Question 1

Q1 is about generating descriptive statistics with Numpy. If you're dealing with code that looks like `np.something`, you're using Numpy.

Important health tip: you have to execute each of the "In [ ]" blocks in order. Otherwise you'll get messages about things not being defined and so forth.

Here are the functions I used to solve question 1.

```
# Given a numpy array, call it 'dat':
dat_sum = dat.sum()        # sum of all values in dat
dat_n = len(dat)           # number of values in dat, aka 'n'.
dat_minus_2 = dat - 2      # subtract two from every cell, get a new array
dat_abs = np.absolute(dat) # absolute value of every cell, get a new array
dat_sq = np.square(dat)    # square value of every cell, get a new array
dat_med = np.median(dat)   # get median value of dat
dat_med2 = np.percentile(dat, 50) # another way to get median value of dat

foo = val / len(dat)       # divide val by n. if you need to divide by (n-1)...
bar = val / len(dat)-1     # due to PEMDAS, this is probably NOT what you want!
baz = val / (len(dat)-1)   # this IS probably what you want.
```

1A is about calculating the *mean*, and should be one simple line of code.

1B is about computing the *variance*. It could be one line of code, but I broke my solution down into several steps.

1C is about computing the *sample mean absolute deviation*. The instructions ask you to use `np.percentile`, though I used `np.median` because it seemed simpler.
1D asks you to type in LaTeX showing the steps that take you from one form of an expression into another. My only advice here is to write it on paper first, and then consult this LaTeX syntax guide for spelling it out: https://en.wikibooks.org/wiki/LaTeX/Mathematics.

Question 2

Question 2 is about the Pandas library. If you're using `pd.something` you're probably using Pandas.
Pandas is all about 'data frames'. You can think of data frames as basically being tabs in a spreadsheet. There's a bunch of functions, filters and other doodads that let you programmatically do the things to data frames that you'd also be able to do in a

spreadsheet. And as you do this, you'll create *new* data frames (so, new tabs in a spreadsheet), and manipulate *those*.

2A is about creating a new `DataFrame` object using `pd.DataFrame( { .. } )`. `pd.DataFrame` function takes a single Python dictionary literal. Your solution should be one simple line of code.
2B is about generating summary statistics from a single column of the data frame. `someDataFrame.X` selects column `X`. You can also write it as `someDataFrame['X']`. Throwing a `.mean()` on the end will calculate the mean of that column. This can be one or two lines of code, depending on if you want to break it down into steps.
2C combines Pandas data frames with numpy calculations. Use `np.median`, `np.abs` as well as `df.o` and `df.o.median()`.
2D has you selecting (maybe 'filtering' is a better word?) particular data from one data frame to make a second data frame. The syntax for filtering sort of looks like an array access: `df[df.n == 8]` means "make a new data frame based on `df` using only those records where the `n` field is `8`. You can also put inequalities in there too, so `df[df.foo < 100]` selects records where the `foo` field is less than a hundred.
2E is about the `groupby` function. When you run `df.groupby('foo')` it creates a new `DataFrameGroupBy` object. These objects are different from the data frame objects that you used earlier. They're analogous to a spreadsheet that is sorted by a column, or columns plural. To sort by several, give it a list instead. Like if your data frame is your music collection, you could do `df.groupby(['artist', 'album'])` to order things primarily by artist, and then by album. The rest of the data is then grouped according to those two keys, and you'll be able to do math on them. You can select stuff off of a group-by object like you do for a data frame: `df.groupby('foo').M` selects column M of the grouped thing. To conclude this gigantic hint, the solution should be one or two lines.

