Computer Science for Artists

Tom Sgouros, Jr.

RISD FAV-1539 October 10, 2017, v2.8

0

Computation as a medium

All over RISD, people are making art with technology originally developed for industrial uses. Looms were not developed for making art, neither were kilns, presses, or cameras. But they do a pretty good job of it now. Computers are no different. They were invented for military purposes—shooting down planes and breaking very difficult codes—but people use them now for all kinds of more peaceful things,[1] some of them as inspiring and beautiful as any painting, if in a very different way.

To use any of these technologies as a medium, you need to develop a mastery, at least in the realm you're working in. This course is meant to help people understand the fundamentals of computer operation at the level needed to control them, and to prepare students to dig deeper in the realms to which their interests and aesthetic sense lead them. The goal is to teach the fundamentals of writing modern computer programs, not necessarily one specific language.

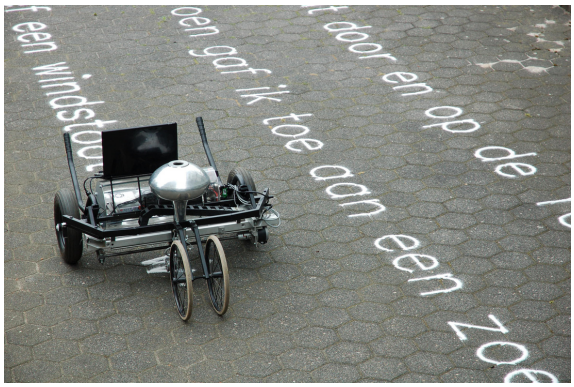


Photo from http://www.gijsvanbon.nl/skryf.html

Because this is a real class, not an abstract one, the class will mostly use a real programming language. Python is relatively easy to get started with, and is widely available. It is used as an extension language to many popular pieces of software, which mostly makes up for its shortcomings, and implements a number of modern features. There are no perfect programming languages, just ones that make different trade-offs than others.

There are 12 weeks of classes outlined here. We are going to emphasize learning by experiment in this class, so each week involves a small amount of lecture material and time to experiment with the new concepts introduced. In addition to the in-class work, there will be a mid-term test and a final exam. More important, however, will be two significant programming projects to complete.

[1] And less peaceful ones, too, as well as some stunningly stupid ones, let's be honest.

0.1 Assignments

From "Pixel" a dance choreographed by Mourad Merzouki

Learning programming through a once-a-week class is probably not the ideal way to do it. It can work, but will require out-of-class application, thought, and independent inquiry. Your success in this class is up to you.

The work of this class will involve weekly assignments, as well as two significant programming projects. The subject of the projects is at the discretion of the student, and can be graphics, virtual reality, databases, AI, whatever. Students will be expected to develop a written proposal for the projects and have it approved by the instructor before beginning. Students will be asked to develop a specification and work plan for their final projects as well.

Projects will be graded on whether they work or not, but also on programming style, clarity, ambition, and documentation. The documentation that goes with the code, should describe problems, explain partially-working code, and say how long the assignment took, and say who your partners were. Assignments are due at **10pm** the evening before class.

0.2 Schedule

September 7	Basics: Variables and Control structures Assignments 1, 2, 3, 4
September 14	Show it to me: Graphics, Assignment 5
September 21	Talking to the world: Input, output, and device control, Assignment 6, 7
September 28	Show it again: More graphics, Assignment 8, 9.
October 5	Rolling your own: Advanced data types, Mid-term test. Assignment 10
October 12	Classes, First project assigned, Assignment 11
October 19	Organizing a big project: On not getting lost
October 26	First project due, discussion, crit. Assignment 12
November 2	Exotica: Abstract data types, Assignment 13, Final project proposal due.
November 9	Final project specs due.
November 16	Computer Babel: What are other languages good for?
November 30	What is Computer Science?, Final projects due.
December 14	Final exam.

0.3 Evaluation Criteria

A Excellent work, imaginative, skillful, accomplished, genuine—the student gets work done on time, and attends class punctually, understands technical and organizational issues, has an energetic approach, uses materials well, deals with challenges in a mature fashion, listens well in class, and gives meaningful feedback to others in critique. The student also has energetic and well-conceived solutions to aesthetic experiments.

B Very good work. The student meets all deadlines and attends class punctually, understands technical and organizational issues. Participates in critique by active listening and offers suggestions to others.

C Average work. Student gets some, but not all work done on time and/or student does not arrive at classes or meetings on time. Misses one class. Lacks curiosity.

D The student has done some work but is not in control of the organization of homework and class work, misses deadlines and is not working as actively or in as focused a manner as required. Misses any combination of two meetings/classes. Lacks curiosity.

F Failing work. Work not done on time. Student is not in control of organization of homework and/or class work. Student is not focused. Misses any combination of three classes or meetings. Lacks curiosity.

The composition of a grade will depend on these factors:

10% Prompt attendance.
10% Involvement in classroom discussion.
15% Progress in understanding concepts over the course of the semester.
65% Completion of assignments, including weekly assignments (15%), two significant projects (30%), and two exams (20%).

The teacher retains the right to fail a student for two unexcused absences.

0.4 Collaboration policy

Students may work together on assignments and projects, though not on exams, and may use code snippets they have acquired on the internet, from a friend, or lying on the street. *However*, all projects will be expected to contain extensive comments and documentation, including explanations of internal functionality. Students must be prepared to explain the operation of any segment of their project to the instructor, *especially* the clever parts.

0.5 Getting ready

To get the most out of this course, you should have a computer with Python installed on it. Python itself is freeware and can be secured in a number of ways. It comes in version 2.7 and in versions greater than 3. We will use Python 3 for this course, though for many projects and for most of the homework, the version will not matter.[2]

There is no required text for this course, though there are lots of good Python books out there. There is a Python web site that has an invaluable set of reference pages, and most programmers make extensive use of them: `https://docs.python.org/2/contents.html`.

The current version of this document can always be found at `http://sgouros.com/cs-fav`. You'll see a revision number at the bottom of the page, to help you make sure you're looking at the newest version.

You will also need a text editor. The Sublime text editor is a good one, and can be set up with a Python module for testing code. Check out here for advice about setting it up for Python: `https://dbader.org/blog/setting-up-sublime-text-for-python-development`. There are plenty of other text editors out there that will suffice: TextWrangler, vi, Nano, Emacs, whatever. TextEdit on OS X is not a great one, but even it can work.

There are lots of Python development environments out there, that have an editor and the Python environment built into one program. IDLE is a Python-specific development environment popular with beginners, and Eclipse is a powerful environment used in the professional programming world. There are plenty of others. You may use one of these—they can make programming much easier for you—but your programs will be tested and run on the instructor's command-line, and the class will not cover their use.

PYTHON PACKAGES

For this class, we will be using a small number of Python packages, including turtle graphics. You are free to use other packages, but in the interests of spending our time on programming concepts, the instructor cannot help to install them on your machine. Also, if you use an unusual or difficult-to-install package, the instructor may not be able to run your project on *his* machine. Please check with the instructor before deciding to do a project or homework assignment with a new package. If the instructor cannot run your assignment, your documentation and comments will be very important.

[2] The differences are subtle, but the versions *are* different. If something like the print function, or division, isn't working for you that you've seen working in class, it might be that you're using version 2 instead of 3.

Contents

1 Basics: Variables and Control structures

WE START WITH THE BASICS of a computer program. A program is just a recipe, a list of steps to undertake: data to bring in, data to write out, operations to perform, screens to blink, speakers to hum, robot motors to move, whatever.

All computer programs are composed of the same components: a bunch of data, and a set of instructions to perform on it. The data might be stored on the computer already, or it might be live, typed by the user, read in on a communication line, observed through a camera, recorded by a motion sensor. The instructions are generally already on the computer, but even these can be composed on the fly.

There are a lot of different computer languages, and they are good for a lot of different things. In a theoretical sense, all computer languages are equivalent to each other[3] but in a practical sense, they are pretty different. The practical aspects have to do not only with the structure of the language—the syntax and semantics of the instruction set it provides—but also the reality of where it is available and where it is not. We'll talk about this more in week 8.

But enough of that, let's talk about the basics.

1.1 Names and things

One important distinction that will come up a lot is between a name and the thing it stands for. The number 6 is just a number, but if we invent a name and give it the value 6, then we have a *variable* we could use for lots of different values:

```
>>> fred = 6
```

Python is an "interpreted" language, as opposed to a "compiled" language. We'll talk more about that distinction in the future. For now, it just means that you can fire up Python, and try it out a little bit at a time.

The >>> in the above is the prompt Python offers you to say, "I am ready to serve you, master." The rest is what you type. Start Python, admire the prompt, type the example, then type `fred` and hit return and see what happens.

Not much, really, but it does confirm for you that the number is attached to that name. Now type:

[3] This is due to a proof from Alan Turing developed in the 1940s, before any of them existed. So far no one has made one up that isn't equivalent in the way he described.

```
>>> fred + 27
```

Pretty much what you'd expect if you took algebra once. Unlike in algebra, however, one-letter variable names like "X" or "Y" are a really bad idea. One of the things we've learned about computer code in the past few decades is that pretty much no one writes programs that never have to be re-written. So make your code as easy to read as possible. You'll hear a lot about "self-documenting code" as you work in this field. There is no such thing, but it is possible to write code that requires a minimum of documentation by choosing names wisely. It doesn't matter to the computer if you call your variables something like `integerNumberOfAttemptsBeforeFailure`, but it will probably matter a lot to you, at some point in the future.

1.1.1 Scope

One complication about names is something called *scope*, the places in your program where you can count on the association of a name and a thing to be valid.

A variable name valid at any place in some program is called a *global variable*. A variable that is only valid in some small part of the program is called a *local variable*. Controlling the scope of your names is part of avoiding making a mess when you take on a big project. Python contains a relatively neat way to contain names you define called *modules*. We'll say more about all of this in the future.

1.2 Data types

There are a lot of different types of data one might ask a computer to handle. Your age, for example, is a data value that might be usefully associated with your name. So is your height and weight. But there's much more: all the text in a book, for example, or all the page headers and fonts in that book. Or a temperature measurement every day for a year, the location of a robot, the location of everything else the robot knows about, a picture of a duck, whatever.

Representing numbers is an interesting story. Everything in a computer is at root represented by binary numbers. The number 4, for example, can be represented as an *integer* — the value after 3 and before 5. An integer is represented by a single binary number, so is bounded by the upper and lower limit of how large a number you can express with the given number of bits. A *floating point* number is composed of two values, a number and an exponent, and the value is the number times the base raised to the exponent. You can express a much bigger (or smaller) number this way, and can do fractions, too, but the values are always approximate.[4]

To make things just a bit more confusing, a 4 can also be represented by the numeral, the three usually straight lines printed on a page that you read as a '4'. Programmers call this *character*

[4] The Scheme computer language refers to integers as *exact numbers* and floating point values as *inexact numbers* for that reason.

data, and there is an *encoding* that maps a *glyph* — the shape you want to see on a page — to a number. The standard encoding, also called ASCII, maps a 4 to the number 52, the number 5 to 53, and so on. You will almost never need the actual values of whatever encoding you are using. However, if you want to read and write data from other languages than English, you may need to know what encoding you are using.

The data used by most computer languages can be divided into two categories: simple and compound. Simple data types are things like an integer or a floating point number or a character of text. Compound types are things like strings of text, arrays of numbers, associations of values, and more. Most modern languages allow you to define your own kinds of data, which allows you to go to town and define even more exotic types: trees, hashes, whatever. We will learn more about these eventually.

Python, which is a modern language, has a few simple data types.

integer positive or negative numbers up to `sys.maxint`.[5] These are represented internally by a binary number.

long positive or negative numbers with no limit. These are represented by a collection of binary numbers, but you seldom need to be aware of that.

float floating point numbers, check out `sys.float_info`.

complex two floating point numbers, real and imag.

bool A boolean value is either `True` or `False`, nothing else.

Then there are some compound data types, made up of collections of other data.

string, unicode character text, in single or double quotes, use `[a]` or `[a:b]` to get pieces of them. Index starts from zero. Get used to that.

list An ordered collection of values. They can be of different types, but order is important. Use brackets: `['eubie', 'noble', 'scott']`. You can access elements of them in the same way as strings, using the number of the element, starting from zero at the first element.

set An unordered collection of values. Pretty much the same as a list, but you can't get elements of them in the same way as a list or a string because they have no order.

tuple A collection of values that constitute a single value, like an aisle and shelf number in a library, or a name and age. Use parens: `('harry', 19)`. Tuples are meant to be immutable, that is, once defined, you can't change them or any element of them.

dict An association between two data types. Used to store information about some name, or data about some number. `s{'harry'} = 19`

[5] At the Python prompt, type `import sys` and then `sys.maxint`.

There are also buffers, bytearrays, xrange, and frozenset compound types. Not really built-in, but added to accommodate Python's popularity. You might want to use one of these in your final project, but you can look them up online if you need them.

You can define other types, too, and we'll get to that.

In Python, the values have a type, and any old name can apply to them. In other languages—C/C++, Java, FORTRAN—the variables are set up to only hold a particular type. That is called *static typing* and Python's kind is *dynamic typing*.

The data types would be no fun without operations to play on them, and that's the other half of a computer program. There are some standard operators, like +, -, /, *, and so on, and a whole bunch more. They come in categories:

Arithmetic Operators `+ - / * ** % //` Use these to combine two values in more or less interesting ways. The `%` operator does a "modulo" and the `//` does an integer divide, sort of.

Comparison (Relational) Operators `== != < > <= >=` These are used to compare two values, equal, not equal, less than, greater then, and so on.

Assignment Operators `= += -= *=` ... Use these to change a value. Changing a variable by adding another value is a common enough operation that Python has an abbreviation for it, so instead of writing `a = a + 1`, you can write `a += 1`.

