同濟大學 TONGJI UNIVERSITY

生物信息学系 DEPARTMENT OF BIOINFORMATICS

Section 6

Python: Filtering Data
Managing Tabular Data
Sorting Data

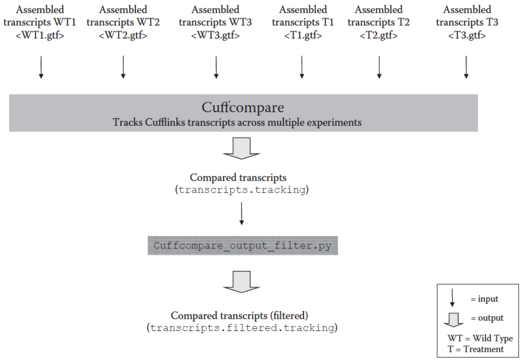
Yong Zhang, Ph. D

School of Life Science and Technology
Tongji University

Apr 4, 2018

yzhang@tongji.edu.cn

Story: Working with RNA-seq output data


Assembled transcripts WT1 <WT1.gtf>
Assembled transcripts WT2 <WT2.gtf>
Assembled transcripts WT3 <WT3.gtf>
Assembled transcripts T1 <T1.gtf>
Assembled transcripts T2 <T2.gtf>
Assembled transcripts T3 <T3.gtf>
Cuffcompare
Tracks Cufflinks transcripts across multiple experiments
Compared transcripts
(transcripts.tracking)
Cuffcompare_output_filter.py
Compared transcripts (filtered)
(transcripts.filtered.tracking)
= input
= output
WT = Wild Type
T = Treatment

Story: Working with RNA-seq output data

- A line in *transcripts.tracking* file:

```
Medullo-Diff_00000001 XLOC_000001    Lypla1|uc007afh.1
q1:NSC.P419.228|uc007afh.1|100|35.109496|34.188903|
   36.030089|397.404732|2433
q2:NSC.P429.18|uc007afh.1|100|15.885823|15.240240|
   16.531407|171.011325|2433
q3:NSC.P437.15|uc007afh.1|100|18.338541|17.704857|
   18.972224|181.643949|2433
q4:CSC.Mmb8.236|uc007afh.1|100|22.594194|21.925964|
   23.262424|225.248080|2433
q5:CSC.Mmb10.251|uc007afh.1|100|22.778360|22.025125|
   23.531595|255.416281|2433
q6:CSC.Mmb21.221|uc007afh.1|100|17.288114|16.675834|
   17.900395|184.487708|2433
```

1. sample label
2. transcript ID
3. Gene ID
4. fmi (fraction of the major isoform)
5. expression mean value
6. expression min value
7. expression max value
8. transcript coverage
9. transcript length

Story: Working with RNA-seq output data

- When a given transcript has not been detected in a sample, the corresponding cell in the table contains a dash:

```
Medullo-Diff_00000002   XLOC_000002   Tcea1|uc007afi.2=
q1:NSC.P419.228|uc007afi.2|18|1.653393|1.409591|
   1.897195|18.587029|2671
    -
q3:NSC.P437.108|uc007afi.2|100|4.624079|4.258801|
   4.989356|45.379750|2671
    -
    -
    -
```

Story: Working with RNA-seq output data

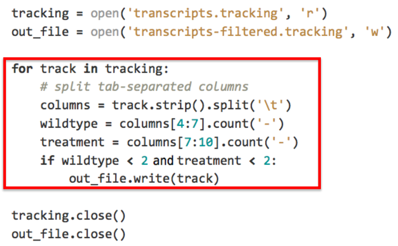
- Replicates are necessary to ensure robustness of data. You want to retain only transcripts that have been observed in the transcriptome of all replicas or at least in two out of three of them.

WT1	WT2	WT3	T1	T2	T3	
q1	q2	q3	q4	q5	q6	save
q1	-	q3	q4	q5	q6	save
q1	q2	q3	-	q5	-	skip
-	q2	q3	q4	-	q6	save
q1	-	-	q4	q5	q6	skip
q1	q2	-	-	-	-	skip

Example Python session

```python
tracking = open('transcripts.tracking', 'r')
out_file = open('transcripts-filtered.tracking', 'w')

for track in tracking:
    # split tab-separated columns
    columns = track.strip().split('\t')
    wildtype = columns[4:7].count('-')
    treatment = columns[7:10].count('-')
    if wildtype < 2 and treatment < 2:
        out_file.write(track)

tracking.close()
out_file.close()
```

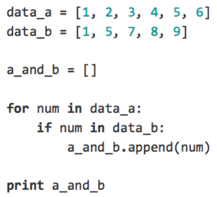
The *for...if* combination



```
tracking = open('transcripts.tracking', 'r')
out_file = open('transcripts-filtered.tracking', 'w')

for track in tracking:
    # split tab-separated columns
    columns = track.strip().split('\t')
    wildtype = columns[4:7].count('-')
    treatment = columns[7:10].count('-')
    if wildtype < 2 and treatment < 2:
        out_file.write(track)

tracking.close()
out_file.close()
```

The *for...if* combination

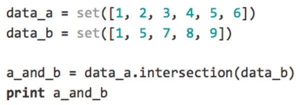
- Write all the elements belonging to both lists to a new list:



```
data_a = [1, 2, 3, 4, 5, 6]
data_b = [1, 5, 7, 8, 9]

a_and_b = []

for num in data_a:
    if num in data_b:
        a_and_b.append(num)

print a_and_b
```

The *for...if* combination

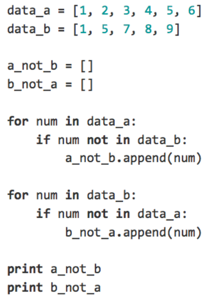
- Another solution:



```
data_a = set([1, 2, 3, 4, 5, 6])
data_b = set([1, 5, 7, 8, 9])

a_and_b = data_a.intersection(data_b)
print a_and_b
```

Differences between two data sets

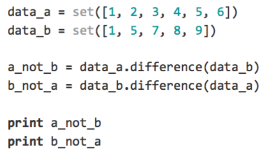
- To collect elements of *data_a* that are not in *data_b* and elements of *data_b* that are not in *data_a*:



```
data_a = [1, 2, 3, 4, 5, 6]
data_b = [1, 5, 7, 8, 9]

a_not_b = []
b_not_a = []

for num in data_a:
    if num not in data_b:
        a_not_b.append(num)

for num in data_b:
    if num not in data_a:
        b_not_a.append(num)

print a_not_b
print b_not_a
```

