



MTH 331

Introduction to
Scientific and Mathematical Computing

Fall 2015

Instructor: Adam Cunningham

University at Buffalo
Department of Mathematics

Contents

1	Getting Started	4
1.1	Course Description	4
1.2	Install Python	4
1.3	Install LibreOffice	5
1.4	Weekly Reports	5
1.4.1	Graphics	6
1.4.2	Fonts	6
1.4.3	Using Styles	6
1.5	IPython Notebook	6
1.5.1	Magics	8
1.5.2	Markdown	9
1.5.3	LaTeX	9
2	Programming Python	10
2.1	Numbers	10
2.2	Booleans	11
2.3	Strings	11
2.4	Type Conversions	13
2.5	Variable Names	14
2.6	Modules	15
2.7	Lists	17
2.8	Tuples	19
2.9	Sets	20
2.10	Dictionaries	21
2.11	Boolean Expressions	22
2.12	If Statements	23
2.13	For Loops	25
2.14	While Loops	27
2.15	Break and Continue	27
2.16	Comprehensions	28
2.17	Functions	28
2.18	Reading and Writing Files	31
2.19	Comments	32

3	NumPy	33
3.1	Array Creation	34
3.2	Array Properties	36
3.3	Array Operations	37
3.4	Accessing Arrays	39
4	Matplotlib	41
4.1	Basic Plotting	42
4.2	A More Complex Plotting Example	43
4.3	Bar Plots	44
4.4	Histograms	45
4.5	Contour Plots	46
4.6	Multiple Plots	47
4.7	Formatting Mathematical Expressions	48
5	Additional Topics	49
5.1	Loading Numerical Files	49
5.2	Animation	50
5.3	Images	52
5.4	Random Number Generation	53
5.5	Sound Files	54
5.6	Linear Programming	55
6	Programming Style	56
6.1	Choosing Good Variable Names	56
6.2	Choosing Good Function Names	57
6.3	No “Magic Numbers”	57
6.4	Comments	57
6.5	Errors and Debugging	58
7	Further Reading	59

I Getting Started

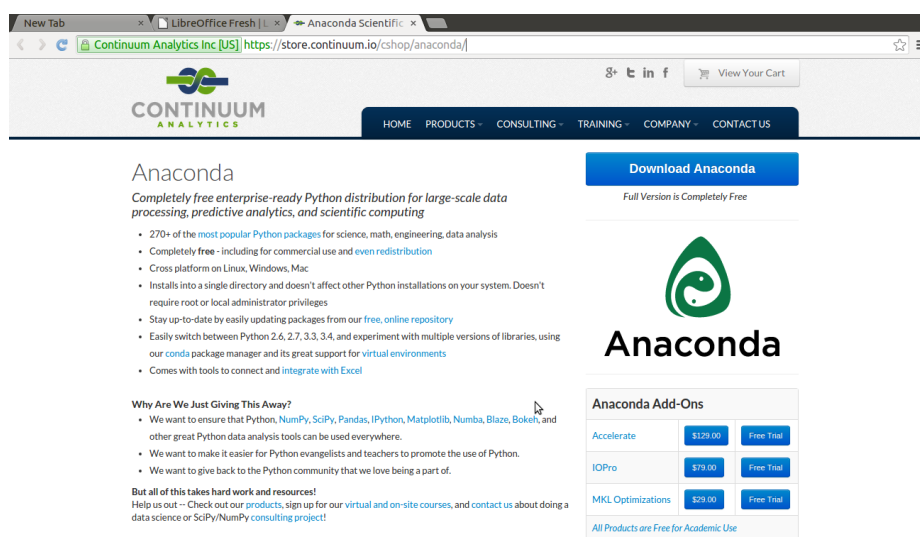
II Course Description

This course covers the following areas:

- Programming using Python, the scientific computing package NumPy, and the plotting library Matplotlib.
- Scientific computing methods used in number theory, random number generation, initial value problems, dynamical systems, root-finding, linear regression and optimization.
- Using computers to explore topics in the mathematical and natural sciences.
- Presentation of experiments, observations and conclusions in the form of written reports.

I.2 Install Python

We will be using Python 2.7. It is recommended that you use the [Anaconda](#) distribution, which is available free on Windows, Mac and Linux and contains all the packages we need (NumPy, SciPy, IPython, Matplotlib).



The screenshot shows the Anaconda website with the following content:

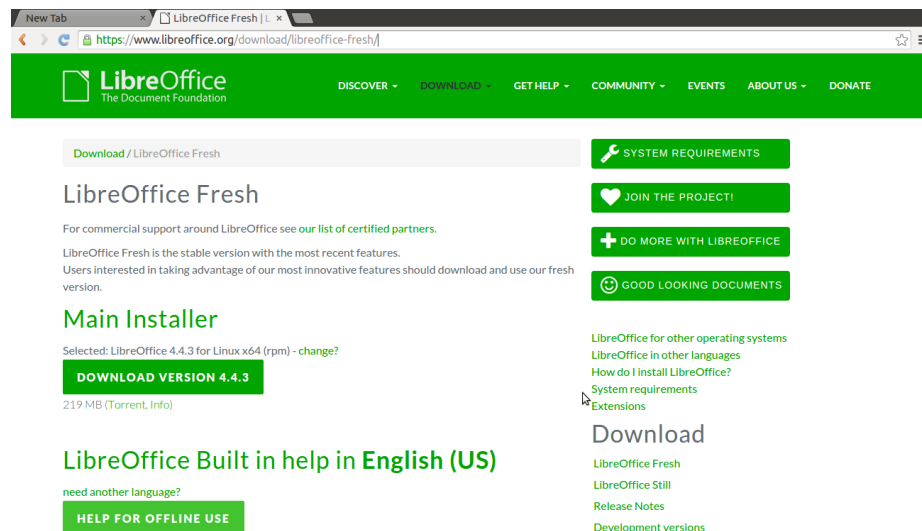
- Navigation Bar:** CONTINUUM ANALYTICS, HOME, PRODUCTS, CONSULTING, TRAINING, COMPANY, CONTACT US.
- Header:** Download Anaconda (Full Version is Completely Free), View Your Cart.
- Main Content:**
 - Anaconda:** Completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing.
 - Features:**
 - 270+ of the most popular Python packages for science, math, engineering, data analysis
 - Completely free - including for commercial use and even redistribution
 - Cross platform on Linux, Windows, Mac
 - Installs into a single directory and doesn't affect other Python installations on your system. Doesn't require root or local administrator privileges
 - Stay up-to-date by easily updating packages from our free, online repository
 - Easily switch between Python 2.6, 2.7, 3.3, 3.4, and experiment with multiple versions of libraries, using our conda package manager and its great support for virtual environments
 - Comes with tools to connect and integrate with Excel
 - Why Are We Just Giving This Away?**
 - We want to ensure that Python, NumPy, SciPy, Pandas, IPython, Matplotlib, Numba, Blaze, Bokeh, and other great Python data analysis tools can be used everywhere.
 - We want to make it easier for Python evangelists and teachers to promote the use of Python.
 - We want to give back to the Python community that we love being a part of.
 - But all of this takes hard work and resources!**
Help us out - Check out our products, sign up for our virtual and on-site courses, and contact us about doing a data science or SciPy/NumPy consulting project!
- Anaconda Add-Ons:**

Add-On	Price	Free Trial
Accelerate	\$129.00	Free Trial
IOPro	\$79.00	Free Trial
MKL Optimizations	\$79.00	Free Trial

All Products are Free for Academic Use

1.3 Install LibreOffice

Weekly reports must be written using LibreOffice Writer. [LibreOffice](https://www.libreoffice.org/) is a free, cross-platform suite of applications including the word processor Writer, which is similar to Microsoft Word.



1.4 Weekly Reports

Reports will be submitted every week on UBLearns as a file “{lastname}{number}.odt” (e.g. “cunningham01.odt”). The final report will be a pdf compilation of all the weekly reports, including a title page and table of contents.

Reports usually need to include:

- An introduction. The topic should explained in a way that would be comprehensible to another member of the class.
- A clear statement of the specific question or task.
- A description of the approach used to tackle the question or task.
- Clearly presented results, including appropriate diagrams and plots.
- An interpretation of the results.
- An appropriate conclusion.
- If appropriate, a list of references to books, articles and websites consulted.
- A complete listing of the Python code used to generate all figures and data used in the report. This should be presented in a way that can be directly copied, pasted, and run by the instructor.

Extra credit will be assigned for extra or unusual work or insight.

1.4.1 Graphics

Reports are graphics-intensive, and will contain a lot of embedded images. Plots made with Matplotlib can be saved as .png (Portable Network Graphics) files, then embedded in a LibreOffice document using **Insert** → **Image** → **From File**. Make sure that **Insert as Link** is *not* checked (this should be the default anyway). Reports are submitted on UBlerns as a single document, so they should contain no dependencies on other files.

Graphs should be labelled in a way that makes the content of the graph clear. They should include the following information (at a minimum):

- Labels for the x- and y-axes, specifying the units when displaying quantities.
- A legend below the graph. Although graphs can be given a title when the graph is created, this information then becomes part of the graph and can't be changed later. Adding this information later using LibreOffice means that the font and style will be consistent across the report, and the description can be easily changed.
- A number for the legend, which can be used to refer to the graph in the report.

1.4.2 Fonts

Feel free to change the default fonts to suit your own tastes. However:

- Try not to mix up *too* many different fonts in one document. A good rule of thumb is to limit the number of different fonts to two or three.
- Titles, subtitles etc can be a different font to the body text.
- Use a fixed-width font such as Courier for computer code. Indenting matters in Python, and is an intrinsic part of the language. Using a fixed-width font ensures that code blocks that are supposed to line up actually do line up on the screen.

1.4.3 Using Styles

Any changes to fonts, font sizes etc should be done by modifying existing styles or creating new ones. This will keep a consistent look across the whole report, save the work of repeated font changes, and allow changes to a style to be made in just one place.

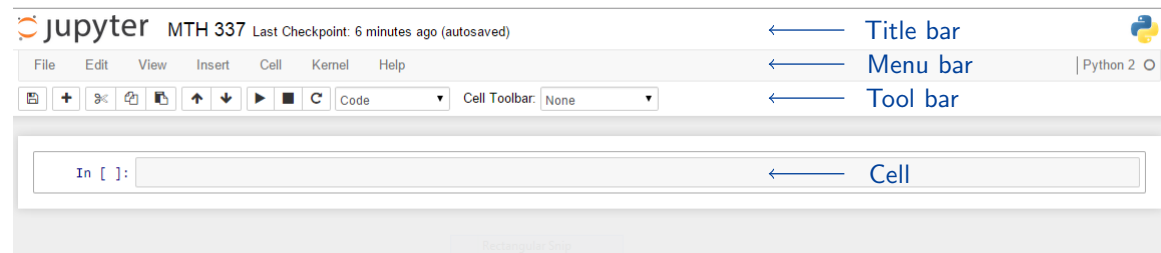
- Modify the built-in styles using **Format** → **Styles and Formatting**, then right click **Modify** on the style.
- Create a new style (e.g. for Python code) using **Format** → **Styles and Formatting**, then right click **New** on the style you want to inherit from.

1.5 IPython Notebook

The development environment we will be using is [IPython notebook](#). This provides:

- An interactive environment for writing and running code.
- A way to integrate code, text and graphics in a single document.

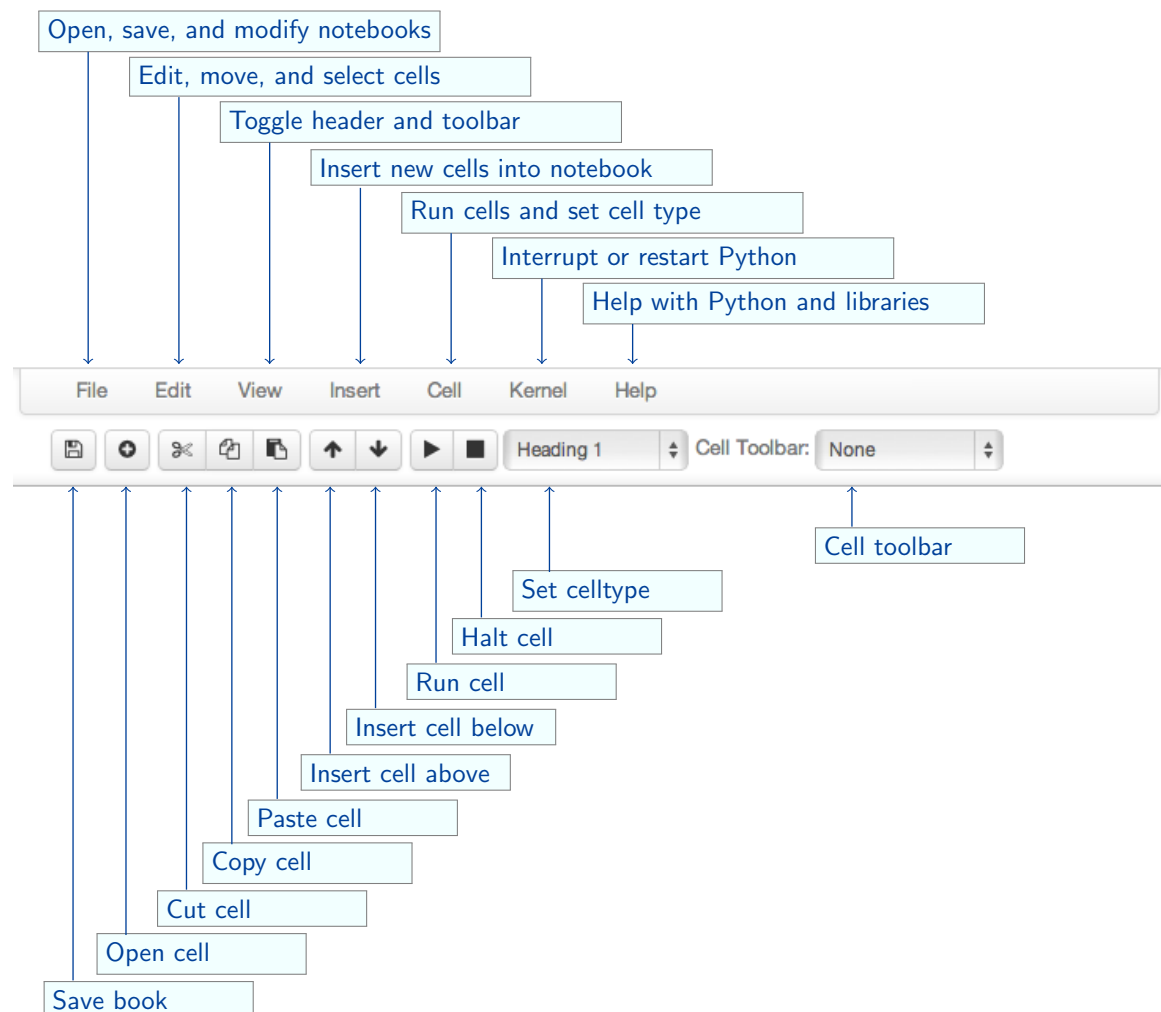
A new IPython notebook opens in a web browser, and looks as follows:




The notebook consists of:

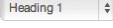
- A **Title** bar, containing the name of the notebook.
- A **Menu** bar, containing all the functions available in the notebook.
- A **Tool** bar, for easy access to the most commonly-used functions.
- A list of cells, containing code or text, and the results of executing the code.