Logical Operators `not and or` For a logical value (the result of a comparison, for example), these operators provide the usual logical result.

Bitwise Operators `& | ^ ~ << >>` And, or, exclusive-or, and not logical operations, per bit, and shift bits left and right. These operators really only make sense for an integer.

Membership Operators `in not in` These are only for sequences; you can use them to determine if some value is contained in the sequence.

Identity Operators `is is not` This is a little more sophisticated than equality. Use these to determine whether two objects are the *same*, not just equal.

One thing that you have to remember is that not all operators have an obvious meaning on all data types. What does "in" mean on an integer, for example? Or "bitwise and" on a list? When you get to define custom data types, you can define a meaning for operators with them, but even there, not all definitions will make sense.

The biggest class of functions is written as `function(argument1, argument2,...)`. You will be writing lots of these. A bunch of these are built in, too, like `len()` which you can use to find the length of things or `print()` which you can use to print things.

Try this:

```
>>> a = "Once upon a midnight dreary, while I pondered weak and weary..."

>>> print(a)
```

```
>>> print(a[21:27])

>>> print(len(a))
```

A useful set of functions is named for the data types. So `int(34.2)` equals `int("34")` equals 34 and `bool(29)` is True while `bool(0)` is False.

There is a `help()` function built in. Try `help(__builtins__)`. You can gauge the progress of your mastery of Python by how much of it you understand.

Assignment 1: Experiments Try some things. Encode a few lines of a poem or song you know into a list of strings and then compute the length of each line. Can you turn that list into a list of numbers, each of which gives the length of a line? Can you also find the ratio between the longest and shortest lines?

Put these experiments into a text file to be mailed to the instructor at `tsgour01@risd.edu`. You will complete all the assignments in this class this way.

1.3 Control structures

Once you have the data worked out, there remains the issue of how to encode the directions — what the computer should *do* with that data. There are three basic categories of control commands, conditionals, loops, and functions.[6]

WHITE SPACE

One of the drawbacks to Python is that white space, which is by definition invisible, has meaning. The Python interpreter uses the indentation of your code to interpret what constitutes a "block" of instructions. In the early days, the interpreter insisted on tabs instead of spaces. It doesn't do that any more, but proper indentation is still important to getting your code to work right. Don't blame me.

1.3.1 Functions

Functions first, because it gives us something to do. A mathematical *function* has inputs, actions, and outputs. In Python, a function definition looks like this:

```
def func(a, b):
    return a + b
```

Remember, in Python, the spaces are important and that's how you know what is nested in what. The editor you're using should be able to help you with the proper indentation. In this example, `func()` is the name of the function (and we often write function names with the parentheses to make it clear these are functions) and `a` and `b` are the *arguments* of the function. The sum of the arguments is returned by the `return` statement there. Try it out.

There are two categories of things a function can do. It can perform some action, and it can provide a *return value*, which means you can substitute the function into an expression, where it will be replaced by its return value. If you type:

```
>>> 4 + func(3, 4)
```

You'll get 11 as a response because the function returns 7 and Python adds it to the 4 in the expression. The other kind of thing a function can do is sometimes known as its *side-effects*. Here's another function that has no return value, but it has a side-effect of printing the sum of its arguments.

```
def printfunc(a, b):
    print(a + b)
```

[6] Actually there is a fourth one, a transfer of control, also called a "goto" or a "skip." These are the black sheep of programming languages and unless you're working in a very low-level language, if you have found yourself needing one, you're doing it wrong.

Using the function *looks* the same as using the addition function above, but it's quite different in important ways. If you type:

```
>>> 4 + printfunc(3,4)
```

You'll get an error because `printfunc()` has no return value.

The difference between a function's result and its side-effects can be confusing, but it is an important distinction to keep in mind, especially as things become more complicated.

Optional argument values You can have an optional argument for your function if you define a default value for it. Specify the default right in the argument list, like this:

```
def myfunc(arg=4):
    print(arg)
```

If you supply no argument for this function, it will use the value 4.

```
>>> myfunc(3)
3
>>> myfunc()
4
```

If you name all the arguments in a function definition, the order in which you write them no longer matters.

CAUTION

Do not use a list or dictionary as a default argument. The reasons why are complicated, but the short version is that it will not do what you think. You can use the value `None` for a default list or dict argument and test for that with the conditional statements you'll learn about in the next section.

2 **Assignment 2: Adding lists** Write a function that accepts two lists of numbers, adds up all the numbers in each list, prints the result, and returns the length of the two lists.

Put this function into a file called something like `myfile.py` and run it at the command line like this:

```
$ python3 myfile.py
```

On Linux or Mac computers, you can arrange your file to execute directly from the command line, like any other command, by inserting this line at the top of the file:

```
#!/usr/bin/env python3
```

You'll also have to make your file executable, which you can do like this:

```
$ chmod +x myfile.py
```

Again, this is only Mac and Linux machines. See

1.3.2 Conditionals

Any useful computer language has some kind of conditional statement: if this, then that. Python's looks like this:

```
if a == b:
    print("a equals b")
```

Notice the double equal sign. This is the test for equality, something different than using an equal sign to set the value of `a`. You can have a more elaborate if statement, too, that has consequences if the condition is *not* met:

```
if a == b:
    print("a equals b")
else:
    print("a does not equal b")
```

You can also have alternative conditions:

```
if a == b:
    print("a equals b")
elif a < b:
    print("a is less than b")
elif a > b:
    print("a is greater than b")
else:
    print("you should never see this printed")
```

The `elif` is to be read "else if".

These can get a lot more complicated, especially the condition itself. Most values in Python can be used in a condition, so you can say:

```
if a:
    print(a)
```

If `a` is an integer value, this will test if it's zero and only print if it's not zero. If it's a string, it will print if it's not an empty string. If it's a list it will print if it's not an empty list, and so on.

Strings have a whole class of methods for testing if a string is all digits, or all letters, or all lowercase, such as `str.isalpha()`, `str.isdigit()`, `str.islower()`. There are several more.

3 **Assignment 3: Comparing lists** Write a function that accepts two lists of any length as arguments, and returns `True` if both lists have two or more entries and if the second value of the first is bigger than the second value of the second. If any condition is not correct, return `False`. The function must not fail no matter how long the arguments are.

1.3.3 Loops

There are two kinds of loops in Python. You can iterate while a condition is true (a *while loop*) or iterate over the elements of a list (a *for loop*).

While loops look like this:

```
a = 0
b = 5
while a < b:
  print(a, "is less than", b)
  a += 1
```

The body of the loop will be executed so long as the condition is true, so you'll see... well try it and see what you'll see.

For loops iterate over members of a sequence. They look like this:

```
for c in "hello":
  print(c)
```

Or this:

```
for c in range(5):
  print(c)
```

Or this:

```
[c + 1 for c in range(4, 10)]
```

Or about a thousand other variations.

What does the `range()` function do in the above?

1.4 Document your code

It is difficult to stress the value of documentation highly enough. Virtually no computer programs are written that do not have to be rewritten at some point. You are writing code not just for a computer to read, but for some other person (maybe you!) to read, at some time in the future.[7] Obviously you need to make your code readable by the computer in order to get it to work at all. Unfortunately, there is no such requirement for the documentation, and so the world is awash with computer code that is at best only partly documented, if it is documented at all. Much of this code will be useless without documentation, so do yourself a favor and please include copious comments in your code.

Python has two ways to document your code. Immediately after the `def` or `class` statement, you can enclose text in triple quotes, like this:

```
def myFunction(arg):
   """This is a function to show off Python's function and class
      documentation features."""
  ...
```

Some people format them this way:

```
def myFunction(arg):
   """
   This is a function to show off Python's function and class
   documentation features.
   """
  ...
```

Use whatever style is easiest for you to read.

You can also use in-line comments. Python will ignore anything to the right of a # character, so you can use that space to describe what you're doing. The more comments in your code, the better.

Finally, when you write comments, please don't do this:

[7] In the context of a class, where assignments don't typically have a long life beyond the semester, perhaps you will not be using this code at some point in the future, but your instructor (me!) will certainly be reading it.

```
# If x is greater than 5 then raise an error.
if x > 5:
  raise ValueError("Input too large.")
```

This comment just rewords the two lines of code that follow it, and is therefore useless. It's generally pretty clear what's actually happening in a computer program. What seldom is clear is *why* it's happening. Something like this is likely to be far more useful information to the person reading this code in the future:

```
# If x is too large, the calculation that follows will fail.
if x > 5:
  raise ValueError("Input too large.")

# Calculation here.
```

This is why claims you will see about "self-documenting code" are usually just gibberish. Whatever descriptive names you give your variables and functions, the code itself will never tell you why some clever trick works, describe the subtlety of any invalid inputs, or provide the high-level view of what's going on. You need to provide lots of description to make it clear what's happening and where your program is headed.

That said, it is still good advice to give your variables and functions descriptive names. Use `height`, not `h`, for example. Python will ignore empty lines, so you can use those to separate your code into conceptual units and make it easier to read.

4 **Assignment 4: Comparing elements of two lists** Write a function to compare each of the elements of two sequences. They can be character strings, lists, or tuples. Have it print out a table of the results, including some kind of summary analysis. You will make up the summary to present, but it should include:

1. The lengths of the two sequences, and
2. The proportion of elements that are the same or different.

Feel free to include any other information you think might be useful or interesting, and don't forget to include lots of comments.

Put this function into a file so you can edit it offline and read it in as you need to. If you put it in a file called `test.py` you can do `import test` and refer to your function `func()` as `test.func()`. If you change the `test.py` file, you can type `reload(test)` to try your changes. THe file is what you will turn in to the instructor.

The real assignment here is to get accustomed to whatever text editor you want to use, and to develop a workflow that will accommodate trying things out, editing, and trying them again. You'll do a lot of that in this course.

2

Show it to me: Graphics

PICTURES REALLY ARE WORTH a thousand words. Images are more striking than text, and the same data, shown visually, has an impact far beyond mere words. So let's find ways to do things with graphics.

Python is not intrinsically a graphics program. It has no built-in graphics primitives or picture functions. But you can still do lots of elaborate and amazing graphics with it, by importing graphics packages into it, or by using graphics programs that have imported Python into them, as an extension language.

Graphics packages are a pretty varied lot, since graphics is a pretty big category. Roughly speaking, there are three or four important categories of functionality in graphics. You can edit and process images, such as JPEGs, and PNGs, you can make drawings, either live or in one of those file formats, and you can do 2D or 3D animation. In addition to these, many users need graphics applications to display graphical representations of data: graphs and charts.

There are thousands of freely available graphics packages out there. Some are more useful (up-to-date, well documented, well written) than others, and some are kind of a pain to install. (See section 10.1 about package installation). The assignments in this class will use the `turtle` graphics package, which comes with most installations of Python. For your class projects, you are not limited to this package (though please see page 4 about using new and unusual packages).

Here are some of the more popular graphics packages.

Turtle Turtle graphics is a vector graphics package that asks you to think of graphics programming as having a small turtle drag a marker around behind it as you issue instructions about where it should crawl. It is not very good at making elaborate graphics, but it is very good at illustrating most of the important concepts you'll need.

Pygame The pygame package is a relatively straightforward collection of animation and graphics primitives suitable for simple game programming. It has extensive online documentation, and installs easily using pip.

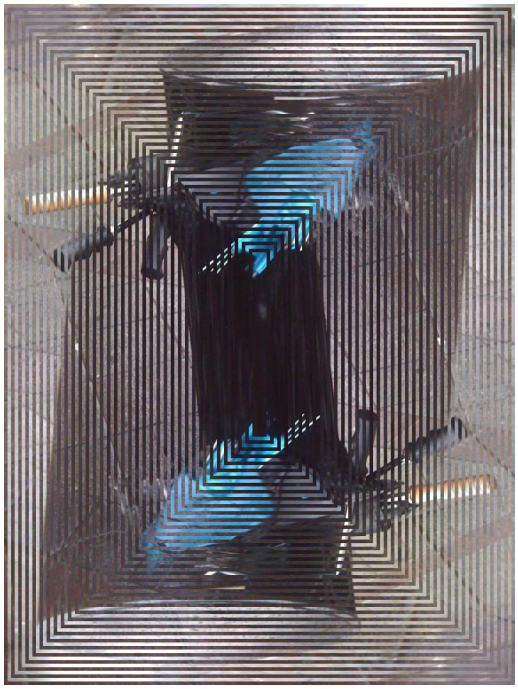
PIL The Python Image Library is another great way to manipulate graphics data from a file. Use it to read graphics data in a variety of different formats, edit the graphics data, and write it back out again. PIL went through a dormant period, and was revived as Pillow, the name of the current best distribution of it.

Here's an example PIL session:

```
>>> from PIL import Image
>>> im = Image.open("umbrellas.jpg")
>>> for i in range(5,355,5):
...     box = (i, i, 720-i, 960-i)
```

```
...     region = im.crop(box)
...     region = region.transpose(Image.ROTATE_180)
...     im.paste(region, box)
...
>>> im.show()
>>> im.save('umbrellas-mod.jpg')
```

And here is the result:



matplotlib This library is an accompaniment to the Numpy and SciPy packages, which are good choices for doing statistical data analysis. Matplotlib is meant for graphing, so it has axis-drawing and line-drawing functions, and you can use it to make all kinds of data presentations. There's an example session on page 22.

PythonMagick ImageMagick is the reigning champion of command-line image-processing convenience. PythonMagick is a Python interface to that functionality, and you can use it for hundreds of functions, from adjusting brightness and contrast to adding annotations to converting images from one format to another. Installation appears to be from source, and it will require the ImageMagick library, so is not for the faint-hearted, or the beginner.

