

BRYN MAWR COLLEGE

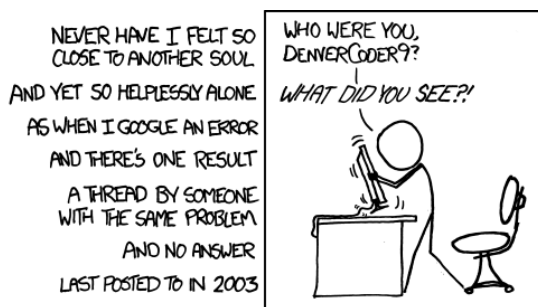
Computational Methods in the Physical Sciences

Module 0: Introduction to Computational Methods in the Physical Sciences

Prerequisite modules: None

Estimated completion time: ½ hour

Learning goals: Become familiar with basic issues in computing and key properties of computing languages



(Image credit: xkcd.com)

See http://www.explainxkcd.com/wiki/index.php/979:_Wisdom_of_the_Ancients for an explanation.)

0.1 Computational Modules

This document is the first in a series of **computational modules** designed to allow an individual to build up computational skills by working through the example problems in the modules, answering questions posed in the modules, and completing a mastery exercise (or several) at the end of each module. Some modules, including this one, may require that you consult other resources for additional information. Modules that should be worked through prior to the one being read will be indicated under **Prerequisite modules** at the top of the first page of the module. (**NOTE:** This module, and Module 1 – A Brief Introduction to Python & Programming, are prerequisites for all subsequent modules and are not listed explicitly in all modules as prerequisites.) Each module also indicates an estimated time range for reading the module and completing the exercises. If you have used Python before or have other computing experience, the lower end of the range may apply to you; if you have not had such experience, the upper end may be more applicable.

In later modules, in addition to Exercises you also will find ***Breakpoint*** questions sprinkled throughout. In debugging environments for programming languages, breakpoints are markers that can be put into code to pause the execution of a program, enabling the programmer to inspect the values of variables and otherwise confirm that the program is doing what it is supposed to be doing at that point. The module *Breakpoint* questions are meant to serve a similar purpose: usually, they get you to pause in your reading, to think about what you've read in order to confirm that your brain is doing what it should be with the material at that point! The answers to the *Breakpoint* questions are provided at the end of each module.

In addition to *Breakpoint* questions, some of the modules contain *Prompts* that ask you to think about the process of learning and your motivations for taking on the challenges of academia. Because learning programming, and algorithmic thinking more generally, is challenging for most people, it helps to be thoughtful about how it's done. This kind of thinking about thinking is called *metacognition*, and studies show that being aware of how learning and thinking happen make those tasks easier and one's efforts more effective. Please include your responses to the *Prompts* in your *e-portfolio*, which is described later in this document.

0.2 Why Learn Computer Programming?

For hundreds of years, progress was made in physics, chemistry, geology, astronomy, etc. without the use of computers. However, as those fields evolved, the problems tackled by scientists grew more complicated and more challenging to solve, due either to the complexity of the systems involved or to the vast amounts of data that needed to be analyzed. The advent of electronic computers made it feasible to attempt to solve some of these problems. Nowadays, the computer is indispensable for applications ranging from genomics (e.g., the Human Genome Project), to computational chemistry (used, for example, to predict the molecular structures of potential new pharmaceutical drugs), to climate modeling (needed to assess the impacts of climate change), to the search for planets beyond our solar system, to modeling the formation of large-scale structure during the evolution of the universe.