The menu bar and tool bar contain the following functions.



Code and text are entered in *cells*, which come in three important types:

- **Code** cells, which contain Python code.
 - Click on a cell with the mouse to start entering code.
 - Enter adds a new line in the cell, without executing the code.
 - Shift-Enter (or clicking the “Play”  button in the toolbar, or **Cell** → **Run** in the menubar) executes the code in the cell and moves the cursor to the next cell.
 - Tab brings up help for the function the cursor is currently in.
- **Markdown** cells contain text formatted using the Markdown language, and mathematical formulas defined using LaTeX math syntax.
- **Heading** cells contain headings to organize the notebook.

The type can be selected by either using the “Cell Type”  pull-down menu in the toolbar, or **Cell** → **Cell Type** in the menubar.

1.5.1 Magics

Magics are instructions to IPython that perform specialized tasks. They are entered and executed in code cells, and prefaced by “%” for a line magic (which just applies to one line) or “%%” for a cell magic (which applies to the whole cell). The main ones we will be using are:

- `%pylab inline` imports numpy and matplotlib (making the functions and variables in these modules available to us), with plots drawn inline (in the notebook itself).
- `%pylab` imports numpy and matplotlib, with plots drawn in separate windows so they can be resized and saved.
- `%run <file>` executes the Python commands in `<file>`.
- `%timeit <code>` records the time it takes to run a line of Python code.
- `%%timeit` records the time it takes to run all the Python code in a cell.

An example of timing code execution using `%%timeit` is as follows,

	Python Code	Explanation
In [1]:	<pre>%%timeit x = range(10000) max(x)</pre> <p>1000 loops, best of 3: 382 μs per loop</p>	The first line “x = range(10000)” is run once but not timed. The “max(x)” code is timed

1.5.2 Markdown

Text can be added to IPython notebooks using Markdown cells. Markdown is a language that can be used to specify formatted text such as italic and bold text, lists, hyperlinks, tables and images. Some examples are shown below.

Markdown	How it prints
An h1 header =====	An h1 header
An h2 header -----	An h2 header
<i>*italic*</i>	<i>italic</i>
bold	bold
This is a bullet list * First item * Second item	This is a bullet list <ul style="list-style-type: none"> • First item • Second item
This is an enumerated list 1. First item 2. Second item	This is an enumerated list <ol style="list-style-type: none"> 1. First item 2. Second item
A horizontal line ***	A horizontal line

1.5.3 LaTeX

Mathematical expressions in Markdown cells are specified using the typesetting language LaTeX. These expressions are identified using: `$\langle\text{formula}\rangle$` for an inline formula (one displayed within a line of text), or `codebox$$\langle\text{formula}\rangle$$` for a larger formula displayed on a separate line. Some of the most useful options are shown below.

<code>\langle\text{formula}\rangle</code>	How it prints
<code>x^2</code>	x^2
<code>x_1</code>	x_1
<code>\frac{1}{2}</code>	$\frac{1}{2}$
<code>\alpha, \beta, \omega</code>	α, β, ω
<code>\sum_{i = 1}^n</code>	$\sum_{i=1}^n$
<code>\int_a^b</code>	\int_a^b
<code>\sqrt{a + b}</code>	$\sqrt{a + b}$
<code>\bar{x}</code>	\bar{x}
<code>(x + y)^2</code>	$(x + y)^2$
<code>\{1, 2, \ldots, n\}</code>	$\{1, 2, \dots, n\}$

2 Programming Python

Python is a flexible and powerful high-level language that is well suited to scientific and mathematical computing. It has been designed with a clear and expressive syntax with a focus on ensuring that code is readable.

2.1 Numbers

The basic numerical types used in Python are:

- Integers.
- Floats (reals).
- Complex numbers (pairs of floats).

Python will automatically convert numbers from one type to another when appropriate. For example, adding two integers yields an integer, but adding an integer and a float yields a float. The main arithmetic operations are +, -, *, /, **, and %, illustrated below.

	Python Code	Explanation
In [1]:	<code>3 + 2</code> 5	Addition
In [2]:	<code>3 - 2</code> 1	Subtraction
In [3]:	<code>3 * 2</code> 6	Multiplication
In [4]:	<code>3 / 2</code> 1	<i>Integer</i> division (truncates - the remainder is discarded)
In [5]:	<code>3. / 2</code> 1.5	Float division (a decimal point turns an integer into a float)
In [6]:	<code>3**2</code> 9	Exponentiation (<i>not</i> 3^2)
In [7]:	<code>26 % 5</code> 1	Remainder (modulo division)
In [8]:	<code>abs(-88)</code> 88	abs returns the absolute value

In [9]:	<code>1 + 2j</code> <code>(1+2j)</code>	Generate a complex number (j is used instead of i)
In [10]:	<code>complex(1, 2)</code> <code>(1+2j)</code>	Another way to generate a complex number
In [11]:	<code>(1+2j).real</code> 1.0	real returns the real part of a complex number
In [12]:	<code>(1+2j).imag</code> 1.0	imag returns the imaginary part of a complex number
In [13]:	<code>abs(3+4j)</code> 5.0	abs returns the modulus when applied to a complex number

Operations are evaluated in standard order - Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. To avoid possible ambiguity, use parentheses to make the order of evaluation clear.

2.2 Booleans

Python also has a Boolean type, which only takes the values True or False. These also work like numbers, where True has the value 1 and False the value 0.

	Python Code	Explanation
In [1]:	<code>True or False</code> True	Logical disjunction
In [2]:	<code>True and False</code> False	Logical conjunction
In [3]:	<code>not True</code> False	Logical negation
In [4]:	<code>True + 41</code> 42	True has the numerical value 1.
In [5]:	<code>False * 41</code> 0	False has the numerical value 0.

2.3 Strings

Strings are sequences of characters. They are identified by surrounding quote marks.

- To generate a string, enclose a sequence of characters in either single (') or double (") quotes (Python doesn't care which).
- A single character in Python is just a one-element string.

- Python strings are *immutable* - once defined, they can't be changed. They can of course still be copied or operated on to create new strings.

	Python Code	Explanation
In [1]:	<pre>print "abc" abc</pre>	print outputs text to the screen (discarding the quotes)
In [2]:	<pre>"abc" + "def" "abcdef"</pre>	Adding two strings makes a new string by concatenation
In [3]:	<pre>"abc"*3 "abccabccabc"</pre>	Multiplying a string by an integer repeats the string
In [4]:	<pre>print "I love 'MTH 337'!" I love 'MTH 337'!</pre>	Embedding quote marks within a string

A `"\"` within a string is used to specify special characters such as newlines and tabs.

	Python Code	Explanation
In [1]:	<pre>string1 = "abc \n def" print string1 abc def</pre>	The <code>"\n"</code> character specifies a newline
In [2]:	<pre>string2 = "abc \t def" print string2 abc def</pre>	The <code>"\t"</code> character specifies a tab

Strings elements are accessed using square brackets, `[]`.

- *Indexing* obtains characters from the string using a single integer to identify the position of the character.
- *Slicing* obtains a substring using `start:stop:step` to identify which characters to select.
- Indexing and slicing is *zero-based* - the first character is at position 0.
- Indexing and slicing is "up to but not including" the stop position.
- A `":"` can be used to select *all* characters either before or after a given position.

	Python Code	Explanation
In [1]:	<pre>"abcde"[1] "b"</pre>	<i>Indexing</i> returns the character at index 1 (indices start at 0, not 1)

In [2]:	<code>"abcde"[-1]</code> <code>"e"</code>	Negative indices count backwards from the <i>end</i> of the string
In [3]:	<code>"abcde"[1:4]</code> <code>"bcd"</code>	<i>Slicing</i> a string from position 1 up to (but not including) position 4
In [4]:	<code>"abcde"[2:]</code> <code>"cde"</code>	Select all characters from position 2 to the end of the string
In [5]:	<code>"abcde"[:2]</code> <code>"ab"</code>	Select all characters from the start of the string up to (but not including) position 2
In [6]:	<code>"abcde"[: :2]</code> <code>"ace"</code>	Select every second character from the whole string
In [7]:	<code>"abcdefg"[1:5:2]</code> <code>"bd"</code>	Select every second character from positions 1 up to 5
In [8]:	<code>"abcde"[: :-1]</code> <code>"edcba"</code>	Reversing a string by reading it backwards

Strings can be formatted using the **format** function. This allows “replacement fields” surrounded by curly brackets `{}` in a string to be replaced by some other data.

	Python Code	Explanation
In [1]:	<code>"{} {}".format("a", "b")</code> <code>"a b"</code>	“Replacement fields” <code>{}</code> are filled in order by format
In [2]:	<code>"1st: {0}, 2nd: {1}".format(3,4)</code> <code>"1st: 3, 2nd: 4"</code>	The arguments to format can also be identified by position, starting at 0

2.4 Type Conversions

Objects can be explicitly converted from one type to another, as long as the conversion makes sense. This is called *type casting*.

- Ints can be cast to floats, and both ints and floats can be cast to complex numbers.
- Complex numbers can't be converted to ints or floats.
- Strings can be cast to numerical types, but only if the string represents a valid number.

Casting is done using the functions **bool**, **int**, **float**, **complex**, and **str**.

	Python Code	Explanation
In [1]:	<code>bool(1)</code> True	Convert integer to boolean
In [2]:	<code>bool(42)</code> True	Any nonzero value counts as True
In [3]:	<code>bool(0)</code> False	Zero equates to False
In [4]:	<code>bool("")</code> False	An empty string is also False
In [5]:	<code>int(2.99)</code> 2	Convert float to integer (the decimal part is discarded)
In [6]:	<code>int("22")</code> 22	Convert string to int
In [7]:	<code>float("4.567")</code> 4.567	Convert string to float
In [8]:	<code>complex("1+2j")</code> (1+2j)	Convert string to complex
In [9]:	<code>float(10)</code> 10.0	Convert integer to float
In [10]:	<code>complex(10)</code> (10+0j)	Convert integer to complex number
In [11]:	<code>str(True)</code> "True"	Convert boolean to string
In [12]:	<code>str(1)</code> "1"	Convert integer 1 to string "1"
In [13]:	<code>str(1.234)</code> "1.234"	Convert float to string

2.5 Variable Names

Variable names can be used to refer to objects in Python. They:

- Must start with either a letter or an underscore.
- Are case sensitive. So `value`, `VALUE`, and `Value` all refer to different variables.
- Are assigned a value using `"="`. The variable name goes to the left of the `"="`, and the value to assign on the right.

	Python Code	Explanation
In [1]:	<pre>x = 5 print x 5</pre>	Assign x the value 5 (note that “=” is used for assignment, <i>not</i> “==”)
In [2]:	<pre>y = x + 3 print y 8</pre>	Assign y the value of x + 3
In [3]:	<pre>course = "MTH 337" print course MTH 337</pre>	course is a string (printed without quotes)
In [4]:	<pre>a, b = 2, 3 print a, b 2 3</pre>	Multiple variables can be assigned at the same time
In [5]:	<pre>a, b = b, a print a, b 3 2</pre>	Values of a and b are swapped (the right hand side is evaluated before the assignment)
In [6]:	<pre>z = 3 z += 2 print z 5</pre>	Same as z = z + 2
In [7]:	<pre>z -= 1 print z 4</pre>	Same as z = z - 1
In [8]:	<pre>z *= 3 print z 12</pre>	Same as z = z * 3
In [9]:	<pre>z /= 2 print z 6</pre>	Same as z = z / 2
In [10]:	<pre>z %= 5 print z 1</pre>	Same as z = z % 5

2.6 Modules

A module is a file containing Python definitions and statements. These allow us to use code created by other developers, and greatly extend what we can do with Python. Since many different modules are available, it is possible that the same names are used by different developers. We therefore need a way to identify *which* module a particular

variable or function came from.