Differences between two data sets

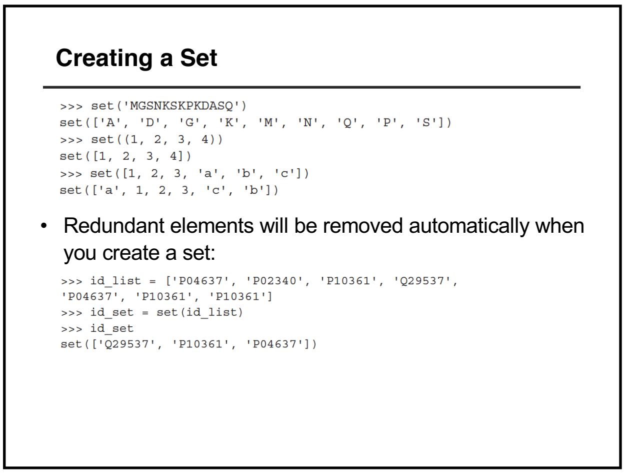
- Another solution:



```
data_a = set([1, 2, 3, 4, 5, 6])
data_b = set([1, 5, 7, 8, 9])

a_not_b = data_a.difference(data_b)
b_not_a = data_b.difference(data_a)

print a_not_b
print b_not_a
```

Sets

- Sets are unordered collections of unique objects.
- Sets are an ideal data structure to remove duplicates and to calculate the intersection, the union, and the difference between two or more groups of objects, as long as the order is not important.
- Sets do not support indexing and slicing operations, but the 'in' and 'not in' operators can be used to test an element for membership in a set.
- The elements of a set must be immutable objects such as numbers, strings, or tuples; thus lists, dictionaries, and other sets cannot be elements of a set.

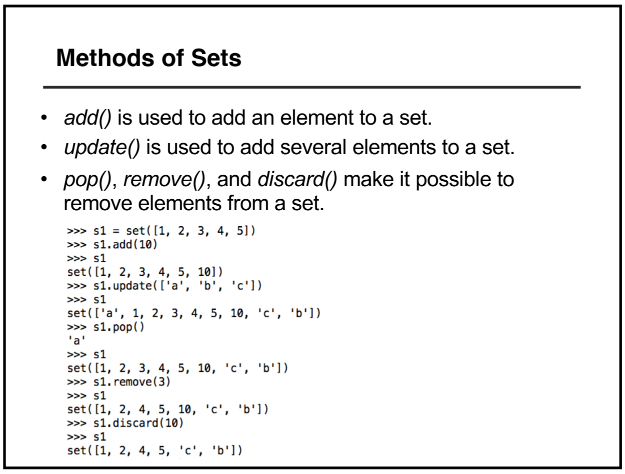


Creating a Set

```
>>> set('MGSNKSKPKDASQ')
set(['A', 'D', 'G', 'K', 'M', 'N', 'Q', 'P', 'S'])
>>> set((1, 2, 3, 4))
set([1, 2, 3, 4])
>>> set([1, 2, 3, 'a', 'b', 'c'])
set(['a', 1, 2, 3, 'c', 'b'])
```

- Redundant elements will be removed automatically when you create a set:

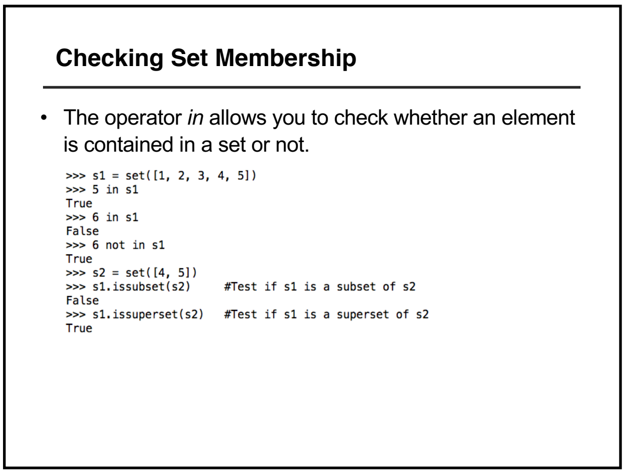
```
>>> id_list = ['P04637', 'P02340', 'P10361', 'Q29537',
'P04637', 'P10361', 'P10361']
>>> id_set = set(id_list)
>>> id_set
set(['Q29537', 'P10361', 'P04637'])
```



Methods of Sets

- *add()* is used to add an element to a set.
- *update()* is used to add several elements to a set.
- *pop()*, *remove()*, and *discard()* make it possible to remove elements from a set.

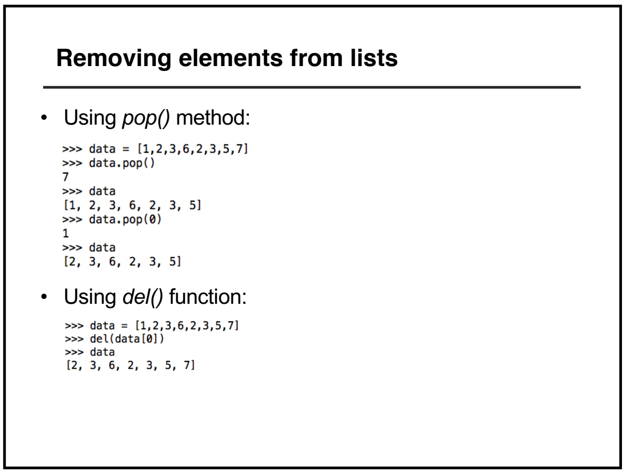
```
>>> s1 = set([1, 2, 3, 4, 5])
>>> s1.add(10)
>>> s1
set([1, 2, 3, 4, 5, 10])
>>> s1.update(['a', 'b', 'c'])
>>> s1
set(['a', 1, 2, 3, 4, 5, 10, 'c', 'b'])
>>> s1.pop()
'a'
>>> s1
set([1, 2, 3, 4, 5, 10, 'c', 'b'])
>>> s1.remove(3)
>>> s1
set([1, 2, 4, 5, 10, 'c', 'b'])
>>> s1.discard(10)
>>> s1
set([1, 2, 4, 5, 'c', 'b'])
```



Checking Set Membership

- The operator *in* allows you to check whether an element is contained in a set or not.

```
>>> s1 = set([1, 2, 3, 4, 5])
>>> 5 in s1
True
>>> 6 in s1
False
>>> 6 not in s1
True
>>> s2 = set([4, 5])
>>> s1.issubset(s2)     #Test if s1 is a subset of s2
False
>>> s1.issuperset(s2)   #Test if s1 is a superset of s2
True
```