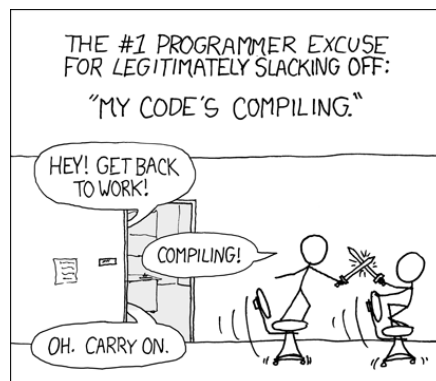
Besides enabling scientists to address the very important applications listed above (and countless others!), learning computer programming is very valuable for another reason: it teaches the programmer to think in an algorithmic way. (Computers can “think” *only* in that way, at least for now, so we – the more mentally-adaptable “species” – are forced to adopt their mode of thought!) By *algorithmic thinking* we simply mean thinking according to an ordered, logical sequence of steps: a recipe is a good everyday example of an algorithm. Algorithmic thinking arguably is the best way to solve most scientific problems (apart, perhaps, from those situations in which intuition might be needed), so becoming more proficient at such thinking can help one become a more effective scientist. Developing your algorithmic thinking skills is, in fact, the primary purpose of these modules. Learning the tools of programming is a secondary (but very important) focus.

0.3 Modern Computer Capabilities

Computer speeds often are characterized in terms of *flops* (or **FLOPS**) – floating-point operations per second – where a floating point operation might be the addition of two floating-point numbers (numbers requiring a decimal representation, in contrast to integers). According to Wikipedia, a typical single-core processor in a laptop or desktop computer can perform roughly 10 billion floating-point operations per second; therefore, its speed would be rated at 10 gigaflops, or 10^{10} flops. As of this writing (Jan. 2015), the fastest computer in the world is located in China, and it has a speed of nearly 34 petaflops, or 3.4×10^{16} flops. It has been projected that an exaflop (10^{18} flops) computer will be built by 2019. Computing power in the 100-10000 teraflop (10^{14} - 10^{16} flops) range makes physically realistic simulations and analyses of structures and processes possible in reasonable time frames.

0.4 Programming Concepts

Computers are instructed to do what we want them to do through the use of **computer code**, written in a **programming language**. The very earliest programmers used **machine code** to program their computers: it can be directly understood and executed by a computer's hardware, but is very difficult to program in. Modern “high level” programming languages readable by humans generally fall into two categories (although the distinctions between them are not precise). **Compiled languages** – e.g., some versions of Basic, C/C++, Fortran, and Java in some forms – are converted by **compilers** (other software) into machine code *before the user's software is run*. These languages tend to produce code that runs very fast – a boon to a programmer doing complicated calculations with large data sets. Their disadvantage is that one must “compile” the code before being able to run it. **Interpreted languages** – including Mathematica, MATLAB, and some other versions of Basic – use interpreters (other software) to translate the code into machine-executable form *when the code is run*: no pre-compilation is required. This approach allows the programmer to see the results of her program sooner, but leads to code that may not run as fast as compiled code. (As already noted, the distinctions between these two types of languages are not clear-cut, and there are other ways of coding that fall in between these two categories.) Python – the language used in these modules – actually employs a hybrid approach, so it is easy to see results quickly but the code also runs quickly.



(Image credit: xkcd.com)

In addition to the compiled/interpreted language distinction, another important distinction associated with modern computing languages is the one between numeric and symbolic computing. As the name suggests, numeric computing involves the processing of actual numbers; e.g., the numerical derivative of $\sin(x)$ would involve computing the differences between its values for nearby points x and $x + \Delta x$, with x having particular values. Some modern languages also can perform symbolic computations; e.g. the symbolic derivative of $\sin(x)$ would be reported as $\cos(x)$, with no numbers involved. Python has a set of functions that one can call on (called "sympy," for *symbolic Python*) that allow for symbolic computation.

Another feature possessed by some programming languages is their ability to be used for ***object-oriented programming***. This programming style, which has become very common in the last few decades, enables the programmer to create reusable code modules that are easy to modify for new applications. Corporate programmers have tended to utilize this approach more than scientists have, but its use in the scientific realm is growing.

It's important to keep in mind that computers are completely literal, no matter the programming language used. They operate based on precisely what you tell them, not on what they think you mean. If your code has an error in it, whether small or large, a computer will not understand and fix it. If you're lucky, it will produce an error message; if you're not, it simply will not do what you want it to, and you may not realize that. Therefore, in cases when incorrect output is not obvious, it's critical to test your code. A good way is to run it on data for which you know what the output should be. Don't assume the output is right just because you get a result!

