1 Computational Thinking

If you were trapped on a desert island, why would you want to be a computer scientist? To be honest, if you were alone, you probably would be better served being a herbologist or handy at spear fishing. On the other hand, if you were one of a thousand people on a desert island, it would be good if at least one of you was a computer scientist. Having someone who understood how to design algorithms and processes would help to ensure efficient and fair distribution of resources, create fast communication protocols in case of emergencies, and design methods of communication that would be likely to reach rescuers.

Computer science is about solving problems computationally. To solve a problem computationally means that you can write out a series of steps which, if followed precisely, generates a solution to a problem. One benefit of being able to solve a problem computationally is that we can probably build a machine to do it. Problems such as addition, subtraction, multiplication, and division are examples of computational problems. So are problems like packing boxes in a truck, detecting faces in an image, or calculating who to play in a fantasy football team each week. All of these problems have algorithms which, if carefully described and followed precisely, solve the problem in a way that is useful.

There are significant differences between the problems listed above, however, and the kinds of solutions they require. Furthermore, there are always many different algorithms for solving a given problem. How do we analyze algorithms to determine which one is a better solution? What does it mean for an algorithm to be better? Some algorithms are faster than others. Some algorithms require more resources like storage and memory. Some algorithms will solve a problem perfectly, but won't return a solution until the sun has swallowed the earth several billion years from now. Computer scientists have developed a set of methods and theory for trying to answer these questions.

As an example, we can generate two algorithms for multiplying positive integers if we have the capability to add and subtract unsigned binary numbers.

What is an unsigned binary number? An unsigned binary number is simply a base 2 representation of the non-negative integers (0 and up). Each digit in a binary number must be either a one or a zero. Just as the digits in a base 10 number are the digit multiplied by a power of 10, so the digits in a binary number are the digit (0 or 1) multiplied by a power of 2.

$$\begin{aligned} 42 &= 101010b \\ &= 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 \\ &= 32 + 0 + 8 + 0 + 2 + 0 \\ &= 42 \end{aligned} \tag{1}$$

There are also ways of representing negative numbers and floating point numbers using binary. Those methods are generally covered in a digital logic or computer architecture course.

Algorithms for multiplying non-negative binary numbers

- Algorithm 1: Given the multiplication $A \times B$ where A and B are non-negative binary numbers

```
Let C start with the value zero
While B is not zero
    Add A to C
    Subtract 1 from B
Return the value of C
```

- Algorithm 2: Given the multiplication $A \times B$ where A and B are non-negative binary numbers

```
Let C start with the value zero
For each digit of B from left to right
    Shift C left by one position
    If the digit is 1
        Add A to C
Return the value of C
```

Example: 11×5

If both numbers are 8 digit binary numbers ($11 = 00001101b$ and $5 = 00000101b$) then the first algorithm goes through its loop five times, while the second algorithm goes through its loop eight times. Which algorithm is faster?

- Will algorithm 1 always execute its loop only five times?
- Will algorithm 2 always execute its loop eight times?
- How many actual additions does each algorithm execute in this example?
- How can we describe the complexity of these two algorithms in more generic terms?
- Will these algorithms always run fast enough to be useful?

These are examples of questions that are important to understand when we are developing algorithms for real world problems. The answers are not always obvious.

Consider the task of packing boxes into a truck. If the boxes are all the same size and shape, the problem is not difficult because the order in which the boxes are placed into the truck doesn't matter. But what if the boxes are all different sizes? Which box should go in first? Which one should go in second? How should they be arranged? If there are a small number of boxes, then it might not matter how they are put into the truck, and any arrangement will work. If there are obviously many more boxes than can possibly fit in the truck, then we know the task can't be done. But in that in-between case it's not clear what the answer is, or how the boxes should be packed.

It turns out that, as far as any computer scientist has been able to determine, the only algorithm that can guarantee finding the optimal packing strategy is to try all of the possible arrangements. Unfortunately, that means we may not be able to discover the answer in time for it to be useful. There may be so many combinations to test that the sun may have swallowed the earth before the algorithm is complete.

There are, however, algorithms that have a good chance of finding a solution that run much, much faster. Such algorithms try to find good solutions to the problem without requiring that it be the absolute best solution. Computer scientists have studied problems like box packing to figure out just how good these approximate algorithms can be. In many cases they can show that the good solutions generated quickly will always be within a certain tolerance of the absolute best possible solution. For real applications that have to produce answers in useful time, that may be good enough.

There are also problems like choosing which players to play each week in fantasy football that will obviously never be 100% correct because they are trying to predict the future. A similar problem is trying to predict the actions of a stock, or the stock market as a whole. The goal with algorithms that deal with uncertainty is to provide an answer that is better than chance (or better than someone else's algorithm) over the long term. These algorithms not only need to be efficient and run fast, but they also need to incorporate uncertainty and probability into their calculations. One reason computer scientists are in high demand, especially in the financial sector, is that they understand how to write algorithms that handle uncertainty and that can learn how to predict the future from data about the past. Given the number of trades based on the decisions of computer models, they are clearly working well enough to make money.

As argued by Jeannette Wing, former head of CMU CS, computational thinking is cross-cutting, enabling, and increasingly important in our society. It allows us to solve complex problems, much as engineering methods allow an engineer to build a complex system. Computer scientists deal not only with real systems, however, but also virtual ones. Virtual systems are not limited by gravity or physics, only by the realities of computation. If we understand how to design and analyze algorithms, then we understand what is possible.

1.1 Abstraction and Computing

When analyzing very big, complex problems, one of the most important tools is abstraction. Abstraction is the process of representing something complex as something simpler, but maintaining the essential qualities of the original. Humans use abstraction all the time to generate more concise, informative descriptions of the world.

For example, when your friend asks what you did this morning, you don't generally give a description of every step you took from the time you got up. Instead, you abstract collections of actions into short descriptions: "I got ready", "I got a really nice cup of coffee", and "I slogged through the snow to get to class". These abstractions get across the main points of your morning that you wish to convey without cluttering the conversation with things like, "and then I took my 1346th step with my left foot and squished into the snow about 6 inches". Your friend, for example, might like to know where to get good coffee. But they're not going to hang around to find out if you have to go through 2000 footsteps to get there.

Finding the right level of abstraction is important.

- If the amount of abstraction is too great, then the abstraction is difficult to use outside of a specific context (prisoner joke telling)
- If the amount of abstraction is too small, then the details confuse the important aspects of the problem.

Example: how would you tell someone to draw a face

- Level 1: Tell them to draw a face. The problem cannot be abstracted much more than this.
- Level 2: Divide the face into components (eyes, ears, nose, mouth) and tell them to draw the specific components.
- Level 3: Describe the face as a series of line segments or arcs
- Level 4: Describe the face as a set of points that are drawn (stippling)

What are the strengths and weaknesses of each level of abstraction?

- Level 1: concise instruction, easy to communicate, but it requires a lot of information in the definition of the instruction and is specific to faces, possibly even a single face.
- Level 2: reasonably concise representation, but somewhat limited in terms of the pictures one could draw with it. Appropriate when drawing many different faces.
- Level 3: more complex set of instructions to draw a single face, but the instructions are generic to most kinds of drawing.
- Level 4: very long and detailed set of instructions, completely generic, but painful to describe a single picture.

How much information is provided by each representation? Note that we have to take into account both the definition of the instructions (protocol) and the set of instructions themselves. But there is a slight difference in how we account for them. The definition of the instructions only needs to be transferred to the target once. The instructions themselves must be transferred each time the program is supposed to run.

- Level 1: all of the information is in the definition. The instruction could be a single bit of information.
- Level 2: most of the information is in the definition. The instructions need to contain multiple bits of information.
- Level 3: more of the information is likely to be in the instructions. The definitions are fairly simple to describe.
- Level 4: almost all of the information is in the instructions. The definition requires little more than one bit of information.

Note that it is difficult to say which level is best without having a particular application in mind. There may be limitations on real-time transfer of information that require extensive definitions to be set up beforehand (think Mars rovers). Alternatively, the system itself may need to be simple (drawing dots) and there may be no significant limitations on communication (think dot-matrix printer).

One of the most important issues in designing algorithms is to decide how much abstraction to use for a particular problem. One common method of building complex software systems is to begin with a highly abstract representation of the problem that highlights the key aspects of the task. Then system developers break apart the high level abstractions and start to describe the next level down. You can imagine this as an

upside-down tree. At some point the developers don't need to break down a particular branch any more and can solve that sub-problem by writing code. Once there is code written for all of the outermost leaves, the system is complete. This is a top-down method of designing systems and works very well for large, complex projects.

Programming languages themselves are abstractions, too. A programming language like Python hides a lot of complexity. A simple command like `print 'hello'` translates into thousands of low level machine instructions. Some of the thing that go on include:

- The interpreter parses the string into a series of basic operations
- The string is passed on to a system that decodes it into a series of characters
- The system looks up the font being used in the terminal
- The system generates images for each letter
- The system puts the images in the right place in the video framebuffer
- The video framebuffer gets refreshed and the characters appear on the screen

If we were to delve into what was happening in a single one of the above steps, we would find a series of low level machine instructions that represent the operations required. Those machine instructions would consist of a limited set of actions. If you distill all the things a computer is built to do, they consist of only four types of actions:

- Store data
- Move data
- Manipulate data
- Adjust control flow based on data

If we then looked at how just one of the instructions was executed, we could see how the data was moving around the CPU. Delving even deeper, we could examine the workings of a single component on the CPU and describe it as a set of digital logic gates. Looking at one digital logic gate, we would see that it is built out of a set of transistors. The configuration of transistors produces an electrical circuit with certain properties. A single transistor consists of layers of silicon through which the electrons move.

The only way we (humans) can create such a complex thing as a computer that has 2 billion transistors, executes 3 billion instructions per second and doesn't melt or blow up is to abstract many, many levels. No one person can be an expert at every level.

Where in the computing hierarchy are we studying?

- Theory/Mathematics
- Applications
- Operating System
- Computer Architecture
- Computer Organization
- Digital Logic
- Electronics

- VLSI Design
- Silicon wafer design
- Physics, chemistry

For this class, we're working somewhere in between the operating system and applications. We are using some applications to build other applications. We are also using some parts of the operating system to build and run our programs. Computer science as a field begins somewhere between digital logic and computer organization and goes all the way up to pure theory. It has significant overlaps with mathematics, computer engineering and electrical engineering in terms of what computer scientists study.

So what is our abstraction? What is a useful way of thinking about how to describe a series of actions to the computer?

- The four basic categories of computer actions form a reasonable basis for our abstraction
- Any python instruction falls into one of the four categories
- But we can build new abstractions for collections of python instructions (e.g. turtle graphics)

Part of the power of computing is that we can create new abstractions by defining collections of instructions as a new concept. For example, we could collect the turtle graphics motions

```
forward(50)
right(90)
forward(50)
right(90)
forward(50)
right(90)
forward(50)
right(90)
```

and call that collection the function `square()`. If we can create that function, then any time we need a square we can call the function instead of writing out 8 function calls.

Therefore, if we give ourselves the ability to define new commands that incorporate collections of other commands, then we have the power to set our level of abstraction arbitrarily.

1.2 Algorithms

Whatever our level of abstraction, the end result of defining a solution to a problem is an algorithm. An algorithm specifies the series of commands required to reach a solution. Each command can also represent an algorithm. For example, the `square()` function defined above contains eight commands that constitute the algorithm for drawing a square.

To describe algorithms to computers we must use some kind of interface. The most common interface is a programming language. A programming language is designed in such a way that there is only a single series of actions defined by the program. Computers can't handle ambiguity, so if there are multiple possible interpretations for a program, the computer cannot run the program.

Because a computer requires us to be explicit about what we want it to do, we have to follow the rules of the programming language, whatever level of abstraction we choose to use. The rules constitute the agreement

between you and the computer about what the words and syntax in the language mean. The rules of the programming language are absolute. If you don't follow them, your program will not work.

Note that rules are different from conventions. The rules everyone has to follow or the program doesn't work. There are also many conventions in programming that people follow in order to make their code follow a more standardized format. This standardization makes it easier for people other than the programmer to read the program and understand what is going on.

Some common conventions include:

- How statements are written. Programming languages often allow you to use white space however you like (including not at all). Appropriate use of white space can make code much more readable.
- How variables are named. Variables hold information. If we use names for variables that are meaningful, then it makes it easier to understand the code and avoid making mistakes.
- How functionality is organized. Programming languages let us break up code into pieces. If we subdivide the code in ways that make sense, it makes it easier to edit and debug.

1.3 Computers

Computers contain many different components

- CPU: the central processing unit, which executes the programs
- Cache: temporary, very fast data storage
- Memory: temporary, fast data storage
- Motherboard: usually contains the CPU, cache, memory, and connectors for peripherals
- Hard drive: permanent, slow magnetic data storage
- CD/DVD drive: permanent, slow optical data storage
- Network: the infrastructure connecting computers, allowing them to exchange data

All aspects of a computer are controlled by the programs that run on the CPU. Anything the computer can physically do can be controlled by a program. Very complex software systems may even run on multiple computers and communicate via a network connection.

If you want to control specific parts of the computer (or create new parts for it), then you need to understand how they work and how the parts of the computer talk to one another. Most of the time we don't need that level of control and we can use functions someone else wrote to do things like read and write files from the hard drive or send data over a network.

2 Describing Algorithms

In order to program a computer we have to use a programming language. There are many to choose from, but to get started we're going to use Python.

Python is an interpreted language, which means that there is a program that converts the python program line by line into instructions the computer can actually execute. Because it is interpreted, the transformation from python to machine instructions takes place every time we want to run the python program.

Interpreted languages are nice because we can interact with the interpreter and try out various things quickly without going through any intermediate steps. Unfortunately, because the interpreter is always between the python and the machine code, interpreted languages can be very slow.

A language such as C or C++ is a compiled language. That means when you are done writing your C program, you use a compiler to convert it to machine language. Then you run the machine language version every time you run the application. That means if you make any changes to the code, you have to compile it again before you run it. Fortunately, it also means that the code will generally run faster than the same operations in python because once the program is in machine code it doesn't require any more interpretation to be run on the computer.

To start up the python interpreter, type python in a Terminal window.

```
$ python
```

2.1 Variables

What are variables?

- Variables hold data
- Any variable in python can hold any kind of data
- Once you assign data to a variable, the variable has a 'type' to it
- You can do different things with different types

What are the rules for the names of variables?

- The name has to start with a letter (not a number)
- Names can have letters or numbers in them (no punctuation marks except _)
- Names can be arbitrarily long
- Capitalization matters
- A variable name cannot be a keyword
- To assign a value to a variable, use the assignment operator (=)
- Assigning a value to a variable creates the variable if it does not already exist

What are the conventions for variables? (Programmers have conventions that let them write code that other people can easily look at, read, and understand.)

- Variables that can change value generally start with lower case letters
- Variables should be descriptive about the values they hold
- Multi-word variables generally use capitalization or underscores (e.g. the_big_one, or theBigOne). For most programming projects, the convention is selected up front.

What can go wrong?

- Breaking a rule
- Typos

Example

To create a variable in python, you simply assign a value to the variable. To see the value of a variable you can print it out or just evaluate it as an expression.

```
>>> x = 50
>>> print x
50
>>> x
50
```

Note that printing out a variable (which prints out its value) is not the same as evaluating an expression, which prints out the value of the expression. The latter is going to be identical to the right side of an assignment statement that would give the variable that value. The difference is obvious in strings, for which evaluating the expression produces a string in quotes.

```
>>> x = 'Hi there'
>>>print x
Hi there
>>> x
'Hi there'
```

Typos are one of the most difficult errors to track down in python because variables are created dynamically; all you have to do is assign a value to a variable name. In the example below, a typo causes the last line of the program to generate an error.

```
>>> abigvariblename = 50
>>> asmallvariablename = 40
>>> anothervariable = abigvariablename + asmallvariablename
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'abigvariablename' is not defined
```

One of the most important things to remember is that once a variable has been assigned a value, that variable takes on the type of its value. You can discover the type of the value a variable contains by using the `type()` function.

```
>>> x = 10
>>> type(x)
<type 'int'>
```

```
>>> x = 10.0
>>> type(x)
<type 'float'>
```

```
>>> x = '10'
>>> type(x)
<type 'str'>
```

```
>>> x = 10L
>>> type(x)
<type 'long'>
```

```
>>> x = complex(10, 10)
>>> type(x)
<type 'complex'>
```

```
>>> x = True
>>> type(x)
<type 'bool'>
```

Casting

2.2 Operators

Operators are the basic methods for manipulating data. Add, subtract, multiply and divide are all standard in Python (as in most programming languages). In addition, Python provides a number of other useful operators, including some support for complex numbers.

addition	`x + y`	subtraction	`x - y`
multiplication	`x * y`	division	`x / y`
floored result of x/y	`x // y`	remainder of x/y	`x % y`
exponentiation (x^y)	`x ** y`	exponentiation (x^y)	pow(x, y)
divmod $(x//y, x\%y)$	`divmod(x, y)`	absolute value	`abs(x)`
complex number	`complex(x, y)`	complex conjugate of x	`x.conjugate()`

Note that the same operator can have different effects on different types. Integer division, for example, is different than floating point division. Integer division will not return a decimal value, while floating point division will.