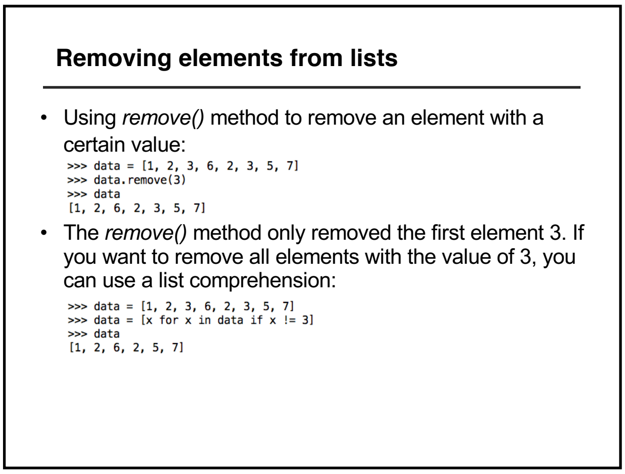
Removing elements from lists

- Using *pop()* method:

```
>>> data = [1,2,3,6,2,3,5,7]
>>> data.pop()
7
>>> data
[1, 2, 3, 6, 2, 3, 5]
>>> data.pop(0)
1
>>> data
[2, 3, 6, 2, 3, 5]
```

- Using *del()* function:

```
>>> data = [1,2,3,6,2,3,5,7]
>>> del(data[0])
>>> data
[2, 3, 6, 2, 3, 5, 7]
```



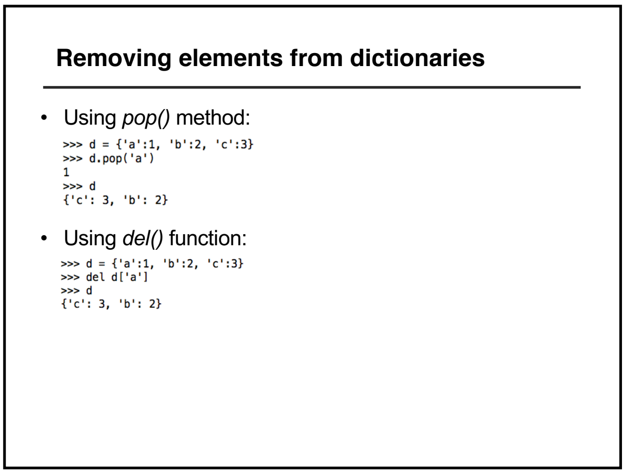
Removing elements from lists

- Using *remove()* method to remove an element with a certain value:

```
>>> data = [1, 2, 3, 6, 2, 3, 5, 7]
>>> data.remove(3)
>>> data
[1, 2, 6, 2, 3, 5, 7]
```

- The *remove()* method only removed the first element 3. If you want to remove all elements with the value of 3, you can use a list comprehension:

```
>>> data = [1, 2, 3, 6, 2, 3, 5, 7]
>>> data = [x for x in data if x != 3]
>>> data
[1, 2, 6, 2, 5, 7]
```



Removing elements from dictionaries

- Using *pop()* method:

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> d.pop('a')
1
>>> d
{'c': 3, 'b': 2}
```

- Using *del()* function:

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> del d['a']
>>> d
{'c': 3, 'b': 2}
```

Deleting particular lines from a text file

- Suppose you have the input file *text.txt* and you want to remove the first and second lines and the fifth and sixth lines:

```
lines = open('text.txt').readlines()
open('new.txt','w').writelines(lines[2:4]+lines[6:])
```

- Another solution:

```
in_file = open('text.txt')
out_file = open('new.txt', 'w')

index = 0
indices_to_remove = [1, 2, 5, 6]
for line in in_file:
    index = index + 1
    if index not in indices_to_remove:
        out_file.write(line)

in_file.close()
out_file.close()
```

Deleting particular lines from a text file

- Using *enumerate()* function:

```
out_file = open('new.txt', 'w')
indices_to_remove = [1, 2, 5, 6]
for index, line in enumerate(open('text.txt')):
    if (index + 1) not in indices_to_remove:
        out_file.write(line)
out_file.close()
```

- Given a list *x*, *enumerate(x)* returns tuples (*i, x[i]*) of indexes *i* and values *x[i]* of the list:

```
>>> x = [1,2,5,6]
>>> for i,j in enumerate(x):
...     print i, j
...
0 1
1 2
2 5
3 6
```

Remove duplicates preserving order

- To remove duplicate lines in a text file and create a new file containing only unique elements.
- Suppose you have the following input file with Uniprot ACs.

```
P04637
P02340
P10361
Q29537
P04637
P10361
P10361
P02340
```

Remove duplicates preserving order

- You want the output to contain only unique Uniprot ACs.

```
input_file = open('UniprotID.txt')
output_file = open('UniprotID-unique.txt','w')

unique = []
for line in input_file:
    if line not in unique:
        output_file.write(line)
        unique.append(line)

input_file.close()
output_file.close()
```

Examples

- Comparing more than two sets of data.

```
a = set((1, 2, 3, 4, 5))
b = set((2, 4, 6, 7, 1))
c = set((1, 4, 5, 9))

triple_set = [a, b, c]
common = reduce(set.intersection, triple_set)
print common
```

Examples

- Function *reduce()* takes two arguments: the first one is a function taking two variables (*f(x,y)*), and the second one is an iterable object *i* (a tuple or a list).
- *reduce(f, i)* passes the first two elements of the iterable (*i[0]* and *i[1]*) to the function *f* and calculates the value returned by the function, then passes that value to *f* as the first argument, and the third element of the iterable (*i[2]*) as the second argument, and so on.

```
>>> def multiply(x,y):
...     return x * y
...
>>> print reduce(multiply, (1, 2, 3, 4))
24
```

Examples

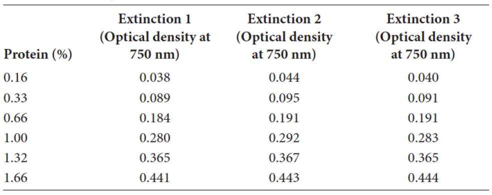
- Compare/update different releases of a database (e.g., Uniprot).
- Suppose you have two Uniprot releases, in the form of lists of Uniprot ACs, and want to see which entries are new, which entries disappeared, and what is absent in either the old or the new release.

