



MTH 331

Introduction to
Scientific and Mathematical Computing

Fall 2016

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	Weekly Reports	5
1.3.1	Presenting and Interpreting Results	5
1.4	Jupyter Notebook	5
1.4.1	Magics	7
1.4.2	Markdown	8
1.4.3	L ^A T _E X	9
2	Programming Python	10
2.1	Numbers	10
2.2	Booleans	12
2.3	Strings	12
2.4	Formatting Strings	14
2.5	Type Conversions	16
2.6	Variable Names	17
2.7	Modules	18
2.8	Lists	19
2.9	Tuples	22
2.10	Sets	23
2.11	Dictionaries	24
2.12	Boolean Expressions	26
2.13	If Statements	26
2.14	Conditional Expressions	28
2.15	For Loops	28
2.16	While Loops	31
2.17	Break and Continue	32
2.18	Comprehensions	32
2.19	Generator Expressions	33
2.20	Functions	34
2.21	Error Handling with Try-Except	36
2.22	Reading and Writing Files	37
2.23	Comments	39

3	NumPy	40
3.1	Array Creation	41
3.2	Array Properties	44
3.3	Array Operations	45
3.4	Array Indexing and Slicing	47
3.5	Indexing with Integer Arrays	48
3.6	Indexing with Boolean Arrays	49
4	Matplotlib	50
4.1	Basic Plotting	51
4.2	A More Complex Plotting Example	52
4.3	Bar Plots	53
4.4	Polar Plots	54
4.5	Histograms	55
4.6	Pie Charts	56
4.7	Contour Plots	57
4.8	Multiple Plots	58
4.9	Formatting Text	59
4.10	Formatting Mathematical Expressions	60
5	Additional Topics	61
5.1	Loading Numerical Files	61
5.2	Images	62
5.3	Animation	63
5.4	Random Number Generation	65
5.5	Sound Files	66
5.6	Linear Programming	67
6	Programming Style	69
6.1	Choosing Good Variable Names	69
6.2	Choosing Good Function Names	70
6.3	No “Magic Numbers”	70
6.4	Comments	70
6.5	Errors and Debugging	71
7	Further Reading	72
	Index	73

I Getting Started

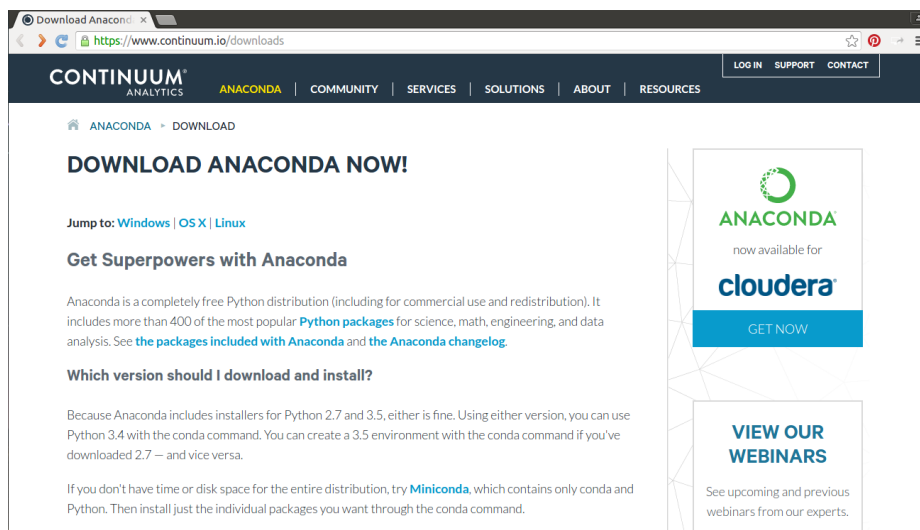
II Course Description

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, linear regression, initial value problems, dynamical systems, random number generation, 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.

1.2 Install Python

We will be using Python 3.5. 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, Matplotlib, Jupyter Notebook).



1.3 Weekly Reports

Reports will be submitted every week on UBlerns as Jupyter Notebook files containing text, code, and results. Files will be identified using your surname and the report number (e.g. “cunningham01.ipynb”). 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 be 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.
- A list of references to books, articles and websites consulted.
- Python code used to generate all figures and data in the report as code cells in the Jupyter Notebook.

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

1.3.1 Presenting and Interpreting Results

Every assignment in this class involves exploring a topic in science and mathematics, generating graphical results, and saying something about those results. 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 title for the graph.
- A number for the title, which can be used to refer to the graph in the report.

The minimum expected for assignments is that the main qualitative features of the results should be described and an attempt made at explanation. Explaining the quantitative features of the results is usually a more difficult and challenging task.

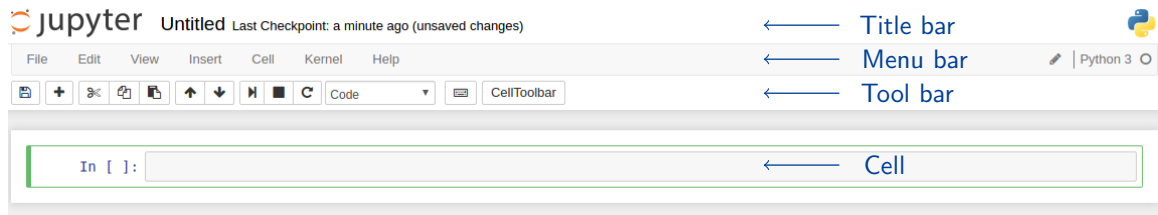
1.4 Jupyter Notebook

The development environment we will be using is the [Jupyter Notebook](#). This provides:

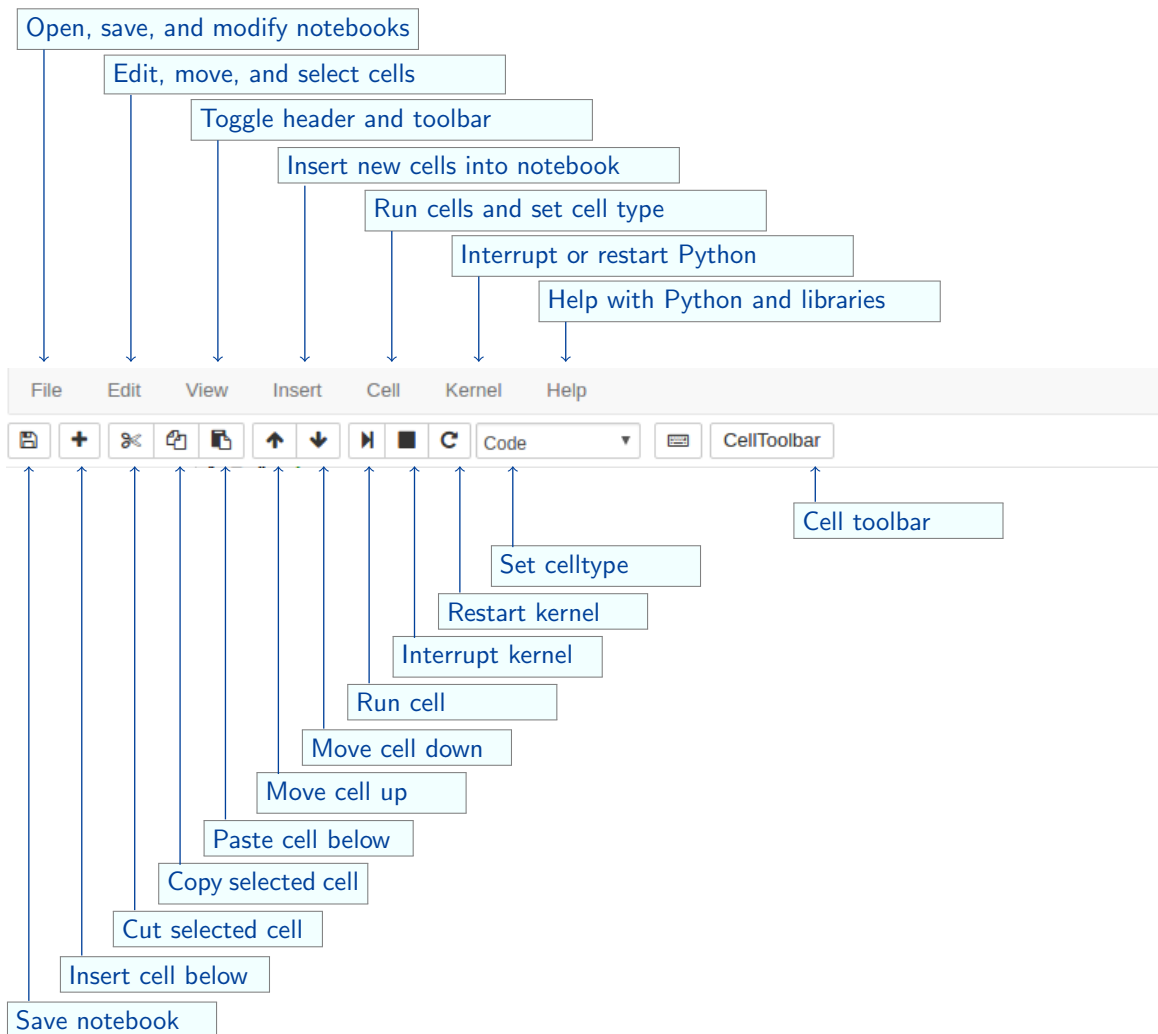
- An interactive environment for writing and running code.
- A way to integrate code, text and graphics in a single document.

A new Jupyter Notebook opens in a web browser, and contains:


- 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 can be of different types. The main types we will use are:

- **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 syntax.

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

1.4.1 Magics

Magics are instructions 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.
- `%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,

```
%%timeit x = range(10000)
max(x)
```

```
1000 loops, best of 3: 884 μs per
loop
```


The line “x = range(10000)” is run once but not timed. The “max(x)” line is timed.



Note that the `%%timeit` magic *must* be the first line in the code cell.

1.4.2 Markdown

Text can be added to Jupyter 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
### An h3 header	An h3 header
#### An h4 header	An h4 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
[UB link](https://www.buffalo.edu/)	UB link
![Python](PythonImage.jpg "Python")	
A horizontal line ***	A horizontal line <hr/>

1.4.3 L^AT_EX

Mathematical expressions in Markdown cells are specified using the typesetting language L^AT_EX. These expressions are identified using `$(formula)$` for an inline formula (displayed within a line of text), or `$$ (formula) $$` for a larger formula displayed on a separate line.

