CS 151 Computational Thinking: Science Applications, Fall 2015

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

Course Description

This course is an introduction to computational thinking: how we can describe and solve problems using a computer. The Science Applications section will focus on reading, writing, managing, and analyzing data for scientific and social science applications. These applications will motivate how and why we would would want to write procedures, control the flow of information and processes, and organize information for easy access and manipulation. Through lectures, short homeworks, and weekly programming projects, you 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. While the projects are focused on scientific and data management applications, the computational thinking skills you learn in this course are applicable to any type of programming or program design you may undertake in the future.

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.

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. mathematical functions)

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 define the collection of mathematical operations

```
  Y = A + B * X + C * X * X
```

and call that collection the function `poly2(A, B, C, X)`. If we can create that function, then any time we need to compute the value of a 2nd-order polynomial we can use the function. While it may seem like we don't gain much, there is much less chance of making a mistake typing a function name than correctly typing the polynomial equation each time you need it. As functions get more complex, the gain becomes even more obvious.

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                   >>> x = 10.0                 >>> x = ’10’
>>> type(x)                  >>> type(x)                  >>> type(x)
<type ’int’>                 <type ’float’>               <type ’str’>


>>> x = 10L                  >>> x = complex(10, 10)      >>> x = True
>>> type(x)                  >>> type(x)                  >>> type(x)
<type ’long’>                <type ’complex’>             <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
euclid	function data	function
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

Global symbol table after the first append:

Symbol	Type	Value
squares	list	ref list1

Global symbol table after the second append:

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]

squares symbol table after 2nd assignment:

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

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 )
```