LECTURES 1–?: INTRODUCTION TO COMPUTER PROGRAMMING AND PYTHON

ANDREW D. WICKERT

1. Why do we program computers?

In order to analyze data, run simulations, check the logical consistency of our ideas, and compare models to the real world, we need numbers. Working with numbers requires quite a lot of repetitive calculation. Doing these calculations ourselves is boring, miserable, and saps our very humanity. Fortunately, computers are excellent at doing repeated tasks. Automating a computer to do our "dirty work" is what programming is all about. In order to send instructions to the computer, we need to write in a language that obeys rules of unambiguous formal logic. This is a computer programming language.

Over this course, there will be many struggles in making this communication between you and the computer work. Therefore, it is important to remember why we program computers: these reasons include to make our lives easier and more pleasant, to accomplish more than we could without the assistance of a computer, to have a perfectly logical companion who can inform us about the internal consistency of our ideas, and to build a numerical laboratory in which we can develop virtual experiments. If it seems that computers are making our lives harder rather than easier, it is time to take a step back and a deep breath, and evaluate what we are doing and why we are doing it.

2. How we program computers

Computer programming languages are a set of commands that we write in human-readable text that eventually are turned into machine-readable binary. This can happen via one of two means:

- In a **compiled language**, we use a program called a "compiler" to translate what we have written into machine language. In the past, this was done by hand – so you can thank your lucky stars for all of the dedicated engineers, mathematicians, programmers, and technicians who have made these compilers that literally tell a computer how to program itself based on your commands to it!
- In an **interpreted language** (or "scripting language"), your commands are not compiled into byte code (machine language) *en masse*. Rather, they all reference pre-compiled functions. Because of this, you don't have to re-compile the program every time you change it and your coding can be very flexible, but these programs often run somewhat slower than compiled programs. This is the trade-off: ease of programming for you costs computational efficiency in this case.

3. Python and the major programming languages

In this course, we will learn Python. It is an interpreted, multi-paradigm, multi-purpose, and easy-to-read open-source language. These mean that:

- **Interpreted:** You don't have to compile the code every time you make a change. This helps speed code development and your learning (critical in this course!) at the cost of some computational efficiency (*not* critical in this course).
- **Multi-paradigm:** This means that the language can be written as an **imperative** language, with a sequence of command that are executed in some defined order, or as an **object-oriented language**, in which "objects"—clusters of related variables and functions: think perhaps of a volume of soil that has a temperature and thermal conductivity (variables) and can conduct heat (a function)—are manipulated.
- **Multi-purpose:** Python cam be used for numerical modeling, data analysis, web development, interfacing with peripheral devices, automating tasks in GIS, and much more.
- **Easy-to-read:** As programming languages go, Python can be "read" fairly well by someone who has no experience with it.
- **Open-source:** Python is freely available and is updated and expanded by a dedicated community of volunteers. Therefore, learning how to use it will not tie your skills to any software package that must be purchased.

Many of these reasons highlight why I have chosen to work with Python for this course. However, it is important to go over a number of the other major programming languages used in the geosciences, at very least such that their names are familiar to you and you will not feel like you are starting from nothing if you have to program in one of them someday. These are:

	Compiled	**Interpreted**
Open-source	Fortran C/C++ Java	Python R Octave (Matlab clone)
Proprietary	?	Matlab IDL

C, C++, and Fortran (f77 and f95) are the most commonly-used compiled programming langauges for scientific computing. These are the true "heavy lifters" in computational science. Fortunately, Python is written in C (giving it a native interface with C code), and a compiler called "f2py" allows Fortran code to be compiled in such a way that Python can talk to it. Java is occasionally used, but is less developed in the numerical realm.

Many of your colleagues in the general geosciences use Matlab, which is a very similar language, except that it is more focused on matrix operations, data analysis, and scientific computing... and that it is closed-source, meaning that you cannot see how it actually works. Furthermore, you need to purchase a license to use it. For those of you who are familiar with Matlab already, I would suggest the Mathesaurus entry "Numpy for Matlab Users" (Numpy is the numerical package for Python) at http://mathesaurus.sourceforge.net/matlab-numpy.html.

IDL is commonly used by atmospheric scientists as well as the community involved with the ENVI GIS package. However, the other major GIS packages, both proprietary (ArcGIS) and open-source (GRASS GIS and QGIS) use Python as their interface language.

R is a programming language designed to perform statistical analyses. It excels in this field. However, outside of this scope, it is fairly limited, which is why we won't learn it in this course. After learning Python, learning R should be straightforward.

The other reasons to teach you Python are that it has functions that streamline file input and output and can connect easily to programs written in other languages, including compiled langauges, making it an effective language to "glue" together pieces of code written in highly efficient (but clunkier to manipulate) C or Fortran code. It also produces excellent graphical outputs. In short, what I am writing that I have chosen Python after thinking long and hard about the decision and using it for an extended period of time myself, and I think that learning it will be a worthwhile use of your time.