Question 3

Question 3 is about plotting data graphically. But really it is an exercise in figuring out how the `matplotlib.pyplot` library works. It is aliased as `plt` at the top of the notebook, so whenever you see `plt`, you're using `matplotlib.pyplot`.
`matplotlib` is similar in spirit to `gnuplot`, so some of you might have a leg up on this one. One strategy to doing these questions is to have [the matplotlib tutorial] on standby. Another is to see if the following examples gets you there.

```
plt.plot(xData, yData, 'bo', label='samples') plt.ylabel('Speed in km/2')
plt.xlabel('Label for X axis')
plt.axhline(300000) # horizontal line at y=300000
plt.legend()        # show a legend
```

3A: This is your first foray into making a graphic using `plt.plot`, which makes a *scatterplot*. You'l plot the experiment trial numbers on the x-axis and the speed of light values captured for each trial in the y-axis. Use `np.arange(1,101)` to generate the list of trial numbers `1..100`, and `speed + 299000` for the speed numbers. The 'bo' indicates blue dots. You'll label the axes using the `plt.xlabel` and `plt.ylabel` functions (see above). Then you'll add two horizontal lines, one for the experimental mean plus `299000`, and another for the currently accepted value `sol`. Use 'b' for the experimental mean (==blue) and 'r' for the accepted value (==red). Then toss on a `plt.legend` and call it done.
3B: Use the same techniques as *3A* but color the dots differently according to which experiment group they're in. To do this you'll use the `expt` array, which has twenty ones, then twenty twos, and so on. Each element of the `expt` array indicates which experiment group that the corresponding element of `xpt` was in.
To filter your `xpt` (that's 1, 2, 3, 4, .. 100), you can use the filtering syntax on it: `xpt[True]` will filter out *nothing*, while `xpt[False]` filters out everything. If you happen to have a number `exptNum`, then you can use it to only filter for the experiment numbers like this: `xpt[exptNum == expt]`.
The thing inside the square brackets will be used to filter. Say that `exptNum` is 3. `exptNum == expt` will then be `True` for only those elements of `expt` that are 3. So by doing this, the whole expression `xpt[exptNum == expt]` will give you the trial numbers for only those in experiment number 3.
Do the same thing with `spd`, e.g. `spd[trialNum == expt]`.
I used a for-loop to get my `exptNum` to go from 1 to 5 inclusive. And the plot command looked like this (replace the `<placeholders>` obviously):

```
plt.plot(<filtered trial numbers>, <filtered and np.mean()-transformed experiment data>, <color goes here>, label='Expt #' + str(exptNum))
```

3C: Boxplots! To get it to look just like the example, you can use `plt.subplots` (note the "s" at the end, this is subplots *plural*).

```
fig,axes = plt.subplots(nrows=1, ncols=2, sharey=True, figsize=(8,4))
```

Lots going on in that line. This is saying: "Yo. Give me a 1 row by 2 column set of subplots, make them share the vertical (Y) axis, and the overall thing be 8 wide by 4 high." Science doesn't understand what units the 8 and 4 are in, but you can twiddle with the numbers until you're happy.

The `fig, axes` variables on the left will hold the return values. You can ignore `fig`. `axes` is a multidimensional array of axis objects. In our case we have 1 row and 2 columns, so you can reference the axes with `axes[0]` for the one on the left, and `axes[1]` for the one on the right. Use each `axis[i]` to plot stuff, just like you do with `plt`:

```
axes[0].boxplot(yData, ['Label for this Plot'])
```

Don't worry about placing the "Current C" text with the arrow. If you're curious, that is done with `axes[0].annotate(..)`.

Here's a gigantic hint. Actually, this is exactly the code to use to generate the right-hand box plot with the combined experiments. I'm including this because it is absurdly hard to describe how to do this without actually showing you how to do it.

```
axes[1].boxplot([speed[expt == x] + 299000 for x in [1,2,3,4,5]])
```

Notice that we're referring to `axes[1]`, which indicates that we're using the second of the two subplots. The data to plot is speed data for experiments numbered 1, 2, 3, 4, and 5. That syntax inside the parens for `boxplot(..)` is completely foreign to me, and if I don't even understand it, my guess is most of you won't be able to get this one right.
So, there. With a bow on top.