The **import** statement is used to make the variables and functions in a module available for use. We can either:

- Import everything from a module for immediate use.
- Import only certain named variables and functions from a module.
- Import everything from a module, but require that variable and function names be prefaced by either the module name or some alias.

	Python Code	Explanation
In [1]:	<pre>from math import pi print pi 3.14159265359</pre>	pi is now a variable name that we can use, but not the rest of the math module
In [2]:	<pre>from math import * print e 2.71828182846</pre>	Everything in the math module is now available
In [3]:	<pre>import numpy print numpy.arcsin(1) 1.57079632679</pre>	Everything in numpy can be used, prefaced by "numpy"
In [4]:	<pre>import numpy as np print np.cos(0) 1.0</pre>	Everything in numpy can be used, prefaced by the alias "np"

If we want to know what a module contains, we can use the **dir** function. This returns a list of all the variable and function names in the module.

	Python Code	Explanation
In [1]:	<pre>import math print dir(math) ['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']</pre>	The math module contains all the standard mathematical functions

2.7 Lists

List are a type of *container* - they contain a number of other objects. A list is an ordered sequence of objects, identified by surrounding square brackets, `[]`.

- To generate a list, enclose a sequence of objects (separated by commas) in square brackets.
- List elements can be of any type, and can be of different types within the same list.
- Lists are *mutable* - once created, elements can be added, replaced or deleted.

	Python Code	Explanation
In [1]:	<pre>mylist = [1, "a", 6.58] print mylist</pre> <code>[1, "a", 6.58]</code>	Use square brackets to create a list
In [2]:	<pre>len(mylist)</pre> <code>3</code>	len returns the number of elements in a list
In [3]:	<pre>list1 = [1, 2, 3] list2 = [4, 5, 6] list1 + list2</pre> <code>[1, 2, 3, 4, 5, 6]</code>	Adding two lists makes a new list by concatenation
In [4]:	<pre>list1 * 3</pre> <code>[1, 2, 3, 1, 2, 3, 1, 2, 3]</code>	Multiplying a list by an integer repeats the list
In [5]:	<pre>list3 = [] print list3</pre> <code>[]</code>	<code>list3</code> is an empty list
In [6]:	<pre>list4 = list() print list4</pre> <code>[]</code>	Another way to create an empty list

Lists can be indexed and sliced in the same way as strings, using square brackets.

	Python Code	Explanation
In [1]:	<pre>primes = [2, 3, 5, 7, 11, 13, 17] primes[1]</pre> <code>3</code>	Access the element at index 1 (indexing starts with 0)
In [2]:	<pre>primes[3:]</pre> <code>[7, 11, 13, 17]</code>	List slicing, start at position 3, through to the end
In [3]:	<pre>primes[:3]</pre> <code>[2, 3, 5]</code>	List slicing, start at the beginning, end at position 2

In [4]:	<code>primes[2:5]</code> <code>[5, 7, 11]</code>	List slicing, start at position 2, end at position 4
In [5]:	<code>primes[::-1]</code> <code>[17, 13, 11, 7, 5, 3, 2]</code>	One way to reverse a list

The **range** function generates a list of integers in a given range. The start, stop and step parameters to **range** are similar to those used to slice lists and strings.

	Python Code	Explanation
In [1]:	<code>numbers = range(5)</code> <code>print numbers</code> <code>[0, 1, 2, 3, 4]</code>	range (<i>n</i>) creates a list of <i>n</i> consecutive integers, starting at 0
In [2]:	<code>teens = range(13, 20)</code> <code>print teens</code> <code>[13, 14, 15, 16, 17, 18, 19]</code>	range (<i>start</i> , <i>stop</i>) creates a list of consecutive integers, from <i>start</i> to <i>stop</i> - 1
In [3]:	<code>evens = range(0, 9, 2)</code> <code>print evens</code> <code>[0, 2, 4, 6, 8]</code>	The third <i>step</i> argument to range specifies the increment from one integer to the next
In [4]:	<code>range(10, 0, -1)</code> <code>[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]</code>	range can also count backwards using a negative step size

List elements can be changed, added, and deleted, modifying an existing list.

	Python Code	Explanation
In [1]:	<code>mylist = ["a", "b"]</code> <code>mylist.append("c")</code> <code>print mylist</code> <code>["a", "b", "c"]</code>	append adds an element to the end of a list
In [2]:	<code>del(mylist[1])</code> <code>print mylist</code> <code>["a", "c"]</code>	del deletes an element from a list
In [3]:	<code>mylist.insert(1, "d")</code> <code>print mylist</code> <code>["a", "d", "c"]</code>	insert inserts an element at a given position
In [4]:	<code>mylist[1] = "e"</code> <code>print mylist</code> <code>["a", "e", "c"]</code>	List elements can be changed by assigning a new element at a given index

Lists can be sorted and reversed.

	Python Code	Explanation
In [1]:	<pre>letters = ["a", "b", "c"] letters.reverse() print mylist ["c", "b", "a"]</pre>	reverse changing an existing list, reversing the order of elements
In [2]:	<pre>numbers = [2, 10, 3, 26, 5] print sorted(numbers) [2, 3, 5, 10, 26]</pre>	sorted returns a sorted list, but does not modify the existing list
In [3]:	<pre>numbers.sort() print numbers [2, 3, 5, 10, 26]</pre>	sort sorts a list in place, modifying the existing list
In [4]:	<pre>sorted(numbers, reverse=True) [26, 10, 5, 3, 2]</pre>	The <i>reverse</i> keyword is used to sort in descending order

The **min** and **max** functions find the smallest and largest items in a list.

	Python Code	Explanation
In [1]:	<pre>numbers = [2, 10, 3, 26, 5] print min(numbers), max(numbers) 2 26</pre>	min and max find the smallest and largest items

2.8 Tuples

Tuples are containers like lists, with the difference being that they are *immutable* - once defined, elements cannot be changed or added. Tuples are identified by surrounding standard parentheses, `()`.

- To generate a tuple, enclose a sequence of objects (separated by commas) in standard parentheses.
- Tuple indexing and slicing works in the same way as for lists and strings.
- It is an error to try to change a tuple element once the tuple has been created.

Tuples are simpler and more efficient than lists in terms of memory use and performance, and are often preferred for “temporary” variables that will not need to be modified.

	Python Code	Explanation
In [1]:	<pre>tuple1 = ("a", "b", "c") print tuple1 ("a", "b", "c")</pre>	Create a tuple using standard parentheses

In [2]:	<code>tuple1[2]</code> <code>"c"</code>	Elements can be indexed just like lists or strings
In [3]:	<code>tuple1[1:]</code> <code>("b", "c")</code>	Slicing works the same way as for lists or strings

Any comma-separated sequence of values defines a tuple, which can be used to assign values to multiple variables at a time.

	Python Code	Explanation
In [1]:	<code>tuple2 = 1, 2, 3</code> <code>print tuple2</code> <code>(1, 2, 3)</code>	A comma-separated sequence of values defines a tuple
In [2]:	<code>(x, y) = (10, 20)</code> <code>print "x =", x</code> <code>print "y =", y</code> <code>x = 10</code> <code>y = 20</code>	The variables on the left-hand side are assigned to the values on the right
In [3]:	<code>a, b = (2, 4)</code> <code>print a, b</code> <code>2 4</code>	The parentheses are not strictly necessary, and can be discarded

2.9 Sets

Sets are containers with the same meaning they do in mathematics - unordered collections of items with no duplicates. Sets are identified by surrounding curly brackets, {}.

- To generate a set, enclose a sequence of objects (separated by commas) in curly brackets.
- Duplicates will be removed when creating a set or operating on existing sets.
- Sets can be used instead of lists when we know that each element is unique and immutable (unchanging).

	Python Code	Explanation
In [1]:	<code>myset = {1, 2, 3}</code> <code>print myset</code> <code>set([1, 2, 3])</code>	Sets are created using curly brackets
In [2]:	<code>myset = set([1, 2, 3, 2])</code> <code>print myset</code> <code>set([1, 2, 3])</code>	Creating a set from a list (note that duplicates are removed)

In [3]: `print set()` `set([])` creates an empty set
`set([])`

The standard mathematical operations for sets are all built into Python.

	Python Code	Explanation
In [1]:	<code>set1 = {1, 2, 3}</code> <code>set2 = {3, 4, 5}</code>	Create 2 sets
In [2]:	<code>1 in set1</code> <code>True</code>	in tests for set membership
In [3]:	<code>set1 set2</code> <code>{1, 2, 3, 4, 5}</code>	Set union (the union operator can also be used)
In [4]:	<code>set1 & set2</code> <code>{3}</code>	Set intersection (can also use the intersection operator)
In [5]:	<code>set1 - set2</code> <code>{1, 2}</code>	Set difference (can also use the difference operator)
In [6]:	<code>set1 ^ set2</code> <code>{1, 2, 4, 5}</code>	Symmetric difference (can also use the symmetric_difference operator)
In [7]:	<code>set1 <= set2</code> <code>False</code>	Test if one set is a subset of another (can also use the is-subset operator)

2.10 Dictionaries

Dictionaries are containers where items are accessed by a key. This makes them different from *sequence* type objects such as strings, lists, and tuples, where items are accessed by position.

- To generate a dictionary, enclose a sequence of `key:value` pairs (separated by commas) in curly brackets.
- The key can be any immutable object - a number, string, or tuple.
- New dictionary elements can be added, and existing ones can be changed, by using an assignment statement.
- It is an error to attempt to access a dictionary using a key that does not exist. This can be avoided by using the **get** method, which returns a default value if the key is not found.
- Order is *not* preserved in a dictionary, so printing a dictionary will not necessarily print items in the same order that they were added.

	Python Code	Explanation
In [1]:	<pre>dict1 = {"x":1, "y":2, "z":3} print dict1 {"x":1, "y":2, "z":3}</pre>	Note the colon in the key:value pairs
In [2]:	<pre>dict1["y"] 2</pre>	Dictionary values are accessed using the keys
In [3]:	<pre>dict1["y"] = 10 print mydict {"x":1, "y":10, "z":3}</pre>	Dictionary values can be changed using the "=" assignment operator
In [4]:	<pre>dict1["w"] = 0 print mydict {"x":1, "y":10, "z":3, "w":0}</pre>	New key:value pairs can be assigned using the "=" assignment operator
In [5]:	<pre>dict1.get("a")</pre>	get returns None if the key does not exist
In [6]:	<pre>dict1.get("a", 42) 42</pre>	get can also return a default value
In [7]:	<pre>dict2 = {} print dict2 {}</pre>	Creating an empty dictionary
In [8]:	<pre>dict3 = dict() print dict3 {}</pre>	Another way to create an empty dictionary

2.11 Boolean Expressions

Boolean expressions are statements that either evaluate to True or False. An important use of these expressions is for tests in *conditional code* that only executes if some condition is met. Examples of Boolean expressions include the standard comparison operators below.

	Python Code	Explanation
In [1]:	<pre>5 == 5 True</pre>	Check for equality
In [2]:	<pre>5 != 5 False</pre>	Check for inequality
In [3]:	<pre>3 < 2 False</pre>	Less than

In [4]: `3 <= 3` Less than or equals
 True

In [5]: `"a" < "b"` Strings are compared by lexicographic (dictionary) order
 True

Note that *any* empty container evaluates to False in a Boolean expression. Examples include empty strings (`""`), lists (`[]`), and dictionaries (`{}`).

2.12 If Statements

Python **if** statements provide a way to execute a block of code only if some condition is True. The syntax is:

```
if <condition>:
    <code to execute when condition True>
<following code>
```

Note that:

- `<condition>` is a Boolean expression, which must evaluate to True or False.
- `<condition>` must be followed by a colon, `:`.
- The block of code to execute if `<condition>` is True starts on the next line, and *must be indented*. The convention in Python is that code blocks are indented with 4 spaces.
- This block of code is finished by de-indenting back to the previous level.

Python Code	Explanation
<pre>In [1]: from math import * if pi > e: print "Pi is bigger than e!" Pi is bigger than e!</pre>	<p>The block of code following the if statement only executes if the condition is met</p>

An **else** statement can be added after an **if** statement is complete. This will be followed by the code to execute if the condition is False. The syntax is:

```
if <condition>:
    <code to execute when condition True>
else:
    <code to execute when condition False>
<following code>
```

The following example illustrates an **if-else** statement.

Python Code	Explanation
<pre>In [1]: x, y = 2**3, 3**2 if x < y: print "x < y" else: print "x >= y" x < y</pre>	The block of code following the else statement executes if the condition is <i>not</i> met

Multiple **elif** statements (short for else-if) can be added to create a series of conditions that are tested in turn until one succeeds. Each **elif** must also be followed by a condition and a colon. The syntax is:

```
if <condition 1>:
    <code to execute when condition 1 True>
elif <condition 2>:
    <code to execute when condition 2 True>
else:
    <code to execute if neither condition is True>
<following code>
```

Python Code	Explanation
<pre>In [1]: score = 88 if score >= 90: print "A" elif score >= 80: print "B" elif score >= 70: print "C" elif score >= 60: print "D" else: print "F" B</pre>	Only the first two conditions are tested - the rest are skipped since the second condition is True

It often happens that we want to assign a variable some value if a condition is True, and another value if a condition is False. Using an **if** statement, we would have:

```
if <condition>:
    x = <true_value>
else:
    x = <false_value>
```

Python provides an elegant way to do this in a single line using a *conditional expression*. This has the following syntax.


```
x = <true_value> if <condition> else <false_value>
```

An example is given below.