Examples

```
# read old database release
old_db = set()
for line in open("list_old.txt"):
    accession = line.strip()
    old_db.add(accession)

# read new database release
new_db = set()
for line in open("list_new.txt"):
    accession = line.strip()
    new_db.add(accession)

# report differences
new_entries = new_db.difference(old_db)
print "new entries", list(new_entries)
old_entries = old_db.difference(new_db)
print "deprecated entries", list(old_entries)
unique_entries = new_db.symmetric_difference(old_db)
print "unique entries", list(unique_entries)
```

Story: Determining protein concentrations

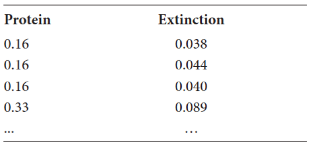
- Lowry's measurement of small amount of protein from rabbit brain.



Protein (%)	Extinction 1 (Optical density at 750 nm)	Extinction 2 (Optical density at 750 nm)	Extinction 3 (Optical density at 750 nm)
0.16	0.038	0.044	0.040
0.33	0.089	0.095	0.091
0.66	0.184	0.191	0.191
1.00	0.280	0.292	0.283
1.32	0.365	0.367	0.365
1.66	0.441	0.443	0.444

Story: Determining protein concentrations

- How can the table with the original data (4 columns) be converted to the following simpler one:



Protein	Extinction
0.16	0.038
0.16	0.044
0.16	0.040
0.33	0.089
...	...

Example Python session

```
table = [
    ['protein', 'ext1', 'ext2', 'ext3'],
    [0.16, 0.038, 0.044, 0.040],
    [0.33, 0.089, 0.095, 0.091],
    [0.66, 0.184, 0.191, 0.191],
    [1.00, 0.280, 0.292, 0.283],
    [1.32, 0.365, 0.367, 0.365],
    [1.66, 0.441, 0.443, 0.444]
    ]

table = table[1:]

protein, ext1, ext2, ext3 = zip(*table)

extinction = ext1 + ext2 + ext3
protein = protein * 3

table = zip(protein, extinction)

for prot, ext in table:
    print prot, ext
```

Representing a two-dimensional table

```
table = [
    ['protein', 'ext1', 'ext2', 'ext3'],
    [0.16, 0.038, 0.044, 0.040],
    [0.33, 0.089, 0.095, 0.091],
    [0.66, 0.184, 0.191, 0.191],
    [1.00, 0.280, 0.292, 0.283],
    [1.32, 0.365, 0.367, 0.365],
    [1.66, 0.441, 0.443, 0.444]
    ]

table = table[1:]

protein, ext1, ext2, ext3 = zip(*table)

extinction = ext1 + ext2 + ext3
protein = protein * 3

table = zip(protein, extinction)

for prot, ext in table:
    print prot, ext
```

Representing a two-dimensional table

- A table can be encoded as a *list of lists*, also called a *nested list*.
- For example, the following table:

1	2	3
4	5	6
7	8	9

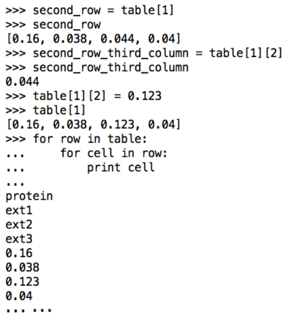
  - encoded as a nested list:

```
square = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  - encoded as a list of tuples:

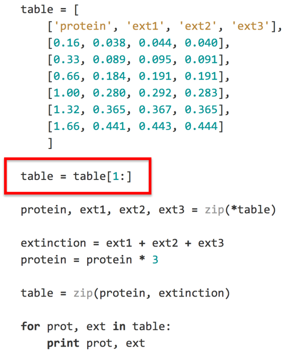
```
square = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

Accessing rows and single cells



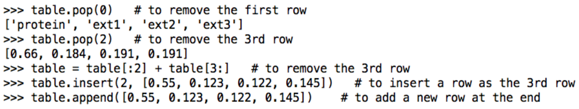
```
>>> second_row = table[1]
>>> second_row
[0.16, 0.038, 0.044, 0.04]
>>> second_row_third_column = table[1][2]
>>> second_row_third_column
0.044
>>> table[1][2] = 0.123
>>> table[1]
[0.16, 0.038, 0.123, 0.04]
>>> for row in table:
...     for cell in row:
...         print cell
...
protein
ext1
ext2
ext3
0.16
0.038
0.123
0.04
... ...
```

Inserting and removing rows



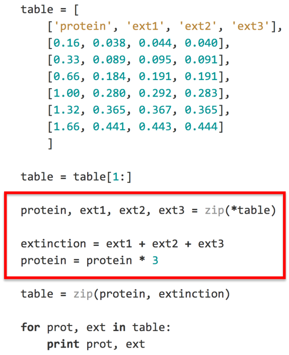
```
table = [
    ['protein', 'ext1', 'ext2', 'ext3'],
    [0.16, 0.038, 0.044, 0.040],
    [0.33, 0.089, 0.095, 0.091],
    [0.66, 0.184, 0.191, 0.191],
    [1.00, 0.280, 0.292, 0.283],
    [1.32, 0.365, 0.367, 0.365],
    [1.66, 0.441, 0.443, 0.444]
    ]

table = table[1:]

protein, ext1, ext2, ext3 = zip(*table)

extinction = ext1 + ext2 + ext3
protein = protein * 3

table = zip(protein, extinction)

for prot, ext in table:
    print prot, ext
```

Inserting and removing rows



```
>>> table.pop(0)   # to remove the first row
['protein', 'ext1', 'ext2', 'ext3']
>>> table.pop(2)   # to remove the 3rd row
[0.66, 0.184, 0.191, 0.191]
>>> table = table[:2] + table[3:]   # to remove the 3rd row
>>> table.insert(2, [0.55, 0.123, 0.122, 0.145])   # to insert a row as the 3rd row
>>> table.append([0.55, 0.123, 0.122, 0.145])    # to add a new row at the end
```

Accessing columns



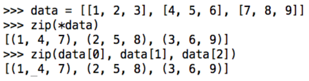
```
table = [
    ['protein', 'ext1', 'ext2', 'ext3'],
    [0.16, 0.038, 0.044, 0.040],
    [0.33, 0.089, 0.095, 0.091],
    [0.66, 0.184, 0.191, 0.191],
    [1.00, 0.280, 0.292, 0.283],
    [1.32, 0.365, 0.367, 0.365],
    [1.66, 0.441, 0.443, 0.444]
    ]

table = table[1:]

protein, ext1, ext2, ext3 = zip(*table)

extinction = ext1 + ext2 + ext3
protein = protein * 3

table = zip(protein, extinction)

for prot, ext in table:
    print prot, ext
```

Accessing columns

- The *zip()* function allows to combine elements from two or more lists.

```
>>> zip([1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

- The asterisk tells the *zip* function to use all lists from the nested lists as arguments.