Question 4

The last set of questions is about loading data from files, processing them, and showing graphs. As in question 3, this uses Pandas and Numpy to do the data manipulation, and `matplotlib.pyplot` to do the graphics.
4A: Prepare data for a chart. It will show the total number of wins for each team in 1998.

```
teams = pd.read_csv('Teams.csv')           # load data from local file, put it into
dataframe called 'teams'.
dat = teams.groupby(REDACTED)              # organize the data by sorting first by
year, then by team.
dat = dat[REDACTED]                        # select the 'wins' column (it is called
'W', because 'Wins' is too long?)
dat = dat.REDACTED                         # within each group (year and team), add
up the number of wins.
w1998 = dat.REDACTED                       # isolate just the wins for the year 1998
```

You *could* do this all on one line, but for your own sake, don't get into that habit.

Each line here manipulates the data and stores it back into the `dat` variable. It is only at the very end that we store the final result in the `w1998` variable (because that's what the code checking assertions want it to be called).
As you proceed with this step, I recommend printing out `dat` after each step to make sure you're on the right track.
4B: This task is about making a "stacked barplot" of the average salary for each team from 1990 through 2010. The brainmelt here occurs when you need to use the `unstack` function. A picture works better than words, so here you go:

DataFrameGroupBy

Team	Year	Salary
Yankees	1995	340
Yankees	1996	370
Yankees	1997	390
Cubs	1995	290
Cubs	1996	295
Cubs	1997	310
Twins	1995	270
Twins	1996	290
Twins	1997	330

Unstack.

Fake data typed into a spreadsheet.

Say this was a Pandas 'DataFrameGroupBy' (the output of a groupby call). Notice how the first two columns have redundant stuff. But each row has a unique (Team, Year) key.

The 'unstack' function will take the data from this form with nine rows into the following table with 3 rows and 3 columns, and the nine salary values on the inside.

DataFrame

	1995	1996	1997
Yankees	340	370	390
Cubs	290	295	310
Twins	270	290	330

Like last time, here's the code with the interesting parts redacted.

```
salaries = pd.read_csv('Salaries.csv') # Load data from a file
dat = salaries.groupby(REDACTED)       # Group by year first then by team
dat = dat.REDACTED                     # Within those groupings, get the average
salary
dat = dat.loc[REDACTED]                # Filter so we only get years 1990--2010
inclusive
dat = dat.unstack()                    # Unstack
sals = dat.fillna(0)                   # Replace missing data with zeros and store as
'sals'
```

4C: Scatterplot again, this time using both the team wins data and the salaries data. The upshot is that if you look at Oakland (and color it with a nice red dot), you can see that they became much more efficient over the years: they won substantially more games and spent about the same amount.

You'll use the `groupby` function on by the team and salary data, and then filter for three years: 1998, 2003, and 2013, and make a scatterplot for each year. (With a nice red dot for Oakland.)
Here's the code for doing it for just 1998. I'll leave it up to you to figure out how you want to do the other two years. I used a for-loop.

```
cols = ['yearID', 'teamID']                # for both teams and salaries, these are
the columns we care about
ww = teams.groupby(cols)['W'].sum()        # group by year and team, filter on wins,
get the sum.
ss = salaries.groupby(cols)['salary'].sum() # group by year and team, filter on
salary, get the sum.

# consider printing out ww and ss here to see what they look like. you might find it
educational. Like, why
# don't the team IDs have the same format? and how could it possibly work like this?

year = 1998
```

```
# get a new data frame for 1998 wins/salary. this has columns 'teamID', 'Payroll',
'Wins'.
# For wins, use the 'ww' DataFrameGroupBy and filter with the year in question.
# For salary, do the same thing but with ss.
dat = pd.DataFrame({ 'Wins' : ww[REDACTED], 'Payroll' : ss[REDACTED]})

dat.plot.scatter('Wins', 'Payroll') # Plot it to a scatter plot

# also do the little red dot for Oakland so you can see them
# getting more efficient as the years progress
plt.title('Year ' + str(year))
plt.plot(ww.loc[year, 'OAK'], ss.loc[year, 'OAK'], 'ro')
```