OpenCV This is the premier computer vision package out there. It is primarily used to find and analyze visual features in image data, but it has a lot of live graphics features that make it suitable for a number of other uses. Plus it has face-recognition software you can add, for fun. Unfortunately, it is often challenging to install and the advanced usages (like face detection) can be difficult to implement. But it is widely used, so there is help available out there and it can do things like capture the output from a USB or built-in camera.

Here's an OpenCV script to capture the output of a built-in camera. Put it in a file called *camera.py* and make it executable (page 50) and see if you can understand what it does.

```
#!/usr/bin/env python
import cv2

vc = cv2.VideoCapture()
vc.open(0)
cv2.namedWindow('result', 1)

while True:
    vc.grab()
    retval, image = vc.retrieve()
    cv2.imshow('result', image)

    if cv2.waitKey(1) >= 0:
        break

cv2.destroyWindow('result')
```

There are also a few graphics programs that use Python as an extension language. This means that if there is some function the program does not have, you can write it in Python and install it into the program.

Maya This is the leading modeling and animation software on the market. Widely used in industry, and widely available, though expensive.

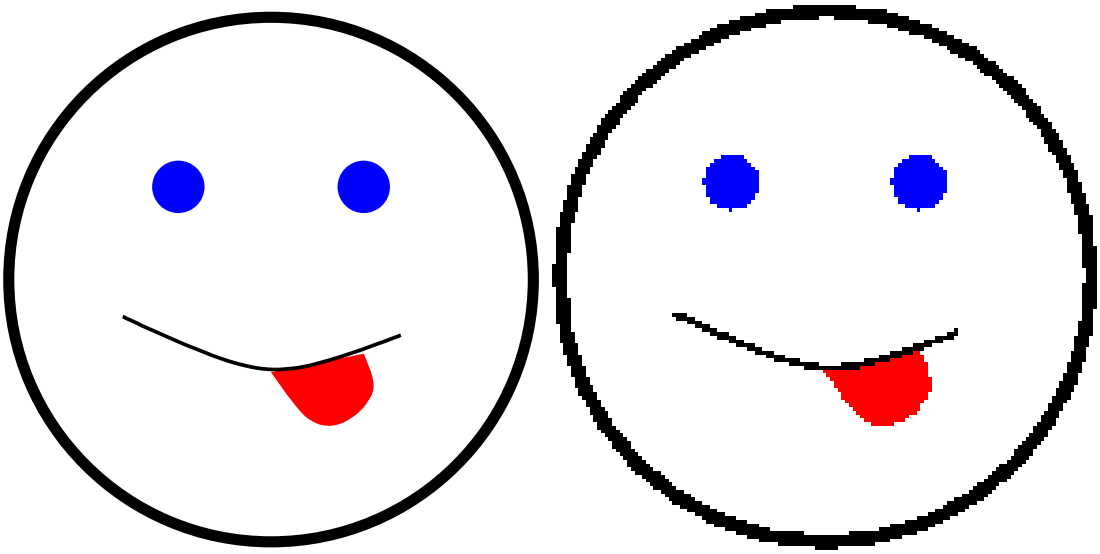
Blender A freeware version of Maya. It's really a different program, but it does many of the same things, and is freely available. Many of the basic Blender functions are themselves written in Python, and each menu option provides little hints about how to use them in a Python script.

Gimp Scheme is the native extension of Gimp, the leading freeware photo image processing program, but you can also write extensions in Python.

In other words, there are lots of ways to draw, edit, and improve pretty pictures with Python. Some basic vocabulary will help, though.

2.1 Raster, vector

The first important distinction in graphics has to do with file formats. There are two categories of graphical image storage formats: vector and raster. Roughly, vector graphics keep track of the position and location of each line that makes up a drawing, while raster graphics keep track of each dot. Vector graphic formats, like PostScript, PDF, and SVG, are best for line art and graphic elements, like fonts and graphs. A vector graphic can be reproduced at any scale without loss of resolution.



Raster graphic formats, by contrast, do not keep track of the lines themselves, but of which pixel of an image is needed to render a line. To make a raster image, the computer splits the image up into little cells and records the color of each cell. Raster formats include GIF, PNG, and BMP, among others. Raster images are sensitive to the scale. That is, if you blow up the image, you can see the pixelation of the lines in the image.

This means that some images are more suited to one kind of image format or the other. In general, raster formats are better for images with many gradations of colors and less sharp edges, while vector formats are better for images with fewer colors and sharp edges. Fonts for printing, for

example, are usually stored as a vector format, while photographs are usually better rendered in one of the raster formats.

2.2 Primitives, screen coordinates

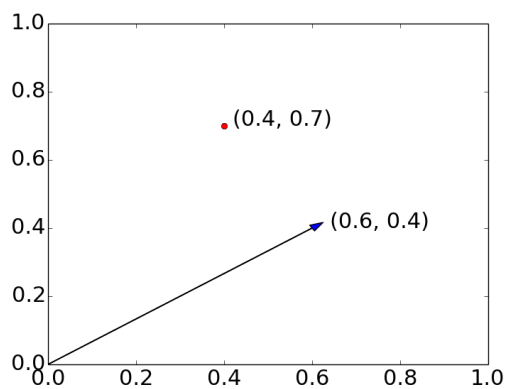
One important thing any graphics package must have is a *coordinate system* to locate points on the image. In graphics, people talk about *screen coordinates* which is how you locate a pixel in an image. Usually, the X coordinate locates the horizontal distance from the left edge of your window or screen, and the Y coordinate locates the vertical distance from the top.

Beyond that, many programs assert some kind of user coordinates on a drawing. Unfortunately, there is little standardization, so one of the first things to do when learning a new graphics program is to figure out which way the X, Y, and Z (for 3D) axes point.

Any graphics package you use is going to have a set of data types and *primitives*, the basic functions you use to assemble an image. But these are going to be different for the different packages. For a vector-drawing package, the primitives will probably include things like drawing lines and dots while an image-processing package is more likely to have primitives like select an area, or adjust contrast.

When you have multiple coordinate systems that have to be managed, you'll often hear of a *transformation matrix*, which is a standard mathematical technique for changing one coordinate system into another. You probably won't need ever to calculate one, but you should know what the word means so you don't choke when reading it.

One distinction that is often made in computer graphics is between a *point*, an identifiable point in space, and a *vector*, a location in space *and* a direction. Often the direction a vector implies will simply be the direction from the origin point of the coordinate space to some point in space, so the written description of the two will seem similar, but they are different things. To the right is a depiction of the two concepts. They are both indicated by a pair of numbers, but they mean different things.



Here's a matplotlib recipe to make the image above illustrating the difference between a point and a vector:

```
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 22})
plt.axis([0,1,0,1])
plt.plot([0.4],[0.7],'ro')
plt.text(0.42, 0.7, '(0.4, 0.7)')
plt.axes().arrow(0,0,0.6, 0.4)
```

```
plt.text(0.64, 0.4, '(0.6, 0.4)')
plt.show()
```

2.3 Turtle graphics

Turtle graphics is where we begin. Here is a turtle graphics program:

```
import turtle            # Says "Let's use the turtle library!"
window = turtle.Screen() # Creates a graphics window.
kim = turtle.Turtle()    # Create a turtle, call it kim.
kim.forward(150)         # Have kim move forward by 150 pixels.
kim.left(90)             # Turn left 90 degrees.
kim.forward(75)          # Move forward 75 pixels.
```

To make the system work, you initialize a screen, create a turtle, and issue commands as in the above example. You can have multiple turtles on your screen, with different names, moving independently.

Here is a list of some of the commands your turtle will obey. There are many more (and several of these functions have different ways to call them), so please consult the turtle graphics documentation, which can be found online.

forward(distance) Move forward `distance` units.

backward(distance) Move backward `distance` units.

right(angle) Turn right `angle` degrees.

left(angle) Turn left `angle` degrees.

goto(x, y) Move to a particular point on the screen given by `x` and `y`.

dot(size, color) Draw a circular dot with diameter size, using color. If size is not given, the larger of pensize+4 and pensize×2 is used.

stamp() Draws a little cursor shape to be left at the current point.

shape(name) Change the turtle cursor shape. Try `"circle"` or `"turtle"` or `"arrow"` to see what happens.

position() Returns a pair of numbers showing the (x,y) position of the turtle.

heading() Returns the current direction of the turtle.

towards(x, y) Returns the angle between the current direction and the line from the current position to the given (x,y) positiion.

distance(x, y) Returns the distance between the current position and the given (x,y) position.

pendown() Put the pen down. Moves after this will make a mark.

penup() Pick the pen up. Moves after this will not make a mark.

width(width) Set the pen width to `width`.

pencolor(color) Set the pen color. You can use either the name of a color, an RGB tuple, or a string like `"#bba903"`.

fillcolor(color) Set the fill color.

begin_fill() Begin outlining a shape to be filled.

end_fill() End outlining a shape to be filled.

write(text) Print some text at the current point.

reset() Clear the screen of whatever this turtle has drawn and return it to the origin.

clear() Clear the screen of whatever this turtle has drawn and leave it wherever it was.

speed(spd) Set or return the speed of the turtle animation. You can set this to a value between 1 and 10 to speed up the animation, where 10 is the fastest. A value of zero turns the animation off completely, and just shows you the result.

showturtle() Show the turtle.

hideturtle() Hide the turtle. This can speed up the animation considerably, at the loss of some amusement value.

A useful function is `turtle.done()` which will pause the program until you close the turtle window. If you don't use this, the window will disappear as soon as you leave Python. If you're running your turtle in some way besides the command line, this will be immediate.

5 **Assignment 5: Tic-tac-toe** Draw a tic-tac-toe game.

1. Write a function to draw a tic-tac-toe board. Write a main program that opens a window and calls this function.
2. Write a function that will put an X or O in any of the nine spaces. Add this to your main program to draw a few and show it works.
3. Change the program to add user input so you can play the game. (Read ahead to find out about the `input()` function.)
4. Extra credit: Write a function to make moves so you can play against the computer. (The credit is for making the attempt. Don't worry if it isn't a *good* player.)

Please remember that you will be graded not only on how the code works, but how well it is described and documented. The code is a set of instructions for the computer, but it is also a message from you to me. Make sure I can read it easily.

TURNING OFF TURTLE ANIMATION

If you weary of the animation, set your turtle's speed to zero, and issue the command `turtle.tracer(0,0)` before beginning your drawing and do `turtle.update()` when drawing is done. This will turn off the animation and make the drawing appear nearly instantaneous. See the documentation online for explanation of these two functions.

3 Talking to the world: Input, output, and device control

As we've seen, a computer program consists of the instructions and the data they operate on. Sometimes it takes a little work to get the data into the program. There are a variety of ways to do this, but the two fundamental types of input are *synchronous*, where the program asks for and then waits for the input. Prompting a user to type some input is an example of synchronous input, as is reading data from a file.

The other kind is *asynchronous input*, where the input comes on its own schedule, and the program has to accommodate it on the fly. When you move a mouse or touch a touchpad and see the cursor move in response, this is usually asynchronous input. The program does not prompt you to move the mouse; you just move it. You will hear more about this in section 4.2.

3.1 Format

We should digress a moment to talk about the `format()` method above. This is a way to print things a bit more nicely than just with a raw `print()` statement. Because printing things out is an important way to debug your programs, this can be important.

The `format()` method looks for pairs of brackets in your string and replaces them with its arguments. So, for example, `"{}".format('henry')` will produce `henry`.

More examples:

```
>>> "{} {} {}".format("one", "two", "three")
'one two three'
>>> "{} is the same as {}".format("one", 1)
'one is the same as 1'
>>> "{} and {} is a funny {}".format('one', 'done', 'phrase')
'one and done is a funny phrase'
```

You can put numbers in the brackets to refer to some of the arguments in case you want to change the order, or repeat them:

```
>>> "{0} plus {0} equals {1}".format('one', 'two')
'one plus one equals two'
>>> "{2}, same as {2}, different from {0}".format(6,3,2)
'2, same as 2, different from 6'
```

You can suggest padding to make columns come out neatly:

```
>>> "{1:3}.{0:02}".format(4,5)
'  5.04'
```

The `{1:3}` instructs format to pad the 5 to take up three spaces, and use spaces for the padding. The `{0:02}` says to pad the 4 (argument 0) to two spaces, using a leading zero.

The `format()` method is a rewrite of the older way to format text output in Python (look it up if you're curious), and it is both extremely flexible, and filled with unlikely features. For example, here's how you can use it to pad a string with asterisks, and to convert an integer number to different bases.

```
>>> '{:*^30}'.format('centered')
'***********centered***********'
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
```

You can also use `format()` to use commas for big numbers, align numbers and text in complicated ways, format dates, access pieces of arguments, access arguments by name instead of position, and too much more. Please check out the Python documentation for a complete list of `format()` features.

6 **Assignment 6: Reformat previous assignments** Rewrite your table summary from assignment 4 with format statements to make the output tidier and easier to read.

3.2 Typed input

We have already encountered the `print()` and `format()` commands for making readable output for your program (section 3.1). What about input? There are a couple of built-in python functions used for getting typed input from a user. The easiest to use is `input()`, which prompts the user and returns the string the user typed, minus the *newline* character that marked the end of the string (what you get when you hit "enter").

```
>>> s = input("Type something: ")
Type something: I'm typing something.
>>> s
"I'm typing something."
```

The argument to `input()` is the string to use as a *prompt*, and the return value is whatever you typed in response to that prompt. The input is limited to a single line of text.

The prompt is optional here, so you can just do without:

```
>>> input()
hello
'hello'
```

3.3 Reading and writing files

Typed input and output is useful, but it's very limited. To be useful, you need to be able to read and write much more data than you can type at a prompt. A way to read and write data from disk files is important to any real application.