```
>>> data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zip(*data)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> zip(data[0], data[1], data[2])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Inserting and removing columns

- A very common usage of *zip()* is to rotate (or transpose) tables.

```
table = zip(*table)
```

- To insert a column, you need to first turn the table, insert a row, and then turn the table back.

```
table = zip(*table)
table.append (['ext4', 0, 0, 0, 0, 0, 0])
table = zip(*table)
```

- You can also delete a column from a table.

```
table = zip(*table)
table.pop(1)
table = zip(*table)
```

Inserting and removing columns

- *zip()* function converts the inner lists to tuples. You need to convert the row to a list again:

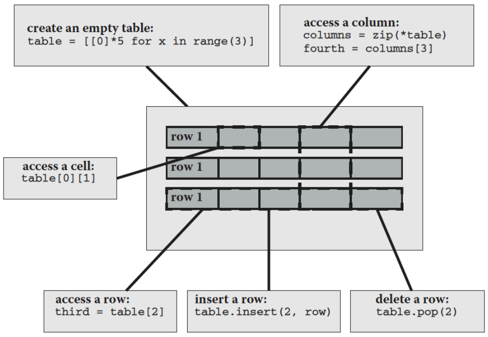
```
table[1] = list(table[1])
table[1][2] = 0.123
```

Combining multiple columns

- The plus *(+)* and multiplication *(*)* operators can be applied to combine and multiply lists and tuples.

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Actions that can be carried out on tables



create an empty table:
```
table = [[0]*5 for x in range(3)]
```

access a column:
```
columns = zip(*table)
fourth = columns[3]
```

access a cell:
```
table[0][1]
```

access a row:
```
third = table[2]
```

insert a row:
```
table.insert(2, row)
```

delete a row:
```
table.pop(2)
```

Methods for storing tables

list in a list
```
[
[1, 2],
[3, 4]
]
```

dictionary in a list
```
[
{'x':1, 'y':2},
{'x':3, 'y':4}
]
```

	x	y
a	1	2
b	3	4

list in a dictionary
```
{
'a':[1, 2],
'b':[3, 4]
}
```

dictionary in a dictionary
```
{
'a':{'x':1, 'y':2},
'b':{'x':3, 'y':4}
}
```

Pros and cons of table representations

- Lists in lists:
  - Pros: Adding and deleting rows to a table is easy. The list can be sorted with a single command.
  - Cons: To find a certain element by its name, you need to run a *for* loop over the entire table, which is slow. To address individual elements, you need to use numerical indices, which makes the code harder to read.
- Dictionaries in dictionaries:
  - Pros: Finding any entry in the table by its name is easy and fast. The explicit labeling of cells by the column names makes the code easier to read.
  - Cons: A dictionary is by definition unsorted, so it is not possible to sort the data in this representation.

Pros and cons of table representations

- Mixed lists and dictionaries:
  - Pros: This combines the advantages of both types. You can choose to use lists for the rows and dictionaries for the columns, or vice versa.
  - Cons: Using the table becomes a little less straightforward, and you need to remember in which way to access rows and in which columns. The code will be a little harder to read.

Convert a nested list into a nested dictionary

```
table = [
['protein', 'ext1', 'ext2', 'ext3'],
[0.16, 0.038, 0.044, 0.040],
[0.33, 0.089, 0.095, 0.091],
[0.66, 0.184, 0.191, 0.191],
[1.00, 0.280, 0.292, 0.283],
[1.32, 0.365, 0.367, 0.365],
[1.66, 0.441, 0.443, 0.444]
]
nested_dict = {}
n = 0
key = table[0]
# to include the header, run the for loop over
# ALL table elements (including the first one)
for row in table[1:]:
    n = n + 1
    entry = {key[0]: row[0], key[1]: row[1], key[2] : row[2], key[3] : row[3]}
    nested_dict['row'+str(n)] = entry
print nested_dict
```

Convert a nested dictionary into a nested list

```
nested_dict = {
'row1': {'protein': 0.16, 'ext3': 0.04, 'ext2': 0.044, 'ext1': 0.038},
'row2': {'protein': 0.33, 'ext3': 0.091, 'ext2': 0.095, 'ext1': 0.089},
'row3': {'protein': 0.66, 'ext3': 0.191, 'ext2': 0.191, 'ext1': 0.184},
'row4': {'protein': 1.0, 'ext3': 0.283, 'ext2': 0.292, 'ext1': 0.28},
'row5': {'protein': 1.32, 'ext3': 0.365, 'ext2': 0.367, 'ext1': 0.365},
'row6': {'protein': 1.66, 'ext3': 0.444, 'ext2': 0.443, 'ext1': 0.441}}

nested_list = []
nested_list.append(['protein', 'ext1', 'ext2', 'ext3'])
for entry in nested_dict:
    key = nested_dict[entry]
    nested_list.append([key['protein'], key['ext1'], key['ext2'], key['ext3']])

print nested_list
```

Examples

- Creating an empty table.

```
>>> table = []
>>> for i in range(6):
...     table.append([0] * 5)
...
>>> table
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

- Another solution:

```
>>> table = [[0] * 5 for i in range(6)]
>>> table
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Examples

- The following solution is not correct.

```
>>> row = [0] * 5
>>> table = [row] * 6
>>> table
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
>>> table[0][1] = 5
>>> table
[[0, 5, 0, 0, 0], [0, 5, 0, 0, 0], [0, 5, 0, 0, 0], [0, 5, 0, 0, 0],
[0, 5, 0, 0, 0], [0, 5, 0, 0, 0]]
```

  - In the resulting table, the rows will not be copies of the empty row, only references to the same row. So, the resulting table contains the same list object three times. Every time you change a cell in one row of the resulting table, all other rows will change simultaneously.