Addition on strings is concatenation. Multiplication of a string by an integer will duplicate the string the specified number of times.

```
>>> a = "abc"
>>> a * 3
"abcabcabc"
```

```
>>> a = "abc"
>>> b = "def"
>>> a + b
"abcdef"
```

Mixing types in expressions leaves you in the most flexible type, if it works at all. Adding a float and an integer results in a float because a float can represent an integer, but an integer cannot represent most floating point values.

Order of operation is important when writing expressions using operators. For example, multiplication and division will occur before addition and subtraction. The complete ordering is given in the book. Use parentheses to clarify the situation and make the expression more readable.

2.3 Functions

One of the most important capabilities of a programming language is the ability to modularize a program into component parts. Most languages do this by permitting the programmer to create functions.

- We can define a new instruction, or function, as a series of instructions
- Functions can take parameters that affect their actions
- Functions allow us to subdivide a problem into more reasonable pieces.
- Functions abstract away from details
- Functions reduce the amount of code we have to type.

Functions are great, because they help us to automate processes and focus on their important aspects (the parameters). Functions can take parameters that define how the function is supposed to work. For example, if you tell someone to draw a line 2in long, the function would be "draw a line" and the parameter would be (2in). The possible values of the parameters define the range of actions a function can execute.

In Python, functions begin with the `def` keyword. This is followed by the name of the function, then the list of function parameters is given inside parentheses. Each parameter is a variable inside the function and the variable names must be legal according to the python rules (start with a character, include only letters, numbers, or an underscore). The final syntax element of the function header is a colon.

```
def myfunction(arg1, arg2):
```

The instructions contained within a function follow the `def` statement. Python uses tabs to delineate what instructions are contained inside a function. If the function statement begins at one level of tabbing, the items within the function need to be tabbed at the next level.

```
def simpleFunction(x):
    forward(x)
    right(20)
    backward(x)
```

There is no other syntax required for the function. The end of the function is indicated by the level of tabbing. Once a statement occurs that is not tabbed over relative to the function header, the function is terminated. White space or blank lines within the function are allowed. Except for using tabs to specify that statements are within a block python doesn't care much about white space.

Parameters allow us to pass values into functions. The parameters are local variables within the function. They start the function with the value assigned to it when the function was called. You can change the value within the function, if necessary, but common practice is to create local variables for values that need to change. As with all variables, parameters can hold any data type. Use informative parameter names.

Example: create a function that prints out the value and type of a parameter

```
>>> def lookat(a):
...         print a
...         print type(a)
...
```

```
>>> lookat(6)
6
<type 'int'>
>>> a = 5.0
>>> lookat(a)
5.0
<type 'float'>
```

Variables within a function have scope. Scope is where in your code you can access the value of a variable. Variables declared inside a function cannot be accessed outside the function.

Parameters are passed to functions by value. That means a copy of the value passed into the function is created for use inside the function. If you change the value of a parameter variable within the function, it does not change anything else.

Example: create a function that tries to modify a variable. Note that the variable b only exists inside the function.

```
>>> def changeit(b):
...     print b
...     b = 50
...     print b
...
```

```
>>> a = 20
>>> changeit(a)
20
50
>>> a
20
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'b' is not defined
```

Functions also let us design top-down solutions to problems. When confronted with a complex problem, we should be able to subdivide into a small number of less complex steps. The complexity of each step may still be significant, but if the subdivision is done well, each individual step will be less complex its parent.

- Think of each subdivision as a function
- Parameterize each step in terms of the important variables that get passed into the function
- Write the high-level code with placeholders for each function.

If each individual step is still complex, we can repeat the process iteratively. At some point, we'll end up with steps that are possible to express in just a few instructions. Then we can write those functions in code.

Example

Consider the task of creating a face of a given size at a particular location and orientation. We can break the problem into a series of steps as below.

1. Move the turtle to the position and orientation of the face
2. Draw the mouth
3. Draw the nose
4. Draw the eyes

Let's create a function for each of these steps. What parameters are required for each component?

1. positionTurtle(x0, y0, a) - requires the (x, y) location and angle a.
2. mouth(size) - requires the size of the face
3. nose(size) - requires the size of the face
4. eyes(size) - requires the size of the face

The first function we can write directly using the commands goto$(x0, y0)$ and left(a). Given the simplicity of the task, there is no reason to subdivide it further.

The mouth function could be quite complex. For example, we could draw lips and teeth, or we could just draw a simple line. In the former case, we probably want to subdivide the task some more, while in the latter we can just encode the line.

Likewise, the nose function could be made complex if we tried to draw nostrils and shading. Or it could again be a single line.

The eyes function makes sense to subdivide since there are two eyes. We could divide it into four steps.

1. Position the turtle for the first eye
2. Draw the first eye
3. Position the turtle for the second eye
4. Draw the second eye

Drawing an eye could then be made a function, possibly parameterized by pupil location.

The end result of this subdivision is a tree whose leaves contain most of the code that does the actual work. Note that by subdividing the task in a top-down fashion we have reduced the amount of code we have to write compared to no subdivision at all. We also didn't have to think very hard about any individual function.

2.3.1 Algorithm design using functions

Modularity, as noted above, is one of the most important aspects of algorithm design. Functions are the primary method of incorporating modularity into algorithm design.

- Code to do a particular task should be written only in one place. Why?
- Modularity enables easy re-use of code. How?
- Modularity enables faster debugging. How?
- Modularity makes it easier to modify functionality. Why?

A common design process when writing small programs is as follows:

1. Define the task using natural language
2. Identify the inputs and outputs of the task
3. Recursively break down the problem into smaller steps
4. Organize the steps, noting the input and output requirements of each step
5. Identify what the individual functions should be, noting where in the steps there are similar input and output requirements.
6. Generate some intermediate representation of the algorithm
   - Flow chart
   - Pseudo-code style comments
   - Pictures or diagrams
7. Write code
8. Verify and test the code

One of the most important habits to develop in programming is to test often. A concept called unit testing, proposes that each function, or unit of a program should be tested individually before being combined as a whole. Python actually has a method for unit testing built into it that we will look at a bit further on in the course.

2.4 Control Flow

Sometimes when writing a solution to a problem, we don't want the same thing to occur every time. Sometimes we want the computer to react to input and change its actions based on the input.

In order to control the flow of a program, we need the following tools.

- Syntax and keywords for the control flow statement
- Expressions that evaluate to true or false
- A method of specifying which statements are dependent upon the expression

The primary method of conditional control flow is the `if` statement. A simple `if` statement controls whether a single block of code is executed or not.

```
if <expression>:
    <statements>
```

In order to generate expressions, we need new operators that evaluate to true or false. Comparison operators provide the tools for testing the relative qualities of variables.

- `a == b` - returns true if the value of a is the same as the value of b
- `a < b` - returns true if the value of a is less than b
- `a <= b` - returns true if the value of a is less than or equal to b
- `a > b` - returns true if the value of a is greater than b
- `a >= b` - returns true if the value of a is greater than or equal to b
- `a != b` - returns true if a is not equal to b

Logical operators let us mix and match expressions that evaluate to boolean values (true or false).

- `a and b` - evaluates to true if both a and b are true
- `a or b` - evaluates to true if a is true, b is true, or both are true
- `not a` - evaluates to true if a is false

So if we wanted to check if a number is within a certain range, we could use the expression

```
a > lowerBound and a < upperBound
```

Note that order of operation is important here. The comparison operators have higher precedence so that the expression does what we expect. In order to avoid mistakes and enhance readability, however, we may want to consider putting parentheses around the two comparison operator expressions.

```
(a > lowerBound) and (a < upperBound)
```

Note that because python is cool, you can also do the same test using the syntax

```
lowerBound < a < upperBound
```

and it will do the right thing.

Given that we can now create boolean expressions, how do we indicate what code is conditional on the expression? Turns out we use the same mechanism used for functions: statements that are consecutively tabbed in after the `if` statement are executed if the expression evaluates to true.

```
if a > b:
    print str(a) + ' is greater than ' + str(b)
```

Often we have cases where we want to do one series of actions if the expression evaluates to true and another series of actions if the expression evaluates to false. In that case, we can use an if-else type of control flow that makes use of the keyword `else` to indicate the code that should be executed if the expression evaluates to false.

```
if a > b:
    print 'a is greater than b'
else:
    print 'b is less than or equal to b'
```

Sometimes we also have cases where there may be many different actions we want to consider on input. So long as we can express each case as a boolean expression, we can consider each case using an if-elif-else type of control flow.

```
if a > b:
    print 'a is greater than b'
elif a < b:
    print 'a is less than b'
else:
    print 'a is equal to b'
```

There can be as many `elif` cases as necessary for the situation.

An `if` statement let's us execute different code based on run-time data. That means, the order in which the code is executed is not pre-determined when the code is written. The program can respond to the particular circumstances in which it is run. Note that the program is still predictable in the sense that the same input ought to produce the same output if it does not incorporate (truly) random numbers.

Example

Consider the guessing game high-low. We can let the computer generate a random number, and then write a function that will tell us if our guess is high or low.

```
from random import *

def highlow(a, b):
    if b < a:
        print 'low'
    elif b > a:
        print 'high'
    else
        print 'correct'

a = int( random() * 1000 )
highlow(a, 500)
```

Control flow can also be nested inside other control structures. For example, consider the case of trying to find the maximum of three numbers. Two approaches we could take are as follows.

1. Test each possible ordering of the numbers
2. Pick two, find the larger value, then test it against the remaining value.
3. Take a guess and keep track of the current guess. Compare it against all other possibilities.

For case one, the code would be of the form if-elif-else. Each test could consist of checking one variable against the other two.

```
def min1(a, b, c):
    if a > b and a > c:
        print a, ' (', b, ' ', c,')'
    elif b > a and b > c:
        print b, ' (', a, ' ', c,')'
    else
        print c, ' (', a, ' ', b,')'
```

In the other case, one pair is test first, followed by the other pair. The code consists of nested if statements.

```
def min2(a, b, c):
    if a > b:
        if a > c:
            print a, ' (', b, ' ', c,')'
        else:
            print c, ' (', a, ' ', b,')'
    else:
        if b > c:
            print b, ' (', a, ' ', c,')'
        else:
            print c, ' (', a, ' ', b,')'
```

Version two can be thought of as a decision tree. Each decision only involves one test and discards some fraction of the possible cases. Ideally, we want to discard as many cases as possible no matter what the decision is.

To optimize a decision tree, what percent of the cases should be discarded at each step?

Now consider how many operations are executed for the two cases. In the first case, the function could get lucky and finish after evaluating two cases (max is a). In the worst case (max is c) the algorithm evaluates four conditions.

In the second case, the algorithm only evaluates two comparisons, no matter what. Therefore, the second algorithm not only matches the best case of the first algorithm, but never does any worse. Decision trees are a powerful method of control flow and let us efficiently find the appropriate course of action given a set of inputs.

The third case approaches the problem as a sequential one. The first number becomes our guess at the max. If there are no other numbers, we're done. If there is a second number, we compare it against our current guess at the max. If it's bigger, replace our current guess with the new value, otherwise don't change it. Note that this algorithm scales to as many numbers as we might have.

```
def min3(a, b, c):
    max = a
    if b > max:
        max = b
    if c > max:
        max = c

    print 'max value is ', max
```

The third case does no more comparisons than the second and is easier to understand and code. It also

expands to more numbers easily. One problem, however, is that it doesn't naturally keep track of which of the numbers is the max, just the value of the maximum. How would we change it to keep track of which variable is the maximum?

Example

Using an if-statement, it is possible to transform a symbol–the value of a variable–into something else. For example, what if we assigned each standard turtle command a single letter symbol and used a single function to execute them?

```
def turtleDo( cmd, value ):
    if cmd == 'f':
        forward(value)
    elif cmd == 'b'
        backward(value)
    elif cmd == 'r'
        right(r)
    elif cmd == 'l'
        left(r)
    else
        print "The character '"+cmd+"' is not a valid symbol"
```

Then we could create a square using the following set of commands.

```
turtleDo('f', 50)  # executes forward(50)
turtleDo('r', 90)  # executes right(90)
turtleDo('f', 50)  # executes forward(50)
turtleDo('r', 90)  # executes right(90)
turtleDo('f', 50)  # executes forward(50)
turtleDo('r', 90)  # executes right(90)
turtleDo('f', 50)  # executes forward(50)
turtleDo('r', 90)  # executes right(90)
```

Consider what the above program does. It transforms turtle commands, which are python code, into data that is all processed by a single function that gets called repeatedly. What can computers do with data?

Another way of thinking about it is that we've created a new abstraction for turtle commands that contains a single function with two parameters. How difficult would it be to add new turtle functions like letting the symbol 's' become a square?

2.5 Iteration

Repetition, or iteration of a series of operations is commonplace in programming. For many tasks, we can define the solution as a series of identical operations on a sequence of things. With the ability to make conditional statements, we can even do different operations on each member of a sequence.

2.5.1 while

Often we can express repetition as occurring until something happens. One way to express that idea is to say that a set of statements should repeat while a condition is true. Once the condition is false, the computer should stop executing the statements.

- We need syntax to express the iteration
- We need an expression that evaluates to true or false
- The body of the iteration needs to do something that will eventually cause the loop to exit

```
while <expression>:
    <statements>
```

Example:

```
while n > 0:
     print n
     n = n - 1
```

2.5.2 Aside: strings as arrays

Strings are a collection of characters. In python, strings can be delineated by either single or double quotes.

- `'this is a string'`
- `"this is also a string"`

If you use one type of quote marker to delineate the string, you can use the other type of quote marker as part of the string.

```
'you can put "quotes" around a word'
"in one of 'two' ways"
```

Python has to store strings in memory. Memory is like a long list of cubby-holes all the same size. Each cubby can hold one byte of data. A byte is eight bits. A bit is a binary digit and can take on the value 1 or 0. Therefore, each byte can hold one of 256 values.

Question: if each color channel for a pixel is represented by one byte, how many different colors can a computer screen display?

$$256 * 256 * 256 = 2^8 * 2^8 * 2^8 = 2^24 \approx 16 \text{ million} \quad (2)$$

Each character in a string is typically represented as a byte of memory. Once upon a time in computer history someone came up with a mapping from numbers to characters. The most common mapping is called

ASCII [American Standard Code for Information Interchange]. ASCII uses 8-bits, or one byte to represent each character.

A string is simply a collection of characters that python has put in consecutive memory locations inside the computer. The name of the string is associated with the address of the first character in the string. When data is stored conceptually (or physically) as a sequence of elements we call that an array.

What if we want to look at a specific character in the string? It turns out the python provides a notation that allows us to look at the specific elements in an array.

```
>>> a = 'abcd'
>>> a[0]
'a'
>>> a[1]
'b'
>>> a[2]
'c'
>>> a[3]
'd'
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Note what happens when we use an index that goes beyond the end of the array. Python generates an error and tells us our index (4) is too big for this array. Array indexes follow three rules.

- Indices must be integers (a[1.0] generates an error)
- The first element in a string is at index 0
- Indices must not try to access elements beyond the length of the string
- Positive indices from 0 to length-1 are valid and index the string from left to right.
- Negative indices from -1 to -length are valid and index the string from right to left.

We can always discover the number of characters in a string using the `len()` function.

```
>>> a = 'lots of characters'
>>> len(a)
18
```

One thing to be aware of is that python strings are **immutable**. That means you can't change the value of a character in a string once it's created. You can, however, build a completely new string and put the new string into the old variable. It's an idiosyncrasy of python.

Why is this section part of iteration? Because if we can loop a certain number of times, then we can go through each element of a string. If we can look at each element of a string, then we can consider each character a symbol and translate it into the appropriate action.

Another cool thing with strings: you can multiply them with integers

2.5.3 for

The `for` loop provides a more convenient syntax and mechanism for iterating over an array or a list of elements.

Syntax:

```
for <variable> in <list or array>:
    <statements>
```

The variable can be any legal variable name in python. The variable has scope only within the for loop and where it has scope it will override any variable of the same name defined elsewhere.

The list or array is a the collection of things over which the loop should iterate. The first time through the loop, the variable gets the value of the first item in the list or array. The second time through the loop, the variables gets the value of the second item in the list or array, and so on, until all of the elements have been processed.

To loop over the elements of a string, we can simply do something like:

```
for curChar in aString:
    print curChar
    < other stuff with the character curChar >
```

If you want to have the loop variable take on a set of consecutive numbers, you can use the built-in `range()` function.

