HOMEWORK PHZ3150: **Introduction to Numerical Computing**
SPRING 2021: **Homework Format and Coding Style Guide**

1. **Homework submission**: Hand in all homework on WebCourses as a single .zip file per assignment, unless otherwise stated, including prose and math. Make a directory called `~/phz3150/handin/hw3_<username>` on your computer, substituting the right assignment number and your username (in Unix, the ˜ means your home directory, such as /home/sushi for user sushi). Before class, put the files you wish to hand in in that directory. Submit `hw3<username>.zip` on WebCourses.

   WebCourses automatically cuts off submissions at the beginning of class. No late homework will be accepted, so **be sure you actually save your files before class starts**! Do not email your homework. Also, make sure all necessary plot and program files (such as functions you are asked to write) are in the directory. We grade by running your program. If it produces errors or fails to produce the assigned output (plot files, etc.), the assignment is incomplete. The best way to ensure a complete assignment is to do this yourself in a fresh Python session from the zip file you plan to submit, and check that everything works and that it produces all the files it needs to.

2. **Main homework file**: Most assignments are programs. Put the answers to the assigned problems in a single, executable, cleanly-coded and commented, Python source-code file named, for example, hw3sushi.py or hw3sushi.ipynb. This is your “main homework file.” It should contain nothing but your homework answers; it is not a log file.

3. **What to print**: In your main homework file, print your name, the class, and the homework assignment number. Print the problem number and any output for that problem. It might look like:

```
Susan Shih
PHZ 3150

HW 3
=== Problem 1 ===
17
[3, 4, 5]

=== Problem 2 ===
Hi there.
=== Problem 3 ===
...
```

4. **Function files**: If you write functions, whether or not they are specifically assigned, put them in one or more separate, non-executable Python source files with reasonable names (e.g., gaussian.py) and import them. This is not a requirement for the early assignments, but it is required later on in the course. If a problem asks you to write a function, state in a comment what file the function is in. This might be the only answer given in the main homework file for that problem.

5. **Function documentation**: All Python functions must have docstrings. For functions specifically assigned in problems, follow the docstring template given in class. Until the template is presented, include a line or few that describes what the function does. If you write helper functions that are not specifically asked for, you may instead include just a single-line docstring.

6. **File formats**: Depending on your coding preference you may save the coding file as a .ipynb (notebook) or a .py (Spyder). Use plain ASCII text files wherever possible, and certainly for all data tables. Other allowed formats: PDF (for writing), FITS or HDF5 (for data, as appropriate), PNG (for images of plots), JPEG (for scans or other photographic images). You can convert MSWord documents to PDF with "Save as..." PDF; check the output before handing it in.

   Math (only) may be handwritten on paper, scanned, and handed in as a PDF or JPEG, though a typeset PDF answer is preferable. Grammar, spelling, and complete sentences count for grade, even in math (remember that "=" is a verb). Math problems must show your logic and calculations. Box or circle final math answers. Do not hand in extraneous files, such as "~" backup files saved by text editors.

7. **File name**s: If the problem asks for a plot, image display, screen shot, or other graphic, include commands for output to a file as part of the program, and commands to put the plot on screen in comments, if they differ. Include the files it makes in your directory when you hand it in (see below). Plots should have titles and sensible axis labels, including units. Put each item in a separate file. The filenames should follow the format: `hw3_sushi_prob2_graph1.png`. Only if requested, put ASCII output to the screen (like tables) in files named like `hw8_sushi_prob2_data1.dat`. ASCII tables should have titles and column headers that distinguish them from one another and that make sense to the reader. You may hand-edit headers onto tables written by the computer.

   Do not invent additional naming components for assigned files. Use graph for anything visual and data for all text files. If you feel the need to make more plots or other output than requested, for example to demonstrate a problem, use `_extra_` instead of `_graph` or `_data`, so that your graph3 file contains your version of the same plot as everyone else's graph3.

8. **Text file contents**: In comments at the top of the main homework file, all other program files, and all table files, state your name, the course number, the assignment name (e.g., HW3) and problem number (if applicable), and the date. In the main homework file, put the problem numbers in comments at the start of each problem. Be sure to comment out any answers that are not commands, such as justifications or explanatory notes, or your homework will crash!

9. **Homework program requirements**: If asked to calculate a value, you must print the value to the screen and also put it in comments in the main homework file. This ensures that you will get some credit, even if your program does not run for us. Make sure the tab character does not appear in your code, even in comments. While they may work for the normal Python interpreter (when run as a program), if tabs are cut-and pasted to the ipython interpreter, they have special meaning. This means the code won't work if we try to run it line by line. Python will use spaces to indent.

10. **Test your work**: Develop the habit of running your homework in a clean directory and Python session and then checking your work after your final run and before handing it in. Look at your plots and `hwX_sushi.out`. Do plots have axis titles? Do they look right? List the directory. Do you have "~" files? Duplicate plots? Not only will this get you higher grades, it is a crucial habit for a successful scientist. If you are right 99.5% of the time (!) and you have 200 steps in a real analysis, chances are your analysis is wrong. Nobody is initially right 99.5% of the time. By checking, you can make it likely you'll be right in the end.

