Reading Data into Dictionaries

Consider CSV data of the form:

```
Alabama,10,20,30,...
Alaska,32,43,56,...
.
.
.
Wyoming,2,0,78,...
```

Write a function `to_data` that takes a filename and returns a dictionary `data` where each key is a state name and each value is a list of integers.

```
>>> data["Minnesota"]
[47, 156, 107, 193, 121, 128]
>>> data["Iowa"]
[15, 36, 52, 57, 62, 45]
```

```
1 import csv
2
3 def data_from_file(filename):
4     with open(filename) as fin:
5         return {state: [int(x) for x in nums]
6                 for state, *nums in csv.reader(fin)}
```

The code above uses both a list and a dictionary comprehension. We are already familiar with list comprehensions in Python, which we have used to construct lists of expressions by iterating through objects:

```
[expression for thing in iterable]
```

A dictionary comprehension constructs a dictionary of (key,value) expression pairs by iterating through objects:

```
{key_expr : value_expr for thing in iterable}
```

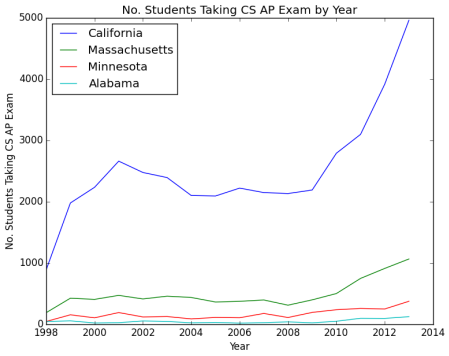
The example also does something called *star unpacking*. A single star * can be used to unpack a sequence or collection without knowing exactly how many values it contains. This usage is very different from multiplication. For example, we can use star unpacking to break an integer list ℓ into its first element, and the "rest" as follows:

```
>>> l = list(range(5))
>>> l
[0, 1, 2, 3, 4]
>>> first, *rest = l
>>> first
0
>>> rest
[1, 2, 3, 4]
```

In the example above, we know that the first column is always going to be a state, and the rest of the row the data for that state. As with list comprehensions, we could accomplish the same goal with other means (here, slicing), but star unpacking gives us a very clean way to do it.

Plotting with Matplotlib

Suppose we want to plot test takers, by state and year. Here is a nice line plot with legend and axis labels.



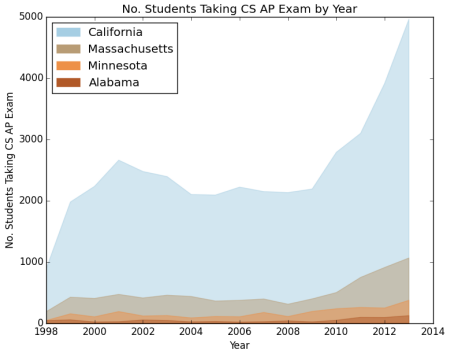
And here is the code.

```
1  import matplotlib.pyplot as plt
2
3  def plot1(data, states, years):
4      for state in states:
5          plt.plot(years, data[state], label=state)
6      plt.legend(loc="best")
7      plt.xlabel("Year")
8      plt.ylabel("No. Students Taking CS AP Exam")
9      plt.title("No. Students Taking CS AP Exam by Year")
10     plt.savefig("out.png")
```

Matplotlib syntax might look familiar to anyone who has used MATLAB. The `plot1()` function above starts by calling `pyplot.plot()`. Then, each successive `pyplot` function modifies the existing plot in some way. The `pyplot` functions have fairly intuitive names, but here are some helpful notes:

- If `pyplot.plot()` is called with just one list argument, it assumes that the list represents y-axis values.
- If `pyplot.plot()` is called with two list arguments, the first list is used as x-axis values, the second as y-axis values.
- The `pyplot.legend(loc=''best'')` argument tells Matplotlib that it should decide where to place the legend. A legend location can also be specified using strings (e.g., 'upper right') or an integer
- If we are using matplotlib interactively (in the REPL), we can use `pyplot.show()` to display our plot in its current state.
  - once we "show" the plot, the plot is "cleared" and we must rebuild it from scratch

A more aesthetically pleasing plot might fill the areas under the curves in. However, this requires us to choose our own colors using colormaps as well as create and place our own legend.



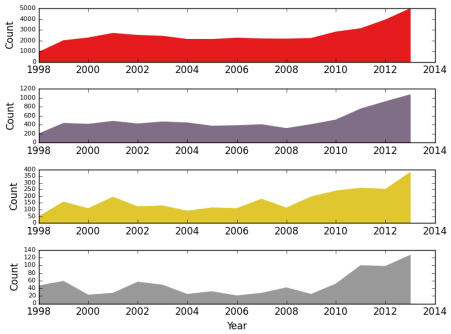

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import matplotlib.patches as mpatches
4  from itertools import count
5
6  def plot2(data, states, years):
7      colors = plt.cm.Paired(np.linspace(0,1,len(states)))
8      patches = []
9      for state,c in zip(states,colors):
10         plt.fill_between(years, data[state], color=c, alpha=0.5)
11         patches.append(mpatches.Patch(color=c, label=state))
12     plt.legend(handles=patches, loc="upper left")
13     plt.xlabel("Year")
14     plt.ylabel("No. Students Taking CS AP Exam")
15     plt.title("No. Students Taking CS AP Exam by Year")
16     plt.savefig("out2.png")
```

In the code above, we use the Paired colormap. This is a color map that is appropriate for Qualitative items where color is only used to distinguish the items from each other. Different color maps (e.g., sequential, diverging) are also available. It is important to consider the type of data that you are repsesenting as well as the way that your plot will be used—not all colors are print- or photocopy-friendly.

Some details to consider:

- `pyplot.fill_between()` is used to make filled polygons between two curves. It takes three equal-length arrays: x, y1, and y2 data. If y2 is omitted, it uses the default line $y = 0$.
- We construct a list of `matplotlib.patches.Patch` objects by providing a color and a text label. We use these patches to build the legend.
- Despite the differences in the way we construct the plot's data, the rest of the function is unchanged.

Suppose we wanted subplots for each of the states. This is often useful for data sets where one axis is aligned, but the other axis has different scales.



We can use the `subplot2grid` command to help out.

```
1
2  def plot3(data, states, years):
3      colors = plt.cm.Set1(np.linspace(0,1,len(states)))
4      for i, state, c in zip(count(), states, colors):
5          ax = plt.subplot2grid((len(states),1),(i,0))
6          ax.fill_between(years, data[state], color=c)
7          ax.set_ylabel("Count")
8          for tick in ax.yaxis.get_major_ticks():
9              tick.label.set_fontsize(8)
10
11     plt.tight_layout()
12     plt.xlabel("Year")
13     plt.savefig("out3.png")
```

- `pyplot.subplot2grid()` takes a two required arguments:
  - a tuple that specifies the grid shape: (height, width)
  - a tuple that specifies the starting point of the top left corner: (y,x), where (0,0) is the top left of the entire plot.
- Notice that the grid shape is the same for each `subplot2grid()` call, however each successive call places that particular plot at a new starting location.
- To create n subplots, the pattern is:
  - (create subplot$_i$, add elements to the sublot$_i$)
  - call `pyplot.tight_layout()`
  - add plot-wide elements.
- Optional arguments `rowspan` and `colspan` allow subplots to take different sizes.