```
range(<start>, <end>, <step)
```

- If you give the range function a single parameter, it generates a list that contains elements from 0 to one less than the given number in increments of 1.
- If you give the range function two parameters, it produces a list starting at the first number and ending at one less than the second number in increments of 1.
- If you give the range function three parameters, it produces a list starting at the first number, incrementing by step, and ending no less than step from the ending number.

Key concept: for loops work through a list of items

- The list can be a string, in which case it works through the elements of the string
- The list can be a set, which is (conceptually) a linear sequence of things
- The range() function is your friend

Simple lists are a comma separated sequence of items, surrounded by brackets. You can index them using the bracket notation, just like strings.

```
listOfNumbers = [1, 2, 3, 4, 5]
listOfStrings = ['ab', 'cd', 'de']
listOfMixture = [1, 'ab', 2, 'bc', 3.0]
```

Example: Use a for loop to iterate over the elements of a list and print out the value and type of each element.

```
>>> def showlist(mylist):
...     for item in mylist:
...             print item, " : ", type(item)
...
>>> alist = [1, "45", 45, "thirty", 30.0]
>>> showlist(alist)
1  :  <type 'int'>
45  :  <type 'str'>
45  :  <type 'int'>
thirty  :  <type 'str'>
30.0  :  <type 'float'>
```

2.5.4 Common loop structures

Loops are the workhorse of programming. One goal of programming is to never, ever have to type a sequence of numbers that follow a pattern. Life is too short.

Some commonly used loop patterns are the following.

- Interactive loops: these generally ask the user for some kind of input. The loop terminates when the user provides the proper input. There are several forms of these kinds of loops.
  - Menu loop: give the user a set of possible choices and use their input to guide the program
  - Sentinal case 1: one of the possible inputs sets the quit flag
  - Sentinal case 2: one of the possible choices exits the loop using a break
- Simple linear for loops: one loop going over a sequence of elements
- Nested loops: one loop inside another, allows manipulation of multi-dimensional concepts

See examples from class on the course web site.

With for loops in python, it is important to remember that both strings and lists can be used as the foundation of the for loop. For example, both of the following for loops does the same thing.

```
a = "abcdef"
for char in a:
print char

b = ['a', 'b', 'c', 'd', 'e', 'f']
for char in b:
print a
```

2.6 Review of program design

Now that you have written a program, the process of program design may actually become meaningful. The most important step, by far, is step number 1. If you have a clear idea of what the critical aspects of the program are, it makes it easier to design solutions and guarantee that you meet the requirements of the problem.

Process:

1. Define the task using natural language (understand the problem)
2. Identify the inputs and outputs of the task
3. Recursively break down the problem into smaller steps
4. Organize the steps, noting the input and output requirements of each step
5. Identify what the individual functions should be, noting where in the steps there are similar input and output requirements.
6. Generate some intermediate representation of the algorithm
    - Pictures
    - Flow chart
    - Pseudo-code style comments
7. Write code
8. Verify and test the code

As you subdivide a problem, one of the things to keep in mind is that you can define the meaning of the functions of a parameter. Make sure the parameters take on a meaning that is appropriate for what they need to do and that makes it easy to write the function.

Example: Subdividing a problem

Problem statement: draw a shape repeatedly along a line of a specified length at a specified spacing

- What are the key objects in the problem definition?
- What is the relationship between the key objects?
- What should the shape be? How do we find out?
- What does 'along a line' actually mean? How do we find out?
- What are the parameters of the top level program?
- Is the line supposed to be drawn in a particular location? How do we find out?

Write out a top level algorithm at a high level of abstraction

```
Go to the starting location and orientation (specified?)

Figure out how many objects to draw

for the number of objects to draw
    draw an object
    move to the next location (pen up or down?)
```

What other functions do we need to define?

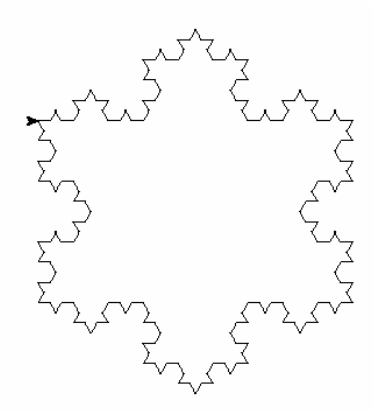
- What is the shape to draw?
- Define how to draw the shape
- What parameters need to be passed to the shape?

Note how the move command in the for loop obviates the need for parameters to the shape. We could give the shape function position parameters instead and calculate them dynamically in the for loop, but then we have to make other decisions.

- Must the line be horizontal?
- We have to either obtain the starting position of the turtle, or have a starting location passed to doline

Example: Using characters to represent specific turtle actions we can create a Koch snowflake.

```
from turtle import *
from turtleUtils import *

def doAction(action):
    if action == 'f':
        forward(5)
    elif action == 'b':
        backward(5)
    elif action == 'l':
        left(30)
    elif action == 'r':
        right(30)
    elif action == 'u':
        up()
    elif action == 'd':
        down()

turtleSetup(500, 500, 100, 100)

up()
goto(-130, 50)
down()

unit = 'ffllffrrrrffllff'
unit2 = unit+'ll'+unit+'rrrr'+unit+'ll'+unit
unit3 = unit2+'ll'+unit2+'rrrr'+unit2+'ll'+unit2
snowflake = (unit3 + 'rrrr') * 3

for i in snowflake:
    doAction(i)

turtleWait()
```



2.7 Lists

As noted above, lists are sequential collections of objects. What is an object?

- A standard data type: int, long, float, str, bool, func
- A programmer defined data structure called a class
- Another list

2D lists (and higher) provide lots of opportunities for organizing information.

Example: organizing color information

What if we want to define a set of colors that our shapes will cycle through in a regular pattern?

Algorithm

- Generate the series of colors
- Loop through the number of shapes to draw
  - Index into the series of colors to determine the shape color
  - Draw the shape

```
colors = [ [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0] ]
for i in range(10):
    colorIndex = i % len(colors)
    color(colors[colorIndex ][0], colors[colorIndex[1], colors[colorIndex][2])
    forward(10)
```

Note the use of the mod operator % when creating the color index to ensure that it is between 0 and the number of colors in the color list. This is a common trick in programming when looping through a sequential array of elements.

The key idea here is that an index is like an id number. In a list, we can store whatever we like at a particular id number. If we need to store all of the information about a shape (size, color, line width), we could do that. All we need to do is be consistent about how we store it.

Consider the squares function from project 2. The squares function takes 8 arguments (x0, y0, dx, dy, angle, red, green, blue). We could store those 8 arguments in a list and then make a list of lists to hold the arguments for lots of different squares.

```
squares = [ [-100,0,10,20,0,1.0,1.0,0.0], [-100,100,20,10,0,0.0,1.0,0.0] ]
```

In the above case, squares[0] would provide the list of arguments for the first square. Each argument would be indexed as squares[0][i] where i is the index of the argument.

To modify a particular argument, the same notation is used only on the left side of an assignment statement.

```
>>> squares[0][1] = 50
>>>squares
[[-100,50,10,20,0,1.0,1.0,0.0], [-100,100,20,10,0,0.0,1.0,0.0]]
```

We can also use 2D lists as the arguments for nested for loops.

```
>>> a = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
>>> for i in a:
...     for j in i:
...             print j,
...     print
...
1 2 3
4 5 6
7 8 9
```

An alternative method of indexing into exactly the same data is to use a nested for loop over the number of rows and columns in the data.

```
>>> for i in range(len(a)):
...     for j in range(len(a[i])):
...             print a[i][j],
...     print
...
1 2 3
4 5 6
7 8 9
```

Neither method cares exactly what the number of rows or columns of a is, they both adapt to the particulars of the variable. Note that both methods can also handle the case where the rows don't all have the same number of columns.

In summary, both lists and strings are objects over which we can iterate using a for loop. The difference between lists and strings is twofold.

- The elements of a string cannot be changed, while the elements of a list are mutable.
- A string can hold only characters, while the elements of a list can be any object.

3 Input/Output

One of the most important aspects of program design is the human-computer interface. There is an entire field of computer science devoted to nothing but the study of how to create effective user interfaces for programs. Input from the world and the ability to output results are critical to effective program function.

When designing real programs, the human-computer interface can easily take more than 50% of the total time required to write the program. The core reason for the extent of HCI programming required fora program is that people do not naturally use the precision required of computers, and computers will only do what you tell them to do.

The difference between a bad user interface and a good user interface can determine whether a program is functional or not. Consider the following series of programs that do exactly the same thing, functionally, but have very different user interfaces.

Note that the function `input()` is one way of getting input from a terminal. The function takes one optional argument, which is a string to print out for the user. Then the function waits for the user to type their input and hit return. At that point, the input function evaluates the expression typed in by the user. Evaluating the expression works for numerical expressions, but it can't handle characters or invalid syntax. If the user doesn't type a legal numerical expression, the function calls an exception.

Example: a variety of user interfaces

```
# bad user interface
def badInterfaceAdd():

    a = input()
    b = input()

    c = "%X" % (a + b)

    print c


# an ok user interface
def okInterfaceAdd():

    a = input("Enter a number: ")
    b = input("Enter a number: ")

    print "The answer is: ", a + b


# a better user interface
def betterInterfaceAdd():

    print 'This program takes two numbers as input and returns their sum\n'

    a = input('Enter the first number: ')
    b = input('Enter the second number: ')

    print
    print 'The sum of', a, 'and', b, 'is', a + b

    print '\nTerminating'
```

```
# a better user interface
def goodInterfaceAdd():

    # Tell the user what the program does
    print 'This program takes two numbers as input and returns their sum\n'

    # Try to grab some input, allowing for exceptions
    try:
        a = input('Enter the first number: ')

    # handle any exeptions
    except Exception:

        # Enter a while loop until the user provides good input
        goodinput = False
        while not goodinput:
            try:
                goodinput = True
                print 'The input must be an integer or floating point number'
                a = input('Please re-enter the first number: ')
            except Exception:
                goodinput = False

    # Grab the second number
    try:
        b = input('Enter the second number: ')

    # handle any exceptions
    except Exception:
        goodinput = False
        while not goodinput:
            try:
                goodinput = True
                print 'The input must be an integer or floating point number'
                b = input('Please re-enter the second number: ')
            except Exception:
                goodinput = False

    # nice formatting of the output, with some white space
    print
    print 'The sum of', a, 'and', b, 'is', a + b

    # Tell the user the program is complete
    print '\nTerminating'
```

If we want to have more control over processing the input string, then we need to grab the raw string and process it character by character. The `raw_input()` function grabs a line of input from the terminal and simply passes back the empty string.

We can use the `int()` and `float()` casting functions to convert from strings to the relevant number type, but those functions will only work on strings that are valid numbers. We could also try and process the string manually, building up the number digit by digit.

Example: Grabbing integers from the command line

For integers, we can try to extract all of the digits at the front of the string, ignoring anything else. The following are several methods of increasing complexity.

```
def simplegrab():

    print 'Enter an integer: '
    s = raw_input()

    valueOfS = int(s)

    return valueOfS


def okgrab():

    print 'Enter an integer: ',
    s = raw_input()

    q = ''
    for char in s:
        if char >= '0' and char <= '9':
            q = q + char
        else:
            break

    print 'processing string: ', q

    valueOfS = int(q)

    return valueOfS
```

```
def bettergrab():

    print 'Enter an integer: ',
    s = raw_input()

    q = ''
    for char in s:
        if char >= '0' and char <= '9':
            q = q + char
        else:
            break

    print 'processing string: ', q

    valueOfS = 0
    if len(q) > 0:
        valueOfS = int(q)
    else:
        raise ValueError, 'Not a valid integer'

    return valueOfS
```

Of course, the `bettergrab()` function only partly solves the problem. By raising an exception we throw the problem back in the lap of the calling function. But there is a bit more robustness to errors and we control the meaning of the exception (no real data entered).

The `eval()` function is a little more general than a specific cast statement like `int()` or `float()` because it evaluates the function generically as though it were a python statement. Therefore, if we pass it a string that is either a floating point or an integer it will return the proper value in the proper type.

In the example above, if we allow strings that are combinations of digits and a decimal point then by using the `eval()` function instead of the `int()` function we can make the grab functions get either type of number from the user.

3.1 Parsing Strings and Lists

Parsing a string or list is the act of going through the elements of a string or list and executing actions based on each element. There are two different ways to parse a string or list that produce the same result, but provide slightly different information inside the loop. Consider the following two code snippets.

```
# note that the string 'fudge' is assigned to the variable fudge
fudge = 'fudge'
for y in fudge:
    print y

for y in range( len(fudge) )
    print fudge[y]
```

In the first for loop, the loop variable y will hold the characters of the string 'fudge'. The first time through the loop it has the value 'f'. The second time through the loop it has the value 'u', and so on.

In the second for loop, the loop variable y will hold the indices of the characters of the string 'fudge'. The first time through the loop it has the value 0. The second time through the loop it has the value 1, and so on. Remember, to get the element of a list at a particular index, use the bracket notation with the index in the brackets.

Both loops to do the same thing, but the loop variable has a different meaning. In the first loop, there is no need to index into the string fudge to get the character; the loop variable is the current character. In the second loop, if you need to take some action based on the position of the character in the string, the loop variable tells you the position of the current character.

The ability to walk through a string and take actions depending upon the value of the characters lets us do many things.

Example: Parsing a string when we know its constituent parts

```
# tell the user what to do
print 'Enter turtle command: ',
commandString = raw_input()

# break if the string is empty
if commandString == '':
    break

# parse the string
command = commandString[0]
argument = commandString[1:]

# check the first character
if command == 'f': # forward

    # get the distance to travel
    distance = eval(argument)
    forward(distance)

elif command == 'r': # right

    # get the angle
    angle = eval(argument)
    right(angle)

elif command == 'l': # left

    # get the angle
    angle = eval(argument)
    left(angle)

elif command == 'u': # up

    up()

elif command == 'd': # down

    down()

else:
    print 'try something else'
```

3.2 File I/O

Files are streams of data. The important thing is how our program interprets the data.

Text files are just big long strings, just stored on a hard drive instead of in a variable in memory. To access a file, we just need to know its name and where it is in the file tree. Just like we can navigate through the file tree of our computer using the terminal, we can use the same method of specifying paths in python. The directory from which you run your program is also the working directory inside the program. Any files in the working directory can be accessed by just their filename. All other files either need to be specified relative to the file tree root (/) or relative to the current working directory.

It's useful to think of text files as strings separated by newlines.

There is a built-in data type called file that lets us open, read, write, and close files.

The file data type is a class. In most respects it's no different than an int or a float. To create a data type of a particular class, you use the name of the class as a function.

Besides the method of creating classes, the main difference between classes and the basic data types is that you can give classes methods. A method is just a function that belongs to a class. Think of it as a trick that the data type knows how to do.

Conceptually, if we had a class called dog, we might be able to do the following:

```
fido = dog()
fido.shake()
fido.fetch()
```

fido is just a variable, a label for a memory location. The first line assigns the variable an object of type dog. The class dog has certain methods that someone has written that do things. We can have the object execute those functions using the dot notation and the method name.

The file class works the same way.

```
fp = file('myfilename', 'w')
fp.write('writing a string to the file\n')
fp.close()
```

Reading files works about the same way.

```
fp = file('myfilename', 'r')
lineOfText = fp.readline()
print 'The contents of the file are:'
print lineOfText
fp.close()
```

Files use something called a file pointer to keep track of where you are in the file. When you read a line from the file, if you call readline again, it reads the next line of the file. The file pointer always moves to point to the next thing that will be read from the file. When you open a file using the 'r' or 'w' mode, the file pointer starts at the beginning of the file. If you open a file using the 'a' mode, the file is opened for writing, but the file pointer starts at the end. Thus, the 'a' mode is for appending things to a file (although it does create the file if the file does not exist).

Trying to open a file for reading that doesn't exist generates an IOError exception. When opening a file, it's always a good idea to try and catch IOError exceptions and handle them gracefully. We can use the try/except structure to catch errors.

```
try:
    fp = file(filename, 'w')
except IOError:
    print 'Unable to open file', filename
    return
```

When reading a file, there are three `read` methods from which to choose.

- read(): reads the rest of the file and returns it as a single string
- readline(): reads from the file up to, and including, the next newline character and returns the string
- readlines(): reads each line from the file and returns a list with each line as an entry in the list

Note that the first and last functions could return very large objects.

One way to think of files is as a stream of data. A text file is just a stream of characters, one after the other. Python lets us treat files like a string or a list when it comes to for loops. You can use a file object as the list or string and the loop variable then becomes the current line of the file.