Reading or writing a disk file involves three steps: opening the file, reading from it or writing to it, and closing it. Python does it with the `open()` function that creates a *filehandle* object, that has a `write()` method for writing and a `close()` method for closing a file:

```
>>> f = open('testfile', 'w') # Open a file for writing.
>>> f.write("Once upon a midnight dreary,") # Write something into it.
>>> f.write("While I pondered, weak and weary.")
>>> f.close()
```

Now go look in the default directory, and you'll see a new file called testfile, which you can examine. Read the file back with the `read()` method, like this:

```
>>> f = open('testfile', 'r')
>>> f.read()
'Once upon a midnight dreary,While I pondered, weak and weary.'
>>> f.close()
```

Notice that you get back *exactly* what you wrote out, and you get it all at once. In this case, it might not be exactly what you want. The `read()` method has an optional argument that says how many bytes of the file you actually want, and that might help, though it is not very convenient for lines of text that might vary in length.

Python can also read lines of text, if they are terminated properly. Here is an example:

```
>>> f = open('testfile', 'w') # Open a file for writing.
>>> f.write("Once upon a midnight dreary,\n") # Write something into it.
>>> f.write("While I pondered, weak and weary.\n")
>>> f.close()
```

The *newline* character `\n` marks the end of a line. Now we can open the file and read it a line at a time, with `readline()`.

```
>>> f = open("testfile", "rU")
>>> f.readline()
'Once upon a midnight dreary,\n'
>>> f.readline()
'While I pondered, weak and weary.\n'
>>> f.readline()
''
>>> f.close()
```

The `r` in the `rU` in the `open()` call means "open this file for reading," and the `U` means "allow different newline conventions." This will allow the code to work equally well on text files generated on Mac, Linux, or Windows which have different conventions for what constitutes the ending of a line of text.

3.3.1 With

Python has a `with` command that you can use to enclose file handling into a block. This has the advantage of doing the file close for you, and handling some basic errors. These lines read an entire file's contents into the `data` variable.

```
with open('testfile', 'r') as f:
    data = f.read()
    ...
```

You can use this, with `as`, to read lines of text from a file. Try this and see what it does:

```
with open('testfile', 'r') as lines:
    for line in lines:
        print(line)
```

This can be a little hard to understand. The `lines` variable here is a file descriptor, not a sequence of characters, and so the loop does successive reads of the file. Try this, and see if you can understand the difference between it and the above:

```
with open('testfile', 'r') as lines:
    for line in lines.read().splitlines():
        print(line)
```

In this case, `lines.read()` sucks in the entire file, and the splitlines() method breaks the lines on the newline character. You'll notice a difference in the output, too, since `splitlines()` does not include the newline character in the output, while the first loop does.

3.3.2 Processing text

When processing lines of text, you may find the `split()` method of strings convenient:

```
>>> "While I pondered, nearly napping".split(" ")
['While', 'I', 'pondered,', 'nearly', 'napping']
```

There is a companion `join()` method that does more or less the opposite:

```
>>> wordlist = "While I pondered, nearly napping".split(" ")
>>> " ".join(wordlist)
'While I pondered, nearly napping'
```

The `join()` takes the string whose method it is (this is the space in the example above), and reproduces it in between all the elements of the `wordlist`. Since `wordlist` was produced by using `split()` with a space, the result is the same as the original string.

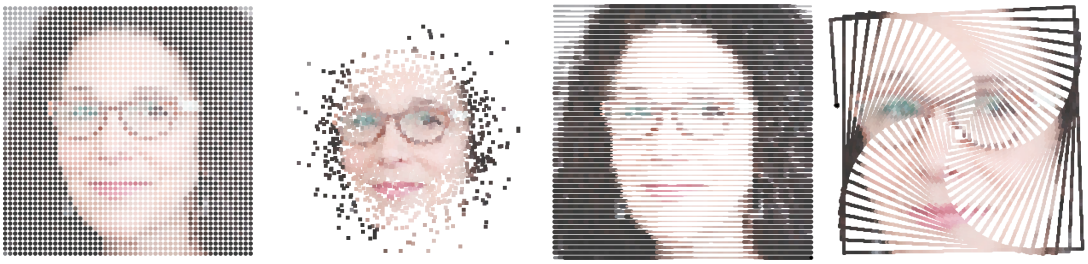
7 **Assignment 7: Rosanne image** On the class web site you will find a text file called `rosanne.txt`. It is a text representation of the photo to the right. Each line of the file contains five numbers. The first two are a horizontal (x) and vertical (y) coordinate measured from the top left corner, and they range from 0 to 99. The other three numbers are a red, green, and blue value, each one ranging from 0 to 255. It looks like this:

```
0,0,147,148,152
1,0,149,150,154
2,0,150,151,155
3,0,150,151,155
...
```

Please write a python program that reads this data from the file (don't change the file name), opens a turtle graphics window and re-creates a version of the image using turtle graphics methods. These might be drawing a line of varying width and color, or using the stamp() method of a turtle object or anything else you can think of. (See right.)

Prepare your program so that I can run it from the command line, using the #!/usr/bin/env python trick. (See page 13.)

Remember that your program must have extensive comments in it. Extra credit for interesting distortions, color changes, or portrayals. More extra credit for good use of functions.



4 Show it again: More graphics

COMPUTER GRAPHICS IS A HUGE SUBJECT. There are many books written about it, and I advise you to read them. What parts of this vast field are relevant to you will depend on whether you are interested in the graphics of a user interface, a data visualization, a virtual reality game, or something else entirely. Python is a good tool to use to explore these options, in part because it is an extension language to many graphics programs, like Maya and Blender.

We will not go into great detail in any part of graphics, but there are a few concepts worth understanding and this chapter will skip gently among those topics, touching them lightly so you can explore them later on your own.

4.1 Turtle 3D

Turtle graphics is a fundamentally two-dimensional enterprise. The turtle is crawling around on a flat plane, where each point it could travel to is identified by two coordinates. But we live in a three-dimensional world, with senses and spatial reasoning affordances tuned to that environment. Wanting to portray elements or objects from our 3D world on a screen is a natural thing to want to do. Fortunately, you can use a turtle to draw anything, so let's use one to draw three dimensions.

An object in space can be defined by a set of 3D coordinates, each of which is a set of three numbers, along with color and line information. By contrast, a drawing on a screen can be defined by a set of 2D coordinates, along with pretty much the same color and line information. In order to portray a 3D object on a 2D plane, we need a way to transform 3D coordinates into 2D coordinates.

One the class web site, you'll find a file called `cube.py`. This contains a definition of the vertices of a cube, and a small number of functions with which it can be drawn. This is an absurdly simplified form of 3D graphics, but stepping through it will help you understand the real thing.

4.1.1 Modeling

The first step in rendering any 3D model is to make the model itself, and the first part of `cube.py` simply defines a cube and some colors for its faces. You'll notice a couple of things about the definition, which is shown below. The first is that the vertices are arranged in groups of three, which might seem odd for a cube.

```
Cube = [(0, 0, 0), (1, 1, 0), (0, 1, 0),
        (0, 0, 0), (1, 0, 0), (1, 1, 0),

        (0, 0, 0), (0, 0, 1), (1, 0, 1),
        (0, 0, 0), (1, 0, 1), (1, 0, 0),

        (0, 0, 0), (0, 1, 1), (0, 0, 1),
        (0, 0, 0), (0, 1, 0), (0, 1, 1),

        (1, 1, 1), (0, 0, 1), (0, 1, 1),
        (1, 1, 1), (1, 0, 1), (0, 0, 1),

        (1, 1, 1), (0, 1, 1), (0, 1, 0),
        (1, 1, 1), (0, 1, 0), (1, 1, 0),

        (1, 1, 1), (1, 0, 0), (1, 0, 1),
        (1, 1, 1), (1, 1, 0), (1, 0, 0)]
```

Though it is certainly possible to define a shape in terms of rectangles or pentagons or dodecagons, it is typical to use triangles, which are simple and whose points always lie in the same plane. This cube is defined by twelve triangles. Notice that the origin lies at one of the corners of the cube. We refer to the space in which a model is defined as the *model space*.

Another notable thing is a little subtle. The triangles in this cube are defined so that when you look at the cube from the outside, the points fall in counter-clockwise order around the perimeter. This is an easy convention to follow, and it helps you figure out if the face to be drawn is facing towards or away from you. If it's facing away from the viewer, why bother drawing it?

The colors of each face are also defined. Typically you would have a color for each vertex of each triangle, to allow color blending on the faces, but this is the simplified version, remember?

```
CubeColors = ["red", "green", "blue", "yellow", "pink", "purple"]
```

4.1.2 Transformation

The second thing you'll notice is that the cube is only one unit wide, and is right at the origin. No problem there, but if we want to see the cube next to another cube, or being held in the hand of a

space alien, or in some other way combined with a second model, we need to be able to move it, change its size, and rotate it so it appears correctly in the scene. Another way to say this is that the scene you are building is defined in a *world space*, and you have to transform the model space coordinates to be in that world space.

The `cube.py` module has a `transform3d` function that takes a 3D point in space and applies a scale and a set of rotations to change the size and orientation of the model. (It does not *move* the model, though, see assignment 8.)

Here is that function. Notice the sneaky for loop that scales the input point. This is called a *list comprehension* for some reason.

```
def transform3d(point3d, argScale, argPitch, argRoll, argYaw):
    """
    Scales and rotates a 3d point.
    """
    # Scale and move the point.
    scaledPt = [argScale * c for c in point3d]

    # Rotate the point through the pitch, roll, and yaw.
    rotatePitch = (scaledPt[0],
                   cos(argPitch) * scaledPt[1] - sin(argPitch) * scaledPt[2],
                   sin(argPitch) * scaledPt[1] + cos(argPitch) * scaledPt[2])

    rotateRoll = (cos(argRoll) * rotatePitch[0] - sin(argRoll) * rotatePitch[1],
                  sin(argRoll) * rotatePitch[0] + cos(argRoll) * rotatePitch[1],
                  rotatePitch[2])

    rotateYaw = (cos(argYaw) * rotateRoll[0] - sin(argYaw) * rotateRoll[2],
                 rotateRoll[1],
                 sin(argYaw) * rotateRoll[0] + cos(argYaw) * rotateRoll[2])

    return rotateYaw
```

The formulas are a bit of a mess to read, but in geometry, if you have a 2D vector (x, y) and an angle θ, then the formula for rotating the vector in the plane is:

$$(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$$

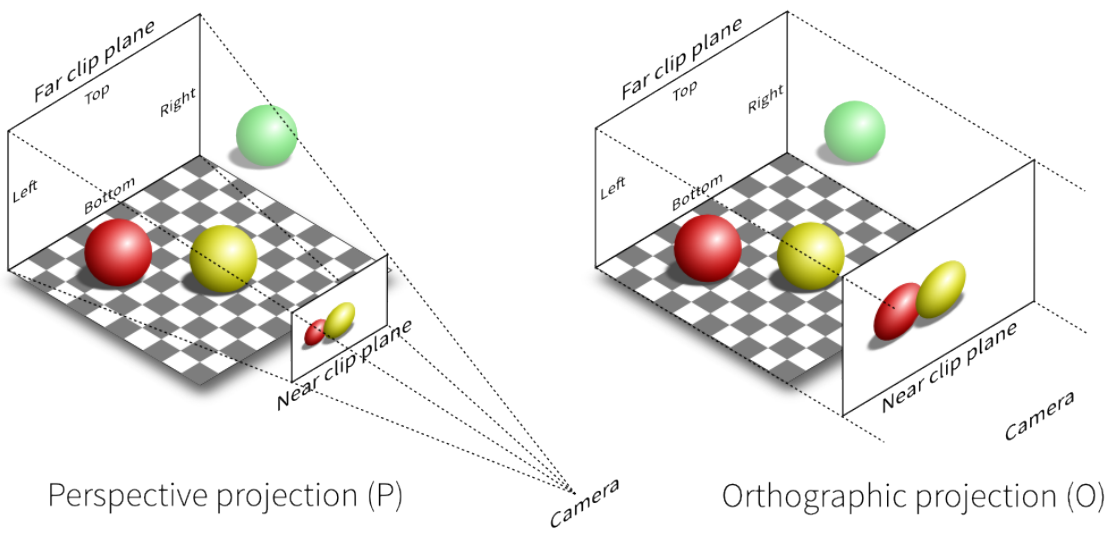
The `transform3d` function just does three rotations like this one, in the three planes of the 3D space: pitch, roll, and yaw.

How would you add an argument to this function that would also reposition the input point after it's been scaled and rotated?

4.1.3 Projection

After a point in a shape has been transformed into world space, it must be mapped onto a point on the screen. This process is called *projection*. There are two common types of projection that start with a 3D object and result in a 2D image: orthographic and perspective. You may be familiar with the terms from a mechanical drawing class.

There are two basic kinds of views of a 3D object. For the *perspective* view, imagine the infinite pyramid shape that you get by drawing lines from your eye through the corners of a screen you're looking at (or a page of paper if you're reading this the old-fashioned way). This shape is called a *frustum*, or a *view frustum*. The screen (or paper) is a plane intersecting that frustum, and the object you want to look at is projected onto that plane via lines that go from each point on the object, to your eye.



For the *orthographic* projection, the situation is similar, except the camera is not a point, but a plane, and all the perspective lines are drawn parallel to one another. In the image above, the objects are drawn on the "near clip plane", but the screen could be any plane that intersects the viewing volume.

The "near clip" and "far clip" in the image refer to the fact that in a complex scene, one usually need not draw everything, because lots of faraway things will be occluded, or too far away to see, and sometimes it's useful to do the same for objects that are too close to the camera. We say these have been "clipped" out of the scene.