Superscripts	
<code>\$x^2\$</code>	x^2
Subscripts	
<code>\$x_1\$</code>	x_1
Fractions	
<code>\$\$\frac{1}{2}\$\$</code>	$\frac{1}{2}$
Greek Letters	
<code>\$\$\alpha, \beta, \gamma, \dots, \omega\$</code>	$\alpha, \beta, \gamma, \dots, \omega$
Series	
<code>\$\$\sum_{i=1}^n\$</code>	$\sum_{i=1}^n$
Integrals	
<code>\$\$\int_a^b\$</code>	\int_a^b
Square Roots	
<code>\$\$\sqrt{a + b}\$</code>	$\sqrt{a + b}$
Overline	
<code>\$\$\bar{x}\$</code>	\bar{x}
Brackets	
<code>\$\$\{1, 2, \dots, n\}\$</code>	$\{1, 2, \dots, n\}$
Matrices	
<code>\$\$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\$</code>	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

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 **. 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.

$3 + 2$ <hr/> 5	<i>Addition</i>
$3 - 2$ <hr/> 1	<i>Subtraction</i>
$3 * 2$ <hr/> 6	<i>Multiplication</i>
$3 / 2$ <hr/> 1.5	<i>Division</i>

<code>3**2</code>	<i>Exponentiation (not 3^2)</i>
9	

Some other useful operations are floor division (`//`), modulus (`%`), and absolute value (`abs`).

<code>3 // 2</code>	<i>Floor division returns the integral part of the quotient.</i>
1	
<code>12 % 5</code>	<i>Modulus returns the remainder.</i>
2	
<code>abs(-88)</code>	abs returns the absolute value.
88	

Python has a built-in complex number type, and knows the rules of complex arithmetic.

<code>1 + 2j</code>	<i>Generate a complex number (<i>j</i> is used instead of <i>i</i>).</i>
<code>(1+2j)</code>	
<code>complex(1, 2)</code>	<i>Another way to generate a complex number.</i>
<code>(1+2j)</code>	
<code>(1+2j).real</code>	real returns the real part of a complex number.
1.0	
<code>(1+2j).imag</code>	imag returns the imaginary part of a complex number.
2.0	
<code>abs(3+4j)</code>	abs returns the modulus when applied to a complex number.
5.0	
<code>z = 1 + 2j</code> <code>w = 3 - 1j</code>	<i>Note that a '1' is needed in front of the <i>j</i>.</i>

<code>z + w</code> ----- (4+1j)	<i>Complex addition.</i>
<code>z * w</code> ----- (5+5j)	<i>Complex multiplication.</i>

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.

<code>True or False</code> ----- <code>True</code>	<i>Logical disjunction</i>
<code>True and False</code> ----- <code>False</code>	<i>Logical conjunction</i>
<code>not True</code> ----- <code>False</code>	<i>Logical negation</i>
<code>True + 41</code> ----- 42	<i>True has the numerical value 1.</i>
<code>False * 41</code> ----- 0	<i>False has the numerical value 0.</i>

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.

<pre>print("abc")</pre> <hr/> <pre>abc</pre>	print outputs text to the screen (discarding the quotes).
<pre>"abc" + "def"</pre> <hr/> <pre>"abcdef"</pre>	Adding two strings makes a new string by concatenation.
<pre>"abc"*3</pre> <hr/> <pre>"abcabcabc"</pre>	Multiplying a string by an integer repeats the string.
<pre>print("I love 'MTH 337'!")</pre> <hr/> <pre>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.

<pre>string1 = "abc\ndef"</pre> <hr/> <pre>print(string1)</pre> <pre>abc</pre> <pre>def</pre>	The <code>"\n"</code> character specifies a newline.
<pre>string2 = "abc\tdef"</pre> <hr/> <pre>print(string2)</pre> <pre>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.

<pre>"abcde"[1]</pre> <hr/> <pre>"b"</pre>	Indexing returns the character at index 1 (indices start at 0, not 1).
--	--

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

2.4 Formatting Strings

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. “Format specifications” within replacement fields define how data is to be formatted, including the field width, padding and number of decimal places.

- Empty replacement fields `{}` are filled in order with the arguments given.
- Numbers inside replacement fields specify arguments by position, starting with zero.

<code>"{} {}".format("a", "b")</code> "a b"	<i>“Replacement fields” <code>{}</code> are filled in order by format.</i>
--	---

```
"1st: {0}, 2nd: {1}".format(3,4)
```

```
"1st: 3, 2nd: 4"
```

The arguments to **format** can also be identified by position, starting at 0.

Format specifications for the field width and padding are provided after a colon ':':

- A field width can be specified using `:n` where `n` is an integer specifying the number of characters in the field.
- If the data has less characters than the field width, the default is to insert extra spaces on the right i.e. to "pad" on the right.
- To pad on the left, use `:>n`. To explicitly pad on the right, use `:<n`. The `>` and `<` are like direction arrows "sending" the data to the indicated side of the field.
- Data can be centered in a replacement field using `:^n`.

```
print "{:5} {}".format("a", 2)
```

```
a 2
```

Using a field of width 5 with the default padding.

```
print "{:>5} {}".format("a", 2)
```

```
a 2
```

Note the left padding in this example.

```
print "{:^5} {}".format("a", 2)
```

```
a 2
```

Now centering 'a' in the field of width 5.

Format specifications for floats allow tables of numerical data to be neatly printed and aligned.

- Integers are referred to using `:d`.
- Floats are referred to using `:f`.
- An integer after a decimal point is used to indicate how many decimal places to display.
- Use `:n.mf` to indicate a replacement field of width `n` for a float printed to `m` decimal places.

```
print "{:5.2f}".format(pi)
```

```
3.14
```

Print pi in a field of width 5 to 2 decimal places

```
print "{:~10.4f}".format(pi)
3.1416
```

Padding can be combined with other options if the padding is specified first.

2.5 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 if the string represents a valid number.

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

<code>bool(1)</code> True	<i>Convert integer to boolean.</i>
<code>bool(42)</code> True	<i>Any nonzero value counts as True.</i>
<code>bool(0)</code> False	<i>Zero equates to False.</i>
<code>bool("")</code> False	<i>An empty string is also False.</i>
<code>int(2.99)</code> 2	<i>Convert float to integer (the decimal part is discarded).</i>
<code>int("22")</code> 22	<i>Convert string to int.</i>
<code>float("4.567")</code> 4.567	<i>Convert string to float.</i>
<code>complex("1+2j")</code> (1+2j)	<i>Convert string to complex.</i>

<code>float(10)</code> ----- 10.0	<i>Convert integer to float.</i>
<code>complex(10)</code> ----- (10+0j)	<i>Convert integer to complex number.</i>
<code>str(True)</code> ----- "True"	<i>Convert boolean to string.</i>
<code>str(1)</code> ----- "1"	<i>Convert integer 1 to string "1".</i>
<code>str(1.234)</code> ----- "1.234"	<i>Convert float to string.</i>

2.6 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.

<code>x = 5</code> <code>print(x)</code> ----- 5	<i>Assign <code>x</code> the value 5 (note that <code>"="</code> is used for assignment, not <code>"=="</code>).</i>
<code>y = x + 3</code> <code>print(y)</code> ----- 8	<i>Assign <code>y</code> the value of <code>x + 3</code>.</i>
<code>course = "MTH 337"</code> <code>print(course)</code> ----- MTH 337	<i><code>course</code> is a string (printed without quotes).</i>

<pre>a, b = 2, 3 print(a, b)</pre> <hr/> <pre>2 3</pre>	<p><i>Multiple variables can be assigned at the same time.</i></p>
<pre>a, b = b, a print(a, b)</pre> <hr/> <pre>3 2</pre>	<p><i>Values of a and b are swapped (the right hand side is evaluated before the assignment).</i></p>
<pre>z = 3 z += 2 print(z)</pre> <hr/> <pre>5</pre>	<p><i>Same as $z = z + 2$.</i></p>
<pre>z -= 1 print(z)</pre> <hr/> <pre>4</pre>	<p><i>Same as $z = z - 1$.</i></p>
<pre>z *= 3 print(z)</pre> <hr/> <pre>12</pre>	<p><i>Same as $z = z * 3$.</i></p>
<pre>z /= 2 print(z)</pre> <hr/> <pre>6</pre>	<p><i>Same as $z = z / 2$.</i></p>
<pre>z %= 5 print(z)</pre> <hr/> <pre>1</pre>	<p><i>Same as $z = z \% 5$.</i></p>

2.7 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.

<pre>from math import pi print(pi)</pre> <hr/> <p>3.14159265359</p>	<p><i>pi is now a variable name that we can use, but not the rest of the math module.</i></p>
<pre>from math import * print(e)</pre> <hr/> <p>2.71828182846</p>	<p><i>Everything in the math module is now available.</i></p>
<pre>import numpy print(numpy.arcsin(1))</pre> <hr/> <p>1.57079632679</p>	<p><i>Everything in numpy can be used, prefaced by "numpy".</i></p>
<pre>import numpy as np print(np.cos(0))</pre> <hr/> <p>1.0</p>	<p><i>Everything in numpy can be used, prefaced by the alias "np".</i></p>

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.

<pre>import math print(dir(math))</pre> <hr/> <pre>["__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", "isinfinite", "isnan", "ldexp", "lgamma", "log", "log10", "log1p", "modf", "pi", "pow", "radians", "sin", "sinh", "sqrt", "tan", "tanh", "trunc"]</pre>	<p><i>The math module contains all the standard mathematical functions.</i></p>
--	--

2.8 Lists

Lists 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.

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

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

<pre>primes = [2, 3, 5, 7, 11, 13, 17] primes[1]</pre> <hr/> <pre>3</pre>	<p><i>Access the element at index 1 (indexing starts with 0).</i></p>
<pre>primes[3:]</pre> <hr/> <pre>[7, 11, 13, 17]</pre>	<p><i>List slicing, start at position 3, through to the end.</i></p>

<pre>primes[:3] [2, 3, 5]</pre>	<p><i>List slicing, start at the beginning, end at position 2.</i></p>
<pre>primes[2:5] [5, 7, 11]</pre>	<p><i>List slicing, start at position 2, end at position 4.</i></p>
<pre>primes[::-1] [17, 13, 11, 7, 5, 3, 2]</pre>	<p><i>One way to reverse a list.</i></p>

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

<pre>mylist = ["a", "b"] mylist.append("c") print(mylist) ["a", "b", "c"]</pre>	<p>append adds an element to the end of a list.</p>
<pre>del(mylist[1]) print(mylist) ["a", "c"]</pre>	<p>del deletes an element from a list.</p>
<pre>mylist.insert(1, "d") print(mylist) ["a", "d", "c"]</pre>	<p>insert inserts an element at a given position.</p>
<pre>mylist[1] = "e" print(mylist) ["a", "e", "c"]</pre>	<p><i>List elements can be changed by assigning a new element at a given index.</i></p>

Lists can be sorted and reversed.