```
fp = file('myfilename', 'r')

for line in fp:
    print line

fp.close()
```

This is an easy way to go through each line of a file and process it. If you wanted to go through each character of the file, you would need to go through each line with a second for loop nested inside the first.

```
fp = file('myfilename', 'r')

for line in fp:
    for char in line:
        print char,

fp.close()
```

The ability to read in a file as a series of lines tucked into a list (the `readlines()` function) provides an easy way to sort the lines of a file. How?

Something we haven't touched on yet is that a list, like a file, is an objects that has methods it knows how to execute. For example, lists know how to do the following:

- `append(object)` - appends object (whatever it is) to the end of the list
- `count(value)` - counts the number of times value (whatever it is) appears in the list
- `index(value)` - returns the index of the first occurrence of value (whatever it is)
- `pop()` - remove and return the last value from the list
- `reverse()` - reverse the elements of a list in place (modifies the ordering of the list)
- `sort()` - sort the items of the list in place (modifies the ordering of the list)

If we read the lines of a file into memory using the readlines() function, then we can apply the sort() function to the list and then write out each line back to a file using a for loop over the elements of the list.

How do you find this stuff out? Python has a help facility built into it. For any data type, you can type

`help( type )`

and it will print out information about the data type. Try it out with the `str`, `list`, or `file` data type and see what other functions are there. Any of these built-in methods you can apply to any object of that type.

Example: saving your work

In an interactive program you may be entering a series of commands. At the end you have a shape you like, but the only way to save the picture is to make a screen capture.

Why not store the commands the user enters and save the whole series of commands to a file when the user quits?

```
def main2():
    print 'starting'
    turtleSetup(500, 500, 200, 200)

    # initialize the variable in which to save the commands
    memory = ''

    while True:
        print 'Enter string to process: ',
        s = raw_input()

        # break out of the loop if the user entered nothing
        if len(s) == 0:
            break

        # process the string
        processString(s)

        # add the current command to memory
        memory += s

    print 'Writing list of commands to file memory.txt'

    # open a file, write the string, and close the file
    fp = file( 'memory.txt', 'w' )
    fp.write(memory)
    fp.close()

    print 'Terminating'
```

What if we wanted to start the whole process with a string read from a file?

3.3 Formatted I/O

When working with files or strings that people or computers have to read, formatting can make a big difference in how easy the file is to read. We can do some formatting using string concatenation, but we can't control the way the numbers are printed, how many significant figures are printed, or how many spaces a number takes up.

To provide more control over how things are formatted, the print statement lets us separate formats from the values that will be printed. In other words, we can include in a string placeholders with formatting information into which a value will be placed when the string prints out. The basic syntax is:

```
print <format string> % ( <comma-separated list of arguments> )
```

The format string is a regular string but with placeholders in it where you want to put numbers. The placeholders have the syntax:

```
%<field size>.<precision><type>
```

The most commonly used types are:

- `d` - integer
- `f` - floating point number
- `s` - string
- `X` - print the integer as a hexadecimal number

The following are some examples and what they do.

```
>>> print "%d" % (5)
5
>>> print "%f" % (5.0)
5.000000
>>> print "%10.2f" % (5.0)
      5.00
>>> print "%0.2f" % (5.0)
5.00
>>> print "%X" % (48)
30
>>> print "%s (%d, %d)" % ('the coordinates are', 5, 6)
the coordinates are (5, 6)
>>> a = [15, 32, 45]
>>> print "( %d, %d, %d )" % (a[0], a[1], a[2])
( 15, 32, 45 )
```

A couple of things to note.

- A zero (0) for the field size means python should use as many spaces as necessary to print the number.
- Any number other than zero for the field size indicates the number of characters python should use to print the number, padding with spaces as necessary.
- You can mix and match characters and formats in the format string. Any actual characters in the format string will appear when the string is printed out.
- You can use variables in the argument list, not just hard-coded numbers

4 Mathematics and Programming

Some of the most common things we use computers to do are execute mathematical algorithms, compute values, and simulate phenomena. Most mathematical expressions have straightforward implementations in code.

Functions of the form y = f(x), for example, can be coded as functions that take a parameter and use it to compute and return the desired value.

Example: computing a polynomial

Polynomials are a useful class of functions that have many applications. For example, an Nth-order polynomial can be expressed as the following.

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \tag{3}$$

Note the patterns that exist in the polynomial. The set of coefficients $(a_0, \ldots, a_n)$ can be represented as a list of floating point values. Each exponent for the independent variable x is the same as the subscript for its coefficient.

Given the pattern, we can write a compact function that takes a polynomial, represented as its complete list of coefficients, a value x, and computes its value.

```
import math

# computes the value of a polynomial defined
# by its coefficient list a at location x
def polynomial(a, x):
    # initialize the summation
    sum = 0.0

    # loop over all of the coefficients
    for i in range(len(a)):
        sum += a[i] * math.pow(x, i)

    # return the value
    return sum
```

Note a couple of items.

- The example uses the notation `import math`, which requires the use of object notation to call the math functions. The good part of this is that we don't have to worry about accidentally using the name of a function in the math library, or any other module we import this way. The object notation lets us specify which version of a function to use.
- The loop makes a complex computation easy to do, as it just sums up the individual terms expressed in terms of the loop variable i.

4.1 Iterative Computations

When modeling many real-world systems we have to include time in the equation. Some equations that include time are of the form

$$y = f(x, t) \tag{4}$$

which says that at a particular x at a particular time t the value of y will always be the same. There is no dependence of the value of y on the value of y at the time step before or after.

In many real-world systems, however, the value of the system at the current time is dependent upon the value of the system at a previous time. Such iterative functions can be expressed as:

$$x_t = f(x_{t-1}) \tag{5}$$

Concepts such as the computation of the future value of money can be expressed easily as iterative computations.

$$\Phi_t = (\Phi_{t-1} + \zeta_{t-1} - \eta_{t-1})\gamma \tag{6}$$

But since we're doing computer science, not math, we can use variables with meaningful names and write the same equation as something understandable.

$$\text{Money}_t = (\text{Money}_{t-1} + \text{Income}_{t-1} - \text{Expenses}_{t-1}) \times \text{InterestRate} \tag{7}$$

A useful way to approach writing this equation is to think of it as a function. The input parameters are each the variables on the right side of the equation. The return value is the new money amount on the left side. We may even want to separate out the computation of the appropriate interest rate and incorporate the time period over which the relation is occurring.

```
# computes the amount of money remaining after a month given
# initial amount, income, expenses, and the annual interest rate
# assumes all transactions take place at the beginning of the month
def monthlyCompute( startAmount, income, expenses, interestRate ):

    # the monthly interest rate is the 12th root of the APR
    monthlyrate = (1.0 + interestRate/100.0) ** (1.0/12.0)

    # compute and return the amount of money at the end of the month
    return (startAmount + income - expenses)*(monthlyrate)
```

The natural control structure for an iterative computation is a for loop. The overall structure of iterative computations is that the variable holding the current value (money, in the above example) is both an argument to the iterative function and holds the value on return.

```
value = startvalue
for i in range( iterations ):
    value = func(value, otherParameters)
```

Example: Computing the value of money over time. Note the following.

- Use of a list to hold the month names
- Use of the modulo operator to loop through the month names appropriately
- Use of the modulo operator to print out a header before each year
- Use of formatted strings to produce nice output
- Parameterization of number of months to think easily in terms of years
- Parameterization of expenses as a fraction of income

```
def main():

    income = 1000
    expenses = 0.95 * income
    interestRate = 5.00

    month = ["jan", "feb", "mar", "apr", "may", "jun",
             "jul", "aug", "sep", "oct", "nov", "dec"]

    money = 1000
    years = 1
    duration = 12 * years
    for i in range(duration):
        money = monthlyCompute(money, income, expenses, interestRate)
        print "%s: %.2f " % (month[i%12], money)
        if i % 12

main()
```

```
$ python money.py
-----
Year 00
jan: 1054.28
feb: 1108.78
mar: 1163.50
apr: 1218.44
may: 1273.61
jun: 1329.00
jul: 1384.62
aug: 1440.47
sep: 1496.54
oct: 1552.84
nov: 1609.37
dec: 1666.13
```

4.2 Random Numbers

Random numbers play a big role in computer science, especially for simulating natural phenomena. The real world contains randomness, and in order to simulate the real world we also need to simulate these phenomena. In most computer systems, random numbers are generated using mathematical computations that generate sequences of what are called pseudo-random numbers. The number sequences have properties similar to true random numbers, but they are computed deterministically and, eventually, will start to repeat. The length of the sequence, however, tends to be very large.

Python's random package, for example, uses a method that has a period of $2^{19937} - 1$. So no simulation or computer program you can write will go through the entire period in your lifetime.

The fact that random number generation is deterministic, however, means we can actually repeat a sequence of random numbers if we reset the random number generator to a specific state. This ability can be important when testing a program so that we can compare different runs when looking for bugs.

It is also important to know this because, if you want your program to act differently each time it is run, you may want to seed the random number generator using something that will be different each time the program is run. The current time, for example, is commonly used to seed random number generators.

Example: use of the time package to set a random seed

- Note the use of the import <package> syntax
- Note that the package is treated as an object and functions in the package as methods
- The time package method `time()` returns time since the Epoch as a float
- The Epoch is the start of January 1st, 1970

```
import time

random.seed(time.time())
```

We've already used the ability of the random package to generate random numbers in the range $[0, 1)$.

- The use of a bracket in range notation means the range includes the number on that end of the range
- The use of a parenthesis in range notation means the range does not include the number on that end of the range
- In the case above, the function returns numbers from 0.0 to almost 1.0, but not including 1.0

There are many other functions available in the random package that are useful in various situations.

- `random.uniform(a, b)` - returns a floating point number in the range $[a, b)$
- `random.randint(a, b)` - returns an integer in the range $[a, b]$
- `random.gauss(mu, sigma)` - returns a floating point value drawn from a Gaussian, or normal distribution parameterized by its mean (mu) and standard deviation (sigma)
- `random.choice( list )` - returns a randomly selected element of the argument, which should be a list or string

- `random.shuffle( list )` - shuffles the elements of the list or string in place
- `random.sample( list, N )` - returns a randomly selected set of N elements from the list

4.3 Strings as models

Over the course of the semester we've worked on building a program through which the user can create a graphical scene, save their work, and later read in the saved information and regenerate the scene.

Consider the things your program can do:

- It can save a user's work to a file
- It can read in data from a file and regenerate what the user did
- With a little extra work the user can read in their prior work and continue adding to it

Note also that your program is not saving turtle instructions; it is saving a sequence of characters. It is your interpreter that is converting those character sequences into actions.

What does python do?

- Reads in a file
- Interprets the file and executes the actions specified by the code in the file
- The code is represented as a sequence of characters

What are the rules for your string interpreter?

- The interpreter takes in a string of characters
- Each character is converted into some action using rules of the form
  `character` $\rightarrow$ `action`
- If a character's definition is not defined, the action it takes is the null action
- The interpreter moves sequentially through the character list
- The interpreter has no memory

Using our interpreter we can create interesting geometric models by manipulating strings. The Koch curves, for example, demonstrate how applying a simple rule to a string can generate a complex shape.

The idea is that there is a base string, a symbol, and a rule. One application of the rule replaces all of the symbols in the base string with the rule.

- Base string: F
- Symbol: F
- Rule: F-FF

```
F
F-FF
F-FF-F-FFF-FF
F-FF-F-FFF-FF-F-FF-F-FFF-FFF-FF-F-FFF-FF
```

The application of the rule occurs simultaneously and in parallel for all instances of the symbol in the base string. When the replacement is complete, the resulting string becomes the new base string and the process can repeat.

This system of strings and rules is a simple version of a grammar called an L-system. A grammar is simply a set of symbols and rules for how the symbols can be manipulated and combined. English has a grammar (a very complex one), and computer programs have a grammar (a much less complex one). L-systems also have a grammar, that is generally very simple.

When combined with an alphabet (all the possible strings in the system), the base string, symbol, and rule define all of the possible strings that can be created by the system through repeated application of the rule. Your interpreter then defines how the string is converted into actions.

The Koch snowflake is an example of an L-system with the given meanings for the system.

- Base string: F–F–F–
- Symbol: F
- Rule: F+F–F+F
- Meaning: F is go forward by distance δ (e.g. 5)
- Meaning: + is go left by angle $\theta = 60^\circ$
- Meaning: - is go right by angle $\theta = 60^\circ$

It turns out that L-systems can be used to model biological systems, notably plants. Consider that a plant consists of cells (or groups of cells). A plant grows by cell division.

- Each cell is replaced by two cells
- A new cell may turn into a different kind of cell than its parent
- The cell subdivision all takes place in parallel

A problem occurs for simple serial strings when a cell produces children cells that branch in different directions.

- The children of the branches are no longer sequential
- The children may have additional children

One thing that differentiates the python interpreter from yours is that python has the ability to remember what it has seen before and can attach labels to sequences of code (functions, variables, etc.).

What if we gave our interpreter the ability to remember something very simple, like the current turtle position and heading? Think about how this relates to a tree branch.

- Start at the trunk and go to the first branching
- Remember the current turtle position
- Trace out the right branch
- When finished with the right branch, restore the turtle position
- Keep parsing the string

One issue that comes up is that if the right branch has a branch, then we need to remember another turtle position. If we can only remember one position, we can only recover from one branching event at a time. If we can remember many branches, then we can have a complex tree.

How do we know what turtle position to restore when we finish drawing a branch? It turns out we always want to restore the turtle to the last position we saved. So we want a data structure with the following properties.

- We can add things to the data structure
- The data structure remembers the order in which we added the elements
- When we take something from the data structure it is always the last thing we put on

What is a data structure we could use to store a sequence of turtle positions and headings that would have these properties?

- We can append something to the end of a list: `a = a + [1]`
- We can retrieve something from the end of a list: `value = a[-1]`
- We can remove something from the end of the list: `a = a[:-1]`

In real life, what does this data structure behave like?

- A stack of plates or trays

Turns out the list data structure supports both of these operations (push and pop). The pop method combines removing the last element and returning it in a single operation.

- `a.append(value)`
- `value = a.pop()`

So how would use this idea in our interpreter?

- Define a character for pushing the current state into memory
- Define a character for popping and restoring the turtle state
- The traditional L-system characters are the left and right bracket: `'[', ']'`

When we begin parsing a string, we just need to create a variable and initialize it to the empty list.

- When we see a left bracket character, append the turtle position and heading to the list.
- When we see a right bracket character, pop the turtle heading and position from the list and reset the turtle position and heading.

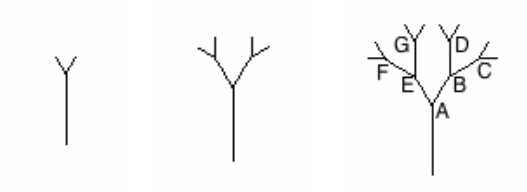
Example: Branching L-system

An example of a simple L-system with branching is as follows:

Base: ffF
Symbol: F
Rule: ff[+F][-F]
Iterations: 3
Angle: 30

The first three strings and their shapes:

```
ffff[+F][-F]
ffff[+ff[+F][-F]][-ff[+F][-F]]
ffff[+ff[+ff[+F][-F]][-ff[+F][-F]]][-ff[+ff[+F][-F]][-ff[+F][-F]]]
```



Consider tracing out the turtle's path for the figure on the right using the labels shown.

Step	Stack	Step	Stack	Step	Stack	Step	Stack
[	A	[	A B	[	A	[	A E
[	A B	[	A B D	[	A E	[	A E G
[	A B C	]	A B	[	A E F	]	A E
]	A B	[	A B D	]	A E	[	A E G
[	A B C	]	A B	[	A E F	]	A E
]	A B	]	A	]	A E	]	A
]	A	]	null	]	A	]	null

In the turtle package, the turtle's position is provided by the `position()` function, which returns a 2-element list containing the x and y location of the turtle. The turtle's heading is provided by the `heading()` function, which returns the orientation of the turtle in the range $[0, 360)$.

We can append things onto the end of a list using the `append()` method. We can take things off the end of a list using the `pop()` method. To consider what they do, the following example shows the equivalent list manipulation statements to append and pop.

```
>>> a = [1, 2, 3]
>>> a = a + [4]
>>> a
[1, 2, 3, 4]
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

```
>>> a = [1, 2, 3]
>>> b = a[-1]
>>> a = a[:-1]
>>> a
[1, 2]
>>> b
3
```

```
>>> a = [1, 2, 3]
>>> b = a.pop()
>>> a
[1, 2]
>>> b
3
```

Review

Variables are labels for locations where data can be stored. In python, any kind of data can be stored in any variable. The rules for variable names determine what kinds of labels you can use.

