CS 151 Computational Thinking, Fall 2011

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

The course is an introduction to computational thinking: how we can describe and solve problems using a computer. Using the Python language, students will learn how to write algorithms, manipulate information, and design programs to make computers useful tools. Through lectures, short homeworks, and weekly programming projects, students will learn about abstraction, how to divide and organize a process into appropriate components, how to describe processes in a computer language, and how to analyze and understand the behavior of their programs. Students will communicate the results of their work through project writeups.

Prerequisites: None

> We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology.
>
> Carl Sagan

Desired Course Outcomes

A. Students can read a simple program and correctly identify its behavior

B. Students can convert a problem statement into a working program that solves the problem.

C. Students understand abstraction and can break down a program into appropriate procedural and object-oriented components

D. Students can generate an approximate model of computer memory and describe how an algorithm affects its contents.

E. Students can communicate the result of their work and describe an algorithm.

This material is copyrighted by the author. Individuals are free to use this material for their own educational, non-commercial purposes. Distribution or copying of this material for commercial or for-profit purposes without the prior written consent of the copyright owner is a violation of copyright and subject to fines and penalties.

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 theories 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 good 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.

Another way of thinking about the level of abstraction is thinking about it as how to define the instruction set, which is a combination of vocabulary and the semantic meaning attached to each vocabulary word.

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.

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. Note that you need instructions to describe spatial relationships.
- Level 3: Describe the face as a series of line segments or arcs in particular locations, lengths and orientations.
- Level 4: Describe the face as a set of points that are drawn (stippling). Each point is defined by an (x, y) position.

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. A close physical example is a rubber stamp with a face etched on it.
- Level 2: Reasonably concise representation, enables variation in the faces, the instruction set enables drawing of a variety of non-face things. A close physical example is a traditional typewriter with a set of stamps.
- Level 3: More lengthy set of instructions to draw a single face, but the instructions are generic to most kinds of drawing. A close physical example is an x-y plotter.
- Level 4: very long and detailed set of instructions, completely generic, painful to describe a single picture, but easy to automate. A close physical example is a dot-matrix printer.

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 needs to be transferred to the target only once. The instructions themselves must be transferred each time the program is supposed to execute.

- Level 1: All of the information is in the definition. The instruction could be a single bit of information, or at most position and orientation (x, y, θ).
- 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. Each instruction has a similar representation.
- Level 4: Almost all of the information is in the instructions. The definition requires little information: put a dot here.

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).

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. The core of a computer is its central processing unit [CPU], which controls how data is moved, stored, and manipulated. However, the CPU is not particularly useful without the ability to input data, output data, and control what it is doing. The standard components of a computer including the following.

- 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.

When we start a consumer application, like a mail or word processing program, the CPU reads the instructions from the hard drive and stores them in the RAM, which is fast memory close to the CPU. It then begins to execute the instructions in the program. A mail program will send messages over the network to other computers. A word processing program will get documents from the hard drive, display the contents on a monitor and enable the user to manipulate the contents in memory before writing the document back to the hard drive.

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 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 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 slower than compiled languages.

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. It also means that the code will generally run faster than the same operations in Python; once the program is in machine code it doesn't require any more interpretation to be run on the computer since it is using the language of the computer hardware.

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

```
$ python
```

When the interpreter starts up, you can type in Python code. To exit the Python interpreter, type control-D (hold down the control key and then type the D key).

Keep in mind that you can always try out python code in the interpreter. It is useful for quick experimentation, or if you have a question about syntax or the meaning of an expression. If you type a line of python code in the interpreter, when you hit return it will immediately execute the code and print its value, if any.

2.1 Variables

One of the basic operations of a computer is that it can store data. Physically, data is stored in memory. Every memory location has an address, which is an actual binary number. Rather than specify a memory location using an address, however, programming languages let us describe memory locations using symbols. We can even abstract away from the physical memory and think about a virtual memory space where each variable represents some arbitrary piece of information. How and where a piece of information is actually stored in memory is not something we necessarily need (or want) to know when writing algorithms.

Variables let us describe operations on a particular piece of information without us needing know what the information is beforehand. In some cases, the operations we want to perform require the information to be of a certain type.

Note that we use variables all the time when we are talking about the world and about manipulating the world. For example, pick up an object in your left hand. Now transfer the object to your right hand. Now set the object back down where it was. Note that these instructions did not specify what you picked up. The description of the process used a variable, in this case called "object", that represented the item you picked up. The particular characteristics or **type** of the thing could vary significantly. However, we can probably put at least one constraint on the type of the thing: you could pick it up and hold it in one hand.

What are variables in Python?

- A variable holds a piece of information, or data (a thing)
- Any variable in Python can hold any kind of data
- Once you assign data to a variable, the variable has a 'type' to it that specifies how to interpret the information (its semantic interpretation)
- The type describes the data held in the variable, not the variable itself
- You can do different things with different types

What are the **rules** for the names of variables in Python?

- The name has to start with a letter or an underscore character _
- A name cannot start with a number.
- Names can have letters or numbers in them but no punctuation marks except _
- Names can be arbitrarily long
- Capitalization matters, so `thing` is different than `Thing`
- A variable name cannot be a keyword like `if` or `for`
- 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

2.2 Assignment

Assignment is the action of moving data from one place to another. When assigning a value to a variable, the value of the expression on the right side of the operator is moved into the variable specified by the expression on the left side of the operator. Therefore, information always flows from right to left in an assignment. It is important to think properly about assignment expressions. The expression

```
x = 6
```

should not be described as "x equals 6". Instead, you should think about it as "x gets 6" or "x is assigned the value 6". A single equals sign is not an equality statement in Python (or most computer languages); it describes the movement of data from one place to another. For example, the expression

```
x = x + 1
```

makes no sense if we say "x equals x plus 1". But it does make sense if we think of it as "assign to x the sum of the current value of x and the integer 1".

Just as there are rules for variable names, there are also **conventions** for variable names. Programmers use conventions to encourage everyone to write code 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.

Variable names are easier to remember than numbers. However, they are still a significant cause of mistakes, or **bugs**, in programs. There are two common types of bugs that occur with variables.

- Using a variable name before it is defined. In Python, you must assign a value to a variable before you can use it on the right side of an assignment.
- Typos, especially improper capitalization or misspellings, are one of the most common errors in programming. If you type the name of a variable incorrectly, it will often look like you are trying to use a variable without previously defining it.

Example: Creating Variables in Python

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 evaluate it as an expression. The latter means just type the name of the variable in the interpreter and hit return. Typing any Python expression into the interpreter asks Python to tell you the value of the 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
```

The problem is that `abigvariablename` is misspelled in the first line, while it is correctly spelled in the third line.

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'>
```

2.2.1 Casting

Sometimes it is important to convert the contents of a variable into a particular type. For example, you may want to convert an integer to a string in order to print it out, or covert a string to an integer in order to do some math. Likewise, you may want to be explicit about doing floating point math versus integer math, regardless of the particular data type of a variable.

The process of converting a data from one type to another is called **casting**. To cast a variable into particular data type, put the name of the data type and then put the variable to cast inside parentheses. The basic data types are int, float, and str, which stand for integer, floating point, and character sequences. Python has many other, more complex data types as well, which we will explore later.

The following example shows conversions from integers to strings and strings to integers.

```
>>> x = "10"

>>> y = 5

>>> print int(x) + y
15

>>> print x + " and " + str(y)
10 and 15
```

2.2.2 Symbol Tables

In order to execute code, the Python interpreter has to keep track of what variables exist, what values they hold, and what their types are. As a programmer, you also need to keep track of what variables you are using, what information they hold, and the type of data they hold. Therefore, you need to have a model of the computer in your head so that you can read code and predict or explain what it is going to do. If you use an incorrect model as the basis for designing and writing code, then the code will not function the way you expect.

A **symbol table** is a useful model of the way Python keeps track of variables and their relevant information. You can use symbol tables to make predications about what a program will do when executed by the Python interpreter and to understand its behavior.

A symbol table is a chunk of memory on the computer. You can think of it as a table with rows and columns. Each row corresponds to a variable. The first column is the variable's symbol, or name, and the second column is the variable's value. Other columns in the table may correspond to information about the variable's type and other information required by the interpreter. For our purposes, it is sufficient to think of the table has containing each variable's name, value, and type.

When you start up the interpreter, Python initializes a global symbol table. Python has many internal variables it uses to keep track of the state of the interpreter, but for our purposes we can think of the initial global symbol table as empty. Now consider the following four commands.

```
>>> a = 5

>>> b = 1.0

>>> c = "hello"

>>> a = b

>>> b = b + 1.0
```

When interpreting first assignment, Python first evaluates the right side of the assignment, which evaluates to the integer value 5. It then examines the current symbol table to see if there is an entry for the symbol `a`. Since there is no entry with that name, it adds a line to the symbol table and sets up the name `a`, the value `5`, and the type `integer`. The symbol table is shown below.

Name	Value	Type
a	5	integer

When interpreting the second assignment, Python evaluates the right side of the assignment, which is the floating point value 1.0. It then examines the current symbol table to see if there is an entry for the symbol `b`. Since there is no entry with that name, it adds a line to the symbol table and sets up the name `b`, the value `1.0`, and the type `float`.

Name	Value	Type
a	5	integer
b	1.0	float

Likewise, for the third line, Python ends up adding a third variable to the symbol table with the name `c`, the value `‘‘Hello’’`, and the type `string`.

Name	Value	Type
a	5	integer
b	1.0	float
c	“hello”	str

When interpreting the fourth line, Python first evaluates the right side of the expression. Since the right side has the symbol `b`, Python looks in the current symbol table to discover if there is a symbol with that name. Upon finding the symbol, Python looks at its value 1.0, which becomes the value of the right side of the assignment. Python then looks to see if there is a symbol `a` in the symbol table. Upon finding the symbol `a`, Python then copies the value of the right side of the expression into the variable `a`. Now the variables `a` and `b` have the same value and type.

Name	Value	Type
a	1.0	float
b	1.0	float
c	"hello"	str

To interpret the fifth line, Python again evaluates the right side of the expression by looking up the current value of `b` and adding to it the value 1.0. The right side of the expression, therefore, has the value 2.0 with the type `float`. It then uses the symbol table to look up the variable `b` again and assigns the value of the right side to the value field of the variable `b`. Note that variable `a` still has the value 1.0. Using the symbol table it is easy to understand why the values of `a` and `b` are different: they refer to different lines of the table and different memory locations.

Name	Value	Type
a	1.0	float
b	2.0	float
c	"hello"	str

Symbol tables are a useful model for how Python stores and manipulates memory. As we introduce new language constructs we will expand how we use symbol tables to form appropriate models of the computer. As noted above, it is essential to have a model for the computer's behavior that enables you to predict the results of a program. Without a correct model, you cannot design code that will work as you intend.

2.3 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 (modulo)	`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. Adding an integer and a string will produce an error because it's not clear what the meaning of the expression should be.

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 as appropriate, even if you don't need them, to clarify the order of operations and make the expression more readable.

The following are a number of examples showing how parentheses can both change the order of operation and make the expression more understandable.

```
>>> (5 * 4) + 3
23
>>> 5 * 4 + 3
23
>>> 5 * (4 + 3)
35
>>> (5 * 4) / 3
6
>>> 5 * 4 / 3
6
>>> 5 * (4 / 3)
5
>>> (5 - 4) + 3
4
>>> 5 - 4 + 3
4
>>> 5 - (4 + 3)
-2
```

2.4 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 reduce errors by encapsulating code into reusable parts.
- Functions reduce errors by permitting us to test parts of the code independently.