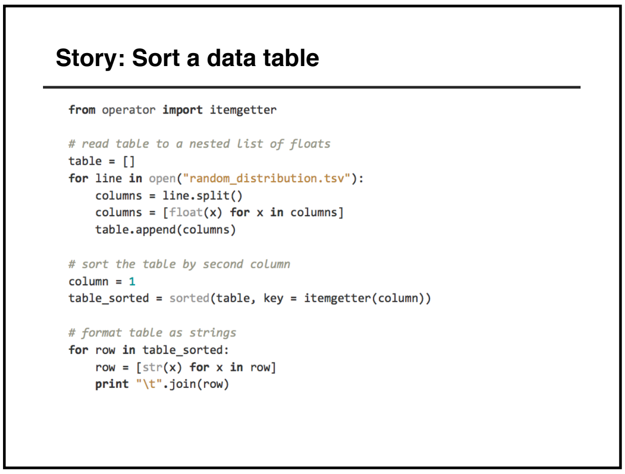
Examples

- How to read files with tabular data?

```
table = []
for line in open('lowry_data.txt'):
     table.append(line.strip().split('\t'))
```

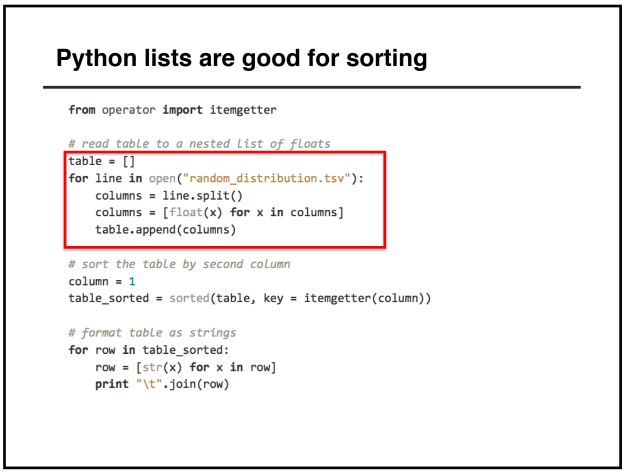
- How to write files with tabular data?

```
out = ''
for row in table:
    line = [str(cell) for cell in row]
    out = out + '\t'.join(line) + '\n'
open('lowry_data.txt', 'w').write(out)
```



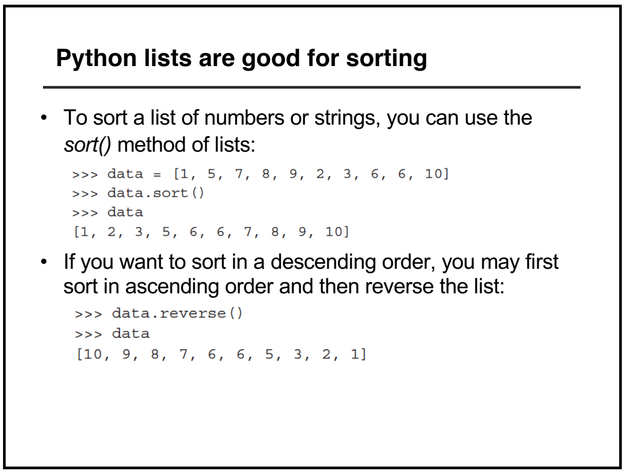
Story: Sort a data table

```
from operator import itemgetter

# read table to a nested list of floats
table = []
for line in open("random_distribution.tsv"):
    columns = line.split()
    columns = [float(x) for x in columns]
    table.append(columns)

# sort the table by second column
column = 1
table_sorted = sorted(table, key = itemgetter(column))

# format table as strings
for row in table_sorted:
    row = [str(x) for x in row]
    print "\t".join(row)
```



Python lists are good for sorting

```
from operator import itemgetter

# read table to a nested list of floats
table = []
for line in open("random_distribution.tsv"):
    columns = line.split()
    columns = [float(x) for x in columns]
    table.append(columns)

# sort the table by second column
column = 1
table_sorted = sorted(table, key = itemgetter(column))

# format table as strings
for row in table_sorted:
    row = [str(x) for x in row]
    print "\t".join(row)
```



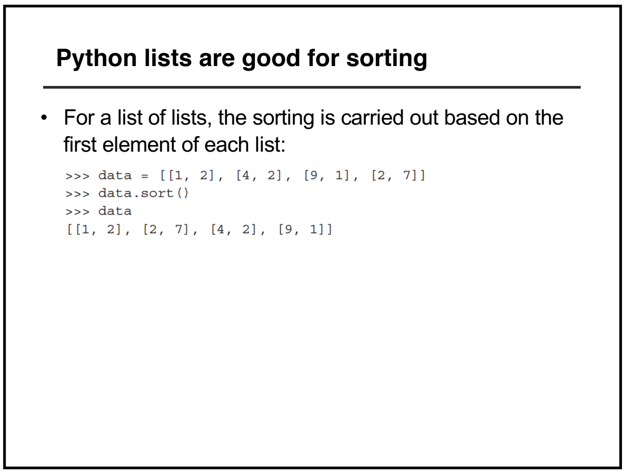
Python lists are good for sorting

- To sort a list of numbers or strings, you can use the *sort()* method of lists:

```
>>> data = [1, 5, 7, 8, 9, 2, 3, 6, 6, 10]
>>> data.sort()
>>> data
[1, 2, 3, 5, 6, 6, 7, 8, 9, 10]
```

- If you want to sort in a descending order, you may first sort in ascending order and then reverse the list:

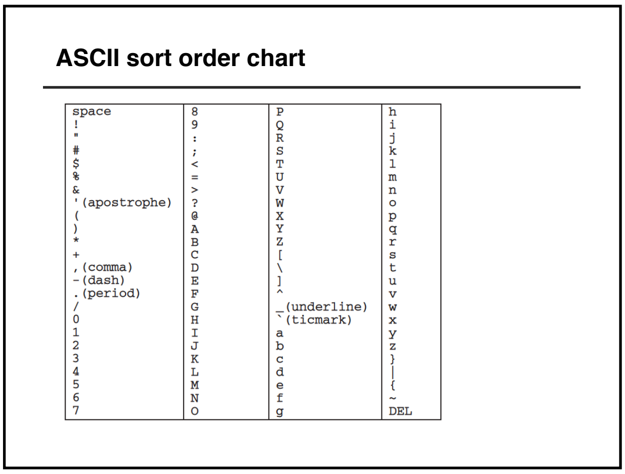
```
>>> data.reverse()
>>> data
[10, 9, 8, 7, 6, 6, 5, 3, 2, 1]
```