The `cube.py` program has a projection function that implements an orthographic projection viewed from the Z direction. This is what you get when you simply drop the Z coordinate from the (x, y, z) points. Here is the projection function, which is pretty trivial:

```
def project3dTo2d(point3d):
    """
    This function projects a 3D input point onto the screen, using
    the offset and ratio determined during the initialization.
    """
    return (Offset + Ratio * point3d[0],
            Offset + Ratio * point3d[1])
```

The function also does a little scaling, to make the objects appear an appropriate size. The `Offset` and `Ratio` values are guessed at during the initialization step.

A perspective projection is more complex, and will usually involve transforming the world space coordinates into *camera space* coordinates before doing the projection. This just means rotating and repositioning the space so the camera is at the origin, still pointing at whatever objects it was pointing at in world space.

4.1.4 Drawing

To draw an object in our system (or in any 3D graphics system), you have to transform each point, and then project it onto the plane, and then draw it. Our cube is organized as triangles, and here's a function that uses two loops to draw the complete set of triangles that make up our cube:

```
def drawCube(verts, cols):
    # Loop through each of six cube faces.
    for face in range(6):

        # Each face has two triangles.
        for triangle in range(2):

            # Calculate the start of the first triangle.
            t = face * 6 + triangle * 3
            drawTriangle(verts[t], verts[t + 1], verts[t + 2], cols[face])
```

A function like this is useful to turn a complicated set of operations (drawing a cube) into a set of much simpler operations (drawing a single triangle). The `drawTriangle` function has only one function, to take three points, decide whether to draw them, and then draw them.

To decide whether to draw each face, we take advantage of the counter-clockwise convention established when the shape was defined. If the face vertices appear in clockwise order, we don't bother drawing it.

```
def drawTriangle(v0, v1, v2, color):
    """
    Given three vertices of a triangle, draw them with the given color.
    """
    # Calculate the transformed positions for the three input points.
    r0 = transform3d(v0, Scale, Pitch, Roll, Yaw)
    r1 = transform3d(v1, Scale, Pitch, Roll, Yaw)
    r2 = transform3d(v2, Scale, Pitch, Roll, Yaw)

    # If the triangle vertices appear in clockwise order when viewed
    # from the positive Z direction, then this condition fails.
    facing = (r1[0] - r0[0]) * (r2[1] - r0[1]) >= (r1[1] - r0[1]) * (r2[0] - r0[0])

    # If facing is true, then draw the triangle.  If not, don't bother.
    if facing:
        ThreeDTurtle.penup()
        ThreeDTurtle.fillcolor(color)
        ThreeDTurtle.begin_fill()
        ThreeDTurtle.goto(project3dTo2d(r0))
        ThreeDTurtle.pendown()
        ThreeDTurtle.goto(project3dTo2d(r1))
        ThreeDTurtle.goto(project3dTo2d(r2))
        ThreeDTurtle.goto(project3dTo2d(r0))
        ThreeDTurtle.end_fill()
```

Assignment 8: 3D Turtle Add a different geometrical shape, like a tetrahedron, cylinder, cone, or sphere to the scene. Modify the `transform3d` function to move the point, as well as scale and rotate it. Use this new function to make your second shape appear in a different place than the cube. How would you make one shape occlude the other if the two overlap?

Extra credit: Make the base colors of all the faces the same, and change them while being drawn by imagining a light source at the 3D point (5, 5, -10). How will that change the color?

4.2 Asynchronous input

Asynchronous events and handlers are another important species of input. The things that happen at a terminal window happen in order: you type something, Python types something, then it is your turn again, and so on. This is synchronous input, though no one ever calls it that. But it is distinguished from *asynchronous input*.

Asynchronous input comes at a time of the user's choice. A web browser does not prompt you for a mouse click; you just click when you feel like it. When you click, it halts execution of the program for a split-second with what is called an *interrupt* or an *event*. Control switches to a piece of code that has been identified as the *interrupt handler* or an *event handler*. This code executes quickly with some new data (like a new mouse position) and returns control to the main program, which goes ahead and draws the new mouse position or whatever it needs to change to react to the new input.

Virtually any program that interacts with a user or the world in an interesting way has to deal with asynchronous input. Your browser doesn't tell you when to click, a game doesn't stop to wait for you to shoot the alien, and a robot doesn't stop for a command.

Python provides some rudimentary asynchronous facilities within the `turtle` package. We can use them to illustrate the principle.

First, we'll need an event handler. Try this in an environment where you have a screen and a turtle working and the turtle is named harry. Here's an event handler:

```
>>> def keyEventHandler():
...     harry.left(20)
...     harry.forward(20)
```

Now we need to tell the system to use your event handler when some event happens. Use the `onkey()` method for this:

```
>>> screen = harry.getscreen()
>>> screen.onkey(keyEventHandler, "x")
>>> screen.listen()
```

You have now told your window to execute the `keyEventHandler()` function whenever the X key is pressed. Try it.

Here's another kind of event handler:

```
>>> def mouseClickHandler(x, y):
...     harry.goto(x, y)
...
>>> screen.onclick(mouseClickHandler)
```

Now go click in your window and see what happens to harry.

9 **Assignment 9: Asynchronous tic-tac-toe** Incorporate asynchronous input to your tic-tac-toe game. When you click on a square of the game, your turtle should move to the middle of the square. When you hit the X or O key, it should draw an X or O in the square. When you click outside the square — this is important — nothing should happen.

Extra credit for making Tab and Shift-Tab move from one square to its neighbor so you can play without a mouse. Tab should move one square to the right, and Shift-Tab one square to the left. Please remember the comments.

5

Rolling your own: Advanced data types

NOW THAT WE HAVE THE BASICS of a computer program down, let's look at how to organize them into something useful. There are a couple of important aspects to this: the organization of the code itself, and the organization of the functionality. These are often tied together for some particular language, but the first is very much a function of the specific language being used, and the second is more about the task at hand.

Modern programming style revolves around the idea of structuring the code around the data you will use it on. This is known as *object-oriented code* or, more recently, *data-driven design*. Other people just call it a good idea.

Python programming style revolves around the idea of modules for organization and object classes for modeling the data. We'll look at the sequence types first, and then look at how modules might help you. After that, we'll move on to look at classes and some important data types and see how to create your own object classes to solve new problems.

5.1 Sequence types

It's worth spending a little more time thinking about the compound data types because we will use them a lot. A sequence is a collection of things. But there are lots of variations of that to think about. A string is a sequence of characters and there isn't a lot more to say about that. But a list can contain different types in it, and so can a tuple. So what's the difference?

Let's imagine a database. You can think of a database as a *table* of data.[8] Here's a table you might use to keep track of students and gerbils in a class:

Gerbil name	Student	Length	Cage
Henrietta	Henry	5.5	1
Pooky	Asha	6.5	2
Rascal	Clement	5.1	3
Han	Qing	3.8	4

Using Python data types, you could hold this table in a list of tuples, like this:

```
gerbils = [('Henrietta', 'Henry', 5.5, 1),('Pooky', 'Asha', 6.5, 2),
           ('Rascal', 'Clement', 5.1, 3),('Han', 'Qing', 3.8, 4)]
```

[8] Real databases are usually composed of lots of tables.

This is a list of values, each one representing a gerbil. A gerbil might be added to the collection, or it might die, or be set loose in the gym, and you'd want to change the list of records. But you wouldn't often change any individual row. Each row is a record that represents a gerbil that you wouldn't change. Saving each row as a tuple will help remind you that these values don't change often.

On the other hand, you might add a column to this table to hold the gerbil's age. This would change each month, and so you might save *that* table as a list of lists instead. Or, more likely, you might create a second list to hold the ages. And wouldn't it be great to have a list that was indexed by the gerbil's name?

5.1.1 Dictionary

Say you have to keep a handful of values, maybe the gerbil ages in the table above. There aren't many ages in the list, so you could just keep the names and ages in two parallel lists:

```
names = ['Henrietta', 'Pooky', 'Rascal', 'Han']
ages =  [12, 13, 11, 12]
```

Then you can do this if you want to find an age:

```
>>> names = ['Henrietta', 'Pooky', 'Rascal', 'Han']
>>> def match(string):
...     for n in range(len(names)):
...             if names[n] == string:
...                     return n
...     return -1
...
>>> match('Pooky')
1
>>> match('Rascal')
2
>>> ages[match('Pooky')]
13
```

This works, but it is a little awkward, and more important, it doesn't scale well. The more entries in your list, the longer it will take to find one. If you have a list with a hundred times as many entries, it will take a hundred times longer to search it. To address this, computer scientists developed a kind of indexed array called a *hash* that uses the index value to compute a memory location and then returns the value stored at that location. This still scales with the number of entries in the array, but much more slowly. Python provides a built-in hash data type, called a *dict* or *dictionary*.

A dictionary is a collection of pairs of data, arranged so we can use the first member of the pair to look up the second member of the pair. The first member can be any Python type, and so can the second.

We define a dictionary like this:

```
age = { 'Henrietta':12, 'Pooky':13, 'Rascal': 11, 'Han':12 }
```

Now you can ask for Pooky's age like this:

```
>>> age['Pooky']
13
```

If we want to print out all the data in our two collections as a single table, we can do this:

```
for g in gerbils:
     print("{0} {1} {2} {3} {4}".format(g[0],g[1],g[2],g[3],age[g[0]]))
```

What are some other ways you could use a dictionary to organize this data?

Assignment 10: Verify Zipf's law Jean-Baptiste Éstoup, a French stenographer, noticed that word frequencies tended to be inverse to their rank. That is, the most frequent word in some large enough sample of language will usually be much more frequent than the second-most frequent, which is much more frequent than the third-most, and so on. The relationship he saw is what we call a *power law*, and if you plot word rank vs. word frequency on a log-log plot, you'll get a straight line. George Zipf, an American linguist, wrote a fair amount about this observation, and it has become known as *Zipf's law*.[9]

Your assignment is to verify Zipf's law on a novel available on the internet. (Check out gutenberg.org, for example.) Your program will need to:

1. Read the novel from a file, and count the words. This means reading each line, splitting the line, and adding the words to a dictionary object.
2. Produce a report about the resulting word frequencies. The frequency of each word is the number of times that word appears divided by the total number of words.

In order to accomplish step 1, you'll need to do away with punctuation and to regularize the case of the words. Step 2 could be done with a statistical report, or with a graph. If you make a graph, remember that a 'log-log' plot means graphing the log of the x variable against the log of the y variable.

[9] This is an example of Stigler's law of eponymy.

5.2 Classes

For a lot of problems with data you'll face, there is not a handy-dandy pre-defined solution already in Python. You'll have data that just isn't modeled well with a list or a tuple. Or maybe you just want to be able to specify the operations available for this data. Or maybe you have some data that *is* modeled well by an existing Python data type, but you want a way to explain what you're doing to your future self who will have to interpret it. All of these needs can be met by using a Python *class* to define a new data type to describe your data.

In a number of ways Python makes no distinction between the name of a variable and a function, and both can be included in a class definition.

The syntax for a class is relatively straightforward. If you have an object that contains a number and a function to run with it, it might look like this:

```
class myclass(object):
    """
    A class for calculating a logarithm.
    """

    def __init__(self, precision = 20):
        self.precision = precision

    def log(self, x):
        """
        Prints the Taylor series approximation of log(x) to the
        given precision.
        """
        if x < 0:
            print("You shouldn't ask for the log of negative numbers.")

        sum = 0
        x = float(x)
        for i in range(1, self.precision):
            sum += (((x - 1) / x)**i ) / i

        return sum
```

Here are the important features of this definition:

- We have included documentation strings. Type `myclass.__doc__` to see. Or better, type `help(myclass)`. (If you have included this definition in a module file called *mc.py*, you'll need to type `help(mc.myclass)`.)
- The `__init__()` function is used to initialize the *data members* of the class. In this case, the function sets the `precision` member, so when you do `a = myclass(10)`, you will get your log values calculated with the first ten members of the series.

- The `precision` argument of the `__init__()` method has a default value, so if you do `a = myclass()` you'll get an object a where `a.precision` is 20.
- We warn you if you're asking for a value that will give a bad result.
- We make sure that x is a floating point value, but there is no warning for that.

To use the class, we first define an object of that class, and then execute the method:

```
>>> from test import myclass # (Assuming you defined it in test.py.)
>>> a = myclass(10)
>>> a.log(2)
0.6929671999007935
>>> import math
>>> math.log(2)
0.6931471805599453
>>> b = mc.myclass(25)
>>> b.log(2)
0.6931471782613077
```

5.2.1 Static data

A class can contain data common to all the members of this class. A variable like this is called a *static member* variable. Here's a class definition with a static member:

```
class myclass(object):
    a = 12
    def __init__(self, b):
       self.b = b
```

With this class definition, any object of this class will have a member called a with a value of 12, and a member called b with a value of whatever is set for it.

```
>>> s = myclass(4)
>>> s.b
4
>>> s.a
12
>>> t = myclass(3)
>>> t.b
3
>>> t.a
12
```

Assignment 11: Flight Data On the class web site you will two text files, called `planes.txt` and `flights.txt`. The first is a list of planes, giving the name of the plane, the model, and the number of hours it has flown. It looks like this:

```
Alpha,RI-400,1412.5
Bravo,RI-600,1200.7
Charlie,RI-400,2502.1
```

Plane Alpha is a RI-400, whatever that is, and has 1412.5 hours of flight time logged. Plane Bravo is a RI-600, and so on. The second file is a record of a week's flights, giving the day of the week, the plane, the starting city and ending city, and the number of hours the flight takes. Each line looks like this:

```
2,Delta,Howport,Easyfield,1.7
2,Juliet,Howport,Abelville,2.1
```

These lines say that on day 2, airplane Delta flew from Howport to Easyfield, taking 1.7 hours, while airplane Juliet flew from Howport to Abelville, taking 2.1 hours.