The interested reader is encouraged to consult the relevant Wikipedia pages for more information on programming languages.

0.5 Starting with Python

For these modules, we will use Python as the main programming language. Its advantages are numerous: it is open-source (free) software, its syntax is easy to learn, it is an interpreted language that allows the user to see the code's output easily, it has extensions that enable one to do sophisticated interactive numerical analysis and visualization (similar to MATLAB and Mathematica), it is a widely-used language (including within the scientific/engineering community), and it is object-oriented. Moreover, a variant known as IPython has been developed which can be run in a browser-based format called a ***notebook***. These notebooks can incorporate not only code, but also text, LaTeX, graphics, videos, and more.

These modules are not meant to teach you the full Python language. Module 1 presents a brief introduction to Python, but it won't be all the information you will need for the modules. Appendix A of this module lists additional resources for learning Python, and computer programming in general. A list of the basic topics you'll need to know about is presented in Appendix B. For instructions on how to download and install tools for running Python on a computer, see Appendix C. The remainder of this section will describe some aspects of programming, and the Python language, that you might not initially pick up in your study but which will be important for these modules.

Currently, there are several versions of Python in common use: version 2.7, and versions 3.3 and 3.4. The 3.X versions differ from 2.7 (and earlier 2.X versions) in several ways, three of which are relevant to us: in the syntax of the `print` statement, which displays text or numeric output onscreen; in their treatment of **integer division** – the division of one integer by another; and in the behavior of the `input` statement, which receives input from a user. (Note: throughout the modules, Python code will be presented in Courier font, as in the word `print` shown above.) Python versions 2.X round the result of an integer division to an integer; for example, they would give the unexpected result $1/2 = 0$. Versions 3.X produce the expected result: $1/2 = 0.5$. In version 2.7 (and earlier), the syntax of the `print` statement was `print "Hello"`, while versions 3.X surround the content to be printed with parentheses: `print("Hello")`. In the 2.X versions, the `input` statement retains the form of the user's input: if it's a number, `input` stores a number; if it's text, `input` stores that text. In the 3.X versions, the `input` statement converts all input to text, for consistency. *For these modules, we will assume use of version 3.X*, but beware that some Python packages may not have every associated extension fully implemented. Here's the prototypical example of a first program – the traditional starting point when learning any programming language:

```
print("Hello World!")
```

Note that, by itself, this is a complete (if uninteresting) Python program! Nothing is required before the `print` statement to make it run. In some languages, every program, even one this simple, requires some initial instructions to the computer. Already, we see here some of the simplicity of Python.

Another way in which Python is easy to learn is that it treats variables very simply. In computer programming languages, **variables** can be one of several **types**, most commonly **integers**, **floating-point numbers**, **booleans**, and **characters/strings**. A string is just a set of characters, e.g. “time” or “x_init”; a floating point number can be thought of simply as a real number; and a boolean is `True` or `False`. In some languages, such as C, each variable must have its type “declared” at the beginning of the program, before it is used (e.g., `int x` declares `x` as an integer); however, the Python language is not “statically typed,” which means that when you need a variable you simply define it, and Python assumes its type or figures out what kind of variable it is from the definition. For example, `x = 1` creates `x` as an integer, while `x = 1.0` or `x = float(1)` creates it as a floating-point number. (This could lead to a problem if the programmer does not keep in mind how a variable has been defined: remember that integer division works differently from normal division in 2.X versions!) Python also supports **complex numbers** as variables; they may appear in a later module.

Besides variables, other key programming structures are **operators** (multiplication, exponentiation, etc.) **loops** (including `for` and `while` loops; these allow a particular calculation to be repeated multiple times in sequence), **functions**, **lists**, and **branching statements** (usually called `if`, or `if-else` statements; some languages call these `switch` or `case` statements; they allow the code to follow different logical paths depending on the state of the code or on user input). Higher level programming languages typically also allow the programmer to include **comments**, which the computer ignores. In Python, these are identified by a hash sign, `#`: everything on a line following a hash is ignored. Multi-line comments are enclosed within triple sets of (single or double) quotes: `""" Here's nice text """`.