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, function definitions 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, blank lines and comments within the function are allowed (in some cases encouraged). 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. Each parameter initially holds the value assigned to it when the function was called. You can change the value contained in a parameter variable at any time within the function. However, common practice is to avoid modifying parameter variables and to instead create local variables for values that need to change. As with all variables, parameters can hold any data type. Use informative parameter names to make reading and writing the code easier.

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 exists only inside the function. The reason is that each function has its own symbol table, which gets deleted when the function exits.

```
>>> 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 the task 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 than 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 too 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 the code for each step as a function.

2.4.1 Function Symbol Tables

In order to predict the behavior of functions in Python, we need a model to represent the state of the computer before, during, and after the function has executed. Just as Python keeps track of variables and other symbols, like functions, in a symbol table, so it keeps track of the variables within a function using a symbol table.

When we define a function, it creates a new entry in the top level symbol table, assuming the symbol does not already exist. If the symbol already exists, then the new function definition overwrites the old value of the symbol. As Python reads through a function definition, it also initializes the symbol table it will need inside the function and sets it aside. The value and type of variables defined in the function are initially undefined.

We we execute a function, Python initializes that function's symbol table. In particular, it initializes the entries for each parameter in the function definition, specifying their values and types upon entering the function. To calculate the initial values, Python looks at the expression for each argument of the function call. Each expression must evaluate to a value of a particular type. That value is then copied into the function's symbol table for the appropriate parameter entry.

Note that the value passed into a function is generally called an **argument**. The symbol that holds the value of the argument inside the function is called a **parameter**.

Inside the function, Python places values for local variables into the function's symbol table. A local variable is any variable used on the left side of an assignment operator within the function. When trying to find the value of a variable, Python always searches the function's symbol table first, then searches the global symbol table if no local variable with the same symbol exists. Note that if the code creates a local variable at any point in the function (uses it on the left side of an assignment), then Python will not look in the global symbol table for that variable because the variable's identifier will exist in the function's symbol table.

The order in which Python searches symbol tables enables situations where local variables hide global variables because they share the same symbol.

In the following example we can follow the execution process through the global and function symbol tables.

```
def euclid(x1, y1, x2, y2):
    dx = x1 - x2
    dy = y1 - y2
    dist = (dx * dx + dy*dy)**0.5
    return dist

a = 0.0
b = 0.0
c = 2.0
d = 2.0
dist = euclid( a, b, c, d )
print dist
```

Global symbol table before the call to the euclid function:

Name	Value	Type
a	0.0	float
b	0.0	float
c	2.0	float
d	2.0	float
euclid	function data	function

Before the call to euclid, the global symbol table contains entries for the four variables a, b, c, d and the function euclid.

Function symbol table at the beginning of euclid:

Name	Value	Type
x1	0.0	float
y1	0.0	float
x2	2.0	float
y2	2.0	float
dx	undef	undef
dy	undef	undef
dist	undef	undef

At the beginning of the euclid function, Python copies the values from a, b, c, d to the parameters x1, y1, x2, y2. The local variables dx, dy, and dist are initially undefined.

Function symbol table at the end of euclid:

Name	Value	Type
x1	0.0	float
y1	0.0	float
x2	2.0	float
y2	2.0	float
dx	-2.0	float
dy	-2.0	float
dist	2.828	float

At the end of the euclid function, all of the variables have values and types. The return statement becomes the value of the function call if the function is on the right side of an assignment. Note that the local variable dist and the global variable dist are in different tables.

Global symbol table on last line of the top level

Name	Value	Type
a	0.0	float
b	0.0	float
c	2.0	float
d	2.0	float
dist	2.828	float

The assignment statement copies the return value of the function to a new entry in the global symbol table called dist. Python clears the function's symbol table when the function exits. Only the global symbol table variables exist at the top level.

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.

2.4.2 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.

2.5 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 tested 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?

2.6 Sequences, Lists, and Arrays

When working with data, or information, we commonly encounter sequences of information. These might be measurements of temperature taken once a day for a year. These might be sequences of letters, which we generally call strings.

Any time we have a sequence of information, we probably want to do some kind of analysis, likely the same kind of analysis to each element of the sequence. For example, we may want to print the string, which we can think of as doing the same operation to each character. We may want to sum all of the temperatures taken over a year in order to calculate an average value. Executing operations on sequences of information is so common that every programming language has features to support it.

Visualizing a task as occurring on sequences of information is a useful aspect of computational thinking. Any time you can describe a task as executing the same operation on each element of a sequence of data it becomes easier to encode and automate.

2.6.1 Lists

Lists are a fundamental data type in Python. Conceptually, they are a sequence of pieces of information. The sequence starts at location 0 and continues for as many pieces of information as are in the list. If there are N items in a list, the last item has the index N-1. For example, if there are ten items in a list, the first index is 0 and the last index is 9.

The syntax for writing out a list is a comma separated sequence of items, surrounded by brackets.

```
>>> listOfNumbers = [1, 2, 3, 4, 5]
>>> listOfStrings = ['ab', 'cd', 'de']
>>> listOfMixture = [1, 'ab', 2, 'bc', 3.0]
>>> print listOfNumbers[0]
1
>>>print listOfStrings[1]
cd
>>>print listOfMixture[4]
3.0
```

To access any element of a list, we use an index notation. If you add a number, in square brackets, to the end of a symbol that holds a list, Python will access the specified element of the list. Python lists are all zero-indexed. Array indexes have the following 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.

All the elements of a list do not have to be the same type. Each location in a list can hold a different kind of information, as shown above. You can modify any element of a list by using the bracket index notation on the left hand side of an assignment. Lists are **mutable**, which means you can change what any location in a list references.

```
>>> example = [1, 2, 3, 4, 5]
>>> print example
[1, 2, 3, 4, 5]
>>> example[2] = 10
>>> print example
[1, 2, 10, 4, 5]
>>> print example
[1, 2, 10, 4, 0]
```

You can add items to the end of a list by using the append method of a list. A method is a function that is attached to data type. Another way of thinking about it is that it is a function that modifies the variable to which it is attached. The following example shows how to add several items to a list.

```
>>> grow = []
>>> print grow
[]
>>> grow.append( 10 )
>>> print grow
[10]
>>> grow.append( 20 )
>>> print grow
[10, 20]
>>> grow.append( 30 )
>>> print grow
[10, 20, 30]
>>> grow.append( 'hut, hut, hut' )
>>> print grow
[10, 20, 30, 'hut, hut, hut']
```

The symbol table representation of a list is important to understand in order to use them properly. A list is an **object**, which means the data for the list is not stored in the symbol table along with its name and type. Instead, the symbol table entry holds a reference to the list object. The same model holds for all mutable objects. You can model the list object as a symbol table itself. That symbol table contains information about the object, such as its length and the information it contains.

Consider the following four commands.

```
>>> squares = [1, 4, 9, 16]
>>> squares.append( 25 )
>>> squares.append( 36 )
>>> print squares
[1, 4, 9, 16, 25, 36]
```

The following are the global and list symbol tables after each operation.

Global symbol table after the first assignment:

Symbol	Type	Value
squares	list	ref list1

squares symbol table after first assignment:

Symbol	Type	Value
length	int	4
data	ref	ref data [1, 4, 9, 16]

Global symbol table after the first append:

Symbol	Type	Value
squares	list	ref list1

squares symbol table after 2nd assignment:

Symbol	Type	Value
length	int	5
data	ref	ref data [1, 4, 9, 16, 25]

Global symbol table after the second append:

Symbol	Type	Value
squares	list	ref list1

squares symbol table after 3rd assignment:

Symbol	Type	Value
length	int	6
data	ref	ref data [1, 4, 9, 16, 25, 36]

The symbol table model also tells us how we expect Python to behave if we try to assign one list to another. The following is a simple example that the symbol table model properly explains.

```
>>> accessorA = [ 'a', 'b', 'c' ]
>>> accessorB = accessorA
>>> print accessorA
['a', 'b', 'c']
>>> accessorB[1] = 'oops'
>>> print accessorA
['a', 'oops', 'c']
```

The first statement creates an entry in the global symbol table for accessorA. It also creates a list object and puts a reference to that object in the entry for accessorA. The second statement creates a new entry in the global symbol table for accessorB and copies the information in accessorA's entry into accessorB's entry. It is important to understand that the information being copies is the reference to the list object, not the list object itself. Therefore, when we use accessorB to index into the list object on the left side of an assignment, it is the same list object referenced by accessorA. This is a condition called **aliasing**. It means that two symbols refer to the same object in memory, and therefore, using one symbol to edit the object causes the effects to appear when referencing the same object using another symbol.

The most important thing to remember is that when assigning one list to another, **it does not make a new copy of the data**. One method of making a copy is to write a loop that goes through the list and makes a copy of each element, placing it into a new list. The expression `a[:]` is a copy of the top level of `a`.

One of the useful features of Python is that we can always discover the length of a list, or any sequence, by using the `len` function on any variable that contains a sequence of information.

```
>>> values = [3, 2, 1]
>>> print len( values )
3
>>> values.append( 0 )
>>> print len(values)
4
```

In addition to numbers and strings, a list can also hold other lists, as in the example below.

```
>>> coords = []
>>> coords.append( [0, 0] )
>>> coords.append( [50, 100] )
>>> print coords
[[0, 0], [50, 100]]
```

The first element in `coords` is the list [0, 0], and the second element in `coords` is the list [50, 100]. Imagine using a list of lists to hold the vertices of an arbitrary polygon. For example:

```
>>> coords = [ [0, 0], [100, 0], [75, 100], [50, 25] ]
```

How could we then use the `coords` list to draw the polygon?

```
for i in range( len( coords ) ):
    pt = coords[i]
    turtle.goto( pt[0], pt[1] )

turtle.goto( coords[0][0], coords[0][1] )
```

The last statement, which closes the polygon, uses a double index on the `coords` list. The first index specifies which position in the `coords` list to access, and the second index specifies which position in the inner list to index. Multi-dimensional lists are a useful way to organize information, especially when each item in a list needs many pieces of information to describe it (e.g. position, color, line type, fill, etc.).

Lists also provide more sophisticated methods of access. In particular, it is possible to specify any subset of a list using a range notation where the start index and end index are separated by a colon. The following are a few examples. Note that if one side of the colon has no number, it means either 'from the start' or 'to the end' of the list.

```
>>> a = [1, 2, 3, 4]
>>> print a[0:2]
[1, 2]
>>> print a[1:]
[2, 3, 4]
>>> print a[:-1]
[1, 2, 3]
>>> print a[3:4]
[4]
```

2.6.2 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.

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. Given the need to support international character sets, a new standard called Unicode uses two bytes for each character, permitting 65,536 different characters.

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 location 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? We can use the same index notation we use for lists or other sequences.

```
>>> 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. As with lists, we can always discover the number of characters in a string using the `len()` function.

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

Unlike some languages, strings in Python 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. What that means is that you cannot change a specific character in an existing string. The following statement produces an error.

```
>>> a = 'a gypo'
>>> a[2] = 't'
TypeError: 'str' object does not support item assignment
```

The ability to index into a sequence of information is tied closely with our ability to iterate, such as using for loops. If we can loop a certain number of times, then we can go through each element of a string, or sequence. If we can look at each element of a string, then we can execute an action on each character.