<pre>letters = ["a", "b", "c"] letters.reverse() print(letters) ["c", "b", "a"]</pre>	<p>reverse changes an existing list, reversing the order of elements.</p>
---	--

<pre>numbers = [2, 10, 3, 26, 5] print(sorted(numbers))</pre> <hr/> <pre>[2, 3, 5, 10, 26]</pre>	<p>sorted returns a sorted list, but does not modify the existing list.</p>
<pre>numbers.sort() print(numbers)</pre> <hr/> <pre>[2, 3, 5, 10, 26]</pre>	<p>sort sorts a list in place, modifying the existing list.</p>
<pre>sorted(numbers, reverse=True)</pre> <hr/> <pre>[26, 10, 5, 3, 2]</pre>	<p>The reverse keyword is used to sort in descending order.</p>

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

<pre>numbers = [2, 10, 3, 26, 5] print(min(numbers), max(numbers))</pre> <hr/> <pre>2 26</pre>	<p>min and max find the smallest and largest items.</p>
--	---

2.9 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.

<pre>tuple1 = ("a", "b", "c") print(tuple1)</pre> <hr/> <pre>("a", "b", "c")</pre>	<p>Create a tuple using standard parentheses.</p>
<pre>tuple1[2]</pre> <hr/> <pre>"c"</pre>	<p>Tuple elements can be indexed just like lists or strings.</p>

<pre>tuple1[1:] ("b", "c")</pre>	<p><i>Slicing works the same way for tuples as for lists or strings.</i></p>
----------------------------------	--

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

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

2.10 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).

<pre>myset = {1, 2, 3} print(myset) set([1, 2, 3])</pre>	<p><i>Sets are created using curly brackets.</i></p>
<pre>myset = set([1, 2, 3, 2]) print(myset) set([1, 2, 3])</pre>	<p><i>Creating a set from a list (note that duplicates are removed).</i></p>

<pre>print(set())</pre>	<i>set([]) creates an empty set.</i>
<pre>set([])</pre>	

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

<pre>set1 = {1, 2, 3} set2 = {3, 4, 5}</pre>	<i>Create 2 sets.</i>
<pre>1 in set1</pre> <pre>True</pre>	<i>in tests for set membership.</i>
<pre>set1 set2</pre> <pre>{1, 2, 3, 4, 5}</pre>	<i>Set union (the union operator can also be used).</i>
<pre>set1 & set2</pre> <pre>{3}</pre>	<i>Set intersection (can also use the intersection operator).</i>
<pre>set1 - set2</pre> <pre>{1, 2}</pre>	<i>Set difference (can also use the difference operator).</i>
<pre>set1 ^ set2</pre> <pre>{1, 2, 4, 5}</pre>	<i>Symmetric difference (can also use the symmetric_difference operator).</i>
<pre>set1 <= set2</pre> <pre>False</pre>	<i>Test if one set is a subset of another (can also use the is-subset operator).</i>

2.11 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.
- Order is *not* preserved in a dictionary, so printing a dictionary will not necessarily print items in the same order that they were added.

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



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.

2.12 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.

<code>5 == 5</code> True	<i>Check for equality.</i>
<code>5 != 5</code> False	<i>Check for inequality,</i>
<code>3 < 2</code> False	<i>Less than.</i>
<code>3 <= 3</code> True	<i>Less than or equals.</i>
<code>"a" < "b"</code> True	<i>Strings are compared by lexicographic (dictionary) order.</i>

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

2.13 If Statements

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

if statement
<pre>if <condition>: <code to execute when condition True> <following code></pre>

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*.

- The block of code to execute is finished by de-indenting back to the previous level.

```
from math import *
if pi > e:
    print("Pi is bigger than e!")
```

```
Pi is bigger than e!
```

*The block of code following the **if** statement only executes if the condition is met.*

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.

if-else statement

```
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.

```
x, y = 2**3, 3**2
if x < y:
    print("x < y")
else:
    print("x >= y")
```

```
x < y
```

*The block of code following the **else** statement executes if the condition is not 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.

elif statement

```
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>
```

The following example illustrates a series of conditions being tested.

```

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

Only the first two conditions are tested - the rest are skipped since the second condition is True.

2.14 Conditional Expressions

It often happens that we want to assign a variable name 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 the same thing in a single line using a *conditional expression*.

Conditional expression

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

An example is given below.

```

x = 22
parity = "odd" if x % 2 else "even"
print(x, "has", parity, "parity")

```

22 has even parity

Note that $x \% 2$ returns the remainder when x is divided by 2. Any nonzero value evaluates as True.

2.15 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.

for loop

```
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.

<pre>for i in [2, 4, 6]: print(i)</pre> <hr/> <pre>2 4 6</pre>	<p><i>Iterate over the elements of a list. The iteration variable <code>i</code> gets bound to each element in turn.</i></p>
<pre>for char in "abc": print(char)</pre> <hr/> <pre>a b c</pre>	<p><i>Iterate over the characters in a string. The iteration variable <code>char</code> gets bound to each character in turn.</i></p>
<pre>for i, char in enumerate("abc"): print(i, char)</pre> <hr/> <pre>0 a 1 b 2 c</pre>	<p>enumerate allows an iteration variable to be bound to the index of each item, as well as to the item itself.</p>

The **range** function generates integers in a given range. It is often used inside a **for** loop to iterate over some sequence of integers.

- The start, stop and step parameters to **range** are similar to those used to slice lists and strings.
- Integers are only generated by **range** as needed, rather than as a list.

<pre>for i in range(3): print(i)</pre> <hr/> <pre>0 1 2</pre>	<p>range(<i>n</i>) generates <i>n</i> consecutive integers, starting at 0 and ending at <i>n</i> - 1.</p>
<pre>teens = range(13, 20) print(list(teens))</pre> <hr/> <pre>[13, 14, 15, 16, 17, 18, 19]</pre>	<p>range(start, stop) generates consecutive integers, from start to stop - 1.</p>
<pre>evens = range(0, 9, 2) print(list(evens))</pre> <hr/> <pre>[0, 2, 4, 6, 8]</pre>	<p>The third step argument to range specifies the increment from one integer to the next.</p>
<pre>for i in range(5, 0, -1): print(i)</pre> <hr/> <pre>5 4 3 2 1</pre>	<p>range can also count backwards using a negative step size.</p>
<pre>total = 0 for i in range(1, 6): total += i print(total)</pre> <hr/> <pre>15</pre>	<p>Sum the numbers from 1 to 5.</p>
<pre>sum(range(1, 6))</pre> <hr/> <pre>15</pre>	<p>Another (simpler) way to sum the numbers in a given range.</p>

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

<pre>mydict = {"x":1, "y":2, "z":3} for key in mydict: print(key)</pre> <hr/> <pre>y x z</pre>	<p>Iteration over a dictionary binds to the key (note that order is not preserved in a dictionary).</p>
--	---

<pre>for value in mydict.values(): print(value)</pre> <hr/> <pre>2 1 3</pre>	<p>Use values to iterate over the dictionary values rather than the keys.</p>
<pre>for key, value in mydict.items(): print(key, value)</pre> <hr/> <pre>y 2 x 1 z 3</pre>	<p>Use items to iterate over the dictionary keys and values together.</p>

We can also iterate in parallel over multiple lists of equal length by using **zip**. This generates a sequence of tuples, with one element of each tuple drawn from each list.

<pre>courses = [141, 142, 337] ranks = ["good", "better", "best!"] zipped = zip(courses, ranks) print(list(zipped))</pre> <hr/> <pre>[(141, "good"), (142, "better"), (337, "best")]</pre>	<p>zip(<i>courses</i>, <i>ranks</i>) generates a sequence of tuples. Each tuple contains one course and one rank, with the tuples in the same order as the list elements.</p>
<pre>for course, rank in zipped: print(course, rank)</pre> <hr/> <pre>141 good 142 better 337 best!</pre>	<p>Multiple iteration variables can be bound at each iteration of a loop.</p>

2.16 While Loops

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

<p>while loop</p> <pre>while <condition>: <code to execute repeatedly> <following code></pre>
--

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

```
i = 3
while i > 0:
    print(i)
    i -= 1
```

```
3
2
1
```

The variable `i` is printed while it remains greater than zero. The code inside the loop must change the value of `i` to ensure that the loop eventually terminates.

2.17 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.

```
vowels = "aeiou"
for char in "bewgfiagf":
    if char in vowels:
        print("First vowel is", char)
        break
```

```
First vowel is e
```

*The **for** loop is terminated by **break** once the first vowel is found.*

```
total = 0
for char in "bewgfiagf":
    if char in vowels:
        continue
    total += 1
print(total, "consonants found")
```

```
6 consonants found
```

*Skip over the vowels using **continue**, and just count the consonants.*

2.18 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 list comprehensions, the syntax is:

List comprehension

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


The code in `<expression>` is evaluated for each item in the `<container>`, and the result becomes an element of the new list. The `<condition>` does not have to be present but, if it is, only elements which satisfy the condition become incorporated into the new list.

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

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

Dictionary comprehension
<pre>{key:value for <variables> in <container> if <condition>}</pre>

The following example creates a dictionary from the elements of a list.

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

2.19 Generator Expressions

We often want to iterate over the elements of some sequence, but don't need to save the sequence. In these cases, a list comprehension is an unnecessary overhead since the list is created and saved in memory for a single use. A more efficient alternative is to use a "generator expression".

- Generator expressions are used like lists inside **for** loops.
- They don't create the entire sequence up front, so memory doesn't need to be allocated for all the elements.
- Instead, the current element is saved, and the next one is produced only when requested by the loop.

Generator expressions have almost the same syntax as a list comprehension, but use parentheses instead of square brackets.

Generator expression
<pre>(<expression> for <variables> in <container> if <condition>)</pre>

An example is given below.