4. Practical interlude: downloading and installing

4.1. **Python interpreter and development environment.** Now we know that we can write code in Python that can be interpreted into a machine language. Great! So how does that happen?

With an interpreter, of course! You must download and install one onto your computer. Linux/*nix and Mac computers come pre-loaded with a basic Python interpreter, but we will need more packages than that, and the Mac version of the Python interpreter is often not up-to-date. Windows machines generally do not come with Python by default.

While the interpreter changes human-readable text into machine language, it does not help humans produce that human-readable text. This is where a development environment comes in. This is something that will format the text for us to help us follow the flow of the code, automatically complete expressions, and more. There are a two main options to do this:

- Command prompt and text editor. In each of these cases, I recommend **ipython** as the program to execute the Python code at the command line interface (= command prompt). Some examples of combinations of these are:
  - Windows: Cygwin and notepad++
  - Linux: Terminal and gedit
  - Mac: Terminal and textwrangler
- Spyder: a Matlab-like Integrated Development Environment (IDE)

Shell and text editor: gedit/notepad++/textwrangler and ipython or Spyder

The Python packages that we will want at the outset of this course are:

- The most recent Python 2.*.* (not Python 3.* – this is a fairly different take on Python that is not connected to the scientific programming packages)
- NumPy (**Num**erical **Py**thon): a package to handle arrays, matrix operations, and more
- SciPy (**Sci**entific **Py**thon): a package that contains special functions, interpolation techniques, linear algebra solvers, and other tools that are useful for scientific programming
- Matplotlib: a plotting library with an extensive range of applications (see http://matplotlib.org/gallery.html); we @TODO: will use the map-plotting extensions to matplotlib later in this course.

4.1.1. *Linux.* Use your package manager. For Debian/Ubuntu, type at the command line:

```
# Basic packages
sudo apt-get install \
python python-numpy python-scipy \
python-setuptools python-matplotlib

# pip (recommended for automatic installs via setuptools)
sudo apt-get install python-pip

# iPython console -- very useful (optional)
sudo apt-get install ipython

# Sypder IDE (I don't personally use it but many others like it: optional)
sudo apt-get install spyder
```

.

(The package "python" should be installed by default[1], but is included in this list for completeness.)

4.1.2. *Windows.* Download **python(x,y)** (https://code.google.com/p/pythonxy/wiki/Downloads) or another full-featured distribution such as **Anaconda**; both of these distributions have been tested successfully with gFlex. Python(x,y), Anaconda, and several others also contain the required packages (including the numerical libraries), the iPython console, and the Spyder IDE; **Spyder** (https://code.google.com/p/spyderlib/) is a nice IDE that will provide a familiar-looking interface for users accustomed to Matlab.

4.1.3. *Mac.* One recommendation is to use a package manager like **homebrew** (http://brew.sh/). With this you can install Python, and then move on to using **pip** (or homebrew) to install the Python modules. A good introduction to this can be found here: http://www.thisisthegreenroom.com/2011/installing-python-numpy-scipy-matplotlib-and-ipython-on-lion. See the **Linux** instructions for the list of packages that you will need; after installing pip, these commands can be substituted as follows, e.g.,

```
# Homebrew
sudo brew install python-numpy
# Pip
pip install numpy
```

.

Recent efforts to download Python distributions (both **Anaconda** and **Enthought**) have not met with success with both gFlex and GRASS, though **Anaconda** has been tested successfully with Windows. As a result, it should be more successful to keep the Python packages managed better by something like **homebrew** with **pip**.

5. Letters and numbers:A look from the computer's point of view

ASCII, binary, Unicode, int, uint, floating point

Python does automatic typing *but* it is important to know these for general programming!

6. Basic Python programming

The following sections provide a brief introduction to programming in Python. A more exhaustive treatment can be found in @TODO: OTHER COURSE (UMN?)

Write this or refer users to the book? Maybe focus here on examples of usage in Earth science.

6.1. **Basic types.**

6.2.

6.3. **A first program.**

[1]Once I was trying to change which version of Python I was using, and without thinking about it (or knowing nearly so much as I do now), I simply typed "sudo apt-get remove python" and entered my password and pressed enter. However, had I looked at the screen to see the list of packages that would also be removed, I would have realized that the list of core functions of my computer that depend on the Python interpreter was enormous. Long story short: had to reboot to a command-line terminal to "apt-get install" and put all of the core software back on my computer, just so I could boot to the desktop!

6.4. **Loops: for, while.**

6.5. **Logical statements and flow control: if, else if, else.**

6.6. **Exceptions: try and except.**

6.7. **Mathematical operators:** Math at last!
Floor division note (with types)

6.8. **Strings and string operators.**

6.9. **Types that combine several items: lists, tuples, Numpy arrays.**

7. Writing programs

7.1. **imperative.**

7.1.1. *Without functions.*

7.1.2. *With functions.*

7.2. **Object-oriented.**

8. General structure of computer programs