Getting started with Python

Many computational assignments for this class use `python 2.7`. You will need the basic Pyghon program and three packages: `scipy` (pronounced "sigh-pie"), `numpy` (pronounced either to rhyme with "scipy" or with "lumpy"), and `matplotlib`. There is a good chance your computer already has these packages installed. The CIMS network machines do. If not, you can download and install it from many sources. I (on a Mac laptop using OS 10.10) have had luck using `homebrew` and `anaconda`.

Python is a *scripting language*, in the category of `Matlab` or `R`. The Wikipedia page `https://en.wikipedia.org/wiki/Python` (click on *programming language*) has more background. The tutorial `http://docs.python.org/2/tutorial/` is a great reference. Most programmers today use a search engine to answer programming questions. For example, if you want to know how to format output, search on: `python output format`.

Download the file `PythonBasics.py` to some directory, go to that directory and type: `python PythonBasics.py`. You should get output that starts:

```
Hello world, from python
Hello world, again, from python
A final hello world from python
```

Look at the file, which illustrates some python basics. The bullet numbers in the comments correspond to the bullets below:

1. The first thing you do in any new programming language is *Hello world*.

2. A *string*, or a *character string*, is a sequence of characters. This makes the variable `WelcomeString` equal to a string. The Python statement `print [string]` prints the string. That's what the bullet 1 line does, but without giving a name to the string.

3. The *concatenation* operator `+` creates a string that is the first one followed by the second one.

4. Python infers (guesses) the type of a variable when it is assigned. This statement creates a variable `x` of type `int`. The variables above all have type `string`.

5. The function `str(arg)` creates a string that represents the value of `arg`. You concatenate these with some white space (blanks) and symbols to get readable output. Good programming practice requires that you do this.

6. Since `x` and `y` are integers, we get an integer divide, which is an integer.

7. The type of `x` can change at any assignment. In this case, what was an `int` becomes a `float`. Python re-typing is supposed to be convenient, but it also is the the source of bugs. You should always be aware of the types of your variables.

8. The floating point divide gives the floating point answer, which is different. Note that the two output lines do not line up: the "3.3333" is not directly below the "3". This makes the output hard to read. It is unprofessional. The argument of `print` is a string, which can be a named variable or not.

9. Python variables, internally, are complicated objects. They know more about themselves than just their values, which is a major difference between Python and C/C++/Fortran. Among other things, a Python variable knows its type. The function `type(arg)` returns the type of *arg*. The output uses concatenation to put in punctuation and white space.

10. A *list* is a Python datatype that is something like an array. The empty list is $\widehat{[]}$. This statement creates a variable `DumbList` of type `list`.

11. The datatype `list` is a *class* in Python. If an operation (technically, a *method*) has been defined for variables (objects) of a given type, you invoke the operation using `var.op(args)`. Here, the operation is `append`. The thing you want to append is `x`. The object (variable) you want to append it to is `DumbList`.

12. A *list* is different from an *array* in that the members of a list need not have the same type. With this append, *Dumblist* has a `float` followed by an `int`.

13. The Python function `len(arg)` returns the length of `arg`. To be consistent with the class nature of the datatype `list`, it really should be `DumbList.len()`. But programming languages are never consistent in this way. That's why you always have to use the search engine to find out how to do things.

14. A `for` loop in Python is a little different. It loops over objects in a list rather than values of a variable. The *control statement* of the `for` loop ends with a colon. The *body* of the `for` loop is all the statements that are indented, two statements in this case. The first statement is too long for one line, so it is *continued* onto three lines, the backslash being the continuation character. There can be no spaces or comments after the \, which is another design flaw of Python. I aligned the parts of this statement (text over text, variable over variable) to make it easier to read. You may lose points if your code is not easy to read in this way. It also helps you spot certain kinds of bugs, if things do not line up as they should.

15. The `str` function applied to a list produces: `[item 0, item 1, ... ]`. They have different types, first `float`, then `int`, then `str`.

16. This seems to create another object `SmartList` and give it the value stored in `DumbList`. But that's not what it does, see bullet 19.

17. You can access items in a `list` using C/C++ style indexing. Here, you make the first item, which was `10.0`, a string.

18. See that the first element of `DumbList` has changed as it should.

19. The corresponding element of `SmartList` has changed too. This is because the the line at bullet 16 did not make a separate copy. It only made `SmartList` *point* to `DumbList`. The values of the items are still stored in the same memory locations. A truly independent copy is created using the Python function `deepcopy`.