Write a program using one or more classes that can read both files and report the following information:

- How many hours each plane has flown after all the flights.
- What cities each plane has visited.
- What planes have visited each city.

Extra credit: Make a map of the flights and planes. The cities live on these coordinates of a 400x400 window with the origin in the middle. You could define a turtle for each plane and reproduce the flights in different colors. How might you prevent them from drawing on top of each other?

Name	X	Y
Abelville	-30	30
Bakerburg	130	-190
Charlestown	90	-50
Dogbury	-170	-90
Easyfield	-130	30
Foxford	110	-50
Greenwich	50	150
Howport	-130	170
Itemland	-50	-190
Jigton	-70	70

6 Organizing a big project: On not getting lost

NOW THAT YOU'VE BEGUN to put together a decent-sized project, you'll have had some experience with trying to organize your code and organize your ideas. It's not actually that easy to keep things organized in a way that allows you to work on something over here without messing things up over there. Things become even more difficult when multiple people are involved, and you have to divide the work up between each other without stepping on each others' toes, or messing up each others' work.

Here are some tips and thoughts about how best to keep the various parts of your program working well and working well together.

6.1 Planning your program

Before starting to code, you are going to have to figure out what your code needs to do. This will require you to think hard about what the inputs and outputs are. If the assignment is "find some chained movie titles," try to be specific about what exactly that means. Does it mean printing them out? If so, printing out what, exactly?

It often helps to write out what is sometimes called *pseudo-code*, where you try to be explicit about what is tested and how your program is to respond. You might write something like this:

> Read a line.
> Remove punctuation from a line.
> Separate the line into separate words.
> Convert words to lower case.
> Repeat.

This gets you a general idea of what to do. To accomplish each step, you'll need to decide things like what is the state of things before and after each line. For example, before line 1, the state is pretty much a blank slate, but you'll say to yourself, read a line from where? And then realize you need a file name, have to open the file, and so on. After the first time through line 1, you'll have a line of your file, but where will it be? In some string variable, perhaps? How do you remove punctuation? Does that mean substituting it with the string `.replace()` method, or does it mean copying each character from your input string to a new output string? Making these decisions will inform the steps you need to take to turn the pseudo-code into code.

6.1.1 Modular programming style

The easiest way to keep your programs easy to understand is to keep the pieces as close to bite-size as you can. Short functions, with explicit inputs and outputs, are much easier to "read" than long complex functions where it's possible to lose the chain of meaning when you are reviewing it.

Start with little pieces you experiment with using the Python interpreter. Try different things until you get what you want without error, then pack that new knowledge into a function, label the inputs and explain the output. Write a little doc string for the function. Once you've done that, you can essentially forget about that function and move onto the next to-do.

Debugging a short program with a small number of variables is a far easier task than for a longer program with many operations and many variables. Building your project out of simple functions, made bullet-proof with careful input checking, will be far easier and more fun than trying to do everything in one large program.

6.1.2 Be careful of inputs

Being clear about the inputs and outputs of your functions will help you a lot. The syntax of declaring a function helps with that, because the arguments (the inputs) to your function are explicitly listed in the parentheses. The outputs to a function come through the `return` statement, so they are easy to identify, too. Because the clarity provided by explicit argument lists is so useful, it's generally considered bad form for a function to rely on global variables, though there are times when it can make sense.

An important precaution to take when writing your functions is not to trust the input. A user (including you) can always forget that a string argument was supposed to be a string, or mistype a file name. Your functions should anticipate the problems, and either exit (use the `return` statement to exit a function) after printing some useful error message, or raising an exception (see Section 6.4).

Here's a way to check if a file exists, without bombing the program:

```
try:
  f = open(fileName, 'r')
except IOError:
  print "I can't open a file called {}".format(fileName)
```

You might have added a `return` after the `print` statement, to end the execution of that function, or done something else to fix the problem.

Here's how to catch a `ValueError`, such as when a user types a string that is supposed to be converted to a number and can't be:

```
try:
  f = float(g)
except ValueError:
  print "{} is not convertible to a float.".format(g)
```

You can use exceptions yourself to complain to the user. For example, you could raise an exception if the user provides input that can't be converted to a string:

```
if g > 10000:
  raise ValueError('g cannot be more than 10,000')
```

This will bomb out of your program (unless you are using a `try/except`) with a `ValueError` and the given message. Make your message meaningful, please.

6.1.3 Object-oriented

Thinking about your program in terms of the data being read in and the kinds of data objects you want to manipulate is not only a good way to conceptualize the task at hand, it is usually a good way to encapsulate the functionality of your program as well. If the functions that read data into your exotic data object are methods of a class, then it becomes quite obvious where to find them. If they are just free-standing functions, it's not as clear. For a small program, this is a minor issue, but small programs often grow into larger ones, and for a large program, trying to figure out which way is up is often a challenge. Start out right, and stay that way, is the best advice.

6.1.4 Avoid premature optimization

Somewhere along the line while designing your program, you will think of a clever way to save some time in the execution. If the program is already working and running, this is probably a good thing to implement. However, if you haven't got it working yet, it's probably better to write your clever idea down and wait until later. It is frequently the case that premature optimization of a program makes it more challenging to debug it.

6.2 Modules

Python modules are just a way to save your work so it's available after you quit Python and restart it again. It's also a way to group a bunch of functionally similar pieces into a single place.

If you tried the last assignment from the previous chapter, you created a little module with your code. The module name is the same as the file name, minus the `.py`.

Try putting the following into a file called *mymod.py*:

```
def count(n):
    for i in range(n):
        print(i)
```

When you want to use the contents of your module, you issue the *import* command. By default, the import command only makes one entry into the namespace, the name of the module. You can then refer to the contents of your module by prefixing the module name to the name of the object. So if your module is called `mymod` and it contains a function called `count()`, you can refer to your function as `mymod.count()` after the import.

If too much typing annoys you, you can do this to abbreviate the namespace entry:

```
import mymod as mm
```

Now you can refer to your function as `mm.count()`.

If you still find that inconvenient, you can also do this:

```
from mymod import count
```

This does not import all of mymod.py, but searches it for the `count()` definition and only imports that. You can even do `from mymod import *`, which brings everything in. When you use the `from` form, you lose some of the organizational leverage of the module namespace, but sometimes that's ok, especially when you're just trying things out.

If you are working on your `mymod` module, and change the `mymod.py` file after you import it, you can use `reload(mymod)` to read the module again.

You can use the built-in `dir()` function to look at the contents of a module:

```
>>> import mymod
>>> dir(mymod)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'count']
```

The first five entries will appear for any module. (Take a look at each one. See if you can figure out how to add documentation for your module to the `__doc__` string.) Your `count()` function is at the end of the list.

PACKAGES AND MODULES

A note about terminology: A Python *package* is pretty much the same thing as a module. But some packages (like PIL) are so large that no one uses them by importing them entire. Instead, users will tend to import submodules piecemeal, as needed.

6.2.1 Module file as executable

One very convenient thing you can do with a module file on Linux or Mac computers is to make it double as an executable file. There are two magic lines to add to your module, to use it as a script. One we've already seen, the first line that makes your script executable. As an example, make your `mymod.py` look like this:

```
#!/usr/bin/env python3

# Definitions go here
def count(n):
    for i in range(n):
        print(i)

if __name__ == "__main__":

    # Execution of those definitions goes here.
    count(4)
```

Notice the indentation in the ending lines, which are part of the `if` statement above it.

If you do "import mymod", you'll be able to use `mymod.count()` in your python environment, and the stuff after the if statement will *not* be executed. But if you run the script at the shell prompt, all of it will execute. Do this at the shell prompt ($):

```
$ chmod +x mymod.py
$ ./mymod.py
0
1
2
3
```

(The `chmod` command makes the file executable. You only have to do this once.)

The value of doing this is that you can combine module definitions and some tests and examples of the use of those modules in the same file. This can be a very convenient way to maintain a large project.

For windows users, just put it all in one file and don't worry about the `#!` line at the top or the weird `if` statement in the middle. Linux and mac users can do this, too, if the scripting thing does not make sense yet:

```
# put class definitions here
class myfirstclass(object):
   ...

# put data definitions here
data1 = myfirstclass(...)
data2 = mysecondclass(...)

# Execute your program here.
data1.doSomething()

# put your display() methods here to print out your data
other_data.display()
```

6.3 Documentation

When starting out on a project of any ambition, it helps a *lot* to start by writing a specification for the project: what will it do, and how? With what inputs and outputs? The pseudo-code you wrote during the planning will help you figure out what should be described in your specification.

After writing a specification for the project, write one for each of the important data objects in it, and for each of the methods that data object will require. You'll need something to read in your data, something to output, processing to do in the middle, whatever. Write it all down.

If you're collaborating with other people, a specification is the way to do that. If your spec is explicit enough about inputs and outputs for the various pieces, then implementing those pieces can be easily delegated and completed independently.

When you're done with the spec, you can use the descriptions as the doc strings for the Python functions. As you write the actual code, you will see places where you were wrong in your descriptions, or where you left something out. Be sure to go back and change the descriptions when you notice the problem. There will never be another time when you are as familiar with the code as when you are first writing it, so writing good documentation only gets harder with time.

6.4 Errors and error handling

By now, you're probably quite familiar with the errors you get from running your functions. There are two different categories of error with any computer program. The first, a *syntax error*, simply implies that you have spelled something wrong, or forgotten a colon, or otherwise typed something that Python simply does not understand. Usually the errors you get will have some kind of hint about what is the matter. Of course, this is Python's interpretation of what's the matter, so your mileage may vary.

Here is a pretty common syntax error, and Python is complaining that there is no colon to mark the end of the `for` statement and the beginning of the loop's block.

```
>>> for c in collection
  File "<stdin>", line 1
    for c in collection
                       ^
SyntaxError: invalid syntax
```

Syntax errors are pretty much fatal. Python simply doesn't know what to do, so it rolls over with its tongue out and goes no further no matter what you do.

The other kind of error is called a *run-time error* or an *exception*, and this involves Python simply not knowing what to do in some situation. For example, maybe someone calls your function with input you didn't anticipate, or you forgot to check whether some list had members before executing a `pop()` on it.

First of all, you should always check the arguments to any function you write, and second, you should check whether a list is empty before doing a `pop()`. Sometimes, however, situations are just too complex to anticipate all the possible errors. Python can recover from exceptions if you show it how with a `try` and `except`. Here is an example:

```
>>> while True:
...     try:
...         input("Type valid input: ")
...         break
...     except:
...         print("That wasn't valid input.")
```

You can also choose to catch only certain exceptions (errors), and the `try` can also have an `else` clause for code to be executed if there are *not* any exceptions while executing the `try` clause.

```
try:
    input("Type valid input: ")
except ValueError:
    print("That wasn't valid input.")
```

```
except:
    print("Something else was the matter.")
else:
    print("That was valid input.")
```

There is also a `raise` command to cause an exception to be raised. You can use this for testing, or you can also use it to raise legitimate error conditions. As with everything else we are covering, you can read much more about this at python.org.

6.5 Version control

One thing you'll notice as you work on projects is that occasionally you'll wish you had saved an earlier version of your code. Perhaps you've made some changes and they didn't work and you want to try again, but when you delete what you thought were all the changes, you're left with something that doesn't work any more. These are problems that all programmers face, and that's why they invented version control systems.

Version control is a way to save intermediate versions of some program, so you can figure out what changed from one day to the next, and who changed it. You can rewind changes back to a saved version if the changes seem useless and most systems will allow you to incorporate changes that other people have made to the same programs.

There is no requirement in this class to use version control, but it can save valuable time on a big project, or on a collaborative project. Git is a popular free version control program, that comes with extensive support via github.com or bitbucket.com. Git is a complex program with a lot of options and functionality, but through github you can find graphical interfaces for Windows, Macs, and Linux machines that make it relatively painless to use.

There are good tutorials about Git at the github web site, and I encourage you to explore them.

Assignment 12: Final project Describe your final project. Use a full page of text to describe the program and what it will do. Fill out these sections:

- Purpose (Doesn't matter if it's a silly purpose.)
- Inputs (What source information will you use?)
- Outputs (What will the program produce?)
- Method (How will you transform the inputs into outputs?)

  Use pseudo-code to describe this.
- Necessary objects (What objects will you need to define?)

Exotica: Abstract data types

THERE ARE A BUNCH of different abstract data types that are in common use to solve different kinds of computing problems in an elegant way. There are many different ways to implement some of these data types in Python. In this chapter we'll look at a few of them.

7.1 Linked list

A *linked list* is an ordered collection of objects where each member of the collection points to the next. Here is a trivial definition of a linked list in Python:

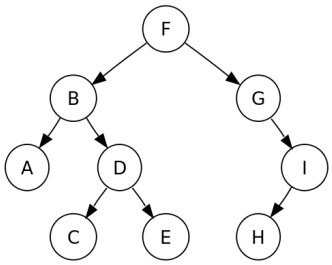
```
class Node(object):

    def __init__(self, data=None, nextNode=None):
        self.data = data
        self.nextNode = nextNode

    def getData(self):
        return self.data

    def getNext(self):
        return self.nextNode
```

If I have some object `p` that is a member of this list, then `p.getData()` is the data belonging to that item in the list, and `p.getnext()` gets me the next item in the list.

if you want to add an item to the list after `p`, you can just create a new node for the list, point to the same `nextnode` as `p` and point `p.nextnode` to your new node. this can be *much* quicker than using python's list.insert() method, especially for big lists.

One thing that's important to understand is that for any linked list, there isn't necessarily an object called a "linked list," only nodes that together make one. the nodes should have a way to link to their neighbor, and a way to change that link. the program that uses them should have a way to add and subtract a node.