Python Code	Explanation
<pre>In [1]: x = 22 parity = "odd" if x % 2 else "even" print x, "has", parity, "parity" 22 has even parity</pre>	<p>Note that <code>x % 2</code> returns the remainder when <code>x</code> is divided by 2. Any nonzero value evaluates as <code>True</code></p>

2.13 For Loops

Python **for** loops provide a way to iterate (loop) over the items in a list, string, tuple, or any other *iterable* object, executing a block of code on each pass through the loop. The syntax is:

```
for <iteration variable(s)> in <iterable>:
    <code to execute each time>
<following code>
```

Note that:

- The **for** statement must be followed by a colon, `:`.
- One or more *iteration variables* are bound to the values in `<iterable>` on successive passes through the loop.
- The block of code to execute each time through the loop starts on the next line, and must be indented.
- This block of code is finished by de-indenting back to the previous level.

Sequence objects, such as strings, lists and tuples, can be iterated over as follows.

Python Code	Explanation
<pre>In [1]: for i in [2, 4, 6]: print(i) 2 4 6</pre>	<p>Iterate over the elements of a list. The iteration variable <code>i</code> gets bound to each item in turn</p>
<pre>In [2]: for char in "abc": print char a b c</pre>	<p>Iterate over the characters in a string. The iteration variable <code>char</code> gets bound to each character in turn</p>

In [3]:	<pre>for i, char in enumerate("abc"): print i, char</pre>	enumerate allows an iteration variable to be bound to the index of each item, as well as to the item itself
	<pre>0 a 1 b 2 c</pre>	
In [4]:	<pre>total = 0 for i in xrange(1, 6): total += i print total</pre>	Sum the numbers from 1 to 5. xrange works like range in for loops, but doesn't generate a list first
	<pre>15</pre>	

Dictionaries elements consist of key:value pairs. When iterated over, variables can be bound to the key, the value, or both.

	Python Code	Explanation
In [1]:	<pre>mydict = {"x":1, "y":2, "z":3} for key in mydict: print key</pre>	Iteration over a dictionary binds to the <i>key</i> (note that order is not preserved in a dictionary)
	<pre>y x z</pre>	
In [2]:	<pre>for value in mydict.itervalues(): print value</pre>	Use itervalues to iterate over the dictionary values rather than the keys
	<pre>2 1 3</pre>	
In [3]:	<pre>for key, val in mydict.iteritems(): print key, val</pre>	Use iteritems to iterate over the dictionary keys and values together
	<pre>y 2 x 1 z 3</pre>	

The function **zip** can be used to iterate in parallel over multiple lists of equal length. This returns a list of tuples, with one element of each tuple drawn from each list.

	Python Code	Explanation
In [1]:	<pre>courses = [141, 142, 337] ranks = ["good", "better", "best!"] zipped = zip(courses, ranks) print zipped</pre>	zip (<i>courses</i> , <i>ranks</i>) creates a list of tuples. Each tuple contains one course and one rank, with the tuples in the same order as the list elements
	<pre>[(141, "good"), (142, "better"), (337, "best")]</pre>	

In [2]:	<pre>for c, r in zipped: print c, r</pre>	Multiple iteration variables can be bound at each iteration of a loop
	<pre>141 good 142 better 337 best!</pre>	

2.14 While Loops

Python **while** loops execute a block of code repeatedly as long as some condition is met. The syntax is:

```
while <condition>:
    <code to execute repeatedly>
<following code>
```

Note that for the loop to terminate, the code must change some part of the condition so that it eventually returns False.

Python Code	Explanation
In [1]: <pre style="background-color: #e0f7fa; padding: 10px;">i = 3 while i > 0: print i i -= 1</pre> <pre>3 2 1</pre>	<i>i</i> is printed while it remains greater than zero. The code must change the value of <i>i</i> to ensure that the loop eventually terminates

2.15 Break and Continue

Sometimes we need to end a loop early, either by ending just the current iteration, or by quitting the whole loop. The statements **break** and **continue** provide a way to do this.

- To end the loop completely and jump to the following code, use the **break** statement.
- To end the current iteration and skip to the next item in the loop, use the **continue** statement. This can often help to avoid nested if-else statements.

Python Code	Explanation
In [1]: <pre style="background-color: #e0f7fa; padding: 10px;">vowels = "aeiou" for char in "bewgfiagf": if char in vowels: print "First vowel is", char break</pre> <pre>First vowel is e</pre>	The for loop is terminated by break once the first vowel is found

In [2]: <pre>total = 0 for char in "bewgfiagf": if char in vowels: continue total += 1 print total, "consonants found"</pre>	Skip over the vowels using continue , and just count the consonants 6 consonants found
--	--

2.16 Comprehensions

Often we want to create a container by modifying and filtering the elements of some other container. Comprehensions provide an elegant way to do this, similar to mathematical set-builder notation. For lists, the syntax is:

```
[<expression> for <variables> in <container> if <condition>]
```

The code in $\langle \text{expression} \rangle$ is evaluated for each item in the $\langle \text{container} \rangle$, and the result becomes an element of the new list. The $\langle \text{condition} \rangle$ does not have to be present but, if it is, only elements which satisfy the condition become incorporated into the new list.

Python Code	Explanation
In [1]: <pre>[i**2 for i in range(5)]</pre> <pre>[0, 1, 4, 9, 16]</pre>	$i**2$ is evaluated for every item i in the list
In [2]: <pre>[d for d in range(1,7) if 6 % d == 0]</pre> <pre>[1, 2, 3, 6]</pre>	Divisors of 6 - only elements passing the test $6 \% d == 0$ are included

We can also use a dictionary comprehension to create a dictionary without needing to repeatedly add key:value pairs.

Python Code	Explanation
In [1]: <pre>{i:i**2 for i in range(4)}</pre> <pre>{0:0, 1:1, 2:4, 3:9}</pre>	Create a dictionary from a list. Note the key:value pairs and surrounding curly brackets

2.17 Functions

Functions provide a way to reuse a block of code by giving it a name. The code can then be executed just by calling the function name, with the option of passing in additional data to be used inside the function. The variables used to identify this additional data are the function *parameters*, and the particular values passed in when the function is called are the function *arguments*.

- Functions take a list of required arguments, identified by position.
- Functions can take *keyword* arguments, identified by name. These can also be assigned default values in the function definition to use if no values are passed in.
- Functions can return one or more values using the **return** statement. Note that functions do not *have* to return a value - they could just perform some action instead. A function stops executing as soon as a return statement is encountered.
- An optional documentation string can be added at the start of the function (before the code) to describe what the function does. This string is usually enclosed in triple quotes.

Functions are defined in Python using the **def** statement, with the syntax:

```
def <name> (<parameters>):
    <documentation string>
    <code>
```

The arguments to a function can be specified by position, keyword, or some combination of both. Some examples using just positional arguments are as follows.

	Python Code	Explanation
In [1]:	<pre>def square(x): return x**2 print square(3)</pre> <p>9</p>	The function exits as soon as the return statement is called
In [2]:	<pre>def multiply(x, y): """Return the product xy""" return x*y print multiply(3, 2)</pre> <p>6</p>	Parameters are bound to input data in the order given. The documentation string is placed after the colon and before the code
In [3]:	<pre>def minmax(data): return min(data), max(data) print minmax([1, 3, 7, 2, 10])</pre> <p>(1, 10)</p>	Multiple values are returned as a tuple

Using *keyword* arguments allows default values to be assigned. This is particularly useful when a function can be called with many different options, and avoids having to call functions with a long list of arguments.

- Keyword arguments are specified using `key=default` in place of a positional argument.
- Using keyword instead of positional arguments means we don't need to remember the order of arguments, and allows the defaults to be used most of the time.

- Positional and keyword arguments can be used in the same function, as long as the positional arguments come first.

	Python Code	Explanation
In [1]:	<pre>def close_enough(x, y, tolerance=.1) return abs(x - y) <= tolerance</pre>	The <i>tolerance</i> argument is 0.1 by default
In [2]:	<pre>close_enough(1, 1.05) True</pre>	The default tolerance of 0.1 is used in this case
In [3]:	<pre>close_enough(1, 1.05, tolerance=.01) False</pre>	The default tolerance is overridden by the value of 0.01

If the number of arguments is not known in advance, functions can be defined to take a variable number of positional arguments and/or a variable number of keyword arguments. We are unlikely to be using these options ourselves, although they occur frequently in the documentation for Matplotlib.

- The positional arguments are usually specified as `*args` and are available as a tuple. Individual positional arguments can then be accessed by indexing into the tuple by position.
- The keyword arguments are usually specified as `**kwargs` and are available as a dictionary. Individual keyword arguments can then be accessed by indexing into this dictionary by key.

We may on occasion need to use a simple function in a single place, and not want to have to define and name a separate function for this purpose. In this case we can define an anonymous or *lambda* function just in the place where it is needed. The syntax for a lambda function is:

```
lambda <arguments> : <code>
```

The **lambda** statement returns an unnamed function which takes the arguments given before the colon, and returns the result of executing the code after the colon. Typical uses for lambda functions are where one function needs to be passed in as an argument to a different function.

	Python Code	Explanation
In [1]:	<pre>ages = [21, 19, 150] names = ["Bruce", "Sheila", "Adam"] data = zip(ages, names) sorted(data, key=lambda x : x[0]) [(19, "Sheila"), (21, "Bruce"), (150, "Adam")]</pre>	The <i>key</i> argument to sorted lets us sort the data based on the first list. Using a lambda function means not having to define a separate function for this simple task

2.18 Reading and Writing Files

Several reports for this class will involve reading and analyzing data that has been stored in a file. This typically involves three steps:

- Open the file using the **open** function. This returns a *file handle* - an object we then use to access the text that the file contains.
- Process the file, either line-by-line, or as a single text string.
- Close the file. This is done using the **close** function.

It is possible to read in the entire contents of a file in one go using the functions **read** and **readlines**. However, we may not want to read the entire contents into memory if we are dealing with a large file and just want to extract some information from the text. In this case, it is preferable to iterate over the lines of text that the file contains.

The following examples assume that a file named "filename.txt" has been created in the directory that Python was started in. This file contains the three lines:

```
Leonard
Penny
Sheldon
```

	Python Code	Explanation
In [1]:	<pre>f = open("filename.txt")</pre>	Open "filename.txt" for reading using open
In [2]:	<pre>for line in f: print line,</pre> Leonard Penny Sheldon	The lines of an open file can be iterated over in a for loop. Note the use of a "," after <code>print line</code> , since each line already ends with a new line
In [3]:	<pre>f.close()</pre>	Close "filename.txt" using close
In [4]:	<pre>f = open("filename.txt") first_names = f.read() f.close() first_names</pre> "Leonard\nPenny\nSheldon\n"	read reads in the whole file as a single string. The newlines at the end of each line are shown as "\n" characters
In [5]:	<pre>print first_names</pre> Leonard Penny Sheldon	Printing a string causes the newline characters "\n" to be outputted as new lines

In [6]:	<pre>f = open("filename.txt") data = f.readlines() f.close() data ["Leonard\n", "Penny\n", "Sheldon\n"]</pre>	readlines reads in the whole file as a list, with each line as a separate string
---------	--	---

Files can also be opened for writing using the "w" option to **open**.

Python Code	Explanation
In [1]: <pre>data = ["Hofstadter", "?", "Cooper"] output_file = open("names.txt", "w") for name in data: output_file.write(name + "\n") output_file.close()</pre>	Write each string in the data list to a separate line in the file. Note that new lines are not automatically included, so they need to be added
In [2]: <pre>f = open("names.txt") last_names = f.read() f.close() print last_names</pre> <p>Hofstadter ? Cooper</p>	Check that the "names.txt" file has been written correctly

2.19 Comments

Comments are text that is included in the code but not executed. They are used to document and explain what the code is doing. Python allows two forms of comment.

- A hash symbol `#` means that the rest of the line is a comment, and is not to be executed.
- A documentation string is surrounded by triple quotes `"""`. Everything inside the quotes is ignored.

Python Code	Explanation
In [1]: <pre># This is a single-line comment</pre>	
In [2]: <pre>"""This is a documentation string, which can span multiple lines"""</pre>	

3 NumPy

NumPy (Numerical Python) is the fundamental package for scientific computing with Python. It defines a new kind of container - the ndarray (usually just referred to as an array) - that supports fast and efficient computation. NumPy also defines the basic routines for accessing and manipulating these arrays.

Arrays have the following properties (among others):

- A *shape*, which is a tuple of integers. The number of integers is the number of dimensions in the array, and the integers specify the size of each dimension.
- A *dtype* (data-type), which specifies the type of the objects stored in the array.

In NumPy, the dimensions of an array are referred to as *axes*. An example of an array with dtype int and shape ((4, 5)) is shown below. The first axis has four elements, each of which is an array with 5 elements.

```
[[ 0  1  2  3  4 ]
 [ 5  6  7  8  9 ]
 [10 11 12 13 14 ]
 [15 16 17 18 19 ]]
```

The main differences between NumPy arrays and Python lists are:

- The objects in a NumPy array must all be of the same type - booleans, integers, floats, complex numbers or strings.
- The size of an array is fixed at creation, and can't be changed later.
- Arrays can be multi-dimensional.
- Mathematical operations can be applied directly to arrays. When this is done they are applied elementwise to the array, generating another array as output. This is *much* faster than iterating over a list.
- Indexing for arrays is more powerful than that for lists, and includes indexing using integer and boolean arrays.
- Slicing an array produces a view of the original array, not a copy. Modifying this view will change the original array.