20. There is no shallow copy/deepcopy issue with simple datatypes. Python is inconsistent in that way.

21. The function `range(m,n)` produces a list of the numbers starting with m and ending at n-1.

22. `range` is the Python mechanism for C/C++/Fortran style iteration loops. `range(m,n)` produces a list of the values `i` will take. In C++ you would write: `for ( int i = m; i < n; i++){ loop body }`. The Python `range` function includes `m` and excludes `n` to make these statements correspond.

23. Some numerical computing!

Active learning is a must! Do something for each bullet point above to make sure you understand the point being made. Some can be trivial, like saying "goodbye world" instead of "hello" for bullet 1, or changing the name of the string variable in bullet 2. Try breaking things to see what happens. Delete the colon in bullet 14 and see the error message. Also, indent the second statement `ListNumber = ...` one extra space. Put a space (white space = cannot be seen in the editor) after the \.

Next, download `aliasing.py` and run it. It should save a plot called `aliasing.png` in the same directory, which is the same as the one that appears on the screen. You have to close the plot on the screen to finish the python script. The program illustrates some points of numerical Python programming practice. It also illustrates the principle *aliasing* from class. There is more than one wave vector that describes the same grid function. The red dots in the plot represent the grid function. The solid and dashed lines represent Fourier modes with different wave vectors, modes that happen to agree at the grid points.

1. Every file should start with a header that says what the file is, who wrote it (with contact information), what the file is called, and what it does.

2. This tells the Python interpreter to "import" the procedures in `numpy` and to refer to them as `np.` For example, you evaluate $q = \sqrt{2}$ using `q = np.sqrt(2)`. Saying `import numpy as nmp` would force you to write `q =`

`nmp.sqrt(2)`. There is always a tradeoff between clarity and conciseness. You should follow the custom of calling numpy np if you want others to be able to read your code.

3. `pylab` is a collection of plot routines that resemble Matlab. For example (see bullet 11), they use Matlab style conventions for line style.

4. There should be a comment for any interesting line of code unless it is totally obvious. In this case, it's obviously an assignment, but what does the variable n mean?

5. I first typed `L = 4`. This was a bug because it made `L` an integer instead of a float. That gave `dx` the value zero.

6. The `zeros` function in `numpy` creates an object of type `ndarray`. This is like a C/C++/Fortran array, a sequence of objects of the same numeric type (float by default). This is like the Matlab command `zeros`. I don't know how to allocate an `ndarray` without filling it with some values. There are other numerical datatypes in numpy, which we will use later.

7. If I wrote the code well, it should be clear what this is.

8. The function `subplots` is part of `matplotlib.pylab`, which is called `pl` (see bullet 2). It returns two things, the second of which is the plot object named `ax`. The plot object `ax` empty, like the empty list created by `basics.py` using `Dumblist = []`. We will add elements to the plot one by one. I wrote this code by searching on "matblotlib examples" and simplifying one of the simplest examples.

9. This defines a string that will be in the *legend* box in the plot. It is a string that knows the value of k. This is an essential part of automating computing and data analysis. You don't type legends and plot titles, you write scripts that do it for you. That way the plots always have enough information for you to interpret them. I expect all computing for this class to be automated in this way.

10. This is like `DumbList.append(arg)` but with the plot. It adds the curve with $x-$coordinates `xa` and $y-$coordinates `f1` to the empty plot. The curve gets label `l1`, which will be used in the legend. It is represented by a solid blue line; `'b'` means blue.

11. The same, except that `'k--'` means: make a black dashed line ("k" is for black because "b" is for blue). The next "curve" has `linestyle` "ro", which has "r" for red and "o" for open circles.

12. Since you're automating, you can take the time to do it thoroughly. Label the x and y axes. Note that the title also has information about the parameters used for the run.

13. Save the plot to a file.

14. Make the plot appear on the screen.

Play with this program. If you replace `np.zeros(n)` with `np.zeros(n-2)`, you will get an "out of bounds" message, which shows that Python does array bounds checking. C/C++Fortran do not. Add the statement `np = 10` before `xm = np.zeros(n)` (I made a mistake like this preparing this routine.). You will get an error message that tells you that the `int` variable `np` does not have a method called `zeros`. Python is an *interpreted* language, not a *compiled* one. *Running* a Python script is like running a Matlab script, not a C/C++/Fortran *program*. The Python *interpreter* (a program called `python`) reads the script file line by line and does what each line says. The line `np = 10` says: "Whatever np might have been before, now it is an integer with the value 10." The next line, `xm = np.zeros(n)`, says: "Use the `zeros` method of the class that `np` belongs to." So the interpreter looks up the type of `np`, which turns out to be `int`. Then it asks the `int` class for its method `zeros`. The `int` class returns an error message saying it has no method with that name. The point is, you don't find out about the error until you run the script, and the interpreter gets to the line that causes the error. A compiled language could have found this error at *compile time*.