Python lists are good for sorting

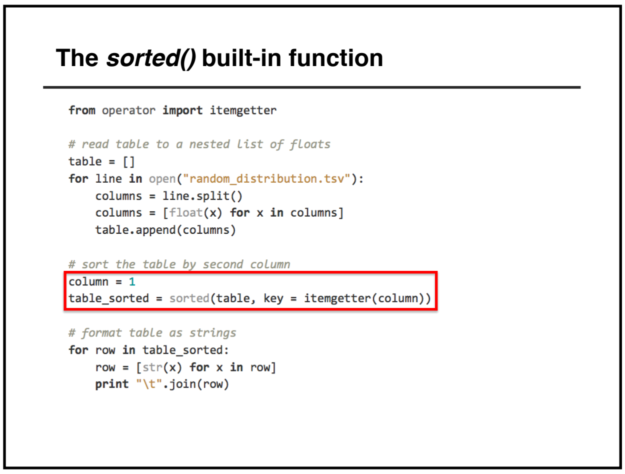
- For a list of lists, the sorting is carried out based on the first element of each list:

```
>>> data = [[1, 2], [4, 2], [9, 1], [2, 7]]
>>> data.sort()
>>> data
[[1, 2], [2, 7], [4, 2], [9, 1]]
```



ASCII sort order chart

```
space
!
"
#
$
%
&
'(apostrophe)
(
)
*
+
,(comma)
-(dash)
.(period)
/
0
1
2
3
4
5
6
7
8
9
:
;
<
=
>
?
@
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
[
\
]
^
_(underline)
`(ticmark)
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
}
|
{
~
DEL
```



The *sorted()* built-in function

```
from operator import itemgetter

# read table to a nested list of floats
table = []
for line in open("random_distribution.tsv"):
    columns = line.split()
    columns = [float(x) for x in columns]
    table.append(columns)

# sort the table by second column
column = 1
table_sorted = sorted(table, key = itemgetter(column))

# format table as strings
for row in table_sorted:
    row = [str(x) for x in row]
    print "\t".join(row)
```

The *sorted()* built-in function

- The advantage of *sorted()* is that it can sort many kinds of data, such as lists, tuples, or dictionary keys, whereas the method *sort()* only applies to lists.
- The *sorted()* built-in function returns a new sorted list:

```
>>> data = [1, 5, 7, 8, 9, 2, 3, 6, 6, 10]
>>> newdata = sorted(data)
>>> newdata
[1, 2, 3, 5, 6, 6, 7, 8, 9, 10]
```

Sorting with *itemgetter*

- *operator.itemgetter(i)(T)* returns the *ith* element of *T*, which can be a string, a list, a tuple, or a dictionary.
- It returns the value associated to key *i*. If you use two or more indices, the function returns a tuple:

```
>>> from operator import itemgetter
>>> data = ['ACCTGGCCA', 'ACTG', 'TACGGCAGGAGACG', 'TTGGATC']
>>> itemgetter(1)(data)
'ACTG'
>>> itemgetter(1, -1)(data)
('ACTG', 'TTGGATC')
```

- If you want to sort table first by the 2nd column and then by the 4th, you can write the column indices into *the itemgetter()* function:

```
new_table = sorted(table, key = itemgetter(1, 3))
```

Sorting in ascending/descending order

- To sort in descending order, the additional argument *reverse* = *True* can be passed to the *sorted()* function:

```
>>> sorted(data, reverse = True)
[30, 9, 9, 8, 8, 6, 5, 4, 3, 2, 1]

table = sorted(table, key = itemgetter(1), reverse = True)
```

Sort a dictionary according to its keys

- To sort a dictionary, you can extract all keys into a list and sort that list:

```
data = {1: 'a', 2: 'b', 4: 'd', 3: 'c',
 5: 't', 6: 'm', 36: 'z'}
keys = list(data)
keys.sort()
for key in keys:
    print key, data[key]
```

Sort a dictionary according to its keys

- The *sorted()* function is shorter to write:

```
data = {1: 'a', 2: 'b', 4: 'd', 3: 'c',
 5: 't', 6: 'm', 36: 'z'}
for key in sorted(data):
    print key, data[key]
```

Sort a Tuple

- Tuples are immutable and therefore cannot be sorted themselves. To sort a tuple, you need to convert it to a list, sort the list, and convert the list back to a tuple:

```
data = (1, 4, 5, 3, 8, 9, 2, 6, 8, 9, 30)
list_data = list(data)
list_data.sort()
new_tup = tuple(list_data)
print new_tup
```

- Using the *sorted()* function:

```
data = (1, 4, 5, 3, 8, 9, 2, 6, 8, 9, 30)
new_tup = tuple(sorted(data))
print new_tup
```

Sorting strings by their length

- You can use the *sorted()* built-in function with a *lambda* function as a custom parameter instead of *itemgetter*.

```
>>> data = ['ACCTGGCCA', 'ACTG', 'TACGGCAGGAGACG', 'TTGGATC']
>>> bylength = sorted(data, key = lambda x: len(x))
>>> bylength
['ACTG', 'TTGGATC', 'ACCTGGCCA', 'TACGGCAGGAGACG']
```

- If you have a table in the form of a nested list, you can use the key argument to specify the column by which you want to sort your table.

```
table = sorted(table, key = lambda col: col[1])
```

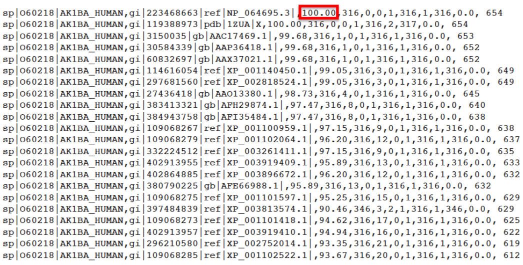
Examples

- Sort a table by the first column, then by the second, then by the third, and so on.

```
from operator import itemgetter

# read table
in_file = open("random_distribution.tsv")
table = []
for line in in_file:
    columns = line.split()
    columns = [float(x) for x in columns]
    table.append(columns)

table_sorted = sorted(table, key=itemgetter(0, 1, 2, 3, 4, 5, 6))
print table_sorted
```