- A variable is created when you assign something to it.
  `agaboo = 15`
- If you try to use a variable before it has been created, it will cause an error
- To modify the value of a variable, use an assignment statement (=). Whatever is on the right side of an assignment gets put into the variable on the left side.
- Variables can hold data types that consist of sequential elements (lists, strings)
  - Use the bracket notation to access a particular element in the sequence
    ```
    abor = [1, 2, 3]
    print abor[1]
    ```
  - Lists and strings also have methods associated with them that can be called
  - You can use the help() function in python to learn about the methods
- Variables can hold data types that are objects, or classes
  - Classes can contain data of various types
  - Classes can have methods associated with them
  - Use the dot notation (variablename.methodname()) to call a method of a class
- Strings are different than variables
  - A string is denoted by quotes (single, or double)
  - A string is a sequence of characters, it is what it is and nothing else
  - A variable is a label for a memory location that can hold data
  - 'varname' is not the same thing as varname

Functions are one way to organize programs. They let you subdivide a program into smaller and smaller bits until the individual bits are simple to write. Functions can also return values.

- You can use a function that returns a value on the right side of an assignment statement.
- The return value of the function is used to evaluate the expression
- The function is called before the expression can be evaluated
- If you want to use the return value of a function, you have to store it in a variable using an assignment
  `huki = myfunction(5)`
- To return a value from a function, use the `return` keyword

5 Zelle Graphics objects

The Zelle graphics package is organized around the concept of objects. Objects are a different way of subdividing problems than breaking them into steps of a process.

Functions divide a problem into parts by looking at the actions required to achieve a solution. The keys to subdividing a problem into functional parts are:

- Identifying the steps required by the solution and looking for duplication
- Identifying the information that needs to be passed around between functions
- Identifying the input/output characteristics of each function
- Dividing the problem sufficiently so that each function is easy to write

An object-oriented approach to design looks at the problem differently. Instead of breaking down the problem into a series of steps, the problem domain is divided by the objects, or 'nouns' that represent parts of the problem description. Both actions and data are then attributed to the critical objects.

- The objects for a particular problem represent the nouns
- The methods of the objects represent verbs in the problem description
- Adverbs and adjectives represent the data, or parameters required for the methods or objects.

5.1 Working with graphics objects

The basic objects in the Zelle graphics package are:

- GraphWin - a window
- Point - a 2D point with (x, y) values
- Line - a line connecting two points
- Rectangle - an axis-oriented rectangle defined by two points
- Circle - you guessed it
- Text - an object for drawing text in the screen

To create an object, use the name of the object as a function. In many cases, an object constructor will take arguments that get stored inside the object when it is created.

```
edge = Line( Point( 50, 50 ), Point( 100, 200 ) )
```

In the above example, the variable `edge` gets a new Line object that connects the points (50, 50) and (100, 200). The example also creates two ephemeral, or temporary Point objects that get passed in as parameters to the Line constructor function. Neither of the Point() objects gets assigned to a variable, so once the Line constructor returns, both Point objects disappear. This is fine, and demonstrates how we can create temporary objects as necessary.

5.1.1 Object methods

Most objects have methods associated with them. A method is simply function that acts on the object. All methods of an object have the object itself as the default first argument. When a method is called, the object on which the method is called is automatically placed as the first argument, which means the programmer does not have to.

Therefore, a method that takes no external arguments, will still have the default self argument.

5.1.2 Object assignment and copying

A significant difference between objects and the standard data types is what happens when you assign an object to a different variable. Consider the two cases below:

Case 1:

```
>>> goofy = 10
>>> pluto = goofy
>>> print goofy, pluto
10 10
>>> pluto = 20
>>> print goofy, pluto
10 20
```

Case 2:

```
>>> from graphics import *
>>> hu = Point(10, 10)
>>> lu = hu
>>> print hu.getX(), hu.getY(), lu.getX(), lu.getY()
10 10 10 10
>>> lu.move(20, 20)
>>> print hu.getX(), hu.getY(), lu.getX(), lu.getY()
30 30 30 30
```

What happened in the second case? Why did hu change when we moved lu?

- When an object is created, the constructor returns a reference to the object
- A reference is like an address, it's the location of the object data, not the object itself
- When a basic data type is placed into a variable, the variable holds the data itself
- When an object is placed into a variable, the variable holds a reference to the object

If we continue to think of variables as labels placed on memory locations, then when a basic data type is placed into a variable, the location in memory that holds the data is given the variable label.

When an object is placed into a variable, the variable becomes a label for a location in memory that holds the address of the actual object data. Therefore, if we copy what is in a variable referencing an object into another variable, the new variable gets a copy of the address and both variables end up referencing the same place in memory.

So how do we make a copy of an object? The objects in the Zelle graphics library all include a `clone()` function that makes a copy of the object's data and then returns a reference to the location of the new copy. If we execute the same example as above, but use the clone method instead of a straight assignment, then we get behavior that matches assignments with the basic data types.

```
>>> from graphics import *
>>> hu =Point(10, 10)
>>> lu = hu.clone()
>>> print hu.getX(), hu.getY(), lu.getX(), lu.getY()
10 10 10 10
>>> lu.move(20, 20)
>>> print hu.getX(), hu.getY(), lu.getX(), lu.getY()
10 10 30 30
```

So to make a real copy of an object use the clone method on the right side of the assignment, not just the variable holding the reference to the object.

5.2 GUI Elements

GUIs define how we interact with a computer. The physical aspects of most GUIs include pointing devices (mouse) and buttons (keyboard, mouse buttons). The computer can capture these inputs and a programmer can specify how the computer reacts.

GUIs have developed over time, and there are many expectations a user has about the way certain visual elements react. If a programmer violates those expectations, it can result in poor performance by the user on the task.

Aspects of GUIs

- Mouse
- Keyboard
- Windows
- Menus
- Buttons
- Sliders
- Cursor
- what else?

Event-based programming

- Generally a main loop
- Each time through the loop, check for events
- Each time through the loop, update anything that needs updating

The Zelle graphics package provides the ability to get information about mouse clicks–including waiting for it to be clicked–and to get text strings from entry boxes in the graphics window. The GraphWin object contains several methods for finding out about the window and the mouse.

- getMouse(): waits for the mouse to be clicked and returns the location
- checkMouse(): returns position of the last click or None (special value)

These functions provide a limited ability to generate GUIs. But you can get most of the usual elements. For example, a click can:

- Activate a button
- Activate a menu (show/hide)
- Activate a slider

In addition, the Entry object creates a text box into which the user can enter text.

- Entry(centerPoint, numCharacters): creates an entry object that lets the user enter some text
- getAnchor(): returns a clone of the center point
- getText(): returns the string that is currently in the box
- setText(string): sets the string in the box
- setSize(point): changes the font size; valid values are [5, 36]
- setStyle(style): changes the font style according to the provide string; valid values are 'normal', 'bold', 'italic' and 'bold italic'
- setTextColor(color): sets the color of the font using the Zelle color methods: either a string or the output of the `color_rgb()` function

Image fun: you can also put images onto the screen

- Image(center, filename): creates an image object from the file specified
- getAnchor(): returns a clone of the center point

Using the getMouse() function, it's straightforward to identify important locations in the scene and use them to create new objects.

5.3 Managing Objects

Whether they are GUI objects or graphical objects, a single scene contains many, many objects within it. If all we want to do is draw static shapes in an image then we don't care about the objects after they are drawn. When an object is no longer needed, any references to the object can be used for something else.

There are some important issues that come up when thinking about objects and references.

- When all references to an object are redirected elsewhere or eliminated, can you get access to the object any more?

  If there are no references left to an object, then the object is effectively lost. There is no reasonable way to figure out where the object was located in memory because memory is just data. Even if you knew the exact contents of the object (in which case, why do you need the object?), there is no guarantee that only one location in memory contains that exact data. If you were to make a mistake in identifying where the object was, you run the risk of corrupting memory that is being used for something else.

- What becomes of the memory used by an object with no more references?

When an object has no more references, python will clean up the memory used by that object. Even when your program has no more references to an object, python keeps track of where all objects are located in memory. It also keeps track of how many references to an object exist by watching variable assignments. When no more references to an object exist, python knows it can get rid of the object and use the memory for something else.

If we want to be able to manipulate objects after they have been created, then we need to keep around references to those objects within our program. How should we go about organizing the objects in an effective and useful way?

- Consider separating the acts of object creation and object manipulation
  - Have the first part of the building function create and store references to all of the objects
  - Have the second part of the building function draw the objects
  - Have the building function return all of the constituent objects
- What data structure makes sense for storing the objects constituting a building?

  A list makes a lot of sense here. It is dynamically sized, and we can use conventions like the outline is the first item on the list, the door is the second item, and the windows are the remaining items. Note that the convention is completely up to you to define. If you have two doors, for example, you could make them the 2nd and 3rd items on the list, or you could make the second item in the list be a list of objects constituting the door. A list also lets you pass around lots of objects within a single variable.
- Write functions that manipulate the items on the list to save coding and reduce errors
  - buildingMove(bldg, dx, dy) - move the building elements by (dx, dy)
  - buildingDraw(bldg, win) - draw the building elements given their current state
  - buildingAnimate(bldg) - animate the building elements
  - buildingClone(bldg) - clone the building elements and return a list containing the clones

Consider what we have done:

- We have a function `building(x0, y0, dx)` that creates a building, initializes it using the given parameters, and returns an object (a list) that contains all of the building elements.
- We have a set of functions that manipulate the building, treating the building as a single object
- Each function takes in the list of building elements as the first argument

Effectively, what we have done in organizing the code this way is to create a building object and a set of methods that apply to that object.

- All of the data required to draw or manipulate the building is in the building object (the list)
- The building is a noun, a thing we want to manipulate
- The methods are verbs, actions that the building can execute
- The methods all know how to manipulate the building data
- The top level program doesn't need to know anything about how the building elements are organized

Summary of first half of the semester:

- data movement
- data storage
- data manipulation
- control flow
- memory model: objects/references v. data types

6 Classes and Objects

What if we want to define our own classes so we can organize the code in an object-oriented manner?

What are the parts of a class?

- Class name, so that a programmer can create objects of the new type
- Class constructor that makes the object
- Class data that holds the relevant information about the object
- Class methods that let us manipulate the object and the object's data
- A syntax to organize it all

Class definition:

```
class <class name>:

  def __init__(self, <other arguments>):
    <constructor code>

  def <member function name>(self, <other arguments>):
    <member function code>
```

The key differences between methods and regular functions are that methods are defined inside a class structure and that the first argument to a method is always the self variable, which is a reference to the object to which the method is to be applied. Otherwise, they act just like regular functions.

The init function is a special function that is called when the user creates an instance of an object by calling the function defined by the class name. For example, in the Zelle graphics package, when you call the function GraphWin(), then the init function for the GraphWin class is called with whatever parameters you pass to the GraphWin() call.

All function and method definitions can specify default values for arguments if the arguments are not given. This is commonly used in the constructor for a class so that an appropriate object is created even if the programmer provides no initial arguments to the object. If an object must have certain parameters, then there should not be any default values for those parameters.

Example: drawing data

Consider an example where we have a set of (x, y) data in a file. We could create such a data set using:

```
import random

fp = file( 'somedata.txt', 'w' )
for i in range(50):
    x = random.gauss(0, 50)
    y = random.gauss(0, 25)
    s = '%10.4f %10.4f\n' % (x, y)
    fp.write( s )

fp.close()
```

Now we want to read in the file and plot the data. Each data point we can treat as an object. Each datum should have a constructor and know how to draw itself, given the appropriate parameters.

```
import graphics

class Datum:
    def __init__(self, tx, ty):
        self.x = tx
        self.y = ty

    def draw(self, x0, y0, dx, dy, win):
        tx = 0 + win.getWidth() * (self.x - x0) / dx
        ty = win.getHeight() + win.getHeight() * (y0 - self.y) / dy

        p = graphics.Point( tx, ty )
        p.setFill( 'blue' )
        p.draw(win)
```

The constructor requires two arguments besides the object itself, which are the x and y values of the data point (in the data space). It simply creates two new fields in the object and stores the values in those fields.

The draw function takes in the data space window to look at (x0, y0, dx, dy) and figures out where to plot the data point. It also decides what the data point will look like. The actual main function is very simple.

```
def main():
    fp = file('somedata.txt', 'r')
    data = []
    for line in fp:
        slist = line.split()
        x = float(slist[0])
        y = float(slist[1])
        data.append( Datum( x, y ) )

    win = graphics.GraphWin('title', 400, 400)
    for datum in data:
        datum.draw( -100, -100, 200, 200, win )

    win.getMouse()
```

Because all of the complexity of drawing each data point is handled by the object, the main function doesn't need to know anything about it. We can also go and change the way a point is drawn in the graph without changing the main function at all. This is the concept of encapsulation. Encapsulation is isolating the details of code from a program that uses an object. So long as the object does the required action, how that is accomplished is irrelevant to the program using the objects.

Encapsulation is one of the primary design principles. The idea is that we want to separate functionality from implementation. That way, any code that uses an object does not have to change if the object's implementation changes.

6.1 Class Design Process

How would we implement a Face as an object that acts just like a Zelle graphics object?

- Define the thing we want to create: outline, nose, eyes, mouth
- Identify the key descriptors, or fields: location, size, anything else?
- Identify the methods the class needs to implement: the Zelle graphics functions for a Circle are a good choice for a Face
  - draw - takes in a window and draws the face given the current state of the face
  - undraw - undraws the face
  - move - moves all elements of the face
  - setFill - set the fill color for the outline circle
  - setOutline - set the outline color for the face
  - setWidth - change the width of the lines on the face
  - clone - generate a duplicate of the object (undrawn)
  - getRadius - get the radius of the face
  - getCenter - get the center of the face
  - getP1 - get the upper left corner of the bounding box around the face
  - getP2 - get the upper left corner of the bounding box around the face
- Identify any other methods we might want to implement
  - construct - a helper function that takes care of all the details of building the face
  - resize - takes in a new radius and changes the size of the face
  - wink - changes the state of one eye
  - speak - changes the state of the mouth
  - makeBubble - takes in some text and generates a bubble with the text in it
  - deleteBubble - removes the bubble

- Create a skeleton for the class that identifies the methods and their inputs and outputs
- Provide help text for the class and for each method using the triple quote method
- Create the skeleton of a test method
  - When a file is run from the command line, a special variable called `__name__` takes on the value `'__main__'`. By testing for that, we can run the test program if that condition is met, but not execute any code otherwise. That allows us to both run a python program and import it into other programs without running the test code. The test function can be part of the class, but usually is not.
- Start writing methods, adding to the test method as they get written. The constructor should be the first thing you write, and it generally defines all of the fields and gives them default values. We use the construct method as the basis for creating the objects because building an object may be used by several of the functions (init and resize both will generate new objects).
- Iterate, creating new methods and fields as necessary

6.1.1 Notes on Class Design

When creating a class, it is common to want to set the descriptors for the task to new values. We may want to do this, for example, when the object is created, when data is read from a file, or through a function that resets the values of the descriptors. If we write code to create the object in three different places, however, then we are likely to end up with problems. Changes in one of the three programs will not be propagated to the other two.

The solution is to have a single constructor function that is used by all three methods (constructor, read, and set) and that takes in all the parameters necessary to set up the fields of the object. The `construct` function plays that role in the Face class.

In the process of writing methods, we may discover that we want to add new fields to the class. For example, if we give the Face the ability to add a bubble, we may want to store a reference to the window in which the face has been drawn, and keep track of whether it's visible or not. We can do that with a field in the class initially set to None, setting it to the window reference when the face is drawn, and setting it back to None when the face is undrawn.

Adding fields to classes should be done parsimoniously to minimize memory usage and maximize speed. However, if functionality requires a flag or a field, don't hesitate to add it to the class. If the field is used only by methods within the class, then you don't need to create any accessor or set methods. If the field is intended to be changed or modified by someone using objects of the class, then you will need to provide accessor and set methods for the field.

- **Accessor method**: a method that returns a copy of the requested value
- **Set method**: a method that takes in a new value for an internal field of a class

In general, we don't want to return references to objects internal to the class because it can cause unexpected consequences. Instead, accessor methods should return clones of internal objects or simple data values. In general, you want to control the flow of operations through method calls rather than letting someone manipulate internal object values directly.

Some characteristics of classes that you'll tend to see include:

- Accessor methods that just return values, or copies of values, tend to be trivially simple. Any class that stores data will generally have accessor functions for most of the data values.
- In addition to a set method that sets up all of the fields of a class, there will often be many set type methods for modifying the value of individual fields in a class.
- Many methods in a class will be short sequences of code that get used often (think of the draw method, for example). Think of the methods as shorthand for a particular sequence.