NumPy is well documented online, with a [standard tutorial](#) and [good introductory tutorial](#) available.

3.1 Array Creation

NumPy arrays can be created:

- From a list. The elements of the list need to all be of the same type, or of a kind that can all be cast to the same type. For example, a list consisting of both integers and floats will generate an array of floats, since the integers can all be converted to floats.
- According to a given shape. The array will be initialized differently depending on the function used.
- From another array. The new array will be of the same shape as the existing array, and could either be a copy, or initialized with some other values.
- As a result of an operation on other arrays. The standard mathematical operators can all be applied directly to arrays. The result is an array of the same shape where the operation has been performed separately on corresponding elements.

The following functions are the main ones you need to know.

<code>array</code>	Create an array from a list.
<code>linspace</code>	Return an array of evenly spaced numbers over a specified interval.
<code>arange</code>	Return an array of evenly spaced integers within a given interval.
<code>empty</code>	Return an a new array of a given shape and type, without initializing entries.
<code>zeros</code>	Return an a new array of a given shape and type, filled with zeros.
<code>ones</code>	Return an a new array of a given shape and type, filled with ones.
<code>empty_like</code>	Return a new array with the same shape and type as a given array.
<code>zeros_like</code>	Return an array of zeros with the same shape and type as a given array.
<code>ones_like</code>	Return an array of ones with the same shape and type as a given array.
<code>copy</code>	Return an array copy of the given object.
<code>meshgrid</code>	Returns a pair of 2D x and y grid arrays from 1D x and y coordinate arrays.

These array creation functions are illustrated below.

	Python Code	Explanation
In [1]:	<pre>from numpy import * x = array([1, 2, 3]) print x</pre> <p>[1 2 3]</p>	array(object) creates an array from a list - note that arrays are printed without commas
In [2]:	<pre>x = array([1, 2, 3], dtype=float) print x</pre> <p>[1. 2. 3.]</p>	array(object, dtype) creates an array of type <i>dtype</i> - the integers are now cast to floats

In [3]:	<pre>x = linspace(0, 1, 6) print x</pre>	<code>linspace</code> (<i>start</i> , <i>stop</i> , <i>num</i>) returns <i>num</i> equally spaced points, including endpoints
	<pre>[0. 0.2 0.4 0.6 0.8 1.]</pre>	
In [4]:	<pre>x = arange(5) print x</pre>	<code>arange</code> works just like <code>range</code> , but returns an array instead of a list
	<pre>[0 1 2 3 4]</pre>	

The functions **`empty`**, **`zeros`** and **`ones`** all take a *shape* argument and create an array of that shape, initialized as appropriate.

	Python Code	Explanation
In [1]:	<pre>x = empty((3, 2)) print x</pre>	<code>empty</code> (<i>shape</i>) returns an array of shape <i>shape</i> , initially filled with garbage
	<pre>[[6.93946206e-310 6.93946206e-310] [6.36598737e-314 6.36598737e-314] [6.36598737e-314 0.00000000e+000]]</pre>	
In [2]:	<pre>x = zeros((2, 3)) print x</pre>	<code>zeros</code> (<i>shape</i>) returns an array of shape <i>shape</i> filled with zeros - note the default type is <i>float</i>
	<pre>[[0. 0. 0.] [0. 0. 0.]]</pre>	
In [3]:	<pre>x = ones((2, 3), dtype=int) print x</pre>	<code>ones</code> (<i>shape</i> , <i>dtype</i>) returns an array of shape <i>shape</i> filled with ones - using <i>dtype=int</i> casts the elements to type <i>int</i>
	<pre>[[1 1 1] [1 1 1]]</pre>	

Arrays can be created directly from other arrays using **`empty_like`**, **`zeros_like`**, **`ones_like`** and **`copy`**.

	Python Code	Explanation
In [1]:	<pre>x = arange(3, dtype=float) print x</pre>	Create an array of floats using <code>arange</code>
	<pre>[0. 1. 2.]</pre>	
In [2]:	<pre>y = empty_like(x) print y</pre>	<i>y</i> has the same shape as <i>x</i> , but is initially filled with garbage
	<pre>[0.00000000e+000 6.51913678e+091 6.95022185e-310]</pre>	
In [3]:	<pre>y = zeros_like(x) print y</pre>	<i>y</i> has the same shape as <i>x</i> , but is initialized with zeros
	<pre>[0. 0. 0.]</pre>	

In [4]:	<pre>y = ones_like(x) print y</pre>	y has the same shape as x, but is initialized with ones
	<pre>[1. 1. 1.]</pre>	
In [5]:	<pre>y = copy(x) print y</pre>	y is a copy of x - changing y will not change x
	<pre>[0. 1. 2.]</pre>	

The function **meshgrid**(x, y) creates two-dimensional arrays from one-dimensional x- and y-coordinate axes. One array contains the x-coordinates of all the points in the xy-plane defined by these axes, and the other contains the y-coordinates.

	Python Code	Explanation
In [1]:	<pre>from numpy import * x = arange(4) y = arange(3) X, Y = meshgrid(x, y)</pre>	meshgrid creates 2D x- and y- coordinate arrays from 1D x- and y- coordinate arrays
In [2]:	<pre>print X</pre> <pre>[[0 1 2 3] [0 1 2 3] [0 1 2 3]]</pre>	X is a 2D array containing just the x-coordinates of points in the xy plane
In [3]:	<pre>print Y</pre> <pre>[[0 0 0 0] [1 1 1 1] [2 2 2 2]]</pre>	Y is a 2D array containing just the y-coordinates of points in the xy plane

3.2 Array Properties

NumPy provides a set of functions for accessing the properties of an array.

	Python Code	Explanation
In [1]:	<pre>x = arange(6) type(x)</pre> <pre>numpy.ndarray</pre>	x is of type numpy.ndarray
In [2]:	<pre>x.dtype</pre> <pre>dtype("int64")</pre>	dtype returns the element type - a 64-bit integer
In [3]:	<pre>x.shape</pre> <pre>(6,)</pre>	x is a 1-dimensional array with 6 elements in the first axis

It is possible to create an array from the elements of an existing array, but with the properties changed. The number of dimensions and size of each dimension can be changed

using **reshape** (as long as the total number of elements is the same), and the dtype can be changed using **astype**.

Python Code	Explanation
<pre>In [1]: x = arange(6).reshape((2, 3)) print x [[0 1 2] [3 4 5]]</pre>	reshape creates a view of an array with the same number of elements, but a different shape
<pre>In [2]: y = x.astype(float) print y [[0. 1. 2.] [3. 4. 5.]]</pre>	astype casts the integers in x to floats in y. This creates a new array - modifying it will not alter the original

3.3 Array Operations

Array arithmetic is done on an elementwise basis.

Python Code	Explanation
<pre>In [1]: from numpy import * x = arange(4) print x [0 1 2 3]</pre>	Create an array of consecutive integers using arange
<pre>In [2]: print x + 1 [1 2 3 4]</pre>	1 is added to every element of the array x
<pre>In [3]: print x * 2 [0 2 4 6]</pre>	Every element of the array x is multiplied by 2
<pre>In [4]: print x ** 2 [0 1 4 9]</pre>	Every element of the array x is squared
<pre>In [5]: y = array([3, 2, 5, 1])</pre>	Create a second array
<pre>In [6]: print x print y print x + y [0 1 2 3] [3 2 5 1] [3 3 7 4]</pre>	The elements of x are added to the corresponding elements of y on an element-by-element basis
<pre>In [7]: print x**y [0 1 32 3]</pre>	Exponentiation is done using corresponding elements

Comparison operators and other Boolean expressions are also applied on an element-by-element basis. The result is an array of booleans.

Python Code	Explanation
<pre>In [1]: x = arange(5) print x print x % 2 == 0 [0 1 2 3 4] [True False True False True]</pre>	The Boolean expression is evaluated for each element separately, resulting in an array of booleans
<pre>In [2]: x = arange(4) y = array([3, 2, 5, 1]) print x print y print x < y [0 1 2 3] [3 2 5 1] [True True True False]</pre>	The comparison is done on an elementwise basis, resulting in an array of booleans

NumPy contains vectorized versions of all the basic mathematical functions. Note that they need to be imported before we can use them. Some examples are given below.

Python Code	Explanation
<pre>In [1]: x = arange(3) print x [0 1 2]</pre>	Create an array
<pre>In [2]: print sin(x) [0. 0.84147098 0.90929743]</pre>	sin is applied to each element, to create a new array
<pre>In [3]: print exp(x) [1. 2.71828183 7.3890561]</pre>	exp is the exponential operator
<pre>In [4]: x = random.randint(5, size=(2,3)) print x [[1 4 3] [2 2 3]]</pre>	random.randint returns an array of a given size filled with randomly selected integers from a given range
<pre>In [5]: print x.min(), x.max() 1 4</pre>	min and max calculate the minimum and maximum values across the entire array
<pre>In [6]: print x.min(axis=0) print x.min(axis=1) [1 2 3] [1 2]</pre>	The <i>axis</i> argument finds each minimum along a given axis. The resulting array is the shape of the original array, but with the given axis removed

In [7]:	<pre>print x.sum() 15</pre>	sum sums all the elements of an array
In [8]:	<pre>print x.sum(axis=0) [3 6 6]</pre>	Providing the <i>axis</i> argument sums along the given axis

3.4 Accessing Arrays

Arrays can be indexed and sliced using square brackets in the same way as lists. An index or `start:end` filter needs to be provided for each axis, separated by commas. Indexing is zero-based, as it is with lists.

	Python Code	Explanation
In [1]:	<pre>from numpy import * x = arange(20).reshape((4, 5)) print x [[0 1 2 3 4] [5 6 7 8 9] [10 11 12 13 14] [15 16 17 18 19]]</pre>	reshape provides a fast way to create a 2D array
In [2]:	<pre>print x[1,2] 7</pre>	Indexing is done into each axis in order - row 1, column 2
In [3]:	<pre>print x[1,:] [5 6 7 8 9]</pre>	Slicing selects every element of the first axis
In [4]:	<pre>print x[:,1] [1 6 11 16]</pre>	Slicing selects the first element of every axis
In [5]:	<pre>print x[1:3, 1:4] [[6 7 8] [11 12 13]]</pre>	Slice rows 1 and 2 using 1:3, then slice columns 1, 2 and 3 using 1:4

NumPy also offers some fancy indexing tricks. The first is to index using an array of integers. An array with axes the same size as the indexing array is returned, with elements selected from the array according to the integers in the indexing array.

	Python Code	Explanation
In [1]:	<pre>x = arange(9)**2 print x [0 1 4 9 16 25 36 49 64]</pre>	First create the array using arange , then square each element

In [2]:	<pre>index = array([1, 3]) print x[index]</pre>	An array is returned containing elements from the first array, selected according to the integers in the second array
	<pre>[1 9]</pre>	
In [3]:	<pre>index = array([[1, 3], [7, 2]]) print index print x[index]</pre>	When indexing using an integer array, the returned array has the same shape as the indexing array
	<pre>[[1 3] [7 2]]</pre>	
	<pre>[[1 9] [49 4]]</pre>	

We can also index using arrays of booleans. In this case, only the elements corresponding to True values in the indexing array are returned. A common use of this technique is to filter out the elements of an array that satisfy some condition. This can be done by first applying the condition to the array to generate an array of booleans, then using the resulting array as an index. The result is that only elements for which the condition holds true are selected.

	Python Code	Explanation
In [1]:	<pre>x = arange(20) index3 = (x % 3 == 0) index5 = (x % 5 == 0)</pre>	<code>index3</code> and <code>index5</code> are boolean arrays containing True elements for the integers that are divisible by 3 and 5 respectively.
In [2]:	<pre>print x[index3]</pre>	Select the elements of <code>x</code> that are divisible by 3
	<pre>[0 3 6 9 12 15 18]</pre>	
In [3]:	<pre>print x[index5]</pre>	Select the elements of <code>x</code> that are divisible by 5
	<pre>[0 5 10 15]</pre>	
In [4]:	<pre>print x[logical_or(index3, index5)]</pre>	The function logical_or performs an elementwise "or". The result is the integers divisible by either 3 or 5
	<pre>[0 3 5 6 9 10 12 15 18]</pre>	

4 Matplotlib

Matplotlib is a 2D plotting library for Python. It can be used to generate graphs, histograms, bar charts, contour plots, scatter plots, and many other kinds of mathematical graphics. The “pyplot” interface provides a MATLAB-like interface for simple plotting, and is the main one we will be using in class. The [online reference](#) provides a full description of the available functions. A [good tutorial](#) is also available online.

The following commands are the main ones used for creating and formatting graphs.

<code>plot</code>	Plot lines and/or markers.
<code>show</code>	Display a figure.
<code>title</code>	Set a title for the graph.
<code>xlabel/ylabel</code>	Set labels for the x and y axes.
<code>xlim/ylim</code>	Get or set the range of x and y values to be displayed.
<code>xticks/yticks</code>	Get or set the locations and labels for the tick marks on the x and y axes.
<code>subplot</code>	Plot multiple graphs in one figure.
<code>figure</code>	Create a new figure.
<code>fill_between</code>	Fill the area between two curves.
<code>legend</code>	Put a legend on the graph.

Colors, line styles, and marker styles can all be set to create customized graphs. These are usually specified as strings, with the most frequently used options as follows.