In the following example, the code prints out each character in a string separately. The print statement is executing the same process on each input, and the inputs are dependent upon the index variable.

```
astring = 'abcd'
for i in range( len(astring) ):
    print astring[i]
```

It is also possible to do more complex operations based upon the values of the string. In the following code, we have three drawing functions for creating a circle, a triangle, and a square at arbitrary locations. Using an if-statement inside the for loop, the particular character in the string determines what shape to draw.

Example: use a string to select which of a set of shapes to draw

```
import turtle

def triangle(x, y, size):
    goto(x, y)
    for i in range(3):
        turtle.forward( size )
        turtle.left( 120 )

def square(x, y, size):
    goto(x, y)
    for i in range(4):
        turtle.forward(size)
        turtle.left( 90 )

def circle(x, y, size):
    goto(x, y)
    turtle.circle( size )

shapestr = 'ttccss'

for i in range( len(shapestr) ):

    x = random.randint(-300, 300)
    y = random.randint(-300, 300)
    size = random.randint( 10, 100)

    if shapestr[i] == 's':
        square( x, y, size)
    elsif shapestr[i] == 't':
        triangle( x, y, size)
    else:
        circle( x, y, size)
```

2.6.3 Tuples

A **tuple** is a third method in Python for storing sequences of information. Like lists, a tuple can hold a variety of types of information, including lists and and other tuples. Once created, however, the tuple is itself **immutable**. That means that the order of information in the tuple and any immutable values within the tuple are fixed. Only immutable objects embedded in the tuple, like lists, are valid on the left side of an assignment.

You can read from the elements of a tuple just like a list, and tuples can be multi-dimensional (tuples within tuples). However, you cannot use an indexed element of a tuple on the left side of an expression.

Tuples are often used to hold information that will not change, such as color values, image data, or other scientific measurements.

2.7 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. We've already looked at the concept of definite loops, or loops for a fixed number of iterations, by using the range function. The `for` statement, however, can do more than just definite loops.

2.7.1 for

The `for` loop provides a more convenient syntax and mechanism for iterating over an array or a list of elements. We've already looked at simple examples of for loops that execute a certain number of times. The formal syntax of a for loop is given below.

Syntax:

```
for <variable> in <sequence>:
    <statements>
```

The loop variable can be any legal variable name in Python. The variable has scope within the enclosing function or module (file) block, so it exists after the for loop is completed.

The sequence is the collection of things over which the loop should iterate. A sequence can be a list, string, or tuple. The first time through the loop, the variable gets the value of the first item in the sequence. The second time through the loop, the variables gets the value of the second item in the sequence, and so on, until the loop processes all of the elements.

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

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

If you want the loop variable to 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 sequence of items

- The list can be a string, in which case it works through the elements of the string
- The list can be a list or tuple, which is are a linear sequence of objects
- The range() function is your friend

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'>
```

2.7.2 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
```

2.7.3 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
```

2.8 Review of program design

Now that you have written some programs, the process of program design may actually be meaningful. The most important step, by far, is step number one. 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.

3 Zelle Graphics objects

The Zelle graphics package is organized around the concept of objects. Objects are collections of information and functions. Objects are defined by a what we call a class definition, which is simply how we specify which information and which functions belong with an object. When we talk about object information, we describe them as **fields**, rather than variables. When we talk about object functions, we describe them as **methods**.

Objects were originally developed to facilitate a different type of design than that generally used with functions.

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.

3.1 Working with graphics objects

The basic objects in the Zelle graphics package are:

- GraphWin - a window
- Image - a collection of pixels in a 2D grid; the data can come from an image file
- Point - a 2D point with (x, y) values
- Line - a line connecting two points
- Rectangle - an axis-oriented rectangle defined by two points
- Circle - a circle defined by a point and a radius
- 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.

```
pt = Point( 50, 50 )
```

In the above example, the variable `pt` gets a new Point object at location (50, 50).

3.1.1 Images

Digital images today are one of the most common methods of sending information across computers. Capturing images is easy, and most of us have some method of taking images with us all of the time. Computers have to be able to both represent and manipulate all of the information in an image.

In its most basic form, an image is a set of measurements of light captured on a 2D planar sensor in a grid arrangement. Each block in the grid is called a **pixel**. For a color image, the camera measures three different properties of light at each pixel, which we generally call the red, green, and blue measurements. Internally, these measurements are often represented as numbers between 0 and 255. A number between 0 and 255 requires 8 bits ($2^8 = 256$), or one byte, to store in memory. That means each pixel requires three bytes in memory. Some modern cameras are now so sensitive that only 256 measurement values is not enough, so they require more memory space per pixel.

The main point is that an image consists of pixels, and for color images each pixel has three values: red, green, and blue. This is the same way we have been thinking about colors in turtle graphics, but the range of values is slightly different ([0, 255] versus [0.0, 1.0]).

Because images form a 2-dimensional grid, almost all programs that read, write, send, or manipulate images will access an individual pixels using two numbers. The two numbers correspond to the row and column of the pixel. You can think of the row and column like an address. First you find the corresponding row (10th street), then you go to the corresponding column (#256).

In the Zelle graphics package, an image is stored in a data type called an Image (big surprise). The Image class has methods for reading and writing images, creating a blank image, and for accessing and modifying pixels. The example below shows how to read an image, request its width and height, move the image to a particular location, and draw it into a window.

Example: Creating and displaying an image

The following code shows how to read an image from a file and display it in a window using the Zelle graphics package.

```
import graphics

# open the image and read the data
theFilename = 'miller.ppm'  # the file name of the image
theImage = graphics.Image( graphics.Point(0, 0), theFilename )

# create a window to display the image, make it the same size as the image
win = graphics.GraphWin(  theFilename, theImage.getWidth(), theImage.getHeight() )

# move the image to the center of the window, then draw it
theImage.move( theImage.getWidth()/2, theImage.getHeight()/2 )
theImage.draw(win)

# wait for the user to click in the window, then close it and terminate
win.getMouse()
win.close()
```

3.1.2 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. In the prior example, the line `win.close()` should be interpreted as executing the close method on the window stored in the variable `win`.

3.1.3 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

So how do we make a copy of an object? The objects in the Zelle graphics library all possess a `clone()` method 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.

Example: Creating and manipulating graphics objects

The following code shows how to import the graphics package and then create different types of objects. Note that some objects take other objects (Points) as arguments. Each of the assignments creates a new object of the specified type.

```
import graphics

win = graphics.GraphWin( 'title', 400, 400 ) # creates a 400x400 window
ptA = graphics.Point( 50, 50 ) # creates a point at (50, 50)
ptB = graphics.Point( 100, 50 ) # creates a point at (100, 50)
lineA = graphics.Line( ptA, ptB ) # creates a line between A and B
circA = graphics.Circle( ptA, 30 ) # creates a circle at ptA with radius 30
```

Lists can hold any type of information, including object references. Therefore, we can easily put all of the objects created by the code above into a list.

```
scene = [ ptA, ptB, lineA, circA ]
```

Finally, since all of the graphics objects support a specific set of methods–e.g. draw, undraw, move–we can loop over the list to manipulate all of the objects in the same way.

```
# the following draws all of the elements into the given window
for element in scene:
  element.draw( win )

# the following moves all the items in the list by
# 20 in x (right) and 40 in y (down)
for element in scene:
  element.move( 20, 40 )
```

We can also duplicate the scene by cloning all of the objects and putting them into a new list. Since cloning creates a copy of the object, we can move the duplicate without modifying the original objects.

```
duplicate = []
for element in scene:
  duplicate.append( element.clone() )

for element in duplicate:
  element.move( -60, -10 )
```

3.2 Designing a Scene

Problem: design an animated scene

1. Define the task
2. Identify the inputs and outputs at a global level
3. Recursively break the problem into smaller steps
4. Identify individual functions, noting steps with similar inputs and outputs
5. Generate intermediate representations of the algorithm
6. Write code
7. Verify and test the code

Note that steps 3-7 repeat and that verifying and testing code should be done continuously during the process of development.

Let's pick a simple animated scene of a set of things moving around a scene (e.g. birds flying). Now we have to more carefully define what we mean by an animated scene of objects.

- The program should create a graphics window.
- The program needs to generate an initial scene consisting of graphical objects.
- The program needs to update the appearance/location of the objects on a regular basis over time.

The output of our program is defined by the task: generate a scene with one or more objects that changes over time. What are the inputs to our program? What are the things we need to define before the program begins?

- How many objects (e.g. birds) do we want to create?
- How many times do we want to update the animation (number of frames)?
- How much delay do we want between frames?
- Other parameters that modify the appearance of objects in the scene (e.g. scale)?

Overall, the task has two components to it. First, we have to create the initial scene and all of the objects within it. If we want to update the scene, we need to keep references to the objects we want to update.

Second, we need to loop over the number of frames and update the scene for each frame.

At this point in the design process we can begin to write our program using top-down design and focusing on correctly handling the inputs. The following shows the top level executable function that correctly handles all of the inputs and has placeholders (print statements) representing the calls to create the scene and update it. We can use this to test that it handles the inputs correctly and that the overall control flow works properly.

The program also introduces the concept of the try/except statement. Python provides this statement to enable the programmer to catch errors that occur when, for example, Python is unable to convert a string to an integer or a floating point number. If an error occurs within the code inside the try block, then control immediately moves to the first statement in the except block. The except block does not execute if no errors occur within the try block.

```
# Bruce Maxwell
# Fall 2010
#
# Design example for an animated scene
#

import graphics
import sys
import time

def main( args ):

  # test if there are enough arguments
  if len( args ) < 4:
    print 'Usage: ' + args[0] + ' <Num birds> <Num Frames> <Delay>'
    exit()

  # try to convert the arguments to proper types
  try:
    numBirds = int( args[1] )
  except:
    numBirds = 5
    print 'Invalid number of birds argument, continuing with ' + str(numBirds)

  try:
    numFrames = int( args[2] )
  except:
    numFrames = 10
    print 'Invalid number of frames argument, continuing with ' + str(numFrames)

  try:
    delay = float( args[3] )
  except:
    delay = 1.0
  print 'Invalid delay time, continuing with ' + str( delay )

  print 'Using: ', numBirds, numFrames, delay

  # create the scene
  print 'creating scene'

  # for the number of frames
  for frame in range( numFrames ):
    # delay
    time.sleep( delay )
    # update the scene
    print 'updating scene, frame', frame

if __name__ == "__main__":
  main( sys.argv )
```

4 Grammars and L-Systems

As computer scientists, one of the things we do is think about how to model systems in the real world. One system of interest is language. Written language is defined by attributes like characters (symbols), words (identifiers), grammar (syntax), and semantics. Computer languages have the same attributes, only simpler and more well-defined than most natural languages.

Once we have defined the symbols of a language and how to generate words, or identifiers, the syntax of a language is defined by its grammar. A grammar is defined as follows.

- An alphabet, or set of symbols that encompass all of the symbols in the language. Some of these symbols are Terminals, or represent actual written characters, while some symbols are Non-terminals that represent conceptual elements of the language (e.g. nouns, verbs, phrases).
- A start symbol, from which all valid strings of the language are derived.
- A set of rules that define the relationship between symbols.

Noam Chomsky defined a series of hierarchies of grammars that encompass very simple to very complex languages.

- Regular grammars: rules can have at most one non-terminal symbol on the right side.
- Context-free grammars: rules can have only one symbol on the left side, which means rule application is independent of context.
- Context-sensitive grammars: rules can have multiple symbols on the left side, which means context can affect which rule is applied. A rule cannot map to a smaller set of symbols.
- Unrestricted grammars: there are no restrictions on the form of the rules.

A grammar describes all of the possible sentences in a language. Grammars are useful for many things, including parsing and compiling computer programs correctly and efficiently. Most word processors also have built-in grammars for natural language that enable them to identify potential problems in sentence structure.

Chomsky's grammars are used by taking one rule at a time and applying the rule to the current string, either in a generative sense (creating sentences) or in a deconstructive sense (diagramming sentences).

Grammars are also useful for modeling things in nature. Astrid Lindenmayer, a biologist, developed a different type of grammar hierarchy that uses a different mode of rule application. L-systems are primarily generative systems that build strings, or sentences, in a language. In L-system grammars, all of the rules apply to all of the symbols simultaneously in order to convert an input string into an output string.

Consider, for example, the development of a bacteria. It may go through a sequence of grow and divide processes over time. We can model this process using a simple L-system.

- Alphabet: A, B
- Start string: A
- Rules: $A \rightarrow BB$; $B \rightarrow A$

Now consider the sequence of strings over several generations. Note how the system displays the exponential growth of cell division.

- A
- BB
- AA
- BBBB
- AAAA
- BBBBBBB
- AAAAAAA

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

Here is an example of a simple DOL system.

- 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.

What happens when we give a graphical meaning to the characters in the string?

- Meaning: F is go forward by distance δ (e.g. 5)
- Meaning: + is go left by angle $\theta = 60°$
- Meaning: - is go right by angle $\theta = 60°$

To interpret the string, loop over each character in the string and execute the action indicated by the character.

The Koch snowflake is a graphical example with an L-system interpretation.

- Base string: F–F–F–
- Symbol: F
- Rule: F+F–F+F

In the Koch snowflake, we take each linear edge of the current shape ’F’ and replace it with four edges where two of the edges poke out as a triangle ’F+F–F+F’. In just two iterations, the original string quickly grows to a complex shape.