```
squares = (i**2 for i in range(1, 4))
for s in squares:
    print(s)
```

```
1
4
9
```

The elements of squares are generated as needed, not saved up front.

2.20 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 statement

```
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.

```
def square(x):
    return x**2
print(square(3))
```

```
9
```

*The function exits as soon as the **return** statement is called*

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

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.

<pre>def close_enough(x, y, tolerance=.1) return abs(x - y) <= tolerance</pre> <hr/>	<p><i>The tolerance argument is 0.1 by default.</i></p>
<pre>close_enough(1, 1.05)</pre> <hr/> <p>True</p>	<p><i>The default tolerance of 0.1 is used in this case.</i></p>
<pre>close_enough(1, 1.05, tolerance=.01)</pre> <hr/> <p>False</p>	<p><i>The default tolerance is overridden by the value of 0.01.</i></p>

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 statement

```
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.

```
ages = [21, 19, 98]
names = ["Bruce", "Sheila", "Adam"]
data = zip(ages, names)
sorted(data, key=lambda x : x[0])

[(19, "Sheila"), (21, "Bruce"), (98, "Adam")]
```

The key 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.

2.21 Error Handling with Try-Except

If an invalid operation is attempted when running code then an error is usually generated. Examples include dividing by zero, indexing past the end of a sequence, creating a floating point number that is too large, or adding two arrays of different sizes. In these circumstances, the code typically breaks at the point of the invalid operation.

A **try-except** statement can be used to handle errors more deliberately than having the code break. This allows us to first try to execute some code that we're not sure will work, and then execute some other code if it doesn't.

try-except statement

```
try:
    <code to try to execute>
except:
    <code to execute if an error is generated>
```

Errors typically have a type associated with them, which specifies the kind of error that has occurred. Examples include `ZeroDivisionError`, `IndexError`, `OverflowError`, and `ValueError`. The "except" part of a try-except statement can be specialized to handle these different kinds of error.

try-except statement with named error types

```
try:
    <code to try to execute>
except ErrorType:
    <code to execute if an error is generated>
```

The following example shows how to gracefully handle an error resulting from division by zero.

```
for i in range(-2,3):
    print(10/i)

-5.0
-10.0
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

The number 10 is being divided by a sequence of integers, one of which happens to be zero. Without error handling, the code breaks when the zero is encountered and a "ZeroDivisionError" is raised.

```
for i in range(-2,3):
    try:
        print(10/i)
    except ZeroDivisionError:
        print("Can't divide by zero!")

-5.0
-10.0
Can't divide by zero!
10.0
5.0
```

*Using a **try-except** statement to handle the ZeroDivisionError allows the loop to run to completion without breaking.*

2.22 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 need 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 "firstnames.txt" has been created in the directory that Python was started in. This file contains the three lines:

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

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

```
data = ["Hofstadter", "?", "Cooper"]
output_file = open("names.txt", "w")
for name in data:
    output_file.write(name + "\n")
output_file.close()
```

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.

```
f = open("names.txt")
last_names = f.read()
f.close()
print(last_names)
```

```
Hofstadter
?
Cooper
```

Check that the "names.txt" file has been written correctly.

2.23 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.

```
# This is a single-line comment
```

No need to comment a comment.

```
"""This is a documentation string."""
```

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 a one-dimensional 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 we use in this class.

<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 functions are illustrated below.

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

<pre>x = linspace(0, 1, 6) print(x)</pre> <hr/> <pre>[0. 0.2 0.4 0.6 0.8 1.]</pre>	<p>linspace(<i>start</i>, <i>stop</i>, <i>num</i>) returns <i>num</i> equally spaced points, including endpoints.</p>
<pre>x = arange(5) print(x)</pre> <hr/> <pre>[0 1 2 3 4]</pre>	<p>arange returns an array of evenly spaced values within a given interval.</p>

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

<pre>x = empty((3, 2)) print(x)</pre> <hr/> <pre>[[6.93946206e-310 6.93946206e-310] [6.36598737e-314 6.36598737e-314] [6.36598737e-314 0.00000000e+000]]</pre>	<p>empty(<i>shape</i>) returns an array of shape <i>shape</i>, initially filled with garbage.</p>
<pre>x = zeros((2, 3)) print(x)</pre> <hr/> <pre>[[0. 0. 0.] [0. 0. 0.]]</pre>	<p>zeros(<i>shape</i>) returns an array of shape <i>shape</i> filled with zeros - note the default type is float.</p>
<pre>x = ones((2, 3), dtype=int) print(x)</pre> <hr/> <pre>[[1 1 1] [1 1 1]]</pre>	<p>ones(<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 int.</p>

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

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

```
y = zeros_like(x)
print(y)
```

```
[ 0.  0.  0.]
```

y has the same shape as x, but is initialized with zeros.

```
y = ones_like(x)
print(y)
```

```
[ 1.  1.  1.]
```

y has the same shape as x, but is initialized with ones.

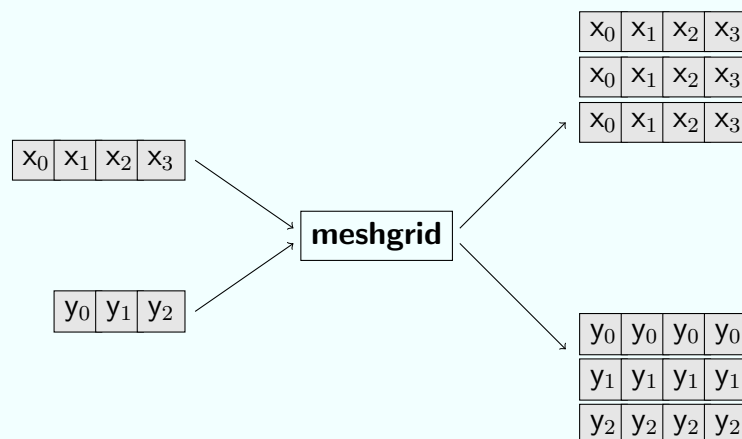
```
y = copy(x)
print(y)
```

```
[0.  1.  2.]
```

y is a copy of x - changing y will not change x.

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.

Meshgrid



An example is shown below.

```
from numpy import *
x = arange(4)
y = arange(3)
X, Y = meshgrid(x, y)
```

meshgrid creates 2D *x*- and *y*- coordinate arrays from 1D *x*- and *y*- coordinate arrays.

<pre>print(X) [[0 1 2 3] [0 1 2 3] [0 1 2 3]]</pre>	<p><i>X is a 2D array containing just the x-coordinates of points in the xy plane.</i></p>
<pre>print(Y) [[0 0 0 0] [1 1 1 1] [2 2 2 2]]</pre>	<p><i>Y is a 2D array containing just the y-coordinates of points in the xy plane.</i></p>

The function **meshgrid** is often used when we want to apply a function of two variables to points in the x-y plane. The following example uses the arrays *X* and *Y* above.

<pre>def distance(x, y): return round(sqrt(x**2 + y**2), 3) print(distance(X, Y))</pre> <pre>[[0. 1. 2. 3.] [1. 1.414 2.236 3.162] [2. 2.236 2.828 3.606]]</pre>	<p><i>The function distance finds the distance of a point (x, y) from the origin, rounded to 3 decimal places by the round function.</i></p>
--	--

3.2 Array Properties

The properties of an array can be accessed as follows.

<pre>x = arange(6) type(x) numpy.ndarray</pre>	<p><i>x is of type <code>numpy.ndarray</code>.</i></p>
<pre>x.dtype dtype("int64")</pre>	<p><i>dtype returns the element type - a 64-bit integer.</i></p>
<pre>x.shape (6,)</pre>	<p><i>x is a 1-dimensional array with 6 elements in the first axis.</i></p>

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**.

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

3.3 Array Operations

Array arithmetic is done on an element-by-element basis. Operations are applied to every element of an array, with each result becoming an element in a new array.

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

```
print(x**y)
```

```
[ 0  1 32  3]
```

Exponentiation is done using corresponding elements of the arrays x and y .

Comparison operators and other Boolean expressions are also applied on an element-by-element basis. The Boolean expression is applied to every element of the array, with each result becoming an element in a new boolean array.

```
x = arange(5)
```

```
print(x)
```

```
print(x % 2 == 0)
```

```
[0 1 2 3 4]
```

```
[ True False  True False  True]
```

The Boolean expression is evaluated for each element separately, resulting in an array of booleans.

```
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]
```

The comparison is done on an elementwise basis between elements of arrays x and y . The result is 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.

```
x = arange(3)
```

```
print(x)
```

```
[0 1 2]
```

Create an array.

```
print(sin(x))
```

```
[ 0.  0.84147098  0.90929743]
```

***sin** is applied to each element, to create a new array.*

```
print(exp(x))
```

```
[ 1.  2.71828183  7.3890561 ]
```

***exp** is the exponential operator.*

```
x = random.randint(5, size=(2,3))
```

```
print(x)
```

```
[[1 4 3]
```

```
[2 2 3]]
```

***random.randint** returns an array of a given size filled with randomly selected integers from a given range.*

<pre>print(x.min(), x.max())</pre> <hr/> <pre>1 4</pre>	<p>min and max calculate the minimum and maximum values across the entire array.</p>
<pre>print(x.min(axis=0)) print(x.min(axis=1))</pre> <hr/> <pre>[1 2 3] [1 2]</pre>	<p>The axis argument finds each minimum along a given axis. The resulting array is the shape of the original array, but with the given axis removed.</p>
<pre>print(x.sum())</pre> <hr/> <pre>15</pre>	<p>sum sums all the elements of an array.</p>
<pre>print(x.sum(axis=0))</pre> <hr/> <pre>[3 6 6]</pre>	<p>Providing the axis argument sums along the given axis.</p>

3.4 Array Indexing and Slicing

Arrays can be indexed and sliced using square brackets in the same way as lists.

- An index or `start:stop:step` filter needs to be provided for *each* axis, separated by commas.
- Indexing is zero-based, as it is with lists.
- Slicing an array produces a view of the original array, not a copy. Modifying this view will change the original array.