One other important thing to know about modern programming languages is that user-written programs often acquire additional capabilities by bringing in functions from pre-existing programs. In Python, one speaks of this process as *importing packages*; in other languages, it's usually referred to as *linking to libraries* (a library being a collection of related functions, e.g., mathematical ones). Such sets of functions in the mathematical realm are designed for very efficient calculation. For example, performing more than simple mathematics in Python requires importing the `math` library.

Important libraries for our purposes are `math` (function list at <https://docs.python.org/2/library/math.html>); the more extensive `numpy` (*numerical python*) library, with lots of additional functions (see <http://docs.scipy.org/doc/numpy/reference/>); the `sympy` (*symbolic python*) library, which has functions for doing symbolic mathematics (e.g., differentiating $\sin(x)$ to get $\cos(x)$; see <http://docs.sympy.org/latest/index.html>); and `matplotlib`, which provides MATLAB-like plotting capability (see <http://matplotlib.org/py-modindex.html>).

With this very basic foundation established, the reader now is directed to read Module 1. It might also be helpful to work through one of the online tutorials listed in Appendix A, or consult a text, to gain some familiarity with Python and basic programming. (The book by Kinder & Nelson listed in the Appendix is a particularly good resource to supplement these modules.) An initial exposure to Python through these tutorials should include the programming topics listed in Appendix B. Appendix C describes how to install Python on your own computer and how to get started using those Python packages and IPython notebooks.

0.6 Tips on Learning Scientific Programming

The modules that follow this one deal with technical materials that may be a bit challenging to learn, depending on your prior experience with programming and algorithmic thinking more broadly. Here are some tips to keep in mind that might be helpful as you read and work through the exercises in the sequence of modules:

- Technical writing cannot be read like a novel. You will need to read the same material more than once for the key ideas to become clear and understandable. In your first reading, you might want to simply skim the text to get a general overview of what it is presenting. On your subsequent readings you can progressively key in on finer details.
- While reading, you might want to highlight or otherwise make note of what seem to be key terms and concepts. If you encounter a term you're not familiar with and its definition is not clear from the text, stop and look it up. While reading, it also could be very helpful to make notes on the passage. Imagine that you are going to give a mini-lecture to a fellow student – what would it be most important to tell them to make the material understandable? If you can identify the key ideas, you probably understand the material well yourself!

- When writing code, and especially when trying to debug it, it can be very helpful to get input from someone else. Just like it's hard to notice poor grammar or typographical errors in your own writing when you've read through it many times, it can be difficult to find bugs in your own code: you know what you *want* the code to do, and your brain sees *that* rather than what's actually there.
- You should feel free to work through the modules with other people, if that would be helpful!
- When writing a big chunk of code, try to break it down into small, self-contained "chunks." Get each of those working and test them with a reasonably broad (but relevant) set of inputs/parameters, *then* combine the chunks appropriately to build the entire package. In this way, you can reduce the amount of code you have to search through to find a bug.
- There usually are many ways of coding a particular functionality. Don't worry at first about finding the "best" method: get something that works, and then you can think about refining it, if you have the time and the need for greater efficiency. That said, your code should be clearly organized, well-commented, and what it's doing should be clear to another programmer. If it's not, then chances are that after not looking at it for a while even *you* won't be able to figure out what it's doing.

0.7 Your E-Portfolio

We hope that you will find many aspects of these modules to be useful to you. One way in which that might happen is that you could use the products of your work on the modules as evidence of your mastery of the programming skills presented in them. You might want to present such products to a graduate program as evidence of your ability to do research involving a computational component, or to a potential employer as evidence that you have the skills matching the requirements of the position they seek to fill. Even if you don't end up using your work for these reasons, "packaging" it in a well-organized form might make it a valuable resource for you in the future if you need to re-learn some of the skills developed by the modules.

