Scientific Computing, Fall 2024
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2024/index.html

Some coding standards for Assignment 1.

Professionalism and attention to detail are what mathematicians call *necessary conditions* for creating useful computational software. This class will treat some issues of software engineering and procedures for creating and verifying computational software. My goal is to convince you that the time spent (wasted??) doing things "right" will be saved in bugs and confusions avoided later in the project.

The first assignment will introduce some computational coding standards. Later assignments will add to these. Some of these apply to all coding in any language. Some apply mainly to Python and languages like Python. Some apply mainly to computational software.

Please download the file `CodingStandards.py` and open it in an editor that shows line numbers. The code you hand in for assignments should follow each of the standards below, which are illustrated in this code. You don't have to do everything exactly as I do it, but what you do should accomplish the same thing.

Style. There are style guides for Python that you should be aware of. You don't have to follow all those rules exactly, but you should be aware of them. If you break them, you should have your personal modifications that you always use. For example, style guides say to indent by 4 spaces. I always use 3, because 4 spaces make (I feel) too much indenting in code with many levels of code block nesting. The guides say never to put anything past column 72, but I violate that rule a little (a few characters more in a line, and only when he helps a lot).

Header. Lines 1-6 are header lines. They tell you who wrote (with contact information) the file, what the language is, when it was written, what it's for, etc. Modules in larger projects might say what project they're for, what the history of the file is, list the names of contributors, say which version it is, etc. Software management tools such as `git` have ways to keep track of some of this information. But in my experience, it also helps to put it at the top of any module. Please do this in your assignments even if it seems pointless. Hopefully this is habit forming.

Names and labels. Lines 8 and 9 import standard packages and use standard names `np` and `plt` for their namespaces. Python lets you choose different names, such as `n` for `numpy`, but then others would have trouble reading your code.

Names. Choose variable and method names that help the reader understand what it does. Style guides have rules for capitalizing and for using underscore to separate words. For example, line 11 uses the name `BinProb` to follow the rule that methods have capitol names and words are indicated by *camelcase* (starting each word with a capitol). Other people prefer to not to capitalize and to use underscore, as `bin_prob`. You decide, but stick to your convention. My personal view is that variables that will be used in formulas should have names that follow mathematical conventions. Long variable names lead to formulas that are hard to read and don't fit in the 72 character limit. For example, line 11 uses `p` for the probability called p in mathematical writing, etc. Mathematicians often define $q = 1 - p$, so line 22 should be easy to understand and remember. If you had instead used names `success_probability_per_trial` and `number_of_trials` for p and n, then the variance formula on line 102 would have been

```
var = number_of_trials*success_probability_per_trial*(1.-success_probability_per_trial)
```

I think line 102 is more readable as it is.

Docstring. Lines 12 through 20 are a *docstring* for the method `BinProb`. A docstring should say what the method does, what its inputs are and what it returns. Give the types or type limitations where necessary. A bunch of comment lines following the `def` command do almost the same thing (explain the method), but some code aware editors use the docstring in a way they could not use comment lines.

Comments. Use comments to explain the choice of variable names and how a variable will be used. Line 23 has a comment that explains that `bp_k` should be understood as `bp[k]` for the current k. Mathematically, the binary probability distribution probability that $k = 0$, is $\Pr(k = 0) = q^n$. Lines 43 through 45 have comments that explain the variable name or explain the formula being used.

String formatting. Lines 60 and 61 are an example of *string formatting.* The stuff in curlies $\{\ldots\}$) in the string on line 60 is *formatting information*, which you can read about in the Python documentation. They have the form {`[name]:[format]`}, which means that the object `name` is bound to should be *formatted* (turned into a string) using the `[format]]` specifications. Here, the variable `p` should be formatted using `7.2f` and `n` should be formatted using `4d`. The format `7.2f` says to use 7 characters in all (some of them blank), with 2 being after the decimal point, and using *fixed point* format. In this format, 100/3 would be rendered as `33.33` (two leading blanks and five printed characters, one of them being the decimal point), and 1/30 would be rendered as `.03` (four leading blanks, a decimal point, two decimal digits). The format `4d` says to treat n as an integer and print it using four characters, using leading blanks as necessary. Line 61 says to use the `format` method (attribute) of the string `outLine` with the `p` in the string equal to the `p` in the code (also for `n`). This results in the float `p` being rendered with the `7.2f` format, etc. The result is

```
   binomial probabilities with p =    0.23 and n =   20
```

Lines 63 to 69 show the value of this kind of string formatting. Line 63 prints column headers for a table. The `k`, `prob`, etc. are spaced so that they are over the columns that will be printed. The `\n` at the beginning and end of the string of line 63 are like typing *return*. They make blank lines above and below the `k prob ....` . Line 65 says to use `4d` to render `k`, `10.2e` to render `pr`, etc. The `10.2e` means to use *exponential* format as though `pr` is a float. It says to use 10 characters in all (leading blanks as needed) then a sign character if needed, then one digit, a decimal point, and 2 more digits followed by `(+- exponent)`. For example, 100/3 would be rendered as `3.33e+01`, which means $3.33 \cdot 10^1$. This is eight characters in all, so there would be two leading blanks. Lines 66 through 68 say to to use `format` on the string `outLine` with `k` given by the code's `k`, `pr` given by the code's `bp[k]`, etc. Running this program gives output

```
k       prob       CLT prob      diff       ratio

  0     5.37e-03     1.07e-02   -5.32e-03    5.02e-01
  1     3.21e-02     3.40e-02   -1.95e-03    9.43e-01
  2     9.10e-02     8.16e-02    9.37e-03    1.11e+00
                        .
                        .
```

The numbers line up in columns that are easy to read. They line up because each number is formatted to the same number of characters. The exponential format allows for numbers of different sizes. For example, the first one is $5.37 \cdot 10^{-3} = .00537$.

Reality checks. Lines 71 and 72 are simple reality checks to see that the total probability is equal to one. It's important to have such checks, which catch many bugs. You might not want to leave these lines in the production version.

Plotting. Lines 84 through 94 illustrate several things you should always do when plotting. Line 84 returns a tuple containing `fig` (an instance of the `figure` class), and `ax` (an instance of the `axes` class). The `axes` object holds plots and a `figure` object holds one or more `axes` objects, but just one in this case. Lines 86 and 87 draw lines to plot the binomial and CLT probabilities. They also give labels for the curves that will go the legend. Line 89 draws faint grid lines across the plot to make it easier to tell what the values in the plot are. Line 90 says to put the legend in the plot. Line 91 creates a plot title using LaTeX formatting (which is that the `r` means). The formatted title contains the values of n and p. This way you can run the code over and over with different n and p values and tell which plot corresponds to which run. Lines 92 and 93 label the horizontal and vertical axes, also using LaTeX formatting. Line 94 saves the plot in a file. You could, instead, make the plot appear on the screen and save it "manually" (typing a save command in the screen window), but that is cumbersome and takes longer. Automation, even must automatically saving plots to files, saves a lot of time and avoids mistakes in the long run.

You will notice that much of the plot `LinearScaleWholeRange.pdf` is uninteresting. The whole right half is a flat line at zero. When you see this kind of thing, you should take the time to plot only the interesting parts. Lines 98 to 120 plots the same data, but only in a range within some number of standard deviations of the mean. Plotting uninteresting lines that don't convey useful information is a sign of laziness and lack of interest.

Changing the plot scaling can also reveal information that you cannot see otherwise. Lines 125 to 135 plot the same probabilities on a log scale so that you can see how close the numbers are to zero.