7.2 Trees



We think of a linked list as sort of like a collection of toys, all linked together, or maybe pearls or something that lines up in a row. A *tree* is sort of like a linked list, but where a node in the list might point to multiple other nodes. We tend to picture a tree as representing a hierarchy, where you have a *parent node* pointing to one or more *child nodes*, with a *root node* at the top of the hierarchy.

The structure of folders on a computer disk is a perfect example of a tree structure. What other kinds of data might be modeled this way?

Here's a trivial tree structure:

```
class node(object):

    def __init__(self, data=none, leftchild=none, rightchild=none):
        self.data = data
        self.leftchild = leftchild
        self.rightchild = rightchild

    def getdata(self):
        return self.data

    def getleftchild(self):
        return self.leftchild

    def getrightchild(self):
        return self.rightchild
```

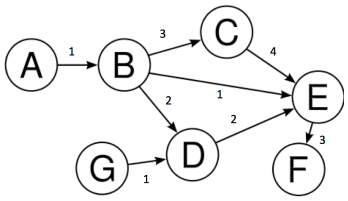
This tree has nodes with only two children each. A more elaborate tree might have a list of child nodes, so there can be an arbitrary number of children. Trees can be singly-connected, where a parent is linked to its children, but not vice versa, or doubly-connected, where the children are also linked to the parent.

Like the linked list, there is no data type that is a tree. Rather, a tree is the name of the structure these nodes make together.

A tree can be made of different kinds of nodes, too. For example, you might have a warehouse which contains shelves which contain boxes which contain files. You might have different objects for each of the different levels of the tree: the warehouse object points to one or more shelf objects, each of which point to one or more box objects, and so on.

7.3 Graph

A *graph* is sort of like a tree, but with no root node. We talk about a graph being made up of *nodes* and the *edges* that connect them. They are typically used to portray some kind of network, and sometimes the connections between the nodes will have a direction, or a *weight*. You might model a map with a graph, or maybe a social network or telephone lines, or who knows what.



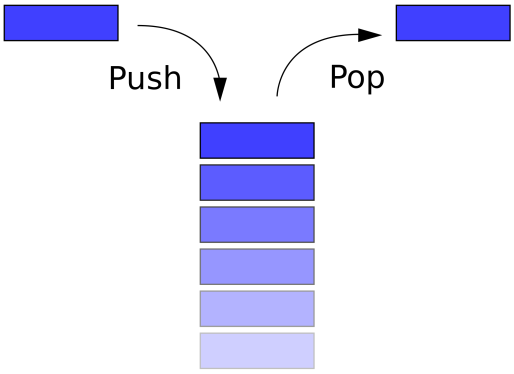
The implementation of a graph would generally be quite similar to a tree, but it might include an indication of direction and possibly other data about any specific edge.

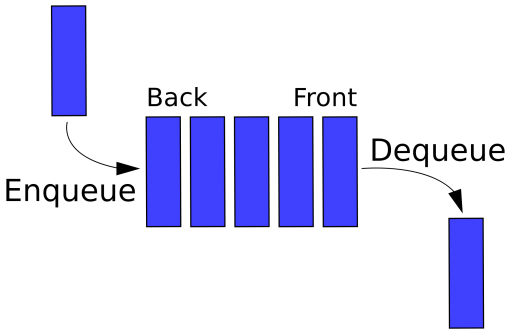
7.4 Stacks and queues

A *stack* and a *queue* are ways to keep track of a changing ordering of objects. A stack can be envisioned as a stack of playing cards, and you can either take a card from the top of the stack, or put a new card on top. We refer to adding a new card as a *push*, as in "pushing" the card onto the stack. Removing the top card is a *pop* operation: we "pop" the card off the stack. Thinking of a Pez dispenser might help the words make more sense.

A queue is a similar idea, but it's more like a line of people waiting for something. You join the line at the end, and exit the line at the front when the bank teller is free to help the next customer. We talk about an *enqueue* and *dequeue* operation for queues, though you'll sometimes hear people use "pop" for dequeue.

A stack structure is also sometimes called *LIFO* for "last in, first out," and you'll often see a queue called *FIFO* for "first in, first out." Here is a stack on the left and a queue on the right:





You can implement a stack or a queue with a python list (lists even have a `pop()` method, though you have to use `append()` for a "push"), or you can write your own class for one. A linked list is a pretty good way to make a queue, since adding an element to the end is easy, and so is dropping one from the beginning.

A stack will require a `push()` and `pop()` function to add an element to the stack and to remove it. If you want to see what's going on under the covers, you should probably implement a `display()` command to print out the stack contents.[10]

Here's a trivial implementation of a stack:

```
class stack(object):
    """
    a simple stack.
    """
    def __init__(self, x):
        """
        the stack starts life as an empty list.  if there is an
        argument for the initialize, we push it onto the stack.
        """
        self.stack = []
        self.stack.append(x)

    def push(self, x):
        self.stack.append(x)

    def pop(self, x):
        return self.stack.pop()

    def disp(self):
        """
        this prints stack items in reverse order.
        """
        for x in self.stack:
            print(x)
```

This will work, but it will display the stack in what many people would consider reverse order. You can reverse the order of a list with the `reverse()` method, but that changes the list order. How would you print them out in a more sensible order without changing the stack?

A queue will need an `enqueue()` and a `dequeue()` function, and should have a `display()` method as well, at least for the debugging phase.

[10] Don't call it `print()`. why not?

7.5 Implementation

To implement any of these data types in Python, you will need not only the outline of how the data type appears, but also a way to get data into it. A tree structure defines a relationship between nodes. To make it a practical reality, you'll need some way to create that structure. Similarly, you'll also want a way to *use* your new structure. That is, a class to contain some new data structure will almost always also include a method to find some object in the structure, or to print out the structure.

For example, we could create a tree structure with an object like this:

```
class treeNode(object):
    def __init__(self, cargo):
        self.cargo = cargo
        self.children = []
```

Each object in the tree has some cargo—a text string, for example—and one or more children, stored in a list, presumably of other treeNode objects.

By itself, this is not terribly useful. You'll need a way to add data to the tree. Here is a version of a tree with an `addChild()` method to add nodes to the tree, and a `disp()` method for printing it out.

```
class treeNode(object):
    def __init__(self, cargo):
        self.cargo = cargo
        self.children = []

    def add(self, testCargo, newChild):
        """Adds a child node to a parent node that matches testCargo."""
        if testCargo == self.cargo:
            self.children.append(treeNode(newChild))
        else:
            for child in self.children:
                child.add(testCargo, newChild)

    def disp(self, lev):
        print "{0}{1}".format("  "*lev, self.cargo)
        for child in self.children:
            child.disp(lev + 1)
```

We have added a method to add children to the tree, and another method to print out a node and its children. Now we can create a tree this way:

```
>>> t = treeNode("root")
```

```
>>> t.disp(0)
root
>>> t.add("root","child1")
>>> t.add("root","child2")
>>> t.disp(0)
root
  child1
  child2
>>> t.add("child1","grandchild1")
>>> t.add("child1","grandchild2")
>>> t.add("child2","grandchild3")
>>> t.disp(0)
root
  child1
    grandchild1
    grandchild2
  child2
    grandchild3
```

You could create a `tree` class to hold the root node, and have its own display function, which might make managing things easier.

```
class tree(object):
    def __init__(self, rootCargo):
        self.root = treeNode(rootCargo)

    def add(self, testCargo, newChild):
        self.root.add(testCargo, newChild)

    def disp(self):
        self.root.disp(0)
```

Now you can manage the tree in a somewhat simpler fashion:

```
>>> t = tree("root")
>>> t.add("root", "child1")
>>> t.add("root", "child2")
>>> t.add("child1","grandchild1")
>>> t.add("child1","grandchild2")
>>> t.add("child2","grandchild3")
>>> t.disp()
root
  child1
    grandchild1
    grandchild2
  child2
```

```
grandchild3
```

Assignment 13: Iron Man Who Fell To Earthquake Write a program to find linked titles from a list of movies you can find at the course web site: `sgouros.com/cs-fav/movie-titles.txt`. The program should pick a random title from the list, and see if the last word from the title matches the first word of any other title. It can then check the last word of the combined title to see if it matches the first word of any other title, and so on.

You will need:

1. An object to hold a movie title, including the data input step.
2. A way to ignore punctuation, probably on the input step.
3. A way to ignore unwanted words (words and dates in parentheses, articles at the beginning of a title).
4. A function that runs the search itself.

Extra credit: It will help if you have a `display()` method that will print the output in a nice way. Maybe something like this:

```
Dawn of the Dead (2004)
            Dead Man
                 Man of the Year
                            Year One
                                 One Fine Day
                                          Day the Earth Stood Still, The
dawn of the dead man of the year one fine day the earth stood still
```

Extra credit: This search can go on forever since there are infinite loops possible, such as with titles like *Run Lola Run* or the combination of *High School Musical 3* and *3 O'Clock High*. See if your code can avoid those pitfalls, perhaps by only using a title once in any chain.

8 Computer Babel: What are other languages good for?

BY NOW, you've all had a chance to appreciate the charms of Python. To a large extent, Python is the current pinnacle of thinking about computer languages: it has user-definable objects, you can throw around functions and data in much the same way, it has advanced execution control that we haven't even covered yet, and much more. But there are still lots of other languages out there. What are they good for? Why do they persist?

To a large extent, a lot of languages persist because no one wants to rewrite programs that are already written. Legacy code is a huge issue for some companies (and governments) because there are huge programs written. Airline reservation systems, for example, are still running code written in the 1960s, patched and updated thousands of times since, but still chugging along. FORTRAN persists because sometime in the 1960s, engineers got some working models of fluid dynamics to work in FORTRAN, and no one wants to do it again.

But there are other reasons. Javascript was an easy-to-add feature of web browsers, good for dopey little tasks like making pieces of a web page blink or reload. But the language grew and people added features, and it stayed universal, more or less, so now all modern web browsers incorporate a Javascript interpreter and that's a huge amount of what web development is about.

Here is a short review of computer languages you are likely to run across. Think of this as a field guide to languages. What you might find interesting or amusing about them is how similar they look to each other at first glance. The spelling of an if statement in Java, C++, Processing, and Javascript is pretty much the same. The curly brackets and parentheses come in the same place, there are semicolons to end the lines, and the same set of conditions. And yet, those similarities mask profound differences in the management of memory and the variable names that point to it.

8.1 Compiled

One distinction among languages worth understanding is the difference between a *compiler* and an *interpreter*. A compiled language is transformed into machine code, which can be run directly by the computer. Compiled languages are used for speed and power, and generally involve a fairly steep learning curve.

C/C++ C, originally written in the 1970s, is still among the best choice for speed on machines ranging from Arduinos to super-computers. The language gives you access to the machine it's running on at a very low level, which is a boon for flexibility and getting code to run fast, and less of a boon since it's hard to make code portable across machines and operating systems. When you count the variants out there, it remains the most popular programming

language.

C++ is basically an object-oriented C, or C with classes, but that difference alone is enough to make it seem like a completely different language, even if valid C programs will compile with a C++ compiler.

C# This is a rewrite of C++ meant to incorporate the lessons learned in Java. It's mostly a Microsoft thing, and is integrally tied up with their .NET architecture.

Objective C Another rewrite of C++, but meant to make it more flexible, rather than more Java-like. Integrally tied up with Apple products and the IOS (phone) operating system.

Java Java was created as an attempt to rewrite C++ for machine-independence and to get the class thing right. Java runs equally well on many different machines and operating systems, though because of that, it runs a bit more slowly than a comparable C/C++ program. The class system in Java is more orderly and easier to use in many ways than the C++ class system. Java is widely used, though slightly less widely used than all the C/C++ variants.

FORTRAN This is an archaic language, the language of science in the 1960s and 1970s. There is some legacy code around and you will likely never see it. It is notable because each line had to begin with six spaces, unless there was an address there. In other words, it was the last language before Python where invisible white space had meaning.

8.2 Interpreted

An interpreted language does not run directly on a computer's hardware. Rather it is "interpreted" by some other program that makes the computer dance for you. Python is an interpreted language; the way to run a Python program is by feeding your program to the Python interpreter, which reads the program and does what it says.

Here's a short and random (i.e. incomplete) list of interpreted languages you are likely to hear about. The wikipedia pages for these languages contain short snippets of code you can use to compare their syntax.

Python Python is widely-used scripting language, used as an extension language by many programs. It has a relatively simple syntax, and supports objects and classes easily.

BASIC The "Beginners All-purpose Symbolic Instruction Code" was designed to be a simplified language for people just learning about programming. It's very, well, basic and was rarely used for much at all before Microsoft chose it as the framework for the macro language for Word and Excel. So now there's Visual Basic, with which you can write programs for word processing and spreadsheets.

Javascript This is the go-to web programming language for code that will be executed on the user's computer. It is a sprawling and powerful language in which it is possible to make a big mess. But if you are disciplined in your approach to a problem, it can be a powerful

tool to make all kinds of graphics and web magic happen. It has precisely nothing at all to do with Java, except for the first four letters of its name. That their names are so alike was more about marketing than anything else.

Ruby Ruby is an object-oriented language, built to be that way from the beginning. The object orientation of Python is sort of an add-on, but in Ruby it was build in from the beginning. Ruby is elegant and fast, but it has not caught on as widely as Python and Javascript. The web development framework called "Ruby on Rails" has proven popular, though, and has produced a surge of recent interest in Ruby.

Perl Perl is a language with perfectly dreadful syntax, and lots of shortcut characters that can leave you completely unable to interpret a program you wrote fifteen minutes before. But it's expressive and can make complicated tasks relatively simple. The biggest reason people use it, though is because its archive (`cpan.org`) is nonpareil and someone already created and filed pretty much whatever short and mechanical task you might want to accomplish.