```
F--F--F--
F+F--F+F--F+F--F+F--F+F--F+F--
F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--
```

4.1 Generating Strings

L-systems are normally used in a generative mode. Beginning with a short start string, consecutive application of a well-designed set of L-system rules results in a series of strings that emulate the growth of an organism. L-systems can also generate many interesting fractal geometric shapes.

In order to generate strings, we first need a data structure to hold the L-system information. The key components of an L-system are the base string and the set of rules. The base string is simply a character string. Each rule consists of two parts: the character to be replaced (symbol) and the string that will replace it (replacement).

A list with two elements provides a mechanism for holding all of this information. The first element is the base string, and the second element is a list of lists, where each sublist is a rule that has two strings in it: the symbol and its replacement. For example, the Koch snowflake L-system above would have the representation given below.

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

The update rule for an L-system is to traverse the current string–starting with the base string–and replace any instance of the left side of a rule (e.g. ’F’) with its replacement (e.g. ’F+F–F+F’). Any character without a rule has the implicit rule of $c \rightarrow c$. In the example above, since + and - do not have rules, they would be copied into the new string.

The pseudo-code for the replacement algorithm is as follows.

```
# Given: the input string (e.g. curString)
# Assign to a variable (e.g. newString) the empty string
# for each character in the current string
    # if there is a rule for the current character
        # add the replacement string to the end of the new string
    # else
        # add the current character to the end of the new string
```

4.2 Interpreter

An interpreter is a program that takes information in one form and converts it to information in another form. In some cases, the new form of the information may be actions taken by the computer. Executing an interpreted programming language is one example (e.g. Python). Creating graphical output is another example (e.g Postscript on a printer).

The key concept in writing an interpreter is that the transformation of information from one form to another is completely defined by the programmer designing the interpreter. If the programmer wants the character F to draw a flock of birds, then the meaning of 'F' as an input to the interpreter is a flock of birds. If the programmer wants the character F to draw a circle, then the meaning of 'F' as an input to the interpreter is a circle.

In the case of context-free grammars, the interpreter need to look only at the current input–in the case of L-systems a single character–in order to determine what to do. Therefore, the interpreter that converts an L-system output string into turtle graphics can loop over the string one character at a time and use an if statement to specify the set of actions for each character.

4.2.1 Branching L-systems

When modeling a biological system, it is not always possible to describe a shape as a single, linear sequence of forward motions and turns. Consider a tree, for example, which may branch out in many directions from a single location.

- Any single branch may have two or more child branches connected to it..
- The child branches may have their own child branches.
- We want to avoid re-tracing our steps, whenever possible.

In order to avoid backtracking, we have to give our interpreter a memory. 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 remember only one position, we can recover from only 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
- We push clean plates on top
- We pop the top plate off the stack to use

The list data structure supports both of the two required 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 L-system characters used for store and restore are the left and right bracket: `'['`, `']'`

When the interpreter begins parsing a string, it first creates a variable for the stack and initializes it to the empty list.

- When the interpreter finds a left bracket character, it appends the turtle's position and heading to the end of the list.
- When the interpreter finds a right bracket character, it pops the turtle heading and position from the end of the list and resets the turtle's position and heading.

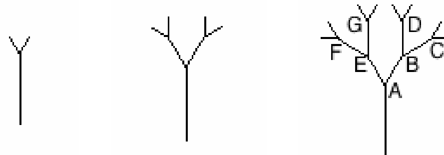
Example: Branching L-system

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 $[0, 360)$.

5 Files

Reading and writing from files is a useful capability on a computer because it permits us to store data indefinitely and pass it between different applications or programs. Files are sequences of data stored on a storage medium of some time. The simplest type of file is a text file, which is a sequence of characters. In addition to the usual alphanumeric and punctuation characters, some of the characters in a text file may be control characters such as a newline ('\n')or tab ('\t'). It's useful to think of text files as strings separated by newlines.

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. A program can access any files in the working directory by using just their filename. All other files either need to be specified relative to the file tree root (/) or relative to the current working directory.

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. To create a new file object, we use the name of the class, which is `file`. In addition to the filename, we need to tell the file class whether the program is going to read the file or write to the file. The following two examples show both types of usage patterns. Note that it is important to close a file, especially after writing content to a file. Otherwise, the operating system may not write the file data to the disk. Closing a file forces the computer to flush all information for a file to the disk.

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

# read all of the contents of a file
fp = file( 'myfilename', 'r' )
text = fp.read()
print 'The contents of the file are:'
print text
fp.close()
```

When a program opens a file for writing, the computer creates a new file–even if an old file of the same name exists–and writes the new data to that file. Therefore, any contents in the old file are lost.

When a program opens a file for reading, the file must exist or Python will throw an exception. A common coding pattern when opening files is to use the `try / except` control structure to check that the file opens correctly.

```
# try opening the file
try:
  fp = file( 'myfilename', 'r' )
except:
  print 'Unable to open file ', 'myfilename'
  exit()

# read the file
text = fp.read()
print 'The contents of the file are:'
print text
fp.close()
```

When reading a file, there are three variations on the `read` method 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.

Python also has a number of useful tools for parsing text file. These include both file class method and string class methods.

Consider the task of reading a document and counting the number of times the word 'the' appears. The algorithm is as follows.

Algorithm	Code
Set a counter variable to 0	`counter = 0`
Open the file	`fp = file( filename, 'r')`
Read all of the lines of the file and store them in a list	`lines = fp.readlines()`
Close the file	`fp.close()`
For each line of the file	`for line in lines:`
– Split the line into words and store the words in a list	`  words = line.split()`
– For each word in the list	`  for word in words:`
—- If the word is 'the',	`    if word == 'the':`
—— increment the counter variable	`      counter += 1`
Print (or return) the counter variable	`print counter`

Executing the code on the right properly counts the number of times the string 'the' appears in the file.

As a second example, consider the following L-system information, stored as a file using the following format.

```
base F--F--F
rule F F+F--F+F
```

Using this format, the first string on each line tells us what piece of L-system information is stored on that line. The remaining strings on the line have specific meanings given the first string. The file class gives us the ability to read a file one line at a time, or to read all of the lines at once, storing each individual line as the elements of a list.

```
oneLine = fp.readline()
```

```
allLines = fp.readlines()
```

In the first case, the file returns a single line of the program, and the variable oneLine receives a single string. In the second case, the file returns a list of all of the lines of the file, so the variable allLines receives a list of strings. Once we have a line of text, stored as a string, we can use the string methods to parse the line into words using the `split` method.

```
words = oneLine.split()
```

Then the program can examine the words one at a time, using appropriate control structures to respond to the contents of the file.

For an L-system file such as the one given above, the algorithm for parsing the file is given below.

1. Open a file
2. Read all of the lines in a file
3. Close the file
4. Initialize an empty L-system, which will be `[ '', [] ]`
5. For each line of the file
   (a) Split the line on spaces and store the word list
   (b) If the first word is 'base'
      - set the base string of the L-system
   (c) if the first word is 'rule'
      - add the rule to the L-system
6. Return the new L-system

The code to implement the algorithm is given below. This functions readFromFile, createLsystem, setBase and setRule are part of project 7.

```
def readFromFile( filename):

  # open, read, and close the file
  fp = file( filename, 'r' )
  lines = fp.readlines()
  fp.close()

  # create a new empty lsystem
  lsys = createLsystem()

  # parse the lines of the file
  for line in lines:
    words = line.split()
    if words[0] == 'base':
      setBase( lsys, words[1] )
    elif words[0] == 'rule':
      addRule( lsys, [ words[1], words[2] ] )

  # return the lsystem
  return lsys
```

6 Classes

A class is a language structure that allows us to define our own objects. An object is a data structure that contains both information (fields) and functions (methods). An object has its own symbol table, just like a module (an imported python file) or a function. The object's symbol table holds all of the information about the fields and methods it contains.

The parts of a class include the following.

- 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

The syntax of a class is as follows.

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 A) defined inside a class block and B) the first argument to a class method is always the variable `self`. The `self` variable is a reference to the object data to which the method should be applied. It is, in effect, a reference to the object's symbol table. That means the function has access to all of the fields and methods defined for that object. In all other ways, methods act just like regular functions. They have their own symbol table, they can have any number of parameters, and variables defined inside a method do not have scope outside of the method.

The class definition itself also creates a symbol table in Python associated with the class name. When we create a new object, which we define as an instance of a class, Python copies the contents of the class symbol table–which are usually method references–to the object's symbol table. That gives the object access to the class methods.

To create a new instance of an object, use the class symbol as a function call and assign its result to a variable. To create a new Point object from the Zelle graphics library, for example, use the syntax below, which creates a Point object and puts a reference to it in the variable `pt`.

```
pt = graphics.Point( 100, 100 )
```

The `__init__` function is a special function that Python calls when the user creates an instance of an object. For example, in the Zelle graphics package, when you call the function GraphWin(), Python calls the `__init__` function for the GraphWin class 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. Default values are 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 information in order to initialize itself, then there should not be any default values for those parameters.

The example below shows several aspects and features of classes. First, the __init__ function adds two fields to the object: name and year. It uses default values in case the calling program does not pass in initial values. Second, it provides methods for getting and setting the values of the name and year fields. These functions are called **accessors**. Third, it defines a method __str__ that is called by Python whenever the object is cast to a string using the `str()` function. This occurs, for example, whenever the object appears as an argument to `print`.