For the reasons indicated above, we will ask you to compile an *e-portfolio* ("electronic" portfolio) of the materials that you develop in working through the set of modules. This e-portfolio should include: all of the Python code (in the form of IPython notebooks or .py files) you develop in solving the Exercises presented in the modules, your answers to the *Breakpoint* questions, and your responses to the *Prompts*. (The Breakpoint answers may be included in IPython notebooks, or in a separate PDF or Word file. Your responses to the Prompts should be compiled into a single PDF or Word document.)

Appendix A – Programming & Numerical Methods Resources

A.1 Programming Resources

There are many valuable resources that one can use to supplement these modules. A good general resource on all aspects of computing is stackoverflow.com. If you search online for an answer to a question – which is a good way to get an answer – this site is likely to come up at or near the top of the list. The site <https://www.python.org/> has lots of good material specifically on Python (and check out https://wiki.python.org/moin/BeginnersGuide/NonProgrammers#Graded_Introductory_Programming_Courses). Other useful resources for programming and Python include (as of this writing) those below. *In each category, the top one or two resources might be the best places to start.*

Tutorials:

<https://docs.python.org/2/tutorial/> – non-interactive.

www.codecademy.com – an extensive interactive introduction that will take a while to work through, but which is very suited to programming novices. Note that you choose Python as the language to learn at the bottom of the main page. (You do not have to create an account unless you want the website to keep track of your progress and remember where you leave off at the end of each session.)

http://www.python-course.eu/python3_course.php

www.learnpython.org – a concise but fairly comprehensive introduction to Python programming.

<http://interactivepython.org/runestone/static/thinkcspy/index.html> – an "interactive textbook."

<http://www.afterhoursprogramming.com/tutorial/Python/Introduction/> – an interactive tutorial

<http://software-carpentry.org/v5/novice/python/index.html> – Contrary to the word in the URL, this is not for novices, but might be a good way to test your skills after you have done some Python programming.

<http://quant-econ.net/py/index.html> – Part 1 is a nice summary of Python basics.

<http://www.tutorialspoint.com/python/> – a somewhat-interactive online tutorial that also provides a downloadable PDF tutorial. This site assumes that you have access to Python installed on a computer. It also has a nice online Quick Reference resource.

<http://learnpythonthehardway.org/book/> – non-interactive, but very user-friendly.

<http://www.cs.hmc.edu/csforall/> – non-interactive, but the basis for the introductory computer science course at Harvey Mudd College. (Section 2 in Chapter 1 is not useful for our purposes.)

Courses:

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-189-a-gentle-introduction-to-programming-using-python-january-iap-2011/>

<https://www.coursera.org/course/pythonlearn> – requires (free) registration.

<https://www.udemy.com/introduction-to-computer-science-and-programming-python/?dtcode=cKFpSGu1kzK8>

Books (there are, of course, many more than those listed here):

A Student's Guide to Python for Physical Modeling, Jesse Kinder and Philip Nelson. An excellent brief survey of Python for scientific computing applications.

Think Python, Allen Downey, Jeffrey Elkin, and Chris Meyers, freely downloadable from this site:
<http://www.greenteapress.com/thinkpython/>

Python for Informatics, Charles Severance (freely downloadable from the coursera site above).

A.2 Numerical Methods Resources

Computational Physics, Mark Newman, CreateSpace Independent Publishing Platform, 2013. Uses Python. Easy to read, very clear explanations and derivations, good examples and choices of topic coverage. A great first choice of text.

Numerical Recipes, 3rd Ed., William Press, Saul Teukolsky, William Vetterling, Brian Flannery, Cambridge University Press, 2007. Uses C++. (The 2nd edition uses C, so its code examples may be easier to understand.) This is the classic text in the field, but it is more advanced than the text above. Explanations tend to be concise, but explicit code is given for many methods and the writing is engaging. Any important method not discussed in detail in the text is likely to have a reference cited.

A Survey of Computational Physics, Rubin Landau, Manuel José Páez, and Cristian Bordeianu, Princeton University Press, 2008. Uses Java. Introduces object-oriented programming early. Includes a number of topics and applications not in Newman's book.

Computational Physics, 2nd Ed., Nicholas Giordano and Hisao Nakanishi, Pearson Prentice Hall, 2006. Presents language-independent code in outline form only. Has lots of nice physics-related applications.

