt

# plot a histogram of the data
plt.hist(normLengths)
plt.show()
```

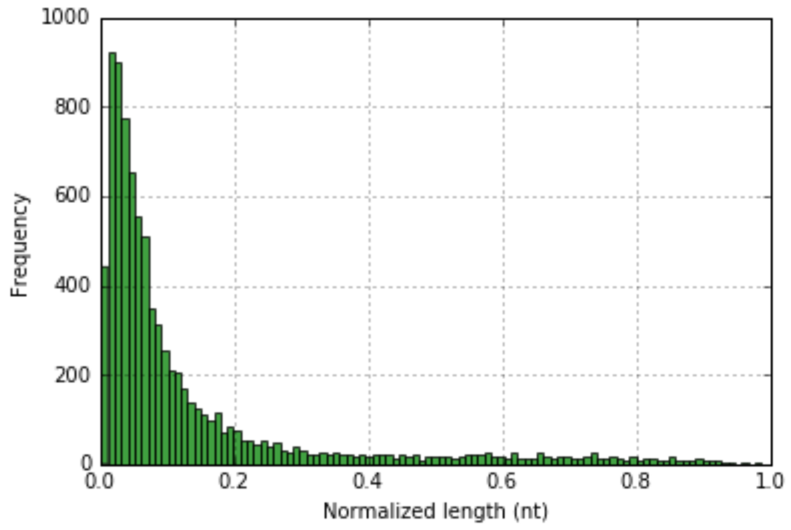
`normLengths` is the list of normalized lengths that we created on the previous slide



Using matplotlib

Here's the same plot, but using some options to make it look nicer:

```
# a nicer histogram
plt.hist(normLengths, 100, facecolor='green', alpha=0.75) #100 is the number of "bins" in the histogram
plt.grid(True)
plt.xlabel('Normalized length (nt)')
plt.ylabel('Frequency')
plt.show()
```


Frequency
Normalized length (nt)