Example: A Student Record

A common example used to teach classes and objects is a database entry. Databases are an ideal application for object-oriented design since they consist of many separate items we want to create, access, modify, and delete. The following is a simple Student object that holds the student's name and year.

```
class Student:
    # initialize the object and create its two fields
    def __init__(self, name='', year=-1):
        self.name = name
        self.year = year

    # accessors for the name and year fields
    def setName(self, name):
        self.name = name

    def setYear(self, year):
        self.year = year

    def getName(self):
        return self.name

    def getYear(self):
        return self.year

    # utility function
    def inRange(self, startYear, endYear):
        return startYear <= self.year and self.year <= endYear

    # called whenever the str() function is used with a Student
    def __str__(self):
        return ' %d  %s' % (self.year, self.name)
```

After Python scans through the above code, the contents of the Student symbol table include function references to each of the methods: __init__, setName, setYear, getName, getYear, inRange, and __str__. Code inside the methods does not have any impact on the class symbol table.

Note the use of methods to get and set object fields. This is a practice called **encapsulation**, which is defined as hiding the details of implementation from a programmer using the class. Encapsulation is one of the main principles of object-oriented design, and it enables classes to change their implementations without impacting programs that use those classes.

To see how we might use the student class, consider the following test function, which would be appropriately placed in the same file as the Student class to test the class functionality.

```
# unit test function
def test():
    students = []
    students.append( Student( 'Andrew', 2011 ) )
    students.append( Student( 'Mary', 2013 ) )
    students.append( Student( 'Katherine', 2012 ) )
    students.append( Student( 'Martha', 2012 ) )

    print 'All students'
    for s in students:
        print s
    print '\n2012 students'
    for s in students:
        if s.inRange(2012, 2012):
            print s

if  __name__ == "__main__":
    test()
```

Each time the code calls the Student function, it creates a new Student object, copies the Student symbol table to the object's symbol table and then calls the `__init__` method. The `__init__` method then adds two new fields to the object's symbol table–name and year–and copies the information from the name and year parameters, which exist in the `__init__` method's symbol table. While the `__init__` method's symbol table disappears once the method completes, the object's symbol table remains in existence as long as we have a reference to it stored in some variable.

Example: A Bird Class

The following is an implementation of the same bird concept we implemented using lists. By using a class, we can make the Bird object act just like any other graphics object by writing methods for draw, undraw, move, and other graphics modifiers.

```
# Example of a class for generating a flying Bird
import graphics
import math
import random

class Bird:
    def __init__(self, x, y, scale):

        # create a bird from graphics objects
        ptA = graphics.Point( x, y )
        ptB = graphics.Point( x - 30*scale, y - 10*scale )
        ptC = graphics.Point( x + 30*scale, y - 10*scale )
        ptD = graphics.Point( x - math.sqrt( 30*30 + 10*10)*scale, y )
        ptE = graphics.Point( x + math.sqrt( 30*30 + 10*10)*scale, y )
        ptF = graphics.Point( x - 30*scale, y + 10*scale )
        ptG = graphics.Point( x + 30*scale, y + 10*scale )

        # wings in 4 positions
        l1 = graphics.Line( ptA, ptB )
        l2 = graphics.Line( ptA, ptC )
        l3 = graphics.Line( ptA, ptD )
        l4 = graphics.Line( ptA, ptE )
        l5 = graphics.Line( ptA, ptF )
        l6 = graphics.Line( ptA, ptG )
        l7 = graphics.Line( ptA, ptD )
        l8 = graphics.Line( ptA, ptE )

        # indicate that no lines are currently drawn
        self.position = -1

        # store the graphics objects
        self.things = [l1, l2, l3, l4, l5, l6, l7, l8]

    # draw the object into the given window
    def draw(self, win, pos=0):
        if pos < 0:
            self.position = 0
        else:
            self.position = pos % 4

        self.things[self.position*2].draw(win)
        self.things[self.position*2+1].draw(win)

    # move all of the graphics items
    def move(self, dx, dy):
        for item in self.things:
            item.move(dx, dy)
```

```
    # undraw any currently drawn graphics items
    def undraw(self):
        if self.position >= 0:
            self.things[self.position*2].undraw()
            self.things[self.position*2+1].undraw()
            self.position = -1

    # set the fill color
    def setFill(self, color):
        for item in self.things:
            item.setFill( color )

    # set the outline color
    def setOutline( self, color):
        for item in self.things:
            item.setOutline( color )

    # set the width of the lines
    def setWidth( self, width ):
        for item in self.things:
            item.setWidth( width )

    # animate the bird
    def animate( self, win ):
        if self.position >= 0:
            self.things[self.position*2].undraw()
            self.things[self.position*2+1].undraw()

        self.position = (self.position + 1) % 4

        self.things[self.position*2].draw(win)
        self.things[self.position*2+1].draw(win)


def test():
    win = graphics.GraphWin('birds', 400, 400)

    bird = Bird( 200, 200, 2)
    bird.draw(win)
    bird.setOutline( 'blue' )
    bird.setWidth( 3 )

    for i in range(30):
        bird.move( random.randint( -5, 5 ), random.randint(-5, 5) )
        bird.animate(win)

    win.getMouse()
    win.close()

if __name__ == "__main__":
    test()
```

In the example above, there are a number of characteristics of classes that make programming easier than using our prior list representation

- We can store information in named fields instead of in specific positions in a list.
- Information stored in a field of an object is available in all methods of the class using the self reference.
- Since information is stored with each object, each object can have unique properties.
- Integrating the Bird object with other graphics objects is trivial: they all support the same methods.

6.1 Multiple Rule L-Systems

Thus far, we have used deterministic context-free L-systems, called DOL-systems, with a single rule. L-systems with one rule are limited in the complexity of objects they can represent. Multi-rule DOL-systems require a slightly more flexible replacement algorithm.

While our existing list representation of L-systems can easily handle multiple rules, moving to a class representation has several benefits. First, we can use named fields to hold the L-system data. Second, we can connect the appropriate functions to L-system objects, simplifying the parameter lists and the code.

Recall that deterministic, context free L-systems are defined by the following rules.

- Each symbol has a single replacement rule (deterministic)
- If a symbol has no explicit rule, then it has the identity rule $S \Rightarrow S$
- 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 new string must be built separately from the original string.

Since each symbol in a DOL-system 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 concatenate the replacement strings into a new list.

Algorithm for a single iteration of multi-rule replacement:

1. Initialize the output string to the empty string.
2. For each character C in the base string
   (a) Set a flag variable to False
   (b) Loop through the list of rules
      - If the symbol for the rule is C, then append the rule to the output string and break and set the flag variable to True
   (c) If there is no rule for C (flag is False), then 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 program finds a rule for the symbol. A simple flag variable that gets set to false prior to looping through the list of rules is sufficient. If the program finds a rule, it sets the flag variable 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:

$$\text{Instructions} \approx \text{Iterations} \times \text{String Length} \times \text{Rules} \tag{2}$$

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 the complexity of the expression

An Example L-system Class

The class below is an example of how we might represent an L-system using a class structure instead of a list. Note that we can use named fields to hold the various pieces of information instead of locations in a list. Likewise, we can override the `__str__` method and have the class create a nicely formatted representation of the L-system information when the user prints an L-system object.

```
class Lsystem:
    def __init__(self):
        self.base = ''
        self.rules = []

    # sets the base string
    def setBase( self, base ):
        self.base = base

    # copies a new rule into the rules list
    def setRule( self, rule ):
        newrule = []
        for item in rule:
            newrule.append( item )
        self.rules.append( newrule )

    # reads the L-system data from a file
    def read( self, filename ):
        # implementation left for lab

    # builds a string given the L-system and number of iterations
    def buildString( self, iterations ):
        # implementation left for lab

    # override the __str__ method to enable nice output
    def __str__(self):
        s = 'base: ' + self.base + '   rules: ' + str( self.rules )
        return s
```

6.2 Inheritance

The interpreter we've developed for converting L-system strings into graphics is a general purpose drawing machine. We can pass any string consisting of valid characters into the interpreter, along with a distance and angle, and have the interpreter draw it. Consider, for example, the following strings and their distance and angle information.

- Triangle: (string: F+F+F+, distance: 50, angle: 120)
- Square: (string: F+F+F+F+, distance: 50, angle: 90)
- Star: (string: [F]+[F]+[F]+[F]+[F]+[F]+[F]+[F]+, distance: 20, angle: 45)

We could write a class to support each of these objects and use the interpreter class to draw them.

```
import interpreter as it

class Triangle:
    def __init__(self):
        self.string = 'F+F+F+'
        self.angle = 120
        self.distance = 50

    def draw(self, x0, y0, scale):
        x = it.Interpreter()
        x.goto( x0, y0 )
        x.drawString( self.string, self.distance * scale, self.angle )

class Square:
    def __init__(self):
        self.string = 'F+F+F+F+'
        self.angle = 90
        self.distance = 50

    def draw(self, x0, y0, scale):
        x = it.Interpreter()
        x.goto( x0, y0 )
        x.drawString( self.string, self.distance * scale, self.angle )

class Star:
    def __init__(self):
        self.string = '[F]+[F]+[F]+[F]+[F]+[F]+[F]+[F]+'
        self.angle = 45
        self.distance = 30

    def draw(self, x0, y0, scale):
        x = it.Interpreter()
        x.goto( x0, y0 )
        x.drawString( self.string, self.distance * scale, self.angle )
```

Note that the fields are all the same and the draw functions are all the same. In fact, the `__init__` functions are all the same except for the hard-coded values on the right side of each assignment. There is duplicate code in each class, which is poor programming practice, likely to cause errors, and a waste of time.

Classes enable us to avoid duplicating code shared among similar classes by enabling a design mechanism called inheritance. Think of inheritance as enabling a tree structure of programming. Put fields and methods shared among a group of classes into a single parent class. Then have each child class inherit the methods and fields of the parent. The child class code needs to include only any class-specific or additional methods and fields. The following set of four classes has exactly the same functionality as the example above, but is much more compact and contains no duplicate code.

Note the syntax of inheritance. When defining the Triangle class, we use the statement

```
class Triangle(Shape):
```

which says to define a class Triangle with the parent Shape. In terms of symbol tables, when Python builds the Triangle symbol table, it copies the contents of the Shape class symbol table–its methods–to the

Triangle class symbol table. Then, when we create an object of type Triangle, Python copies the contents of the Triangle class symbol table to the object's symbol table.

Any methods defined in the child class **override** the parent method. Therefore, the `__init__` method in the Triangle class replaces the reference to the Shape `__init__` method in the Triangle class symbol table.

```
class Shape:

    def __init__(self, string, angle, distance):
        self.string = string
        self.angle = angle
        self.distance = distance

    def draw(self, x0, y0, scale):
        interpreter.goto( x0, y0 )
        interpreter.drawString( self.string, self.distance * scale, self.angle )

# Triangle class inherits Shape
class Triangle(Shape):

    def __init__(self):
        Shape.__init__(self, 'F+F+F+', 120, 50 )

# Square class inherits Shape
class Square(Shape):

    def __init__(self):
        Shape.__init__(self, 'F+F+F+F+', 90, 50 )

# Star class inherits Shape
class Star(Shape):

    def __init__(self):
        Shape.__init__(self, '[F]+[F]+[F]+[F]+[F]+[F]+[F]+[F]+', 45, 30 )
```

What should be clear from the Shape class examples is that inheritance can save us a lot of time and effort coding. Furthermore, it reduces the chance of future errors by eliminating duplicate code. Finally, if we want to modify the implementation of the draw function to incorporate new capabilities or parameters, making a single change will change the functionality for all classes, as shown in the following example.

```
    def draw(self, x0, y0, scale, color = (0, 0, 0) ):
        interpreter.goto( x0, y0 )
        interpreter.color( color )
        interpreter.drawString( self.string, self.distance * scale, self.angle )
```

Our symbol table model of Python's internal state enables us to describe inheritance in terms of copying, or overwriting entries in a symbol table. The Shape symbol table contains an entry for each method. The definition of the Triangle symbol table tells Python to copy the Shape symbol table to the Triangle symbol table. Then the def statement within the Triangle class tells Python to overwrite the `__init__` entry of the Triangle symbol table. When some code creates a new Triangle object

```
t = Triangle()
```

Python copies the Triangle class symbol table entries–including the overwritten `__init__` entry–into the new object's symbol table. The fields created by the Shape `__init__` go into the object's symbol table.