A Primer on Scientific Programming with Python, 2nd Ed., Hands Petter Langtangen, Springer, 2011. Clear writing, lots of code samples provided. Focuses more on programming than on scientific applications.

Appendix B – Basic Python topics to know

- Types of variables: integers, floating point numbers ("floats"), strings, booleans
- Basic mathematical operations: multiplication, division, addition, subtraction, power ($x**y$), modulo ($x\%y$ gives the remainder after dividing x by y an integer number of times; e.g., $16\%7 = 2$)
- Lists, arrays, and their differences: `[7, "howdy", 42.00]`, `[6, 12, 18, 24]`
- The simple print statement: `print("text")`

- The formatted print statement:

```
print("You typed the number %f %d times." %(num, times))
```

Here, the `%f` is a placeholder indicating that a floating point number will be printed, and the `%d` indicates an integer to be printed. The second `%` followed by the two variables in parentheses (`num`, `times`) are substituted for the `%f` and `%d`, in order.

- The input statement: `reply = input("Enter the value of x: ")`
- if and if-else statements:

```
if x > 100:
    print("x is pretty big")
```

```
if x > 100:
    print("x is pretty big")
else:
    print("x is not so big")
```

- The for loop:

```
for i in [1, 2, 3]:
    print(i+1)
```

- The while statement:

```
n = 0
while n < 4:
    print(n+1)
    n = n + 1
```

- Constructing simple functions:

```
def func():
    <function statements here...>
```

- Importing functions from modules (`from module import function` or `from module import *`) and importing entire modules (`import module`).

Appendix C – How to install Python on your computer or use it online

To install and run Python on your computer, you can use an open source product such as Anaconda or Canopy, both of which include many useful Python packages. You also can install Python and desired add-on packages directly.

To download and install Continuum's Anaconda, use this link:

<http://continuum.io/downloads>

To download and install Enthought's Canopy, use this link:

<https://store.enthought.com/downloads/>

Be careful to choose the correct package for your computer's platform (Windows, Mac, or Linux), and choose the right installer for your computer's processor (32-bit or 64-bit).

To find out if your computer is running a 32-bit or 64-bit version of Windows, do the following:
For Windows 7:

1. Click the **Start** button, right-click **Computer**, and then click **Properties**.
2. Under **System**, you can view the system type.

For Windows XP:

1. Click the **Start** button, right-click **My Computer**, and then click **Properties**.
2. If you don't see "x64 Edition" listed, then you're running the 32-bit version of Windows XP. If "x64 Edition" is listed under **System**, you're running the 64-bit version of Windows XP.

To check whether your Intel-based Mac has a 32-bit or 64-bit processor:

1. Choose **About This Mac** from the **Apple** menu in the upper-left menu bar, then click **More Info**.
2. Open the **Hardware** section.
3. Locate the **Processor Name** and consult the table below to determine your Mac's processor type.

Processor Name	32- or 64-bit
Intel Core Solo	32 bit
Intel Core Duo	32 bit
Intel Core 2 Duo	64 bit
Intel Quad-Core Xeon	64 bit
Dual-Core Intel Xeon	64 bit
Quad-Core Intel Xeon	64 bit
Core i3	64 bit
Core i5	64 bit
Core i7	64 bit

To check if your Linux installation is 32-bit or 64-bit:

Run the 'uname -m' command. If the output appears as i686, it means that it's 32-bit. If it's 64 bit, the output would be x86_64.

After you have downloaded the package of your choice, follow the instructions on the corresponding website to install it on your computer.

As an alternative to installing one of the packages described above, you can download Python itself from the following websites:

<https://www.python.org/download>
<https://www.python.org/download/windows/>
<https://www.python.org/download/mac>

If you install the Python package instead of Anaconda or Canopy, you can edit and run Python scripts in IDLE, the Integrated DeveLopment Environment for Python and the default environment for most Python installations.

More information on using Anaconda

If you chose to use Anaconda, then after downloading and installing the package open the Launcher. In its window, you can choose to install the Spyder app – the Scientific PYthon Development EnviRonment, an interactive development environment (IDE) included with Anaconda – and/or the IPython notebook environment, which runs in your own browser.