<pre>from numpy import * x = arange(20).reshape((4, 5)) print(x)</pre> <hr/> <pre>[[0 1 2 3 4] [5 6 7 8 9] [10 11 12 13 14] [15 16 17 18 19]]</pre>	<p>reshape provides a fast way to create a 2D array from a 1D array.</p>
<pre>print(x[1,2])</pre> <hr/> <pre>7</pre>	<p>Indexing is done into each axis in order - row 1, column 2.</p>
<pre>print(x[1,:])</pre> <hr/> <pre>[5 6 7 8 9]</pre>	<p>Slicing can be used to select every element of the first axis.</p>

<pre>print(x[:,1])</pre> <hr/> <pre>[1 6 11 16]</pre>	<p><i>Slicing can also be used to select the first element of every axis.</i></p>
<pre>print(x[1:3, 1:4])</pre> <hr/> <pre>[[6 7 8] [11 12 13]]</pre>	<p><i>Slice rows 1 and 2 using 1:3, then slice columns 1, 2 and 3 using 1:4.</i></p>

3.5 Indexing with Integer Arrays

Although slicing can be used to return a subset of elements from an array, its use is restricted to either selecting consecutive elements, or to selecting elements that are separated by the same fixed amount. We sometimes want to select instead some *arbitrary* subset of an array, possibly even selecting the same element more than once. Integer array indexing provides a way to do this.

- The index is itself a NumPy array of integers.
- Each integer in the index array selects a corresponding element from the target array.
- The result is an array of the same shape as the indexing array.

<pre>x = arange(9)**2</pre> <pre>print(x)</pre> <hr/> <pre>[0 1 4 9 16 25 36 49 64]</pre>	<p><i>First create the array using arange, then square each element.</i></p>
<pre>index = array([1, 3])</pre> <pre>print(x[index])</pre> <hr/> <pre>[1 9]</pre>	<p><i>An array is returned containing elements from the first array, selected according to the integers in the second array.</i></p>
<pre>index = array([[2, 3], [7, 2]])</pre> <pre>print(index)</pre> <hr/> <pre>[[2 3] [7 2]]</pre>	<p><i>The indexing array contains integers that are used to index into the target array. Note that the same elements (2, in this case) can be selected more than once.</i></p>
<pre>print(x[index])</pre> <hr/> <pre>[[4 9] [49 4]]</pre>	<p><i>When indexing using an integer array, the returned array has the same shape as the indexing array.</i></p>

3.6 Indexing with Boolean Arrays

We can also index using arrays of booleans. In this case, the indexing array acts like a *mask*, filtering out only the elements in the target array that correspond to `True` values in the indexing array.

```
x = arange(5)
mask = array([True, True, False, False, True])
print(x)
print(x[mask])
```

```
[0 1 2 3 4]
[0 1 4]
```

Only True elements in the mask are selected.

A common use of this technique is to select 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.

```
x = arange(20)
index3 = (x % 3 == 0)
index5 = (x % 5 == 0)
```

index3 and index5 are boolean arrays containing True elements for the integers that are divisible by 3 and 5 respectively.

```
print(x[index3])
[ 0  3  6  9 12 15 18]
```

Select just the elements of x that are divisible by 3.

```
print(x[index5])
[ 0  5 10 15]
```

Select just the elements of x that are divisible by 5.

```
print(x[logical_or(index3, index5)])
[ 0  3  5  6  9 10 12 15 18]
```

*The function **logical_or** performs an elementwise "or". The result is the integers divisible by either 3 or 5.*



The logical operators **not**, **or** and **and** do not get applied elementwise when applied to NumPy arrays. The functions **logical_not**, **logical_or** and **logical_and** need to be used instead.

4 Matplotlib

Matplotlib is a 2D plotting library for Python. It can be used to generate graphs, histograms, bar charts, contour plots, polar 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:

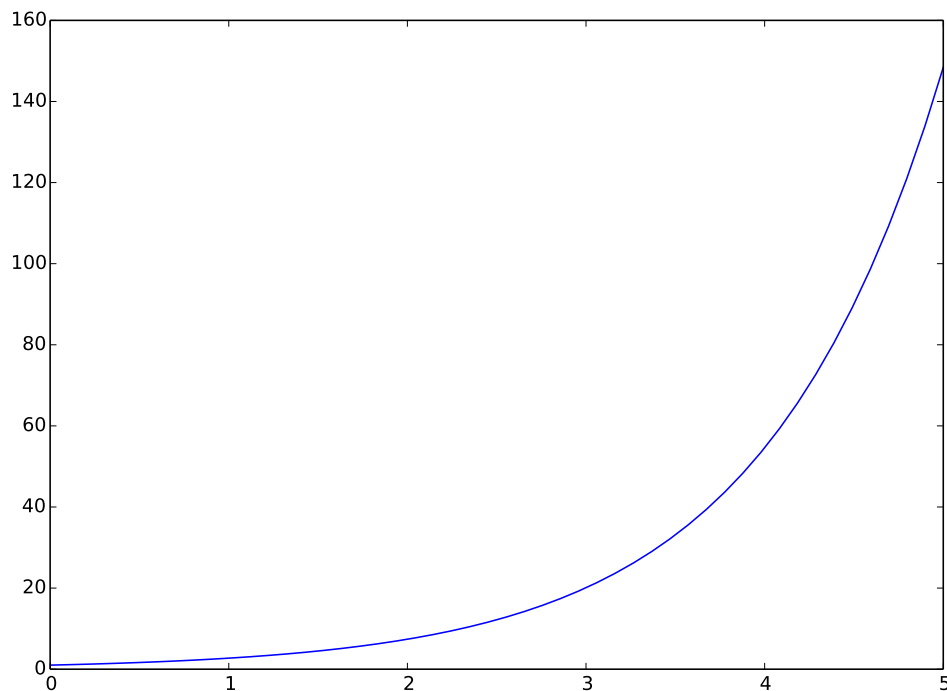
- The `x` parameter contains the x-coordinates of the points to plot, and the `y` parameter contains the y-coordinates.
- We need to import the required NumPy and Pylab functions for array manipulation and plotting. If using Jupyter Notebook, this can also be done using the IPython magic `%pylab`. In 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 $[0, 5]$.

Plotting an Exponential Function

```
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:

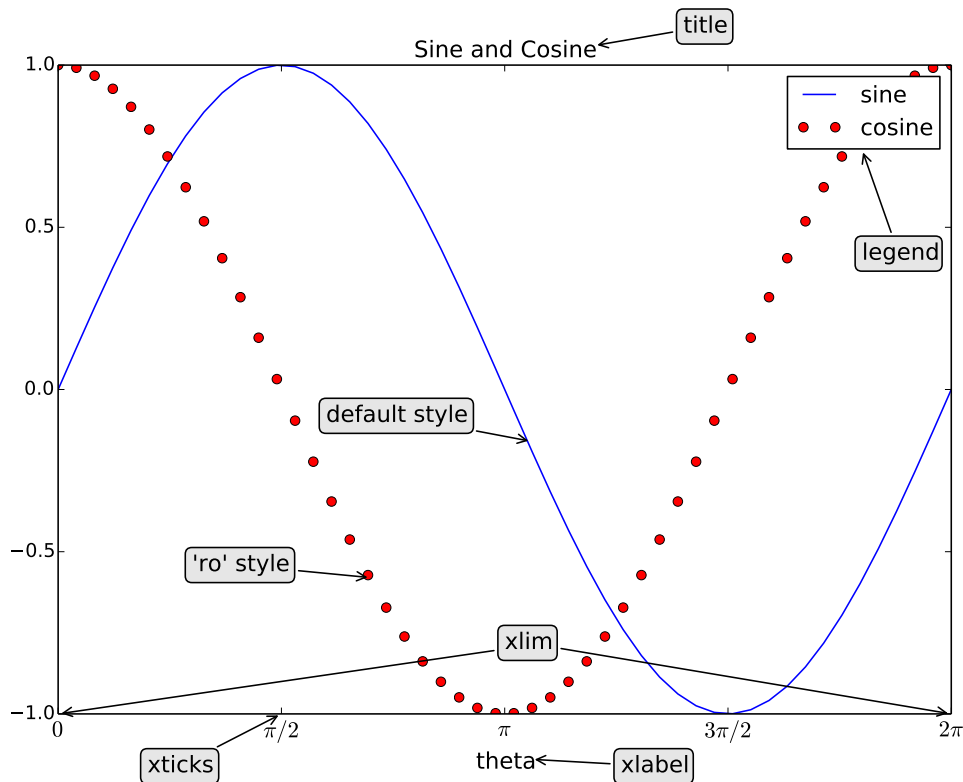
- The third argument to plot can be used to set colors, line types and marker types.
- Plot can be called multiple times, followed by a single call to show.

Plotting Two Graphs on a Single Figure

```

from numpy import *           # Imports linspace, sin, cos
import pylab as pl          # Import plotting functions
x = linspace(0, 2*pi, 50)   # Plot 50 points on the x-axis
pl.figure(figsize=(10,7))   # Set the size of the figure
pl.plot(x, sin(x), label='sine') # Default style is a blue line
pl.plot(x, cos(x), 'ro', label='cosine') # Use 'ro' for red circles
pl.xlabel('theta')          # Label the x-axis
pl.xlim(0, 2*pi)           # Limit x-axis to this range
ticks = [i*pi/2 for i in range(5)] # Locations of ticks on x-axis
labels = [r'$0$', r'$\pi/2$', r'$\pi$', # Labels for the x-axis ticks
          r'$3\pi/2$', r'$2\pi$'] # - these are LaTeX strings
pl.xticks(ticks, labels, size='large') # Add the x-ticks and labels
pl.title('Sine and Cosine') # Add a title
pl.legend()                 # Legend uses the plot labels
pl.show()                   # Finally, show the figure

```



4.3 Bar Plots

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

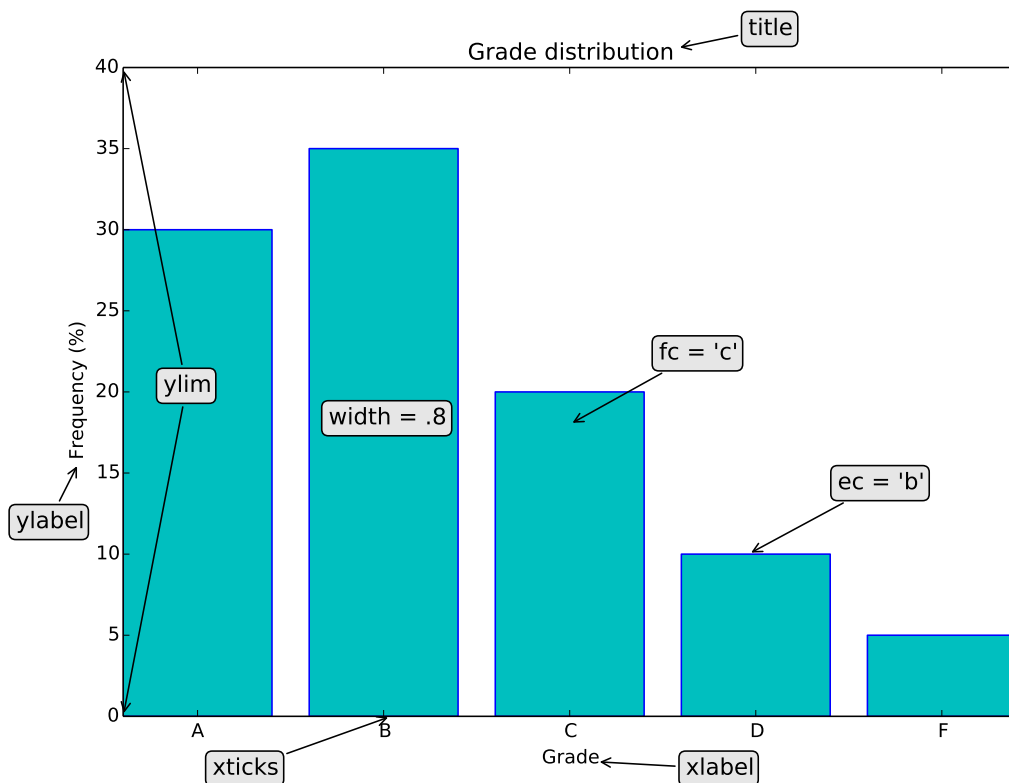
- Bars are described by their height, width, and position of the left and bottom edges.
- The `width` argument can be used to make bars thinner or thicker.
- The face color and edge color of the bars can be specified independently.

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.

Bar Plot