Style options		Colors	
"-"	solid line	"b"	blue
"- -"	dashed line	"g"	green
"."	point marker	"r"	red
"o"	circle marker	"c"	cyan
"s"	square marker	"m"	magenta
"+"	plus marker	"y"	yellow
"x"	x marker	"k"	black
"D"	diamond marker	"w"	white

4.1 Basic Plotting

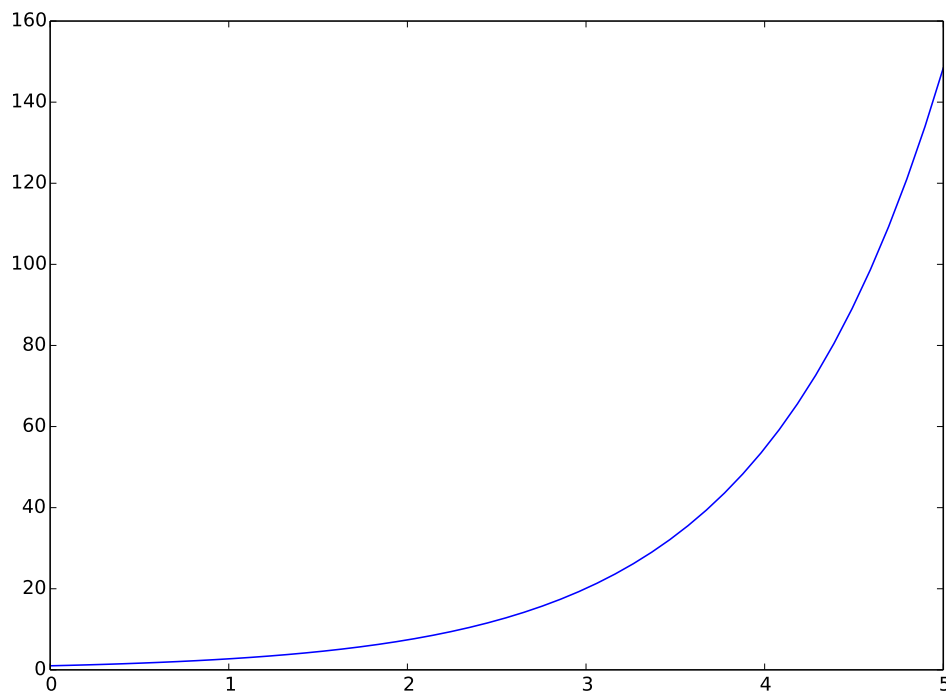
Functions can be graphed using a call to **plot**(*x*, *y*), followed by a call to **show**. Note that:

- The *x* parameter contains the x-coordinates of the points to plot, and the *y* parameter contains the y-coordinates.
- We need to import the required NumPy and Pylab functions for array manipulation and plotting. If using IPython notebook, this can also be done using the IPython magic `%pylab`. In this case, the call to **show** is done for us automatically when the cell is executed.
- The default plotting behavior is to connect the points with a blue line.

The following example plots the exponential function in the range [0, 5].

```
from numpy import *          # Import everything from numpy
import pylab as pl          # Import plotting functions from pylab

x = linspace(0, 5)          # Create array of equally spaced values
pl.plot(x, exp(x))          # Plot the exponential function
pl.show()                   # Finally, show the figure
```



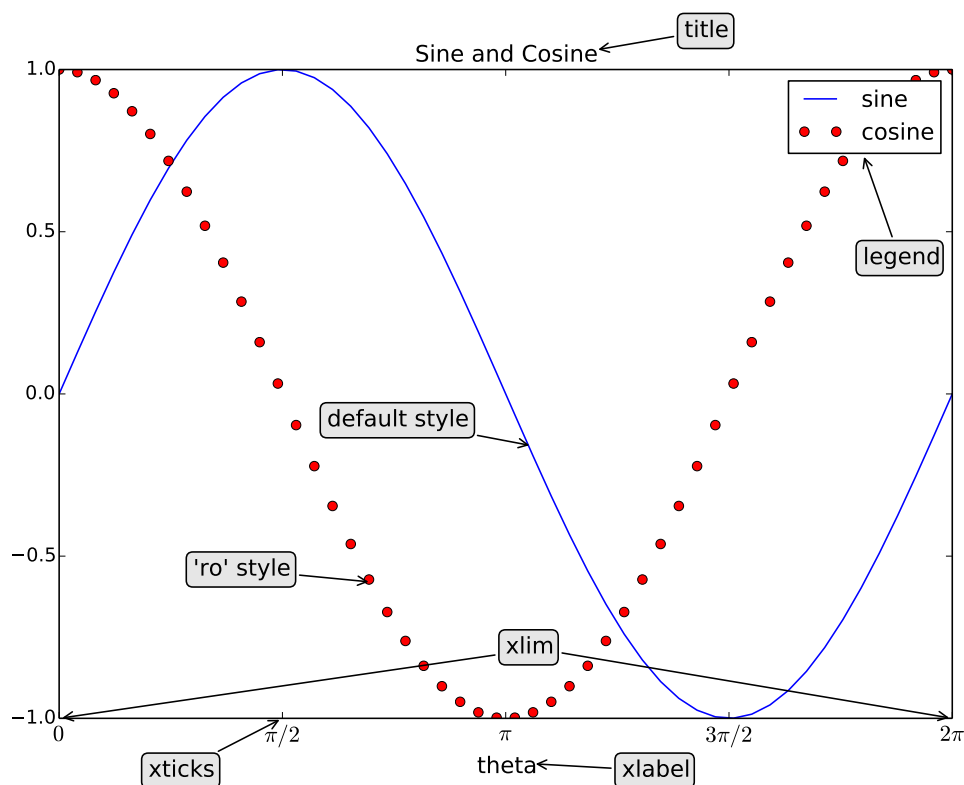
4.2 A More Complex Plotting Example

A range of options are available for customizing plots. These are illustrated in the example below, which plots a sine and cosine curve on the same graph. Note that:

- The third argument to plot can be used to set colors, line types and marker types.
- Plot can be called multiple times, followed by a single call to show.

```
from numpy import *
import pylab as pl
x = linspace(0, 2*pi, 50)
pl.figure(figsize=(10,7))
pl.plot(x, sin(x), label='sine')
pl.plot(x, cos(x), 'ro', label='cosine')
pl.xlabel('theta')
pl.xlim(0, 2*pi)
ticks = [i*pi/2 for i in range(5)]
labels = [r'$0$', r'$\pi/2$', r'$\pi$', r'$3\pi/2$', r'$2\pi$']
pl.xticks(ticks, labels, size='large')
pl.title('Sine and Cosine')
pl.legend()
pl.show()
```

Imports linspace, sin, cos
Import plotting functions
Plot 50 points on the x-axis
Set the size of the figure
Default style is a blue line
Use 'ro' for red circles
Label the x-axis
Limit x-axis to this range
Locations of ticks on x-axis
Labels for the x-axis ticks
- these are LaTeX strings
Add the x-ticks and labels
Add a title
Legend uses the plot labels
Finally, show the figure



4.3 Bar Plots

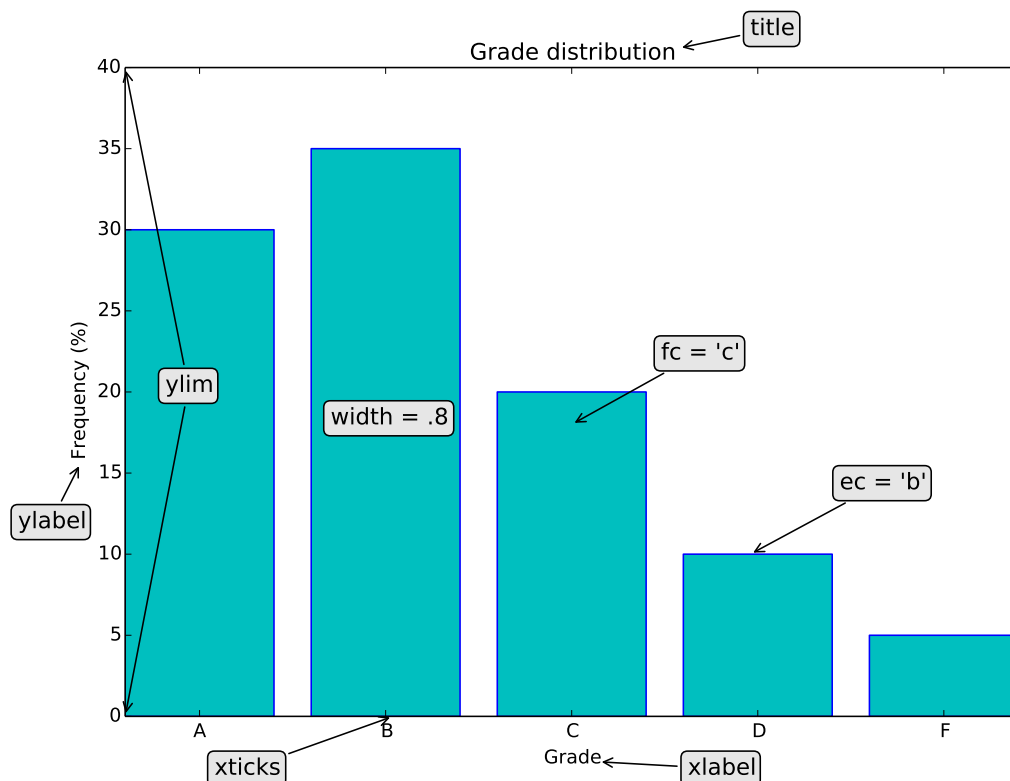
The function **bar** is used to create bar plots.

- Bars are described by their height, width, and position of the left and bottom edges.
- The *width* argument can be used to make bars thinner or thicker.
- The face color and edge color of the bars can be specified independently.

The following example shows a bar plot with the face color set to "c" (cyan) and edge color set to "b" (blue). Labels are positioned at the centers of the bars.

```
import pylab as pl                                # Import plotting functions

grades = ['A', 'B', 'C', 'D', 'F']                # Used to label the bars
freqs = [30, 35, 20, 10, 5]                      # Bar heights are frequencies
width = 0.8                                       # Relative width of each bar
ticks = [width/2 + i for i in range(5)]          # Ticks in center of the bars
pl.bar(range(5), freqs, fc='c', ec='b')           # fc/ec are face/edge colors
pl.xticks(ticks, grades)                         # Place labels for the bars
pl.ylim(0, 40)                                   # Set the space at the top
pl.title('Grade distribution')                   # Add a title
pl.xlabel('Grade')                              # Add a label for the x-axis
pl.ylabel('Frequency (%)')                      # Add a label for the y-axis
pl.show()                                        # Finally, show the figure
```



4.4 Histograms

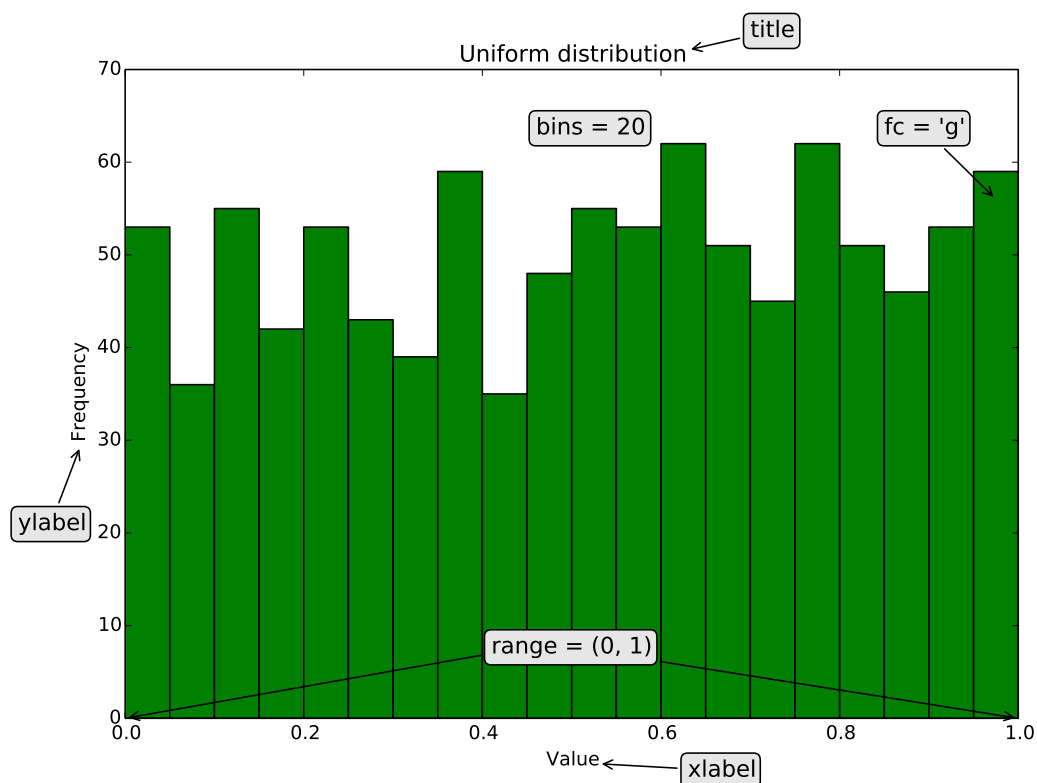
The function **hist** is used to plot histograms. These group numerical data into “bins”, usually of equal width, in order to show how the data is distributed.

- Each bin covers a range of values, with the height of each bin indicating the number of points falling in that range.
- The first argument is an array or sequence of arrays.
- The *bins* argument specifies the number of bins to use.
- The *range* argument specifies the range of values to include.