PHP Currently the go-to web programming language for code that will be executed on the server. It's especially useful for accessing databases in a flexible and friendly way. Its syntax is vaguely Perl-ish, which reflects its heritage.

SQL This is used to make queries for database systems. You can use it to construct complicated queries like "give me all the students whose name starts with P and who are over the age of 20 and who have no sister or brother and live in Rhode Island." Most modern database applications don't write SQL queries directly, but use objects defined in some other language to construct them.

Processing, Wiring These look like C, but are meant as easy-to-learn languages. Processing is optimized for computer graphics and Wiring for control of microcontrollers. They are good learning languages, though maybe not great for big projects. You'll use Wiring if you want to program on an Arduino.

R language R is a peculiar interpreted language that has very few merits as a language except that it is optimized for statistical analysis and graphics. If you want to make cool graphical output that looks clear and authoritative, it is an obvious place to go. Same thing if you have some sophisticated statistical technique you want to turn loose on your data.

Scheme, Lisp Lisp is a language from the 1960s with features that weren't widely used until 40 years later when they were included in Java and Python. It has a syntax that no one likes, so it has fallen from favor. But Lisp programmers are always saying maddening things like we were doing object oriented programming long before it was cool, and the maddening part is that they're completely right. Scheme is a variant of Lisp that looks similar (lots of parentheses (nested, even)) that you might run across because it is an extension language to programs like Gimp.

Shells A *shell* is a kind of interpreted language that is worth a closer look. A shell is used to make commands directly to a computer's operating system, and the good thing about them is that you can write little (or not-so-little) programs in that language. So if you have a hundred files that need to be processed by ImageMagick, for example, you can click a hundred times to make that happen, or write a little loop in a shell script to do it a hundred times for you.

8.3 Declarative

There is another category of languages out there that is important to know about. We have been learning about procedural programming, which means writing recipes about what to *do*. A procedure involves steps: step one, do this, step two, do that, step three, do that other thing. A declarative language describes what something *is*. HTML is a widely used declarative language, and is used to describe a web page using "tags" to describe "elements" of a web page.

HTML looks like this:

```
<html>
  <body>
    <h1>My web page</h1>
    <p>Some text on my web page.</p>
  </body>
</html>
```

This uses the tags h1, p, body, and html to describe elements describing a web page with a header and a little bit of text.

HTML is a special case of something called SGML, which stands for Standard Generalized Markup Language, which is mostly the format that uses the open (`<element>`) and close (`</element>`) tags to mark an element. XML is a set of languages widely used to describe things in a similar fashion. It is set up as a flexible language that can be used to describe lots of different things, so there are thousands of variants.

9

What is Computer Science?

9.1 AI

9.2 Robotics

9.3 Modeling

9.4 Algorithms, auctions, etc

9.5 Communication theory

10 Reference Section

These are just a few miscellaneous things about Python and other stuff, recorded here to be useful to people who might need them. They are not essential for this class, but students might find them interesting, useful, or both.

10.1 Standard Python modules

There are a lot of modules that come with Python, and are used for all kinds of different things. There is a `math` module with definitions of math functions, and a `sys` module for functions having to do with the computer system you're working on, and an `os` module for things having to do with the operating system, and so on. It is much too long a list to include here, but the Python web site has lots of help to offer (http://python.org).

Python comes with a `pip` utility for installing and managing packages. When configured correctly, this will download and install Python packages, ready to use. At the command line (Mac and Linux) try typing `pip help` and see what it suggests. Try `pip list` to see what packages are already installed on your system.

When you type `import some_module`, Python searches for the module in a given set of directories. Python's method of searching for modules is pretty elaborate, but that can make it confusing. My advice is to ignore all the other advice and follow mine, here.

The search "path" is a list of directories in which Python searches for a module. You can see it with the `sys` module. Try this: type `import sys`, then examine the list called `sys.path`. To add directories to this path, you can `import site`, and then look at `site.USER_SITE`. This will show you a directory name. A file in that directory that ends in `.pth` can be used to add directories to your path.

A file called *mypath.pth* containing the following will make Python look in a directory called */users/me/python2.7/site-packages* for modules:

```
import sys; sys.path.insert(1, "/users/me/python2.7/site-packages");
```

The first element in the `sys.path` list is special, so it's advisable to insert your additions as the second element, which the above example does. (The insert() method inserts the element before the given index. In this case the index is 1, which is the second element, so the directory given will wind up second in the list. If this is confusing, try it out.) If there are two modules of the same name on your computer, Python will use the one it finds first.

NOTE
One flaw with the `pip` utility is that it sometimes installs files in a directory that is not on the Python path. You can see the directory where a module has been installed with `pip show`, and then use the instructions above to make sure that directory is part of the module path.

10.2 Python objects

Reading and writing text is convenient, but it's sometimes even more useful to read and write Python objects directly. There are a couple of ways to do this. You can use the Python `pickle` module if you're only going to be reading data that you wrote, or use the `JSON` module if you're going to be trading these with other people. The pickle files are more versatile, and can encode a wider range of data types, while the JSON files are more secure and human-readable, though they can encode a smaller range of data types.

10.2 Pickle

Here's how to pickle a dictionary:

```
>>> import pickle
>>> poems_i_like = { "Poe": "The Raven", "Thayer": "Casey at the Bat"}
>>> pickle.dump( poems_i_like, open( "poems.p", "wb" ) )
```

And here's how to read it back again:

```
>>> import pickle
>>> poems_i_like = pickle.load( open( "poems.p", "rb" ) )
```

The *poems.p* file will not contain anything you can read or edit, but it will hold a true copy of your dictionary, list, string, and so on. You cannot pickle functions or your own classes. But you *can* write a `pickle()` method for your classes that will pickle all the data values in the class into some file, and a companion `depickle()` method for reading them back out again.

```
import pickle
class myclass(object):
    def __init__(self, data):
        self.data = data
    def pickle(self, file):
        pickle.dump(self.data, open(file, "wb"))
    def depickle(self, file):
        self.data = pickle.load( open(file, "rb"))
```

With this definition imported as `tp`, you can do this:

```
>>> a = tp.myclass([1,2,4,8])
>>> a.data
[1, 2, 4, 8]
>>> a.pickle("picklefile")
>>> b = tp.myclass(["hello"])
>>> b.data
['hello']
>>> b.depickle("picklefile")
>>> b.data
[1, 2, 4, 8]
```

The `b` object has read the data from `a` via the file called *picklefile*.

CAUTION

Your mother's advice was good: don't accept pickles from strangers. This is a basic security precaution. Pickle files can contain malicious code, and cannot be considered secure if they come from an unknown source.

10.2 JSON

JSON is an internet-standard method for communicating data.[11] Storing Python values in a text file can be thought of as communicating the data values from yourself into the future, so this is an approved application.

The `json` module is a little more limited than the older `pickle` module, but it has some claim to universality, which many programmers find compensates handsomely, and you can peek into the file unaided, which makes it much more secure.

The module has a `json.dumps()` method for converting an object to its JSON equivalent.

```
>>> import json
>>> poems_i_like = { "Poe": "The Raven", "Thayer": "Casey at the Bat"}
>>> json.dumps(poems_i_like)
'{"Thayer": "Casey at the Bat", "Poe": "The Raven"}'
>>> json.dumps(poems_i_like, sort_keys=True)
'{"Poe": "The Raven", "Thayer": "Casey at the Bat"}'
```

A dictionary doesn't look so different as a JSON-encoded value than it did as the Python definition, but it is part of a standard vocabulary, and that's what makes the difference.

[11] It stands for "JavaScript Object Notation" which tells you something about its roots as a part of Javascript, an almost universally available scripting language for web applications.

Unlike the `pickle` module, `json` is just about translating Python objects into strings that can be stored and retrieved from text files. It does not do the file opening and closing for you, so you have to do something like this:

```
>>> f = open('testfile', 'w')
>>> f.write(json.dumps(poems_i_like))
>>> f.close()
```

To read it back, use the `json.loads()` method. Combine with the file management for something like this:

```
>>> f = open("testfile", "r")
>>> p = json.loads(f.read())
>>> p
{u'Thayer': u'Casey at the Bat', u'Poe': u'The Raven'}
```

The "u" in the strings means that the round-trip into the JSON format has made the strings into unicode characters. The `json.dumps()` function has a number of arguments you can use to make the output easier for a person to read (`sort_keys`, `indent`, etc.), and they make no difference to the `json.loads()` function.

The `json` module has a number of other features, including an encoder and decoder class for creating specialized objects with their own encoding and decoding mechanisms.

10.3 Command-line commands

Though its importance to the world of computing is greatly diminished from a decade or two ago, it is still possible to control most computers with typed input and output, also called command-line control or *shell* commands. The shell[12] is a program that listens to typed input and responds accordingly by running the programs indicated by the user input. It is an interpreter, very like Python in some respects, but designed specifically to run other programs.

On the Mac, a shell is usually made available through a program called Terminal, though there are several others available. Linux has several, such as Xterm, wxTerm, and more. These are the programs that create the window in which text is entered, and run the shell to interpret the typed input and produce responses.

Python, like the shell, is a scripting language, executed one line at a time. Though Python often appears in fancier contexts, controlling graphics programs and so on, at root it is simply a list of commands to be executed one at a time. It makes sense to run simple Python programs through a shell.

[12] Actually "a" shell; there are many different shells out there.

To use a shell in a Mac or Linux context (also Cygwin on Windows), it helps to know a small number of important commands:

cd *directory* "Change directory" Moves the working directory to the given directory. (~ is shorthand for your home directory.)

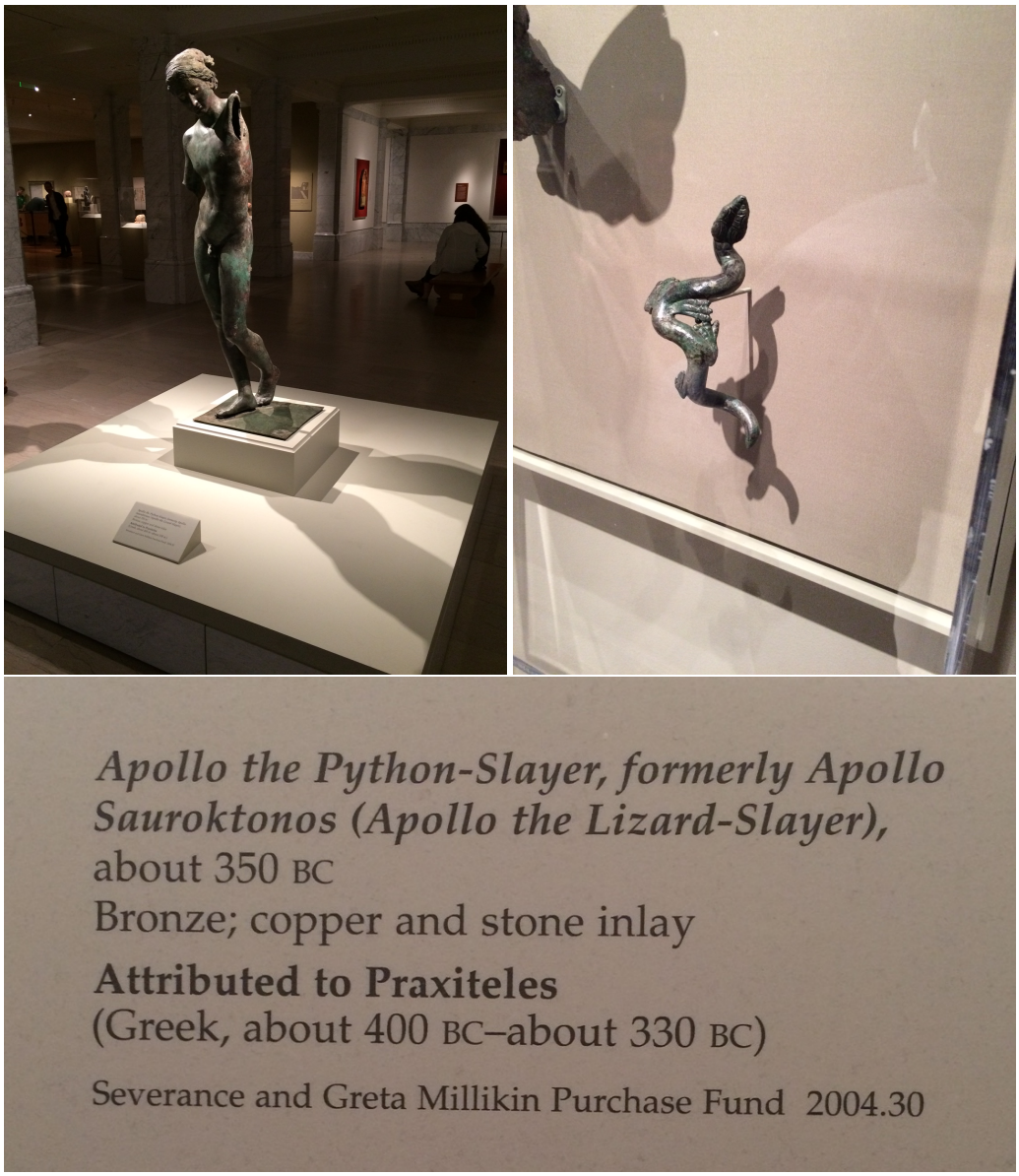
pwd "Print working directory" Identifies the working directory.

ls *directory* "List" Show the contents of a directory. If you leave the directory off, you'll get the contents of the working directory.

chmod *arg file* "Change permissions" You have to tell the shell it's ok to execute a file by giving it "execute" permission like this: `chmod +x testfile.py`.

rm *filename* "Remove" a file. Delete it.

There are lots of others, but these will do.



From the Cleveland Museum of Art.

Index

Index of assignments