Example: The Datum and DataCollection class from the current assignment are a variation on the Zelle Graphics objects

Basic idea:

- Graphing and data analysis are common and useful tasks
- Graphing requires that we have a visual representation of data
- The graphics package gives us the ability to draw things to a screen
- The Datum class connects the basic data with the visual representation of it

When working with data, like height versus shoe size, the data is in a space that is meaningful to it. If we want to make a graph, however, that means we need to put points on a window in the image. Those points must be in pixel coordinates for the window, which has nothing to do with the data space. Therefore, the Datum class has both data fields (held in x, y, z) and window location fields (held in x0, y0, dr, and color).

By defining a set of methods that duplicate the methods of the Zelle graphics objects, the Datum object effectively becomes a graphics object, and we can use it as such. However, it also has the associated data information. In that sense, it plays two roles. (Having multiple roles is not necessarily good object-oriented design, however, so it might be worth reworking this design next time around.)

The DataCollection object is the glue that connects those two roles together. It knows how to visually position the Datum objects based on their values and the characteristics of a graph. By also giving the DataCollection object the types of methods associated with a Zelle graphics object, it too can be treated as a simple graphics object from the point of view of drawing, moving, and undrawing the graph. The benefit is that the DataCollection object can then be used as part of a program to read, graph, and manipulate data visually.

6.2 Polymorpishm

Polymorphism refers to the ability of a variable or function to have several meanings or actions. When referring to a variable, polymorphism is the ability of a variable to hold different data types.

- Python variables are naturally polymorphic: they can take on any data type
- Python lists are polymorphic, because the elements of the list can be any data type
- In other languages, variables may have types, and only certain kinds of variables can be polymorphic

When referring to a function, polymorphism is the ability of a function name to have several different meanings. In most programming languages that allow polymorphic functions, the difference between the functions is usually the number of parameters.

- Python does not allow generic function polymorphism. Only the most recently defined version of a function is used by the interpreter, whether it is a regular function or a class. Creating a new function or method with the same name overwrites the old function definition. This is important to remember, because mistakenly creating a duplicate function name can cause problems.
- Python does allow you to define functions with default variable values. This capability effectively lets a function be polymorphic, in that it can be called with different numbers of parameters.

Example: A polymorphic function

```
def func1(a, b=0, c=0, d=0):
    return a + b + c + d

print func1(5)
print func1(5, 5)
print func1(5, 5, 5)
print func1(5, 5, 5, 5)
print func1(5, d=6)
print func1(c=12, a=10)
```

The above example shows a number of different capabilities.

- By giving an argument a default value, that makes the argument optional when the function is called.
- When calling a function with just the values, the values line up with the argument list.
- It is possible to put the arguments out of order if you use the name of the argument and given it a value when calling the function

The last capability applies to any function and any argument set. When named in a function call, the arguments do not need to be in order. The only requirement is that arguments without a default value must be in the argument list when the function is called.

Polymorphism, along with encapsulation (hiding the details of data storage and code), is one of the primary object-oriented software design principles. We've already seen the benefits of polymorphism in the building and peace symbol code: we can use a list to hold many different types of graphics objects. Further, since all of those objects support the same basic set of methods, we can write programs that use those methods without worrying about the particular type of the object in the list.

6.3 Polymorphism in Standard Operations

Looking at most of the standard data types and packages (list, string, random, time) there are many functions that are polymorphic in nature: they have one or more default arguments.

Looking in the `random` package, the `seed` function, for example, takes an optional argument. The default behavior is to set the seed from current time–something that is commonly done by developers. If the

argument is provided, then the argument's value is used as the seed.

The string package also contains many functions with optional arguments. The following are two examples.

- `atoi(s, base=10)` - converts a string into an integer; the optional argument sets the base of the number
- `split(s, sep=None, maxsplit=-1)` - splits a string into a list of words and returns a list containing all of the words. The first optional argument specifies what character separates the words (default is white space) and the second default argument is the maximum number of words to put into the list.

Lists are a naturally polymorphic object because they hold a collection of elements that can be of any type. The following is perfectly fine python.

```
q = [1, 'a', [3, 2], 4.5, 'blah' ]
```

List also have a number of polymorphic functions that let the programmer use either the default behavior or specify it explicitly.

- `pop([index])` - remove and return the item at index (default is last item in the list)
- `index(value, [start, [stop]])` - return the index of the first occurrence of value between optional arguments start and stop.
- `sort(compfunc = None)` - sort the list in place

The default behavior for the sort function is to use the less than operator to determine the order in which the elements should be placed.

- For the basic data types, less than is well defined: $4.5 < 5.5$ or $3 < 8$
- What about for more complex data types like objects with multiple fields?
- What if we want to sort in reverse order?

In order to sort lists of more complex objects, we need a way to tell the sort function how to order two elements of the list. Rather than write two sort functions, the sort method of the list class takes an optional argument: a function that determines the ordering of two elements of the list.

Functions are blocks of code. They are text in a file, and once they are converted into machine language, they consist of blocks of memory filled with instructions. When a function is defined, the name of the function becomes a reference to the block of code defining the function's actions. This is no different than an object.

- The data for an object is a chunk of memory
- A variable can hold a reference to the object
- When a method is applied to the variable, it uses the reference to get access to the object's data (self)

Likewise, for functions:

- The data for a function (its code) is a chunk of memory
- The function name holds a reference to the code
- When a function is called, the reference specifies the code that should be executed

Python treats functions and objects similarly. The function's name is a variable that is a reference to a function. It works the same way as a reference to an object.

- Have to be careful, you can overwrite a function definition by simply assigning something to the function name.
- But you can assign a function reference to a variable
- You can pass function references as arguments to other functions

---

Example: sorting Datum objects

```
class Datum:
    """Class for holding scientific data and drawing it on screen"""

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def toString(self):
        return "%8.2f %8.2f %8.2f" % (self.x, self.y, self.z)

def datumsortX(a, b):
    """Utility function that sorts two Datum objects based on their X value"""
    if a.x < b.x: return -1
    elif a.x > b.x: return 1
    else: return 0

def datumTest():
    collection = []
    for i in range(5):
        collection.append( Datum( random.random(),
                                  random.random(),
                                  random.random() ) )

    print '----------------\ninitial order'
    for dm in collection:
        print dm.toString()

    print '----------------\ndefault sort'
    collection.sort()
    for dm in collection:
        print dm.toString()

    print '----------------\nSort on X'
    collection.sort(datumsortX)
    for dm in collection:
        print dm.toString()
```

6.4 Inheritance

Consider the aggregate graphics objects we've created so far.

- Peace symbol
- Building
- Face

What did they all have in common?

- All have an anchor point (x0, y0) independent of their components
- All have a scale factor (dx) independent of their components
- All have a list to hold the components
- If we use the same field names, then we can write common algorithms
  - draw
  - undraw
  - move
  - setFill
  - setOutline
  - clone

The most significant difference between the three classes is the set of objects and their relative positions. We would like to avoid duplication of fields and code, if possible.

- Duplication of code creates more potential for mistakes
- Consolidating code reduces time spent programming
- Consolidated code is easier to debug

The model developed to represent this kind of relationship is called inheritance. A parent class contains all of the common code and data fields, while the child, or derived class contains the unique data or methods required for implementation.

The concept is similar to a taxonomic tree in biology. Mammals all share a set of characteristics that separate them from other creatures, but individual mammal species each have unique attributes. The concept of inheritance is extremely powerful, because it permits you to leverage code many times. Code written for a parent class is reused for each child class.

Note that inheritance is more than simply writing functions outside of a class structure that assume a particular design for the fields of a number of different classes. Inheritance, by incorporating the methods inside the parent class, follows the principle of encapsulation: only the programmer writing the parent class needs to know the particulars of implementation.

Example: Parent aggregate class and two child classes.

Peace symbol

Target

Billboard

Note power of leveraging common code

Note where methods need to be overwritten

- the init methods build different lists of objects
- the setFill method may need to be different
- clone method will be different if the objects have different fields

---

6.5 Review of Objects and Object-Oriented Design

Object-oriented design principles

- **Modularity**: making functional units that can be re-used in many contexts
- **Encapsulation**: isolating implementation from function
- **Polymorphism**: the ability to treat different objects the same way
- **Inheritance**: capturing commonality in a base class that can be extended to handle special cases

Objects are a collection of data and algorithms the form a coherent package. We describe an object as a noun and methods as verbs, or actions the noun can take, but the important aspect of the analogy is that the object is supposed to be meaningful and well-defined.

- Objects generally are limited in their scope and functionality
- The number and type of things an object can do (its methods) are generally limited in scope
- When an object takes on the role of more than one thing (noun) it makes the design process more difficult
- Objects are the primary mechanism for achieving encapsulation in a regular and rigorous manner
- Objects are the primary mechanism for enabling inheritance (code and data sharing)
- Objects naturally encourage modularity
- Objects naturally enable polymorphism though child methods overwriting parent methods
- Objects are both an abstraction that lets us organize large projects and a syntactic mechanism to encourage structured programming

Objects do not allow the computer to do anything it couldn't do otherwise. Computers can still only do four things: move, store, manipulate, and adjust control flow. They are simply an abstraction that lets us put sequences of code and data into collections with which we can reason and build even more abstract concepts.

7 L-Systems

Deterministic, context free L-systems, called DOL-systems, are the simplest form of L-systems.

- Each symbol has a single replacement rule (deterministic)
- Adjacent symbols do not affect what rules apply to a given symbol (context free)
- All rules are applied simultaneously to the base string

Example: Multiple rule L-system

$a \rightarrow ab$
$b \rightarrow c$
$c \rightarrow a$

Iteration	String
0	a
1	ab
2	abc
3	abca
4	abcaab
5	abcaababc

In order to implement multiple rules, we need to simulate simultaneous expansion of each symbol by its replacement rule. Since we are implementing the concept on a serial machine, however, we have to keep track of what symbols have been replaced and which have not.

- Replacing all instances of one symbol and then all instances of another symbol will not work
- The collection of replacement strings must be kept separate from the original string

With DOL-systems, since each symbol is independent in its rule selection and replacement, we can process the base string from left to right. Each symbol in the base string is a key for looking up its replacement rule. We can then concatenate the replacement strings into a new list.

Algorithm for a single iteration of multi-rule replacement:

- Initialize the output string to the empty string.
- For each character C in the base string
  1. Loop through the list of rules
     - If the symbol for the rule is C, then append the rule to the output string and break
  2. If there is no rule for C, the append C onto the output string (identity rule)

If we implement the above algorithm using a list and an inner loop, then we need to keep track of whether the rule has been found. A simple flag variable that gets set to false prior to looping through the list of rules is sufficient. If the rule is found, the flag variable gets set to True.

What is the computational efficiency of this algorithm?

- How many times does the whole thing iterate?
- How many times does the outer loop iterate?
- How many times does the inner loop iterate?

Since nested for loops multiply, the expression for the number of iterations is:

$$\text{Instructions} \approx \text{Iterations} \times \text{String Length} \times \text{Rules} \tag{8}$$

This kind of approximate calculation is important for designing efficient programs. If we know all of the parameters to a program that affect the number of instructions that will be executed, we can describe the computational complexity as some formula based on those numbers.

- The exact number of instructions isn't as important
- Knowing how the length of the computation will change for different inputs is important
- There is some factor K that multiplies the above expression to tell us the exact number of instructions
- The factor K doesn't change

How could we improve the algorithm? What factor could we potentially modify?

- We can't change the number of iterations
- We have to process every character in the base string
- Can we more efficiently find the proper rule?

What if we could use the symbol as a lookup, like an index into a list or string?

Characters are just numbers between 0..255. (chr(a) / ord(a)) What if we created an array with 255 entries?

- ord(x) - returns the number representation of the character x
- chr(x) - returns the character representation of the number x

Replacement algorithm using an indexed array:

1. Initialize the array to the identity rule
2. Replace the rules for symbols with non-identity rules
3. Set base to the base string
4. For the number of iterations
   (a) Set resultstring to the empty string
   (b) For each character ch in base
      - Add the rule indexed by ord(ch) to resultstring
   (c) Set base to the value in resultstring
5. Return base

The algorithm above removes the need to loop through the rules for each character. This reduces the complexity of the algorithm to just $N \approx$ Iterations $\times$ Characters.

Issues:

- The trick only works for the 255 individual characters
- What if we have more rules, or maps, than 255?
- How do we avoid wasting so much space?

7.1 Dictionaries

We'd like something like a dictionary:

- organized for easy lookup
- only those words with definitions are actually in the dictionary

Because the idea of mapping a key to a value is so common, Python provides a data structure with that capability called a dictionary. A dictionary acts in many ways like a list or a string. The difference is that, instead of using numbers to index into the dictionary, you use keys. A key can be any non-mutable value.

Non-mutable (changeable) values include:

- Strings
- Numbers (both floating point and integer types)
- Boolean values
- None
- Tuples of non-mutable types - a comma separated list of values, possibly in parentheses

---

What's a tuple?

A tuple is any comma separated list of items, possibly enclosed in parentheses. It is simply an ordered sequence of things. Tuples support a variety of operations and behave a lot like lists. The difference is that a tuple cannot be changed, similar to a string.

A tuple can appear on the left or right side of an expression. If it appears on the left side, then each element of the tuple must be a mutable variable. The mapping from right to left is simple ordering.

```
>>> x, y = 4, 5
>>> x
4
>>> y
5
>>> 3, y = 6, 9
SyntaxError: can't assign to literal
```

A dictionary uses curly-brackets instead of the square brackets like a list. You create an entry in a dictionary by assigning a value to the dictionary indexed by the keyword.

```
d = {}
d['key'] = 'value'
```

Just like a list, the value of a dictionary entry can be any type (numbers, strings, lists, objects, etc.). Also like a list, there are a number of useful methods for dictionaries.

- keys() - returns the list of keys in the dictionary, which can be useful for looping over the dictionary entries.
- has_key(key) - returns True if there is an entry in the dictionary for the value in key.
- values() - returns a list of the values in the dictionary.
- items() - returns a list of the key-value pairs in the dictionary as 2-element tuples.
- clear() - deletes all entries in the dictionary.
- get(key, default) - returns the value for key or the value in default if the entry for key doesn't exist.

To delete a single entry in a dictionary, the syntax is

```
del d[key]
```

which deletes the entry indexed by key from dictionary d.

Example: Multiple rule string replacement using a dictionary.

```
# initialize base and number of iterations
base = 'a'
iterations = 4

# initialize the rules
rules = {}
rules[ 'a' ] = 'abc'
rules[ 'b' ] = 'a'

# for the number of iterations
for i in range(iterations):

    # initialize the temporary result string to the empty string
    resultstring = ""

    # for each character in the base string, replace it with the appropriate rule
    for ch in base:
        resultstring = resultstring + rules.get(ch, ch)

    # copy resultstring back to base for the next iteration
    base = resultstring

print '\nFinal'
print base
```

7.1.1 How do dictionaries work?

Think about using three digit numbers as keys and strings as the values. Say, for example, that we wanted to map the number 123 to the string 'thunder' and the number 456 to the string 'lightning.

```
d = {}
d[123] = 'thunder'
d[456] = 'lightning'
```

Since we see the user has given us a number with 3 digits, we could create an array with 1000 spaces and use the 3 digit number as an index. That would fill up two locations in the 1000 element array with values, leaving the rest blank. While it's easy to map the key to an index in the array, that seems like a big waste of memory unless we store a lot more values in the dictionary.

What if, when we create the dictionary, we only create an array with a small number of spaces, like ten? How could we map the three digit numbers to a one digit number? One method is to use a function, like the modulo operation. If N is the number of elements in the array, we can find the array index by taking the key value modulo N.

```
index = key % N
```

The function that converts a key into an array index is called a hash function. Using the modulo N hash function, the number 123 maps to index 3 and the number 456 maps to index 6. It is important to note that, since we cannot recreate the original key from an index (the index 3 maps to any key ending in 3), we have to store the key along with the value in the dictionary entry. For example, if someone called the has_key() method with the number 153, the dictionary has to be able to return False, even though there is an entry at index 3 from the number 123.

There are still some problems with trying to map a large space into a small space.

- What if we use the keys 123 and 453, which map to the same space in the array?
- What if we try to store more than 10 items in the dictionary?

To solve the first problem, the dictionary has to have a method of conflict resolution. If the programmer were to add the line

```
d[233] = 'rain'
```

to the dictionary, the modulo hash function would say to put the string 'rain' at location 3. But location 3 already has 'thunder' in it. We could have the dictionary store multiple items in a single array entry, using a list to hold all of the key-value pairs. That means the dictionary would need to search the list of key-value pairs for any array locations with multiple entries.

An alternative is to have a standard algorithm for finding an empty space. For example, the dictionary could just keep adding one to the index until it found an empty space in the array. To discover if an entry is in the dictionary, if the key is not in the proper array location, it would need to search consecutive spaces until it either finds the proper key or finds an empty space.