6.3 Design Using Classes

Classes enable and support four principles of object-oriented programming that can be beneficial in designing large systems.

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

Modularity is well-supported by functions, and both classes and the use of methods in classes support modularity in class design. Modularity is universal across languages and modes of programming.

Encapsulation, however, is not supported by functions, because all data for a function must either be hard coded into the function or passed in as an argument. Therefore, the programmer using a function must know the form of the data required by the function. Object-oriented design using classes supports the hiding of implementation details, such as how the data required by a method is stored. A programmer using a class does not need to keep track of the data it uses, and the data stored in an object does not need to be maintained or passed between functions by the programmer. In many object-oriented languages permit the class designer to specify certain methods and fields as private, which means they can be accessed only in code within the class.

Inheritance is specific to an object-oriented design approach. Inheritance enables child classes to use the methods and fields defined in the parent class. It helps avoid code duplication, assists modularity and encapsulation, and in languages with static typing it enables polymorphism.

Polymorphism is well supported in dynamically typed languages as any variable can hold any data type. However, in typed languages (e.g Java or C++) polymorphism is enabled by classes and inheritance. Variables typed as a parent class can hold and make use of objects typed as a child class. In Python, polymorphism is ubiquitous. If a set of classes support a common set of methods, we can write algorithms based on those common methods.

7 Dictionaries

What is a dictionary? It is a collection of items arranged in alphabetical order.

- Each item has a unique location.
- The item has a key that indicates its location in the collection.

Why do we use dictionaries?

- We need to keep track of a large number of items.
- We need to be able to quickly locate each item.
- We need to keep information only about specific items, not every possible item (e.g. word).

The idea of mapping a key–a word–to a value–a definition–is both common and useful. 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 dictionaries use keys to access data, not numbers. In Python, 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.

```
>>> a = (1, 2, 'hello')
>>> a
(1, 2, 'hello')
>>> a[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> (x, y) = (4, 5)
>>> x
4
>>> y
5
```

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. As noted above, a keyword can be any non-mutable type.

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

Just like a list, the value of a dictionary entry can be any type, mutable or immutable (numbers, strings, lists, objects, etc.). The dictionary class also includes a number of useful methods.

- keys() - returns the list of keys in the dictionary, which can be useful for looping over the 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: Looping over the contents of a dictionary.

```
>>> a = {'a': 13, 'b': 18, 'c' : 42}
>>> for item in a.keys():
...     print item, a[item]
a 13
c 42
b 18
```

7.1 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 make an array with only 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 new hash function.

Example: Multiple rule string replacement using a dictionary.

Using a dictionary makes the task of multi-rule string replacement simpler. Consider the task of executing one round of replacement using a replace method in the Lsystem class. If we store the set of rules in a dictionary we can use each rule's symbol as the key and each rule's replacement as the value in the dictionary entry.

```
def replace( self, curstring ):
  nstring = ''
  for ch in curstring:
    nstring += self.rules.get( ch, ch )
  return nstring
```

7.2 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

```
pair = [ [ 'blue', 'sky' ], [ 'green', 'grass' ], [ 'red', 'apple' ] ]
dt = dict()  # create a dictionary using its class name

for pair in keyValueList:
    dt[ 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.

Because the idea of building a dictionary from a list of pairs is so common, the constructor for the dictionary class, the `__init__` function, can take in a list of pairs and build the dictionary from that list. The list of pairs could also be a tuple with tuple pairs.

```
listpair = [ [ 'blue', 'sky' ], [ 'green', 'grass' ], [ 'red', 'apple' ] ]
tuplepair = ( ( 'road', 'construction' ), ( 'traffic', 'light' ) )

d1 = dict( listpair )  # pass the list of pairs to the class __init__
d2 = dict( tuplepair ) # pass the tuple of tuple pairs to the class __init__
```

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.

```
# open, read, and close the file
fp = file( 'filename', 'r' )
lines = fp.readlines()
fp.close()

# create a new dictionary
dt = {}

# for each line, split it and make the first word the key
for line in lines
    line = line[:-1]       # remove the newline
    words = line.split()    # split

    dt[ words[0] ] = words[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]
```

7.3 Stochastic L-Systems

The next modification to L-systems we're going to make is to permit more than one rule per symbol to exist. Another way to phrase it is that there may be more than one possible replacement for each symbol.

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 dictionary where the key is the symbol, and a list containing the replacement string is the value.

```
{ 'F' : [ 'f[+F]F[-F]F' ] }
```

The complete set of rules is a dictionary of the individual rules.

```
{ 'F' : [ 'f[+F]F[-F]F' ], 'f' :  [ 'ff' ] }
```

To incorporate multiple replacement strings, we can add the alternatives to the value field list of replacements.

```
{ 'F' : [ 'F[+F]F[-F]F', 'F[-F]F[++F]' ] }
```

The symbol is the key, and the possible replacement symbols are in the list `self.rules[key]` . Therefore, if we want to randomly choose a replacement rule, we can use the choice function in the random package.

```
random.choice( self.rules[key] )
```

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

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

```
def replace(self, bstring):
    """Uses the L-system parameters to execute one iteration of replacement"""

    newstring = ''
    for ch in bstring:
           newstring += random.choice( self.rules.get(ch, [ch]) )

    # return the string
    return newstring
```

Because we have only 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). Note the use of the `get` method to access the dictionary. If the particular key does not exist as a rule in the L-system definition, then the get function returns the character in a list as the result.

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

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

- Model: lines with random widths connecting locations imperfectly
- Algorithm:
  1. Get the start point A and the end point B
  2. Perturb A and B using a Gaussian distribution
  3. Move the turtle to A without drawing
  4. Select a randomized width
  5. Move the turtle to B drawing the line
  6. Move the turtle to the proper end point without drawing
  7. Reset the width
- Implementation:
  - Need to set up a field in the interpreter specifying the style
  - Modify the 'F' and 'f' cases to call the proper drawing method

8.1 Line Representation

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.

$$P_{\text{mid}} = (P_1 + P_2)/2 \tag{3}$$

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 $K \in [0, 1]$.

$$P = P_1 + K(P_2 - P_1) \tag{4}$$

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.

8.2 Some NPR Ideas

Sketch: split the line into two parts, possibly vary the width

- Calculate the midpoint
- Perturb the end points and perturb the midpoint twice
- Pick a randomized width
- Draw between the start point and the first mid point
- Pick a randomized width
- Move to the second mid point and draw the second line
- Make sure the turtle gets back to its proper location and orientation

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

Splattering: a sequence of circles along an approximate line

- Generate a sequence of points along the line
- Jitter each point in x and y using a Gaussian distribution
- Place a circle of varying radius (Gaussian or uniform) at each location

8.3 Manipulating Colors

Color provides some interesting opportunities in NPR. For example, a brush stroke may not produce completely identical colors. Some individual brush hairs may contain more paint than others, leading to different levels of saturation within a single stroke. The following color manipulations provide some useful results.

Chromaticity

- The chromaticity of an R, G, B color is the color values divided by their sum. The chromaticity represents the color without the intensity (dark red and bright red have the same chromaticity).

$$\{r, g, b\} = \frac{\{R, G, B\}}{R + G + B} \tag{5}$$

- The meaning of chromaticity tells us that scaling a color uniformly in all color channels maintains the chroma of the color with varying intensity. The only restriction is that the scale factor $\gamma \in [0, 1]$. The scalar γ must be the same for all three color channels, even if it is the result of a random number generator (like random.random()), in order to scale the color properly.

$$(R', G', B') = (R, G, B) * \gamma \tag{6}$$

Saturation

- Saturation describes the strength of a particular color, or chroma compared to a pure grey or white. Shades of grey are completely unsaturated colors, since all of the color channels have same value. A pure red, such as $(1, 0, 0)$, would constitute a fully saturated color, since the minimum common value across the color channels is zero.
- Saturation is normally defined as a function of the maximum or minimum channel and the color's overall intensity. One definition for the HSI color space is given below, where $m = \min\{R, G, B\}$ and $I = (R + G + B)/3$.

$$S_{HSI} = 1 - \frac{m}{I} \tag{7}$$

- An alternative definition of saturation takes into account the difference between the maximum and minimum color channels $C = \max\{R, G, B\} - \min\{R, G, B\}$ and the value of the maximum color channel $V = \max\{R, G, B\}$.

$$S_{HSV} = \frac{C}{V} \tag{8}$$

- One method of creating a set of colors with the same chroma but different saturation is to create a weighted average of the original color and a grey value. If the original color is (R, G, B), and the maximum color value $V = \max\{R, G, B\}$, then the following expression gives a family of colors with the same chroma, but different levels of saturation, where $S_{HSV} \in [0, 1]$. Given $S_{HSV} = 1$, the expression returns the original color. Given $S_{HSV} = 0$, the expression returns a grey color.

$$(R', G', B') = S_{HSV}(R, G, B) + (1 - S_{HSV})(V, V, V) \tag{9}$$

9 Algorithms

9.1 Recursion

Consider the following problem statement.

> Using the turtle, draw a sequence of squares from left to right, where the right side of one square is touching the left side of the next square. The first square should be of size X. The last square should be of size one. Each square should be half the size of its left neighbor.

How could we write a program to solve the task? One method would be to calculate the size and position of all of the squares and then draw them.

A different approach is to think of the problem one square at a time. We know the size of the first square (X). Therefore, the size of the next square must be X/2, and it must start X to the right of the first square. The only difference between the process needed to draw the first square and the process needed to draw the second square is the size of the square.

What if we create a function that executes all of the operations necessary to draw one square?

```
def makeSquare( X, x0, y0 ):
  s = shape.Square(X)
  s.draw(x0, y0)
```

What if we then add one line to the function that specifies how to draw the next square?

```
def makeSquare( X, x0, y0 ):
  s = shape.Square(X)
  s.draw(x0, y0)
  makeSquare( X/2, x0 + X, y0 )
```

While this will draw all of the correct squares, it will not stop. Therefore, we need to check if $X >= 1$ before drawing a square.

```
def makeSquare( X, x0, y0 ):
  if X >= 1:
    s = shape.Square(X)
    s.draw(x0, y0)
    makeSquare( X/2, x0 + X, y0 )
```

Now the function will stop calling itself once it reaches the final size square.

Recursion is the idea of defining a function that calls itself in order to repeat a sequence of operations. There are two requirements for recursion to work properly.

1. There must be a base case where the function does not call itself.
2. Each time the function is called, it must make progress towards the base case.

If a function is recursive, but does not meet both of these conditions, then the recursion is not guaranteed to stop in a finite length of time.

An important aspect of recursion is how a programming language implements functions. Python uses symbol tables to represent the active variables within a function. Each time a function is called, it creates a new symbol table that exists for the duration of the function call. In the case of recursion, each recursive call creates a new symbol table.

For example, consider the following version of the same function as above.