The following example plots a histogram of 1000 samples drawn from a uniform probability distribution over $[0, 1)$.

```
from numpy import *           # Make random.rand available
import pylab as pl           # Import plotting functions

x = random.rand(1000)         # 1000 random values in [0, 1)
pl.hist(x, bins=20, range=(0,1), fc='g') # Create histogram with 20 bins
pl.title('Uniform distribution')        # Add a title
pl.xlabel('Value')                      # Add a label for the x-axis
pl.ylabel('Frequency')                  # Add a label for the y-axis
pl.show()                              # Finally, show the figure
```



4.5 Contour Plots

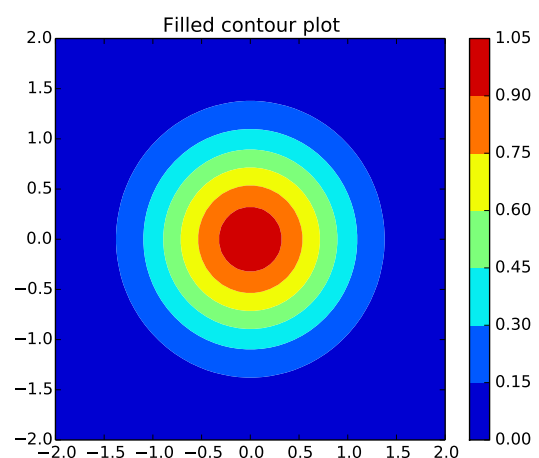
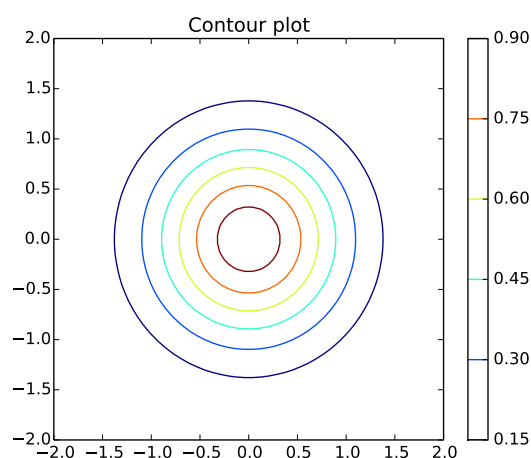
The functions **contour** and **contourf** are used for contour plots and filled contour plots respectively. These are projections of a graph surface onto a plane, with the contours showing the level curves of the graph.

- The first two arguments are one dimensional arrays representing the x- and y-coordinates of the points to plot.
- The third coordinate is a two dimensional array representing the z-coordinates.
- Contour levels are automatically set, although they can be customized.
- A colorbar can be added to display the level curves.

The following examples are of a filled and unfilled contour plot of the two-dimensional Gaussian function, $f(x, y) = e^{-(x^2+y^2)}$.

```
import pylab as pl          # Import plotting functions
from numpy import *         # Import numpy

x = linspace(-2,2)          # Locations of x-coordinates
y = linspace(-2,2)          # Locations of y-coordinates
XX, YY = meshgrid(x, y)     # meshgrid returns two 2D arrays
z = exp(-(XX**2 + YY**2))    # z is a 2D Gaussian
pl.figure(figsize=(14,5))   # Set the figure dimensions
pl.subplot('121')           # First subplot, 1 row, 2 columns
pl.contour(x, y, z)         # Contour plot
pl.title('Contour plot')    # Title added to first subplot
pl.colorbar()               # Color bar added to first subplot
pl.subplot('122')           # Second subplot
pl.contourf(x, y, z)        # Filled contour plot
pl.title('Filled contour plot') # Title added to second subplot
pl.colorbar()               # Color bar added to second subplot
pl.show()                   # Finally, show the figure
```



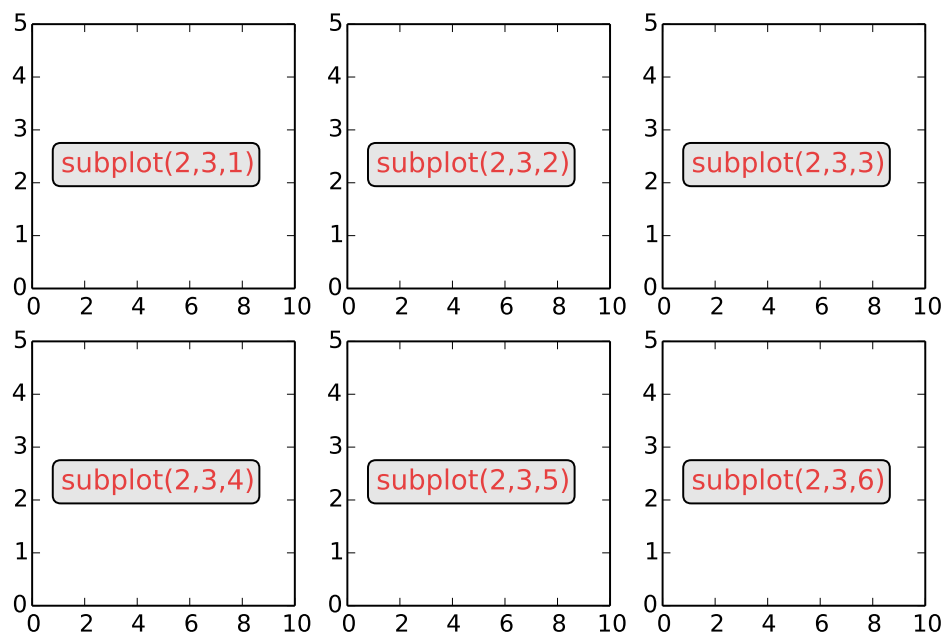
4.6 Multiple Plots

The function **subplot** is used to plot multiple graphs on a single figure. This divides a figure into a grid of rows and columns, with plotting done in the currently active subplot.

- Calls to subplot specify the number of rows, number of columns, and subplot number.
- Subplots are numbered from left to right, row by row, starting with 1 in the top left.
- All plotting is done in the location specified in the most recent call to subplot.
- If there are less than 10 rows, columns and subplots, subplot can be called with a string argument. For example, subplot(2, 3, 4) is the same as subplot("234").

The example below uses 2 rows and 3 columns. The “subplot” calls displayed on the figure show which call corresponds to each grid location.

```
import pylab as pl                                # Import plotting functions
fig=pl.figure(figsize=(8,5))                      # Set the figure dimensions
nrows=2                                           # Number of rows
ncols=3                                           # Number of columns
for i in range(nrows*ncols):
    pl.subplot(nrows,ncols,i+1)                  # Subplot numbering starts at 1
```



4.7 Formatting Mathematical Expressions

LaTeX provides a way to format mathematical expressions in Matplotlib graphs in a similar way to IPython notebook Markdown cells.

- Expressions are identified using `r"$\langle formula \rangle$"`.
- The syntax for `\langle formula \rangle` is the same as that described in section 1.5.3.
- These expressions can be used anywhere a string is used, such as titles, axis and ticks labels, and legends.

5 Additional Topics

5.1 Loading Numerical Files

We often need to load files containing numerical data into a NumPy array for further processing and display. Such data files typically consist of:

- Header information. This describes what the data represents and how it is formatted.
- A set of rows of numerical data. Each row contains the same number of values, separated by some string such as a comma or tab.

The NumPy function `numpy.loadtxt` can be used to load such data. This returns a NumPy array, where each row corresponds to a line in the data file. The first argument to this function is the data file name. Some of the optional keyword arguments are shown below.

- *dtype*. This is the data type of values in the array, which are floats by default.
- *delimiter*. This is the string used to separate values in each row. By default, any whitespace such as spaces or tabs are considered delimiters.
- *skiprows*. This is the number of rows to ignore at the start of the file before reading in data. It is usually used to skip over the header information, and defaults to 0.

The example shown below uses a file called "weather.dat", which contains the following:

Day	High-Temp	Low-Temp
1	77	56
2	79	62

This data is loaded as follows.

Python Code		Explanation
In [1]:	<pre>from numpy import loadtxt</pre>	Import the loadtxt function
In [2]:	<pre>data = loadtxt("weather.dat", dtype=int, skiprows=1)</pre>	Load the "weather.dat" file, skipping the first row, and creating an array of integers
In [3]:	<pre>print data [[1 77 56] [2 79 62]]</pre>	The data array is a 2×3 integer array - for floats, the <i>dtype</i> argument would not be used

5.2 Animation

An animation consists of a sequence of frames which are displayed one after the other. Animation using Matplotlib essentially involves updating the data associated with some drawn object or objects (such as points or lines), and redrawing these objects. Producing an animation therefore involves the following steps:

- Set up the variables and data structures relating to the animation.
- Draw the first frame.
- Repeatedly update the frame with new data.

Animations are generated using `FuncAnimation` from the `matplotlib.animation` module. This takes the following required arguments:

- *fig*. This is the figure in which the animation is to be drawn. It can be obtained using either of the Matplotlib `figure` or `subplots` functions.
- *func*. This specifies the function to call to perform a single step of the animation. It should take a single argument which is the frame number (an integer). The frame number is used to update the values of drawn objects such as points or lines. If the *blit* keyword argument is True, this function should return a tuple of the modified objects that need to be redrawn.

`FuncAnimation` also takes the following keyword arguments.

- *frames*. An integer specifying the number of frames to generate.
- *init_func*. This specifies the function which is called once at the start to draw the background that is common to all frames. If the *blit* keyword argument is True, this function should also return a tuple of the modified objects that need to be redrawn.
- *interval*. This argument specifies the time (in ms) to wait between drawing successive frames.
- *blit*. If True, the animation only redraws the parts of the plot which have changed. This can help ensure that successive frames are displayed quickly.
- *repeat*. If True (the default), the animation will repeat from the beginning once it is finished.

The following example for IPython Notebook animates a point circling the origin with constant angular velocity. The `animate` function is defined to update the position of the point based on the frame number.

```

%pylab # Note %pylab, not %pylab inline
from matplotlib import animation

omega = .02 # Angular velocity
fig, ax = subplots(figsize=(4,4)) # Get the figure & axes for the plot
ax.set_aspect('equal') # Make the axes have the same scale
point, = plot([], [], 'ro', ms=10) # "point" is the object drawn by plot
xlim(-1.5,1.5) # - note that "plot" returns a tuple
ylim(-1.5,1.5) # Set limits for the entire animation

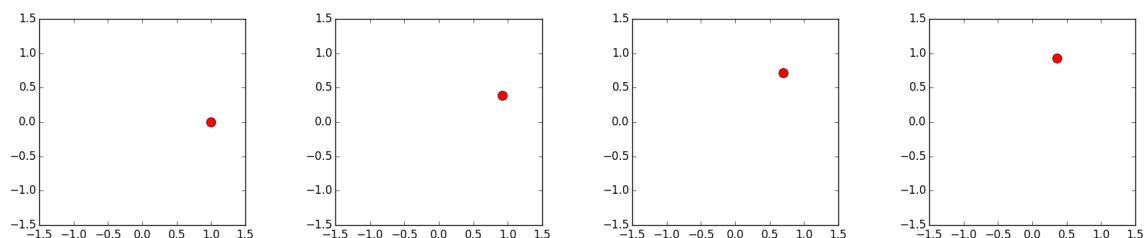
# Initialization function. This is called once to plot the background.
def init():
    point.set_data([], [])
    return point, # Return a tuple of the modified objects

# Animation function. This is called once per animation step.
# The integer i is the frame number.
def animate(i):
    x = cos(i*omega)
    y = sin(i*omega)
    point.set_data(x, y) # Update the x, y coordinates of the point
    return point, # Return a tuple of the modified objects

# Start the animator with a call to "FuncAnimation"
animation.FuncAnimation(fig, animate, init_func=init, frames=100,
                        interval=20, blit=True)

```

Some frames from this animation are shown below.



Note that in IPython Notebook the IPython magic we need to use is `%pylab` rather than `%pylab inline`. Inline graphs in IPython Notebook are static, meaning that once drawn, they cannot be updated. Using `%pylab` generates graphs in a separate window, where the updated data can be displayed.

5.3 Images

Matplotlib provides functions for saving, reading, and displaying images. These images are either 2- or 3-dimensional NumPy arrays. In both cases, the first two axes of the array correspond to the rows and columns of the image. The third axis corresponds to the color of the pixel at each (column, row) coordinate.

- For a 2D array, the array values are floats in the range $[0, 1]$. These represent the luminance (brightness) of a grayscale image from black (0) to white (1).
- For a 3D array, the third axis can have either 3 or 4 elements. In both cases, the first three elements correspond to the red, green, and blue components of the pixel color. These can be either floats in the range $[0, 1]$, or 8-bit integers of type 'uint8'. A fourth element corresponds to an “alpha” value representing transparency.

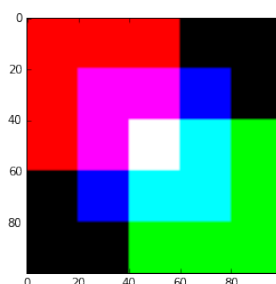
The main functions we use are:

`imread` Read an image file into an array.

`imsave` Save an image to file.

`imshow` Display an image array.

	Python Code	Explanation
In [1]:	<code>img = zeros((100, 100, 3))</code>	Create an image array of 100 rows and columns, set to zero
In [2]:	<code>img[:60,:60,0] = 1.</code>	Set the top-left corner to red
In [3]:	<code>img[40:,40:,1] = 1.</code>	Set the lower-right corner to green
In [4]:	<code>img[20:80,20:80,2] = 1.</code>	Set the center square to blue
In [5]:	<code>imsave("squares.png", img) img2 = imread("squares.png") imshow(img2)</code>	Save the img array to the "squares.png" file, read the file back to the img2 array, and display the image



5.4 Random Number Generation

NumPy provides a library of functions for random number generation in the `random` module. These return either a sample, or an array of samples of a given size, drawn from a given probability distribution. The main functions we use are:

- `random.rand` Samples are drawn from a uniform distribution over $[0, 1)$.
- `random.randint` Samples are integers drawn from a given range.
- `random.randn` Samples are drawn from the “standard normal” distribution.
- `random.normal` Samples are drawn from a normal (Gaussian) distribution.