When the dictionary runs out of space, it will need to create more. Generally, this is done by doubling the size of the dictionary and copying all of the current entries into new locations, using a different hash function.

7.1.2 Building Dictionaries

How do we go about building and using dictionaries in a useful way? Some common things we may want to do include:

- Building a dictionary from a list of key-value pairs
- Building a dictionary from a file
- Building a dictionary in an on-line manner as we read in data

Case 1: Given a list of key-value pairs, how would we build a dictionary from it?

- Given: a list of 2-element lists or tuples
- Initialize an empty dictionary
- Loop over the list of key-value pairs
- Index the dictionary using the key and assign it the value

```
dict = {}

for pair in keyValueList:
    dict[ pair[0] ] = pair[1]
```

The most important line is the assignment statement in the for loop. The right side of the assignment is the value to be stored in the dictionary. The left side creates (or accesses) the dictionary entry associated with the key and assigns the value to it.

Case 2: Given a file with columns of data, how would we build a dictionary from it?

The basic idea is to read through the file line by line. One of the columns will play the role of the key, while the remaining columns will be the value field. In this case, we have to make a decision about how to store the data in the value field. Some options include:

- Store the items as a list (mutable)
- Store the items as a tuple (non-mutable)

The actual process of reading the data involves reading a line, splitting the line into strings, processing each string and storing the data into the entry indexed by the key.

```
fp = file( 'filename', 'r' )

dict = {}

while True:
    line = fp.readline()
    if line == '':
     break

    line = line[:-1]       # remove the newline
    line = line.split()    # split

    dict[ line[0] ] = line[1:]
```

Case 3: Consider the case of reading in a text document and counting the occurrence of each word.

The basic idea is to go through the file word by word.

- Grab the next word
- If the program has not seen it before, create a dictionary entry with the value 1 in it
- Otherwise, add one to the existing value of the dictionary entry for the word.

When the process is complete, each unique word will have its own entry in the dictionary along with a count of how many times that word was used. The syntax of the inner loop is straightforward.

```
counter = {}

for w in words:
    counter[w] = counter.get(w, 0) + 1
```

The above three lines do a number of things.

- The first line creates an empty dictionary
- The second line loops through all of the things in the words list.
- The right side of the third line first checks if w is a valid key using the get method. If it is valid, the word has been seen before and the get method returns the current value of the dictionary entry. If it is invalid, then the get method returns 0. In both cases, the result is added to one.
- The assignment statement then takes the value generated by the right side and puts it into the dictionary entry for the word in w, creating the entry if it doesn't yet exist.

Example: Reading a document and counting word frequency.

```
# a comparison function that uses the second item in a list, tuple, or string
def wordsort(a, b):
    if a[1] > b[1]:
        return -1
    elif a[1] < b[1]:
        return 1
    return 0

# open file and initialize variables
fp = file('constitution.txt', 'r')
counter = {}
totalWords = 0

# loop over all of the lines of the file
while True:

    # read a line and check for an empty line (EOF)
    line = fp.readline()
    if line == '':
        break

    # go through and remove commas and periods
    newline = ''
    for c in line:
        if not (c == '.' or c == ','):
            newline += c

    # split newline and put the result into a words list
    words = newline.split()

    # build the dictionary, creating new entries as new words get found
    for w in words:
        counter[w.lower()] = counter.get(w.lower(), 0) + 1
        # keep track of the total number of words
        totalWords += 1

# number of words is the length of the dictionary
numWords = len(counter)
print 'Words in document:      ', totalWords
print 'Number of words:        ', numWords

# get an list of key - value pairs
wordlist = counter.items()

# sort the list
wordlist.sort(wordsort)

# print out the top 20 words
show = 20
print '\nTop', show,'words:\n'
for i in range( show ):
    print wordlist[i][0], " : ", wordlist[i][1]
```

7.2 L-systems with Stochastic Rule Selection

The next modification to L-systems we're going to make is to permit more than one rule per symbol to exist. During replacement, one of the rules is randomly selected.

The first thing we need to do is decide how to represent multiple replacement strings for a given symbol. The current method of representing a single rule is to use a list where the first element of the list is the symbol, and the second element of the list is the replacement string.

```
[ 'F', 'f[+F]F[-F]F' ]
```

The complete set of rules is simply a list of the individual rules (a list of lists).

```
rules = [ [ 'F', 'f[+F]F[-F]F' ], [ 'f', 'ff' ] ]
```

We have two options for representing multiple rules for a single symbol.

1. Treat the two rules for the same symbol as completely separate rules.

   ```
   rules = [ [ 'F', 'F[+F]F[-F]F' ], [ 'F', 'F[-F]F[++F]' ], [ 'f', 'ff' ] ]
   ```

2. Keep a single rule (list) per symbol, but have multiple replacement strings.

   ```
   rules = [ [ 'F', 'F[+F]F[-F]F', 'F[-F]F[++F]' ], [ 'f', 'ff' ] ]
   ```

What are the benefits of keeping the rules separate?

- We can treat each rule the same at the rule level: one symbol, one replacement
- We don't have to change any existing rules to add or subtract rules for a symbol

What are the benefits of putting all the possible replacement strings in one place?

- We don't have to search for all the rules that apply to a given symbol
- We can still use a dictionary to keep track of the rules
- One symbol, one entry in the list of rules

While both options are workable, we're going to follow the second: putting all the replacements for a given symbol into the same list. The way to think about a single rule, therefore, is that if we have a variable rule that holds a single rule with multiple replacements

```
rule = [ 'F', 'F[+F]F[-F]F', 'F[-F]F[++F]' ]
```

The symbol is in `rule[0]` and the possible replacement symbols are in the list `rule[1:]`. Therefore, is we wanted to randomly choose a replacement rule, we could use the choice function in the random package.

```
random.choice( rule[1:]  )
```

The choice function selects a random element of the list given to it, with each element of the list having the same probability of being selected.

Example: L-system generation with multiple rules and duplicate rules

```
def generateString(self):
    """Uses the L-system parameters to create and return a string"""
    # students

    #make a copy of the base string
    base = self.baseString

    #iterate
    for i in range(self.iterations):

        newbase = ''
        for ch in base:

            if self.rules.has_key(ch):
                newbase += random.choice( self.rules[ch] )
            else:
                newbase += ch

        base = newbase

    # return the string
    return base
```

Because we only have a single entry in the dictionary for each symbol, the algorithm looks almost identical to the case where there is only one replacement string for each symbol. The only real change is that we are using the random.choice() function to select one of the replacement rules in the dictionary entry (which is now a list of replacement strings).

7.3 L-systems and Non-Photorealistic Rendering

Non-photorealistic rendering is creating images using a computer that intentionally avoid realism. Often, the intent is to simulate a particular artistic style such as impressionism, pointillism, or technical drawings. There isn't any requirement to draw simple lines when interpreting an L-system, and replacing a Line object with a Crayon or Brush object can result in some interesting visual effects.

The basic process for NPR is as follows.

1. Select a style or a description of a style you want to achieve
2. Develop a model of how the style is created
3. Convert the model into an algorithm for generating the style
4. Implement the algorithm as part of a drawing system

The models can be simple or complex. Watercolor, for example, requires extensive modeling in order to get reasonable results. The best work to date models the shape of the paper, the amount of paint and water on the brush, the action of water through the paper's capillaries, and the flow of pigment particles.

Some examples of NPR styles that have been implemented include:

- Impressionist oil painting
- Watercolor
- Van Gogh's fluid style
- Pen and ink (cross-hatching)
- Pointillism
- Mosaics
- Technical drawing

Crayon/Marker: how would we make a drawing look like Crayon or marker?

- Model: small lines with random widths
- Algorithm: given two end points, generate a sequence of lines that approximately connect them, using different widths and orientations.
- Implementation: modify the forward algorithm to create a Crayon object instead of a Line object.

Key concepts

- One approach is to divide the line into two pieces and draw the two pieces with small random offsets (perturbations). The ideal mid-point is defined as average of the two endpoints. Note that the point equations need to be calculated for both x and y.

$$P_{\text{mid}} = (P_1 + P_2)/2 \tag{9}$$

  Both the endpoints and the midpoint can be perturbed by a small random amount (add a small random value to x and y) in order to get two slightly imperfect lines instead of one perfect one.

For dividing a line into multiple segments, the parametric representation of a line is useful. Any point between P_1 and P_2 can be written as a function of a parameter $K \in [0, 1]$.

$$P = P_1 + K(P_2 - P_1) \tag{10}$$

Using a parametric equation, it is possible to generate several random numbers between 0 and 1 and use those numbers to subdivide the line in random places.

- For perturbing the end points, using a Gaussian distribution is reasonable, as the process of generating crayon/marker is likely close to a normal, or Gaussian distribution.
- If we use the AggregateBase parent class to build the Crayon class, then we get the benefit of all of the code that is already written, overriding only those methods that are necessary.

Pen and Ink: how could we make a tree drawing look like pen and ink with cross-hatching?

- Like the crayon, could use many lines for one
- Could make outlines with cross-hatching in the middle
- Could just slightly perturb the lines and use cross-hatching for the leaves

Brush: lots of line segments representing ink dipped by bristles

- Use many approximately parallel lines
- Perturb lengths slightly
- Use a combination of thicknesses
- Would be nice to blend similar colors

Subtleties of color...

8 Ghosts in the Machine

Now that you have some idea of what a computer can do and how to give it instructions, we're going to take a brief tour of what's inside.

At the level of the computer hardware, the computer repeats two actions.

- Fetch the next instruction
- Do what the instruction tells it to do

The concept is identical to the string parsing we do to convert an L-system character list into a set of actions. The only difference is that at the machine level there is digital logic interpreting the machine instructions and causing the flow of electrons from one place to another on the chip.

Most instructions are straightforward:

- Move some data from location A to location B
- Take data from locations A and B, manipulate it, then store the result
- Send or retrieve data from memory (short or long-term)

- Evaluate a condition and choose the next instruction to execute based on the result.

We call functions in code by using the function name and a list of arguments in parentheses. But what goes on when a function is executed?

Almost all computers have special instructions for calling a function and returning from a function.

- CALL - saves the return address and sets up the computer so execution begins at the function.
- RETURN - gets the return address and sets up the computer so execution begins at the line after the function was called.

These two instructions have to communicate. CALL has to store a value and RETURN has to retrieve it. What if there are other CALL-RETURN pairs in between? Functions can call other functions, after all, to an arbitrary depth.

The pattern of behavior that we need is identical to the brackets in L-systems: '[' and ']'. Just like the brackets, we need to store information and retrieve it later. Just like the brackets, other bracket pairs may be in between. Since we always want the computer to return to the most recent return address, a stack makes sense as a way to store the return address information.

It turns out that the computer has a system stack it uses for that purpose, among other things. The system stack stores return addresses as well as other temporary information associated with functions. For a simple function with no arguments and no return value, the only piece of information required is the return address.

- The calling program must store the return address and jump to the function (CALL).
- The function executes.
- The function calls RETURN to pop the return address and return to the calling program.

What about a function that takes arguments?

- The calling program must store the arguments (on the stack?)
- The calling program stores the return address on the stack and jumps to the function
- The function can find the arguments by looking on the stack
- The function executes
- The function calls RETURN to pop the return address and return to the calling program.

What about a function that also has a return value and local variables?

1. The calling program prepares space for the return value on the stack
2. The calling program prepares the arguments for the function on the stack
3. The calling program stores the return address on the stack and jumps to the function
4. The function creates space for any local variables it uses
5. The function executes
6. The function assigns the return value to the space on the stack
7. The function calls RETURN to pop the return address and return to the calling program.

8. The calling program has to clean up the arguments (top of the stack)
9. The calling program has to handle the return value (top of the stack)

We can think of the structure as a frame. Every function has a stack frame associated with it.

- Let the 0 point of the stack frame be the return address location for the current function
- Negative offsets point to the prior stack frame, arguments, and return value
- Positive offsets point to the local variables
- A calling program pushes its stack frame just before a CALL and pops it just after.

8.1 Recursion

What if a function calls itself? What happens?

- Same thing as if some other program called the function
- That call to the function gets its own set of parameters
- That call to the function gets its own local variables
- That call to the function gets its own return address back to wherever it was called

A function calling itself is called recursion. Recursion isn't always a useful thing to do.

- If a function has no parameters, calling itself will lead to an infinite loop (sort of)
- If a function has parameters, but the parameters don't change, the same thing happens
- If a function does not change its behavior depending upon the parameters, the same thing happens

When is recursion useful? Recursion is useful when a function has parameters **and** modifies its behavior depending upon the parameters.

- There has to be a set of parameter values for which the function does not call itself. This is called the terminal case.
- When the function calls itself, the parameters passed to the next instance of the function must change in a way such that the terminal case will be reached.

A simple recursive function, therefore, will have a structure that looks like:

```
function definition with at least one parameter in the argument list

    If the parameter(s) fit the terminal case
        return a (possibly empty) value

    Execute whatever code is necessary
    Call the function with parameters closer to the terminal case
```

Example: Drawing a sequence of trees of diminishing size.

One of the things we might want to do is draw a tree, then reduce its size, change locations, and draw it again. We can do this using recursion with a test on the size of the tree. Once it gets too small, we stop calling the function.

```
def drawsystem(lsystem, zturtle):

    # Terminal case: if the distance gets below 2, it returns without calling itself
    if lsystem.getDistance() < 1:
        return

    # Non-terminal case: move the turtle and decrease distance by 0.8
    zturtle.up()
    zturtle.setheading(0)
    zturtle.forward(lsystem.getDistance() * 10)
    zturtle.left(90)
    zturtle.forward(lsystem.getDistance() * 2)
    zturtle.down()

    # draw the tree
    zturtle.lsystem(lsystem)

    # decrease the size of the tree
    lsystem.setDistance( lsystem.getDistance() * 0.8 )

    # recursive call
    drawsystem( lsystem, zturtle )

    return
```

Conceptually, the above code forms a chain of function calls. Each time the function is called, the return address is pushed onto the stack along with the parameters and any local variables for that instance of the function. This allows each instance of the function to act as an independent call. The local variables and parameters are not shared between the different instances, just as local variables and parameters are not shared between different functions.

Recursion works well as an approach when you can describe the solution to a problem as the solution to a small piece of it plus the solution to the rest of it. So long as the rest of the problem is simpler than the original problem, the algorithm is making progress towards the terminal case.

Conceptually, the idea is similar to induction. Induction in mathematics is the idea that there is a base case for which the solution is well defined. An inductive rule connects one step in the process to the next. The base case plus the inductive rule defines a sequence.

Recursion goes the opposite direction. We start with a complex problem. The recursion rule explains the solution by dividing the problem into two parts: a partial solution and a recursive solution to the rest of the problem. The partial solution we can calculate immediately. Then we apply the algorithm to the rest of the problem. Since the problem gets smaller at each step, at some point we reach a problem that we know how to solve directly. Then we merge all of the results back together to generate the final solution.

Example: Binary search of a list.

Searching a sorted list of items is a common activity. There is a simple recursive algorithm that solves the problem efficiently. The basic idea is to look at the middle element of the list and compare it to the target. If the middle element is larger than the target, recursively search the lower half of the array. If the middle element is smaller than the target, search the top half. If the middle element happens to be the target, return success.

The algorithm terminates when either the target is found or there is no more data left to search. Since the data divides in half each time–and we're dealing with integer data–the maximum depth of the recursion is defined by the original number of elements in the list.

```
# returns True and the index of theValue if in theList, False and -1 otherwise
def binsearch( theList, theValue, startIndex=0, endIndex=None, depth=0 ):

    # if endIndex is None, search the whole list
    if endIndex == None:
        endIndex = len(theList) - 1

    # print out some information
    print 'binsearch at depth: ', depth,
            'searching from', startIndex, 'to', endIndex

    # terminal case: startIndex is greater than the endIndex
    if startIndex > endIndex:
        return (False, -1)

    # calculate the midpoint
    mid = (startIndex + endIndex) / 2

    # check the midpoint
    if theList[mid] == theValue:
        return (True, mid)

    # otherwise, check the bottom half or the upper half
    if theList[mid] > theValue:
        return binsearch( theList, theValue, startIndex, mid-1, depth+1 )
    else:
        return binsearch( theList, theValue, mid+1, endIndex, depth+1 )
```

The binsearch algorithm is a valid recursive solution because it has a base case (two, actually) and because it always reduces the scope of the problem.

In the above example we make use of a couple of Python features.

- When someone calls binsearch() with just the list as an argument, the algorithm begins searching the entire list.
- When the algorithm reaches a base case, we use a tuple to return success or failure as well as the index of the value matching the target.

Example: Towers of Hanoi

Perhaps the most commonly used example of recursion is the Towers of Hanoi problem. The setup is that you have three towers. One tower has a stack of rings on it from the largest on the bottom to the smallest on the top. The task is to move the rings to a different tower, moving the rings one at a time and never having a larger ring on top of a smaller ring.