```
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 Polar Plots

The function **polar** is used to create polar plots. These plot radius against angle in polar coordinates.

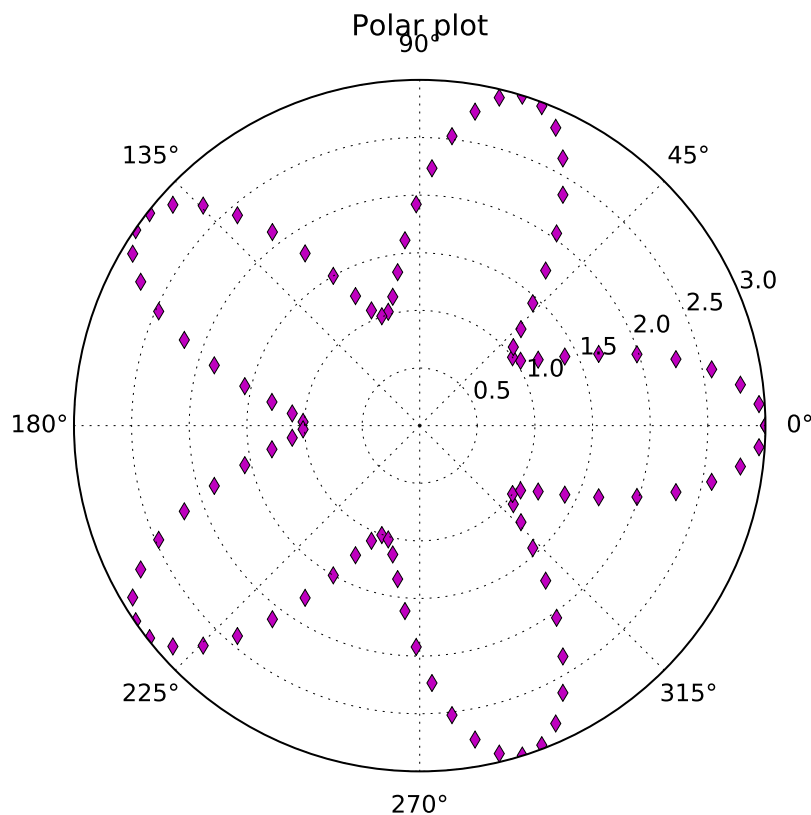
- The first argument to **polar** is an array of angles, and the second argument the corresponding radii.
- Colors, line types and marker types are specified in the same way as **plot**.
- **polar** can be called multiple times, followed by a single call to **show**.

The following example shows a polar plot with the marker style to "d" (diamond) and the color set to "m" (magenta).

Plotting in Polar Coordinates

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

theta = linspace(0, 2*pi, 100) # Create array of equally spaced values
r = 2 + cos(5*theta)           # Generate radius as a function of angle
pl.polar(theta, r, marker='d', ls='None', color='m')
pl.title('Polar plot')        # Add the title
pl.show()                     # Finally, show the figure
```



4.5 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.

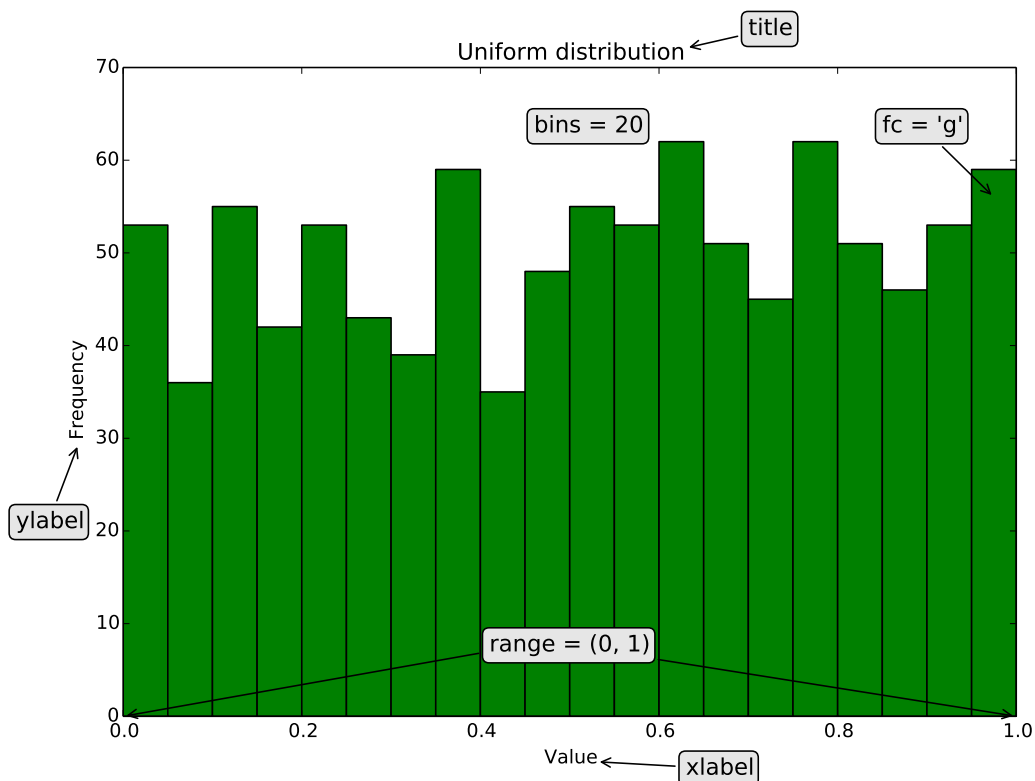
- Each bin covers a range of values, with the height of each bin indicating the number of points falling in that range.
- The first argument is an array or sequence of arrays.
- The *bins* argument specifies the number of bins to use.
- The *range* argument specifies the range of values to include.

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

Plotting a Histogram

```
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
```



4.6 Pie Charts

The function **pie** is used to create pie charts. These are a type of graph in which a circle is divided into wedges that each represent a proportion of the whole.

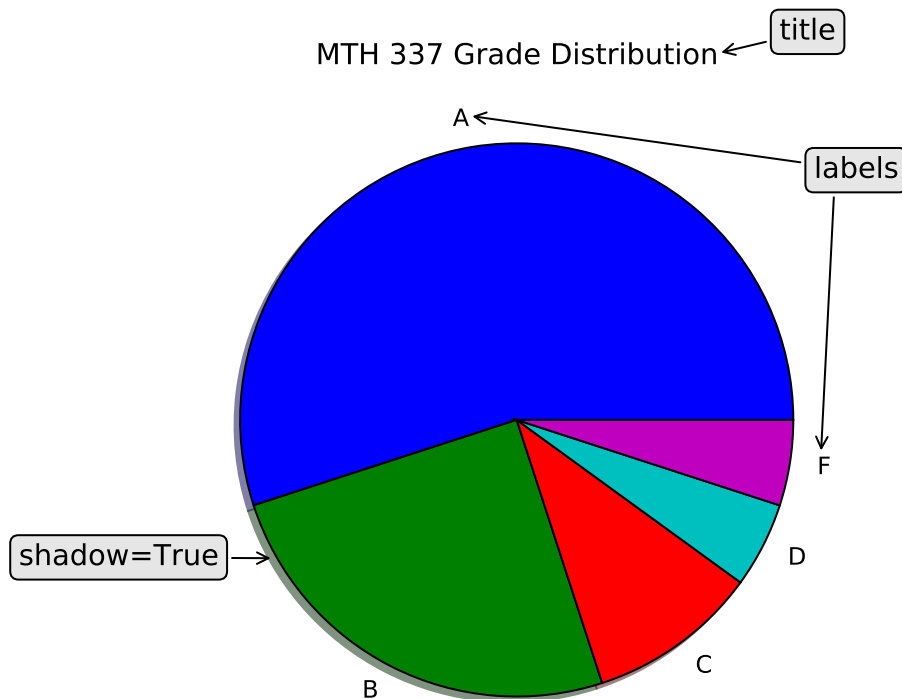
- The first argument to **pie** is a sequence of values used for the wedge sizes.
- The *labels* argument is a sequence of strings providing the labels for each wedge.
- The *shadow* argument is a boolean specifying whether to draw a shadow beneath the pie.

The following example shows a pie chart with *shadow* set to `True` .

Plotting a Pie Chart

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

percentages = [55, 25, 10, 5, 5] # Wedge sizes
labels = ['A', 'B', 'C', 'D', 'F'] # Sequence of labels for the wedges
pl.axes(aspect=1)           # Aspect ration = 1 for a true circle
pl.pie(percentages, labels=labels, shadow=True)
pl.title('MTH 337 Grade Distribution') # Add a title
pl.show()                   # Finally, show the figure
```



4.7 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.