```
def makeSquare( X, x0, y0 ):
  if X >= 1:
    makeSquare( X/2, x0 + X, y0 )
    s = shape.Square(X)
    s.draw(x0, y0)
```

The re-ordered version draws the squares from smallest to largest. If all of the function calls shared a single symbol table, this version would not work properly because the last function call would have the smallest value of X. Instead, when a particular function call terminates, Python removes its symbol table and reverts back to the prior symbol table.

Example: Binary search of a list.

Searching a list of sorted items is a useful capability. A simple recursive algorithm solves the problem efficiently. The idea is to look at the middle element of the sorted 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, 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 $log_2(N)$, where N is the number elements in the list.

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

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

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

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

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

    # otherwise, search 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 it always reduces the scope of the problem.

9.2 Interpreters

An interpreter is a program that converts information from one form to another. The interpreter we have used to convert L-system strings into graphics uses a simple approach that converts each individual character into a graphical action. The characters require no context in order to execute the proper action. In other words, the action is not dependent upon the previous or subsequent characters in the string.

Unfortunately, single characters provide a limited selection of actions. Actions such as forward, turn left, or turn right become more powerful if we can give them parameters. Parameters also permit parameterized L-systems, which can model a greater variety of natural forms.

Consider the string `FF(120)+F(60)+F(60)+F(120)+`. Let the numbers in parentheses modify the following character's action. Therefore, the first `(120)` modifies the first left turn.

How would we write an interpreter to execute the string properly? Identifying a parameter is straightforward: it is a numeric value in between two parentheses. The action it modifies is the next action in the sequence. Therefore, the interpreter needs to keep track of when a parameter starts, the parameter string seen so far, and whether to use a parameter to modify the next action. When executing an action, the interpreter needs to make use of the parameter's value if it exists.

In order to properly handle the parameters, the interpreter needs to be watching for the parentheses. When it finds an open parenthesis, it needs to enter a state where it does nothing but append the input characters into a string until it reaches the close parenthesis. At that point, it can convert the parameter string to an actual value and continue with normal interpretation of the string. Note that these three cases need to be different than the main if-then-else structure that interprets the action characters.

The new interpreter algorithm is as follows. Note that this version shows the cases only for 'F', '+', '-', '(', and ')'. The push and pop operators '[' and ']' work identically as before.

Parametric Interpreter

```
def drawString( self, instring, distance, angle ):

 Initialize the stack to the empty stack
 Initialize the parameter string to the empty string
 Initialize the parameter value to None
 Initialize the parameter state to False

 Loop over the input string

   # handle parameters
   if the character is '('
     assign to the parameter string the empty string
     assign to the parameter state the value True
     continue to the next character

   else if the character is ')'
     assign to the parameter value the cast of the parameter string to a float
     assign to the parameter state the value False
     continue to the next character

   else if the parameter state is True
     append to the parameter string the current character
     continue to the next character


   # begin a new if-then-else statement
   if the character is 'F'
     if the parameter value is None
       Send the turtle forward by distance
     else
       Send the turtle forward by distance * parameter value

   else if the character is '+'
     if the parameter value is None
       Turn the turtle left by angle
     else
       Turn the turtle left by the parameter value

   else if the character is '-'
     if the parameter value is None
       Turn the turtle right by angle
     else
       Turn the turtle right by the parameter value

   # note this is inside the loop, outside the if-then-else
   Set the parameter value back to None
```

The new interpreter will convert the string `FF(120)+F(60)+F(60)+F(120)+` into two forward motions by the `distance` drawString function parameter, a left turn by 120°, another forward by `distance`, a left turn by 60°, and so on through the string.

9.3 Parametric L-systems

In addition to enabling more complex shapes, the purpose of generating a parametric interpreter is to enable parametric L-systems. A parametric L-system supports parameters action characters. Parametric L-systems enable us to model systems where the meaning of the action characters changes with each iteration. For example, a specific 'F' symbol may represent a short branch after one iteration of replacement, but a long branch after multiple iterations of replacement.

Consider the L-system below and two rounds of replacement.

```
base (100)F
rule (x)F  (x)F[+(x*0.67)F][-(x*0.67)F]

(100)F
(100)F[+(67)F][-(67)F]
(100)F[+(67)F][-(67)F][+(67)F[+(45)F][-(45)F]][-(67)F[+(45)F][-(45)F]]
```

Note that the rule is defined in terms of a variable `x` and the rule's symbol consists of the parameter expression `(x)` and the character it modifies. The replacement also contains both `x` and expressions of `x`.

The base string contains an actual numeric value, which replaces `x` in the replacement process. The result is that, during replacement, the expressions of `x` in the replacement modify the initial value of 100, making each subsequent branch shorter.

Executing a round of replacement for a parametric L-system is more challenging than non-parametric versions because the context of a character matters. Also, the algorithm needs to match numeric values in the base string to expressions in the replacement string.

Since the parameter information comes before the symbol it modifies, determining if a character has a parameter is similar to the process used by the interpreter. The '(' character initiates collecting the parameter string; the ')' character terminates the parameter string and converts it to a value.

For standard characters, the algorithm needs to first test if there is a parameter for the character. If not, then it can execute replacement as usual, looking up the correct rule and inserting the replacement into the new string.

For a character with a parameter, the algorithm first needs to check if there is a parameterized version of the rule. In the L-system above, `(x)F` is an example of a parameterized rule. If a parameterized rule exists, the algorithm then picks a replacement–randomly, if the rule has more than one–and substitutes the actual parameter value for the `x` placeholder, evaluating expressions as necessary. The algorithm appends the resulting string onto the new string.

If there is no parameterized rule for the character, then the algorithm needs to check if a non-parameterized rule exists in the L-system. If it does, then it needs to replace all unparameterized instances of the key with the parameterized version. For example, consider the L-system below and two rounds of replacement.

```
base (10)F
rule F  +F--F+

(10)F
+(10)F--(10)F+
++(10)F--(10)F+--+(10)F--(10)F++
```

The parameter follows the character it modifies, enabling the base string to control the meaning of a symbol, even if the rules are not parameterized.

The final case is when the input string has a parameterized symbol, but no explicit rule exists for the character in the L-system. In that situation, the character replaces itself and the parameter follows along. The L-system below shows an example. Note how the parameterized '+' characters in the base string propagate through the replacement process. Because there is no rule for the '+' character, the parameters in the base string affect only the specific cases in the base string and do not propagate to other instances of the '+' character.

```
base F(30)+F(60)+F(90)+
rule F  F+F-

F(30)+F(60)+F(90)+
F+F-(30)+F+F-(60)+F+F-(90)+
F+F-+F+F--(30)+F+F-+F+F--(60)+F+F-+F+F--(90)+
```

The replacement algorithm for parameterized L-systems is given below as pseudo-code. The `substitute` and `insert` functions, included in full, take care of the process of evaluating expressions and handling parameter value substitution.

The `substitute` function takes in 1) a replacement string that may have multiple parameterized symbols using expressions of `x` and 2) a numeric value. It substitutes the numeric value for `x` in each of the parameter expressions and then evaluates each expression to generate a new numeric value. The output string has numeric values in place of the `x` expressions for each parameterized character.

The `substitute` function works by looping over the input string. If it finds no parentheses, then the output string will be the same as the input string.

If, during the loop, it finds an opening parenthesis, then it starts collecting characters into a string until it reaches the closing parenthesis. Then it converts the parameter string, which could be either a number or a mathematical expression of `x`, into a lambda function using the `eval` Python function. A lambda function is a nameless function that returns the value of its last expression. By assigning the lambda function to the variable `lambdafunc`, the `substitute` function can then call it with the numeric value for `x` and get back a new numeric value. Concatenating the new numeric value in between opening and closing parentheses completes the substitution of the `x` expression.

The `insertpar` function is somewhat simpler. It takes an input string, a parameter string and a symbol. It's purpose is to place a parameter string, inside parentheses, in front of each occurrence of symbol in the input string.

Parameterized L-system Replacement

```
def replace(self, istring):

  Initialize the output string to the empty string
  Initialize the parameter string to the empty string
  Initialize the parameter value to None
  Initialize the parameter state to False

  for each character c in the input string

    # handle parameters
    if c is '(' then we're starting a parameter
      assign to the parameter string the empty string
      assign to the parameter state the value True
      continue to the next character

    else if c is ')' then we're ending a parameter
      assign to the parameter value the float cast of the string
      assign to the parameter state the value False
      continue to the next character

    else if the parameter state is True
      append to the parameter string the current character
      continue to the next character


    # handle regular characters
    if the parameter value is not None

      assign to a local variable (key) the expression '(x)'+c

      if key is in the dictionary self.rules
        assign to a variable (replacement) a random choice from self.rules[key]
        add to tstring the result of self.substitute( replacement, parval )
      else
        if c is in the dictionary self.rules
          assign to a variable (replacement) a random choice from self.rules[c]
          add to tstring the result of self.insertpar( replacement, parstring, c )
        else
          add to tstring the string '(' + parstring + ')' + c
      set parval to None

    else (no parameter, so just a standard replacement rule)
      if c is in self.rules
        add to tstring a randomly chosen replacement from self.rules[c]
      else
        add to tstring the value c

  return tstring
```

Parametric L-systems Parameter Substitution Function

```
# given: a sequence of parameterized symbols using expressions
#        of the variable x and a value for x
#
# substitute the value for x and evaluate the expressions
def substitute(self, sequence, value ):

    # an expression string and state variable
    expr = ''
    exprgrab = False

    # the output sequence
    outsequence = ''

    # for each character in the sequence
    for c in sequence:

        # if a parameter expression starts
        if c == '(':
            # set the state variable to True (grabbing the expression)
            exprgrab = True
            expr = ''
            continue

        # else if a parameter expression ends
        elif c == ')':
            # set the state variable to False (expression completed)
            exprgrab = False

            # create a function out of the expression
            lambdafunc = eval( 'lambda x: ' + expr )

            # execute the function and put the result in a (string)
            newpar = '(' + str( lambdafunc( value ) ) + ')'

            # add the new numeric parameter to the output sequence
            outsequence += newpar

        # else if the state variable is True (grabbing an expression)
        elif exprgrab:
            # add the character to the expression
            expr += c

        # else not grabbing an expression and not a parenthesis
        else:
            # add the character to the out sequence
            outsequence += c

    # return the output sequence
    return outsequence
```

The `insertpar` function loops over all of the characters in the input string. It copies all characters not equal to the symbol directly from the input to the output string. If it finds a character that matches the symbol, it first inserts the parameter string–in parentheses–into the output string and then copies the character.

Parametric L-systems Parameter Insertion Function

```
# given: a sequence, a parameter string, a symbol
#
# inserts the parameter, with parentheses, before each
# instance of the symbol in the sequence

def insertpar(self, sequence, parstring, symbol):