The solution can easily be described using recursion for a stack of N rings, moving them from post 1 to post 3.

1. Move the top N-1 rings from post 1 to post 2, using post 3 as a reserve
2. Move the bottom ring from post 1 to post 3
3. Move the N-1 rings on post 2 to post 3, using post 1 as a reserve

The code is extremely simple.

```
# moves contents of post1 to post3 with post2 being reserve
def hanoi( N, post1, post2, post3 ):

    # terminal case, only one ring so take the action
    if N == 1:
        print 'Moved disk from', post1, 'to', post 3
        return

    # move N-1 rings from post 1 to post 2
    hanoi( N-1, post1, post3, post2 )

    # move biggest ring from post1 to post 3
    hanoi( 1, post1, post2, post3 )

    # move N-1 rings from post 2 to post 3
    hanoi( N-1, post2, post1, post3 )

# call the function
>>> hanoi(3, 'a', 'b', 'c')
Moved disk from a to c
Moved disk from a to b
Moved disk from c to b
Moved disk from a to c
Moved disk from b to a
Moved disk from b to c
Moved disk from a to c
```

The number of moves required is large as N gets bigger, but the solution is extremely easy.

Function References

Functions are sequences of instructions. In python, functions are actually just the python code itself (which is a sequence of instructions), and the interpreter dynamically figures out what to do each time the function is called. These instruction sequences are located in the same memory as everything else (variables, data, strings, etc.). The only difference between data and a function is how programs make use of them.

Functions have an address in memory, which is how they are called. In python (as in many languages), the name of the function acts as a label for the location that contains the address of the function. Therefore, the name of the function is a reference to the actual function, just as with objects a variable holds a reference to the actual object.

It is possible to write code that modifies the code in a function. It is even possible to write code that modifies itself. It's not a terribly safe thing to do, but it is possible (in fact, it's one of the tricks used to hack machines).

It is also possible to carry around function references and treat them just like object references. Any time a function name is used without parentheses at the end of it, the function name acts like an object reference. Therefore, we can copy the function reference to other variables, if we wish.

Any variable that holds a function reference can be used to call the function. The syntax to call a function is to place parentheses with any required arguments after the variable holding the function reference. In the example below, `functionA` is a regular function and `fa` is a variable that is assigned a copy of the reference to functionA. We can then use `fa` to call the function.

```
fa = functionA
fa( args )
```

Being able to manipulate function names is a great help when handling menus and figuring out what to do. It allows us to separate the code for handling the GUI from the specifics of which menu actions correspond to which functions.

Consider, as an example, a menu with five items: New, Open, Close, Save, and Quit. A typical GUI loop might look like:

```
Done = False
while not Done:

    # wait for a mouse click
    event = win.getMouse()

    selection = menu.select(event)
    if selection[0] == 0: # New
        doNew(win)
    elif selection[0] == 1: # Open
        doOpen(win)
    elif selection[0] == 2: # Close
        doClose(win)
    elif selection[0] == 3: # Save
        doSave(win)
    elif selection[0] == 4: # Quit
        doQuit(win)
        Done = True
```

Think about how the code and the items in the menu are linked.

- The menu item numbers are hard-coded into the event loop
- The functions that execute each option have to be in the correct case
- The ordering of the menu items cannot change without also changing the main loop code
- Adding new items in the menu requires re-ordering of the item numbers

This method of organizing the main loop means that changes in one part of the program–e.g. menu ordering or contents–have to be echoed in a different part of the program. That can lead to bugs, mistakes, and strange behavior if the code is not properly updated.

How can we structure the code so that the the main loop and the menu definitions are unlinked?

- Within the main loop we want a single process to occur no matter what the menu selection is.
- We need is a data structure that maps the return value of the menu selection to an action.
- Before the main loop we need to build the data structure.

What kind of data structure lets us map an arbitrary value to another value? How about a dictionary?

How can we store an action in a variable? How about a function reference?

Using a dictionary and function references, we can create a data structure that maps each menu item string to an action.

- Since the menu item strings are mapped to functions, ordering in the menus doesn't matter
- If the same string appears in multiple menus, it maps to the same function with no extra work
- If a string is deleted from a menu, that action will never occur.
- When a new menu item is added, the only change in the code is to add a new dictionary entry.

The structure of the code now has two parts: build the dictionary, then execute the loop. Inside the loop, the long if-elif statement of the prior version is replaced by a single test to see if the menu item is a key in the dictionary. If the key exists, the corresponding action is executed. Note that all of the action functions must return True or False, depending upon if the program is supposed to quit.

```
action = {}
action[ 'New' ] = doNew
action[ 'Open' ] = doOpen
action[ 'Close' ] = doClose
action[ 'Quit' ] = doQuit

Done = False
while not Done:

    # wait for a mouse click
    event = win.getMouse()

    selection = menu.select(event)
    if action.has_key( selection[1] )
        Done = action[ selection[1] ]( win )
```

8.2 Parameters, Global Variables, Classes, and Dynamic Object Fields

Note that in the above example, the variable containing the window reference is passed to every action function. Since we can't adjust the argument list for each action function, we have to somehow give each function access to all of the information it needs.

Consider the functions above:

- doNew should create a new blank window and needs to store it somewhere.
- doOpen should read in a file, store the shapes, and draw the shapes into a window.
- doSave should access a list of all the shapes so far and write them to a file.
- doQuit should close the window and return True

doNew needs access to a field that is intended to hold a window reference, or list of window references. doOpen and doSave need access to the window, but also to a shapes list that contains all of the shapes that have been created.

There are a three general options for how we might get information into a function.

- Parameters: pass in a variable or variables that hold the required information. The variables are copied onto the stack when a function is called so the function has access. If an object reference is placed on the stack, then the function can access the object's fields and modify them. A useful way to make use of parameters is to create a class specifically to hold application-wide information. Create one instance of the class at the beginning of your program and pass it to each action function.

  As parameters make use of the stack to get passed into a function, they have limited scope; they are only accessible within the function. Information transfer to functions is explicit and made through argument lists for functions.

- Global Variables: make the generally useful data globally accessible. Outside of any function you can create variables by assigning a value to the symbol.

  ```
  gblShapeList = []
  ```

  Within an action function is you have the line:

  ```
  global gblShapeList
  ```

  then when you use the variable `gblShapeList` it will refer to the one declared outside the function.

  Global variables can be implemented if we have some space in memory that is not part of the stack, in which we can store data. The global statement inside the program tells the Python interpreter to go find the top level variable by the given name. Python has to keep track of all active symbols in a table, and the `global` keyword lets it disambiguate which variable is intended by the programmer in the case of name conflicts.

  Global variables, while convenient, do not lend themselves to good software design. As global variables do not need to be passed into a function, someone reading a program is unaware that the function may be accessing or modifying data in the global variable. Passing in the same information as a parameter makes it clear what information the function requires.

- Application Class: build an Application class that contains all of the fields required by all of the action functions. Each action function can then be built as a method of the Application class, giving them access to all of the required information.

  The constructor for an Application class generally sets up and initializes the application, creating a window, the menus, and the action table, for example. The action methods manipulate the Application data/fields as required. Because all of the action methods have access to the self object reference, they can get access to everything they need.

  An Application method may also have a main loop function which handles all events for the program. The main loop function only terminates when the user has told the application to quit.

  Under this style of organizing, the main program looks very simple and exists at a high level of abstraction.

```
def main():
    myapp = Application( 'MyDraw', 800, 600 )
    myapp.mainLoop()
    myapp.terminate()
```

There are also hybrid approaches that mix the above design strategies. For example, one could build an Application class that does not have action methods, but which holds all of the required fields and information. The Application class then gets passed around to all of the action functions to ensure they have the information they need. Conceptually, this is no different than making the action methods part of an Application class and having access to the self variable.

One final approach that is unique to Python is to make use of the ability to create dynamic fields in objects. For example, many actions in the draw program require access to the window reference and to a list of shapes. The window is a GraphWin object, whose standard fields are defined by its constructor. But we can add fields to an object dynamically.

For example, we might want the application to keep track of the following information:

- The list of shapes that have been drawn
- The current color selection
- The current style selection
- The help text object (so action methods can modify it)

It's easy to create new fields in the GraphWin object to old these items.

```
win.shapes = []
win.style = 'default'
win.color = [0.0, 0.0, 0.0]
win.helpText = Text( winwidth/2, winheight-20 )
```

By assigning values to the fields, the fields are created and assigned the specified value. In effect, we are co-opting the window class and using it as a vehicle for passing information to the action functions. It plays the same role as a specifically designed Application class.

Co-opting a class seems like it breaks the concept of modularity and encapsulation. But it does not do so directly. All of the data and functionality of the original class is still in place, and dynamically adding fields does not change that. We are only adding fields to the class and making use of them. The problem that arises

is subtle in the sense that we can break encapsulation if we accidently add a field that already exists in the object. If we duplicate an original field name, it will have unintended consequences.

In the end, the organizational scheme that probably makes the most sense is to create an Application class that is either purely informational (collects all the relevant information into a single object), or comprehensive (collects both information and methods into a single class). It best preserves encapsulation, modularity, and provides a nice level of abstraction.

9 Debugging

Debugging is a critical skill in any design process. There are two key pieces of debugging:

1. You need to know what your system should be doing.
2. You need to know what your system is actually doing.

The difference between the two is what is wrong.

The best way to debug a program is to have the program tell you what it's doing (#2). The `print` statement is your friend. You can print out the contents of any variable in Python, and you can print out strings at useful places in the program. Watching your program run by following print statements lets you know when it does something unexpected. Printing out the contents of variables tells you why it is taking an action.

With practice, you will tend to have fewer bugs of the syntactic variety and many more of the logic variety. Syntax bugs are caught by the Python interpreter (or compiler in other languages). Logic bugs, on the other hand, must be caught by you.

10 Summary and Review

Highlights of the semester

- What can a computer do?
- Programming as abstraction: making things general / using parameters
- The main structures of algorithms: assignments, conditionals, and loops
- Organizational structures of languages: functions, classes
- Data structures: strings, tuples, lists, stacks, hash tables (dictionaries)
- Input/output: reading and writing files
- Grammars: describing a set of valid strings using rules
- Interpretation: decoding symbols into actions
- Basis for design: encapsulation, modularity, inheritance, polymorphism
- Memory models: objects, object references, variables, system stack
- Recursion: solving big problems by solving simpler problems
- Graphics: turtle and otherwise, thinking visually
- Graphical User Interfaces: menus, clicks, events
- Debugging
- A programming language: Python

Sometimes it's difficult to see the forest for the trees (or bugs). This is why it's important to have a clear understanding of the problem and the steps required by the solution.

Thinking about a problem computationally means trying to describe a series of steps that a computer could execute to solve it. The actual implementation of the required steps is just a matter of syntax and good design decisions.

There is no substitute for practicing design and coding, but they are not the most critical skill to possess. The most critical skill is the ability to formulate the steps required to solve the problem.

10.1 Examples

While Python is a useful language to learn, there are many others. If you have an understanding of the main structures of computational algorithms, it is not difficult to take what you've learned and apply it to other languages.

Python

```
import random

N = 100
data = []
for i in range(N):
    data.append(random.random() * 10.0)

sum = 0.0
count = 0.0
for value in data:
    sum += value
    count += 1

mean = sum/count

print 'Mean is %.2f' % mean
```

C: Note the explicit variable typing and the different for loop style.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

int main(int argc, char argv[]) {

  int N = 100;
  float *data = malloc(sizeof(float) * N);
  float sum, count, mean;
  int i;

  srand48(time(NULL));
  for(i=0;i<N;i++) {
    data[i] = drand48() * 10.0;
  }

  sum = 0.0;
  count = 0.0;
  for(i=0;i<N;i++) {
    sum += data[i];
    count++;
  }

  mean = sum / count;

  printf("Mean is %.2f\n", mean);

  return(0);
}
```

C++: Not much difference from C, slight changes in memory and I/O calls.

```
#include <sys/time.h>
#include <iostream>
#include <cmath>

int main(int argc, char argv[]) {

  int N = 100;
  float *data = new float[100];

  srand48(time(NULL));
  for(int i=0;i<N;i++) {
    data[i] = drand48() * 10.0;
  }

  float sum = 0.0;
  float count = 0.0;
  for(int i=0;i<N;i++) {
    sum += data[i];
    count++;
  }

  float mean = sum / count;

  std::cout << "Mean is " << mean << std::endl;

  return(0);
}
```

PHP: More similar to python, but a mixture of C and python concepts.

```
<html>
<body>
<?
$N = 100;
$data = array();

for($i=0;$i<$N;$i++) {
  $data[$i] = (float)rand() / (float)getrandmax() * 10.0;
}

$sum = 0.0;
$count = 0.0;
for($i=0;$i<$N;$i++) {
  $sum += $data[$i];
  $count++;
}

$mean = $sum / $count;

echo 'Mean is '.$mean;
?>
</body>
```

Java: All programs in Java are class methods. A class must have a main method in order to be an executable program. Note explicit variable typing, syntax and for loops by C. Memory management is very different, however, as the programmer does not need to free memory (like Python).

```
import java.util.*;

public class mean100 {

    static public void main(String[] argv) {

double mean;
double sum;
double count;
double[] data;
int N = 100;
Random gen;

gen = new Random();
data = new double[N];

for(int i=0;i<N;i++) {
    data[i] = gen.nextDouble() * 10.0;
}

sum = 0.0;
count = 0.0;
for(int i=0;i<N;i++) {
    sum += data[i];
    count++;
}

mean = sum / count;

System.out.println( "Mean is "+mean );
    }

};
```

11 Sorting: The Outdoor Lecture

Sorting data is a useful thing to be able to do.

Bubble sort:

The idea with bubble sort is to repeatedly go through the list and compare neighbors. If the neighbors should be swapped, do so. When you can loop through the list and nothing changes, the sort is complete.

```
def bubbleSort( alist ):
# define a bubble sort function that takes in a list
def bubbleSort( alist ):

    changed = True

    # loop while nothing has changed
    while changed:
        print alist

        # set changed to false
        changed = False

        # loop over the contents of the list
        for i in range( len(alist) - 1 ):

            # check if the neighbors need to be swapped
            if alist[i] > alist[i+1]:

                # python swap thingy
                alist[i], alist[i+1] = alist[i+1], alist[i]

                # we changed something
                changed = True
```

How many times might bubble sort execute the outer loop?

- Each time through the loop, one number is guaranteed to get put into its proper place (the largest out of place number.
- Each time through the loop all N elements are tested
- An out of order list will need N iterations to sort the list

The number of steps in the algorithm, in the worst case, is proportional to $N \times N$. We can say that the complexity of the algorithm is approximately N^2.

Quicksort:

Quicksort is based on the idea of executing a simpler task and then subdividing the problem. The simpler task is putting all the big things in the upper half of the array and all the small things in the lower half of the array. Then the procedure is repeated for each half of the array. At some point the individual sub-arrays will consist of one element, which means they are sorted.

The procedure is:

- If the list is length one, return
- Pick a random rotation element (last one in the list, for example)
- Set FirstAfterSmall to the location of the first element
- Go through the list
  - If an element is smaller than the pivot
    - Swap the element with the element indexed by FirstAfterSmall
    - Increment FirstAfterSmall
- Swap the rotation element with FirstAfterSmall
- Recursively process the part of the list up to FirstAfterSmall
- Recursively process the part of the list after FirstAfterSmall

The following is an example of the algorithm in action:

Step	Current State	FirstAfterSmall
original	4 3 6 2 7 5	0
0	*4* 3 6 2 7 **5**	1
1	*4 3* 6 2 7 **5**	2
2	*4 3* 6 2 7 **5**	2
3	*4 3 2* 6 7 **5**	3
4	*4 3 2* 6 7 **5**	3
final	*4 3 2* 5 6 7	3
0	[4 3 **2**] 5 [*6* **7**]	0 / 0
1	[4 3 **2**] 5 [*6* **7**]	0 / 1
2	[4 3 **2**] 5 [*6* **7**]	0 / 1
final	[2 4 3] 5 [*6* **7**]	0 / 1
0	2 [4 **3**] 5 [6] 7	0 / 0
final	2 [3 4l] 5 [6] 7	0 / 0
final	2 3 4 5 6 7	

It turns out that the complexity of quicksort is dependent upon picking good rotation elements. If you always pick the largest remaining element as the basis for splitting the list, then only one element in the list gets sorted each iteration and the complexity is about N^2.

On the other hand, if you can pick good rotation elements and split the list about in half each time, then the complexity approaches $N \log(N)$, which is very good. It turns out that, in practice, it's not hard to pick reasonable rotation elements, so quicksort is a commonly used sorting algorithm.