To open Spyder, click on the Spyder icon or simply type 'spyder' in the command prompt or terminal. In Spyder, you can open, view and edit existing files, and create new files in the Editor (the left-hand panel) using the icons in the toolbar at the top of the window, or via the respective options in the File menu. You can execute the files by clicking the green triangular button in the toolbar or choosing the 'run' option in the Run menu. You can run a portion of a file by highlighting that portion and selecting 'Run selection' under the Run menu. If you attempt to execute code that contains errors, a message will appear in the Console window at lower-right. Small orange triangles also will appear in the Editor window in front of the lines containing the errors; you can see descriptions of the errors if you hover your cursor over the triangles.

You can also use the in-browser IPython notebook by typing 'ipython notebook' in the Anaconda command prompt or the terminal window, or by launching the notebook environment from the Launcher. This will open a new tab in your browser; the page will present a list of the directories from which you can choose to run a pre-existing notebook, or you can start a new one using the button in the upper-right. More information on using notebooks is provided below.

More information on using Canopy

If you chose to use Canopy, after downloading and installing it click on the Enthought Canopy icon and then click on **Editor** in the next window. In the Canopy editor, you can choose to write programs in a Python file or in the interactive IPython notebook. To edit a new Python file, click the 'Create a new file' button and, by default, a new Python file will appear. You can also do this by choosing **File** under the main menu and then **New** under the File menu. From here you can select either IPython notebook, a Python file, or various other options.

In a Python file in Canopy, you can write your program and run the file at any time by clicking on the green triangle in the Run bar below the main menu. The Python conversation window will show the results, or an error message, for every execution.

You can also select **Help** under the main menu imbedded in the IPython notebook to: view more information on Python, learn of several libraries supported by the IPython notebook, see more detailed commands, view example files and gallery, etc.

IPython

An IPython notebook provides a browser-based interactive computing environment. Besides being able to run executable code, the notebook also supports commentary text and other rich media such as LaTeX, HTML, and images.

The Anaconda and Canopy distributions already include IPython, so you can simply use the IPython notebook in your distribution, although you might want to check <http://ipython.org/install.html> to make sure you have the latest version of IPython. If you have just basic Python installed, then go to the website below and follow the instructions to download and install the IPython notebook environment:

<http://ipython.org/ipython-doc/stable/install/install.html>

After installation, you can open and use an IPython notebook by typing 'ipython notebook' in a terminal or command prompt window on your computer (or by using the appropriate launcher from Anaconda or Canopy).

You also can run IPython on a remote server if you have been given an account on one. You connect to it via a browser, just as to any other web page, and all input and output is done through the browser.

Notebooks have an input/output format like that of Mathematica, with each input in a "cell." Each cell is like a single file – it can include one or more lines of code, commentary text, mathematical equations in LaTeX format, etc. Cells can be modified both before and after execution.

The cell format provides the advantage of breaking down code into smaller sections for easier reading and debugging. Most if not all implementations will allow you to run either one cell at a time or all cells at once; you may even be able to run just a part of a cell. See the instructions or help system for your particular implementation.

To write commentary text in a notebook, select the **Markdown** option from the **Cell** menu. Combine the Shift and Enter keys to complete Markdown input. Double-click a Markdown cell to enable modification or rewriting. You can also conveniently transform any cell between Markdown or Code formats by highlighting the cell and using the scrollable menu at the right end of the toolbar.

For more information on the topics discussed in this Appendix, see:

<http://ipython.org/install.html>

<http://windows.microsoft.com/en-us/windows/32-bit-and-64-bit-windows#1TC=windows-7>

<http://support.apple.com/kb/HT3696>

<http://mylinuxbook.com/5-ways-to-check-if-linux-is-32-bit-or-64-bit/>

<http://docs.continuum.io/anaconda/index.html#id2>

<http://code.google.com/p/spyderlib/wiki/Features>

<https://www.enthought.com/products/canopy/>

<http://ipython.org/ipython-doc/dev/notebook/>