Examples

- Sort the output of blast according to a parameter of your choice (e.g., sequence identity percentage).



```
sp|O60218|AK1BA_HUMAN,gi|223468663|ref|NP_064695.3|,100.00,316,0,0,1,316,1,316,0.0, 654
sp|O60218|AK1BA_HUMAN,gi|119388973|pdb|1ZUA|X,100.00,316,0,0,1,316,2,317,0.0, 654
sp|O60218|AK1BA_HUMAN,gi|3150035|gb|AAC17469.1|,99.68,316,1,0,1,316,1,316,0.0, 653
sp|O60218|AK1BA_HUMAN,gi|30584339|gb|AAP36418.1|,99.68,316,1,0,1,316,1,316,0.0, 652
sp|O60218|AK1BA_HUMAN,gi|60832697|gb|AAX37021.1|,99.68,316,1,0,1,316,1,316,0.0, 652
sp|O60218|AK1BA_HUMAN,gi|114616054|ref|XP_001140450.1|,99.05,316,3,0,1,316,1,316,0.0, 649
sp|O60218|AK1BA_HUMAN,gi|297681560|ref|XP_002818524.1|,99.05,316,3,0,1,316,1,316,0.0, 649
sp|O60218|AK1BA_HUMAN,gi|27436418|gb|AAO13380.1|,98.73,316,4,0,1,316,1,316,0.0, 645
sp|O60218|AK1BA_HUMAN,gi|383413321|gb|AFH29874.1|,97.47,316,8,0,1,316,1,316,0.0, 640
sp|O60218|AK1BA_HUMAN,gi|384943758|gb|AFI35484.1|,97.47,316,8,0,1,316,1,316,0.0, 638
sp|O60218|AK1BA_HUMAN,gi|109068267|ref|XP_001100959.1|,97.15,316,9,0,1,316,1,316,0.0, 638
sp|O60218|AK1BA_HUMAN,gi|109068279|ref|XP_001102064.1|,96.20,316,12,0,1,316,1,316,0.0, 637
sp|O60218|AK1BA_HUMAN,gi|332224512|ref|XP_003261411.1|,97.15,316,9,0,1,316,1,316,0.0, 635
sp|O60218|AK1BA_HUMAN,gi|402913955|ref|XP_003919409.1|,95.89,316,13,0,1,316,1,316,0.0, 633
sp|O60218|AK1BA_HUMAN,gi|402864885|ref|XP_003896672.1|,96.20,316,12,0,1,316,1,316,0.0, 632
sp|O60218|AK1BA_HUMAN,gi|380790225|gb|AFE66988.1|,95.89,316,13,0,1,316,1,316,0.0, 632
sp|O60218|AK1BA_HUMAN,gi|109068275|ref|XP_001101597.1|,95.25,316,15,0,1,316,1,316,0.0, 629
sp|O60218|AK1BA_HUMAN,gi|397484839|ref|XP_003813574.1|,90.46,346,3,2,1,316,1,346,0.0, 629
sp|O60218|AK1BA_HUMAN,gi|109068273|ref|XP_001101418.1|,94.62,316,17,0,1,316,1,316,0.0, 625
sp|O60218|AK1BA_HUMAN,gi|402913957|ref|XP_003919410.1|,94.94,316,16,0,1,316,1,316,0.0, 622
sp|O60218|AK1BA_HUMAN,gi|296210580|ref|XP_002752014.1|,93.35,316,21,0,1,316,1,316,0.0, 619
sp|O60218|AK1BA_HUMAN,gi|109068285|ref|XP_001102522.1|,93.67,316,20,0,1,316,1,316,0.0, 612
```

Examples

```
from operator import itemgetter

input_file = open("BlastOut.csv")
output_file = open("BlastOutSorted.csv","w")

# read BLAST output table
table = []
for line in input_file:
    col = line.split(',')
    col[2] = float(col[2])
    table.append(col)

table_sorted = sorted(table, key=itemgetter(2), reverse=True)

# write sorted table to an output file
for row in table_sorted:
    row = [str(x) for x in row]
    output_file.write("\t".join(row) + '\n')

input_file.close()
output_file.close()
```

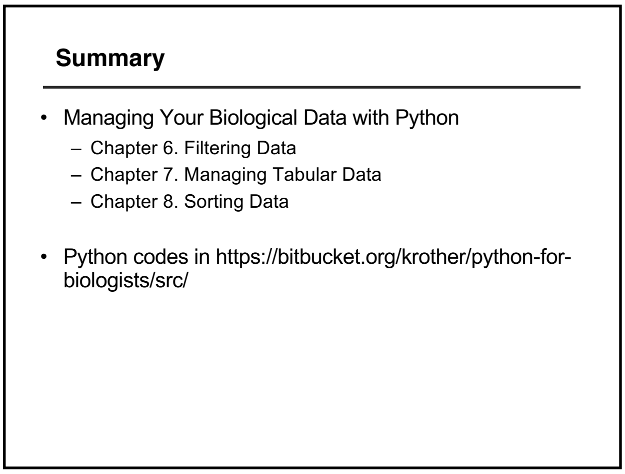
Examples

- Sort hemoglobin PDB entries on the basis of their RMSD (4th column), and then by the sequence length of the protein (5th column):

```
PDB ID,Chain ID,Exp. Method,Resolution,Chain Length
"1A4F","A","X-RAY DIFFRACTION","2.00","141"
"1C7C","A","X-RAY DIFFRACTION","1.80","283"
"1CG5","A","X-RAY DIFFRACTION","1.60","141"
"1FAW","A","X-RAY DIFFRACTION","3.09","141"
"1HDA","A","X-RAY DIFFRACTION","2.20","141"
"1IRD","A","X-RAY DIFFRACTION","1.25","141"
"1KFR","A","X-RAY DIFFRACTION","1.85","147"
"1QPW","A","X-RAY DIFFRACTION","1.80","141"
"1SPG","A","X-RAY DIFFRACTION","1.95","144"
"1UX8","A","X-RAY DIFFRACTION","2.15","132"
"1WXR","A","X-RAY DIFFRACTION","2.20","1048"
"2AA1","A","X-RAY DIFFRACTION","1.80","143"
"2D5X","A","X-RAY DIFFRACTION","1.45","141"
"2DHB","A","X-RAY DIFFRACTION","2.80","141"
```

Examples

```
from operator import itemgetter

input_file = open("PDBhaemoglobinReport.csv")
output_file = open("PDBhaemoglobinSorted.csv","w")

table = []
header = input_file.readline()
for line in input_file:
    col = line.split(',')
    col[3] = float(col[3][1:-1])
    col[4] = int(col[4][1:-2])
    table.append(col)

table_sorted = sorted(table, key=itemgetter(3, 4))

output_file.write(header + '\n' )
for row in table_sorted:
    row = [str(x) for x in row]
    output_file.write("\t".join(row) + '\n')

input_file.close()
output_file.close()
```



Summary

- Managing Your Biological Data with Python
  - Chapter 6. Filtering Data
  - Chapter 7. Managing Tabular Data
  - Chapter 8. Sorting Data

- Python codes in https://bitbucket.org/krother/python-for-biologists/src/