11. **Dos and Don'ts - Our Coding Style Guide**:

    (a) All integer-valued floating-point numbers should have decimal points after them. For example, if you have a time of 10 sec, do not use `x = np.e * 10`, use `x = np.e* 10.` instead. For example, an item count is always an integer, but a distance is always a float. A decimal in the range (-1,1) must always have a zero before the decimal point, for readability:

    ```
    x = 0.23  # Right!
    x = .23   # WRONG
    ```

    The purpose of this one is simply to build the decimal-point habit. In Python it's less of an issue now, but sometimes code is translated, and integer division is still out there. For that reason, in other languages, it may be desirable to use a decimal point even for counts, unless integer division is never wanted. Make a comment whenever you intend integer division and the language uses the same symbol (/) for both kinds of division.

    (b) Use spaces around binary operations and relations (`=<>+-*/`). Put a space after ",". Do not put space around "=" in keyword arguments, or around "**".

    (c) Do not put `plt.show()` in your homework file! You may put it in a comment if you like, but it is not necessary. Just save the plot. If you say `plt.ion()`, plots will automatically show while you are working.

(d) Use:

```
     import matplotlib.pyplot as plt
```

NOT:

```
     import matplotlib.pylab as plt
```

Refer to `example.py` if you don't remember which, or remember that pyLAB is like matLAB, and that's not what we're after. Why? Pylab loads `pyplot` (good), but also imports a lot of NumPy into the Matplotlib namespace. The latter behavior is why you can say `plt.linspace()` when it is really `np.linspace()`. The reason they did this was to make Matplotlib work more like Matlab. But, we don't care about that. Loading `matplotlib` does not turn on the interactive mode that displays all plots. You can do that with `plt.ion()`. See chapters 6 and 17 of the Matplotlib manual for more detail.

(e) Keep lines to 80 characters, max, except in rare cases that are well justified, such as very long strings. If you make comments on the same line as code, keep them short or break them over more than a line:

```
code = code2 # set code equal to code2
# Longer comment requiring much more space because
# I'm explaining something complicated.
code = code2
code = code2 # Another way to do a very long comment
# like this one that runs over more than
# one line.
```

(f) Keep blocks of similar lines internally lined up on decimals, comments, and = signs. This makes them easier to read and verify. There will be some cases when this is impractical. Use your judgment (you're not a computer, you control the computer!):

```
x    =   1.     # this is a comment
y    = 378.2345 # here's another
fred = chuck    # note how the decimals, = signs, and
                # comments line up nicely…
alacazamshmazooboloid = 2721 # but not always!
```

(g) Put the units and sources of values in comments:

```
t_planet = 523. # K, Smith and Jones (2016, ApJ 234, 22)
```

(h) Align similar, adjacent code lines to make differences pop out and reduce the likelihood of bugs. For example, it is much easier to verify the correctness of:

```
a         = 3 * x + 3 * 8. * short1        - 5. *...
a_altname = 3 * x + 3 * 8. * shortothernum - 5. *...
```

than:

```
a = 3 * x + 3 * 8. * short1 - 5. * np.exp(np.pi * omega *
t)
a_altname = 3 * x + 3*9* shortothernum - 5. * np.exp(np.pi
* \ omega*t)
```

(i) Assign values to meaningful variables, and use them in formulae and functions:

```
ny = 512
nx = 512

image = np.zeros((ny, nx))
expr1 = ny * 3

expr2 = nx * 4
```

Otherwise, later on when you upgrade to 2560x1440 arrays, you won't know which of the 512s are in the *x* direction and which are in the *y* direction. Or, the student you (now a senior researcher) assign to code the upgrade won't! Also, it reduces bugs arising from the order of arguments to functions if the args have meaningful names. This is not to say that you should assign all numbers to functions. This is fine:

```
circ = 2 * np.pi * r
```

(j) In addition to the docstring (see above), functions should be internally commented. Say where information comes from, give information about algorithms chosen, the logical flow, and any coding choices you made.

(k) If you modify an existing function, you must either make a Git entry or, if it is not under revision control, include a Revision History section in your docstring and record your name, the date, the version number, your email, and the nature of the change you made.

(l) Choose variable names that are meaningful and consistent in style. Document your style either at the head of a module or in a separate text file for the project. For example, if you use CamelCaps with initial capital, say that. If you reserve initial capitals for classes, say that. If you use underscores for variable subscripts and camelCaps for the base variables, say that. There are too many good reasons to have such styles for only one to be the community standard. If certain kinds of values should get the same variable or base variable, such as fundamental constants or things like amplitudes, say that.

(m) It's best if variables that will appear in formulae are short, so more terms can fit in one 80 character line.

Overall, having and following a style makes code easier to read. And, as an added bonus, if you take care to be consistent, you will write slower, view your code more times, and catch more bugs as you write them. Thus, for codes of any significant size, writing pedantically commented and aligned code is almost always faster than blast coding, if you include debugging time.