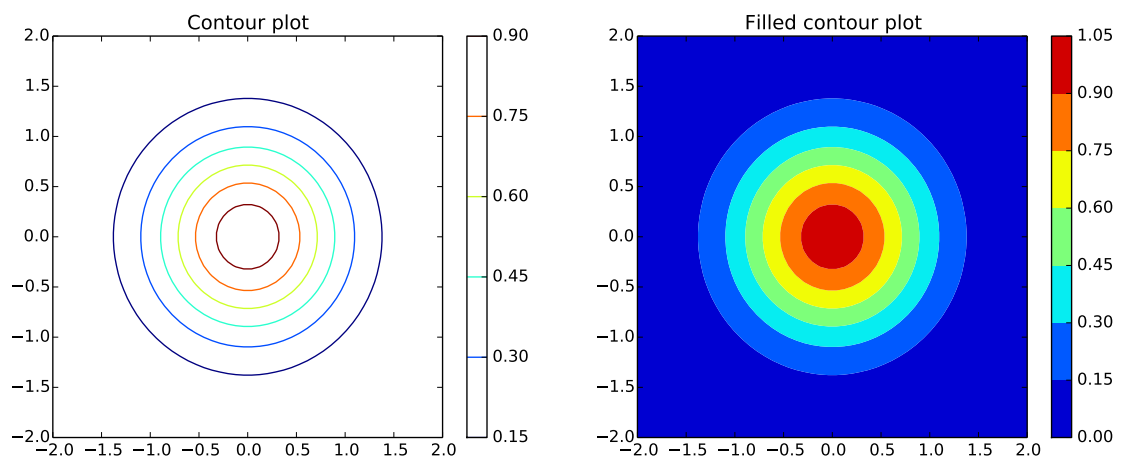
- The first two arguments are one dimensional arrays representing the x- and y-coordinates of the points to plot.
- The third coordinate is a two dimensional array representing the z-coordinates.
- Contour levels are automatically set, although they can be customized.
- A colorbar can be added to display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

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

Filled and Unfilled Contour Plots

```
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
```



4.8 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.

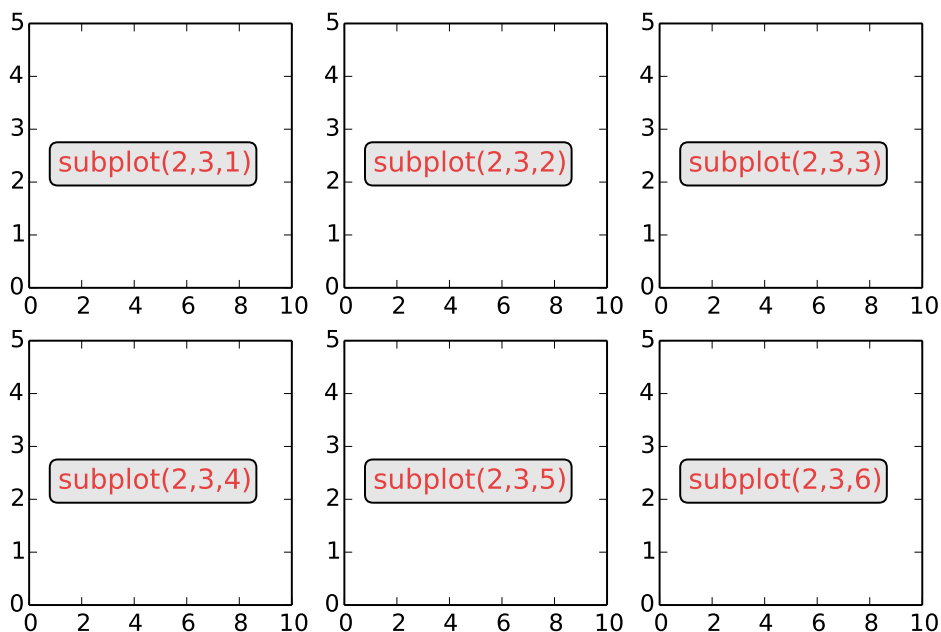
- Calls to subplot specify the number of rows, number of columns, and subplot number.
- Subplots are numbered from left to right, row by row, starting with 1 in the top left.
- All plotting is done in the location specified in the most recent call to subplot.
- If there are less than 10 rows, columns 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.

Displaying Multiple Plots with **subplot**

```
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
```



4.9 Formatting Text

The function `text` is used to add a text string to a plot at a given position.

- The first three positional arguments specify the x-position, y-position, and text string.
- The `fontsize` argument specifies the size of the font to use.
- The `fontstyle` argument specifies the style of font to use ('normal', 'italic' etc).
- The `fontweight` argument specifies how heavy the font should be ('normal', 'bold').
- The `family` argument specifies the font family to use ('serif', 'sans-serif' etc).
- The `color` argument specifies the color of the text.

These options can be combined together (for example, to specify text that is bold, red, italic and 14-point). The example below illustrates the use of these options.

Formatting Text

```
sizes = [10,12,14,16,18]
for sizepos, size in enumerate(sizes):
    text(0, sizepos, "Font size = {}".format(size), fontsize=size)

styles = ['normal', 'italic', 'oblique']
for stylepos, style in enumerate(styles):
    text(1, stylepos, "Style = {}".format(style), fontstyle=style)

families = ['serif', 'sans-serif', 'monospace']
for familypos, family in enumerate(families):
    text(2, familypos, "Family = {}".format(family), family=family)

weights = ['normal', 'bold']
for weightpos, weight in enumerate(weights):
    text(3, weightpos, "Weight = {}".format(weight), fontweight=weight)

colors = ['r', 'g', 'b', 'y', 'c']
for colorpos, color in enumerate(colors):
    text(4, colorpos, "Color = {}".format(color), color=color)
```

Font size = 18

Color = c

Font size = 16

Color = y

Font size = 14 *Style = oblique* Family = monospace

Color = b

Font size = 12 *Style = italic* Family = sans-serif **Weight = bold**

Color = g

Font size = 10 Style = normal Family = serif Weight = normal

Color = r

4.10 Formatting Mathematical Expressions

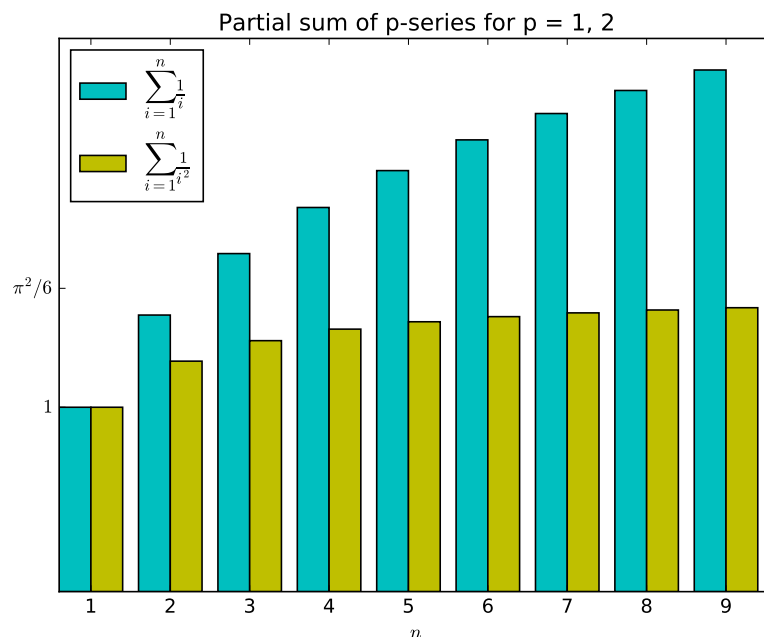
\LaTeX provides a way to format mathematical expressions in Matplotlib graphs in a similar way to Jupyter Notebook Markdown cells.

- Mathematical expressions are identified using `r"$\langle formula \rangle$"`.
- The syntax for $\langle formula \rangle$ is the same as that described in section 1.4.3 on \LaTeX .
- These expressions can be used anywhere a string is used, such as titles, axis and tick labels, and legends.

The example below illustrates several examples of mathematical expressions using \LaTeX .

Formatting Mathematical Expressions with \LaTeX

```
x = arange(1,10)
y1 = cumsum(1/x) # cumsum calculates the cumulative sum
y2 = cumsum(1/(x**2))
width = 0.4
bar(x, y1, width=width, fc='c', label=r'$\sum_{i=1}^n \frac{1}{i}$')
bar(x+width, y2, width=width, fc='y', label=r'$\sum_{i=1}^n \frac{1}{i^2}$')
ticks = x + width # Shift the x-ticks to center on the bars
xlabels = [str(val) for val in x] # Labels must be strings
xticks(ticks, xlabels)
ticks = [1, pi**2/6]
ylabels = [r'$1$', r'$\pi^2/6$'] # Note that \pi renders as a symbol
yticks(ticks, ylabels)
xlabel(r'$n$')
legend(loc='upper left')
title('Partial sum of p-series for p = 1, 2')
```



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

```
from numpy import loadtxt
```

*Import the **loadtxt** function.*

```
data = loadtxt("weather.dat", dtype=int,  
              skiprows=1)  
print(data)
```

```
[[ 1 77 56]  
 [ 2 79 62]]
```

*Load the "weather.dat" file, skipping the first row, and creating a 2×3 array of integers. For floats, the *dtype* argument would not be used.*

5.2 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.

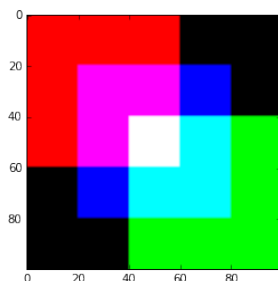
`imshow` Display an image array.

The following example creates an image as a 3D NumPy array of floats. The red, green and blue color components of the image are then set directly using array slicing.

Image Files: Creating a NumPy Image Array

```
img = zeros((100, 100, 3)) # Create an image array of 100 rows and columns.
img[:60,:60,0] = 1. # Set the top-left corner to red.
img[40:,40:,1] = 1. # Set the lower-right corner to green.
img[20:80,20:80,2] = 1. # Set the center square to blue.

imsave("squares.png", img) # Save the img array to the "squares.png" file
img2 = imread("squares.png") # Read the file back to the img2 array
imshow(img2) # Display the image
```



5.3 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 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 Jupyter 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.