The following examples illustrate the use of these functions.

	Python Code	Explanation
In [1]:	<pre>from numpy import * print random.rand() 0.723812203628</pre>	Return a single random number uniformly drawn from the interval $[0, 1)$
In [2]:	<pre>print random.rand(3) [0.74654564 0.58764797 0.15557362]</pre>	Return an array of 3 random numbers drawn from $[0, 1)$
In [3]:	<pre>print random.rand(2, 3) [[0.65382707 0.71701863 0.5738609] [0.22064692 0.57487732 0.5710538]]</pre>	Return an array of size (2, 3) of random numbers drawn from $[0, 1)$
In [4]:	<pre>print random.randint(7) 3</pre>	Return an integer drawn from $\{0, 1, 2, 3, 4, 5, 6\}$. Note that 7 is not included
In [5]:	<pre>print random.randint(5,9,size=(2,4)) [[5 5 5 8] [7 8 7 6]]</pre>	Return an array of size (2, 4) of integers drawn from $\{5, 6, 7, 8\}$
In [6]:	<pre>print random.randn(3) [0.47481788 -0.7690172 0.42338774]</pre>	Return array of samples drawn from “standard” normal distribution ($\mu = 0, \sigma = 1$)
In [7]:	<pre>print random.normal(100, 15) 111.676554337</pre>	Return a sample drawn from a normal distribution with $\mu = 100, \sigma = 15$

5.5 Sound Files

Sound is a vibration that propagates through a medium such as air as a wave of pressure and displacement. Recording devices such as microphones convert this wave to an electrical signal. This signal is then sampled at regular intervals and converted to a sequence of numbers, which correspond to the wave amplitude at given times.

The WAVE (or WAV) file format is a standard for storing such audio data without compression. WAVE files contain two main pieces of information:

- The rate at which the wave has been sampled, usually 44,100 times per second.
- The audio data, usually with 16 bits used per sample. This allows $2^{16} = 65,536$ different amplitude levels to be represented.

The module **scipy.io.wavfile** provides functions to read and write such files.

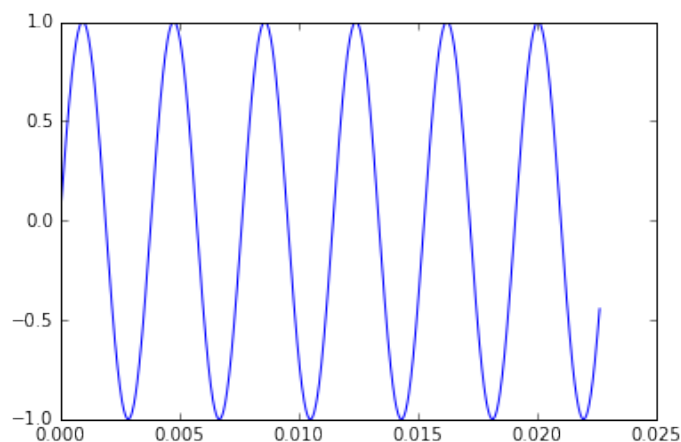
`scipy.io.wavfile.read` Read a WAV file, returning the sample rate and the data.

`scipy.io.wavfile.write` Write a NumPy array as a WAV file.

The following example creates and saves a WAV file with a single frequency at middle C, then plots the first 1000 samples of the data.

```
from numpy import linspace
from scipy.io import wavfile
from pylab import plot, show

rate = 44100                                     # Number of samples/second
end = 10                                          # The file is 10 seconds long
time = linspace(0, end, rate*end+1)            # Time intervals are 1/rate
freq = 261.625565                               # Frequency of "middle C"
data = sin(2*pi*freq*time)                     # Generate the sine wave
wavfile.write("middleC.wav", rate, data)        # Write the array to a wav file
plot(time[:1000], data[:1000])                 # Plot the first 1000 samples
show()                                           # Finally, show the figure
```



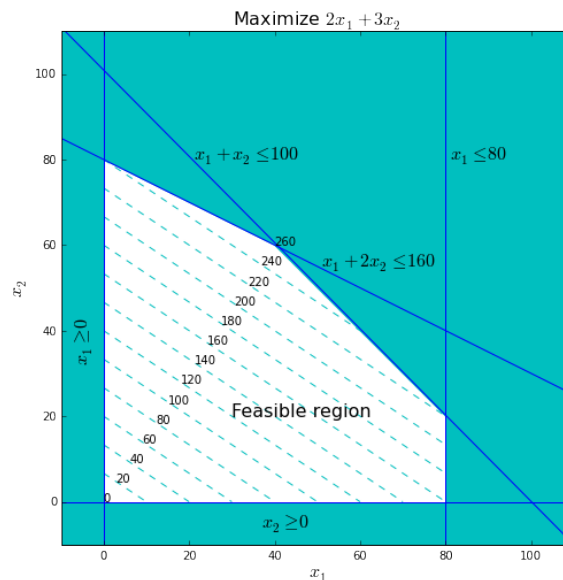
5.6 Linear Programming

Linear programming problems are a special class of optimization problem. They involve finding the maximum (or minimum) of some linear objective function $f(\mathbf{x})$ of a vector of variables $\mathbf{x} = (x_1, x_2, \dots, x_n)$, subject to a set of linear equality and inequality constraints.

Since the objective function and constraints are linear, we can represent the problem as:

Maximize $\mathbf{c}^T \mathbf{x}$, where the vector \mathbf{c} contains the coefficients of the objective function,
 subject to $\mathbf{A}_{ub} * \mathbf{x} \leq \mathbf{b}_{ub}$, where \mathbf{A}_{ub} is a matrix and \mathbf{b}_{ub} a vector,
 and $\mathbf{A}_{eq} * \mathbf{x} = \mathbf{b}_{eq}$, where \mathbf{A}_{eq} is a matrix and \mathbf{b}_{eq} a vector.

An example of such a problem would be: $\mathbf{x} = \{x_1, x_2\}$. Maximize $f(\mathbf{x}) = 2x_1 + 3x_2$ subject to the inequality constraints (i) $0 \leq x_1 \leq 80$, (ii) $x_2 \geq 0$, (iii) $x_1 + x_2 \leq 100$, and (iv) $x_1 + 2x_2 \leq 160$. This example is graphed below, showing the level curves of $f(\mathbf{x})$.



The function `scipy.optimize.linprog` implements the “simplex algorithm” we discuss in class to solve this problem. The arguments to this function are the values \mathbf{c} , \mathbf{A}_{ub} , \mathbf{b}_{ub} , \mathbf{A}_{eq} and \mathbf{b}_{eq} given above. An optional *bounds* argument represents the range of permissible values that the variables can take, with `None` used to indicate no limit.

Applying **linprog** to this problem is done as shown below. Note that **linprog** finds the minimum of $f(\mathbf{x})$, so we use negative values for the \mathbf{c} coefficients to find a maximum.

```
c = array([-2, -3])           # Negative coefficients of f(x)
A_ub = array([[1, 1], [1, 2]]) # Matrix of the inequality coefficients
b_ub = array([100, 160])      # Vector of the inequality upper bounds
bounds = [(0, 80), (0, None)] # Each tuple is a (lower, upper) bound
result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds)
print result.x                # The "x" field holds the solution
```

This yields the correct solution for x_1 and x_2 , as seen in the graph above:

```
[ 40.  60.]
```

6 Programming Style

This chapter contains some tips on how to make programs easier to read and understand. Programs are written first and foremost to be understood by human beings, not by computers. Ideally, it should be possible a year from now for you to pick up the code that you're writing today and still understand what you were doing and why. (It should also be possible for the instructor to understand it a week from now...)

6.1 Choosing Good Variable Names

Good variable names make reading and debugging a program much easier. Well chosen names are easy to decipher, and make the intent clear without additional comments.

- A variable name should fully and accurately describe the data it represents. As an example, `date` may be ambiguous whereas `current_date` is not. A good technique is to state in words what the variable represents, and use that for the name.
- Names that are too short don't convey enough meaning. For example, using `d` for date or `cd` for current date is almost meaningless. Research shows that programs with variable names that are about 9 to 15 characters long are easiest to understand and debug.
- Variable names should be problem-oriented, referring to the problem domain, not how the problem is being solved. For example, `planet_velocity` refers to the problem, but `vector_3d` refers to how this information is being represented.
- Loop indices are often given short, simple names such as `i`, `j` and `k`. This is okay here, since these variables are just used in the loop, then thrown away.
- If loops are nested, longer index names such as `row` and `column` can help avoid confusion.
- Boolean variables should have names that imply either True or False. For example, `prime_found` implies that either a prime has been found, or it hasn't.
- Boolean variables should be positive. For example, use `prime_found` rather than `prime_not_found`, since negative names are difficult to read (particularly if they are negated).
- Named constants should be in uppercase and refer to what the constant represents rather than the value it has. For example, if you want to use the same color blue for the font in every title, then define the color in one place as `TITLE_FONT_COLOR` rather than `FONT_BLUE`. If you later decide to have red rather than blue titles, just redefine `TITLE_FONT_COLOR` and it keeps the same meaning.

6.2 Choosing Good Function Names

The [recommended style](#) for naming functions in Python is to use all lowercase letters, separated by underscores as necessary. As with variable names, good function names can help make the intent of the code much easier to decipher.

- For procedures (functions that do something and don't return a value), use a verb followed by an object. An example would be `plot_prime_distribution`.
- For functions that return values, use a description of what the returned value represents. An example would be `miles_to_kilometers`.
- Don't use generic names such as `calculate_stuff` or numbered functions such as `function1`. These don't tell you what the function does, and make the code difficult to follow.
- Describe everything that the function does, and make the function name as long as is necessary to do so. If the function name is too long, it may be a sign that the function itself is trying to do too much. In this case, the solution is to use shorter functions which perform just one task.

6.3 No "Magic Numbers"

Magic numbers are numbers such as 168 or 9.81 that appear in a program without explanation. The problem with such numbers is that the meaning is unclear from just reading the number itself.

- Numbers should be replaced with named constants which are defined in one place, close to the start of your code file.
- Named constants make code more readable. It's a lot easier to understand what `HOURS_PER_WEEK` is referring to than the number 168.
- If a number needs to change, named constants allow this change to be done in one place easily and reliably.

6.4 Comments

It's not necessary to comment every line of code, and "obvious" comments which just repeat what the code does should be avoided. For example, the endline comment in the following code is redundant and does nothing to explain what the code is for.

```
x += 1 # Add 1 to x
```

Good comments serve two main purposes:

- "Intent" comments explain the purpose of the code. They operate at the level of the problem (*why* the code was written) - rather than at the programming-language level (*how* the code operates). Intent is often one of the hardest things to understand when reading code written by another programmer.

- “Summary” comments distill several lines of code into one or two sentences. These can be scanned faster than the code itself to quickly understand what the code is doing. For example, suppose you are creating several different graphs for a report. A summary comment before each plot and its associated set of formatting commands can describe which figure in the report the code is producing.

Endline comments are those at the end of a line, after the code. They are best avoided for a number of reasons.

- Endline comments are short by necessity as they need to fit into the space remaining on a line. This means that they tend to be cryptic and uninformative.
- Endline comments are difficult to keep aligned (particularly as the code changes), and if they’re not aligned they become messy and interfere with the visual structure of the code.

A final note is to get in the habit of documenting code files. At the top of every file, include a block comment describing the contents of the file, the author, and the date the file was created. An example would be:

```
# MTH 337: Intro to Scientific and Mathematical Computing, Fall 2015  
# Report 1: Primitive Pythagorean Triples  
# Created by Adam Cunningham 8/31/2015
```

6.5 Errors and Debugging

The following suggestions may help to reduce errors.

- Test each function completely as you go.
- In the initial stages of learning Python, test each few lines of code before moving on to the next.
- Add “print” statements inside a function to print out the intermediate values of a calculation. This can be used to check that a function is working as required, and can always be commented out afterwards.

In the event of an error being generated, IPython will typically give as much information as possible about the error. If this information is not sufficient, the `%debug` magic will start the IPython debugger. This lets the current values of variables inside a function be examined, and allows code to be stepped through one line at a time.

7 Further Reading

The following books may prove useful for further study or reference.

- L. Felipe Martins. *IPython Notebook Essentials*. Packt Publishing Ltd, Birmingham. 2014.
A fairly short introduction to using NumPy and Matplotlib in IPython Notebooks. This is not a Python tutorial, although there is a brief review of Python in the appendix.
- Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press. 2004.
A general guide to code writing and software construction, this book focuses on questions of software design rather than any specific language. More useful to an intermediate-level programmer who wants to improve their skills. No references to Python.
- Bruce E. Shapiro. *Scientific Computation: Python Hacking for Math Junkies*. Sherwood Forest Books, Los Angeles. 2015.
A tutorial for Python, NumPy and Matplotlib that also covers many of the same scientific and mathematical topics as this class.
- John M. Stewart. *Python for Scientists*. Cambridge University Press, Cambridge. 2014.
A good introduction to Python, NumPy, Matplotlib and three-dimensional graphics. Extensive treatment of numerical solutions to ordinary, stochastic, and partial differential equations.