    # initialize a return string
    tstring = ''

    # for each character in the input string
    for c in sequence:
        # if the character is the symbol
        if c == symbol:
            # add the parameter string in parentheses
            tstring += '(' + parstring + ')'
        # add the character
        tstring += c
    # return the output string
    return tstring
```

10 3-D Turtle

Turtle graphics are not limited to two dimensions. The concept of drawing by orienting and moving a local coordinate frame applies to any number of dimensions. In two dimensions, the local coordinate frame is represented by the turtle's position (x, y) and its orientation on the x-y plane, θ. When we move to three dimensions, the concepts of turning and moving forward are the same, but there are more ways to turn than just left and right.

The 3D turtle provided as part of the last project is designed to emulate the standard 2D turtle. There are a few differences, and the 3D turtle does not support some of the new functionality in the Python 2.6 turtle package such as the pencolor and fillcolor functions. For the most part, however, a few simple modifications to the Interpreter class allows the existing turtle code to work without modification.

1. Import turtleTk3D at the top of the interpreter.py file.

   ```
   import turtleTk3D
   ```

2. Set up a global variable called turtle and assign it the value None

   ```
   turtle = None
   ```

3. Inside the `__init__` function of the Interpreter class, create a new turtle object and assign it to the turtle global variable

   ```
       global turtle
       turtle = turtleTk3D.Turtle3D()
   ```

   Now all of your remaining interpreter code will work as usual.

10.1 Rotations

In two dimensions, all rotations occur about the Z-axis, which points out of the screen. A left turn is a positive rotation about the Z-axis, and a right turn is a negative rotation about the Z-axis. The turtle is limited to these rotations because it cannot orient itself outside of the 2D plane. It's forward orientation can change, but the turtle's perception of up is always pointing out of the screen.

In three dimensions, rotations can occur about any of the three axes that define the turtle's local coordinate frame. In its default initial position, the turtle's X axis points right on the screen; the Y axis begins pointing up on the screen; the Z axis $(X \times Y)$ points out of the screen. As the turtle moves and rotates, its coordinate frame also moves and rotates. Forward will always be in the direction of the turtle's X axis, and up will always be the turtle's Z axis.

The three types of rotations are commonly called yaw (left and right), pitch (up and down), and roll (rolling left and right). It is possible to achieve any final orientation, relative to the current orientation, using some combination of roll, pitch, and yaw.

- Yaw: rotation to the left or right, which is rotation about the Z axis, the turtle's up direction.
  - Positive yaw (+) is left
  - Negative yaw (-) is right.
- Pitch: rotation down or up, which is rotation about the Y axis, the turtle's left direction.
  - Positive pitch (&) is down
  - Negative pitch (ˆ) is up.
- Roll: rolling left or right, which is rotation about the X axis, the turtle's forward direction.
  - Positive roll (\) is to the right
  - Negative roll (/) is to the left.

If you have no pitch or roll commands in a string, then your shape will stay on the plane of the screen and will be flat. If you include a pitch or roll, then your shape becomes 3D.

In the language of L-systems, we can use the characters +, -, &, ˆ, \, and / to represent positive and negative yaw, positive and negative pitch, and positive and negative roll.

For example, consider the following string. It generates a set of lines that form a cube.

```
F&F&&F&+F&F&&F&+F&F&&F&+F+&F^F+F+F+F+^F&
```

One problem with the above formulation is that the string overwrites several lines twice. This could cause unexpected results when using some of the NPR methods. Instead, we can make use of the push and pop feature to store and restore the turtle's state, avoiding overwriting any lines.

```
[F+[&F]F+[&F]F+[&F]F+&F^F+F+F+F+]
```

The above string writes each side only once and returns the turtle to its original position.

10.2 Updating the Interpreter

The interpreter does require some updates to make use of the third dimension and the new rotations. These modifications include the following.

- Modify the place function to incorporate a z coordinate (leave orient as a scalar value)
- Modify the goto function to allow a z coordinate
- Modify the various styles to work in 3D, which means you need Z coordinates and perturbations
- Modify the shape class to include roll and pitch (orientation is yaw).

  Call the place function first (yaw), then set the roll, and then the pitch. Note that the order of the rotations is important in 3D. Different orderings of the same set of rotations will generally result in different final orientations.

10.3 Representing the Turtle in 3D

In 2D, a complete description of the turtle's position and orientation is given by three values: (x, y, θ). When we add a third dimension, we have to store additional information to completely represent the turtle.

Three coordinates are sufficient to represent the turtle's location: (x, y, z). Three coordinates are also sufficient to represent the turtle's orientation. However, such a representation can be difficult to visualize or manipulate. Instead, we can represent the turtle's orientation using two 3-vectors that represent the turtle's forward and up directions.

$$(\vec{F}, \vec{U}) = ((F_x, F_y, F_z), (U_x, U_y, U_z)) \tag{10}$$

The forward vector $\vec{F}$ is the direction in which the turtle will travel given a forward command. The up vector $\vec{U}$ specifies the z-axis of the turtle's coordinate system, which is the axis that defines left and right turns. The turtle's default orientation on the screen is $O_d = ((1, 0, 0), (0, 0, 1))$, which means the turtle is facing to the right and the up vector comes out of the screen.

The 3D turtle `heading` function will return the current forward and up vectors. The turtle's `setheading` function expects either a single scalar or a pair of 3-valued vectors. Given a single scalar, it gives the turtle the default up vector $(0, 0, 1)$ and orients the turtle in the x-y plane according to the given angle.

The 3D turtle is actually quite forgiving about how the forward and up vectors are defined. Neither vector needs to be a unit vector, and the two vectors do not need to be orthogonal. The only requirement is that the two vectors not point in the same direction. The forward vector always defines the direction in which the turtle will move. The difference between the forward vector and the up vector will define the turtle's up orientation.

10.4 Subdivision Shapes

A subdivision shape, or surface, is defined by an initial collection of shapes and a rule for building a new set of shapes from the existing set.

Circle

- Begin with a set of four lines that form a square with corners on the unit circle.
- Subdivision rule: divide each line in half and move the midpoint to the unit circle boundary.
- After executing a series of subdivisions, draw lines between adjacent points

Bezier curve

- Given the point set (A, B, C, D)
- Calculate the following

$$A_1 = (A + B)\frac{1}{2}$$
$$B_1 = (B + C)\frac{1}{2}$$
$$C_1 = (C + D)\frac{1}{2}$$
$$A_2 = (A_1 + B_1)\frac{1}{2}$$
$$B_2 = (B_2 + C_2)\frac{1}{2}$$
$$A_3 = (A_2 + B_2)\frac{1}{2}$$

- Build two new point sets: (A, A_1, A_2, A_3) and (A_3, B_2, C_1, D).
- When drawing a line segment, set the forward turtle vector to point from the start to the end vertex.
- Set the up vector to something other than the forward vector

Sphere

Just as we can create a circle using a subdivision shape, we can also create a sphere using triangles and subdividing them into smaller and smaller pieces. It is possible to draw each triangle using a series of 3D goto statements. However, that does not enable the use of the NPR methods implemented using the forward function. Therefore, the question becomes how to draw an arbitrary triangle using a string representation.

- Begin with a set of eight triangles that form an octahedron with points on the unit sphere. Each triangle is defined by three 3-D points.

```
pointlist = [ ( (0, 0, 1), (1, 0, 0), (0, 1, 0) ),
              ( (1, 0, 0), (0, 0, -1), (0, 1, 0) ),
              ( (0, 0, -1), (-1, 0, 0), (0, 1, 0) ),
              ( (-1, 0, 0), (0, 0, 1), (0, 1, 0) ),
              ( (0, 0, 1), (0, -1, 0), (1, 0, 0) ),
              ( (1, 0, 0), (0, -1, 0), (0, 0, -1) ),
              ( (0, 0, -1), (0, -1, 0), (-1, 0, 0) ),
              ( (-1, 0, 0), (0, -1, 0), (0, 0, 1) ) ]
```

- The subdivision step divides each triangle into four new triangles. First, find the midpoint of each edge segment, then create four new triangles. If the original triangle vertices are A, B and C, the following defines the new set of triangles.

$AB = (A + B)\frac{1}{2}$
$BC = (B + C)\frac{1}{2}$
$CA = (C + A)\frac{1}{2}$

The new triangles are $T_1 = (A, AB, CA)$, $T_2 = (B, BC, AB)$, $T_3 = (C, CA, BC)$, and $T_4 = (AB, BC, CA)$.

The challenging part of drawing a triangle is handling it in a manner that lets us also use the NPR styles. The concept is straightforward: the three points on the triangle define a plane, so there exists a plane on which we can draw the triangle as three forwards and two left turns. We just need to calculate the plane, the forward distances, and the angles.

Triangles are necessarily convex. If we assume the points are defined in a counter-clockwise manner, it makes the calculations easier. First, calculate the length of each side– L_{AB}, L_{BC}, and L_{CA}.

The forward vector for the first turtle motion is the vector $\vec{AB}$. A reasonable left vector is $-\vec{CA}$. The cross-product of these two vectors tells us the proper up vector. In python, if `forward` holds the forward vector and `left` holds the left vector, the following expression generates an appropriate `up vector`.

```
up = ( forward[1]*left[2] - forward[2]*left[1],
      forward[2]*left[0] - forward[0]*left[2],
      forward[0]*left[1] - forward[1]*left[0] )
```

If we normalize the three line segment vectors, the following calculations produce the two outer angles.

$$\text{length of V} = L_V = \sqrt{V_x^2 + V_y^2 + V_z^2} \tag{11}$$

$$\text{normalized vector V} = \hat{V} = (V_x/L_V, V_y/L_V, V_z/L_V) \tag{12}$$

$$\text{cosine of first outer angle} = c_0 = \hat{AB} \cdot \hat{BC} \tag{13}$$

$$\text{cosine of second outer angle} = c_1 = -\hat{BC} \cdot \hat{CA} \tag{14}$$

$$\text{angle, in degrees} = \theta_i = \cos^{-1}(c_i)\frac{180}{\pi} \tag{15}$$

After placing the turtle at vertex A and orienting it to the forward and up vectors defined above, we can create the following parameterized string, inserting the calculated values for L_{AB}, L_{BC}, L_{CA}, θ_0, and θ_1.

$(L_{AB})F(\theta_0) + (L_{BC})F(\theta_1) + (L_{CA})F$

One way to generate this string is to use a formatted string (see section 4.5.2 of the textbook). The following expression substitutes the appropriate values into the fields defined by the `%.3f` format specifier, which indicates Python should format the number as a floating point value with three decimal places.

```
# generate the string to draw
s = '(%.3f)F(%.3f)+(%.3f)F(%.3f)+(%.3f)F' % (Lab, T0, Lbc, T1, Lca) }

# draw the string s
dev.drawString( s, self.distance*scale, self.angle )
```

10.5 Nudge

The 3D turtle also includes a function called nudge. The nudge method takes in a 3-element vector and adds it to the forward orientation vector, recalculating the turtle's orientation as necessary. The nudge method enables you to easily simulate the effect of gravity or wind on trees.

- Use the 3D turtle's nudge capability to simulate forces on your shapes (e.g. trees)
- Nudge moves the orientation of the turtle towards the nudge direction
- Can build it in as an optional step in your interpreter's forward function
  - Define gravity as either None or a 3-vector
  - If gravity is not None, nudge the turtle with the 3-vector
- Keep the gravity vector so all elements are between -1 and 1

11 What Have You Learned?

Highlights of the semester

- What can a computer do?
- A programming language: Python
- 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
- Programming as abstraction: making things general / using parameters
- 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

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.

11.1 How Can This Help You?

Being more comfortable with a computer is a skill you will always have. It will give you an edge over people who are uncomfortable, or who do not have a good mental model for how a computer works.

Art: for an interactive art exhibition, how would you describe the behavior of the installation as a computer program?

English/Linguistics: say you want to compare the average and degree of variation in sentence lengths of Hemmingway and Faulkner. How would you do it?

Government/Economics: say you have a set of data in a file and you want to read in the data, calculate averages, and print them out neatly. How would you do it?

Environmental Studies: if you want to combine information from multiple overlapping maps in a particular manner, you can write scripts (in Python) to do simple (or complex) calculations.

Biology: how would you calculate statistics on a set of data? (Did you know Excel has a programming language?)

Life: say you're playing D&D and you don't have a 12-sided die? How could you make an automatic character generator?

11.2 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>
```

©2011 Bruce A. Maxwell

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 );
  }

};
```