Animation: A Point Circling the Origin

```

%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

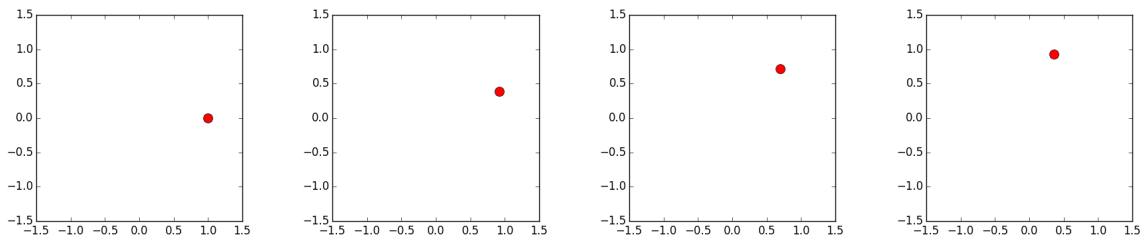
# Initialization function. This is called once to plot the background.
def init():
    point.set_data([], [])
    return point,                          # Return a tuple of the modified objects

# 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)

```

Some frames from this animation are shown below.



Note that in Jupyter Notebook the IPython magic we need to use is `%pylab` rather than `%pylab inline`. Inline graphs in Jupyter 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.

5.4 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.
- `random.choice` Samples are drawn from a given list or 1D array.

The following examples illustrate the use of these functions.

<pre>from numpy import * print(random.rand())</pre> <hr/> <pre>0.723812203628</pre>	<p>Use random.rand to generate a single number uniformly drawn from the interval $[0, 1)$.</p>
<pre>print(random.rand(3))</pre> <hr/> <pre>[0.74654564 0.58764797 0.15557362]</pre>	<p>Use random.rand to generate an array of 3 random numbers drawn from $[0, 1)$.</p>
<pre>print(random.rand(2, 3))</pre> <hr/> <pre>[[0.65382707 0.71701863 0.5738609] [0.22064692 0.57487732 0.5710538]]</pre>	<p>Use random.rand to generate a 2×3 array of random numbers drawn from $[0, 1)$.</p>
<pre>print(random.randint(7))</pre> <hr/> <pre>3</pre>	<p>Use random.randint to generate an integer drawn from $\{0, \dots, 6\}$.</p>
<pre>print(random.randint(5,9,size=(2,4)))</pre> <hr/> <pre>[[5 5 5 8] [7 8 7 6]]</pre>	<p>Use random.randint to generate a 2×4 array of integers drawn from $\{5, 6, 7, 8\}$.</p>
<pre>print(random.randn(3))</pre> <hr/> <pre>[0.47481788 -0.7690172 0.42338774]</pre>	<p>Use random.randn to generate an array of samples drawn from the “standard” normal distribution.</p>
<pre>print(random.normal(100, 15))</pre> <hr/> <pre>111.676554337</pre>	<p>Use random.normal to generate a sample drawn from a normal distribution with $\mu = 100$, $\sigma = 15$.</p>

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 WAV file format is a standard for storing such audio data without compression. WAV 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.

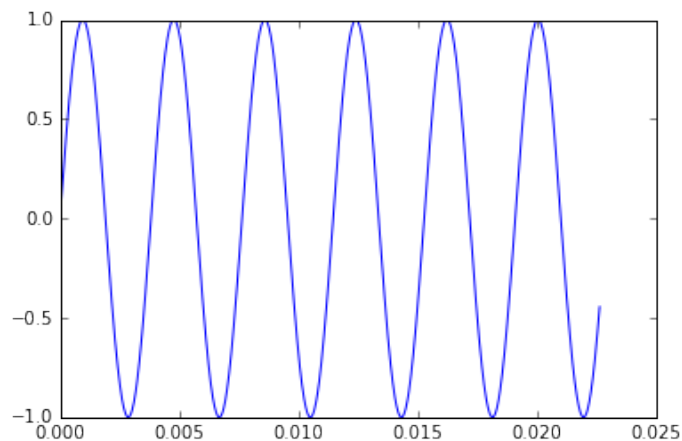
`scipy.io.wavfile.write` Write a NumPy array as a WAV file.

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.

WAV File: Middle C

```
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

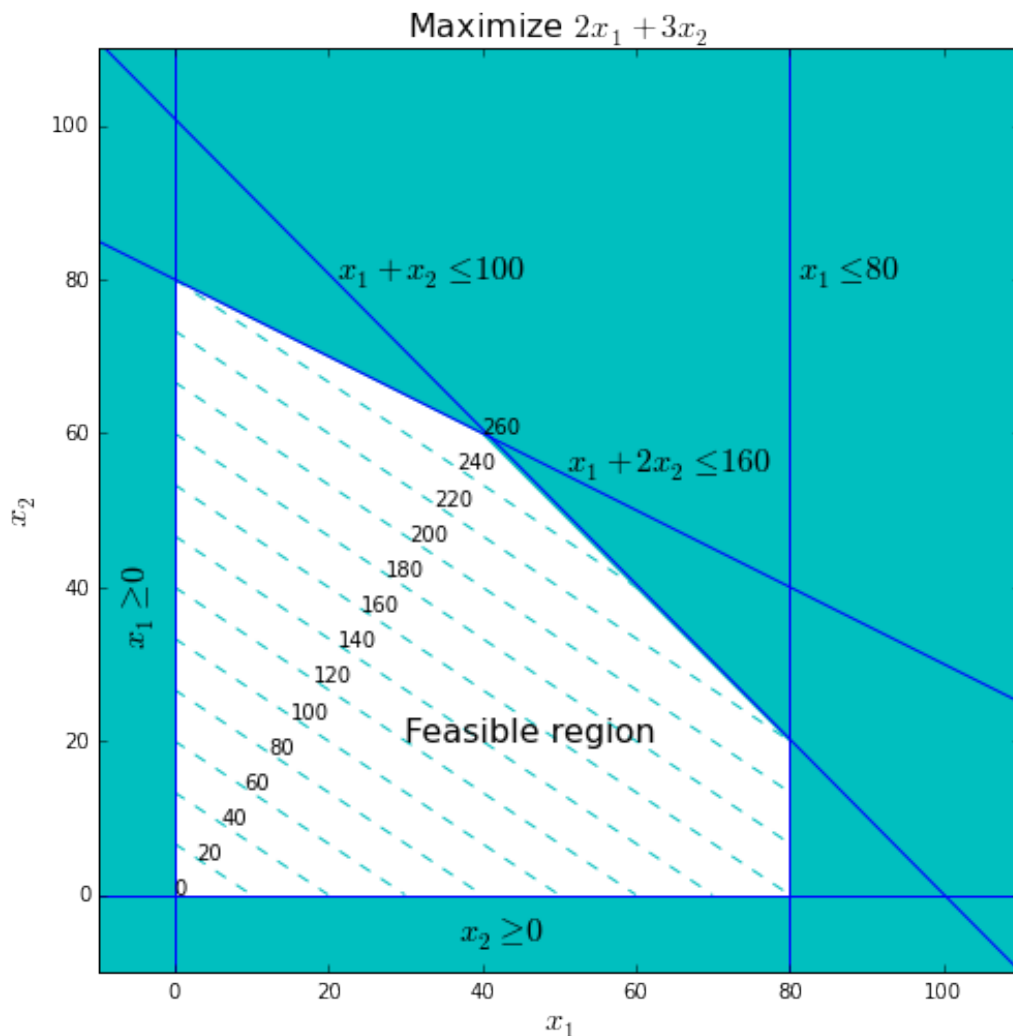
Linear programming problems are a special class of optimization problem. They involve finding the maximum (or minimum) of some linear objective function $f(\mathbf{x})$ of a vector of variables $\mathbf{x} = (x_1, x_2, \dots, x_n)$, subject to a set of linear equality and inequality constraints.

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.

Linear Programming: Finding the Maximum Value

```
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 of the result holds the solution
```

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

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



Note that **linprog** finds the minimum of $f(\mathbf{x})$. To find the maximum, the negative of the **c** coefficient values needs to be used instead.

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:

Sample File Header

```
# MTH 337: Intro to Scientific and Mathematical Computing, Fall 2016  
# Report 1: Primitive Pythagorean Triples  
# Created by Adam Cunningham 8/31/2016
```

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 Jupyter 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.

Index

%%timeit magic, 7

L^AT_EX, 9, 60

abs, 11

and, 49

animate, 63

animation, 63

arange, 42, 45, 48

around, 44

array creation, 41

astype, 44, 45

bar, 53

bar plots, 53

Boolean expressions, 26

Boolean type, 12

break, 32

close, 37, 38

comments, 39, 70

complex numbers, 10

conditional expressions, 28

continue, 32

contour, 57

contour plots, 57

contourf, 57

copy, 42

def, 34

dictionaries, 24

dictionary comprehensions, 33

difference, 24

dir, 19

dtype, 40

dtype, 44

elif, 27

else, 27

empty, 42

empty_like, 42

enumerate, 29

exp, 46

filled contour plots, 57

floats, 10

for, 28, 29, 32, 33, 38

format, 14, 15

function names, 70

functions, 34

generator expressions, 33

hist, 55

histograms, 55

if, 26–28

if-else, 27

imag, 11

import, 18

integers, 10

intersection, 24

IPython magics, 7

items, 31

Jupyter Notebook, 5

lambda, 36

len, 20

line styles, 50

linear programming, 67

linprog, 68

list comprehensions, 32

lists, 19

loadtxt, 61

logical_and, 49

logical_not, 49

logical_or, 49

magic numbers, 70

Markdown, 8
marker styles, 50
math, 19
Matplotlib, 50
max, 22, 47
meshgrid, 43, 44
min, 22, 47
modules, 18

not, 49
NumPy, 40

ones, 42
ones_like, 42
open, 37, 38
or, 49

pie, 56
pie chart, 56
plot, 54
polar, 54
polar plots, 54
print, 13
programming style, 69

random numbers, 65
random.normal, 65
random.rand, 65
random.randint, 46, 65
random.randn, 65
range, 29, 30
read, 37, 38

readlines, 37, 38
real, 11
reshape, 44, 45, 47
return, 34

scipy.io.wavfile, 66
sets, 23
shape, 40
show, 51
sin, 46
sorted, 36
sound, 66
strings, 12
subplot, 58
sum, 47
symmetric_difference, 24

text, 59
try-except, 36, 37
tuples, 22
type casting, 16

union, 24

values, 31
variable names, 17, 69

WAV files, 66
while, 31

zeros, 42
zeros_like, 42
zip, 31